Poul Klausen

C# 9

GUI programs with WPF

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 9: GUI PROGRAMS WITH WPF

SOFTWARE DEVELOPMENT

C# 9: GUI programs with WPF: Software Development
1st edition

ISBN 978-87-403-3579-8

CONTENTS

FOREWORD

This book is the nine in a series of books on software development. The programming language is C#, and the language and its syntax and semantics fills obviously much, but the books have also largely focus on the process, and how to develop good and robust applications. This book is like the next book about development of programs with a graphical user interface, a topic that was treated in the book C# 2 and also in book C# 6. This book and the next one deal with a number of details that there was no room for in the previous books, and besides, this book deals with how to work with graphics and draw geometric shapes in the window. It is not so often that there is a need to draw directly in the window, but the examples are found and especially in connection with the development of custom components. The development of custom components also plays a significant role in the book.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in C#. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it, and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples, exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code for the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with a larger sample program, which focus primarily is on the process and an explanation of how the program is written. On the other hand appears the code only to a limited extent, if at all, and the reader should instead study the finished program code, perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

All sample programs are developed and tested on a Windows machine, and the development tool is Visual Studio. Therefore, you must have Visual Studio installed and Microsoft

provides a free version that is fully adequate. Visual Studio is an integrated development environment that provides all the tools needed to develop programs. I do not want to deal with the use of Visual Studio, which is quite straightforward, and it is a program that you quickly become familiar with.

Finally a little about what the books are not. It is not "a how to write" or for that matter a reference manuals to C#, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

The book C# 2 is an introduction to writing programs with a graphical user interface using WPF and the previous books has shown many examples of applications with a graphical user interface. This book is a continuation as WPF includes many details that are not included in C# 2, and important topics are

1. More on events
2. More on properties
3. More on commands
4. Custom controls
5. A WPF navigation application
6. A WPF application as code
7. Drawing shapes
8. Drag and drop
9. Animations

Most of these topics do not directly relate to each other but are topics that explain in detail how WPF works and partly describe concepts that are necessary to know in order to develop GUI programs with a complex user interface.

1.1 A WPF APPLICATION

When you create a new WPF project in Visual Studio, many folders and files are created. I have created a new WPF Application project and translated it:

```
C:\Temp\TestWPF
```

Then Visual Studio created 36 libraries (folders) and 35 files. It is many files and it is somewhat unclear - and fortunately also unnecessary - to get an overview of what it is all about. However, it may be good to have a small idea of what files are being created and what they are used for.

If you opens a command prompt, goes to the folder *C:\Temp* and executes the command

```
DIR /s TestWPF > list
```

you create a text file showing all the project files where I have removed some lines and here all lines that references the current directory and the parent directory:

```
Directory of C:\Temp\TestWPF
16-03-2020 10:13        <DIR>       TestWPF
16-03-2020 10:13             1.127 TestWPF.sln

 Directory of C:\Temp\TestWPF\.vs
16-03-2020 10:13        <DIR>       TestWPF

 Directory of C:\Temp\TestWPF\.vs\TestWPF
16-03-2020 10:14        <DIR>       v16

 Directory of C:\Temp\TestWPF\.vs\TestWPF\v16
16-03-2020 10:14        <DIR>       Server

 Directory of C:\Temp\TestWPF\.vs\TestWPF\v16\Server
16-03-2020 10:14        <DIR>       sqlite3

 Directory of C:\Temp\TestWPF\.vs\TestWPF\v16\Server\sqlite3
16-03-2020 10:14             0 db.lock
16-03-2020 10:14             4.096 storage.ide
16-03-2020 10:14            32.768 storage.ide-shm
16-03-2020 10:14           778.712 storage.ide-wal

 Directory of C:\Temp\TestWPF\TestWPF
16-03-2020 10:13               189 App.config
16-03-2020 10:13               368 App.xaml
16-03-2020 10:13               328 App.xaml.cs
16-03-2020 10:13        <DIR>      bin
16-03-2020 10:13               501 MainWindow.xaml
16-03-2020 10:13               629 MainWindow.xaml.cs
16-03-2020 10:13        <DIR>      obj
16-03-2020 10:13        <DIR>      Properties
16-03-2020 10:13             4.177 TestWPF.csproj

 Directory of C:\Temp\TestWPF\TestWPF\bin
16-03-2020 10:13        <DIR>      Debug

 Directory of C:\Temp\TestWPF\TestWPF\bin\Debug

 Directory of C:\Temp\TestWPF\TestWPF\obj
16-03-2020 10:13        <DIR>      Debug
```

```
 Directory of C:\Temp\TestWPF\TestWPF\obj\Debug
16-03-2020 10:13               2.332 App.g.i.cs
16-03-2020 10:13               7.195 DesignTimeResolveAssembly
                                     ReferencesInput.cache
16-03-2020 10:13               3.039 MainWindow.g.i.cs
16-03-2020 10:13        <DIR>        TempPE
16-03-2020 10:13              13.990 TestWPF.csprojAssemblyReference.
                                     cache
16-03-2020 10:13                 208 TestWPF_MarkupCompile.i.cache
16-03-2020 10:13                  53 TestWPF_MarkupCompile.i.lref

 Directory of C:\Temp\TestWPF\TestWPF\obj\Debug\TempPE

 Directory of C:\Temp\TestWPF\TestWPF\Properties
16-03-2020 10:13               2.420 AssemblyInfo.cs
16-03-2020 10:13               2.612 Resources.Designer.cs
16-03-2020 10:13               5.612 Resources.resx
16-03-2020 10:13               1.042 Settings.Designer.cs
16-03-2020 10:13                 201 Settings.settings
```

If you start in the project folder there is a file with the same name as the project:

```
TestWPF.sln
```

It is the project's solution file and is a file which alone has interest for Visual Studio. It is a text file which primary tells Visual Studio which projects this solution contains. In this case there is only one.

The project is called *TestWPF* and has a subdirectory with the same name:

```
TestWPF\TestWPF
```

and it contains the most important files viewed by the programmer, and in many contexts, in fact, the only files that the programmer needs to take an interest in. Here you will find the program's project file:

```
TextWPF.csproj
```

It is a fairly comprehensive file which contains all the settings regarding the project in question. It is also a file that is aimed solely for Visual Studio and a file that the programmer is not supposed to maintain, but it is a text file and the content is XML, so the file is both readable and modifiable. In particular, you should be aware that if a solution has multiple projects, there will be a folder under each project folder for each project and the project files will be in the respective folders.

If you look further in the project folder, you will also find the files for the main program window:

```
MainWindow.xaml
MainWindow.xaml.cs
```

The first is the XML file for the user interface and the second the C# code (code behind), and it is thus these two files that this whole book has essentially dealt with. There is also an application configuration file (see book C# 5):

```
App.config
```

It is also one of the files that you as a programmer possibly have needs to modify. The default version does not contain much besides a version number for the runtime system, but in general the file is used to define parameters that the program needs. It is an XML file, and it can be edited by anyone without having to translate the program, exactly what the idea is. The file could, for example contain information about databases or equivalent, and it is the purpose for an administrator to maintain the file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5"
    />
  </startup>
</configuration>
```

In the project folder you also finds the program's start file:

```
App.xaml
App.xaml.cs
```

and it consists in the same way as the main window of an XML file and a C# file. This file I have not looked at much until now, but it is a file that you sometimes must maintain. If you look at the C# file, it does not contain much:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Threading.Tasks;
using System.Windows;

namespace TestWPF
{
  public partial class App : Application
  {
  }
}
```

but it defines a class called *App* that inherits the class *Application*. If you look at the XML file, it does nor contain much, and the result is as follows:

```
<Application x:Class="TestWPF.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
             presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:TestWPF"
             StartupUri="MainWindow.xaml">
     <Application.Resources>

     </Application.Resources>
</Application>
```

A little short, it says that a class must be generated for a Windows program and that the start window is called *MainWindow*. From the programmer's point of view, the most important thing is that you can define resources that should be accessible to the entire program, and you can also specify a different startup window if needed for any reason. On the whole, this class is the place if you need something to happen before the program starts. It may not be that often, but the opportunity and the examples are there. It is especially the place to add resources such as styles that should be available for the entire application.

It might be hard to see what the *App* class has to do with a program, but here you have to think about the whole philosophy behind WPF. A UI class is defined by two files, one being a usual C# file, the other being XML. When the program is translated, these two files are used by the compiler to create the finished class file, which is why it is said that in WPF, classes are created on the fly as part of the build process. If you in this case look under the folder

```
TestWPF\TestWPH\obj\Debug
```

you can find the file

```
App.g.i.cs
```

From the name you can see that it is a C# file and it is actually the class file for the *App* class. If you open the file you will find, among other things, following:

```
namespace TestWPF {
    public partial class App : System.Windows.Application {
      [System.Diagnostics.DebuggerNonUserCodeAttribute()]
      [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuil
      dTasks",
        "4.0.0.0")]
      public void InitializeComponent() {

        #line 5 "..\..\App.xaml"
        this.StartupUri =
          new System.Uri("MainWindow.xaml", System.UriKind.Relative);

        #line default
        #line hidden
      }
```

```
      [System.STAThreadAttribute()]
      [System.Diagnostics.DebuggerNonUserCodeAttribute()]
      [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuil
      dTasks",
        "4.0.0.0")]
      public static void Main() {
        TestWPF.App app = new TestWPF.App();
        app.InitializeComponent();
        app.Run();
      }
    }
}
```

The whole thing is not so easy to interpret, but you can see, that a method *InitializeComponent()* has been created which opens the main window - or at least sets a reference to the window. You can also see that a *Main()* method has been created. It creates an instance of the *App* class and executes *InitializeComponent()*, and finally executes the method *Run()*, which is defined in the base class and starts it all.

If you look further at the *TestWPF\TestWPF\obj\Debug* folder, you can see that there are many files (and I even delete a few) and these are generally files that Visual Studio creates as part of the build process. Among other things you can find the file

```
MainWindow.g.i.cs
```

which is the finished C# code for the class *MainWindow*. If you add a component in the XML for *MainWindow* and opens the file, you will see the other part of the partial class *MainWindow*, and that the class has an instance variable for this component.

All files in the folder should be viewed as temporary files that are Visual Studio's own files.

There is also the folder

```
TestWPF\TestWPF\Properties
```

There are 5 files:

```
AssemblyInfo.cs
Resources.Designer.cs
Resources.resx
Settings.Designer.cs
Settings.settings
```

These are all text files and can thus be opened with for example *Notepad*. The files are intended for Visual Studio and contain general settings for how to build the finished assemblies and including settings for the resources. You can maintain these files manually, but that's not the idea, and Visual Studio has tools for that purpose.

Then there is the folder

```
TestWPF\TestWPF\bin\Debug
```

which contains the compiled program. Here, you should pay particular attention to two files (the others have to do with debug):

```
TestWPF.exe
TestWPF.exe.config
```

The first is the assembly for the compiled program, while the second is the config file and is actually a copy of *App.config*. Of course, the folder can contain other important files, including private assemblies. If you in Visual Studio build the project as a *Release* version the same files are created, but this time stored in the directory:

```
TestWPF\TestWPF\bin\Release
```

2 MORE ON EVENTS

An event is typically raised by a component such as a button when the user clicks on it with the mouse. Events may also occur in other contexts, for example when the window is displayed on the screen, when the size of the window is changed, etc., and although usually associated with the user's use of the keyboard or mouse, it may also be the system that raises an event to tell about the condition of a component or container.

When a component or container raises an event, nothing happens immediately, but the component that raises the event may have associated an event handler, which is a method that is executed when that event occurs. In this method you can then write the code that you want to execute as the result of the event in question.

An event handler is defined by a delegate who defines its signature and it varies, but as an example the event handler for a button will have the signature

```
private void cmdOk_Click(object sender, RoutedEventArgs e)
{
  ...
}
```

where the first parameter is a reference to the component, container or object that raised the event, while the second parameter is dependent on the event in question.

In WPF, routed events are used, and it is primarily a question that an event can be handled by several handlers, and from the programmer's point of view, the most important challenge is to learn in what order events reach the different elements of the user interface.

Consider the following user interface as an example:

```
<Window x:Class="EventProgram.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
  <DockPanel>
    <Grid>
      <Button Name="cmd" Content="Knap" Width="100" Height="25"/>
    </Grid>
  </DockPanel>
</Window>
```

where there is a *Window* containing a *DockPanel* containing a *Grid* containing a *Button*. That is the button sits in a hierarchy of containers. If the button raises an event because users click on it with the mouse, that event can be captured by all containers in the button hierarchy. In other words, it means that the mouse click is not only observed by the button, but also by its containers - which are actually also clicked on. The advantage of this is that you can control at what level the event in question must be handled. Consider the following extension of the above XML:

```
<Window x:Class="EventProgram.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300" Button.Click="Window_
    Click">
  <DockPanel Button.Click="DockPanel_Click">
    <Grid Button.Click="Grid_Click">
      <Button Name="cmd" Content="Knap" Width="100" Height="25"/>
    </Grid>
  </DockPanel>
</Window>
```

where the difference is that a *Click* event handler is associated with the *Window, DockPanel* and *Grid* objects. None of these containers can raise a *Click* event, but since a *Button* can, the container can do so in the form of an attached event. Note the syntax *Button.Click* for an attached event. Thus, a container can capture an event sent by one of its child components. When the user clicks the button, it has no event handler for a *Click* event, but it sits in a *Grid* and it has, and the *Grid* component sits in a *DockPanel*, which also has a *Click* handler and the same goes for the *Window* object. That is that a routed event is transmitted through the hierarchy of nested components and anyone can choose to process that event.

You may not be interested in that. The event handler parameter has the type *RoutedEventArgs*, and this type has a property *Handled*, which can be set to true to indicate that the event should not be passed further up the hierarchy, for example

```
private void DockPanel_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("DockPanel");
  e.Handled = true;
}
```

It means that the *Window* object do not receive this event.

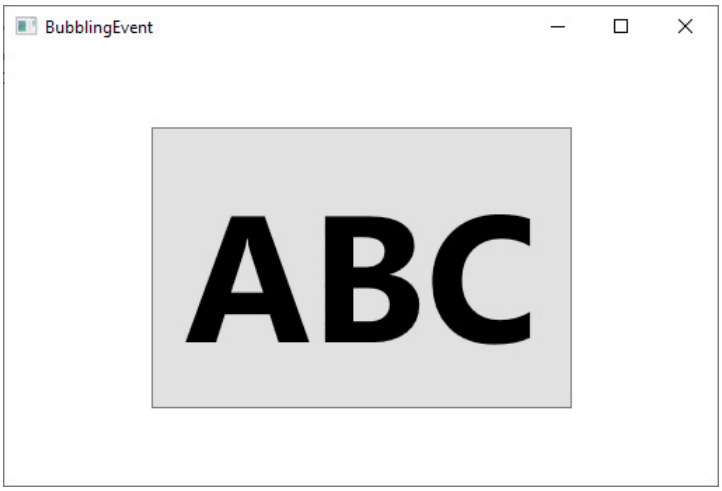
There are three types of events:

1. *Direct events*, which can be captured by the component that causes the event. A direct event is not passed on in the hierarchy. A good example of a direct event is a *MouseLeave* event.
2. *Bubbling events*, there are events that can be processed by the component that causes the event, which then bubble up through the hierarchy of containers, so that all containers up to the *Window* object can capture that event what is as illustrated above. A good example of a bubbling event is a *MouseDown* event.
3. *Tunneling events*, which works similar to bubbling events, but such that it is the outermost container that first gets the opportunity to catch the event. An example is a *PreviewMouseDown* event.

In the .NET framework, tunneling events will always start with the word *Preview*, and they will always occur in pairs along with a bubbling event, for example *PreviewKeyDown* and *KeyDown*. Note that the event handler argument is the same object for pairs of tunneling and bubbling events. It means that if you set the property *Handled* for a tunneling to *true*, the chain is disconnected there.

The following example shows a bubbling event in the form of a *Click* event for a button. The example opens a window with a *Grid* which has a *ScrollViewer* with a *Button*. Clicking on the button raises an event, but the same event is catch by the three containers in the button's visual tree.

The window is as follows:



and the most important is the XML code:

```
<Window x:Class="BubblingEvent.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:BubblingEvent"
        mc:Ignorable="d"
        Title="BubblingEvent" Height="350" Width="525"
          Button.Click="MainWindow_Click">
  <Grid Button.Click="Grid_Click">
    <ScrollViewer HorizontalScrollBarVisibility="Auto"
      VerticalScrollBarVisibility="Auto" Button.Click="ScrollViewer_
      Click">
    <Button Content="ABC" Width="300" Height="200" FontSize="128"
      FontWeight="Bold" HorizontalContentAlignment="Center"
      Click="Button_Click" />
    </ScrollViewer>
  </Grid>
</Window>
```

The program code consists of only four event handlers, each of which opens a message box. First, note the button that raises a *Click* event and there is nothing new in it. If you look at the *ScrollViewer* component, it also captures the button's *Click* event. It can, as it bubbles up the hierarchy. A *ScrollViewer* has no *Click* event, but since it contains a *Button* object, it can refer to the *Button* event. Here you should especially notice the syntax where you write the name of the class *Button* followed by the name of the event:

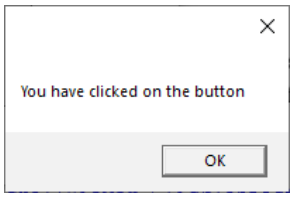
```
Button.Click="ScrollViewer_Click"
```

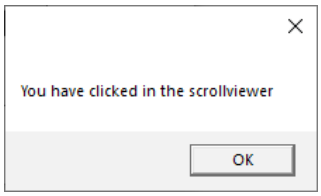
In this context, you are talking about an attached event, similar to the fact that it attach a *Button* object's *Click* event within a *ScrollViewer*.

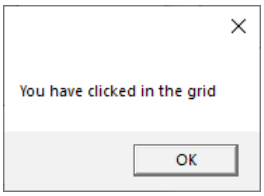
In this case, the same event is also captured by the *Grid* component (as an attached event):

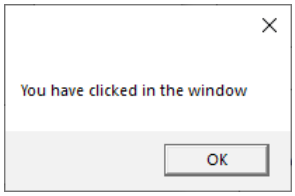
```
<Grid Button.Click="Grid_Click">
```

and correspondingly of the *Window* object. The result is that if you execute the program and click the button you will receive the following message boxes in the order shown:









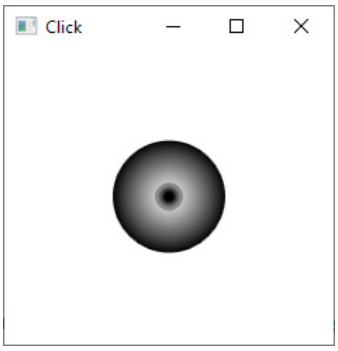
Especially note that you can stop an event tunneling or bubbling by setting the event argument's *Handled* property to *true*. If, in this case, you set *true* in the event handler for the *ScrollViewer*:

```
private void ScrollViewer_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("You have clicked in the scrollviewer");
  e.Handled = true;
}
```

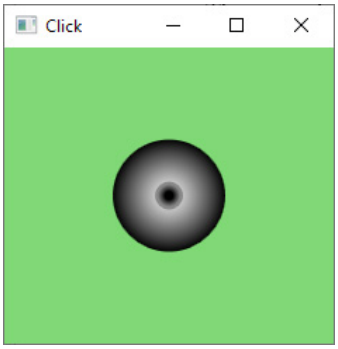
then neither the *Grid* container nor the *Window* object will catch the *Button.Click* event.

2.1 A USER CONTROL

It is possible to expand with custom routed events, and the following example will show how to do it. The example opens a window showing a circle:



Clicking on the circle will color the window background with the random color:



The task can be solved in many other and simpler ways than is the case here, but the circle is a user control, which when clicked raises a routed event, which is then captured by the component's container, which is a *Grid*.

To the project is added a *User Control* (see a later chapter on user controls), which is a component that, like a window, has an XML file and a code behind C# file. The XML file is as follows:

```
<Grid MouseDown="Grid_MouseDown">
  <Ellipse >
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Color="Black" Offset="1"/>
        <GradientStop Color="White" Offset="0"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
  <Ellipse Width="20" Height="20" >
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Color="Black" Offset="0.3"/>
        <GradientStop Color="Gray" Offset="0.8"/>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</Grid>
```

and defines two *Ellipse* controls in a *Grid*. An *Ellipse* is dealt with in a later chapter of the book, but it is a component that draws an ellipse, and in this case it is filled in with a *GradientBrush* which mixes two colors. The result is two ellipses within each other. The *Grid* component is associated with a *MouseDown* event handler. It should raise a routed event, and that means that if a user clicks on the ellipse, a routed event is raised.

The component is called *ClickControl*, and the first step is to register the new event in the class - here the *ClickControl* class, which owns the event:

```
public partial class ClickControl : UserControl
{
  public static readonly RoutedEvent ClickedEvent =
    EventManager.RegisterRoutedEvent("Clicked", RoutingStrategy.Bubble,
    typeof(RoutedEventHandler), typeof(ClickControl));

  public event RoutedEventHandler Clicked
  {
    add
    {
      AddHandler(ClickedEvent, value);
    }
```

```
    remove
    {
      RemoveHandler(ClickedEvent, value);
    }
  }

  public ClickControl()
  {
    InitializeComponent();
  }

  private void Grid_MouseDown(object sender, MouseButtonEventArgs e)
  {
    RaiseEvent(new RoutedEventArgs(ClickedEvent, this));
  }
}
```

Routed events are registered using a static method in the class *EventManager*, where you register the event under a name - here *Clicked* - the event type (its routing strategy), the type of the handler (*RoutedEventHandler* is a delegate in the WPF framework), and the type of the class that can raise the event.

The event must be defined in the class that happens with a so-called event wrapper:

```
public event RoutedEventHandler Clicked
{
  add
  {
    AddHandler(ClickedEvent, value);
  }

  remove
  {
    RemoveHandler(ClieckedEvent, value);
  }
}
```

Back there is the event handler for the user's click on the ellipse, which raises an event of the type *Clicked*. Here's you should note how to raise an event with the method *RaiseEvent()*.

The window's XML does not contain much:

```
<Window x:Class="ClickProgram.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:ClickProgram"
        mc:Ignorable="d"
        Title="Click" Height="250" Width="250">
  <Grid Name="grid" local:ClickControl.Clicked="Grid_Clicked" >
    <local:ClickControl Width="80" Height="80"/>
  </Grid>
</Window>
```

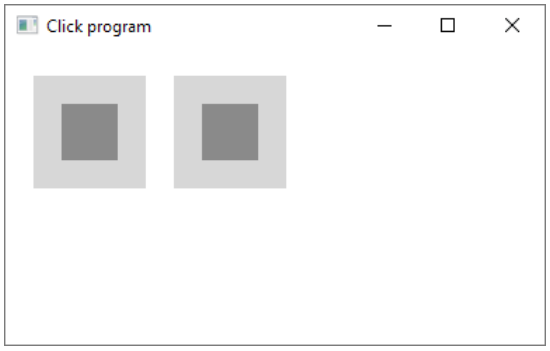
First, note that a namespace is defined for the program namespace:

```
xmlns:local="clr-namespace:ClickProgram"
```

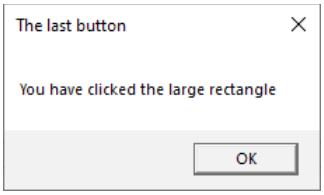
Next, note that the *Grid* of the window contains a *ClickControl* component and that you can use user controls in XML in the same way as other controls. Finally, note that the *Grid* container captures *Clicked* events from the *ClickControl* component, and notices the syntax how to specify it with an attached property. In the C# code there is a corresponding event handler which sets the background color to a random value, but it does not contain anything new and does not appear here.

EXERCISE 1: TWO CLICKS

Create a new WPF application project which you can call *ClickProgram*, The program must open the following window:



Here the two squares are a user control. If the user click on the light gray area the control must raise an event and if the user click on the dark gray area the control must raise another event. If the program is running and the user clicks on one of the four areas the program must open a message box, for example:

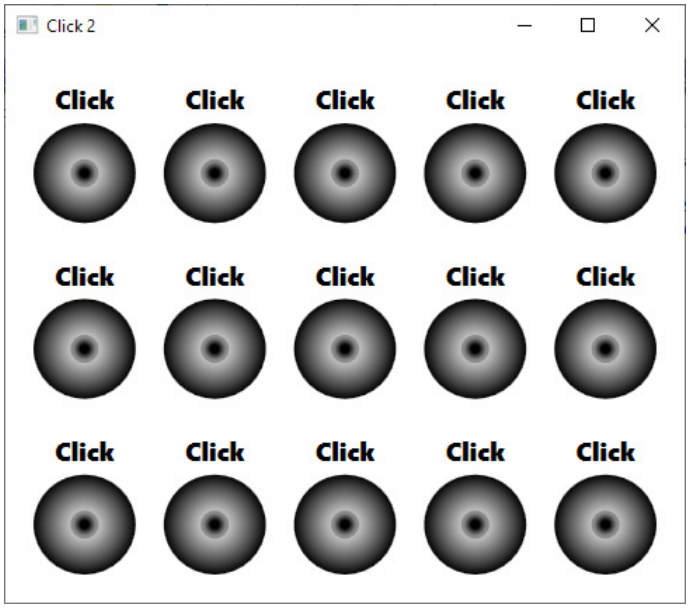


To write the program add a *User Control* to your project. You can call the control for *TwoClickControl* and it should be written in exactly the same way as the above example, when you in the XML use a *Rectangle* instead of an *Ellipse* and when you must register two *RoutedEvents* which you can call respectively *LargeClickedEvent* and *SmallClickedEvent*. To raise these events when the user click the mouse you can in XML add *MouseDown* event handlers to the *Rectangle* components.

When you write the XML for the *MainWindow* you can for example use a *StackPanel* and add two *TwoClickControl* components. You must note to use these components, it happens in much the same way as other WPF components and you can add event handlers for the two new events as usual.

EXERCISE 2: CLICKTWOPROGRAM

Write a WPF application which you can call *ClickTwoProgram*. The program must open the following window, where the window shows 15 controls in a *UniformGrid*:



A control must be a user control containing a label and a *ClickControl* (se the example above). When a user clicks the outer circle for a control the background must in the same way as in the example show a random color, and if the user clicks inner circle the text (Click) must get a random color. To write the program you should follow the below guidelines.

1. Create a new WPF application project which you can call *ClickTwoProgram*. Add the user control *ClickControl* from the above example. Remember to change the namespace.
2. Modify the user control *ClickControl* such the control raises an event, when the user clicks the outer circle and another event when the user clicks the inner circle.
3. Add a user control which you can call *TextUserControl* when the control in a *Grid* should contain a *Label* with the text *Click* (bold and a size on 18) and a *ClickControl*. The new control must add event handlers for the two events from the *ClickControl* and selects a random color for respectively the component's background and for the text color for the label.
4. Define a *UniformGrid* with 3 rows and 5 columns for *MainWindow*. Add 15 *TextUserControl* components the window when you should do that on code behind.

If you run the program and clicks on the circles the result could be:

Try to resize the window to see what happens. You can also try to change the number of rows and columns in the *UniformGrid*.

3 DEPENDENCY PROPERTIES

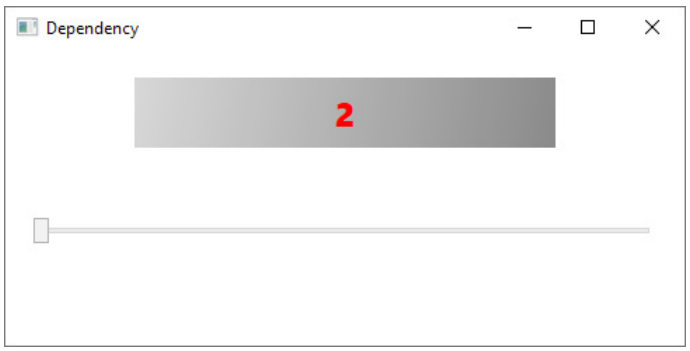
A very central concept in WPF is *dependency properties*, which in many ways actually act as a conventional property. The various WPF components have a large number of properties that can be used either in XML or in C# code, and most of them are dependency properties. For example are properties like *Width* and *Height* dependency properties, and as a programmer you are rarely aware of whether a property is a conventional CLR property or a dependency property.

A dependency property can be defined as follows:

1. A class that defines a dependency property must inherit, directly or indirectly, the class *DependencyObject*.
2. A dependency property is in the defining class defined as a *public*, *static readonly* object of the type *DependencyProperty*. If you want to define a property called *MyName*, the object is called *MyNameProperty*.
3. The dependency object in question must be registered with a static method *DependencyProperty.Register()*, typically executed inline or in a static constructor.
4. A usual CLR property is defined which is used to read or change the value of the dependency object.

The purpose of the dependency properties is primarily that it should be possible to refer to an object's properties in XML, as well as to support data binding. This also means that almost all properties for WPF components are dependency properties.

As an example to show how to write a dependency property and how it is different from a regular CLR property the program *DependencyProgram* opens the following window that has a label and a slider:



The *Label* component shows a prime number and if you click on the component you get the next prime number. You can also change the value of the prime by moving the slider:

The rectangle with the prime is not a usual *Label* component, but a user control with a *Label*, and I have called the control for *PrimeControl*. The XML code is simple

```
<Grid>
  <Label x:Name="display" Height="50" Width="300" Content="2"
    VerticalContentAlignment="Center" HorizontalContentAlignment="Center"
    FontSize="24" FontWeight="Black" Foreground="Red" >
    <Label.Background>
      <LinearGradientBrush>
        <GradientStop Color="LightGray" Offset="0"/>
        <GradientStop Color="Gray" Offset="1"/>
      </LinearGradientBrush>
    </Label.Background>
  </Label>
</Grid>
```

and the only thing to note is that a *LinearGradientBrush* is defined for the background. The meaning of the component is that it must have a dependency property for the prime value. It is defined in the code as follows:

```
public partial class PrimeControl : UserControl
{
  public static readonly DependencyProperty PrimeProperty =
    DependencyProperty.Register("Prime", typeof(ulong),
    typeof(PrimeControl),
    new UIPropertyMetadata((ulong)2, new PropertyChangedCallback(PrimeC
    hanged)),
    new ValidateValueCallback(ValidatePrime));
```

```
  public PrimeControl()
  {
    InitializeComponent();
  }

  public ulong Prime
  {
    get { return (ulong)GetValue(PrimeProperty); }
    set { SetValue(PrimeProperty, value); }
  }

  public static bool ValidatePrime(object value)
  {
    try
    {
      ulong number = (ulong)value;
      return number >= 2 && Primes.IsPrime(number);
    }
    catch
    {
      return false;
    }
  }

  private static void PrimeChanged(DependencyObject obj,
    DependencyPropertyChangedEventArgs args)
  {
    ((PrimeControl)obj).display.Content = "" + args.NewValue;
  }
}
```

The class has a property called *Prime*, and it is defined as a *public static readonly* property of the type *DependencyProperty* and named *PrimeProperty*:

```
public static readonly DependencyProperty PrimeProperty
```

Here you should especially notice the name, which is the name *Prime* (the name it should be known as) followed by the word *Property*. A dependency property must be registered, and in this context you must state:

1. its name that here is *Prime*
2. its type that here is *ulong*
3. its class from which it is defined, here it is *PrimeControl*

4. its metadata, specifying a default value (here 2), and a delegate to a callback method, which is executed when the value changes
5. a delegate to a validation method that defines which values are legal - if you assign an invalid value to a dependency property, an exception is raised

The class *PrimeControl* inherits *UserControl* and this class inherits indirectly *DependencyObject* there is a prerequisite for the class *PrimeControl* can have dependency properties.

Once the property in question is registered, a wrapper must be written so that it can be used with the same syntax as a standard CLR property. Here you must first note the name and how to refer to the static property with respectively *GetValue()* and *SetValue()*. This wrapper should always be like a thin wrapper that does nothing but reference the value and must not contain any program logic as the runtime bypasses it, and contains for example the *set* part program code, it will not be executed at all in many contexts.

Then there is the validation method, which here must validate that the value has the correct type, that is can be parsed to, in this case an *ulong*, which is a prime number. In particular, note that the validation method for prime numbers is not part of this class, but is a static method in a class *Primes*, which will be explained shortly.

Finally, there is the method *PrimeChanged()* that ties it all together. It is executed when the value is changed and updates the user interface to show the value of the prime, that is the value of the dependency property *Prime*.

Then there is the main window:

```
<Window x:Class="DependencyProgram.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:DependencyProgram"
        mc:Ignorable="d"
        Title="Dependency" Height="250" Width="500">
  <StackPanel Margin="20">
    <local:PrimeControl x:Name="primeCtrl" MouseDown="primeClick"/>
    <Slider Minimum="2" Maximum="10000000000000"
    LargeChange="100000000000"
      Margin="0,50,0,0" ValueChanged="Slider_ValueChanged"/>
  </StackPanel>
</Window>
```

Here you must note:

1. A namespace *local* is defined for the project namespace.
2. The window has a *PrimeControl* component with a *MouseDown* event handler.
3. The window has a *Slider* with an associated event handler.

Before I look at the C# code, I want to mention the class of *Primes*, which represents the sequence consisting of the prime numbers:

```
public class Primes : INotifyPropertyChanged
{
  public event PropertyChangedEventHandler PropertyChanged;
  private ulong _number = 2;

  public ulong Prime
  {
    get { return _number; }
    set
    {
       number = Next(value);
      OnPropertyChanged("Prime");
    }
  }

  public void Next()
  {
    if (_number == 2) _number = 3; else _number = Next(_number + 2);
    OnPropertyChanged("Prime");
  }

  protected void OnPropertyChanged(string name)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(name));
  }

  public static ulong Next(ulong number)
  {
    if (number <= 2) return 2;
    if (number % 2 == 0) ++number;
    for (; !IsPrime(number); number += 2) ;
    return number;
  }
```

```
  public static bool IsPrime(ulong n)
  {
    if (n == 2 || n == 3 || n == 5 || n == 7) return true;
    if (n < 11 || n % 2 == 0) return false;
    for (ulong k = 3, m = (ulong)Math.Sqrt(n) + 1; k <= m; k += 2)
      if (n % k == 0) return false;
    return true;
  }
}
```

The class is simple enough, but it is prepared for data binding by implementing the interface *INotifyPropertyChanged*, which simply defines a single method *OnPropertyChanged()*. The class has an event *PropertyChanged*, and the method raises this event if the sequence has changed the value (which indicates a new prime number) and if there is an observer for that event. In this way an observer get a message every time the sequence is changed. It can happen in two situations. Either by executing the method *Next()*, which steps the sequence one element, or by assigning it a value.

Finally, there is the C# code for the main window:

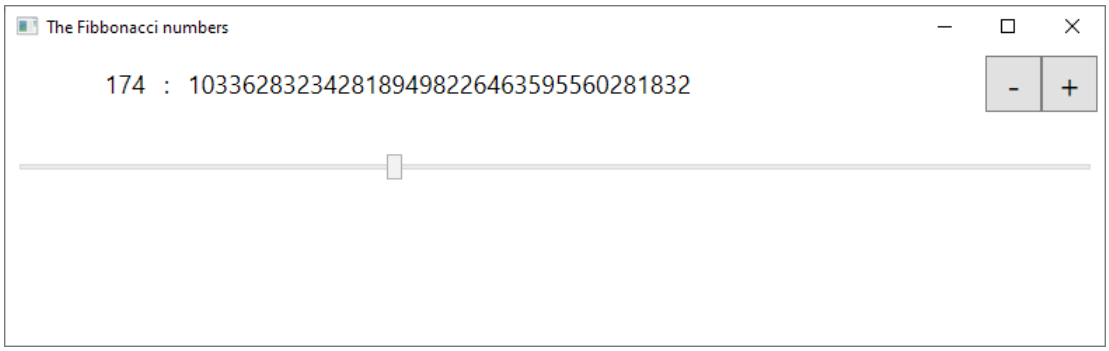
```
public partial class MainWindow : Window
{
  private Primes primes = new Primes();

  public MainWindow()
  {
    InitializeComponent();
    Binding bind = new Binding("Prime");
    bind.Source = primes;
    primeCtrl.SetBinding(PrimeControl.PrimeProperty, bind);
  }

  private void primeHandler(object sender, MouseButtonEventArgs e)
  {
    primes.Next();
  }
  private void Slider_ValueChanged(object sender,
    RoutedPropertyChangedEventArgs<double> e)
  {
    primes.Prime = (ulong)e.NewValue;
  }
}
```

First, note that the class has an object of type *Primes*. In the constructor, a *Binding* object is created for a property named *Prime*. The binding object is associated with a data source, which is *Primes*, and the binding object is bound to this source through its *Prime* property. Finally, the binding object is used to bind the data source to the *PrimeControl* component's dependency property, and it is here that it is important that it is a dependency property.

Note the two event handlers, both of which change the state of the object's state, it indicates a different prime number. A notification is then sent, which is captured by the binding and results in the user interface – the *PrimeControl* component - is being automatically updated.

EXERCISE 3: THE FIBONACCI SEQUENCE

In this exercise you must write a program like the above example that opens a window with a user control, a control that implements two dependency properties. When you run the program it should open the following window:



The program shows the Fibonacci numbers as a number sequence where the numbers are indexed from 0 and onwards. The buttons are used to step the sequence a step back and forth, and the slider is used to scroll through the Fibonacci numbers with index from 0 to 500. To solve the task, I would recommend that you follow the following guidelines.

1) Create a new WPF project as you can call *FiboProgram*.

2) A number sequence is a sequence of numbers (in this exercise integers) indexed by an *int* from 0 onwards, and an example is the Fibonacci numbers:

0, 1, 1, 2, 3, 5. 8, 13, 21, 34, 55, ...

For this program you must add a class representing a number sequence:

```
using System.Numerics;
using System.ComponentModel;

namespace FiboProgram
{
  public abstract class Sequence : INotifyPropertyChanged
  {
    public event PropertyChangedEventHandler PropertyChanged;
    protected BigInteger value = BigInteger.Zero;
    protected int index = 0;

  public Sequence(BigInteger value)
  {
    this.value = value;
  }

  public int Index
  {
    get { return index; }
    set
    {
      while (value < index) _Prev();
      while (value > index) _Next();
      OnPropertyChanged("Index");
      OnPropertyChanged("Value");
    }
  }

  public BigInteger Value
  {
    get { return value; }
    set
    {
      while (this.value < value) _Prev();
      while (this.value > value) _Next();
      OnPropertyChanged("Index");
      OnPropertyChanged("Value");
    }
  }

  public void Next()
  {
```

```
    _Next();
    OnPropertyChanged("Index");
    OnPropertyChanged("Value");
  }

  public void Prev()
  {
    _Prev();
    OnPropertyChanged("Index");
    OnPropertyChanged("Value");
  }

  protected void OnPropertyChanged(string name)
  {
    if (PropertyChanged != null)
      PropertyChanged(this, new PropertyChangedEventArgs(name));
  }

  protected abstract void _Next();

  protected abstract void _Prev();
  }
}
```

A sequence is represented by an index and a value which is a *BigInteger*. The class has two variables (properties) for the state and is defined *INotifyPropertyChanged* and it means that if one of the two properties are changed the class fires a *PropertyChanged* event indicating that a particular attribute has changed. It happens in the *set* part of the two properties as well in the two methods *Prev()* and *Next()*. The class is abstract as the two methods to step the sequence back and forth are defined abstract and must be implemented in the classes for the concrete sequences.

When you have the above abstract class you must write a class representing the *Fibonacci numbers*:

```
namespace FiboProgram
{
  public class Fibonacci : Sequence
  {
    public Fibonacci() : base(BigInteger.Zero) {}
    public static bool IsLegal(BigInteger number) { ... }
    protected override void _Next() { ... }
    protected override void _Prev() { ... }
  }
}
```

You should note that the class has a static method to validate where a *BigInteger* is a legal *Fibonacci* number.

3) If you look at the above window, the first row showing the index number, the value of the sequence and the two buttons must be a user control. Add a user control called *SequenceControl* to the project with a design as shown above. Note you only have to write the XML for the user control.

4) Design the *MainWindow* when it should only contain a *SequenceControl* and a *Slider*:

```
<Grid Margin="5">
  <Grid.RowDefinitions>
    <RowDefinition Height="40"/>
    <RowDefinition/>
  </Grid.RowDefinitions>
  <local:SequenceControl x:Name="seq" />
  <Slider Grid.Row="1" Minimum="0" Maximum="500" Margin="0,30,0,0" />
</Grid>
```

5) The user control must have two dependency properties:

```
public static readonly DependencyProperty IndexProperty =
  DependencyProperty.Register("Index", typeof(int),
  typeof(SequenceControl),
  new UIPropertyMetadata(0, new PropertyChangedCallback(IndexChang
  ed)),
  new ValidateValueCallback(ValidateIndex));
public static readonly DependencyProperty ValueProperty =
  DependencyProperty.Register("Value", typeof(BigInteger),
  typeof(SequenceControl),
  new UIPropertyMetadata(BigInteger.Zero,
  new PropertyChangedCallback(ValueChanged)),
  new ValidateValueCallback(ValidateValue));
```

You must implements these properties, which are methods that validate and change the properties as well as the wrapper methods. The implementation of the validate method for the property *Value* is a problem, and you can let it always return *true*.

Then there are the buttons which must step the sequence back and forth, and here the problem is that the class does not know the sequence and then cannot change the state of the sequence. The event handlers for the buttons should raise routed events

```
public static readonly RoutedEvent PrevClickedEvent =
  EventManager.RegisterRoutedEvent("PrevClicked", RoutingStrategy.
  Bubble,
  typeof(RoutedEventHandler), typeof(SequenceControl));
public static readonly RoutedEvent NextClickedEvent =
  EventManager.RegisterRoutedEvent("NextClicked", RoutingStrategy.
  Bubble,
    typeof(RoutedEventHandler), typeof(SequenceControl));
```

Look at chapter 2 for how to create routed events and how to raise a routed event.

Note the step is the core step in solving the exercise, and if all are implemented the custom control should be finish.

6) As the nest step you must write the code behind for the main window:

1. You must bind the two properties, see above for how to do that
2. You must assign event handlers for the two routed events raised by the user control
3. You must assign an event handler to the slider

The the program should works and let you step back and forth in the sequence,

7) The solution have some problems. One is that the user control when evaluating the property *Value* always return *true*. To solve this problem you can add a static method

```
public static bool IsLegal(BigInteger number)
```

to the class *Fibonacci* which evaluates where the parameter is a legal Fibonacci number. You could then add the following definition

```
public delegate bool SquenceValidator(BigInteger number);
```

to the file with the code behind for the user control and extend the control with the following property:

```
public static SquenceValidator Validator { get; set; }
```

You can then use this property in the method *ValidateValue()*.

Another problem is, that the user control and the slider are not synchronized. If you step the sequence using the buttons the slider is not updated. You can do that using data binding.

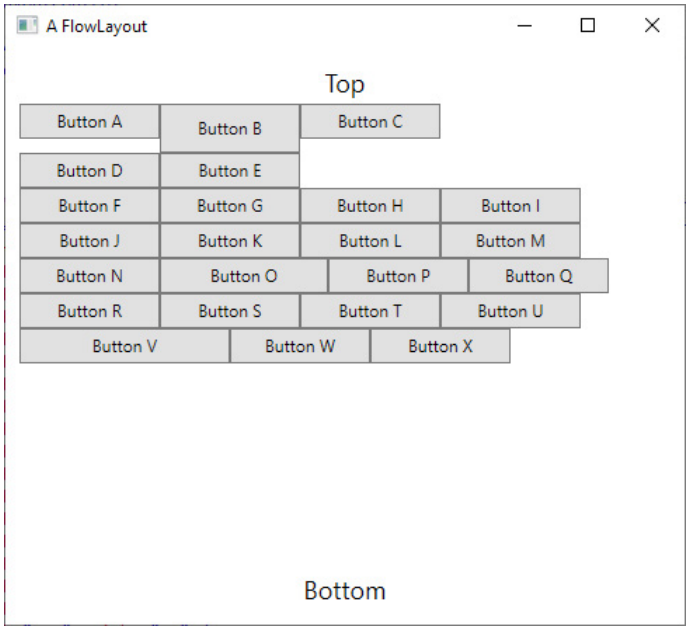
8) Add a class *Power2* to the project which must implements the sequence of powers of 2, which is a class that inherits the class *Sequence*. Test the program using this class instead of the class *Fibonacci*. Only one statement in the *MainWindow* class should be changed.

3.1 ATTACHED PROPERTIES

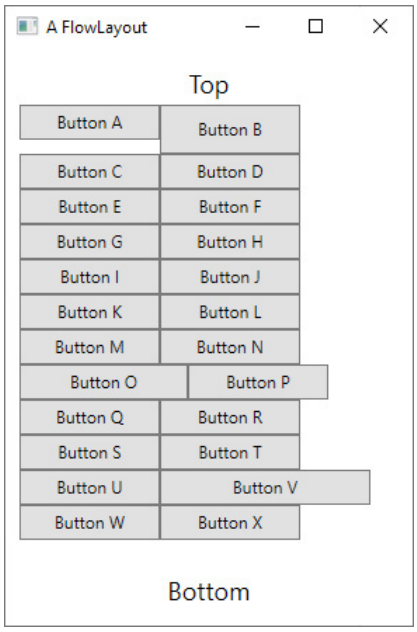
An attached property is a dependency property, and it is defined in the same way as a dependency property, but registered with the method *RegisterAttached()*. Attached properties are defined for a container and used to allow the container's components (objects) to access the container's properties. You know these properties from for example a *Grid* which has properties as *Grid.Row*, *Grid.Column* and so on. The following example creates a layout control to show how you define an attached property and how they works.

As mentioned, a dependency property is basically a property that can be assigned a value directly in XML or by means of data binding, in XML, but also with a style or an animation. The name dependency property is because its value depends on several elements in the user interface. As mentioned most properties of UI elements are actually dependency properties and including properties such as *Text* (*TextBlock*) and *Content* (*Button*). You can as shown above also define dependency properties yourself, and this example defines a layout panel named called *FlowPanel*, which in many ways corresponds to a *WrapPanel* and has two dependency properties and two attached properties. In addition to dependency properties and attached properties, the example also shows how to define your own layout panel.

If you run the program, you get the following window:



and then a window with 24 buttons of different sizes. If you now resize the window, the buttons will be distributed differently and the result could be the following:



The panel works in that it places the components in a number of rows starting from the upper left corner. In each row, as with a *WrapPanel*, there is the number of components that can be accommodated, but with dependency properties you can specify both a minimum and a maximum number of components. The components are adjusted within the individual row to the top edge, and finally, with attached properties, you can specify that there should be a line break either before or after a particular component.

The layout panel itself is created as a custom control. The result is a class (called *FlowPanel*) that inherits *Control*. In this I have changed the base class so that the class inherits the *Panel* class instead:

```
public class FlowPanel : Panel
{
```

The panel defines two dependency properties, which are called:

- *ElementMin*
- *ElementMax*

where the first specifies the minimum number of elements to be on a line (before the line wraps), while the second specifies the maximum number of elements. For both properties, if they have the default value (value 0), they are ignored. The panel also defines two attached properties:

- *BreakBefore*
- *BreakAfter*

which inserts a line break respectively before and after an element. When the panel shows the elements, the two dependency properties override the two attached properties so that the smallest and largest number of elements cannot be overridden by a *BreakAfter* or *BreakBefore*.

A dependency property always refers to the class (type) in which it is defined and cannot be used in another class, but the dependency properties follow the normal inheritance rules. As shown in the previous example, a dependency property is defined as a static object of the type *DependencyProperty* in the class to which the property relates. In this case, it is the dependency properties of the class (type) *FlowPanel*. The name of a dependency property is the name of the property followed by the word *Property*, for example *ElementMinProperty*. A dependency property must be registered before it can be used, and so this registration must be done in the declaration (as below) or in a static constructor. When registering a dependency property, you specify as shown above four values:

1. Its name, which is a string.
2. Its data type (all types are legal).
3. The type for which this property is defined.

4. Metadata, which is additional data that can be assigned. You can basically specify a default value or a callback method. In this case, callback methods are executed when the property changes value. In this case, the two dependency properties have a callback method which ensures that the minimum value cannot be greater than the maximum value. Note that it is not necessary to specify metadata, but only the last parameter is allowed to be null.

Similar to all that, the two dependency properties can be defined as follows:

```
public static readonly DependencyProperty ElementMinProperty =
  DependencyProperty.Register("ElementMin", typeof(int),
  typeof(FlowPanel),
  new PropertyMetadata(ElementMinChanged));
public static readonly DependencyProperty ElementMaxProperty =
  DependencyProperty.Register("ElementMax", typeof(int),
  typeof(FlowPanel),
  new PropertyMetadata(ElementMaxChanged));
```

Compared to the previous example, validation methods are not defined this time - it is not necessary, but the callback methods that are performed when the value is changed validates the values and ensure that the two properties do not get illegal values. Below is the callback method for *ElementMin*, which ensures that the value cannot be negative or greater than *ElementMax*:

```
private static void ElementMinChanged(DependencyObject obj,
  DependencyPropertyChangedEventArgs e)
{
  int min = (int)obj.GetValue(ElementMinProperty);
  int max = (int)obj.GetValue(ElementMaxProperty);
  if (min < 0) obj.SetValue(ElementMinProperty, 0);
  else if (min > max) obj.SetValue(ElementMaxProperty, min);
}
```

Note in particular that if *ElementMin* is set to a value greater than *ElementMax*, then *ElementMax* is set to the same value. Of course, it is a choice and there could be other solutions. The callback method for *ElementMax* is basically identical.

You can define dependency properties for any type that inherits the class *DependencyObject*, what all UI components do in practice. In WPF, most properties are, as mentioned, dependency properties, but it is not really straightforward to understand the difference between a

dependency property and a conventional property (a CLR property). A dependency property is defined as a static object, and technically the class *DependencyObject* has a collection of dependency properties. When a dependency property is registered, it is inserted into this collection as a *DependencyProperty* object with the name of the property as the key. To refer to that property, a *DependencyObject* defines two methods *GetValue()* and *SetValue()*, which with that property as parameter respectively reads and changes its value. Since you want to keep the usual notation for properties, you define so-called property wrappers, whose sole purpose is to ensure the same syntax as for CLR properties:

```
public int ElementMin
{
  get { return (int)GetValue(ElementMinProperty); }
  set { SetValue(ElementMinProperty, value); }
}

public int ElementMax
{
  get { return (int)GetValue(ElementMaxProperty); }
  set { SetValue(ElementMaxProperty, value); }
}
```

An attached property is as mentioned a dependency property registered with the method *RegisterAttached()*. The type that defines an attached property defines static methods, which have a parameter that defines the object which defines that property, for example the methods *GetBreakBefore()* and *SetBreakBefore()* (see below). For example, if you have a *FlowPanel* which contains a *Button*

```
<Button Name="cmdC" Content="Knap C" Width="100" Height="25"
  local:FlowPanel.BreakBefore="True" />
```

then the attribute *BreakBefore* is parsed to the following method call:

```
FlowPanel.SetBreakBefore(cmdC, true);
```

This method will then on the object *cmdC* call the method *SetValue()*:

```
cmdC.SetValue(FlowPanel.BreakBeforeProperty, true)
```

which stores that property for the container in a collection in the *Button* object along with the value. The runtime system therefore knows which property at the component's container to initialize and with what value. This is apparently a long way, but the reason is that since a component can occur in many different containers, it would in practice be impossible to equip the component with properties for all the properties that one should be able to reference corresponding to all possible containers. In this case, the two attached properties are defined as follows:

```
public static DependencyProperty BreakBeforeProperty =
  DependencyProperty.RegisterAttached("BreakBefore", typeof(bool),
  typeof(FlowPanel), null);
public static DependencyProperty BreakAfterProperty =
  DependencyProperty.RegisterAttached("BreakAfter", typeof(bool),
  typeof(FlowPanel), new PropertyMetadata(false));
```

To use attached properties, it is necessary to define static *get* and *set* methods. These methods are necessary to refer to the two properties in XML.

```
public static bool GetBreakBefore(UIElement element)
{
  return (bool)element.GetValue(BreakBeforeProperty);
}

public static void SetBreakBefore(UIElement element, bool value)
{
  element.SetValue(BreakBeforeProperty, value);
}

public static bool GetBreakAfter(UIElement element)
{
  return (bool)element.GetValue(BreakAfterProperty);
}

public static void SetBreakAfter(UIElement element, bool value)
{
  element.SetValue(BreakAfterProperty, value);
}
```

Now for the class *FlowPanel* two dependency properties and two attached properties are defined, and then there is the rest of the class that makes it a panel. To define a layout panel, you basically need to override two methods from the *Panel* class:

1. *MeasureOverride()*, which calculates how much space the components take up.
2. *ArrangeOverride()*, which is the method that places the components in the panel.

The first method has a parameter that specifies the current size of the panel, and the method must traverse all elements in a loop, and for each element the method *Measure()* must be executed. This method must be called to ensure that the *DesiredSize* element has a correct value. It is this value that is used to calculate the required size. The second method *ArrangeOverride()* puts out the components in the panel determined by their *Desiredsize*. Therefore, *MeasureOverride()* is performed before *ArrangeOverride()*. I will not show these methods here as they as they fill up a lot and are relatively technical, but the reader is encouraged to study the online version of the program.

Now the panel is done and I will show how it is used in the program:

```
<Window x:Class="ALayoutControl.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:ALayoutControl"
        mc:Ignorable="d"
        Title="A FlowLayout" Height="450" Width="500">
  <Grid Margin="10" Background="White">
    <Grid.RowDefinitions>
      <RowDefinition Height="30"/>
      <RowDefinition Height="*"/>
      <RowDefinition Height="30"/>
    </Grid.RowDefinitions>
    <TextBlock Text="Top" Grid.Row="0" ... />
    <TextBlock Text="Bottom" Grid.Row="2" ... />
    <local:FlowPanel Grid.Row="1" VerticalAlignment="Top"
    x:Name="wrapper"
      ElementMin="2" ElementMax="6">
      <Button Content="Button A" Width="100" Height="25" />
      <Button Content="Button B" Width="100" Height="35" />
      <Button Content="Button C" Width="100" Height="25"
        local:FlowPanel.BreakAfter="True" />
```

```
      <Button Content="Button D" Width="100" Height="25" />
      <Button Content="Button E" Width="100" Height="25"
        local:FlowPanel.BreakBefore="True" />
      <Button Content="Button F" Width="100" Height="25"
        local:FlowPanel.BreakBefore="True" />
      <Button Content="Button G" Width="100" Height="25" />
      <Button Content="Button H" Width="100" Height="25" />
      <Button Content="Button I" Width="100" Height="25" />
      <Button Content="Button J" Width="100" Height="25" />
      <Button Content="Button K" Width="100" Height="25" />
      <Button Content="Button L" Width="100" Height="25" />
      <Button Content="Button M" Width="100" Height="25" />
      <Button Content="Button N" Width="100" Height="25" />
      <Button Content="Button O" Width="120" Height="25" />
      <Button Content="Button P" Width="100" Height="25" />
      <Button Content="Button Q" Width="100" Height="25" />
      <Button Content="Button R" Width="100" Height="25" />
      <Button Content="Button S" Width="100" Height="25" />
      <Button Content="Button T" Width="100" Height="25" />
      <Button Content="Button U" Width="100" Height="25" />
      <Button Content="Button V" Width="150" Height="25" />
      <Button Content="Button W" Width="100" Height="25" />
      <Button Content="Button X" Width="100" Height="25" />
    </local:FlowPanel>
  </Grid>
</Window>
```

At the very first, there is a *Grid* with three rows alone for "something" at the top and bottom. The *FlowPanel* panel is thus nestled in a *Grid*. The panel places 24 buttons. The buttons have different sizes to show how the panel handles it. Using the dependency properties, the panel defines that each row must have at least 2 elements and a maximum of 6 elements. Note that these are custom dependency properties and that they are seen from XML as all other properties. Note that buttons C, E and F define an attached property for a line break. The result is, for example, that the first three elements (buttons) appear on the first line.

4 COMMANDS

A graphical user interface consists of a window with components that the user can do something with either using the mouse or the keyboard. When this happens to a component, it will typically raise an event. Others - windows, objects - can register as observers for that event and thus be notified that it has occurred and then do something. Thus events are closely related to the class in which they are defined or at least the classes derived from them. From the programmer's point of view, a large part of the work of writing programs with a graphical user interface thus consists of writing event handlers chopped up at certain events. This handling of events is fundamental in WPF, but Microsoft has defined an alternative, called commands, which corresponds to a number of typical tasks such as copy / paste that is built into the framework. The goal is of course to make the code simpler, as you have many functions that must be there by default, but also because you often want to activate the same functions in several ways, for example from a menu and a toolbar. In fact, I have used and processed commands before, but the following adds a few things and including commands which are part of the framework.

A command consists of four parts:

- a *Command* object representing the task to be performed
- a command source that triggers the command when it has to be executed
- a handler, which is a method that is executed when the command is activated
- a *CommandBinding* object that keeps track of which commands are associated with source objects and handlers

Exactly, a *Command* is an object that has a property that returns an object that implements the interface *ICommand*.

There are a number of pre-defined or built-in commands that you can use immediately. There are static objects that are referred to as properties in five static classes:

- *ApplicationCommands*
- *ComponentCommands*
- *EditingCommands*
- *MediaCommands*
- *NavigationCommands*

Each of these classes has many *Command* objects, some with default input bindings (certain key combinations), while for others it is left to the programmer to bind to certain objects and write the required handlers.

To use a *Command*, you must

- decide whether the command should be one of the built-in commands (and it often will be), or whether it should be a custom *Command*
- associate the command with one or more UI components and assign the command to one or more user interactions (keys, mouse)
- write a handler and thus the method the command should execute
- create a *CommandBinding* object that binds the command to the handler and possibly for a method *Command.CanExecute()* (explained below)
- add the binding to the component's (possibly the window object's) *Command* collection that triggers the command

Many WPF components implement the *ICommandSource* interface, which allows you to assign a command that fires automatically. For example implements a *Button* and a *MenuItem* the interface *ICommandSoure*, and if it has associated a *Command*, then the command is automatically executed when the user clicks on the component. The command has a *Command* property assigned either in XML or in code. You can also to a command associate specific user interactions in the form of certain key combinations or actions with the mouse, so that the command is executed when the actions in question occurs.

As mentioned, it is necessary to create a *CommandBinding* object, which binds the command to a handler, which is a method with the following signature:

```
void UserCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
}
```

and as the name says, *ExecutedRoutedEventArgs* is an object derived from *RoutedEventArgs*. The *CommandBinding* object's task is to link the entire command architecture together. A control or window has a *CommandBinding* collection, and if you attach a *CommandBinding* object to this collection, the corresponding *Command* will be executed when the current user interaction occurs.

Any component inclusive the *Window* object has its own *CommandBinding* collection, and commands, like *RoutedEvents*, will bubble up in the visual tree. Here commands will search for an associated binding starting in the component from which it originated.

A command that does not have a binding is automatically inactive, and nothing happens if you try to trigger the command, and a control if the *Command* property refers to that command will appear disabled in the user interface. However, you may be interested in a command with a binding and the corresponding control should be inactive. For example

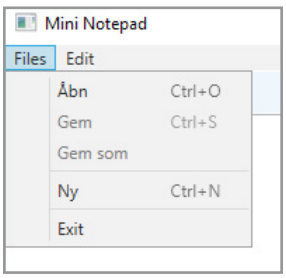
you could want a *Save* button to be disabled until the document is changed. This can be controlled by associating a method *CanExecute()* (the explanation follows):

```
void User_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
}
```

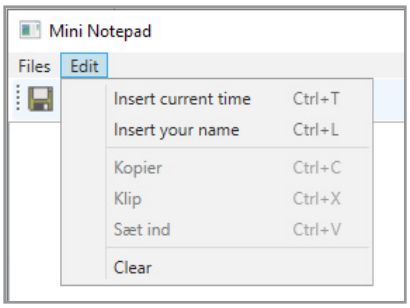
4.1 USING COMMANDS

To use commands I will show an example that you can think of as a simplified *Notepad*, that is a program where you can edit the contents of a simple text file. Note that I also used this examples in the book C# 2, but without commands. The program must be able to open and save a file, and for this you must especially be able to find files or their location in the file system. For that I will use standard file dialogs. However, they do not exist as WPF components, so the program uses the corresponding Windows dialog boxes. The functionality should be simple and the application's user interface should consist of a window with a menu, a toolbar and a multi-line *TextBox*. The toolbar has five functions (buttons), and each function has a corresponding function in the menu:

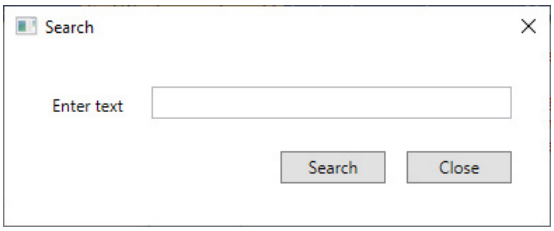
The *File* menu has four menu items:



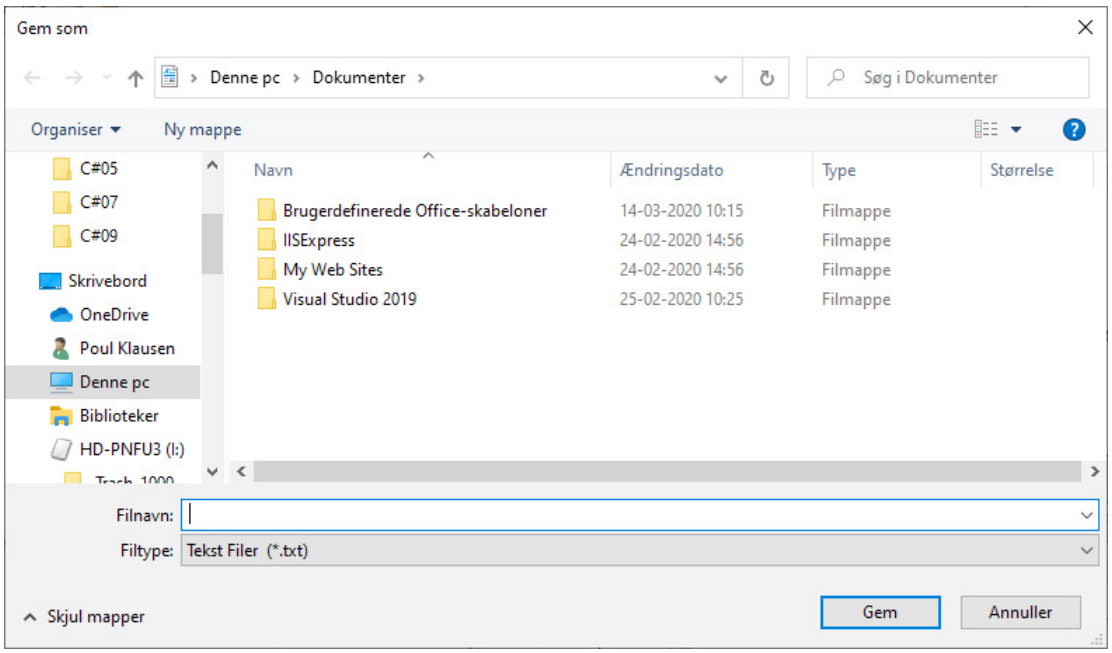
and the same goes for the menu *Edit*, but a menu with six menu items:



If something is entered in the window and you press Ctrl + F, you get a search window:



If you have entered something and you press the *Save* button (or select the *Save* menu item) you will get the following default dialog box:



The same goes for *Save as* and *Open*.

I will start with the XML to the *MainWindow*:

```
<Window x:Class="MiniNotepad.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:MiniNotepad"
        mc:Ignorable="d"
        Title="Mini Notepad" Height="450" Width="800"
        Activated="MainWindow_Activated" Closing="MainWindow_Closing">
  <Window.CommandBindings>
    <CommandBinding Command="New" Executed="NewExecuted"/>
    <CommandBinding Command="Open" Executed="OpenExecuted"
      CanExecute="OpenCanExecute"/>
    <CommandBinding Command="Save" Executed="SaveExecuted"
      CanExecute="SaveCanExecute"/>
    <CommandBinding Command="SaveAs" Executed="SaveAsExecuted"
      CanExecute="SaveAsCanExecute"/>
  </Window.CommandBindings>
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="Files">
        <MenuItem Command="Open"/>
        <MenuItem Command="Save"/>
        <MenuItem Command="SaveAs"/>
        <Separator/>
        <MenuItem Command="New"/>
        <Separator/>
        <MenuItem Header="Exit"/>
      </MenuItem>
      <MenuItem Header="Edit">
        <MenuItem Command="local:CustomCommands.InsertTime"/>
        <MenuItem Command="local:CustomCommands.InsertName"/>
        <Separator/>
        <MenuItem Command="ApplicationCommands.Copy"/>
        <MenuItem Command="ApplicationCommands.Cut"/>
        <MenuItem Command="ApplicationCommands.Paste"/>
        <Separator/>
        <MenuItem Name="Clear" Header="Clear" Click="Clear_Click"/>
      </MenuItem>
```

```
    </Menu>
    <ToolBar DockPanel.Dock="Top" >
      <Button Command="Save">
        <Image Source="Icons\save.gif" Width="20" />
      </Button>
      <Button Command="SaveAs">
        <Image Source="Icons\saveas.gif" Width="20" />
      </Button>
      <Separator/>
      <Button Command="ApplicationCommands.Copy">
        <Image Source="Icons\copy.gif" Width="20" />
      </Button>
      <Button Command="ApplicationCommands.Cut">
        <Image Source="Icons\cut.gif" Width="20" />
      </Button>
      <Button Command="ApplicationCommands.Paste">
        <Image Source="Icons\paste.gif" Width="20" />
      </Button>
    </ToolBar>
    <TextBox Name="txtEdit" ScrollViewer.VerticalScrollBarVisibility="A
    uto"
      AcceptsReturn="True" TextChanged="txtEdit_TextChanged">
    </TextBox>
  </DockPanel>
</Window>
```

The window element defines two event handlers for respectively *Activated* and *Closing*. The first is used solel1y to give the input field focus at program launch, while the second is used to give the user a warning when the program is closed and the content is changed so that the user can save the content.

Next, 4 *Command* bindings are defined for the four file operations:

Command	Method	Control	Action
New	*NewExecuted()*		Create a new file
Open	*OpenExecuted()*	*OpenCanExecute()*	Open a *File Open* dialog box
Save	*SaveExecuted()*	*SaveCanExecute()*	Save the file, and if the file is not created open a *File Save* dialog box
SaveAs	*SaveAsExecuted()*	*SaveAsCanExecute()*	Open a *File Save* dialog box

The methods are relatively simple and I do not show the code here. When examining the code, you should especially notice how to open the various file dialog boxes.

The next part of the XML code creates the menu. In the *File* menu you refer to the above commands. The first two menu items in the *Edit* menu bind to custom *Commands* (see below), while the next three menu items bind to standard *ApplicationCommands* for copy, cut and paste. Since these are standard commands, you do not have to do anything in the program, and all the usual copy / paste functionality is automatically supported. The *Edit* menu has one last menu item associated with a regular event handler. This is only to show that customary event handling and commands can easily coexist.

The program also has a toolbar. It has five features, all of which are also available in the menu. You should notice how they bind to the same commands as in the menu.

Finally, there is the input field, where you should especially notice the event handler for *TextChanged*.

The first two commands bound to the *Edit* menu are custom commands, and are defined as follows:

```
class CustomCommands
{
  private static RoutedUICommand insertName_command;
  private static RoutedUICommand insertTime_command;

  static CustomCommands()
  {
    insertName_command = CreateCommand(Key.L, "Insert your name",
    "NameCommand");
    insertTime_command =
      CreateCommand(Key.T, "Insert current time", "TimeCommand");
  }

  static RoutedUICommand CreateCommand(Key key, string text, string
  name)
  {
    InputGestureCollection gesture = new InputGestureCollection();
    gesture.Add(new KeyGesture(key, ModifierKeys.Control));
    return new RoutedUICommand(text, name, typeof(CustomCommands),
    gesture);
  }
```

```
  public static RoutedUICommand InsertTime
  {
    get { return insertTime_command;}
  }

  public static RoutedUICommand InsertName
  {
    get { return insertName_command; }
  }
}
```

Two commands are defined as static objects and are called:

- *insertName_command*
- *insertTime_command*

They are created in a static constructor with the parameters

- the key combination that should trigger the command
- a description
- the name of the command

Note that here alone is defined how the command is activated and what it is called, but nothing about what it does. The commands are bound to actions in the main program:

```
public MainWindow()
{
  InitializeComponent();
  SeekBinding();
  BindCommand(CustomCommands.InsertName, InsertNameHandler,
  InsertNameEnabled);
  BindCommand(CustomCommands.InsertTime, InsertTimeHandler,
  InsertTimeEnabled);
}

private void BindCommand(RoutedUICommand command,
  ExecutedRoutedEventHandler handler, CanExecuteRoutedEventHandler
  enabled)
{
  CommandBinding binding = new CommandBinding();
  binding.Command = command;
```

```
  binding.Executed += handler;
  binding.CanExecute += enabled;
  CommandBindings.Add(binding);
}

private void InsertTimeEnabled(object sender,
CanExecuteRoutedEventArgs e)
{
  e.CanExecute = txtEdit.CaretIndex >= 0;
}

private void InsertTimeHandler(object sender, ExecutedRoutedEventArgs e)
{
  int p = txtEdit.CaretIndex;
  if (p >= 0)
  {
    DateTime dt = DateTime.Now;
    string time =
      string.Format(" {0} {1} ", dt.ToLongDateString(),
      dt.ToLongTimeString());
    string text = txtEdit.Text;
    txtEdit.Text = text.Substring(0, p) + time + text.Substring(p);
    txtEdit.CaretIndex = p + time.Length;
  }
}
private void InsertNameEnabled(object sender,
CanExecuteRoutedEventArgs e)
{
  e.CanExecute = !txtEdit.Text.Contains(userText);
}

private void InsertNameHandler(object sender, ExecutedRoutedEventArgs e)
{
  txtEdit.AppendText(userText);
}
```

The first command - *InsertName* - is bound to insert my name and address at the end of the document. The command can be executed if the text does not already appear in the document.

The second command inserts the date and time in the document where the cursor is.

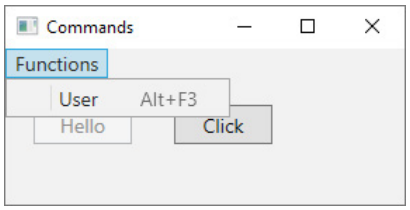
The constructor has another command which calls the method *SeekBinding()*. It also defines a *Command* binding:

```
private void SeekBinding()
{
  CommandBinding cb = new CommandBinding(ApplicationCommands.Find);
  cb.CanExecute += SeekCanExecute;
  cb.Executed += SeekExecuted;
  CommandBindings.Add(cb);
}

private void SeekCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
  e.CanExecute = txtEdit.Text.Length > 0;
}

private void SeekExecuted(object sender, ExecutedRoutedEventArgs e)
{
  SeekDialog dlg = new SeekDialog();
  dlg.Seek += SeekHandler;
  dlg.Show();
}

private void SeekHandler(object sender, SeekRoutedEventArgs e)
{
  string text = e.Text;
  if (text.Length > 0)
  {
    int pos = txtEdit.Text.IndexOf(text);
    if (pos >= 0)
    {
      txtEdit.Select(pos, text.Length);
      txtEdit.ScrollToLine(txtEdit.GetLineIndexFromCharacterIndex(pos));
    }
    else
    {
      txtEdit.Select(0, 0);
    }
  }
}
```

It is again an *ApplicationCommad* command called *Find* and linked to Ctrl + F. The command can be executed if text is entered. When executed, a window opens where you can enter a search text, but before is assigned a handler that searches for the text, when you in the search window click *Search*. If the handler finds the text, it is highlighted and the window is scrolled so that the text is visible.

EXERCISE 4: TWO COMMANDS

Create a new WPF application project as you can call *Commands*. Design the following *MainWindow*:



It is a very simple window with a menu with a single menu item and two buttons. The exercise is

1. assign an *Open* command the *Hello* button
2. assign a custom command to the *Click* button and the menu item (the same command)

When the program starts, the *Hello* button must be disabled. When you click on the *Click* button, click on the menu item or enter Alt+F3 the program must open a simple message box and enable the *Hello* button. When you then click on the *Hello* button the program must open another message box and disable the button again.

To keep track of where the *Hello* button should be disabled or not, add an instance variable to the class *MainWindow*:

```
private bool ready = false;
```

To assign the *Open* command you must do the following:

1. Add the two command methods for an *ApplicationCommands.Open* command to the class *MainWindow*, where the methods must be called *OpenCommand_CanExecute()* and *OpenCommand_Executed()*.
2. Define a command binding for an *ApplicationCommands.Open* command in XML
3. Set the *Command* property for the button in XML

For the custom command you must add a static class to create the command:

```
public static class CustomCommands
{
  public static readonly RoutedUICommand User =
    new RoutedUICommand("User", "User", typeof(CustomCommands),
    new InputGestureCollection()
    {
      new KeyGesture(Key.F3, ModifierKeys.Alt)
    }
  );
}
```

Then you must assign the command in the same way as above.

5 USER DEFINED CONTROLS

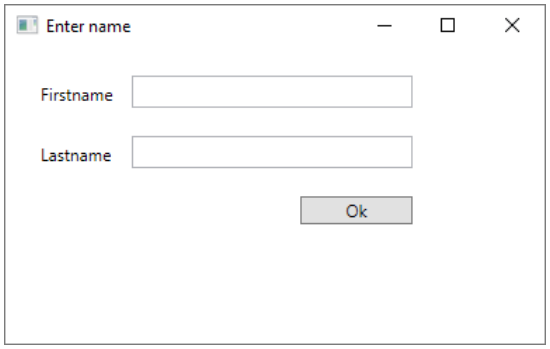
In this chapter I will look at how to create custom controls. I already have used custom controls above, but there is a lot more to note and in fact it is a relatively complex area. There are basically three ways

- *user controls*, which is really just a matter of gathering and processing multiple WPF controls as a whole
- *custom controls*, inheriting an existing control
- *template controls*, overriding how to render an existing control

In the following, I will show how to create custom controls in each of the three ways.

5.1 USER CONTROLS

A user control is also sometimes called a composite control similar to that it is composed of other controls. You create a user control by adding a User Control (WPF) to your project, and then the rest goes almost by itself. This example shows a classic example of a user control, which consists of a label and an input field. If you run the program, you get the window:



The window contains two user controls, both of which (encapsulate) a *Label* and a *TextBox*, and then there is a button. If you enter in an input field, the text in the *Label* component becomes bold, to indicate that it is now the control that has the focus. Clicking on the button will get a message box showing the contents of the two user controls. The reason for such user control should be something, that in a window you often need input fields and that such an input field will usually have a text attached. Or, the rationale is to show how to create and use a *user control*.

To add a user control to a project you add a *User Control (WPF)*. In this case I have added a user control called *InputControl* and the result is an XML file for the user interface as well as a code behind. The XML file is:

```
<UserControl x:Class="NameProgram.InputControl"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
             presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
             compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/
             blend/2008"
             xmlns:local="clr-namespace:NameProgram"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="800">
  <StackPanel Name="panel" Orientation="Horizontal"
  VerticalAlignment="Center"
    Height="23">
    <Label Name="lbl" Content="{Binding Header}" Width="{Binding
    HeaderWidth}" />
    <TextBox Name="txt" Width="100" GotFocus="txt_GotFocus" LostFocus=
      "txt_LostFocus" Text="{Binding Test}" Width="{Binding TextWidth}"
      />
  </StackPanel>
</UserControl>
```

There really isn't much to explain, but you should note that the type this time is *UserControl* instead of *Window*. Otherwise, the control is defined as a *StackPanel* with a *Label* and a *TextBox*. As for the last one, note that it captures events related to focus and also that the components are bound to properties in code behind.

Then there is the C# code, which should primarily define the properties that the component must make available. A user control generally has the same properties as other UI controls, but these properties relate to the container itself and thus a *UserControl* and are basically inherited to the individual components. To the extent that a user control must have special properties for the individual components, they must be defined in code behind and in order for them to work when working with the design in XML, they must be defined as dependency properties. In this case I need four properties:

1. *Header*, which is the header text and as default is blank
2. *HeaderWidth*, which is the width of the header as an *int* and has the default value 50

3. *Text*, which is the content of the input field and as default is blank
4. *TextWidth*, which is the width of the input field as an *int* and has the default value 100

The code behind can then be written as:

```
public partial class InputControl : UserControl
{
  public static readonly DependencyProperty HeaderProperty =
    DependencyProperty.Register("Header", typeof(string),
    typeof(InputControl));
  public static readonly DependencyProperty HeaderWidthProperty =
    DependencyProperty.Register("HeaderWidth", typeof(int),
    typeof(InputControl),
    null, new ValidateValueCallback(ValidateWidth));
  public static readonly DependencyProperty TextProperty =
    DependencyProperty.Register("Text", typeof(string),
    typeof(InputControl));
  public static readonly DependencyProperty TextWidthProperty =
    DependencyProperty.Register("TextWidth", typeof(int),
    typeof(InputControl),
    null, new ValidateValueCallback(ValidateWidth));

  public InputControl()
  {
    InitializeComponent();
    DataContext = this;
  }

  public string Header
  {
    get { return (string)GetValue(HeaderProperty); }
    set { SetValue(HeaderProperty, value); }
  }

  public int HeaderWidth
  {
    get { return (int)GetValue(HeaderWidthProperty); }
    set { SetValue(HeaderWidthProperty, value); }
  }

  public int TextWidth
  {
    get { return (int)GetValue(TextWidthProperty); }
    set { SetValue(TextWidthProperty, value); }
  }

  public string Text
```

```
  {
    get { return (string)GetValue(TextProperty); }
    set { SetValue(TextProperty, value); }
  }

  public static bool ValidateWidth(object value)
  {
    try
    {
      int number = (int)value;
      return number >= 0;
    }
    catch
    {
      return false;
    }
  }

  private void txt_GotFocus(object sender, RoutedEventArgs e)
  {
    lbl.FontWeight = FontWeights.Bold;
  }

  private void txt_LostFocus(object sender, RoutedEventArgs e)
  {
    lbl.FontWeight = FontWeights.Normal;
  }
}
```

It's not very strange in that, but in this case it's just wrapper properties for the two controls, and the possibilities are quite limited. Of course, a user control can also have other properties for its own instance variables, just as it can have methods and define events. The result of it all is that it is simple to define user controls that way.

Back there is the main program that uses the control:

```
<StackPanel Margin="20">
  <local:InputControl x:Name="txtFirst" Header="Firstname"
  HeaderWidth="70"
    TextWidth="200" />
  <local:InputControl x:Name="txtLast" Header="Lastname"
  HeaderWidth="70"
    TextWidth="200" Margin="0,20,0,0" />
  <Button Name="cmd" Content="Ok" Width="80" Margin="190,20,0,0"
    HorizontalAlignment="Left" Click="cmd_Click" />
</StackPanel>
```

and here you should especially notice that a user control is used in the same way as all other controls, but also that you can use its properties in XML.

This example of a user control is extremely simple and the primary is to show what a user control is. You should note that in the preceding chapters I have shown examples of more advanced user controls.

EXERCISE 5: LABELPROGRAM

In this exercise you you must make an improvement on the above user control *InputControl*. Create a new project called *LabelProgram* and copy the user control *InputControl* to this project. Remember to change the namespace. Then create the same *MainWindow* as in the program *NameProgram* (you can copy the code from this project). Test the program. It should work in the same way as the program *NameProgram*.

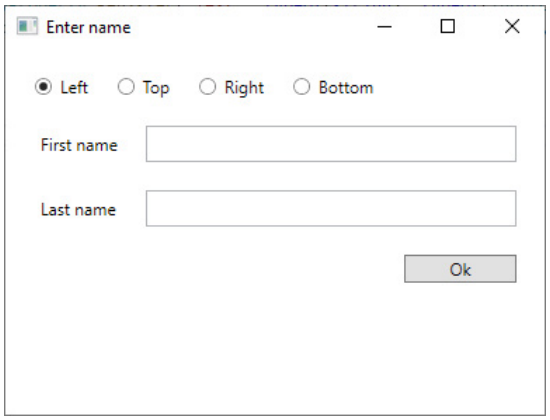
You must then change the user control such the label can be placed left, above, right or below the *TextBox*. Note that this means that you must redesign the control.

Add the following type to the project:

```
public enum LabelPosition { LEFT, TOP, RIGHT, BOTTOM };
```

and then add a dependency property of the type *LabelPosition* called *Position*. The property must indicate where to place the label, and if the property's value is changed, the label must move to right position.

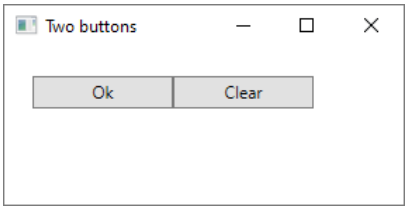
You must then update the main window to:



where it still has two *InputControl* controls. The *RadioButtons* at the top should be used to move the labels.

5.2 CUSTOM CONTROLS

A *custom control* is also a user control, but unlike the above, a custom control is a control that inherits another control. A good example is the control *FlowPanel* in chapter 3. The following example - called *TwoButtons* - is a very simple example of a custom control and should only serve to show what it is. The program opens a window with two buttons:



Here the left is a custom control, while the right is a usual control.

It is very simple to program a custom control, as you simply have to add a class to the project, which inherits another component. In this case, it is a class that inherits the class *Button*:

```
using System.Windows.Controls;

namespace TwoButtons
{
  public class SpecialButton : Button
  {
    private int _count = 0;

    public int Count
    {
      get { return _count; }
    }

    public int Step { get; set; }

    public void Clear()
    {
    _count = 0;
  }

    protected override void OnClick()
    {
      _count += Step;
      base.OnClick();
    }
  }
}
```

The class extends a *Button* with two new properties. One is a readonly property that is counted each time the button is clicked. The second is a property which defines how much the value of *Count* should be counted when the button is clicked. In addition, there is a method that sets the value of *Count* to 0. Most important, however, is the last method that overrides *OnClick()* and is the method that is executed when the button is clicked. Note that it is important that the method ends with calling the base class's *OnClick()*.

Below is the XML code for the main window:

```
<StackPanel Orientation=”Horizontal” VerticalAlignment=”Top”
Margin=”20”>
  <local:SpecialButton x:Name=”cmd” Content=”Ok” Width=”100”
  Height=”23” Step=”2”
    Click=”cmd_Click” />
  <Button Name=”clr” Content=”Clear” Width=”100” Height=”23”
Click=”clr_Click” />
</StackPanel>
```

Here you should note that you can use the new control and its properties from XML, but there is a big problem, however, as the designer window does not know it and so you do not get help.

EXERCISE 6: ENTERBOX

A *TextBox* control is a component to enter text and as default you can enter any text. Many times you are interested in the fact that it is only possible to enter numbers, and one way to solve it is to define a custom control derived from *TextBox*. In this exercise you must write such a control.

Create a new project called *EnterProgram* and add a custom control called *EnterBox*, which is a class called *EnterBox* that inherits *TextBox*. The custom control should be used to enter one of the following items:

1. ordinary text
2. integers, may be preceded by a minus sign
3. integers as hexadecimal numbers, may be preceded by a minus sign
4. decimal numbers, may be preceded by a minus sign

Add the following type to the file with the custom control:

```
public enum EnterMode { NORMAL, INTEGER, DECIMAL, HEXA };
```

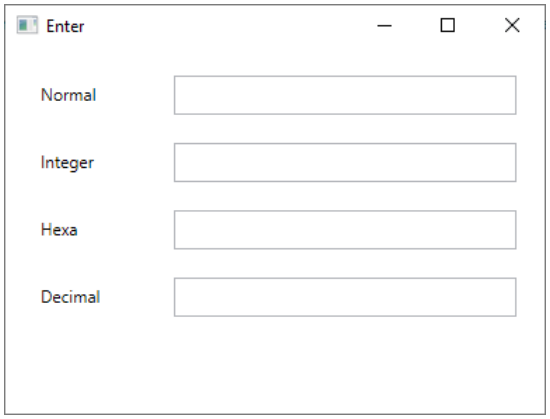
The control must have a dependency property *Mode* of that type that should define which of the above four items the control should be used for.

Next you must override the event handler *OnPreViewKeyDown()*:

```
protected override void OnPreviewKeyDown(KeyEventArgs e)
{
  if (Mode == EnterMode.INTEGER)
    { if (IsInteger(e.Key)) base.OnPreviewKeyDown (e); else e.Handled
    = true; }
  else if (Mode == EnterMode.DECIMAL)
    { if (IsDecimal(e.Key)) base.OnPreviewKeyDown (e); else e.Handled
    = true; }
  else if (Mode == EnterMode.HEXA)
    { if (IsHexa(e.Key)) base.OnPreviewKeyDown (e); else e.Handled =
    true; }
  else base.OnPreviewKeyDown (e);
}
```

where the three methods *IsInteger()*, *IsHexa()* and *IsDecimal()* should be used to validate which keys are legal for each of the three types the control should accept.

You must then write the *MainWindow* to test your new custom control where the window should have four *EnterBox* controls to enter an item of each of the four types: Ordinary string, integer, hexadecimal integer and a decimal number:

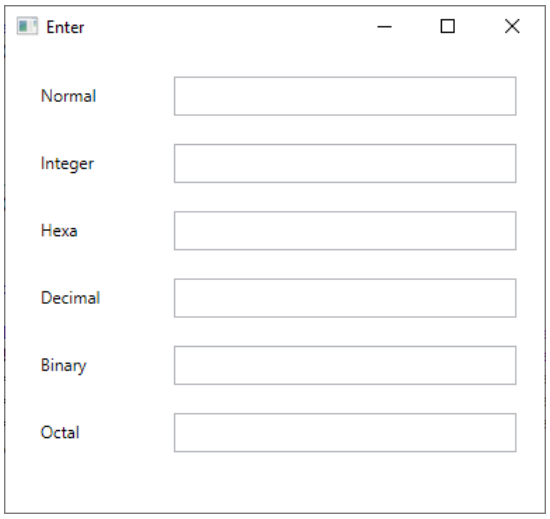


Here is one problem back. The control is basically a *TextBox* and has a property *AcceptsReturn* which is a dependency property. In principle, it makes no sense to use an *EnterBox* as multi-line control. To solve this issue, override this property in a static constructor:

```
static EnterBox()
{
  AcceptsReturnProperty.OverrideMetadata(typeof(EnterBox),
    new FrameworkPropertyMetadata(false, null, AcceptsReturnModified));
}
private static object AcceptsReturnModified(DependencyObject d, object
baseValue)
{
  return false;
}
```

EXERCISE 7: AN EXPANDED ENTERBOX

Make a copy of the project from exercise 5. Update the user interface with two new fields:



where the first should be used to enter a binary number and the other to enter an octal number. Add a dependency property *Lets* for a string, where the user in a string can indicate which keys an *EnterBox* should accept. Use this property to enter a binary number.

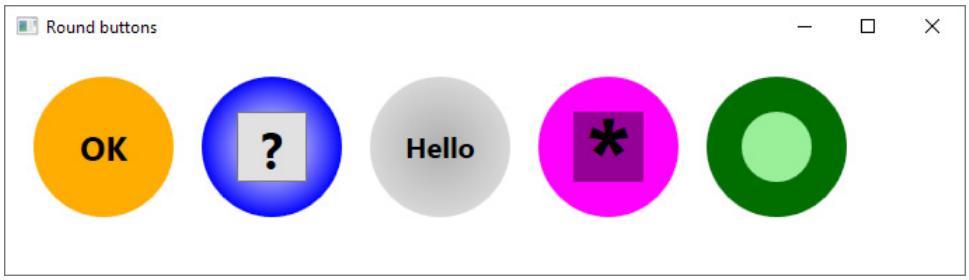
This solution can be smart for entering short texts, where there are only certain characters you want to accept, but you cannot immediately specify special keys such as arrow keys, backspace and so on. To solve these problems add another dependency property to *EnterBox* when the type must be *List<Key>*. Use this property for an *EnterBox* field to enter an octal number, when it should be possible to user ordinary keys for digits as well as numeric keyboard, left arrow, right arrow and backspace.

5.3 TEMPLATE CONTROLS

Above, I have shown two examples of user defined controls, the first one is composed of existing controls and called a *user control*, while the second inherits an existing control and is called a *custom control*. The use is different, with the first one typically used to define a control to be used in several places in the same application, while the second is more general and can be easily used in different applications. Writing a user control is simple, while the work of writing a custom control can be greater (see, for example, the control *FlowPanel* from chapter 3).

For both a user control and a custom control, the work includes defining how the control works, but in many contexts you are also interested in changing how the control looks and thus how the control is displayed in the user interface. It is easy with a *template control*. Each component has a default control template that defines how the component should be displayed. It is possible to define its own template for a component and thus completely determine how it appears on the screen. In this section I show how.

A classic example of a custom control is a round button. Since it is a component that must have the same properties as a regular button, but simply needs to be drawn or displayed in a different way, it is obvious to use a template. The example *ARoundButton* opens a window with 5 (actually 7) buttons:



Each circle is a button and the difference is how they are made and how they behave with the mouse. If you look at the first one, it is defined as follows:

```
<Button Width="100" Height="100" Click="cmdOk_Click">
  <Button.Template>
    <ControlTemplate>
      <Grid>
        <Ellipse Fill="Orange"/>
        <Label VerticalAlignment="Center" HorizontalAlignment="Center"
          FontWeight="Bold" FontSize="24" Content="OK" />
        </Grid>
      </ControlTemplate>
    </Button.Template>
</Button>
```

It is a usual *Button* component and so you can use all the usual properties such as *Width* and *Height* as well as the *Click* event. However, you cannot use the properties that have to do with how the component is displayed on the screen, as they are not used by the component's template, which this time has been replaced by a custom template. This applies to properties such as *Content* and *Background*, but also the font used. The component is relatively hard-coded and will for example always show the same text, and nothing happens when you click on it - other than the event handler is executed - but there is no visual effect that shows that you click with the mouse.

The next button is actually a button with a button and is defined as follows:

```
<Button Margin="20,0,0,0" Width="100" Height="100"
  Template="{DynamicResource RoundedButton}" Click="cmdOuter_Click">
  <Button.Content>
    <Button Content="?" FontSize="36" FontWeight="Bold" Width="50"
    Height="50"
    Click="cmdInner_Click" />
  </Button.Content>
  <Button.Background>
    <RadialGradientBrush>
      <GradientStop Color="White" Offset="0" />
      <GradientStop Color="Blue" Offset="1" />
    </RadialGradientBrush>
  </Button.Background>
</Button>
```

This time the template is a static resource defined in *App.xaml*:

```
<ControlTemplate x:Key="RoundedButton" TargetType="Button">
  <Grid>
    <Ellipse Fill="{TemplateBinding Background}"/>
    <ContentPresenter HorizontalAlignment="Center"
    VerticalAlignment="Center"/>
  </Grid>
</ControlTemplate>
```

First, note that it is a control template for a *Button*. It says the button should appear as a *Grid* with an *Ellipse* and a so-called *ContentPresenter*. It is (a very comprehensive) class used to present some content in a *Content* control. It will be the object associated with the button's *Content* property. The result is that the button content can be anything and it is centered in the button. In addition, the background color of the ellipse is associated with the button's *Background* property.

Looking at the definition of the button in the main window, the button's template is set to the above custom template. *Content* is a usual button and the background is drawn with a *GradientBrush*. Note that the outer (the round blue button) and the inner button each have their own event handler, thus acting as two buttons. In order for the inner button not to reach the outer button (it is a routed event), the event handler must define it as processed:

```
private void cmdInner_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("I'm inner");
  e.Handled = true;
}
```

Visually, the button is still "dead", so there is no visual effect of clicking it. The inner button does not have a custom template and thus behaves in the usual way.

Then there is the middle button containing the text Hello. Moving the mouse over it shows it as gray:

and clicking on it (holding down the mouse button) changes the size. That is that there are now effects attached to the button regarding the mouse. In button's XML there is not much to see:

```
<Button Margin="20,0,0,0" Background="Tomato" Content="Hello"
  Click="cmdHello_Click" Style="{StaticResource RoundedStyle}" />
```

and the whole thing is of course hidden in the style of the button. Note that apart from this style, the button and its properties are completely defined as usual. The style is defined in *App.xaml* and in turn is both comprehensive and complex:

```
<Style x:Key =”RoundedStyle” TargetType =”Button”>
  <Setter Property =”Foreground” Value =”Black”/>
  <Setter Property =”FontSize” Value =”24”/>
  <Setter Property =”FontWeight” Value =”Bold”/>
  <Setter Property=”Width” Value=”100”/>
  <Setter Property=”Height” Value=”100”/>
  <Setter Property =”Template”>
    <Setter.Value>
      <ControlTemplate TargetType =”Button”>
        <Grid x:Name=”cmdGrid”>
          <Ellipse x:Name=”cmdFill” Fill=”{TemplateBinding Background}”
          />
          <Label x:Name=”cmdText” Content =”{TemplateBinding Content}”
            FontSize=”20” FontWeight=”Bold” HorizontalAlignment=”Center”
            VerticalAlignment=”Center” />
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property=”IsMouseOver” Value=”True”>
            <Setter TargetName=”cmdFill” Property=”Fill”>
              <Setter.Value>
                <RadialGradientBrush>
                  <GradientStop Color=”LightGray” Offset=”0”/>
                  <GradientStop Color=”DarkGray” Offset=”1”/>
                </RadialGradientBrush>
              </Setter.Value>
            </Setter>
          <Setter TargetName =”cmdText” Property=”Foreground”
          Value=”Gray”/>
        </Trigger>
        <Trigger Property = “IsPressed” Value=”True”>
          <Setter TargetName=”cmdGrid” Property=”RenderTransformOrigin”
            Value=”0.5,0.5”/>
```

If one systematically follows the definition from start to finish, it is not really difficult to understand the content, but first a remark. If you look at this chapter, in particular this example and to some extent also other examples in this book, I have used concepts that first are explained later. These are concepts (components) such as *Rectangle* and *Ellipse* as well as *GradientBrush*, and these are all concepts that are dealt with in more detail later in this book together with similar substance.

The style is a usual style for a *Button*. First, values for font and size are defined. Note that these values can be overridden for the individual properties in the declaration of the button in XML if desired. Specifically, a template is defined where the button is drawn as an ellipse and a label, but so that these are bound to the two properties *Background* and *Content*.

Next, a trigger is defined for respectively. *IsMouseOver* and *IsPressed*. The first causes the button to be drawn with a different background and a different text color as the mouse moves over the button. The second causes a simple animation (also explained later) when the button is clicked.

The fourth button is defined as follows:

```
<Button Margin="20,0,0,0" Background="Magenta"
  Style ="{StaticResource RoundedStyle}" Click="cmdMagenta_Click">
  <Button.Content>
    <Grid>
      <Rectangle Fill="DarkMagenta" Width="50" Height="50" />
      <TextBlock Text="*" FontWeight="Bold" FontSize="72"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
  </Button.Content>
</Button>
```

It uses the same style as the previous button, but its *Content* is a *Grid* with a *Rectangle* with a *TextBlock*. The button should primarily show that *Content* can be anything. You should note that you can click on both the rectangle and the ellipse.

Finally, there is the last button

```
<Button Margin="20,0,0,0" Background="DarkGreen"
  Style ="{StaticResource RoundedStyle}" Click="cmdOuter_Click">
  <Button.Content>
    <Button Style="{StaticResource RoundedStyle}"
Background="LightGreen"
     Width="50" Height="50" Click="cmdInner_Click" />
  </Button.Content>
</Button>
```

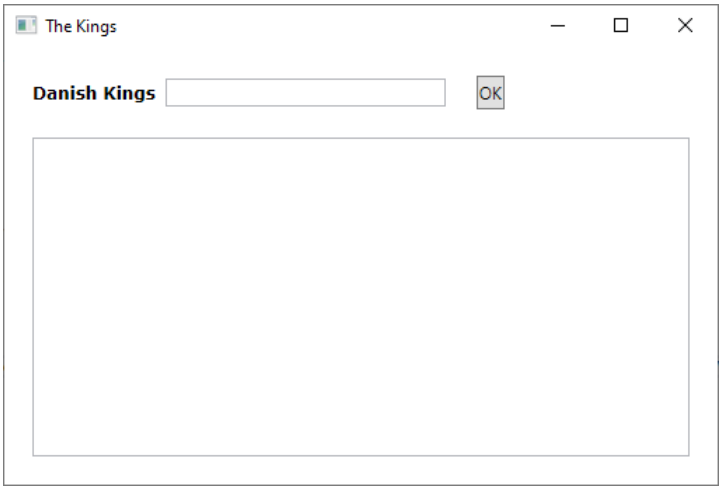
which is a button with a button and both buttons have the *RoundedStyle* style. For example, you must note that if you click on the inner button, it is only this button which is animated.

The conclusion is that writing template control is not very simple, and it generally requires a high degree of familiarity with styles, but with practice it is possible.

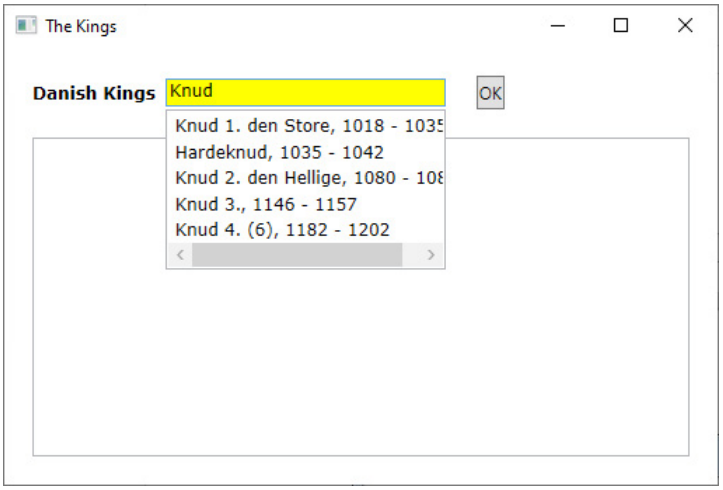
5.4 AN AUTO COMPLETE TEXTBOX

As the last example of a custom control, I will look at an example similar to the control in *NameProgram* that is a control with a label and an input field. It should be possible to specify values for multiple properties, and in addition, the input field must have a list of values that should be considered as suggestions and which should work in that way as you enter text in the input field, they are filtered values away from the list that do not match the entered text. In addition, if you place the cursor on a particular line in the list and press Enter, that line must be inserted in the field. A control of that kind is usual called an auto complete field, and in fact is a *ComboBox* of that kind.

If you run the program you get the following window:



where the text and input field is a User Control, while the window also has a button and a list box. If you enter a text and click the button, the text is inserted as a line in the list box. The result is shown below after entering the text *Knud*:



The list in the above window shows names of Danish kings as well as there reign. These are text representations of objects of the form:

```
public class King
{
  public int From { get; private set; }
  public int To { get; private set; }
  public string Name { get; private set; }

  public King(string name, int from, int to)
  {
    Name = name;
    From = from;
    To = to > 0 ? to : 9999;
  }

  public override string ToString()
  {
    return string.Format("{0}, {1} - {2}", Name,
      From > 0 ? "" + From : "", To < 9999 ? "" + To : "");
  }
}
```

Below is a list of Danish kings:

```
public class Kings : IEnumerable<King>
{
  private List<King> list;

  public Kings()
  {
    CreateList();
  }

  public int Count
  {
    get { return list.Count; }
  }

  public King this[int n]
  {
    get { return list[n]; }
  }
```

```
  public IEnumerator<King> GetEnumerator()
  {
    return list.GetEnumerator();
  }

  System.Collections.IEnumerator System.Collections.IEnumerable.
  GetEnumerator()
  {
    return GetEnumerator();
  }

  private void CreateList()
  {
    list = new List<King>();
    for (int i = 0; i < data.Length; ++i)
    {
      try
      {
        string[] elem = data[i].Split('-');
        int from = int.Parse(elem[0].Trim());
        int to = int.Parse(elem[1].Trim());
        string name = elem[2].Trim();
        list.Add(new King(name, from, to));
      }
      catch
      {
      }
    }
  }

  #region
    private static string[] data =
    {
      "0000 - 0958 - Gorm den Gamle",
      "0958 - 0986 - Harald 1. Blåtand",
      "0986 - 1014 - Svend 1. Tveskæg",
      ....
    };
  #endregion
}
```

This time the control is complex and I will not show the whole code. Relating to the XML there is not much to note besides a *TextBlock* and a *TextBox* in a *StackPanel*:

```
<StackPanel Name="panel" Height="24" Orientation="Horizontal" >
  <TextBlock Name="head" Text="Text" VerticalAlignment="Center"
  Margin="0,0,5,0" />
  <TextBox Name="txt" Margin="2" Width="100" />
</StackPanel>
```

The control must have several properties, which are both overrides of existing properties as well as new ones. This time they are defined (in the same way as the standard components) as the dependency properties. Below I have shown the definition of two where the first (with *new*) overrides *Height* while the second is a new property:

```
public new static readonly DependencyProperty HeightProperty =
  DependencyProperty.Register("Height", typeof(double),
  typeof(AutoControl),
  new FrameworkPropertyMetadata(new PropertyChangedCallback(OnHeightCh
  anged)));
public static readonly DependencyProperty HeaderProperty =
  DependencyProperty.Register("Header", typeof(string),
  typeof(AutoControl),
  new FrameworkPropertyMetadata(new PropertyChangedCallback(OnHeaderCh
  anged)));
```

For both properties, *OnChanged* handlers are defined which are executed if the value of the property is changed:

```
private static void OnHeightChanged(DependencyObject sender,
  DependencyPropertyChangedEventArgs e)
{
  ((AutoControl)sender).panel.Height = (double)e.NewValue;
}

private static void OnHeaderChanged(DependencyObject sender,
  DependencyPropertyChangedEventArgs e)
{
  ((AutoControl)sender).head.Text = (string)e.NewValue;
}
```

and finally, wrapper properties must be defined:

```
public new double Height
{
  get { return (double)GetValue(HeightProperty); }
  set { SetValue(HeightProperty, value); }
}

public string Header
{
  get { return (string)GetValue(HeaderProperty); }
  set { SetValue(HeaderProperty, value); }
}
```

The class has a total of 16 dependency properties and you are encouraged to investigate which ones and how they work. The corresponding code is similar to the above and fills some, and the only difference in principle is that some have default values and that some have no *OnChanged* handler.

The controller also has an event:

```
public delegate
  void ValidateOnFocusLostEventHandler(object sender, ValidateEventArgs
  e);
public event ValidateOnFocusLostEventHandler ValidationOnFocusLost;
```

When the *TextBox* field loses focus (when the text is changed), this event is raised and notifications are sent to any event listeners. The argument is as follows:

```
public class ValidateEventArgs
{
  public AutoControl.ValidationValues ValidationValue { get; set; }
}
```

where the type is:

```
public enum ValidationValues { Valid, Invalid, Unknown }
```

The idea is that the user program can validate the content of the text field, and then color the background accordingly. The colors are set as *Brush* objects that are defined as dependency properties, and the result is that if there is a listener with a control method, then the background color is set correspondingly when the field loses focus.

The class has 3 instance variables:

```
private bool lostFocus = false;
private bool listOpen = false;
private ListBox lst = new ListBox();
```

They all have to do with the list of suggested values. The second last is the list box, and finally the first two are some help variables used to control what happens when the input field loses focus. The list is attached to the window when the controller loads:

```
private void Control_Loaded(object sender, RoutedEventArgs e)
{
  txt.TabIndex = TabIndex;
  lst.VerticalAlignment = VerticalAlignment.Top;
  lst.HorizontalAlignment = HorizontalAlignment.Left;
  lst.Visibility = Visibility.Collapsed;
  lst.Width = txt.Width;
  Pair<Grid, FrameworkElement> pair = FindParentGrid();
  if (pair != null)
  {
    Grid parent = pair.First;
    FrameworkElement child = pair.Second;
    parent.Children.Add(lst);
    if (GridCol == -1) GridCol = Grid.GetColumn(child);
    if (GridRow == -1) GridRow = Grid.GetRow(child);
    SetListMargin(child);
  }
}
```

First, some properties are defined concerning the width and alignment of the list. The control has a limitation as it must sit directly or indirectly in a *Grid* (there must be a *Grid* in the visual tree of the control). The method *FindParentGrid()* is used to find this *Grid* (the innermost if several). Once this *Grid* is found, the *ListBox* component is added to it in the same cell as the control itself. Finally, the method calls *SetListMargin()*, which has the task of measuring where the list should be displayed in relation to the input field.

```
private void SetListMargin(FrameworkElement child)
{
  Point rootPoint;
  if (txt.Equals(child) || !Grid.GetRow(txt).Equals(Grid.GetRow(lst)))
    rootPoint = new Point(0, 0);
  else
  {
    GeneralTransform transform = txt.TransformToAncestor(child);
    rootPoint = transform.Transform(new Point(0, txt.ActualHeight));
  }
  Thickness childMargin = child.Margin;
  lst.Margin = new Thickness(rootPoint.X + childMargin.Left,
                             rootPoint.Y + childMargin.Top + 2, 4, 0);
}
```

The class constructor must primarily associate the necessary event handlers:

```
public AutoControl()
{
  InitializeComponent();
  lst.KeyDown += lst_KeyDown;
  this.Loaded += new RoutedEventHandler(Control_Loaded);
  txt.TextChanged += new TextChangedEventHandler(TextChanged);
  txt.GotFocus += new RoutedEventHandler(txt_GotFocus);
  txt.LostFocus += new RoutedEventHandler(txt_LostFocus);
  txt.PreviewKeyDown += new KeyEventHandler(txt_PreviewKeyDown);
  lst.PreviewMouseDown += new MouseButtonEventHandler(lst_
  PreviewMouseDown);
  lst.MouseDown += new MouseButtonEventHandler(lst_MouseDown);
  IsPreClick = false;
}
```

The rest of the code regarding primarily these event handlers and are generally simple. However, there is a single exception. When you press a key, the list must be filtered for values that no longer need to be included, and that requires some details.

Then there is the main window:

```
<Grid>
  <DockPanel Margin="20">
    <StackPanel DockPanel.Dock="Top" Orientation="Horizontal">
      <local:AutoControl x:Name="kingControl"
        Header="Danish Kings" TextWidth="200"
        FontFamily="Verdana" FontWeight="Bold" Unknown="Yellow"
        Invalid="Tomato"
        Valid="LightGreen" />
      <Button Content="OK" Margin="20,0,0,0" Click="OK_Click"/>
    </StackPanel>
    <ListBox Name="lstKings" Margin="0,20,0,0" />
  </DockPanel>
</Grid>
```

and here you should especially note:

- that it all sits in a *Grid* as needed to apply this custom control and how to set values for its properties
- especially the last three properties which are the colors to be used for the background of the input field for content control

The C# code is:

```
public partial class MainWindow : Window
{
  public MainWindow()
  {
    InitializeComponent();
    InitList();
  }

  private void InitList()
  {
    Kings kings = new Kings();
    foreach (King king in kings) kingControl.Add(king.ToString());
    kingControl.ValidationOnFocusLost += ValidateKing;
  }

  private void OK_Click(object sender, RoutedEventArgs e)
  {
    lstKings.Items.Add(kingControl.Text);
    kingControl.Text = "";
    kingControl.Focus();
  }
```

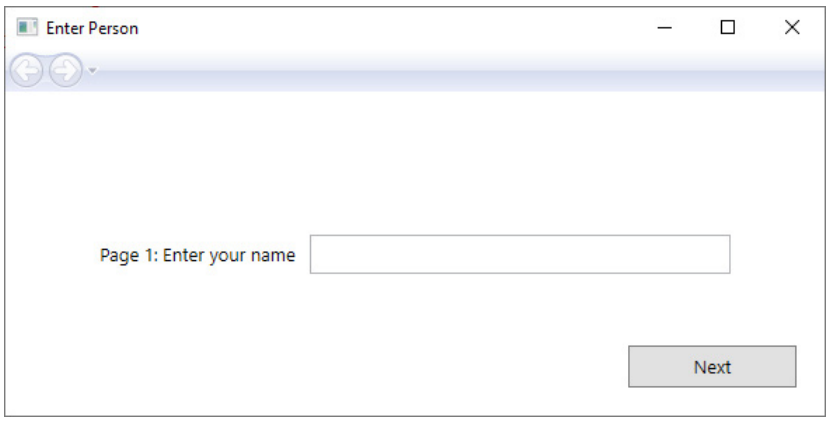
```
  public void ValidateKing(object sender, ValidateEventArgs e)
  {
    e.ValidationValue = Suggestion.ValidationValues.Invalid;
    string text = ((Suggestion)sender).Text;
    string[] elems = text.Split(' ');
    int n = elems.Length - 1;;
    if ((n >= 2) && (ToYear(elems[n - 2]) >= 0 || ToYear(elems[n]) >=
    0))
      e.ValidationValue = Suggestion.ValidationValues.Valid;
  }

  private int ToYear(string text)
  {
    try
    {
      return int.Parse(text);
    }
    catch
    {
      return -1;
    }
  }
}
```

Here, the most important are the last two methods used to check the contents of the field. The field is legal if it ends with at least one year (positive integer).

6 PAGE NAVIGATION

When you open your browser you can navigate back and forth between pages that you have already visited. Many windows applications are also page oriented and for such applications it can sometimes be appropriate with a design so that the program behaves a bit like a browser. A good example is Windows Explorer which works this way. WPF has classes that support the development of such applications. It's quite simple and the following shows how.

The program *NavigationProgram* opens the following window:



It is not a normal *Window* class, but instead a *NavigationWindow* which shows a *Page*. Note that the window show navigations buttons. The application has two other pages, and you go to the next button by clicking *Next*. When you reach the last page you can click a button, and the program shows a message box showing the entered text. On the way you can go back and forth using the navigation buttons.

To create the application I create a usual WPF Application project, but in code behind I change the base class:

```
namespace NavigationProgram
{
  public partial class MainWindow : NavigationWindow
  {
    public MainWindow()
    {
      InitializeComponent();
    }
  }
}
```

such the class now inherits *NavigationWindow*. It is a class that inherits *Window*. This means that I also have to modify the XML:

```
<NavigationWindow x:Class="NavigationProgram.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:NavigationProgram"
  mc:Ignorable="d"
  Title="Enter Person" Height="300" Width="600" FontSize="13"
  Source="Page1.xaml">
</NavigationWindow>
```

The element type is changed to *NavigationWindow*, and the elements has no content. Instead the attribute *Source* has a value for the first *Page* to show.

To create a page you add a new item to your project and this time it must be a *Page* item. It is a class that look like a Window with a XML file and a code behind. The XML for the first page is:

```
<Page x:Class="NavigationProgram.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:mc="http://schemas.openxmlformats.org/markup-
      compatibility/2006"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:local="clr-namespace:NavigationProgram"
      mc:Ignorable="d"
      d:DesignHeight="250" d:DesignWidth="600" Title="Page1"
      FontSize="13">
  <Grid Margin="20">
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <TextBlock Text="Page 1: Enter your name"
      VerticalAlignment="Center"/>
      <TextBox Width="300" Height="28" Margin="10,0,0,0"
        VerticalContentAlignment="Center"/>
    </StackPanel>
    <Button Content="Next" Height="30" Width="120"
    HorizontalAlignment="Right"
      VerticalAlignment="Bottom" Click="Button_Click"/>
  </Grid>
</Page>
```

There are not much to note, but the element type is *Page* and the button references an event handler defined in code behind:

```
public partial class Page1 : Page
{
  public Page1()
  {
    InitializeComponent();
  }

  private void Button_Click(object sender, RoutedEventArgs e)
  {
    NavigationService.Navigate(new Uri("Page2.xaml", UriKind.
    Relative));
  }
}
```

The event handler navigate to the next page which happens using a *NavigationService* that has methods used to navigate between pages. When the method *Navigate()* is performed the current page (that is *Page1*) is inserted in a history so you later can go back to this pages.

The two other pages are almost identical with *Page1* and I will not show the code here, but I will show the code for the event handler for the last page:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("Thank for your information");
  while (NavigationService.CanGoBack) NavigationService.
  RemoveBackEntry();
  NavigationService.Refresh();
}
```

Here I have used other navigation methods and you are encouraged to investigate what other methods the class *NavigationService* provides and in particular what events may occur.

6.1 SENDING DATA BETWEEN PAGES

The above example shows that it is quite simple to write a program with a user interface using pages. There are examples on programs where this design is fine, but a typical use

of page navigation is programming a wizard where the user through a number of steps must initialize one or another action. It is not possible with the above design, as the pages not immediately can use the same data. The solution is to use a *PageFunction* instead of a *Page*. It is a class derived from *Page* but with the possibility to send a parameter to a page when it is opened and to return a value from the page. It is a generic class where the type parameter defined the object the page should work on.

The program *NavigationProgram1* works in exactly the same way as the above program, but the three pages are used to initialize an object of the type:

```
public class Person
{
  public string Name { get; set; }
  public string Mail { get; set; }
  public string Phone { get; set; }
}
```

The first page has the type *Page* as in the first example and the XML is the same, but the event handler in code behind is changed:

```
public partial class Page1 : Page
{
  public Page1()
  {
    InitializeComponent();
  }

  private void Button_Click(object sender, RoutedEventArgs e)
  {
    Person pers = new Person { Name = txtName.Text };
    PageFunction2 func = new PageFunction2(pers);
    func.Return += FunctionReturn;
    this.NavigationService.Navigate(func);
  }
```

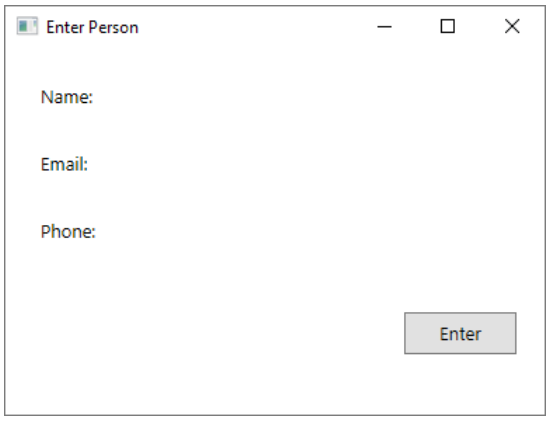
```
  public void FunctionReturn(object sender, ReturnEventArgs<Person> e)
  {
    if (e != null)
    {
      MessageBox.Show(e.Result.Name + "\n" + e.Result.Mail + "\n" +
        e.Result.Phone);
    }
    while (NavigationService.CanGoBack) NavigationService.
    RemoveBackEntry();
    NavigationService.Refresh();
  }
}
```

The handler create a new *Person* object and initialize the *Name* with the value the user must have entered. Next a *PageFunction2* object is created the *Person* object as parameter, and to this object is added an event handler which shows the value of the *Person* object. *PageFunction2* is a *PageFunction* object:

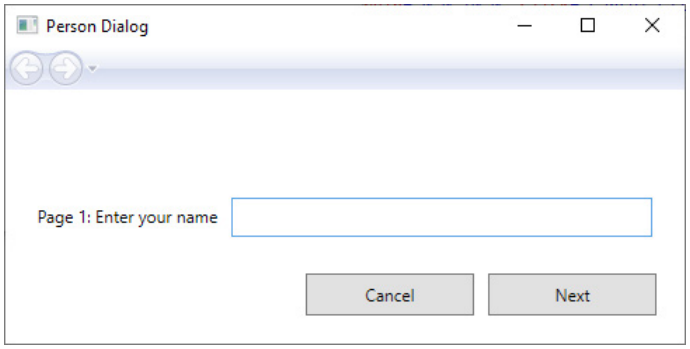
```
public partial class PageFunction2 : PageFunction<Person>
{
  private Person person;

  public PageFunction2(Person person)
  {
    InitializeComponent();
    this.person = person;
  }

  private void Button_Click(object sender, RoutedEventArgs e)
  {
    person.Mail = txtMail.Text;
    PageFunction3 func = new PageFunction3(person);
    func.Return += FunctionReturn;
    this.NavigationService.Navigate(func);
  }

  public void FunctionReturn(object sender, ReturnEventArgs<Person> e)
  {
    if (e != null)
    {
      OnReturn(e);
    }
  }
}
```

The class is is in principle identical with *Page1* but the constructor has a parameter which is a *Person*. The event handler for the button initialize the property *Mail* and forwards the *Person* object to another *PageFunction* object. Note the event handler *FunctionReturn* which this time returns a return value from a *PageFunction3* object to the caller for the object, which is the *Page1* object. The class *PageFunction3* look like the above class and initialize the *Phone* property for the *Person* object and the return this object.

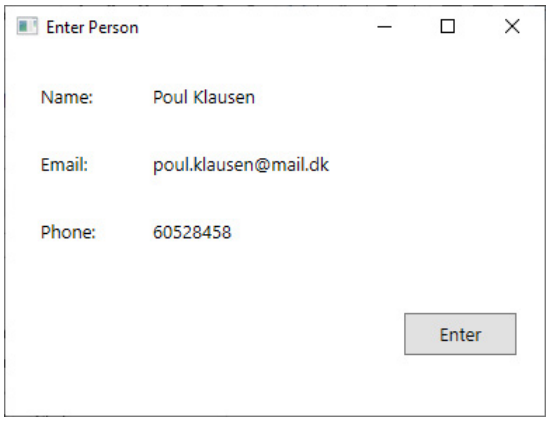
EXERCISE 7: A PERSON WIZARD



In this exercise you must write a program that works in the same way as the above example, but the program must have the above *MainWindow*. There are three labels which from start has no content. When the user click *Enter* the program should open a dialog box as a *NavigationWindow* where the user in three pages should initialize a *Person* object and if all three properties are initialize and the user clicks *OK* the empty labels in the above window should be updated. The first page could be:



and you should note, that this time there is a *Cancel* button. If you enter in all three pages the result of the main window could be:

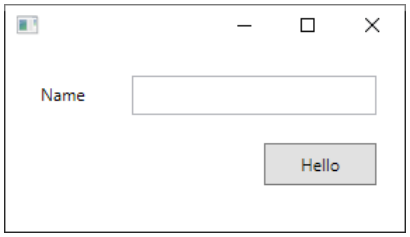


7 A WPF APPLICATION WITHOUT XML

A C # program is basically a class with a static *Main()* method and typically the program uses objects instantiated from other classes. If you look at a WPF program, the class for a window consists of an XML part that defines the user interface and a C# part that defines the program logic. All elements in the XML part are translated into objects of a class and basically the name of an element is the name of a corresponding class. In the end, a WPF program like all other programs is a C# program with a static *Main()* method and the program must therefore also be able to be written alone in C# without the use of XML.

In this chapter I will show how, although there are not many reasons for it, if at all any. If you have a program with an advanced user interface it is both time consuming and extensive to write the under interface in C#, but it is possible. If in any way you should be interested in directly writing the user interface in code, it should be that you have control over which code is written and fully understand how the code works. This is in contrast to the code that the compiler automatically generates from XML, a code that you does not immediately see.

The program *HelloWPF* has two windows. The program opens the following window:



where the user can enter a name. If so (enter then name *Alfred*) and the user clicks *Hello* the program opens the following window which is a modal dialog box:

You should note that the program has two ordinary windows as is windows that it would be very simple to write in XML. In this case I have started with Console Application project called *HelloWPF*. When Visual Studio creates the project it sets references for the most used assemblies (in console applications) and these does not include assemblies for WPF classes. As so the first step is set three references:

1. *PresentationCore*
2. *PresentationFramework*
3. *WindowsBase*

and then the classes for the graphical user interface are available. The main class can then be written as:

```
using System;
using System.Windows;
using System.Windows.Controls;
namespace HelloWPF
{
  public class Program : Window
  {
    private TextBox txtName;
    public Program()
    {
      Width = 300;
      Height = 170;
      Grid grid = new Grid();
      grid.Margin = new Thickness(20);
      grid.RowDefinitions.Add(new RowDefinition { Height = new
      GridLength(48) });
      grid.RowDefinitions.Add(new RowDefinition());
      grid.ColumnDefinitions.Add(new ColumnDefinition {
        Width = new GridLength(70) });
      grid.ColumnDefinitions.Add(new ColumnDefinition());
      grid.Children.Add(new Label { Content = "Name" });
      txtName = new TextBox { VerticalContentAlignment =
      VerticalAlignment.Center,
        Margin = new Thickness(0, 0, 0, 20) };
      Grid.SetColumn(txtName, 1);
      Button cmdHello = new Button { Content = "Hello", Width = 80,
      Height = 30,
        HorizontalAlignment = HorizontalAlignment.Right,
        VerticalAlignment = VerticalAlignment.Top };
      cmdHello.Click += Hello_Click;
      Grid.SetRow(cmdHello, 1);
      Grid.SetColumn(cmdHello, 1);
```

```
      grid.Children.Add(txtName);
      grid.Children.Add(cmdHello);
      Content = grid;
    }

    private void Hello_Click(object sender, RoutedEventArgs e)
    {
      (new HelloWindow(txtName.Text)).ShowDialog();
    }

    [STAThread]
    static void Main()
    {
      Application app = new Application();
      app.Run(new Program());
    }
  }
}
```

When you see the code it is easy to understand what is going on. It is the constructor that creates the window. You should especially note how to initialize an attached property. The most important is the *Main()* method which creates an *Application* object that is an object representing a windows application. The application start with the method *Run()* which as parameter has the object representing the window. The *Main()* method has an attribute *STAThread*. It is necessary and means the program can use the original COM technology used by Windows. The code for the dialog box is basically identical and I will not show it here.

8 COMPONENTS AND GRAPHICS

I will end the treatment of WPF with a few remarks on how WPF draws the window and its components. I will primarily focus on what WPF makes available to the programmer and what options the programmer has for modifying how components are drawn and displayed. In connection with this, I will also discuss animations and how it is simple in WPF to animate geometric objects.

A window is a rectangular area of the screen, and when the window and its components are to be displayed, it means that the window occupying certain pixels and displaying each pixel in a certain color determined by the properties of the window / component, size, position, background, etc. By manipulating the screen's pixels in this way, the runtime system can draw lines, curves, etc. based on the properties of graphic objects and fill areas with a specific color or pattern. In order for WPF to draw a given graphic object, a pen must be attached to the object that tells WPF how lines and other boundaries should be displayed, and a brush must be attached that tells WPF how areas should be completed.

When WPF draws a component, it is said that WPF renders the component. Looking at all the above examples, the programmer has not been involved in that process. This is something WPF has taken care of, corresponding to the fact that everything necessary is built into the classes or base classes of the individual components, and there is rarely any reason to draw geometric shapes yourself, but the possibilities are there and how exactly is the goal of the following. In addition, WPF has three ways to render graphic objects:

1. As *Shape* objects, where graphic objects are derived from the type *Shape* and are found in the namespace *System.Windows.Shapes*. These objects behave in the same way as all other components of the user interface, and they are used to a large extent in the same way as for example the components *Button*, *Label* etc.
2. As *Drawing* objects, which in many ways are alternatives to *Shape* objects, but are instead objects whose types are derived from *System.Windows.Media.Drawing*. These classes are not in the same way components and do not have the same options regarding events such as *Shape* objects. In return, they are simpler and more efficient. They can thus have a positive impact on the performance of the program.
3. As *Visual* objects, there are types derived from *System.Windows.Media.Visual*. These are even more lightweight objects than *Drawing* objects, and they give the programmer optimal degrees of freedom in terms of both graphics and performance. On the other hand, you can only work with these objects in C#, but not in XML, which makes development more difficult.

Finally, there are animations where WPF has the necessary logic built in, so it is easy to define animations for components and other graphical objects. It is the subject for the next chapter.

8.1 SHAPES

A *Shape* object is a component similar to it is derived from *FrameworkElement*, and a *Shape* object therefore has the same properties as other UI components. It is an object that can be immediately placed in a panel using the usual properties, and it is an object that supports events handling as for example concerning the mouse. This means that it is easy to work with graphic objects, as it all takes place as with other components, but conversely, it is also exactly the disadvantage. The goal of graphic objects is often purely visual and e.g. to display a geometric figure, and therefore there is often no need for all the functionality provided by a *FrameworkElement*, and a *Shape* objects thus use many resources that are not needed and that can go beyond the program's performance. This is why WPF has other ways of defining graphic objects. The above remark does not mean that one should avoid using *Shape* objects, but simply that one should be aware of the resource consumption. If, for example you has a window with 100 or more *Shape* objects, one should probably consider another solution.

Perhaps the biggest advantage of *Shape* objects is that they can be defined exclusively in XML. This means that you can immediately see in the designer window how a figure is displayed on the screen, which makes the work somewhat easier. Of course, you can also define *Shape* objects in the code, but it rarely provides benefits, unless you need to dynamically create geometric objects. There are basically 6 Shape objects:

1. *Line*
2. *Rectangle*
3. *Ellipse*
4. *Polygon*
5. *Polyline*
6. *Path*

and except for the last one, the names tell what the objects show. The latter represents a coherent curve.

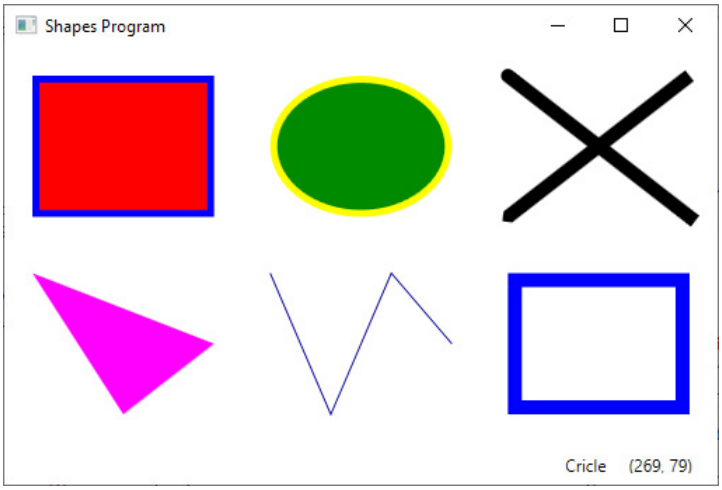
The classes of course have different properties, but the following are important and apply to all *Shape* objects:

1. *DefineGeometry* which returns a data structure that specifies how the component is drawn (which points belong to the figure)
2. *Fill* which indicates the brush used to draw the interior of the figure (figure surface)
3. *GeometryTransfom* which makes it possible to define a transformation that must be performed before the figure is drawn
4. *RenderTransfom* which allows you to define a transformation that must be performed after the figure is drawn
5. *Stretch* which indicates depending on the current layout panel how the figure should fill the space available
6. *Stroke* which indicates the pen with which the circumference / demarcation of the figure is to be drawn
7. *StrokeThickness* which indicates the thickness of the pen to be drawn with
8. *StrokeDashArray* which specifies how lines and dashes should be dotted (default is fully drawn)
9. *StrokeStartLineCap* which specifies the shape that a line should start with
10. *StrokeEndLineCap* which specifies the shape that a line should end with

There is reason to pay special attention to the fact that if you draw a *Shape* object, but do not specify a brush or a pen, then the figure is not visible.

As mentioned, it is easy to work with *Shape* objects, but a documentation of all possibilities is after all extensive. I must therefore confine myself to showing through some examples, some of the possibilities that exist. The rest of this chapter - and by the way pretty much the rest of the book - therefore consists of examples that primarily draw geometric shapes in the window.

The first example shows how to draw shapes in a window as objects whose type is derived from the type *Shape*, which are components in the same way as *Button*, *TextBox* and so on. The example opens a window with six shapes, which are arranged using a *UniformGrid*:



At the bottom there is a *StatusBar*, and if you move the mouse over a shape, the name of the figure and the coordinates of the mouse are displayed.

The XML code for the user interface is quite extensive, but this is mainly due to the fact that three event handlers are defined for each shape:

```
MouseEnter="rect_MouseEnter"
MouseLeave="shape_MouseLeave"
MouseMove="shape_MouseMove"
```

In order for the code not to fill up too much, I have shown these event handlers as three dots:

```
<DockPanel>
  <StatusBar DockPanel.Dock="Bottom" HorizontalAlignment="Right"
    Background="White" Height="30">
    <Label Name="lblShape" />
    <Label Name="lblPoint" Margin="0,0,10,0"/>
  </StatusBar>
  <UniformGrid Name="grid" Rows="2" Columns="3">
    <Rectangle Name="rect" Fill="Red" Stroke="Blue" StrokeThickness="5"
    Margin="20"
      ... />
    <Ellipse Name="circle" Fill="Green" Stroke="Yellow"
    StrokeThickness="5"
      Margin="20" ... />
    <Canvas Name="cross" Margin="20" Background="White" ... >
      <Line Name="line1" StrokeStartLineCap="Round"
      StrokeEndLineCap="Square"
```

```
        X1="0" Y1="0" X2="{Binding ElementName=cross,
        Path=ActualWidth}"
        Y2="{Binding ElementName=cross, Path=ActualHeight}"
        Stroke="Black"
        StrokeThickness="10" MouseMove="line_MouseMove" />
      <Line Name="line2" StrokeStartLineCap="Triangle"
      StrokeEndLineCap="Flat"
        X1="0" Y1="{Binding ElementName=cross, Path=ActualHeight}"
        X2="{Binding ElementName=cross, Path=ActualWidth}" Y2="0"
        Stroke="Black"
        StrokeThickness="10" MouseMove="line_MouseMove" />
      </Canvas>
      <Polygon Name="poly" Points="20 20 150 130 100 30" Fill="Magenta"
      ... />
      <Polyline Name="pline" Stroke="DarkBlue" Points="20 20 80 130 120
      40 150 60"
        ... />
      <Rectangle Stroke="Blue" StrokeThickness="10" Margin="20" ... />
  </UniformGrid>
</DockPanel>
```

The layout is simple and consists of a *DockPanel*, which at the bottom has a status bar and otherwise a *UniformGrid* with two rows and three columns. The status bar contains two *Label* components that event handlers can refer to and update according to the position of the mouse. The grid has the following components:

1. A *Rectangle*, which is filled with a red color (brush) and has a blue border drawn with a pen of 5 pixels. The object processes the three mouse events that I have mentioned above.
2. An *Ellipse* that is filled with a green color and has a yellow border on 5 pixels. The object processes the three mouse events.
3. A Canvas that also deals with the three events. This Canvas is assigned a background color. Otherwise the background would be blank and would not capture events from the mouse. The *Canvas* object encapsulates two Line objects:
   - The first *Line* object goes from the upper left corner to the lower right corner and is drawn with a black pen of 10 pixels. The coordinates for the lower right corner are tied to the current size of the *Canvas* object, so that the line automatically adjusts to the size of the window.
   - The second *Line* object goes from the lower left corner to the upper right corner and is drawn in the same way with a black pen of 10 pixels. Two of the coordinates are bound to the current size of the *Canvas* object.

The two *Line* objects refer to their own *MouseMove* event handler. This means that when the mouse is moved above the *Canvas* object, the same event handler is executed as for the other figures, but when the mouse then hits one of the two *Line* objects, it is another. Also note how to specify how a line should end (*Square*, *Round*, *Flat* and *Triangle*)

4. A *Polygon*, which in this case is a triangle defined by three corners. The polygon has no edge, but it is filled with a violet color.
5. A *Polyline* defined by four dots drawn with a blue line of 1 pix.
6. A *Rectangle* drawn with a blue pen of 20 pixels. This time there is no *Fill* color and you have to note that the inside of the rectangle does not catch the mouse - it is empty.

Note that for the two *Rectangle* objects and the *Ellipse* object, no size is specified. You can, of course, but in this case the size is determined by the size of the current grid cell. Next, if you look at the *Polygon* and *Polyline* objects, they have fixed sizes because of the coordinates. However, they are not used as they are set dynamically in the code, but they are included here partly to define the number of points and partly to visualize the objects under the design.

Then there is the code, which takes up a lot of space due to the many event handlers, but is otherwise simple. I will not show these event handlers here, but I will show how to dynamically adjust the size of the *Polygon* and *Polyline* object. In order to do so, the window must capture two events regarding the activation of the window, and if the size of the window changes:

```
<Window x:Class="Exam100.MainWindow" ...
        Activated="Window_Activated" SizeChanged="Window_SizeChanged">
```

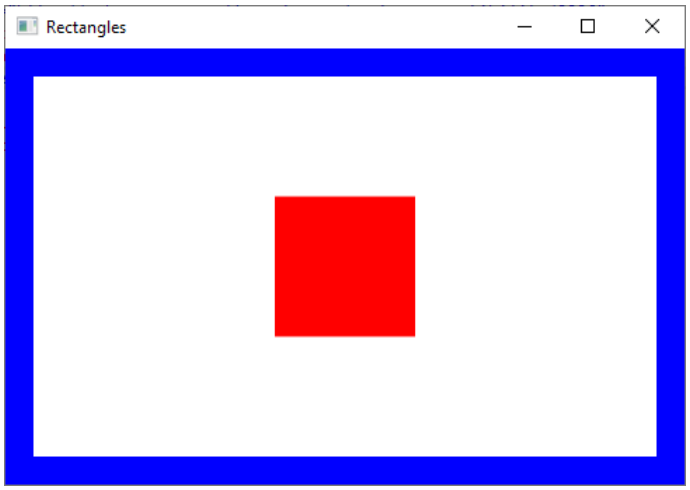
Event handlers for these events calls the following method:

```
private void Resize()
{
  double width = grid.ActualWidth / 3 - 40;
  double height = grid.ActualHeight / 2 - 40;
  pline.Points[1] = new Point { X = 20 + width / 3, Y = 20 + height };
  pline.Points[2] = new Point { X = 20 + 2 * width / 3, Y = 20 };
  pline.Points[3] = new Point { X = 20 + width, Y = 20 + height / 2 };
  poly.Points[1] = new Point { X = 20 + width / 2, Y = 20 + height };
  poly.Points[2] = new Point { X = 20 + width, Y = 20 + height / 2 };
}
```

The method determines the size of a cell in the grid and then sets the coordinates of the *Polygon* and *Polyline* objects to fit the current size. This method is called when the window is activated and when its size changes, and the result is that the size of the objects is adjusted to the window.

EXERCISE 8: RECTANGLES

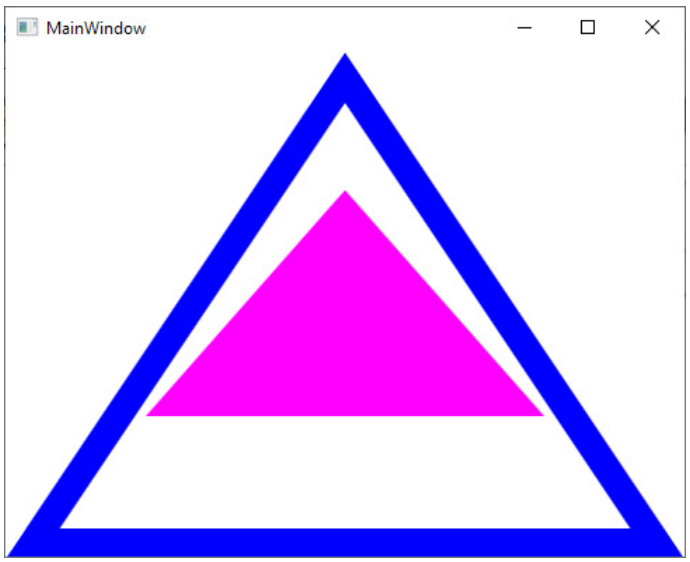
Write a program which opens the following window:



where the red square must be shown in the center of the window and the blue rectangle must follow the edge of the window when the window size is changed. Note that you do not need to write any C# code.

EXERCISE 9: TRIANGLES

Write a program which opens the below window when the vertices of both triangles must follow the size of the window. If you have problems with the corners of the blue triangle if the corners get very pointed just ignore the problem. It requires a little trigonometry to solve the problem precisely.



8.2 PEN AND BRUSH

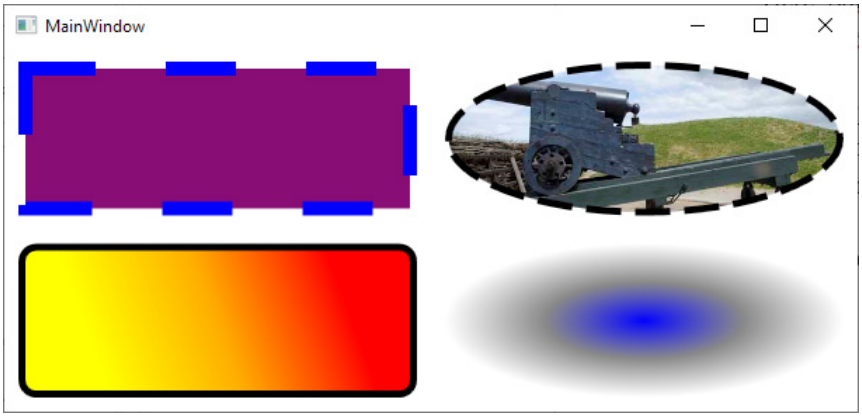
Pen and Brush are the two most important tools for drawing figures in a window, no matter what kind of figure it is.

A brush is an object that determines how a figure is filled in, and there are generally the following types:

1. *SolidColorBrush*, which fills the shape with a color.
2. *ImageBrush*, which fills the figure with an image.
3. *LinearGradientBrush*, which fills the figure with a linear gradient.
4. *RadialGradientBrush*, which fills the figure with a radial gradient.
5. *DrawingBrush*, which fills the shape with a *Drawing* object.
6. *VisualBrush*, which fills the figure with a *Visual* object.

Here, *SolidColorBrush* and *ImageBrush* are simple to use and create, while *LinearGradientBrush* and *RadialGradientBrush* require a little more, and the last two require a lot more.

As for a pen, there is not much to say, but it is used to draw an edge around a figure or possibly just a line or other curve. You can specify the color and line thickness as well as the endpoints - the endings (see example above), and finally you can specify whether the line should be dotted.



The following program opens a window with two rectangles and two ellipses (see above). Clicking on the top right rectangle fills it with a random color. The layout is a *UniformGrid* with 2 rows and 2 columns, and each of the four cells contains a shape:

```
<UniformGrid Rows="2" Columns="2">
  <Rectangle Name="rect" Margin="10" Fill="White" StrokeThickness="10"
    Stroke="Blue" StrokeDashArray="5" MouseUp="rect_MouseUp" />
  <Ellipse Margin="10" StrokeThickness="5" Stroke="Black"
  StrokeDashArray="5 2" >
    <Ellipse.Fill>
      <ImageBrush ImageSource="Kanon.jpg" Stretch="UniformToFill" />
    </Ellipse.Fill>
  </Ellipse>
  <Rectangle Margin="10" StrokeThickness="5" Stroke="Black"
  RadiusX="10"
    RadiusY="10">
    <Rectangle.Fill>
      <LinearGradientBrush StartPoint="0 1" EndPoint="1 0">
        <GradientStop Color="Yellow" Offset="0.2"></GradientStop>
        <GradientStop Color="Orange" Offset="0.5"></GradientStop>
        <GradientStop Color="Red" Offset="0.8"></GradientStop>
      </LinearGradientBrush>
    </Rectangle.Fill>
  </Rectangle>
  <Ellipse Margin="10" >
    <Ellipse.Fill>
      <RadialGradientBrush>
        <GradientStop Color="Blue" Offset="0"></GradientStop>
        <GradientStop Color="Gray" Offset="0.5"></GradientStop>
        <GradientStop Color="White" Offset="1"></GradientStop>
      </RadialGradientBrush>
    </Ellipse.Fill>
  </Ellipse>
</UniformGrid>
```

In addition, the project has attached an image in the form of a local resource. The upper right rectangle has a blue border of 10 pixels. With

```
StrokeDashArray="5"
```

you indicates that the line around the rectangle must be dotted. The number 5 indicates that the line is drawn alternately as a blue line segment and a blank line segment, all of which are equal in length and have a length 5 times the line thickness and in this case thus 50. You should note that a line segment continues if you come to a corner:

You should also note that it is the center of the line that delimits the rectangle. In this case, the thickness is 10, which means that a 6 pixel blue border is drawn around the rectangle, while the 4 pix occupies part of the rectangle itself.

The rectangle has an event handler that associates a *SolidColorBrush* with a random color:

```
private void rect_MouseUp(object sender, MouseButtonEventArgs e)
{
  rect.Fill = new SolidColorBrush(Color.FromRgb((byte)rand.Next(256),
                  (byte)rand.Next(256), (byte)rand.Next(256)));
}
```

The upper right ellipse is filled in by an *ImageBrush*, where the image is attached to the project as a resource. There is not much to explain, but note that the ellipse has a dotted black border defined as follows:

```
StrokeDashArray="5 2"
```

Here, the number 5 means that there is first a black piece is 5 times the width (that is 25) and then a blank piece twice the thickness of the line (that is 10). Note that you can specify more complex patterns to dash a line.

The lower left rectangle uses a *LinearGradientBrush*. The principle is that two or more colors are mixed along a straight line. In this case, the line runs as a diagonal from the lower left corner to the upper right corner. In this case, there are three color. The color starts with yellow (in the lower left corner), but from 20% of the diagonal, the yellow color mixes with an orange color until you have reached halfway where the color is orange. Then the color is mixed with a red color until you reach 80% along the diagonal. The last part is red.

Also note that the rectangle has rounded corners and how to define it by two radii.

The lower right ellipse is colored with a *RadialGradientBrush*, which in the same way as a *LinearGradientBrush* mixes two or more colors, but this time it happens from the center of the ellipse and towards the periphery. This time there are also three colors. The color starts with blue in the center and gradually changes to gray until you have reached halfway to the periphery. Then mix the color with white until you reach the periphery, where the color is white. Note that this ellipse has no edge.

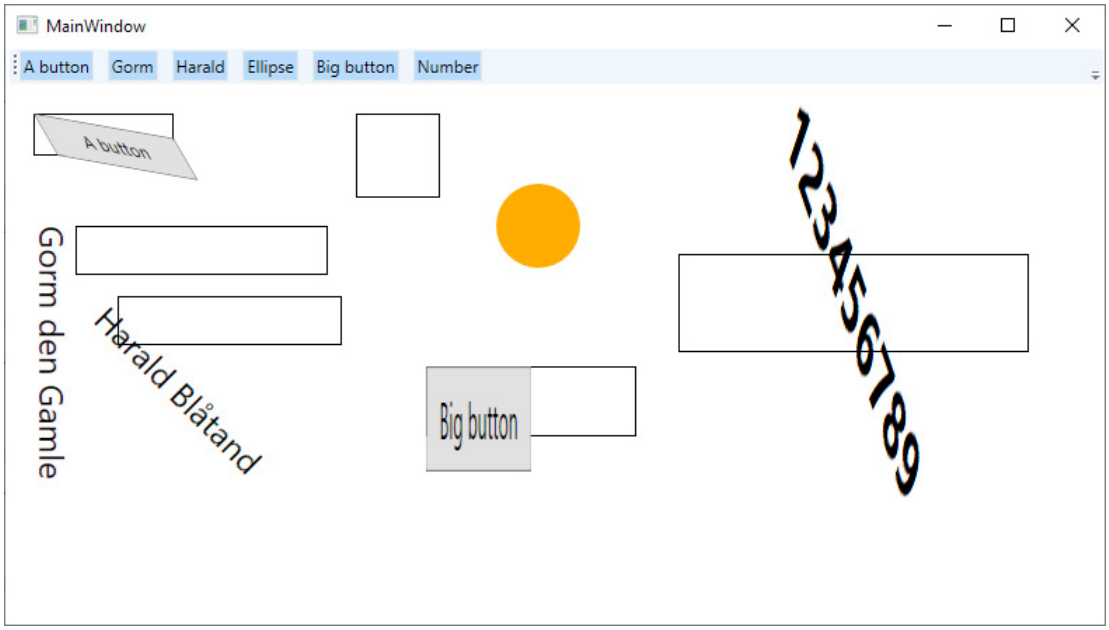
A gradient brush can have all the colors and stops that you may want, and as the example shows, it can be a little difficult to define a brush with many color stops and achieve exactly the effect that you may be interested in. Visual study has in fact, a built-in brush editor that one can use to create a brush. If you select a figure in XML and go under properties, you will find a tool under *Brush*, which can be used to edit a brush.

8.3 TRANSFORMATIONS

In this context, a transformation is an operation that modifies how a component is displayed. Basically, there are the following transformations:

1. *TranslateTransform*, which moves (displaces) a component in the (x, y) coordinate system.
2. *RotateTransform*, which rotates a component an angle
3. *ScaleTransform*, which scales an object along either the x or y axis or both
4. *SkrewTransform*, which rotates a component about one or both axes
5. *MatrixTransform*, where you define an arbitrary transformation using a matrix
6. *TransformGroup*, which consists of a collection of transformation objects

The following program shows the different transformations (except *MatrixTransform*). If you run the program, you get the window:



which contains a *ToolBar* with six *CheckBox* components (all checked) and a *Grid* with two buttons, three texts and an ellipse, all transformed. For each of the six components, a rectangle is also drawn, which shows the location of the component if it had not been transformed. The effect of the toolbar is that you can turn the transformation of the individual components on and off, and the purpose is partly to show what a transformation means, but also how to define a transformation in the C# code.

Attached to the two buttons is an event handler, which does nothing but display a simple message box. The purpose is to show that the buttons - and thus also the area that you click on - also work after they have been transformed.

The layout is simple and consists of a *DockPanel* and a *Grid*. The transformations are defined in both XML and C# code. The window is defined as follows and this time is quite extensive. I have therefore not included the entire code and omitted the attributes that are not directly related to transformations.

```
<DockPanel>
  <ToolBar DockPanel.Dock="Top" >
    <CheckBox Name="check1" Content="A button" ... Click="check1_Click"
    />
    <CheckBox Name="check2" Content="Gorm" ... Click="check2_Click" />
    <CheckBox Name="check3" Content="Harald" ... Click="check3_Click" />
    <CheckBox Name="check4" Content="Ellipse" ... Click="check4_Click" />
    <CheckBox Name="check5" Content="Big button" ... Click="check5_
    Click" />
    <CheckBox Name="check6" Content="Number" ... Click="check6_Click" />
  </ToolBar>
  <Grid>
    <Rectangle ... />
    <Button Name="cmd1" ... Content="A button" Click="Button_Click">
      <Button.RenderTransform>
        <SkewTransform AngleX="30" AngleY="10" />
      </Button.RenderTransform>
    </Button>
    <Rectangle ... />
    <TextBlock Name="txt1" ... Text="Gorm den Gamle">
      <TextBlock.RenderTransform>
        <RotateTransform Angle="90" />
      </TextBlock.RenderTransform>
    </TextBlock>
    <Rectangle ... />
    <TextBlock Name="txt2" ... Text="Harald Blåtand">
      <TextBlock.RenderTransform>
        <RotateTransform Angle="45" />
      </TextBlock.RenderTransform>
    </TextBlock>
    <Rectangle ... />
    <Ellipse Name="elip" ... Fill="Orange" >
      <Ellipse.RenderTransform>
        <TranslateTransform X="100" Y="50" />
      </Ellipse.RenderTransform>
    </Ellipse>
    <Rectangle ... />
    <Button Name="cmd2" ... Content="Big button" Click="Button_Click">
      <Button.RenderTransform>
        <ScaleTransform ScaleX="0.5" ScaleY="1.5" />
      </Button.RenderTransform>
```

```
    </Button>
    <Rectangle ... />
    <TextBlock Name="txt3" ... Text="123456789"
     RenderTransformOrigin="0.5 0.5" >
      <TextBlock.RenderTransform>
        <TransformGroup>
          <RotateTransform Angle="45" />
          <ScaleTransform ScaleX="0.5" ScaleY="1.5" />
        </TransformGroup>
      </TextBlock.RenderTransform>
    </TextBlock>
  </Grid>
</DockPanel>
```

The toolbar contains 6 *CheckBox* components, each with its own event handler. I discuss these event handlers below.

The top button is transformed with a *SkrewTransform*, which twists a component at an angle horizontally, vertically or both ways. The angles are given in degrees. You should notice that you can still click the button after it is twisted, and the area that is clicked corresponds to what is displayed on the screen. Also note how to define a transformation as a value for *RenderTransform*.

The next component is a *TextBlock*, and it is transformed with a *RotateTransform*, which rotates a component a number of degrees. It is in fact the transformation whose effect may be the most difficult to predict, and if one does not think about it well, the result may be different than expected - especially if the rotation is combined with other transformations. Unless otherwise stated, the figure is rotated around the upper left corner (of the circumscribing rectangle), but you can specify the point on which to rotate.

Next follows another *TextBlock* component, which is also transformed with a rotation, and the only difference is that the angle is different.

The circle - an *Ellipse* component - is transformed with a *TranslateTransform*, which move the figure to another position. You enter an (x, y) coordinate set, which indicates the position to which the upper left corner of the figure (in this case the circumscribing rectangle) is to be moved.

The lower button shows the effect of a *ScaleTransform*, where the dimensions of a figure are scaled horizontally, vertically or in both directions. Also in this case, note that one can still click the button after it is scaled.

The last transformation is the most complex, as it is a composed transformation. This is done with a *TransformGroup*, which can contain a number of transformations, all of which are then performed. In this case, the transformation is composed of a rotation and a scaling. Note that a different transformation point is defined this time:

```
RenderTransformOrigin="0.5 0.5"
```

which means that the transformation is determined from the center of the figure. In principle, it is easy enough to define composite transformations with a *TransformGroup*, but one must be aware that the sequence means something. If you do not, the effect may be different than you thought.

Defining transformations in XML is easy - especially because you can immediately see the result in the designer window. Of course, you can also define transformations in C#, and that is the meaning of the six checkboxes at the top of the window. Each checkbox corresponds to one of the six transformations. As an example, the event handler for the first checkbox is shown below:

```
private void check1_Click(object sender, RoutedEventArgs e)
{
  if (check1.IsChecked == true) cmd1.RenderTransform = new
  SkewTransform(30, 10);
  else cmd1.RenderTransform = null;
}
```

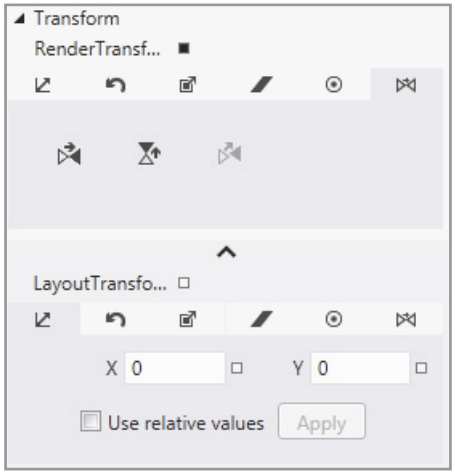
and there is not much to notice, perhaps just apart from undoing a transformation by setting *RenderTransform* to *null*.

As another example, here is the event handler for the last checkbox:

```
private void check6_Click(object sender, RoutedEventArgs e)
{
  if (check6.IsChecked == true)
  {
    TransformGroup group = new TransformGroup();
    group.Children.Add(new RotateTransform(45));
    group.Children.Add(new ScaleTransform(0.5, 1.5));
    txt3.RenderTransform = group;
  }
  else txt3.RenderTransform = null;
}
```

It is, of course, the most complex, but if one looks at the code, there is hardly much surprising.

Relating to transformations, it is also worth mentioning that Visual Studio has a built-in editor that can help define transformations and in that context, what properties it is that you can set. If you select one of the above figures and go under properties, you get the following options under the item *Transform*:



If you examine a component's properties (or the image above regarding a component's *Transform* properties), you can see that there are two possibilities for associating a transformation:

1. *RenderTransform*
2. *LayoutTransform*

In many cases, they will give the same result, but the difference has something to do with how the runtime system draws shapes. The process basically consists of three steps:

1. measure, where the system measures / calculates how much the components take up and thus how much space they need in the window
2. arrange, where the system calculates the location of the components in relation to the specific layout panel
3. render, where systems draw the components

The two transformations are performed at different times in this process:

1. LayoutTransform
2. Measure
3. Arrange
4. RenderTransform
5. Render

That is a transformation associated with a component's *LayoutTransform* will affect the subsequent measure and arrange process, while a transformation associated with a component's *RenderTransform* will only affect the render process.

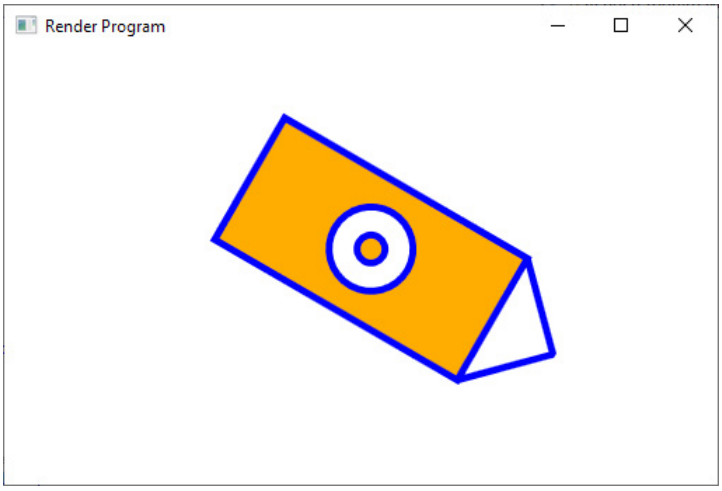
8.4 A PATH

Above I have treated the basic types for geometric shapes: *Line*, *Rectangle*, *Ellipse*, *Polygon* and *Polyline*, all of which are derived from the type *Shape*. However, I am missing the type *Path*, which is really just a collection of other shapes that can be treated as one unified figure, and a *Path* is used to define complex 2D shapes. A *Path* consists of independent geometric shapes, which must be derived from the type *Geometry*, and there are the following finished classes:

1. *LineGeometry*, which represents a straight line
2. *RectangleGeometry*, representing a rectangle
3. *EllipseGeometry*, representing an ellipse
4. *GeometryGroup*, which represents a collection of *Geometry* objects
5. *CombinedGeometry*, which can merge two different *Geometry* objects into a single shape
6. *PathGeometry*, which represents a figure composed of lines and curves

As the names suggest, the *Geometry* objects correspond to similar *Shape* objects, but they are much simpler and require far fewer resources and are thus correspondingly more efficient.

The next example called *RenderProgram* shows how to define a *Path* and the program opens the window below:



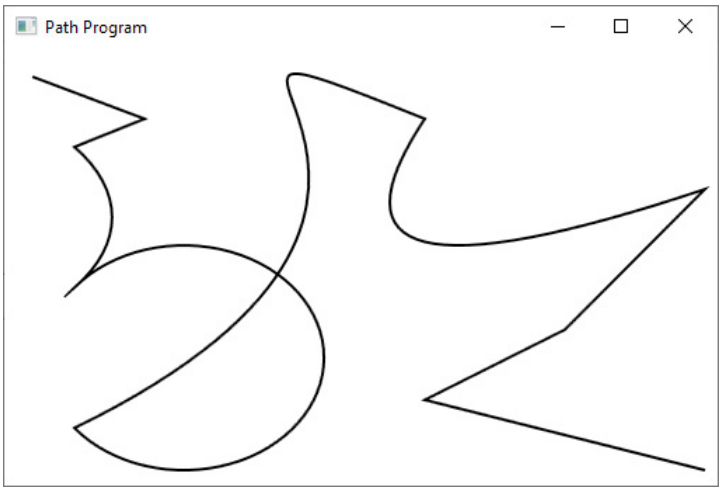
You should especially note that you can click on the figure (you get a message box) and where the figure catches the mouse - you can click on everything orange and everything that is blue, but not something white that is not part of the figure.

The figure above is a *Path*, which consists of a *GeometryGroup* with a *RectangleGeometry* object, two *EllipseGeometry* objects and two *LineGeometry* objects. After the figure is defined, a transformation consisting of a rotation and a translation is associated:

```
<Grid>
  <Path HorizontalAlignment="Left" VerticalAlignment="Top"
   Margin="0,0,0,0"
    Stroke="Blue" StrokeThickness="5" StrokeEndLineCap="Triangle"
    Fill="Orange"
    MouseUp="Path_MouseUp">
    <Path.Data>
      <GeometryGroup>
        <RectangleGeometry Rect="0,0,200,100"/>
        <EllipseGeometry Center="100,50" RadiusX="30" RadiusY="30" />
        <EllipseGeometry Center="100,50" RadiusX="10" RadiusY="10" />
        <LineGeometry StartPoint="200,0" EndPoint="250,50" />
        <LineGeometry StartPoint="200,100" EndPoint="250,50" />
      </GeometryGroup>
    </Path.Data>
    <Path.RenderTransform>
      <TransformGroup>
        <RotateTransform Angle="30" />
        <TranslateTransform X="200" Y="50" />
      </TransformGroup>
    </Path.RenderTransform>
  </Path>
</Grid>
```

It is simple enough to define a figure that way, and the only thing that is not obvious is what happens when figures overlap. If a figure overlaps another figure, the figure at the top will be excluded from the finished Path. In this case you have a *RectangleGeometry* and above it an *EllipseGeometry* object. The result is that the area of the rectangle that the ellipse covers is excluded from the overall figure (therefore the ellipse is white). The innermost (smallest) ellipse overlaps the first, and since it is not part of the figure, the last ellipse becomes one of the figure.

As mentioned above, a *PathGeometry* represents a figure composed of lines and curves. If you run the program *PathProgram*, you get the following window:

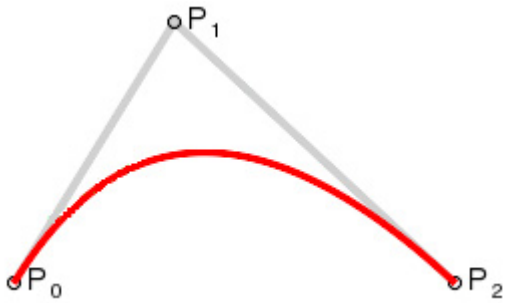


It looks complex, and so is it somewhere. The figure is composed of 7 so-called segments, and there can be any number. The following segments are available:

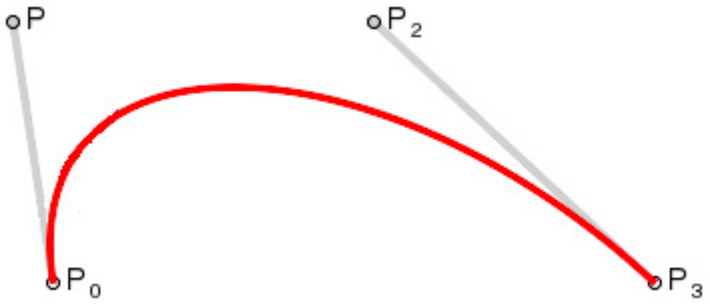
1. *LineSegment*, which represents a line segment
2. *ArcSegment*, which represents an arc between two points
3. *QuadraticBezierSegment*, which represents a curve between two points controlled by a single control point
4. *BezierSegment*, which represents a curve between two points controlled by two control points
5. *PolyLineSegment*, which represents a polyline (a curve composed of a number of line segments)
6. *PolyQuadraticBezierSegment*, which represents a curve composed of a number of quadratic bezier segments
7. *PolyBezierSegment*, which represents a curve composed of a number of Bezier segments

In the figure above, the first 5 segment types occurs.

I will not go into a more detailed description of Bezier curves here, but instead refer to other sources on the subject, and there is plenty of information on the subject on the Internet. In short, a quadratic Bezier curve is a curve between two points and then a single control point, and such that the straight lines through the endpoints and the control point are tangents to the curve:



A cubic Bezier curve or just a Bezier curve is constructed in the same way, but this time there are only two control points:



Both of the above curves can be expressed using formulas, and a program can thus easily determine the points of the curve.

It is a bit cumbersome to define a *PathGeometry* manually, but the principle is that you specify a starting point, and then the segments are drawn as a continuous curve. In this case, a curve is drawn with a large pen of 2 pixels, which starts at the point (20, 20). As it turns out, you has to nest many definitions before reaching the segments of the curve:

```
<Path Stroke="Black" StrokeThickness="2">
  <Path.Data>
    <PathGeometry>
      <PathGeometry.Figures>
        <PathFigure StartPoint="20,20">
          <PathFigure.Segments>
            <LineSegment Point="100,50" />
            <LineSegment Point="50,70" />
            <ArcSegment Point="50,170" Size="200,100"
             SweepDirection="Clockwise" />
            <ArcSegment Point="50,270" Size="100,80"
             SweepDirection="Clockwise"
              IsLargeArc="True" />
            <BezierSegment Point1="400,100" Point2="50,-50"
             Point3="300,50" />
            <QuadraticBezierSegment Point1="200,200" Point2="500,100"/>
            <PolyLineSegment Points="400,200 300,250 500,300" />
          </PathFigure.Segments>
        </PathFigure>
      </PathGeometry.Figures>
    </PathGeometry>
  </Path.Data>
</Path>
```

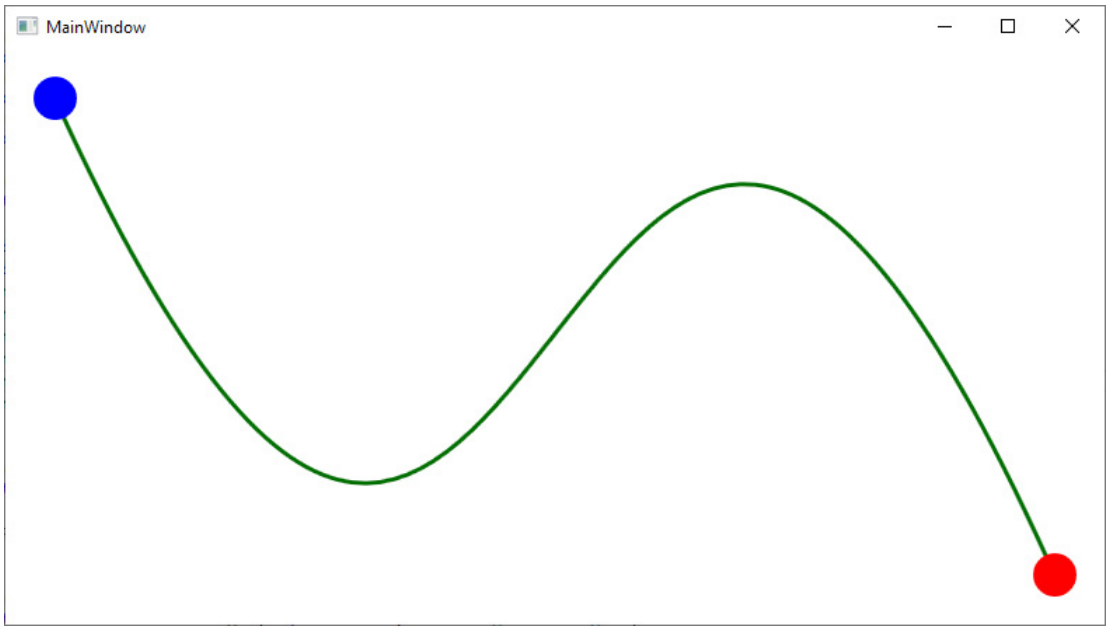
Defining a *LineSegment* is simple, as you only need to specify an endpoint. The result is a line segment from the current point (either the start point of the curve or the end point from the previous segment) to that end point. The first *LineSegment* will therefore draw a line segment from (20, 20) to (100, 50), while the second draws a line segment from (100, 50) to (50, 70).

An *ArcSegment* is an arc from the current point to an endpoint. You also specify a *Size*, which is the size of an ellipse through the two points. One can finally specify the orientation (whether it should be the arc on the one side or on the other side of the line through the two points).

Next follows a *BezierSegment* and a *QuadraticBezierSegment*, which are defined with respectively 3 or 2 points (the first point is the starting point and is implicitly given). Finally, there is a *PolyLineSegment*.

EXERCISE 10: BEZIERPROGRAM

Write a program which you can call *BezierProgram*. The program must open the following window:



When you resize the window the lower right circle and the bezier curve must follow the window size.

The program can be written in several ways, but one option is to add the following class:

```
public class PointViewModel : ViewModelBase
{
  private double x;
  private double y;

  public PointViewModel()
  {
  }

  public PointViewModel(double x, double y)
  {
    X = x;
    Y = y;
  }
```

```
  public double X
  {
    get { return x; }
    set
    {
      if (x != value)
      {
        x = value;
        OnPropertyChanged("X");
        OnPropertyChanged("P");
      }
    }
  }

  public double Y
  {
    get { return y; }
    set
    {
      if (y != value)
      {
        y = value;
        OnPropertyChanged("Y");
        OnPropertyChanged("P");
      }
    }
  }

  public Point P { get { return new Point(X, Y); } }
}
```

In *MainWindow* you can then define three *PointViewModel* objects and bind the bezier segment to these points.

9 DRAWINGS

As the next point, I will look at how to work with objects of the type *Drawing*, which, as mentioned, use fewer resources and thus can have an impact on the efficiency of the program. The starting point is *Geometry* objects that I have mentioned above, and in addition there are the following *Drawing* objects:

1. *GeometryDrawing*, which defines a geometric figure
2. *ImageDrawing*, which allows you to draw an image
3. *DrawingGroup*, which combines a collection of *Drawing* objects into a single figure

Once you have defined a *Drawing* object, it should appear on the screen, which often happens with a *DrawingBrush* or a *DrawingImage*.

If you run the program *DrawingProgram*, you will get the following window:

The background is colored with a *RadialGradiantBrush*, and the button has been given a different visual representation. You can still click on the button, and if you do, you will get a message box. The example should show how both the background and the button are defined with a *Drawing* object.

A *Drawing* object is typically defined as a *GeometryDrawing* object. For such an object, one must (usually) define three things:

1. *Geometry*, which, as the name suggests, defines the geometry of the figure, and thus what kind of figure it is.
2. *Brush*, which indicates how the surfaces of the figure are to be filled
3. *Pen* that indicates how to draw lines

Below, two *Drawing* objects are defined. The first defines a geometry, which is a rectangle filled with a *RadialGradientBrush*. This time a pen is not defined as no edge needs to be drawn. The object is used as the *Source* for an *Image* component, and since it sits in a grid, it will fill the entire window:

```
<Image Stretch="Fill">
  <Image.Source>
  <DrawingImage>
    <DrawingImage.Drawing>
      <GeometryDrawing>
        <GeometryDrawing.Geometry>
          <RectangleGeometry Rect="0,0 20,20" />
        </GeometryDrawing.Geometry>
        <GeometryDrawing.Brush>
          <RadialGradientBrush>
            <GradientStop Color="Yellow" Offset="0"/>
            <GradientStop Color="Orange" Offset="0.5"/>
            <GradientStop Color="Red" Offset="1"/>
          </RadialGradientBrush>
        </GeometryDrawing.Brush>
        </GeometryDrawing>
      </DrawingImage.Drawing>
    </DrawingImage>
  </Image.Source>
</Image>
```

The next *Drawing* object is used as the background for a button. The *Geometry* is this time defined as a *GeometryGroup* consisting of a rectangle and an ellipse. The figure is filled in with a *LinearGradientBrush*, and the edge is drawn with a 5 pixel pen, which is colored with another *LinearGradientBrush*.

```
<Button Name="cmdButton" Content="Big button" Width="200" Height="50"
FontSize="24"
  FontWeight="Bold" Click="cmdButton_Click" >
  <Button.Background>
    <DrawingBrush Stretch="None">
      <DrawingBrush.Drawing>
        <GeometryDrawing>
          <GeometryDrawing.Geometry>
            <GeometryGroup>
              <RectangleGeometry Rect="0,0 200,50"/>
              <EllipseGeometry RadiusX="20" RadiusY="20"
              Center="100,25"/>
            </GeometryGroup>
          </GeometryDrawing.Geometry>
          <GeometryDrawing.Brush>
            <LinearGradientBrush>
              <GradientStop Color="Green" Offset="0.2"/>
              <GradientStop Color="Yellow" Offset="0.8"/>
            </LinearGradientBrush>
          </GeometryDrawing.Brush>
          <GeometryDrawing.Pen>
            <Pen Thickness="5">
              <Pen.Brush>
                <LinearGradientBrush>
                  <GradientStop Color="Black" Offset="0.45"/>
                  <GradientStop Color="LightGray" Offset="0.55"/>
                </LinearGradientBrush>
              </Pen.Brush>
            </Pen>
          </GeometryDrawing.Pen>
        </GeometryDrawing>
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Button.Background>
</Button>
```

If you look at the button, you set its *Background* property to a *DrawingBrush*, which is a brush defined by a *Drawing* object, which here is a *GeometryDrawing*. Looking at the *Image* component, its *Source* property is set to a *DrawingImage*, which is also associated with a *Drawing* object.

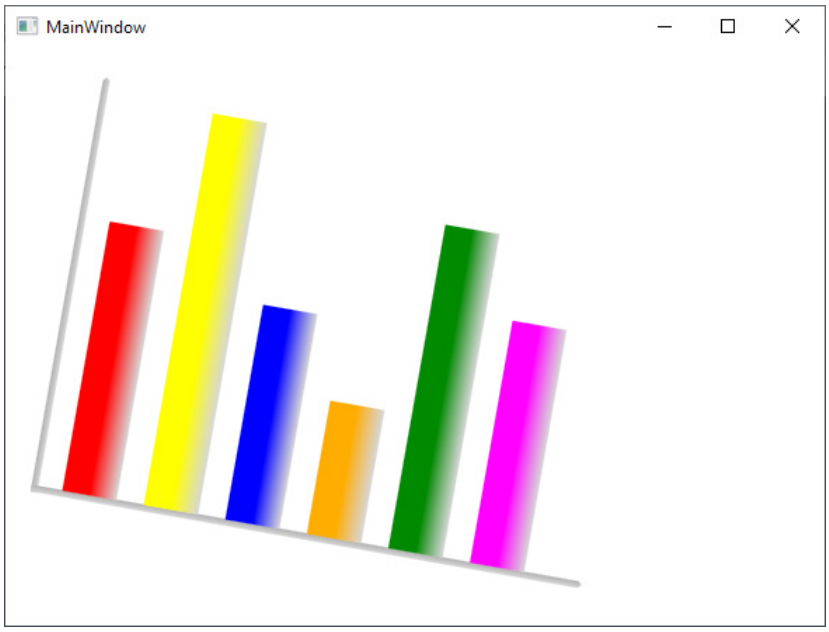
Clicking on the button opens a message box, and at the same time the button gets focus. The button is then drawn in a special way, which shows that it has focus. Since the window has only this one component, the button continues to have focus, and it is therefore necessary to manually remove focus:

```
private void cmdButton_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show("Big button");
  Keyboard.ClearFocus();
}
```

You should note how to remove focus from the code.

9.1 DRAWING A CHART

As the next example I will show how to draw a chart:



In principle, the program does not show much new, but it should show how to create a *Drawing* object as a *DrawingGroup*, which is then transformed. The window contains a single *Image* component whose *Source* is a *DrawingImage*. This time the code fills a lot and I have included only the most important. The source of the image object is a *DrawingGroup*, which consists of

1. 2 *LineGeometry* objects for the axes
2. 6 *RectangleGeometry* objects for the columns

The definition of the two line objects is in principle similar and takes up a lot of space, but this is primarily due to the line being colored with a gradient brush. I have only shown the definition of the first object.

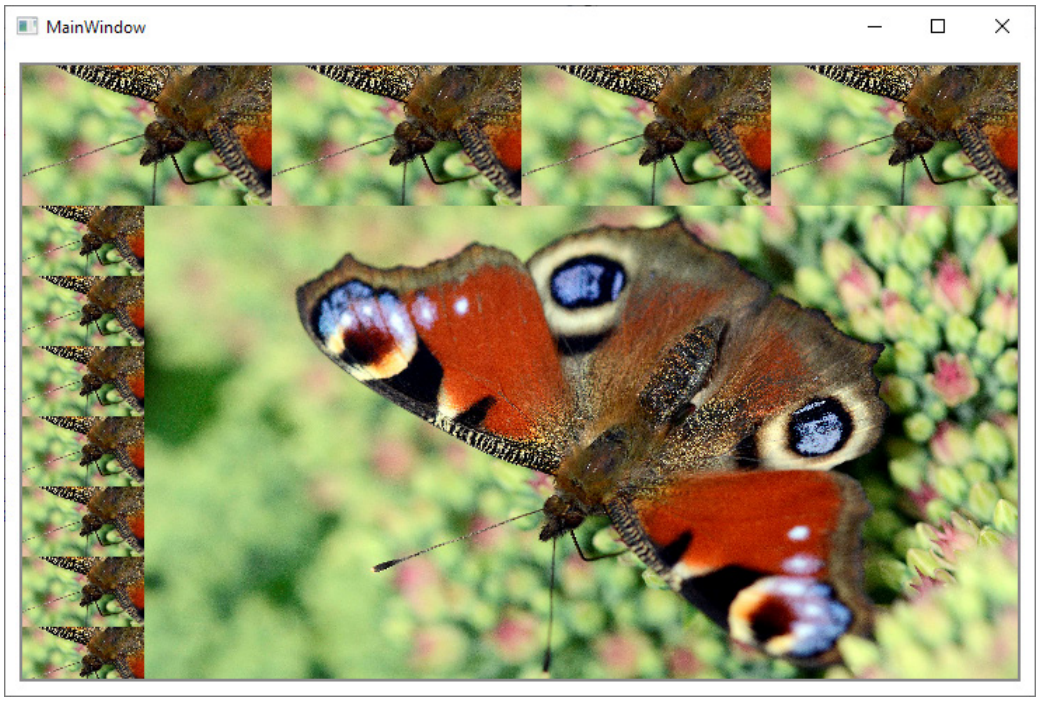
```
<Grid Margin="20">
  <Image Width="400" Height="300" ... >
    <Image.Source>
      <DrawingImage>
        <DrawingImage.Drawing>
          <DrawingGroup>
            <GeometryDrawing>
              <GeometryDrawing.Geometry>
                <LineGeometry StartPoint="0,0" EndPoint="0,300" />
              </GeometryDrawing.Geometry>
              <GeometryDrawing.Pen>
                <Pen Thickness="5" EndLineCap="Square"
                StartLineCap="Triangle">
                  <Pen.Brush>
                    <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
                      <GradientStop Color="DarkGray" Offset="0"/>
                      <GradientStop Color="LightGray" Offset="1"/>
                    </LinearGradientBrush>
                  </Pen.Brush>
                </Pen>
              </GeometryDrawing.Pen>
            </GeometryDrawing>
            <GeometryDrawing>
              <GeometryDrawing.Geometry>
                ...
              </GeometryDrawing>
              <GeometryDrawing>
                <GeometryDrawing.Geometry>
                  <RectangleGeometry Rect="20,100 40,198" />
                </GeometryDrawing.Geometry>
                <GeometryDrawing.Brush>
                  <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
                    <GradientStop Color="Red" Offset="0.5"/>
                    <GradientStop Color="LightGray" Offset="1"/>
                  </LinearGradientBrush>
                </GeometryDrawing.Brush>
              </GeometryDrawing>
              <GeometryDrawing>
                ...
              </GeometryDrawing>
            </DrawingGroup>
```

```
        </DrawingImage.Drawing>
      </DrawingImage>
    </Image.Source>
    <Image.RenderTransform>
      <TransformGroup>
        <RotateTransform Angle="10"/>
        <TranslateTransform X="50" />
      </TransformGroup>
    </Image.RenderTransform>
  </Image>
</Grid>
```

After the entire figure is defined - exclusively in XAML - the Image component is transformed with a rotation and a horizontal offset. It is only to show that the figure can be treated as a single figure.

9.2 DRAWING AN IMAGE

The program *ImageProgram* opens the following window:



The project has as local resources associated with two images, which are simple jpg images. The XML code is:

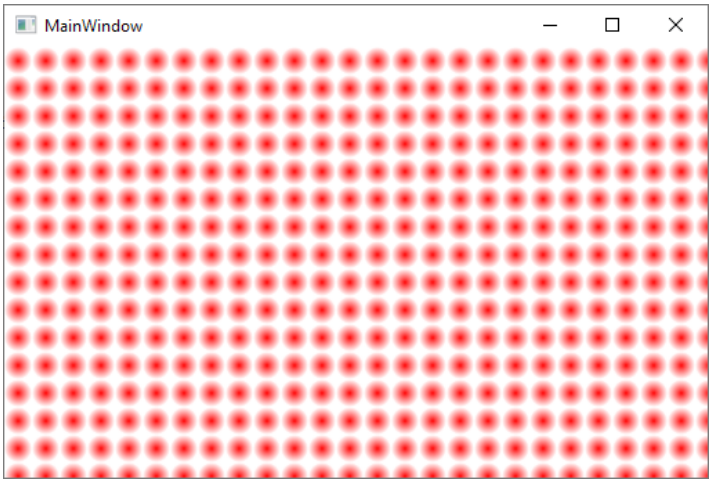
```
<Grid Margin="10">
  <Border BorderBrush="Gray" BorderThickness="2" Background="LightGray"
    HorizontalAlignment="Left" VerticalAlignment="Top">
    <Image Stretch="None">
      <Image.Source>
        <DrawingImage>
          <DrawingImage.Drawing>
            <DrawingGroup>
              <ImageDrawing Rect="0,0,178,100" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="178,0,178,100" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="356,0,178,100" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="534,0,178,100" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="0,100,87,50" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="0,150,87,50" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="0,200,87,50" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="0,250,87,50" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="0,300,87,50" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="0,350,87,50" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="0,400,87,50" ImageSource="img2.jpg"/>
              <ImageDrawing Rect="87,100,625,350" ImageSource="img1.jpg"/>
            </DrawingGroup>
          </DrawingImage.Drawing>
        </DrawingImage>
      </Image.Source>
    </Image>
  </Border>
</Grid>
```

It's not much to notice. The window has an *Image* component whose *Source* is defined by a *DrawingImage,* which is composed of *ImageDrawing* objects.

The image component is inserted into a *Border* object. Note how to draw a frame around a component in this way. I have primarily included the component in this example because I have not used it before, but it is a very useful component that actually provides quite a few options, i.a. by styling its header. It is a component that is worth knowing.

9.3 MANY DRAWINGS

The program *CirclesProgram1* opens the following window:



The example should show:

- How to create *Drawing* objects in code behind
- What significance many objects have for the performance of the program.
- How to capture mouse clicks in the code

The functionality of the program is simple. If you click with the mouse on one of the red circles, the color changes to blue, and if you click again, the color changes to red. You should notice that if you click between the circles (the white area between the circles), nothing happens.

You should also notice how the program behaves if you resize the window and especially if you maximize the window. If you do, many circles must be drawn, and since they are stained with a gradient brush, you will find that it takes time. It takes a long time to create such a large number of *Drawing* objects and get the window rendered.

On the XML side, there is not much this time - only a single *Image* component:

```
<Grid>
  <Image Name="bg" Stretch="None" MouseDown="bg_MouseDown" />
</Grid>
```

On the other hand, the code is both comprehensive and complex:

```
public partial class MainWindow : Window
{
  private enum COLOR { RED, BLUE };
  private Brush redBrush = new RadialGradientBrush(Colors.Red, Colors.
  White);
  private Brush blueBrush = new RadialGradientBrush(Colors.Blue,
  Colors.White);
  private COLOR color = COLOR.RED;

  public MainWindow()
  {
    InitializeComponent();
  }

  private void Window_Activated(object sender, EventArgs e)
  {
    bg.Source = BuildImage(ActualWidth, ActualHeight);
  }

  private void Window_SizeChanged(object sender, SizeChangedEventArgs e)
  {
    bg.Source = BuildImage(ActualWidth, ActualHeight);
  }

  private DrawingImage BuildImage(double width, double height)
  {
    return new DrawingImage(BuildDrawing(width, height));
  }

  private Drawing BuildDrawing(double width, double height)
  {
    double size = 20;
    DrawingGroup drawing = new DrawingGroup();
    for (double x = 0; x < width; x += size)
      for (double y = 0; y < height; y += size)
        drawing.Children.Add(CreateDrawing(x, y, size));
      return drawing;
  }

  private GeometryDrawing CreateDrawing(double x, double y, double size)
  {
    GeometryDrawing drawing = new GeometryDrawing();
```

```
    EllipseGeometry ellip = new EllipseGeometry(new Rect(x, y, size,
    size));
    drawing.Geometry = ellip;
    drawing.Brush = CurrentBrush();
    return drawing;
  }

  private Brush CurrentBrush()
  {
    return color == COLOR.RED ? redBrush : blueBrush;
  }

  private void ColorFlip()
  {
  if (color == COLOR.RED) color = COLOR.BLUE; else color = COLOR.RED;
  foreach (Drawing d in
    ((DrawingGroup)((DrawingImage)bg.Source).Drawing).Children)
    ((GeometryDrawing)d).Brush = CurrentBrush();

  }

  private void bg_MouseDown(object sender, MouseButtonEventArgs e)
  {
  Point pt = e.GetPosition((Image)sender);
  foreach (Drawing d in
    ((DrawingGroup)((DrawingImage) bg.Source).Drawing).Children)
    if (((GeometryDrawing)d).Geometry.FillContains(pt))
    {
      ColorFlip();
      return;
    }

  }
}
```

First, two *Brush* objects are defined, which must be used to color the circles. They are simple enough and note how they are defined with default gradient stop points.

Next, note that the program has event handlers for both activating the window and resizing the window. It is these handlers that result in the creation of an image source for the *Image* component. When these events occur, the method *BuildImage()* is called with the window size as the parameter. The size is sent along with it to calculate how many circles to create. The circles are created and linked to the window in the method *BuildDrawing()* method. *drawing* is an object of the type *DrawingGroup*, and you should especially note that it has

a collection named *Children*, and that you can add *Drawing* objects to this collection. The individual objects are created by the method *CreateDrawing()*, which creates a *GeometryDrawing* for an ellipse, and which is drawn with a brush returned by the method *CurrentBrush()*. The most important thing in all this is to notice how you in C# can create geometric objects and associate them with a component in the user interface.

The component *bg* has an event handler for mouse clicks. The idea is that the figure should be drawn with the opposite brush if a circle is clicked. If an area outside the circles is clicked instead, nothing should happen. The challenge is to determine if a circle has been clicked. This is done by going through all the children of the *Image* component's image source. This leads to many types of casts:

```
((DrawingGroup)((DrawingImage) bg.Source).Drawing).Children)
      if (((GeometryDrawing)d).Geometry.FillContains(pt))
```

- *bg.Source* is a *DrawingImage*
- such an object has a *Drawing*, which in this case is a *DrawingGroup*
- such an object has a collection *Children* with *Drawing* objects
- each *Drawing* object is in this case a *GeometryDrawing*
- and such an object has a *Geometry* which has a method *FillContains()*

If you find an ellipse (equivalent to clicking on an ellipse), the method *ColorFlip()* is called, which changes the current brush and plots all circles. It requires a throughput across all circles and thus the same sequence of type casts as shown above.

If you run the program, as mentioned above, you will notice that the machine cannot keep up to update the user interface - the work is simply too big. There are several solutions to the problem, and the most obvious is to create the necessary *Drawing* objects in advance, and then show only those that are needed. This will help with the problem, since then changing the size of the window does not have to create new graphic objects, but it will not solve the problem as the objects still need to be drawn. You can see this by changing color (clicking on a circle). All that happens here is that the image is drawn again, but no new objects are created. With that maximized window, it is still a comprehensive process that takes a long time.

The program *CirclesProgram2* is exactly the same as above, and the only difference is that the graphic objects (circles) this time are *Shape* objects instead of *Drawing* objects. The goal is to show that these objects compared to *Drawing* objects provide an even slower update of the screen. In addition, the goal is also to show how to work dynamically with *Shape* objects in C#. This time, the user interface consists only of a grid with a name:

```
<Grid Name="grid">
</Grid>
```

Since the user interface must contain *Ellipse* components, and since the number can only be determined at the time of execution, these objects must be created and added to the user interface in C#. The code is similar to the example above, but of course other types are used, and the code is simpler:

```
public partial class MainWindow : Window
{
  private enum COLOR { RED, BLUE };
  private Brush redBrush = new RadialGradientBrush(Colors.Red, Colors.
  White);
  private Brush blueBrush = new RadialGradientBrush(Colors.Blue,
  Colors.White);
  private COLOR color = COLOR.RED;

  public MainWindow()
  {
    InitializeComponent();
  }

  private void Window_Activated(object sender, EventArgs e)
  {
    BuildShapes(ActualWidth, ActualHeight);
  }

  private void Window_SizeChanged(object sender, SizeChangedEventArgs e)
  {
    BuildShapes(ActualWidth, ActualHeight);
  }

  private void BuildShapes(double width, double height)
  {
    grid.Children.Clear();
    double size = 20;
    for (double x = 0; x < width; x += size)
      for (double y = 0; y < height; y += size)
        grid.Children.Add(CreateShape(x, y, size));

  }
```

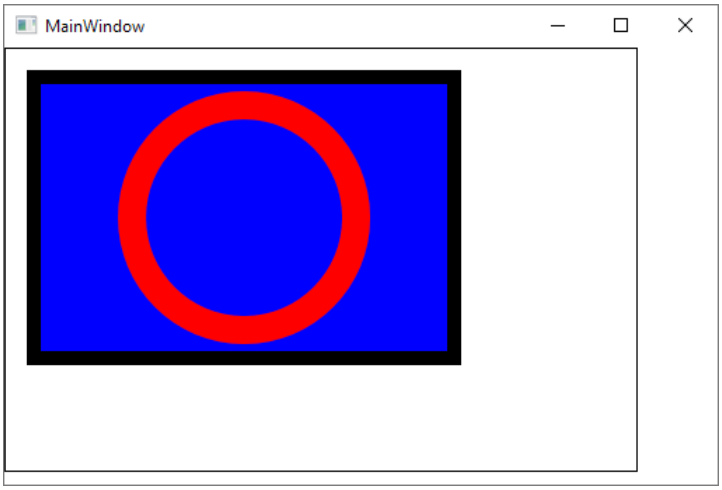
```
  private Brush CurrentBrush()
  {
    return color == COLOR.RED ? redBrush : blueBrush;
  }

  private Shape CreateShape(double x, double y, double size)
  {
    Ellipse ellip = new Ellipse();
    ellip.Width = size;
    ellip.Height = size;
    ellip.Fill = CurrentBrush();
    ellip.HorizontalAlignment = HorizontalAlignment.Left;
    ellip.VerticalAlignment = VerticalAlignment.Top;
    ellip.Margin = new Thickness(x, y, 0, 0);
    ellip.MouseDown += Ellip_MouseDown;
    return ellip;
  }

  private void Ellip_MouseDown(object sender, MouseButtonEventArgs e)
  {
    if (color == COLOR.RED) color = COLOR.BLUE; else color = COLOR.RED;
    foreach (UIElement elem in grid.Children)
      if (elem is Ellipse) ((Ellipse)elem).Fill = CurrentBrush();
  }
}
```

Note that all variables (enum and brush objects) and event handlers for activating the window and resizing the window are unchanged. The method *CreateShape()* creates an *Ellipse* and sets values for the ellipse's properties. Since an *Ellipse* is a component, it captures events from the mouse, and you can thus attach an event handler to each *Ellipse* component. This makes event handling significantly simpler than in the previous example, as this time you do not have to calculate whether a circle has been clicked. You should also note how the individual *Shape* objects are this time linked directly to the window's *Grid* panel, which is done in the method *BuildShapes()*.

10 VISUALS

The last type of graphics that WPF supports is based on objects derived from the type *Visual*. These are objects that give the programmer the greatest degrees of freedom, as everything has to be written in C#, but conversely it is also both technical and comprehensive and many lines of code often have to be written. Generally, do not use this graphics layer unless there are special needs. I will show four examples, and they are by no means exhaustive, but they do show a little about what is needed.

The first example opens the following window:



where a rectangle and a circle are drawn. The window has an *Image* component, and the example should show how to create an image as a *Visual* object and use it as an image source. There is not much on the XML page:

```
<Window x:Class="VisualProgram.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" Loaded="Window_
        Loaded">
  <Grid>
    <Border BorderThickness="1" BorderBrush="Black"
      HorizontalAlignment="Left" VerticalAlignment="Top" >
      <Image Name="img" Stretch="None"
        HorizontalAlignment="Left" VerticalAlignment="Top" />
    </Border>
  </Grid>
</Window>
```

The *Image* component is located in a *Border* component, and the only reason is that a frame must be drawn around the image so that its size is visible.

You should also note that the window captures the *Loaded* event that occurs before the window appears on the screen, and it is the handler *Window_Loaded()* that do the work:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  DrawingVisual dv = new DrawingVisual();
  using (DrawingContext dc = dv.RenderOpen())
  {
    dc.DrawRectangle(Brushes.Blue, new Pen(Brushes.Black, 10),
      new Rect(20, 20, 300, 200));
    dc.DrawEllipse(null, new Pen(Brushes.Red, 20), new Point(170,
    120), 80, 80);
  }
  RenderTargetBitmap bmp = new RenderTargetBitmap(450, 300, 96, 96,
    PixelFormats.Pbgra32);
  bmp.Render(dv);
  img.Source = bmp;
}
```

The image is represented by a *DrawingVisual*, which is a *Visual* object. It can be used to render graphic objects on the screen. These objects are created with a *DrawingContext* object, which provides several drawing tools, and in this case two are used: *DrawRectangle()* and *DrawEllipse()* which draw respectively a rectangle and an ellipse. You should note that the object is created with *RenderOpen()* and that it happens in a *using* statement so that *Dispose()* is executed. It is necessary. In this case, the program draw:

1. A rectangle filled with a blue color and a black border of 10 pixels. The upper left corner is (20, 20) and width and height are respectively 300 and 200, where the numbers indicate pixels.
2. An ellipse that is not filled and drawn with a red border of 20 pixels. The center is (170,120) and both radii are 80.

After the geometry is defined, the whole thing must be converted to a bitmap. First, a bitmap of the type *RenderTargetBitmap* is created. The first two parameters indicate the width and height of the image in pixels that are respectively 450 and 300. The next two are the resolution, which is measured in pixels per inch. 96 pixels per inch is typical, but of course there are other options. The last parameter is the pixel format, and here there are many options. *Pbgra32* means standard RGB colors with 8 bits per channel. After the image is created, it is rendered based on the definition of the geometry, and finally the image is attach to the *Image* component.

The next example is basically the same as the previous one, where shapes are drawn in a window by attaching an image to an *Image* component. The example should primarily show how to draw text. The program opens the following window:

The program is basically written like the previous example, and on the XML page, only an *Image* component is defined:

```
<Grid>
  <Image Name="img" Stretch="None" HorizontalAlignment="Center"
    VerticalAlignment="Center" />
</Grid>
```

In the code, there is the event handler for the event *Loaded*, which creates the image:

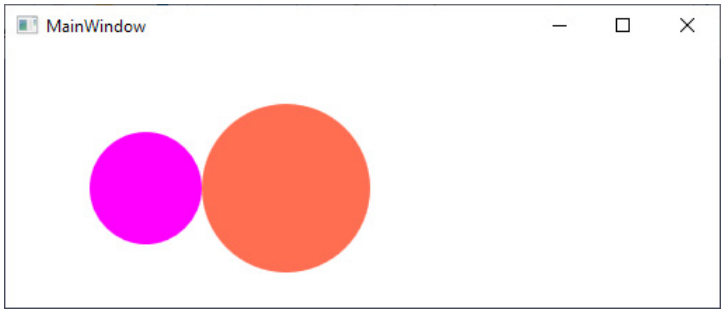
```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  LinearGradientBrush bg = new LinearGradientBrush();
  bg.StartPoint = new Point(0, 1);
  bg.EndPoint = new Point(1, 0);
  bg.GradientStops.Add(new GradientStop(Colors.LightGreen, 0.2));
  bg.GradientStops.Add(new GradientStop(Colors.DarkGreen, 0.8));
  FormattedText text = new FormattedText("Egil Skallagrimsson",
    new System.Globalization.CultureInfo("da-DK"),
    System.Windows.FlowDirection.LeftToRight,
    new Typeface(FontFamily, FontStyles.Italic, FontWeights.Bold,
    FontStretches.Normal), 48, Brushes.Red);
  DrawingVisual dv = new DrawingVisual();
  using (DrawingContext dc = dv.RenderOpen())
  {
    dc.DrawRectangle(bg, null, new Rect(0, 0, 570, 240));
    dc.DrawEllipse(Brushes.Yellow, new Pen(Brushes.Blue, 10), new
    Point(270, 120), 250, 100);
    dc.DrawText(text, new Point(40, 80));
  }
  RenderTargetBitmap bmp =
    new RenderTargetBitmap(570, 240, 100, 90, PixelFormats.Pbgra32);
  bmp.Render(dv);
  img.Source = bmp;
}
```

There is not much to explain, but more to take note of. You should primarily note:

1. How to create a *GradientBrush* in C# and including how to set the different properties.
2. How to define a text as a *FormattedText* object. There are many parameters, but it is easy enough to interpret the meaning.

Otherwise, the image is created completely in the same way as in the previous example.

10.1 HIT TEST



If you run the example *HittestProgram*, you will get the above window. If, for example you click on the orange figure, it will be transformed:

The example should thus show how to capture mouse clicks for a *Visual* object, and the example also shows another way of placing a figure in a window. The starting point this time is a class, which represents a simple component that can be placed in a window:

```
public class Circles : FrameworkElement
{
  private VisualCollection visuals;

  public Circles()
  {
    visuals = new VisualCollection(this);
    visuals.Add(CreateCircle(Brushes.Magenta, new Point(100, 100), 40));
    visuals.Add(CreateCircle(Brushes.Tomato, new Point(200, 100), 60));
    MouseDown += Host_MouseDown;
  }
```

```
  protected override int VisualChildrenCount
  {
    get { return visuals.Count; }
  }

  protected override Visual GetVisualChild(int index)
  {
    return visuals[index];
  }

  private Visual CreateCircle(Brush brush, Point center, double radius)
  {
    DrawingVisual dv = new DrawingVisual();
    using (DrawingContext dc = dv.RenderOpen())
    {
      dc.DrawEllipse(brush, null, center, radius, radius);
    }
    return dv;
  }

  private void Host_MouseDown(object sender, MouseButtonEventArgs e)
  {
    Point pt = e.GetPosition((UIElement)sender);
    VisualTreeHelper.HitTest(this, null, new HitTestResultCallback(Mous
    eCallback),
      new PointHitTestParameters(pt));
  }

  public HitTestResultBehavior MouseCallback(HitTestResult result)
  {
    if (result.VisualHit.GetType() == typeof(DrawingVisual))
    {
      DrawingVisual dv = (DrawingVisual)result.VisualHit;
      if (dv.Transform == null) dv.Transform = new SkewTransform(25, 25);
      else dv.Transform = null;
    }
    return HitTestResultBehavior.Stop;
  }
}
```

The class is in this case in the same file as the main window, but it does not matter, and in most cases you would probably place the class in its own file. With this class available, you can write the XML code as follows:

```
<Window x:Class="HittestProgram.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:HittestProgram"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
      <local:Circles/>
    </Grid>
</Window>
```

The grid panel has a component of the type *Circles*, and note the syntax, and that

```
<local:Circles/>
```

means that an object of the type *Circles* (an image) is instantiated and inserted in the panel.

The class *Circles* inherits the class *FrameworkElement* and is thus a component that can be inserted into the user interface. At the same time, it provides opportunities for an object of the class to capture events from the mouse. Most importantly, however, the class inherits all the logic necessary to render the content. In this case, the class must represent a collection of *Visual* objects, and the class therefore consists of a collection for that purpose:

```
private VisualCollection visuals;
```

This collection is created in the constructor and two ellipses of the type *DrawingVisual* are added. Finally, the event handler for the mouse is associated. This handler is complex. First, the coordinates of the mouse are determined relative to the window that contains the component. Next, test whether the mouse points to one of the two circles, and thus whether the component has been clicked. For this, a class *VisualTreeHelper* is used, which has different help methods regarding elements of a component's visual tree. In this case, the method *HitTest()* is used, which tests whether the component has been clicked. The parameters mean the following:

1. *this*, the object to be tested for a hit
2. *null*, method defined by a delegate *HitTestFilterCallback*, which is a callback method that specifies areas of the current *Visual* that are not to be included in the hit area
3. *HitTestResultCallback*, which is a delegate to a callback method that performs the test itself
4. test parameters in the form of the point for which the hit is to be tested, which is encapsulated here in a *PointHitTestParameters*

All in all, a complex function call. The callback method is called *MouseCallback()*, and if a *Visual* in the class's collection is clicked, the object is transformed (or a transformation is removed if it was already transformed).

The complexity of this example is the hit test, and it's one of the kind that it's best to take note of, that it's the way it's written, and at best, you probably cannot remember what to write. Somewhere, of course, it must also be complex, as the system must calculate whether the position of the mouse falls within the area constituting that component. You should also note that nothing needs to be done to render the object. The framework manages this quite automatically.

10.2 VISUAL OBJECTS IN C#

As a conclusion to the chapter on Visual Objects, I will return to the example with many drawings and look at the same problem, but this time the circles are *Visual* objects. The program is called *CirclesProgram3*.

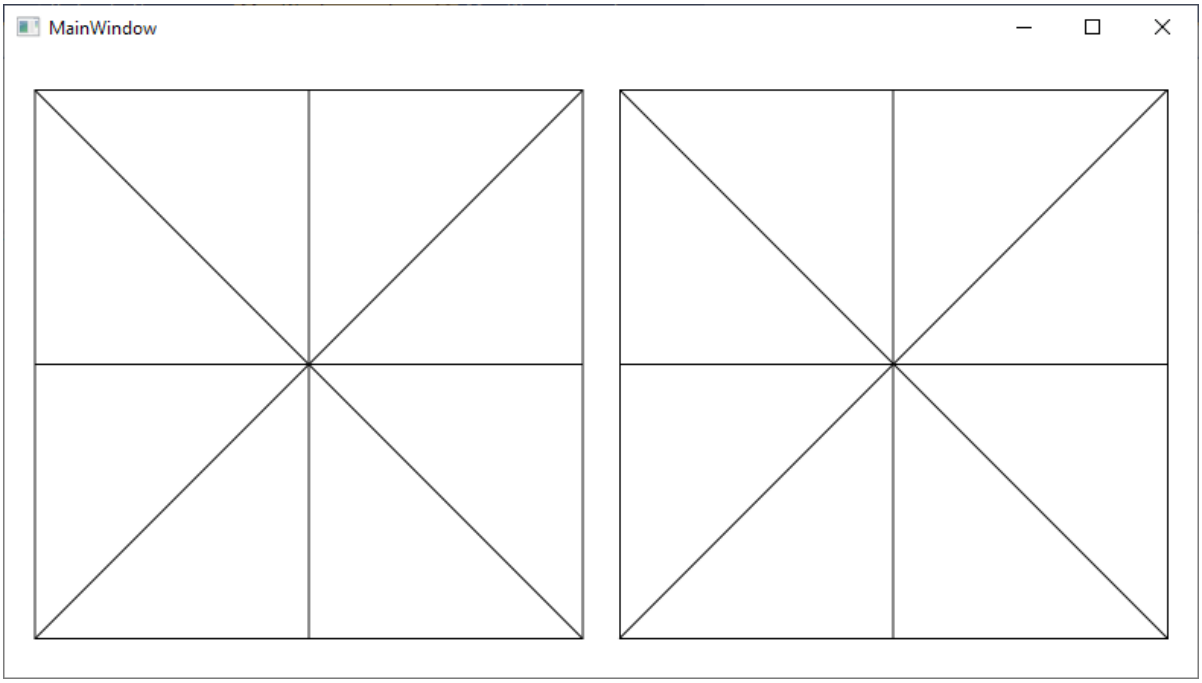
However, there is a significant difference, as the individual circles are colored with a filled brush rather than a gradient brush. This version of the program is much faster (when updating the screen when the window is resized or the window is maximized), but this is not due to the objects - which here have the type *DrawingVisual* - but rather that a gradient brush is not used. The purpose of this change is to show how much it means for performance to use a gradient brush. Of course, one must also be aware that the example is a bit extreme, as a maximized window (depending on the screen resolution) must render over 5000 ellipse objects. The XML code is identical to the previous example:

```
<Window x:Class="CirclesProgram3.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:gr="clr-namespace:Exam111"
        Title="MainWindow" Height="350" Width="525">
  <Grid>
    <gr:Circles/>
  </Grid>
</Window>
```

In principle, this also applies to the C# code, which only consists of a single class *Circles*. It is in principle also identical to the class *Circles* from the previous example, however this time it also captures the *SizeChanged* event, which fires when the size of the component changes.

10.3 SOME DETAILS ABOUT GRAPHICS

Perhaps the most important reason for using the graphical primitives is probably the development of your own controls and thus controls which are drawn in a way completely determined by the programmer. In principle, it is not difficult and the following program (*ASimpleControl*) shows a window with two custom control:



The two controls draws exactly the same control and on the screenshot above the two figures looks identical, but if you run the program you will observe a difference. All lines are drawn with a pen that is 1 pix, but the vertical lines in the left figure are shown some blurred and as they were not drawn with a pen with 1 pix. If you slowly resize the window you will observe that the width of the lines in the left figure changes depending on the size of the window. That is the left control is not drawn correct.

The code for the left control is:

```
public class TheControl1 : FrameworkElement
{
  protected override void OnRender(DrawingContext dc)
  {
    Point point1 = new Point(20, Height / 2);
    Point point2 = new Point(Width - 20, Height / 2);
    Point point3 = new Point(Width / 2, 20);
    Point point4 = new Point(Width / 2, Height - 20);
    Point point5 = new Point(20, 20);
    Point point6 = new Point(Width - 20, Height - 20);
    Point point7 = new Point(20, Height - 20);
    Point point8 = new Point(Width - 20, 20);
    Rect rect = new Rect(20, 20, Width - 40, Height - 40);

    Pen pen = new Pen(Brushes.Black, 1);
    dc.DrawRectangle(Brushes.White, pen, rect);
    dc.DrawLine(pen, point1, point2);
    dc.DrawLine(pen, point3, point4);
    dc.DrawLine(pen, point5, point6);
    dc.DrawLine(pen, point7, point8);
  }

  protected override void OnMouseDown(MouseButtonEventArgs e)
  {
    MessageBox.Show("You have clicked the mouse on the left
    component");
  }
}
```

The method *OnRender()* is a method which is called by the runtime system, when a control has to be redrawn. The parameter is a *DrawingContext* object, and you can draw the component using drawing primitives for this object either directly or by being called auxiliary methods from that method. In this case the method is quite simple and defines

8 points used as end points for the four lines and a rectangle. Then is defined a pen and the five shapes can be drawn.

The class has also and override of *OnMouseDown()* to capture event from the mouse, just to assign some action to the component.

Then there is the problem with the drawing the lines correct. The problem has to do with how WPF draw graphic, so a little explanation is needed.

When you draw a line in WPF you will experience that the line often appear blurry. The reason for this is the antialiasing system that spreads the line over multiple pixels if it doesn't align with physical device pixels. WPF seeks to draw images independent of the current screen resolution. This means you specify the size of a user interface element in inches, and not in pixels. In WPF is a logical unit 1/96 of an inch, because most screens have a resolution which is 96 dpi. So in most cases 1 logical unit is the same as 1 physical pixel, but it is independents of the current screen resolution. When the lines appear blurry, it is because the coordinates means center points on the lines and not edges. With a pen width of 1 the edges are drawn between two pixels exactly. As mentioned the system use antialiasing which is a software technique for diminishing step-like lines that should be smooth. Step-like occur because the output device, the monitor or printer, doesn't have a high enough resolution to represent a smooth line. It means that when the runtime system must determine the colors of the individual pixels in a figure it colored a pixel depending of how much of the pixel that falls within the figure, and in this way pixels for a step-like figure will be blurred. If you have to draw a vertical and horizontal straight line, and the logical coordinates define a line through the center of the physical pixels the runtime system will use antialiasing on pixels on both sides of the line, and the line appears blurry.

If you do not want this effect (and for the most of the time you do) you must tell the render system to use physical coordinates instead of logical coordinates and exactly to round logical coordinates to physical coordinates. How to, is shown in the following method that is the method from the class for the right control in the above program:

```
protected override void OnRender(DrawingContext dc)
{
  Point point1 = new Point(20, Height / 2);
  Point point2 = new Point(Width - 20, Height / 2);
  Point point3 = new Point(Width / 2, 20);
  Point point4 = new Point(Width / 2, Height - 20);
  Point point5 = new Point(20, 20);
  Point point6 = new Point(Width - 20, Height - 20);
  Point point7 = new Point(20, Height - 20);
  Point point8 = new Point(Width - 20, 20);
  Rect rect = new Rect(20, 20, Width - 40, Height - 40);

  Matrix m =
    PresentationSource.FromVisual(this).CompositionTarget.
    TransformToDevice;
  double dpiFactor = 1 / m.M11;
  Pen pen = new Pen(Brushes.Black, 1 * dpiFactor);

  double halfPenWidth = pen.Thickness / 2;

  GuidelineSet guidelines = new GuidelineSet();
  guidelines.GuidelinesX.Add(rect.Left + halfPenWidth);
  guidelines.GuidelinesX.Add(rect.Right + halfPenWidth);
  guidelines.GuidelinesY.Add(rect.Top + halfPenWidth);
  guidelines.GuidelinesY.Add(rect.Bottom + halfPenWidth);

  guidelines.GuidelinesX.Add(point1.X + halfPenWidth);
  guidelines.GuidelinesY.Add(point1.Y + halfPenWidth);
  guidelines.GuidelinesX.Add(point2.X + halfPenWidth);
  guidelines.GuidelinesY.Add(point2.Y + halfPenWidth);

  guidelines.GuidelinesX.Add(point5.X + halfPenWidth);
  guidelines.GuidelinesY.Add(point5.Y + halfPenWidth);
  guidelines.GuidelinesX.Add(point6.X + halfPenWidth);
  guidelines.GuidelinesY.Add(point6.Y + halfPenWidth);

  dc.PushGuidelineSet(guidelines);
  dc.DrawRectangle(Brushes.White, pen, rect);
  dc.DrawLine(pen, point1, point2);
  dc.DrawLine(pen, point5, point6);
  dc.Pop();

  dc.DrawLine(pen, point3, point4);
  dc.DrawLine(pen, point7, point8);
}
```

First the method defines the same points and rectangle as in the left control, but the pen width is adjusted with a factor. This factor is 1 divides by the screen resolution as pixels per inch (which for the most is 96), and the factor will for most screens be 1. The the half of the pen with is determined (note the pen width do not need to be 1 but can have other values, also values less than 1). This value is used to adjust the coordinates using a *GuideLineSet* that is a collection containing all coordinates which the system should convert to nearest physical coordinates. Before the figures are drawn the guidelines are pushed on a stack within the *DrawingContext* and figures are drawn. If you now longer need the guidelines you should pop the stack, and in this case the last two lines are drawn as in the left control and the using antialiasing.

11 DRAG AND DROP

This chapter is about drag and drop, which deals with:

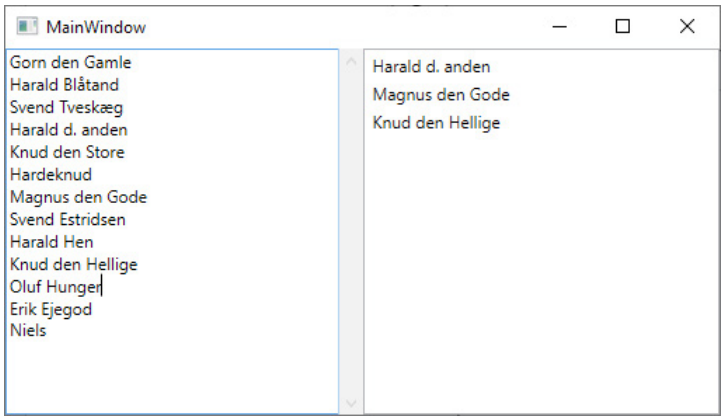
1. how to move an item from one place in the user interface to another by dragging the item with the mouse
2. how to resize a component using the mouse by selecting the component and dragging in special marks

Slightly different, it is a method where you use the mouse to select one or more objects and drag them over a drop area, where you then throw them when you release the mouse.

This operation is built into WPF and its components, and as long as you only use standard components, it is simple enough, but it can quickly develop into something that requires a lot of programming and with many details.

I want to start simple with a program (called *DropLines*) where the user interface has a *TextBox* (left) and a *ListBox* (right):

In the input field you can enter lines. You can then select a line and drag it over the list box and drop it:



If you try the program, be aware that it does not work properly, as the program always performs a copy. Normally you have to hold down the control key for a copy. Otherwise a move occurs. I will solve that problem in the next example.

The layout is a grid with two columns. In one column there is a multiline *TextBox* encapsulated in a *ScrollViewer*, and in the other column there is a list box:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <ScrollViewer Grid.Column="0">
    <TextBox Name="txtLines" AcceptsReturn="True"/>
  </ScrollViewer>
  <ListBox Name="lstLines" Grid.Column="1" AllowDrop="True"
   Drop="lstLines_Drop"/>
</Grid>
```

You should note that the *ListBox* component has set its *AllowDrop* property set to *true*. This means that the component allows dropping objects. However, a little more is needed, because these objects also need to be handled. It happens in a *Drop* event dealer:

```
private void lstLines_Drop(object sender, DragEventArgs e)
{
  try
  {
    if (!e.Handled)
    {
      string line = txtLines.SelectedText;
      if (line != null && line.Trim().Length > 0) lstLines.Items.
      Add(line.Trim());
    }
  }
  catch
  {
  }
}
```

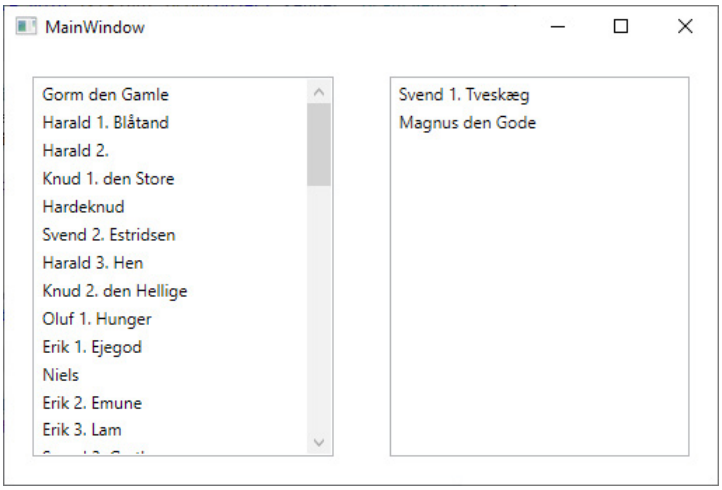
If the event in question has not already been processed, the text in the input field that is selected is retrieved and if there is a selected text, it is inserted as a line in the list box. You should note that the solution is quite hardcoded relative to the two components. I do not use

the parameter *e* of the type *DragEventArgs*. I will do that in the next example. You should also note that the *TextBox* component has built-in drag and drop logic, and you can immediately move around texts - and even correctly, so you can perform both moving and copying.

In practice, drag and drop is important, and users will generally expect this functionality. On the other hand, it is not always simple to implement, and there are 6 steps:

1. Capture a drag operation as a combination of *MouseLeftButtonDown* and *MouseMove*
2. Decide which data you want the drag operation to include and encapsulate this data in a *DataObject*, which in addition to data has information about format and effect
3. Start the operation by calling *DoDragDrop()*
4. Set *AllowDrop* to true for the components where it should be possible to drop
5. Register a handler for the event *DragEnter* to detect that you are over the drop area, that the format is correct by calling *GetDataPresent()* and in case of drop set the drop argument's *Effect* property
6. Call the method *GetData()* as a result of the *Drop* event

It sounds complicated, and it is, but in the next examples, I will show how. The first example *DropLines1* is relatively simple, and it opens a window with two list boxes:



When the window opens, all names (above *Danish* kings) are in the left list box, and you can then drag names over to the other list box - and vice versa. If you move a name from one list box to another, the name is removed from the first list box.

The starting point is the class *Kings* from earlier. After the project is created, this class is added to the project and the class' namespace is changed. The class represents a collection

of objects of the type *King*, where a *King* object represents a Danish king. Since these two classes are mentioned earlier, I will not look further at them here.

The XML code is as follows:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition/>
    <ColumnDefinition/>
  </Grid.ColumnDefinitions>
  <ListBox Name="lstLeft" Grid.Column="0" Margin="20"
  DisplayMemberPath="Name"
    AllowDrop="True" PreviewMouseLeftButtonDown="lst_
    PreviewMouseLeftButtonDown"
    Drop="lstLeft_Drop" />
  <ListBox Name="lstRight" Grid.Column="1" Margin="20"
  DisplayMemberPath="Name"
    AllowDrop="True" PreviewMouseLeftButtonDown="lst_
    PreviewMouseLeftButtonDown"
    Drop="lstRight_Drop" />
</Grid>
```

and here is the interesting definition of the list boxes. Since both list boxes are bound to a collection of objects of the type *King* the property *DisplayMemberPath* defines that it is the value of the *King* class's *Name* property that is to be displayed. For both list boxes, it is defined that it must be possible to drop objects. When the user drops an object, a *Drop* event is fired, and therefore both list boxes have associated a corresponding event handler as the method performed when dropping an object. According to the above 6 points to implement drag and drop, one has to capture the start of a drag operation and it happens with an event handler for *PreviewMouseLeftButtonDown* that starts the operation if it makes sense. Here you should note that the operation is not started by a *MouseLeftButtonDown*, but a *PreviewMouseLeftButtonDown* event. Note in particular that both list boxes use the same event handler for *PreviewMouseLeftButtonDown*.

The rest regarding drag and drop are all performed in C#. The first step is to tie the two list boxes to collections:

```
private ObservableCollection<King> leftSource =
  new ObservableCollection<King>(new Kings());
private ObservableCollection<King> rightSource = new
ObservableCollection<King>();

public MainWindow()
{
  InitializeComponent();
  lstLeft.ItemsSource = leftSource;
  lstRight.ItemsSource = rightSource;
}
```

It has nothing to do with drag and drop, but since the two list boxes are bound to a data source of the type *ObservableCollection*, the list boxes are automatically updated if the data source is changed. The rest of the code regarding implementation of drag and drop:

```
private void lst_PreviewMouseLeftButtonDown(object sender,
MouseButtonEventArgs e)
{
  try
  {
    ListBox parent = (ListBox)sender;
    object data = GetData(parent, e.GetPosition(parent));
    if (data != null && data is King)
      DragDrop.DoDragDrop(parent, data, DragDropEffects.Move);
  }
  catch
  {
  }
}

private void lstLeft_Drop(object sender, DragEventArgs e)
{
  try
  {
    King data = (King)e.Data.GetData(typeof(King));
    rightSource.Remove(data);
    leftSource.Add(data);
  }
  catch
  {
  }
}
```

```
private void lstRight_Drop(object sender, DragEventArgs e)
{
  try
  {
    King data = (King)e.Data.GetData(typeof(King));
    leftSource.Remove(data);
    rightSource.Add(data);
  }
  catch
  {
  }
}

private object GetData(ListBox source, Point pt)
{
  try
  {
    UIElement element = source.InputHitTest(pt) as UIElement;
    if (element != null)
    {
      object data = DependencyProperty.UnsetValue;
      while (data == DependencyProperty.UnsetValue)
      {
        data = source.ItemContainerGenerator.ItemFromContainer(element);
        if (data == DependencyProperty.UnsetValue)
          element = VisualTreeHelper.GetParent(element) as UIElement;
        if (element == source) return null;
      }
      if (data != DependencyProperty.UnsetValue) return data;
    }
  }
  catch
  {
  }
  return null;
}
```

I want to start from behind with the method *GetData()*. As mentioned above, step 2 of the drag and drop operation is to determine the object to which the operation relates. *GetData()* has two parameters, the first being the list box that the mouse points to and the second being the coordinates of the mouse relative to the list box. A *ListBox* has (in the same way as other components) a method *InputHitTest()*, which from a point returns a reference to the element within which the coordinates of the point fall. In this case, it will be the item (exactly a *TextBlock*) in the list box - if the mouse points to an item. If

this is the case, a loop is started for the purpose of finding the object that the element visualizes in the list box. A *ListBox* has a property *ItemContainerGenerator*, which represents relationships between UI elements and the data source, and for that collection there is a method *ItemFromContainer()*, which from a UI element tries to determine a corresponding data object. If this is not possible, the method returns a value *UnsetValue*, which means that the element does not refer to a data object. If this is the case, you try to go one level up in the visual tree, and if this is not possible (equivalent to reaching the list box yourself), the method returns *null*. The loop will therefore stop by the method returning *null* or a data object has been found. In this case, the object is returned.

Looking at the event handler

```
lst_PreviewMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
```

it starts by calling *GetData()* to determine the line clicked and thus the object to be transferred during the drag operation. If a *King* object is found, the operation is started by calling method *DoDragDrop()*:

```
DragDrop.DoDragDrop(parent, data, DragDropEffects.Move);
```

which, based on the parameters, creates a *DragObject*. Note in particular that you also specify the effect, which in this case is *Move*, where elements are moved from one list box to another. The operation thus does not support *Copy*, where you hold down the control key while dragging the mouse. It is steps 2 and 3 in the implementation of drag and drop.

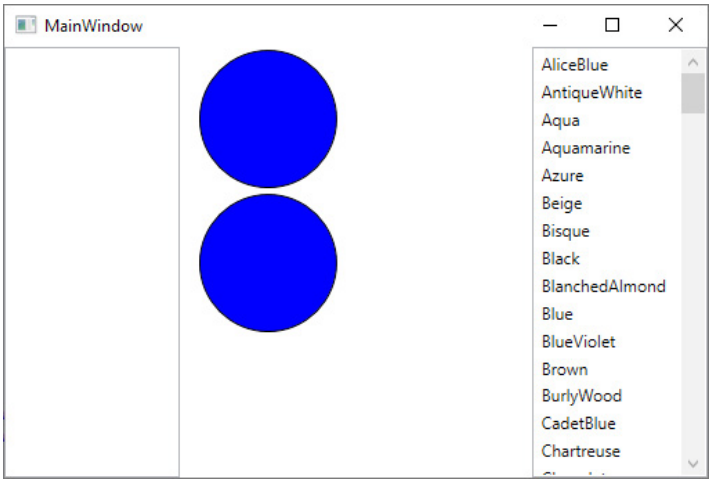
Step 4 of setting *AllowDrop* to *true* for the components where one should be able to drop is performed in XML.

In this case, step 5 is not included, but it is in the next example. It is used to tell what to do while dragging the mouse.

The last step is the event handlers for *Drop*, and thus what should happen if you drop an object. It consists of retrieving the object that is linked to the event argument. If it works well it is a *King* object, the two data sources are updated.

11.1 DRAG AND DROP OBJECTS

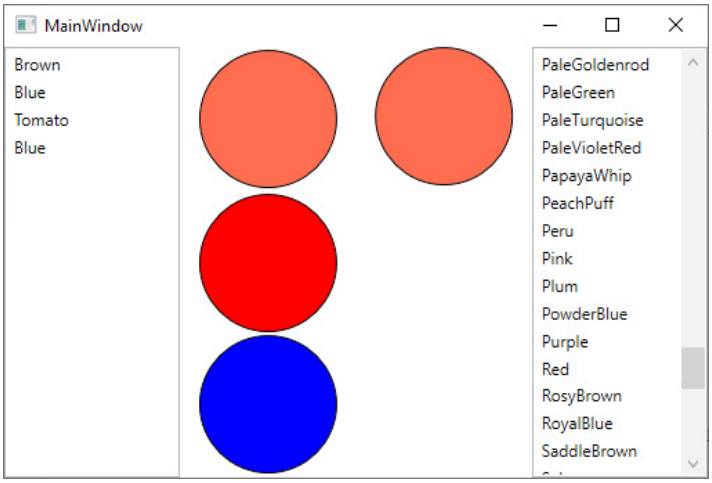
This example, like the previous example, deals with drag and drop and how to move objects using the mouse. The objects are this time resp. a brush and an ellipse. When the program starts, you get the following window:



where there is a list box on the left and right, while in the middle there are two panels (which are not visible) that can contain Shape objects. The right list box contains from the start a series of names of standard colors that represent standard Brush objects. The left panel initially has two *Ellipse* objects. Then you can do the following:

- Drag a color from the right list box to the left list box.
- Draw a color from one of the two list boxes and drop it on a circle.
- Drag a circle from a panel and drop it on the left list box.
- Drag a circle from one panel to the other panel.
- Draw a circle and drop it on another circle.
- Copy a circle (hold down the control key) and drop it in a panel.

After several drag and drop operations, the result could be the following:



The start is a class *PredefinedBrushes* which represents a collection with all standard *Brush* objects from the framework class *Brushes*. This collection contains objects of the type:

```
public class PredefinedBrush
{
  public string BrushName { get; private set; }
  public Brush BrushColor { get; private set; }

  public PredefinedBrush(string name, Brush brush)
  {
    BrushColor = brush;
    BrushName = name;
  }

  public override string ToString()
  {
    return BrushColor.ToString();
  }
}
```

and is thus a class that associates a name with a brush. It is a bit technical to create a collection of all objects in *Brushes*. This is done through the use of reflection (and I have shown how before), so I will not discuss this class in more detail here.

The circles are defined as a *CustomControl* with the following XML:

```
<Grid>
  <Ellipse x:Name="shape" StrokeThickness="1" Stroke="Black"
  Height="100"
  Width="100" Fill="Blue" />
</Grid>
```

The class must contain some logic for drag and drop, and the code is explained in more detail below.

Then there is XML for the window, which is primarily a *Grid* with 4 columns:

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <ListBox Name="lstColors" Grid.Column="0"
  DisplayMemberPath="BrushName"
    AllowDrop="True" Drop="lstColors_Drop"
    PreviewMouseLeftButtonDown="lstColors_PreviewMouseLeftButtonDown" />
  <StackPanel Grid.Column="1" Background="White" AllowDrop="True"
    DragOver="panel_DragOver" Drop="panel_Drop">
    <local:Circle Margin="2" />
    <local:Circle Margin="2" />
  </StackPanel>
  <StackPanel Grid.Column="2" Background="White" AllowDrop="True"
    DragOver="panel_DragOver" Drop="panel_Drop">
  </StackPanel>
  <ListBox Name="lstBrushes" Grid.Column="3"
  DisplayMemberPath="BrushName"
    PreviewMouseLeftButtonDown="lstBrushes_PreviewMouseLeftButtonDown" />
</Grid>
```

You should primarily note the following:

1. The first list box supports drag and drop in the same way as in the previous example, and note that it should display the name of a *PredefinedBrush*.
2. The second list box supports drag but not drop. So you can not drop objects on it.
3. The two panels are both of the type *StackPanel* and they are prepared for drag and drop. Note that for both panels, *Background* is set. Although it has no visible effect (the color is white), it is necessary as otherwise there is nothing to drop on. Both panels use the same event handlers, and here you need to pay special attention to the *DragOver* event, which helps to have an effect when the mouse is moved across the panel and a drop object is attached.

This time the C# code fills a lot and I will explain the most important below.

First there is the class *Circle* and thus the code for the program's *CustomControl*. In addition to the shape object from the user interface, it has a single instance variable, which is used in connection with drag and drop:

```
public partial class Circle : UserControl
{
  private Brush currentFill = null;
```

The class also has a copy constructor:

```
public Circle(Circle c)
{
  InitializeComponent();
  shape.Height = c.shape.Height;
  shape.Width = c.shape.Height;
  shape.Fill = c.shape.Fill;
}
```

which is used when a *Circle* object is dropped over a panel as a copy. Since the class is a *CustomControl*, it inherits virtual methods from the base class concerning the mouse. The necessary logic for drag and drop can therefore be implemented by overriding these methods.

If you move the mouse over a circle and the left button is pressed, you must start a drag operation with a *DataObject*, which refers to the circle and the brush with which the circle is drawn:

```
protected override void OnMouseMove(MouseEventArgs e)
{
  base.OnMouseMove(e);
  try
  {
    if (e.LeftButton == MouseButtonState.Pressed)
    {
      DataObject data = new DataObject();
      data.SetData(typeof(Circle), this);
      PredefinedBrush brush = PredefinedBrushes.GetBrush(Fill);
      if (brush != null) data.SetData(typeof(PredefinedBrush), brush);
      DragDrop.DoDragDrop(this, data, DragDropEffects.Copy |
      DragDropEffects.Move);
    }
  }
  catch
  {
  }
}
```

A *DataObject* has a collection for the objects to which the drag operation relates. The objects are associated with the method *SetData()*, where the first parameter identifies the object with either a type or a string, and the second parameter is the object. After the objects are captured, the operation is started with *DoDragDrop()*, where the parameters are the component that starts the operation, data objects and an effect, which here can be both *Move* and *Copy*. Apparently, the above method is performed continuously while the mouse is moving, but while a drag is in progress, *MouseMove* events are no longer fired. Instead, *DragOver* events are fired:

```
protected override void OnDragOver(DragEventArgs e)
{
  base.OnDragOver(e);
  try
  {
    e.Effects = DragDropEffects.None;
    if (e.Data.GetDataPresent(typeof(PredefinedBrush)))
    {
      if (e.KeyStates.HasFlag(DragDropKeyStates.ControlKey))
        e.Effects = DragDropEffects.Copy; else e.Effects = DragDropEffects.
      Move;
    }
    e.Handled = true;
  }
  catch
  {
  }
}
```

In this case, the method tests whether there is a drag object of the type *PredefinedBrush*, and if so, the effect is set to either *Copy* or *Move*, and the result is that the cursor will show whether one can drop an object. The cursor is set in the method *OnGiveFeedback()*:

```
protected override void OnGiveFeedback(GiveFeedbackEventArgs e)
{
  base.OnGiveFeedback(e);
  try
  {
    if (e.Effects.HasFlag(DragDropEffects.Copy)) Mouse.SetCursor(Cursors.
    Cross);
    else if (e.Effects.HasFlag(DragDropEffects.Move)) Mouse.
    SetCursor(Cursors.Pen);
    else Mouse.SetCursor(Cursors.No);
    e.Handled = true;
  }
  catch
  {
  }
}
```

If a drag operation is in progress and you move the mouse over a circle, the following method is performed:

```
protected override void OnDragEnter(DragEventArgs e)
{
  base.OnDragEnter(e);
  try
  {
    currentFill = shape.Fill;
    if (e.Data.GetDataPresent(typeof(PredefinedBrush)))
    {
      PredefinedBrush brush =
        (PredefinedBrush)e.Data.GetData(typeof(PredefinedBrush));
      shape.Fill = brush.BrushColor;
    }
  }
  catch
  {
  }
}
```

The circle's brush is stored in the variable *currentFill*, and if the drag object has a *PredefindeBrush*, the circle's *Fill* property is set to it. The result is that when the cursor is moved over a circle, the consequence of a drop operation is shown by staining the circle with the brush that is

drag with the mouse. Moving the cursor out of the circle (without dropping) returns its color to the color that was saved:

```
protected override void OnDragLeave(DragEventArgs e)
{
  base.OnDragLeave(e);
  shape.Fill = currentFill;
}
```

Back there is the drop operation:

```
protected override void OnDrop(DragEventArgs e)
{
  base.OnDrop(e);
  try
  {
    if (e.Data.GetDataPresent(typeof(PredefinedBrush)))
    {
      PredefinedBrush brush =
        (PredefinedBrush)e.Data.GetData(typeof(PredefinedBrush));
      if (brush != null)
      {
        shape.Fill = brush.BrushColor;
        if (e.KeyStates.HasFlag(DragDropKeyStates.ControlKey))
          e.Effects = DragDropEffects.Copy;
        else
          e.Effects = DragDropEffects.Move;
      }
    }
    e.Handled = true;
  }
  catch
  {
  }
}
```

There is not much to notice, except that the drop operation is only performed if there is a *Brush* that can be dropped.

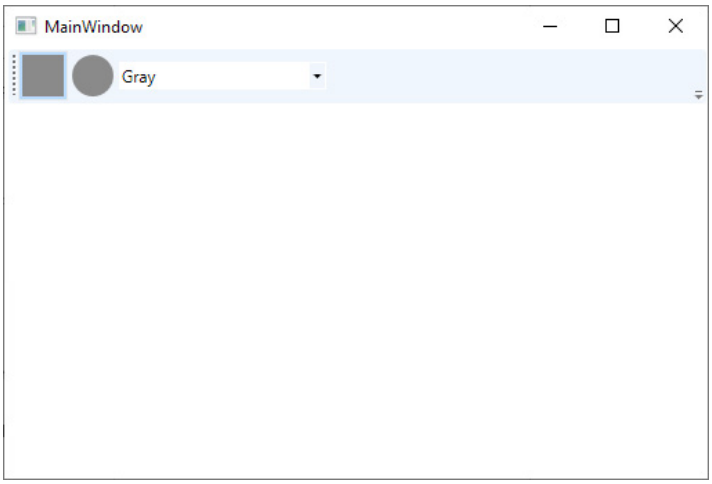
It was the *Circle* class, that is now a *CustomControl*, which supports drag and drop, so that you can drag a *Circle* and its *Brush*, and so you can drop a *Brush*.

The main window has the logic for dragging objects from the right list box as well as the drag and drop for the left list box, and finally the two panels for circles support for drag and drop. Part of the code is similar to the code from the previous example, I will not show the code here.

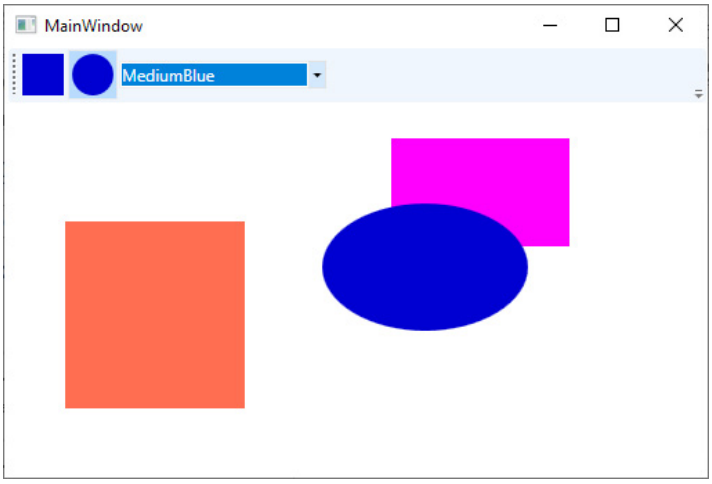
11.2 CHANGING THE SIZE OF AN OBJECT

The last example in this chapter deals with how to dynamically resize an object with the mouse, and how to move the object by dragging it with the mouse. This is typically done by selecting an object by drawing a thin frame around the object with some anchor points that can be grasped with the mouse. This operation is to some extent built into WPF in the form of an *Adorner*. It is a class, and one can best think of an *Adorner* object as a layer that is loaded over a component of the user interface and provides several anchor points that one can drag in with the mouse. Although WPF has an *Adorner* class, a specific *Adorner* must be written, and it can be quite complex, whatever it is in this case.

If you run the program, you will get a window, as shown on the next page. At the top there is a toolbar with two radio buttons and a combo box. The two radio buttons are displayed as resp. a *Rectangle* and an *Ellipse*. The combo box contains the same standard colors as in the previous example, and if you select a color, the color of the radio buttons changes accordingly.



In addition to the toolbar, the window has a *Canvas*, and if you double-click on this *Canvas*, an object corresponding to the radio button selected in the toolbar is inserted. The color of the figure is set to the color selected in the combo box. You can then select a shape, move it and resize it. If you right-click on a figure, it will be deleted. Below is an example where three figures have been inserted:



First, I took the *PredefinedBrushes* class from the previous project (and changed its namespace). It should be used as a data source for the combo box.

The construction of the user interface is simple:

```
<DockPanel>
  <ToolBar DockPanel.Dock="Top">
    <RadioButton Name="cmdRectangle" IsChecked="True">
      <Rectangle Name="rectangleShape" Fill="Gray" Width="30"
      Height="30" />
    </RadioButton>
    <RadioButton Name="cmdEllipse">
      <Ellipse Name="ellipseShape" Fill="Gray" Width="30" Height="30" />
    </RadioButton>
    <ComboBox Name="lstBrushes" Width="150"
    DisplayMemberPath="BrushName"
      SelectionChanged="lstBrushes_SelectionChanged" />
  </ToolBar>
  <Canvas Name="canvas" Background="White" />
</DockPanel>
```

The only thing to note is that the combo box processes an event to update the color of the two Shape objects:

```
private void lstBrushes_SelectionChanged(object sender,
            SelectionChangedEventArgs e)
{
  try
  {
    Brush brush = ((PredefinedBrush)lstBrushes.SelectedItem).BrushColor;
    rectangleShape.Fill = brush;
    ellipseShape.Fill = brush;
  }
  catch
  {
  }
}
```

Next, an *Adorner* class must be written, and that is where all the work lies. The window class defines this time several variables:

```
public partial class MainWindow : Window
{
  private AdornerLayer adorner;
  private bool isDown;
  private bool isDragging;
  private bool isSelected = false;
  private UIElement selectedElement = null;
  private Point start;
  private double left;
  private double top;
```

which is used for the following:

- *adorner*, reference to a current *Adorner* object
- *isDown*, keeps track of whether the mouse button is held down
- *isDragging*, keeps track of whether a drag operation is in progress
- *isSelected*, keeps track of whether a *Shape* object is selected
- *selectedElement*, refers to the object that is selected
- *start*, is the starting position of the object before it is moved
- *left* and *top*, indicate how much the object has moved relative to start

When the program starts, it processes the *Loaded* event:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  lstBrushes.ItemsSource = PredefinedBrushes.Brushes;
  lstBrushes.SelectedIndex = PredefinedBrushes.GetIndex("Gray");
  MouseLeftButtonDown += Window_MouseLeftButtonDown;
  MouseLeftButtonUp += DragFinished;
  MouseMove += Window_MouseMove;
  MouseLeave += Window_MouseLeave;
  canvas.PreviewMouseLeftButtonDown += canvas_
  PreviewMouseLeftButtonDown;
  canvas.PreviewMouseLeftButtonUp += DragFinished;
  canvas.MouseLeftButtonDown += AddShape;
  canvas.MouseRightButtonDown += RemoveShape;
}
```

The first two statements bind the combo box to the standard *Brush* objects. The rest of the statements define the event handlers that the program will need. I start from the back. The left click button handler is used to add a *Shape* to the window:

```
private void AddShape(object sender, MouseButtonEventArgs e)
{
  if (e.ClickCount == 2)
  {
    Shape shape;
    if (cmdRectangle.IsChecked == true)
      shape = new Rectangle { Height = 40, Width = 80 };
    else
      shape = new Ellipse { Height = 50, Width = 50 };
    shape.Fill = ((PredefinedBrush)lstBrushes.SelectedItem).BrushColor;
    Canvas.SetLeft(shape, e.GetPosition(canvas).X);
    Canvas.SetTop(shape, e.GetPosition(canvas).Y);
    canvas.Children.Add(shape);
  }
}
```

A Shape object is created, and depending on which radio button is pressed, a *Rectangle* or an *Ellipse* is created. Next, set the color corresponding to the value of the combo box, and finally attached properties for the figure's location in the window are set. Note that the

location is determined by the position of the mouse. Finally, the figure is added as a child to the *Canvas* in the window.

The last event handler regarding *Deleting* a *Shape*:

```
private void RemoveShape(object sender, MouseButtonEventArgs e)
{
  try
  {
    Point pt = e.GetPosition((Canvas)sender);
    HitTestResult result = VisualTreeHelper.HitTest(canvas, pt);
    if (result != null) canvas.Children.Remove(result.VisualHit as
    Shape);
  }
  catch
  {
  }
}
```

The task is primarily to determine whether a figure has been clicked, and this is done as previously shown with the static method *HitTest()* in the class *VisualTreeHelper*. If you find an object, it is removed from the *Canvas* object's *Children* collection.

All the other event handlers have to do with drag of objects, but first a small help method that stops a drag operation:

```
private void StopDragging()
{
  if (isDown)
  {
    isDown = false;
    isDragging = false;
  }
}
```

and which does nothing but set the two flags to *false*. This method is called if the user releases the mouse, or runs the mouse outside the window:

```
MouseLeftButtonUp += DragFinished;
MouseLeave += Window_MouseLeave;
canvas.PreviewMouseLeftButtonUp += DragFinished;
```

The event that starts a drag operation is the following:

```
private void canvas_PreviewMouseLeftButtonDown(object sender,
          MouseButtonEventArgs e)
{
  if (isSelected)
  {
    isSelected = false;
    if (selectedElement != null)
    {
      adorner.Remove(adorner.GetAdorners(selectedElement)[0]);
      selectedElement = null;
    }
  }
  if (e.Source != canvas)
  {
    isDown = true;
    start = e.GetPosition(canvas);
    selectedElement = e.Source as UIElement;
    left = Canvas.GetLeft(selectedElement);
    top = Canvas.GetTop(selectedElement);
    adorner = AdornerLayer.GetAdornerLayer(selectedElement);
    adorner.Add(new ResizingAdorner(selectedElement));
    isSelected = true;
    e.Handled = true;
  }
}
```

If *isSelected* is *true*, it means that an object is already selected with an *Adorner*, and this selection must then be removed and the selected object must be released by the *Adorner* object. If the object that the mouse is pointing to is not the *Canvas* object itself, it must be marked as the object of a new drag process. It includes a part:

- *isDown* is set to *true* to control that a drag process is in progress
- *start* is set to the point that the mouse points to (relative to the *Canvas* object)
- *selectedElement* is set to refer to the element to which the drag process relates
- *left* and *top* are set to the coordinates of the upper left corner of the object

- add an *Adorner* object is created for the operation
- the current object must be associated with this *Adorner*
- finally set *isSelected* to *true*

There are two additional event handlers. The first occurs if the mouse is moved over the window. If a drag operation is started (*isDown* is *true*), check whether the *isDragging* flag should be set, which is done by reading the mouse coordinates and measuring them in relation to the start position. If the difference is large enough - expressed by the constant *MinimumVerticalDragDistance* - the flag is set and the object is moved by calculating a new position:

```
private void Window_MouseMove(object sender, MouseEventArgs e)
{
  if (isDown)
  {
    Point p = Mouse.GetPosition(canvas);
    if (!isDragging && ((Math.Abs(p.X - start.X) >
        SystemParameters.MinimumHorizontalDragDistance) ||
        (Math.Abs(p.Y - start.Y) > SystemParameters.
        MinimumVerticalDragDistance)))
      isDragging = true;
    if (isDragging)
    {
      Canvas.SetTop(selectedElement, p.Y - (start.Y - top));
      Canvas.SetLeft(selectedElement, p.X - (start.X - left));
    }
  }
}
```

The last event handler regarding the drag operation concerns mouse click outside the *Canvas* object. In this case, a possible drag operation is completed.

Finally, there is the *Adorner* object, which is relatively complex:

```
public class ResizingAdorner : Adorner
{
  private Thumb topLeft, topRight, bottomLeft, bottomRight;
  private VisualCollection children;

  public ResizingAdorner(UIElement adornedElement) :
  base(adornedElement)
  {
    children = new VisualCollection(this);
    BuildCorner(ref topLeft, Cursors.SizeNWSE);
    BuildCorner(ref topRight, Cursors.SizeNESW);
    BuildCorner(ref bottomLeft, Cursors.SizeNESW);
    BuildCorner(ref bottomRight, Cursors.SizeNWSE);
    bottomLeft.DragDelta += BottomLeft;
    bottomRight.DragDelta += BottomRight;
    topLeft.DragDelta += TopLeft;
    topRight.DragDelta += TopRight;
  }

  private void BottomRight(object sender, DragDeltaEventArgs args)
  {
    FrameworkElement element = AdornedElement as FrameworkElement;
    Thumb thumb = sender as Thumb;
    if (element == null || thumb == null) return;
    FrameworkElement parent = element.Parent as FrameworkElement;
    SetSize(element);
    element.Width = Math.Max(element.Width + args.HorizontalChange,
      thumb.DesiredSize.Width);
    element.Height = Math.Max(args.VerticalChange + element.Height,
      thumb.DesiredSize.Height);
  }

  private void TopRight(object sender, DragDeltaEventArgs args)
  {
    ...
  }

  private void TopLeft(object sender, DragDeltaEventArgs args)
  {
    ...
  }
```

```
private void BottomLeft(object sender, DragDeltaEventArgs args)
{
  ...
}

protected override Size ArrangeOverride(Size finalSize)
{
  double w1 = AdornedElement.DesiredSize.Width;
  double h1 = AdornedElement.DesiredSize.Height;
  double w2 = this.DesiredSize.Width;
  double h2 = this.DesiredSize.Height;
  topLeft.Arrange(new Rect(-w2 / 2, -h2 / 2, w2, h2));
  topRight.Arrange(new Rect(w1 - w2 / 2, -h2 / 2, w2, h2));
  bottomLeft.Arrange(new Rect(-w2 / 2, h1 - h2 / 2, w2, h2));
  bottomRight.Arrange(new Rect(w1 - w2 / 2, h1 - h2 / 2, w2, h2));
  return finalSize;
}

private void BuildCorner(ref Thumb corner, Cursor cursor)
{
  if (corner != null) return;
  corner = new Thumb();
  corner.Cursor = cursor;
  corner.Height = corner.Width = 8;
  corner.Opacity = 0.50;
  corner.Background = new SolidColorBrush(Colors.MediumBlue);
  children.Add(corner);
}

private void SetSize(FrameworkElement element)
{
  if (element.Width.Equals(Double.NaN))
    element.Width = element.DesiredSize.Width;
  if (element.Height.Equals(Double.NaN))
    element.Height = element.DesiredSize.Height;
  FrameworkElement parent = element.Parent as FrameworkElement;
  if (parent != null)
  {
    element.MaxHeight = parent.ActualHeight;
```

```
      element.MaxWidth = parent.ActualWidth;
    }
  }

  protected override int VisualChildrenCount
  {
    get { return children.Count; }
  }

  protected override Visual GetVisualChild(int index)
  {
    return children[index];
  }
}
```

In short, the meaning is as follows:

- The first four *Thump* variables represent four points that can be grasped with the mouse to drag the size of the object.
- The last variable is a collection that contains references to the visual objects that the operation includes that is the objects to be moved or resized.
- The constructor creates all objects. It happens among other by calling the method *BuildCorner()*. There is not much mystery in that method, but you should note that this is where you define what the four grip points should look like. It is also by calling *BuildCorner()* in the constructor that you associate the cursor to be used. Finally, the constructor associates event handlers for the four grip points.
- There are also a few other important help methods. *SetSize()* must ensure that the width and height of the object that is resized are given the correct values. *ArrangeOverride()* must ensure that the *Adorner* object itself is displayed with the correct size. It is a method that is overridden from the base class.
- Then there are the four event handlers, which are where it all happens and where you calculate the new size. I only showed the code for the first one.
- Finally, there are two overrides of methods from the base class. They must be taken into account for the runtime system in order for the object to be rendered correctly.

12 ANIMATIONS

As the last chapter I will look at animations, and WPF has a very large number of classes, which make it relatively easy to animate objects in a window, at least so long that there are simple components such as buttons, *Shape* objects, etc. The classes for animations can be found in the namespace:

System.Windows.Media.Animation

and there are actually quite a few. This is because there are three kinds of animations, and for each kind there is a class for each type that one must be able to animate, e.g. *double*, *int*, *Color*, *string*, etc. The three kinds of animations are:

1. The first category animates a value continuously over an interval.
2. The second category animates a value over a discrete set of values.
3. The third category animates an object along a curve.

In general, an animation regardless of category can be associated with any dependency property, and the individual animations can relate to very different values, which vary the corresponding classes a lot, but they all have the following three properties:

To, which represent the end value of the animation
From, which represents the initial value of the animation
By, which indicates in which step animation should change the animated value

All animation classes inherit a common basic class, which defines the following properties:

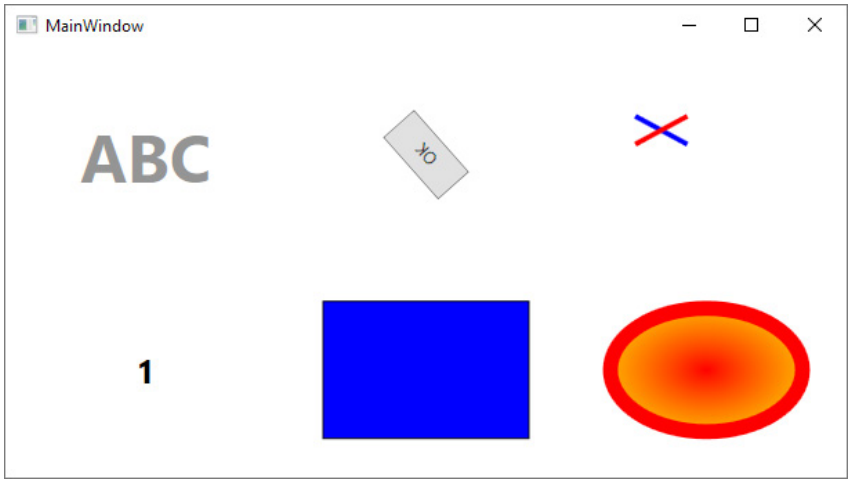
1. *AutoReverse*, where you can specify whether the animation should repeat itself in reverse order (true).
2. *BeginTime*, where you can specify how long it takes before the animation starts.
3. *Duration*, where you specify how long the animation should take.
4. *FillBehavior*, where you can specify what should happen when the animation is completed.
5. *RepeatBehavior*, where you can specify whether the operation should repeat itself.

With these introductory remarks, it is time to show what needs to be done and what needs to be written. I will show four examples which relate to simple animations and you will see that a lot has to be written and many different classes have to be used and it is quite difficult to remember how to write an animation.

12.1 ANIMATIONS IN XML

You can define animations both in XML and in C #. If you do it in XML you have to write less, but if you have to do it manually as in this example, in my opinion it is not absolutely easier, but it probably has something to do with experience.

The program opens a window with a Grid, which has two rows and three columns, and each of the six cells starts an animation, which runs until the program terminates:



The effect is as follows:

1. ABC is a *Label*, and the program animates its transparency and thus its *Opacity* property. The result is that the text changes from completely black to hidden and back to black, and this operation is repeated.
2. 1 is a *TextBlock*, and you animate the text (the component's *Text* property) so that it runs through a sequence of numbers (from blank to the number 123456789). It is thus an example of an animation that runs through a discreet family of values.
3. The button animates a rotation about the center, and the result is a button that rotates constantly, and you have to observe that you can still click on it.
4. The rectangle switches between three colors blue, green and red, and there is a shift every 2 seconds.
5. The cross animates a scaling where the figure size is scaled from full size to nothing. Like the button, it is an example of animation of a transformation.
6. The ellipse animates its *StrokeThickness* property and thus the edge of the ellipse.

It's all written exclusively in XML, and it obviously takes up a lot of space. I want to show the code of three figures and the button. I start with the *Label* component:

```
<Label Name="lblText" Content="ABC" ... >
  <Label.Triggers>
    <EventTrigger RoutedEvent="Label.Loaded">
      <BeginStoryboard>
        <Storyboard TargetProperty="Opacity">
          <DoubleAnimation From="1" To="0" Duration="0:0:5"
            RepeatBehavior="Forever" AutoReverse="True" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Label.Triggers>
</Label>
```

An animation is defined with a Trigger, and in this case (and incidentally also in all the other cases) the animation must be started when the program (component) is loaded. Next, the animation is defined as a *StoryBoard* (it can consist of several). In this case, there is one that pertains to the component's *Opacity* property. The animation itself is defined as a *DoubleAnimation* animation, which is a class for animating a *double* (the *Opacity* of the component). The characteristics of the animation are:

1. the value - its *Opacity* - must be animated from 1 to 0
2. the duration is 5 seconds
3. the animation must be repeated, but every other time in reverse order

It is thus an example of an animation that animates a value continuously over an interval. When you see the animation, it is not so difficult to understand (and it is also a simple animation, which only relates to a single property), but it can be difficult to remember what it is that you have to write.

I will then look at the rectangle:

```
<Rectangle Name="rect" Stroke="Black" StrokeThickness="1" Fill="Blue"
... >
  <Rectangle.Triggers>
    <EventTrigger RoutedEvent="Rectangle.Loaded">
      <BeginStoryboard>
        <Storyboard TargetProperty="Fill.Color">
          <ColorAnimationUsingKeyFrames Duration="0:0:6"
            RepeatBehavior = "Forever">
            <DiscreteColorKeyFrame Value="Blue" KeyTime="0:0:0" />
            <DiscreteColorKeyFrame Value="Green" KeyTime="0:0:2" />
            <DiscreteColorKeyFrame Value="Red" KeyTime="0:0:4" />
          </ColorAnimationUsingKeyFrames>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Rectangle.Triggers>
</Rectangle>
```

Again, the animation is defined with a trigger, and this time the value being animated is the *Fill.Color* of the rectangle and thus a color that is animated over three discrete values. This time the animation has the type *ColorAnimationUsingKeyFrames*, where a *KeyFrame* defines a value, as well as the time when the value should change. Otherwise, the principle is the same as in the first example.

Then there is the button:

```
<Button Name="cmdOk" Content="OK" ... Click="cmdOk_Click">
  <Button.RenderTransform>
    <RotateTransform Angle="0" CenterX="30" CenterY="15"/>
  </Button.RenderTransform>
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation From="0" To="360" Duration="0:0:2"
            RepeatBehavior="Forever" Storyboard.TargetName="cmdOk"
            Storyboard.TargetProperty=
            "(Button.RenderTransform).(RotateTransform.Angle)" ></
            DoubleAnimation>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

First, a rotation is defined which must rotate around the center of the button. The rotation is in many ways defined as in the first example in the form of a *DoubleAnimation*, which must animate over the range from 0 to 360, and the animation must take 2 seconds and otherwise repeat itself forever. The value to be animated this time is not a property of the component itself, but rather of its transformation. It requires a special and slightly complicated syntax.

Finally, there is the cross, which is two lines in a Canvas:

```
<Canvas Name="canvas" Grid.Column="2" Grid.Row="0" Margin="50" >
  <Line Name="line1" Stroke="Blue" X1="0" Y1="0" StrokeThickness="10"
    X2="{Binding ElementName=canvas, Path=ActualWidth}"
    Y2="{Binding ElementName=canvas, Path=ActualHeight}">
    <Line.RenderTransform>
      <ScaleTransform ScaleX="1" ScaleY="1" />
    </Line.RenderTransform>
    <Line.Triggers>
      <EventTrigger RoutedEvent="Line.Loaded">
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation From="1" To="0" Duration="0:0:2"
              RepeatBehavior="Forever" Storyboard.TargetName="line1"
              Storyboard.TargetProperty=
              "(Line.RenderTransform).(ScaleTransform.ScaleX)" />
```

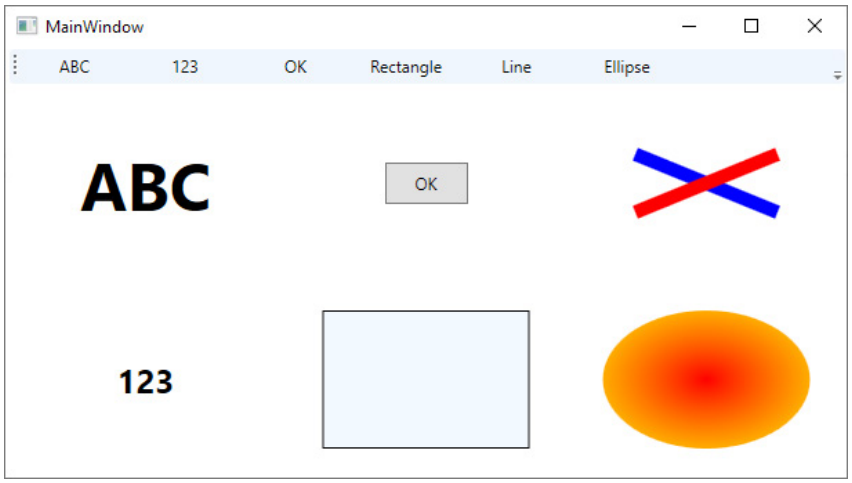
```
          <DoubleAnimation From="1" To="0" Duration="0:0:2"
            RepeatBehavior="Forever" Storyboard.TargetName="line1"
            Storyboard.TargetProperty=
            "(Line.RenderTransform).(ScaleTransform.ScaleY)" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Line.Triggers>
</Line>
<Line Name="line2" Stroke="Red" X1="0"
  Y1="{Binding ElementName=canvas, Path=ActualHeight}"
  StrokeThickness="10"
  X2="{Binding ElementName=canvas, Path=ActualWidth}" Y2="0" >
  <Line.RenderTransform>
    <ScaleTransform ScaleX="1" ScaleY="1" />
  </Line.RenderTransform>
  <Line.Triggers>
    <EventTrigger RoutedEvent="Line.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation From="1" To="0" Duration="0:0:2"
            RepeatBehavior="Forever" Storyboard.TargetName="line2"
            Storyboard.TargetProperty=
            "(Line.RenderTransform).(ScaleTransform.ScaleX)" />
          <DoubleAnimation From="1" To="0" Duration="0:0:2"
            RepeatBehavior="Forever" Storyboard.TargetName="line2"
            Storyboard.TargetProperty=
            "(Line.RenderTransform).(ScaleTransform.ScaleY)" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Line.Triggers>
</Line>
</Canvas>
```

Each line is animated individually with a scaling. There are thus two animations. For each animation, there are two values (scaling in each of the two directions) that must be animated. The animation's storyboard therefore defines two animations. Except that it takes up a little more space and that there are two values that need to be changed, the principle is the same as for the animation of the button.

The example shows how it is relatively simple to animate properties at the components of the user interface.

12.2 ANIMATIONS IN C#

It's basically the same example (same animations) as above, but this time the animations are written in C#. If you run the program, you get the following window:



The difference is that this time the window at the top has a toolbar with six *CheckBox* components, and they are used to start and stop one of the six animations. Another difference is that this time the rectangle switches between all the standard colors, and finally the cross is scaled around the center.

The program uses the class *PredefinedBrushes*, which represents the standard colors.

The animations themselves are defined in the event handlers for the six *CheckBox* components, and I will show the code for the same four animations as in the previous example. I omit to show the XML code as it contains nothing new and nothing at all to do with animations.

The event handler for the first *CheckBox* is as follows and defines the animation of the *Label* component:

```
private void cmdLabel_Click(object sender, RoutedEventArgs e)
{
  if (cmdLabel.IsChecked == true)
  {
    DoubleAnimation animation = new DoubleAnimation();
    animation.From = 1.0;
    animation.To = 0.0;
    animation.AutoReverse = true;
    animation.RepeatBehavior = RepeatBehavior.Forever;
    animation.Duration = new Duration(TimeSpan.FromSeconds(10));
    lblText.BeginAnimation(Label.OpacityProperty, animation);
  }
  else lblText.BeginAnimation(Label.OpacityProperty, null);
}
```

If you compares with the corresponding animation written in XML, the code is easy enough to understand. A *DoubleAnimation* object is created and its attributes are appropriately defined. The most important thing is how to define *Duration* as a *TimeSpan* object and how to define what kind of property you want to animate. Finally, note how to stop an animation by setting the animation of that property to null.

Below is the animation of the rectangle:

```
private void cmdRect_Click(object sender, RoutedEventArgs e)
{
  if (cmdRect.IsChecked == true)
  {
    ObjectAnimationUsingKeyFrames animation = new
    ObjectAnimationUsingKeyFrames();
    animation.RepeatBehavior = RepeatBehavior.Forever;
    TimeSpan span = new TimeSpan();
    foreach (PredefinedBrush brush in PredefinedBrushes.Brushes)
    {
      animation.KeyFrames.Add(new DiscreteObjectKeyFrame(brush.
      BrushColor, span));
      span = span.Add(new TimeSpan(0, 0, 0, 0, 500));
    }
    rect.BeginAnimation(Rectangle.FillProperty, animation);
  }
  else rect.BeginAnimation(Rectangle.FillProperty, null);
}
```

As in the previous example, it is a discrete animation, but this time there are many values, namely one for each brush in *PredefinedBrushes*. The type is *ObjectAnimationUsingKeyFrames*, and the *Rectangle* object's *Fill* property is animated. Corresponding to the discrete values, it is equivalent to changing the brush every half second. The rectangle will therefore run through all standard colors.

The last two animations (which I will show here) relate to transformations. The program defines two variables for this:

```
public partial class MainWindow : Window
{
  private RotateTransform rotation = null;
  private ScaleTransform scale = null;
```

The reason for these variables is that it must be possible to stop the animation and thus set the animation of the animated value to zero. The animation of the button is written as follows:

```
private void cmdButton_Click(object sender, RoutedEventArgs e)
{
  if (cmdButton.IsChecked == true)
  {
    DoubleAnimation animation = new DoubleAnimation();
    animation.Duration = new Duration(TimeSpan.FromSeconds(5));
    animation.From = 0;
    animation.To = 360;
    animation.RepeatBehavior = RepeatBehavior.Forever;
    rotation = new RotateTransform();
    rotation.CenterX = 30;
    rotation.CenterY = 15;
    cmdOk.RenderTransform = rotation;
    rotation.BeginAnimation(RotateTransform.AngleProperty, animation);
  }
  else rotation.BeginAnimation(RotateTransform.AngleProperty, null);
}
```

It is also similar to what is written in XML, but you should note that the animation is defined for the property *Angle* of the transformation and not the button.

Finally, there is the code for the animation of the cross:

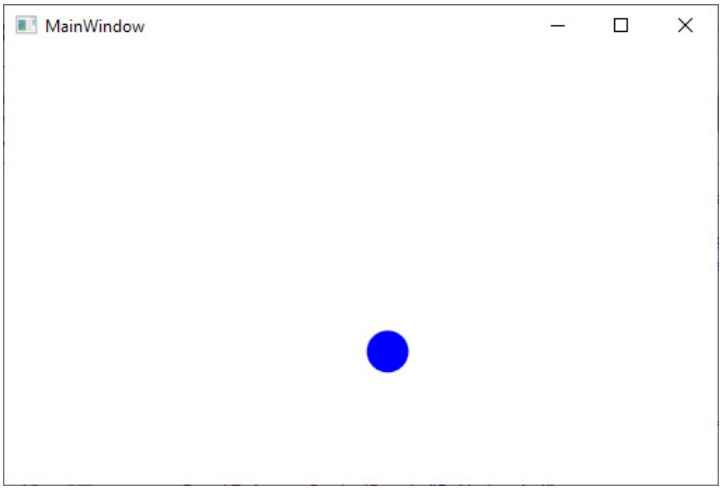
```
private void cmdLine_Click(object sender, RoutedEventArgs e)
{
  if (cmdLine.IsChecked == true)
  {
    DoubleAnimation animation = new DoubleAnimation();
    animation.Duration = new Duration(TimeSpan.FromSeconds(5));
    animation.From = 1;
    animation.To = 0;
    animation.AutoReverse = true;
    animation.RepeatBehavior = RepeatBehavior.Forever;
    scale = new ScaleTransform();
    scale.CenterX = canvas.ActualWidth / 2;
    scale.CenterY = canvas.ActualHeight / 2;
    line1.RenderTransform = scale;
    line2.RenderTransform = scale;
    scale.BeginAnimation(ScaleTransform.ScaleXProperty, animation);
    scale.BeginAnimation(ScaleTransform.ScaleYProperty, animation);
  }
  else
  {
    scale.BeginAnimation(ScaleTransform.ScaleXProperty, null);
    scale.BeginAnimation(ScaleTransform.ScaleYProperty, null);
  }
}
```

It is defined a little differently and a little simpler than in XML, as it is the same transformation that is attached to both lines. Also note that the scaling is performed with the center of the Canvas object determined dynamically. Also note that there are two properties that need to be animated.

12.3 PATH ANIMATION IN XML

The last example is a classic example where a point in the form of an *Ellipse* is animated by moving along a curve, which here is a Bezier curve (the curve is not displayed). It is thus an example of an animation where an object moves along a curve, and as with the first case, I show both an XML example and a C# example.

In this example, the animation is written exclusively as XML:




```
<Canvas>
  <Path Fill="Blue" Margin="20">
    <Path.Data>
      <EllipseGeometry x:Name="ellip" Center="10,100" RadiusX="15"
      RadiusY="15" />
    </Path.Data>
    <Path.Triggers>
      <EventTrigger RoutedEvent="Path.Loaded">
        <BeginStoryboard>
          <Storyboard TargetName="ellip" TargetProperty="Center">
            <PointAnimationUsingPath Duration="0:0:3"
            RepeatBehavior="Forever"
              AutoReverse="True" >
              <PointAnimationUsingPath.PathGeometry>
                <PathGeometry
                  Figures="M 10,100 C 35,0 135,0 160,100 180,190 285,300
                  470,0" />
              </PointAnimationUsingPath.PathGeometry>
            </PointAnimationUsingPath>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger>
    </Path.Triggers>
  </Path>
</Canvas>
```

You should note that the code does not actually take up much space and that the animation is defined in exactly the same way as in the first example. The panel is a *Canvas* with a *Path* whose geometry is an ellipse (and it is of course easy to define a more advanced geometry if desired). Next, the animation is defined as a *Trigger*, and a *Storyboard* tells that it is the center

of the ellipse that needs to be animated. The type of animation is *PointAnimationUsingPath*, which means that the center of the ellipse must be animated along the subsequent curve. It is defined using a special encoding. Here, M stands for Move, and 10,100 is the starting point. C means that the remaining points define a curve, which here is a *PolyBezierSegmet* consisting of two segments. I will not go into details here with this encoding, but the important thing is that Visual Studio can interpret the result.

You should primarily be aware that path animations are defined - well enough with other classes - in exactly the same way as other animations.

12.4 PATH ANIMATION IN C#

The last example is completely identical to the previous example and gives the same result. Merely, the animation this time is written in C#, and it's actually complex.

The window itself consists only of a Canvas, which should show the animation:

```
<Canvas Name="canvas" Margin="20" >
</Canvas>
```

Then there is the code, which uses a number of new types:

```
private void CreateAnimation()
{
  Path path = CreatePath();
  canvas.Children.Add(path);
  PointAnimationUsingPath animation = new PointAnimationUsingPath();
  animation.PathGeometry = CreatePathGeometry();
  animation.Duration = TimeSpan.FromSeconds(3);
  animation.RepeatBehavior = RepeatBehavior.Forever;
  animation.AutoReverse = true;
  Storyboard.SetTargetName(animation, "Ellip");
  Storyboard.SetTargetProperty(animation,
    new PropertyPath(EllipseGeometry.CenterProperty));
  Storyboard story = new Storyboard();
  story.RepeatBehavior = RepeatBehavior.Forever;
  story.AutoReverse = true;
  story.Children.Add(animation);
  path.Loaded += delegate(object sender, RoutedEventArgs e) { story.
  Begin(this); };
}
```

```
private PathGeometry CreatePathGeometry()
{
  PathGeometry path = new PathGeometry();
  path.Figures.Add(CreateFigure(CreateBezier()));
  path.Freeze();
  return path;
}

private PathFigure CreateFigure(PolyBezierSegment segment)
{
  PathFigure fig = new PathFigure();
  fig.StartPoint = new Point(10, 100);
  fig.Segments.Add(segment);
  return fig;
}

private PolyBezierSegment CreateBezier()
{
  PolyBezierSegment segment = new PolyBezierSegment();
  segment.Points.Add(new Point(35, 0));
  segment.Points.Add(new Point(135, 0));
  segment.Points.Add(new Point(160, 100));
  segment.Points.Add(new Point(180, 190));
  segment.Points.Add(new Point(285, 300));
  segment.Points.Add(new Point(470, 0));
  return segment;
}

private Path CreatePath()
{
  EllipseGeometry ellip = new EllipseGeometry(new Point(10, 100), 15,
  15);
  RegisterName("Ellip", ellip);
  Path path = new Path();
  path.Data = ellip;
  path.Fill = Brushes.Blue;
  return path;
}
```

If I start from the back, then the last method creates a *Path* with the ellipse and there is nothing new. The next one (still from behind) creates the Bezier segment with the same points as in the XML example. You have to write a lot, but other than that, there is nothing new here either. The next two methods are again used to create the path along which the figure is to be animated. You should note that here are some new types in the form of

PathFigure and *PathGeometry* (which were also used in the XML example). Also note that the geometry is frozen:

```
path.Freeze();
```

The reason for this is performance, which makes it easier for WPF to render the graphics.

The above methods are used in *CreateAnimation()* to create the animation. First, a *Path* is created, which is added to the window's *Canvas*. Next, the animation and a corresponding *StoryBoard* are defined, and finally an anonymous event handler is defined to start the animation.

Here, too, the similarities with the XML version are clear, but many lines of code have to be written, and the code is far from obvious.

13 A CHART LIBRARY

By a chart library, one usually understand a family of classes that you can use in a program to represent numbers using graphs. Examples are histograms, bar charts, pie charts, and more. A good example of using such tools is a spreadsheet that offers different forms of graphical representation of numbers. The aim of the following project is to develop such a library as well as write a program that can illustrate how the library can be used. There are many such libraries, and they differ as to which graphs there are, how "nice" the graphs are, which options are possible, and also as the most important how easy the library is to use.

The purpose of the project is not so much the value of the library (there exists a lot of that kind), but it is a very good exercise in the use of the above ways of drawing in the window in a WPF program, especially if the aim is to develop "nice" graphs (whatever it may be) and if you focus on developing a user-friendly library. Regarding the latter, keep in mind that user-friendliness is not the same as the greatest possible flexibility, but to a greater extent that the library is easy to understand and use. Finally, it is extremely important that such a library is robust and that the graphs behave sensibly if used on an incomplete or inappropriate data basis, and it is actually not so easy.

In the following, I would not focus on the process, but instead what is made and how the library is used. Of course, it is recommended that you study the finished code thoroughly and there is a lot of it. In reality, there are two projects where one is the class library, while the other is a test program.

13.1 THE LIBRARY

Basically, a graph or chart should visualize numbers associated with categories (labels). A classic example is year numbers, where there are associated a number with each year. The current library should basically offer the following graphs:

- *Line*, where a graph consists of straight lines between points
- *Histogram* or a *bar chart*, which is probably the most commonly used graph
- *Circle* or pie chart

Each of these three basic graphs is available in several variants. A line graph has two variants

- *Curve*, where the graph is instead drawn as a soft curve
- *Plot* where the graph is drawn as points - a point for each category (label), which may be associated with thin straight lines

A bar chart is drawn as default with vertical bars, but there is a variant in which the bars instead are drawn horizontally, and finally there is a variation of both the vertical and horizontal bar charts with a simple 3D effect.

Finally, there is a variant of the pie chart, where the chart is drawn as a speedometer - just to point out that it is only the imagination (and the ability to draw nice graphs) that limit the graphs, that such a library should contain.

The library thus contains 9 basic (or simple) graphs, which are defined by the following type:

```
public enum GraphType { LINE, CURVE, PLOT, VBAR, HBAR, VBAR3D,
HBAR3D, PIE, SPEED }
```

In addition, for a plot graph, you can indicate which shape is used to represent the graph's points (plots), and these options are defined by the following type, where the names indicate which figure is used:

```
public enum PlotType { CROSS, STAR, OPENCIRCLE, OPENSQUARE,
                       OPENTRIANGLE,
                       OPENRHOMBE, CLOSEDRHOMBE, CLOSEDCIRCLE,
                       CLOSEDSQUARE, CLOSEDTRIANGLE }
```

A graph as above is called simple (or single-valued) corresponding to the fact that in principle it is defined by two arrays:

1. *labels* that is an array of strings and represent the categories and hence the independent variable or the the x axis
2. v*alues* which is an array of numbers of the same length as *labels* and represent the dependent variable or the y axis

A simple graph is thus a graph for a mathematical function, and in addition to the two arrays, a graph has attached other parameters, which indicates how the graph is to be drawn.

A graph can also be multi-valued, which means that each label has several numbers and the difference is that the array *values* in principle are a 2-dimensional array. A multi-valued graph is as default drawn as several simple graphs after each other (that is, as a series of graphs). A multi-valued graph may in particular be merged, which means that the individual

graphs are drawn in the same coordinate system - to the extent that it makes sense. This is exactly the part that makes the development of the following library both complex and comprehensive.

Basically, a graph is defined using two data structures. The rationale for these data structures is that the definition of a graph has a number of properties, all of which have to do with how the graph is to be drawn. A chart is basically defined by two data structures which are relatively comprehensive as they define how the graph should be drawn. Although the code takes up a lot of space, I have chosen to show most of the code for both data structures, as the comments explain the individual properties and thus in general how a chart is defined and what settings are possible. The first is called *SingleValues* and represents the y values of a simple graph:

```
/// <summary>
/// Represent the y-values som a single graph. It is basically an
    array with
/// graph values as well as a number of properties that primarily
    indicate how
/// the y-axis should be drawn. Apart from the function values, all
    properties
/// have a default value.
/// </summary>
public class SingleValues
{
  ...
  /// <summary>
  /// Type of this graph. Default is VBAR and the a vertical bar
graph.
  /// </summary>
  public GraphType Type { get; set; }

  /// <summary>
  /// Label to (name of) the y axis, which is drawn above (to the
      right of)
  /// the y-axis. Default is blank.
  /// </summary>
  public string Label { get; set; }
```

```
/// <summary>
/// Description of (title for) this graph, which is drawn above the
    graph.
/// Default is blank.
/// </summary>
public string Header { get; set; }

/// <summary>
/// Minimum value to y axis (default = CROSS, and then the value is
/// determined from the graph's values)
/// </summary>
public double Begin { get; set; }

/// <summary>
/// Maximum value to y axis (default = 0, and then the value is
    determined
/// from the graph's values)
/// </summary>
public double End { get; set; }

/// <summary>
/// Number of intervals on the y axis (default = 0, and then the
    value is
/// determined from the graph's values)
/// </summary>
public int Points { get; set; }

/// <summary>
/// Number of decimals for the y-axis (default = 0, and then the
    value is
/// determined from the graph's values)
/// </summary>
public int Dec { get; set; }

/// <summary>
/// The color used to draw the graph. The color must be one of 51
    colors
/// defined in the class Colors (the file MultiValues.cs).
/// </summary>
public Brush Color { get; set; }

/// <summary>
```

```
/// Shape used to draw a plot graph (default is null) and the value
    is only
/// used for a PLOTGRAPH)
/// </summary>
public PlotType Plot { get; set; }

/// <summary>
/// Special parameter whose application depends on the graph type.
    In the
/// current implementation of
/// the library the value is unused.
/// </summary>
public double Param1 { get; set; }

/// <summary>
/// Special parameter whose application depends on the graph type.
    In the
/// current implementation of
/// the library the value is unused.
/// </summary>
public double Param2 { get; set; }

/// <summary>
/// Special parameter whose application depends on the graph type
/// (default = false). In this implementation the value is only used
    by
/// a PLOT graph th indicate where to draw lines
/// between the plots.
/// </summary>
public bool IsOn { get; set; }

/// <summary>
/// The values for this graph representing as an array of the type
    double.
/// </summary>
public double[] Values { get; private set; }

/// <summary>
/// The number of values for this graph. The number of values must
    be the
/// same as the number
/// of labels for independent axis defined in the class MultiValues.
```

```
  /// </summary>
  public int Length
  {
    get { return Values.Length; }
  }

  /// <summary>
  /// Reade-write override of the index operator, so you can refer to
      the
  /// individual graph
  /// values when the graph has to be drawn.
  /// </summary>
  /// <param name="i">Index for the array Values</param>
  /// <returns>The graph's value at index i</returns>
  public double this[int i]
  {
    get { return Values[i]; }
    set { Values[i] = value; }
  }
}
```

The class *SimpleValues* only defines a single dependent variable. A graph that can be single-valued or multi-valued is defined as follows, and the other data structure used to define a chart is:

```
/// <summary>
/// Defines the data used to draw a chart. A chart consists as minimum
    of
///   labels for the independent axis (x-axis)
///   values for one or more graps (y-values)
/// A MultiValues object must the as minimum have two parameters
///   a list with SingleValues objects, one object for each graph
///   an array of labels for the independent axis
/// For each SingleValues must the length be equal to the number of
    labels for the
/// independens axis.
/// A MultiValues structure has also other properties which all have
    to do with how
/// the chart should be drawn.
/// </summary>
```

```
public class MultiValues
{
  ...
  /// <summary>
  /// Graphs for this multi graph. Each graph represents y-values for
      a graph.
  /// </summary>
  public List<SingleValues> Values { get; private set; }

  /// <summary>
  /// Label to the x-axis, default is blank. The label is drawn to
      the right of
  /// (at top of) the x-axis.
  /// </summary>
  public string Label { get; set; }

  /// <summary>
  /// Specifies whether to draw gitter lines (true that is default) or
      not
  /// (false).
  /// </summary>
  public bool HasGitter { get; set; }

  /// <summary>
  /// Indicates whether graphs should be merged into a single figure
      (false
  /// and is default).
  /// Not all graphs can be merged, and if not graphs in a chart are
      drawn
  /// a series side by side.
  /// The control supports the following merging of graphs as one
      graph:
  /// 1) VBAR, LINE, CURVE, PLOT (VBAR unstacked)
  /// 2) HBAR
  /// 3) VBAR stacked
  /// 4) VBAR3D
  /// 5) HBAR3D
  /// </summary>
  public bool IsMerged { get; set; }

  /// <summary>
```

```
/// Specifies whether all graphs must use the same y axis (true and
    is
/// default), the property is
/// only used for a merged graph.
/// </summary>
public bool HasSame { get; set; }

/// <summary>
/// Specifies whether BAR and BAR3D graphs should be stacked (true)
    or drawn
/// side by side (false that is default), the property is only used
    for a
/// merged graph
/// </summary>
public bool IsStacked { get; set; }

/// <summary>
/// Scaling in the x-axis (horizontal) direction, 1 = unscaled is
    default.
/// The value must be between 0.5 and 3.
/// </summary>
public double ScaleX { get; set; }

/// <summary>
/// Scaling in the y-axis (horizontal) direction, 1 = unscaled is
    default.
/// The value must be between 0.5 and 3.
/// </summary>
public double ScaleY { get; set; }

/// <summary>
/// Texts for the independent axis (x axis or label axis). All
    graphs uses
/// the same independent axis.
/// </summary>
public string[] Labels { get; set; }

/// <summary>
/// Used for a merged graph to calculate the minimum value for the
    y-axix.
/// </summary>
public double Minimum { get; set; }
```

```
  /// <summary>
  /// Used for a merged graph to calculate the maximum value for the
      y-axix.
  /// </summary>
  public double Maximum { get; set; }
}
```

The same file as the class *MultiValues* has a class *Colors* which defines colors used to draw the charts. The class is a simple class and is implemented as a singleton, and I will not show the code here.

As mentioned, there are 9 different graphs, as well as the possibility to merge multiple graphs in the same coordinate system. The graphs are drawn using *Drawer* classes, which are tasked with drawing a particular graph as well as capturing clicks with the mouse. If you click on a graph with the mouse, it must raise an event:

1. if clicking on a label (on the x-axis), the name is associated with the event
2. if clicking on the graph itself, the event is associated to both the name on the x-axis and the corresponding function value

An event has a value of the following type:

```
/// <summary>
/// Represent the value of a function for a given label on the
    independent
/// axis. The type is used to represent function values when double
    clicking
/// on a chart.
/// </summary>
public class FunctionValue
{
  public FunctionValue(string name, double? value)
  {
    Name = name;
    Value = value;
  }

  /// <summary>
  /// The value on the independent axis (the x-axis).
  /// </summary>
  public string Name { get; private set; }
```

```
  /// <summary>
  /// The function value which may be null if the object represents a
      double
  /// click on the axis.
  /// </summary>
  public double? Value { get; private set; }

  /// <summary>
  /// True if the value is not null.
  /// </summary>
  public bool HasValue
  {
    get { return Value != null; }
  }
}
```

Here is *value null* if a label is clicked. The reason is that a graph may be multi-valued and thus there may be several values attached to a particular label.

Next there is the component *Chart* which is a custom control and a control to be used in a window. Below I have shown part of the code, and the most important is the comments, which tells how the components works and should be used. For details you must visit the finish code.

```
/// <summary>
/// Class which define a chart as custom control. The class inherits
    the
/// class FrameworkElement. It is the basic base class of a WPF
    component and
/// defines all the basic properties. On the other hand, it is left to
    the
/// component itself to draw the component in the window.
/// </summary>
public class Chart : FrameworkElement
{
  /// <summary>
  /// Defines a property as a dependency property that defines the width
  /// of a border for the chart. The default value is 1.
  /// If you do not want a border you must set the value to 0.
  /// </summary>
  public static readonly DependencyProperty BorderStrokeProperty = ...
```

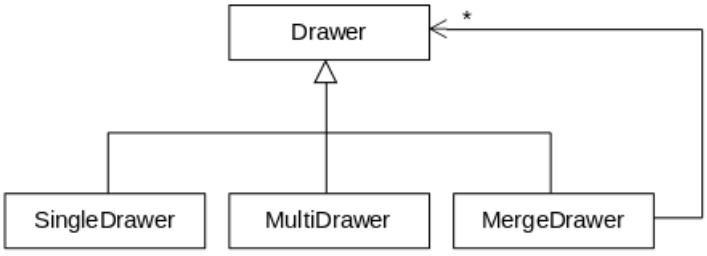
```
/// <summary>
/// Defines a property as a dependency property that defines the color
    of
/// a border. The default value is black.
/// </summary>
public static readonly DependencyProperty BorderColorProperty = ...

/// <summary>
/// Defines the font size as a dependency property. The font size
/// is used to draw all text in the chart.
/// </summary>
public static readonly DependencyProperty FontSizeProperty = ...

/// <summary>
/// Defines the font family as a dependency property. The font family
/// is used to draw all text in the chart.
/// </summary>
public static readonly DependencyProperty FontFamilyProperty = ...

/// <summary>
/// Defines a gap as a dependency property. A gap is used to
    separate graphs
/// in a chart with several graphs. The default value is 20.
/// </summary>
public static readonly DependencyProperty GapProperty = ...

/// <summary>
/// Defines a scaling property as a dependency property. ScaleX is
    used to
/// scale a graph in the independents axis direction. The default
    value is
/// 1 and means unscaled.
/// </summary>
public static readonly DependencyProperty ScaleXProperty = ...

/// <summary>
/// Defines a scaling property as a dependency property. ScaleY is
    used to
/// scale a graph in the dependents axis direction. The default
    value is 1
/// and means unscaled.
/// </summary>
```

```
public static readonly DependencyProperty ScaleYProperty = ...

/// <summary>
/// Event that fires when the user clicks on a graph.
/// </summary>
public event EventHandler<ChartEventArgs> ChartClicked;

private readonly MultiValues data; // data structure for this chart
private readonly List<Drawer> drawers; // drawer objects to draw
this chart

public Chart(MultiValues data)
{
  data.ScaleX = ScaleX;
  data.ScaleY = ScaleY;
  this.data = data;

  // set the font family to the font family for the window
  FontFamily = (FontFamily)GetValue(Window.FontFamilyProperty);

  // creates the drawers needed to draw this chart
  // these are utility methods in the name space Tools and the
     implementation
  // of these methods is all what this library is about
  drawers = DrawerFactory.CreateDrawer(data, FontFamily, FontSize);
}

// The method which draws the component using a DrawingContext,
   which
// is a class representing drawing tools to draw shapes in a
   window.
protected override void OnRender(DrawingContext drawingContext)
{
  // creates a Graphic object used to draw this chart
  // it is an encapsulation of a DrawingContext and the DpiFactor
     used
  // to compensate for antialiasing when drawing straight lines and
  // rectangles
  Graphics g = new Graphics { Dc = drawingContext, DpiFactor = ...
  };
  double height = 0;
  double offset = 0;
```

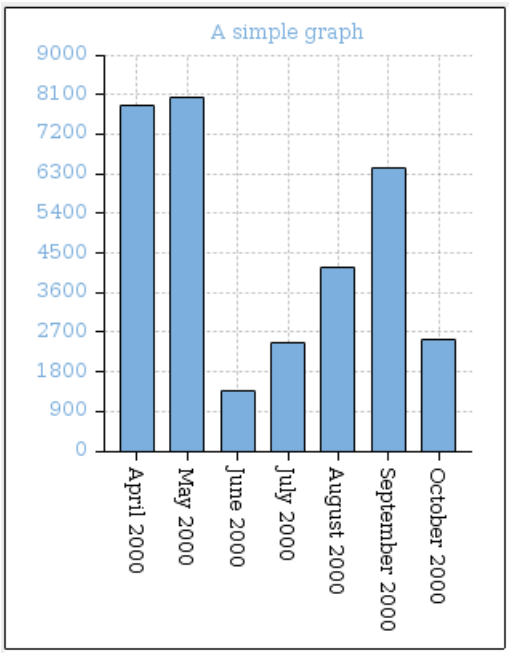
```
    // draw the component using the drawers
    if (data.IsMerged)
    {
      double w = ((MergeDrawer)drawers[0]).Draw(g, 0, 0, offset);
      offset += w;
      height = drawers[0].GetHeight(0);
    }
    else
    {
      for (int n = 0; n < data.Values.Count; ++n)
      {
        double w = ((SingleDrawer)drawers[n]).Draw(g, n, 0, 0, offset);
        if (w > 0)
        {
          offset += w + Gap;
          double h = drawers[n].GetHeight(n);
          if (height < h) height = h;
        }
      }
    }

    // defines the components width and height after the component is
    drawn
    Width = offset;
    Height = height;

    // draw the border
    if (BorderStroke > 0) g.DrawRectangle(null,
    g.CreatePen(BorderColor,
      BorderStroke), new Rect(0, 0, Width, height));
  }

  /// <summary>
  /// Event handler for double click. All drawers can test where a
      point falls
  /// in some area for the drawers current drawing. If so the drawing
      returns
  /// a FunctionValue, and if the chart has onservers this are notified
      the
  /// FunctionValue. The method must transform the mouse coordinates
      from window
  /// to coordinates to coordinates for the chart.
```

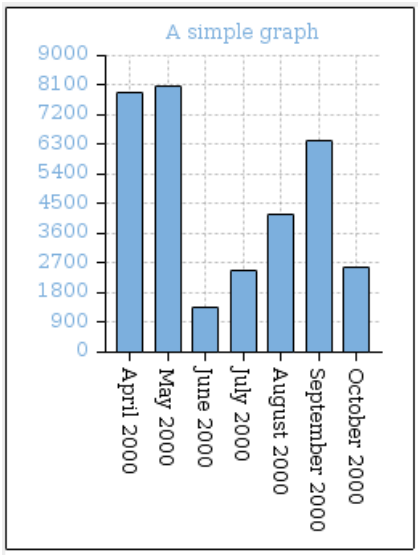
```
  /// </summary>
  /// <param name="e">EventArgs for mouse click</param>
  protected override void OnPreviewMouseDown(MouseButtonEventArgs e)
  {
    if (e.ClickCount == 2)
    {
      Point p = e.GetPosition(null);
      Window window = Window.GetWindow(this);
      Point q = this.TransformToAncestor(window).Transform(new Point(0,
      0));
      int x = (int)(p.X - q.X);
      int y = (int)(p.Y - q.Y);
      for (int n = 0; n < drawers.Count; ++n)
      {
        FunctionValue functionValue = drawers[n].GetClicked(n, x, y);
        if (functionValue != null && ChartClicked != null)
          ChartClicked(this, new ChartEventArgs(functionValue));
      }
    }
    e.Handled = true;
  }

  ...

}
```

The component is created on the basis of a *MultiValues* object and hence the definition of a chart. In addition, you must specify the font to be used and you can specify a gap between the individual graphs (if it is a multi-valued graph) and if a border must be drawn around the graph. Using the graph definition, the constructor creates a list of *Drawer* objects, where there is a *Drawer* for each graph type. These *Drawer* objects are created using the class *DrawerFactory*, which is a simple factory class. The class *Chart* uses these *Drawer* objects to draw the component. That is, the *Drawer* classes are absolutely key classes:

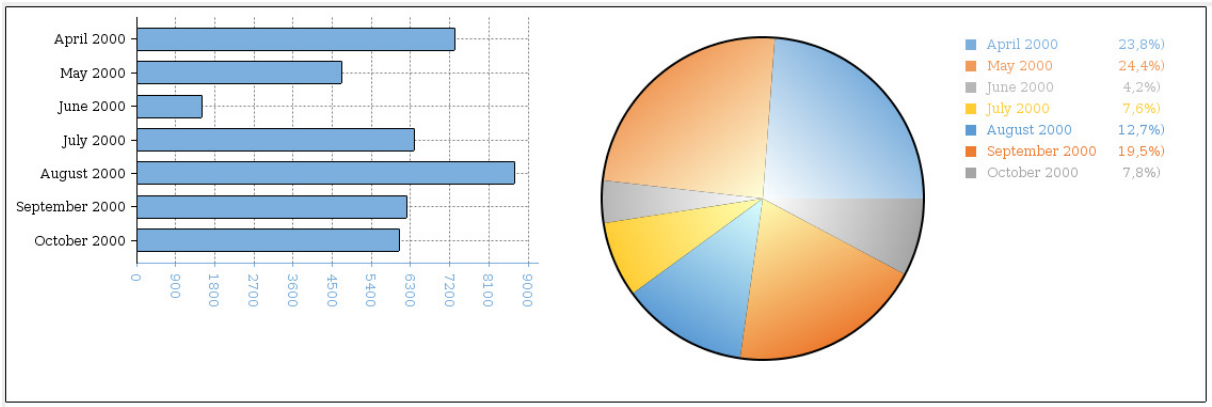


Drawer is an abstract class that defines three methods that define the width and height of the nth graph in a *Chart*, and also a method that returns a *FunctionValue* object if this *Drawer* is clicked. In general, a graph is drawn as a rectangular figure with a margin. The margin is used, for example, to the axis of the coordinate system, to a header text, and what the margin of the figure is used for and thus the size of the margin is thus determined by the current graph (not all graphs have a coordinate system). *SingleDrawer* is an abstract class, which implements the abstract methods as well as methods that determine a graph's margin and draw the graph. The class also defines scaling functions, and here you should be aware that they only scale the graph, but not the margin. As an example, two graphs are shown below, where the difference only is that the second is scaled:



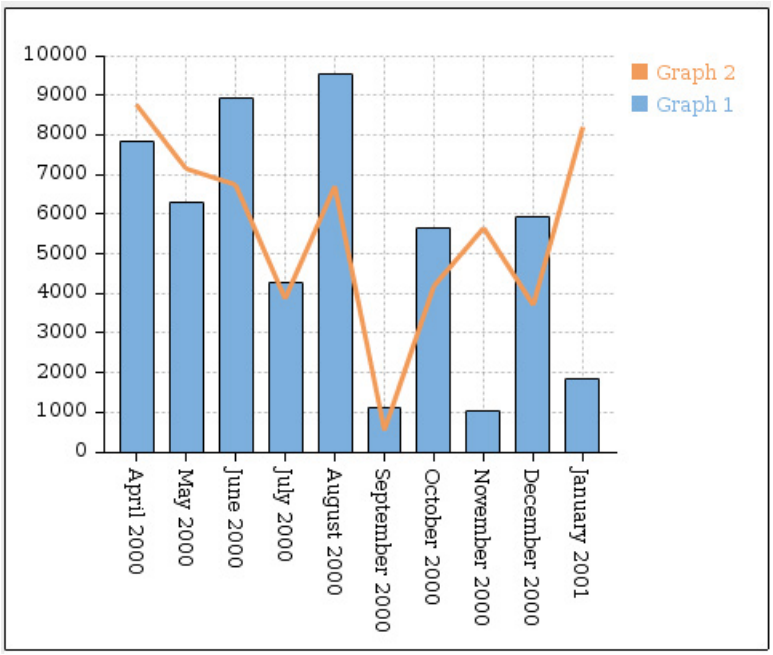


The type of the above graph is *VBAR*, and there is a specific class *VbarDrawer*, derived from *SingleDrawer*. It is this drawer used to draw the above graphs. There are correspondingly 8 other specific *Drawer* classes, which are derived from *SingleDrawer*. As another example, there is shown below a multi-valued graph, where the first has the type *HBAR* and is drawn with an *HbarDrawer*, while the other is of the type *PIE* and is drawn with a *PieDrawer*:



The code for these 9 concrete drawer classes is similar to each other and, in principle, they are quite simple, although there may of course be many details attached to implementing the drawing functions. The classes use more auxiliary classes, including classes that draw the axes. Here, especially the classes that draw the y axes are complex, as they typically depend on the actual numbers to make a sensible division of the axis.

Then there is the class *MergeDrawer* which is also a concrete drawer, but it draws the graph using other *Drawer* objects. If you have a multi-valued function, you can draw the graphs in the same coordinate system, which is what I have called a merge graph. As an example, below is shown a merge of a *LINE* graph and a *VBAR* graph:



Now you can not generally merge all graphs. For example, it makes no sense to merge a histogram (a VBAR graph) and a circle graph, and it makes no sense to merge a VBAR and a HBAR graph. The *MergeDrawer* class starts by dividing the graphs into some categories similar to how they can be merged, and for each of these categories, a *Drawer* class is defined. It leads to no less than 14 new *Drawer* classes, all of which are in principle similar to the above, but instead, they are derived from the abstract class *MultiDrawer*, as a single drawer this time always draws a multi-valued graph. There are so many *MultiDrawer* classes due to

1. that you can specify whether merged graphs should use the same y axis or each use there own y axis
2. that you can specify whether bar charts should be drawn side by side or stacked

Specifically, it corresponds to the following *MultiDrawer* classes:

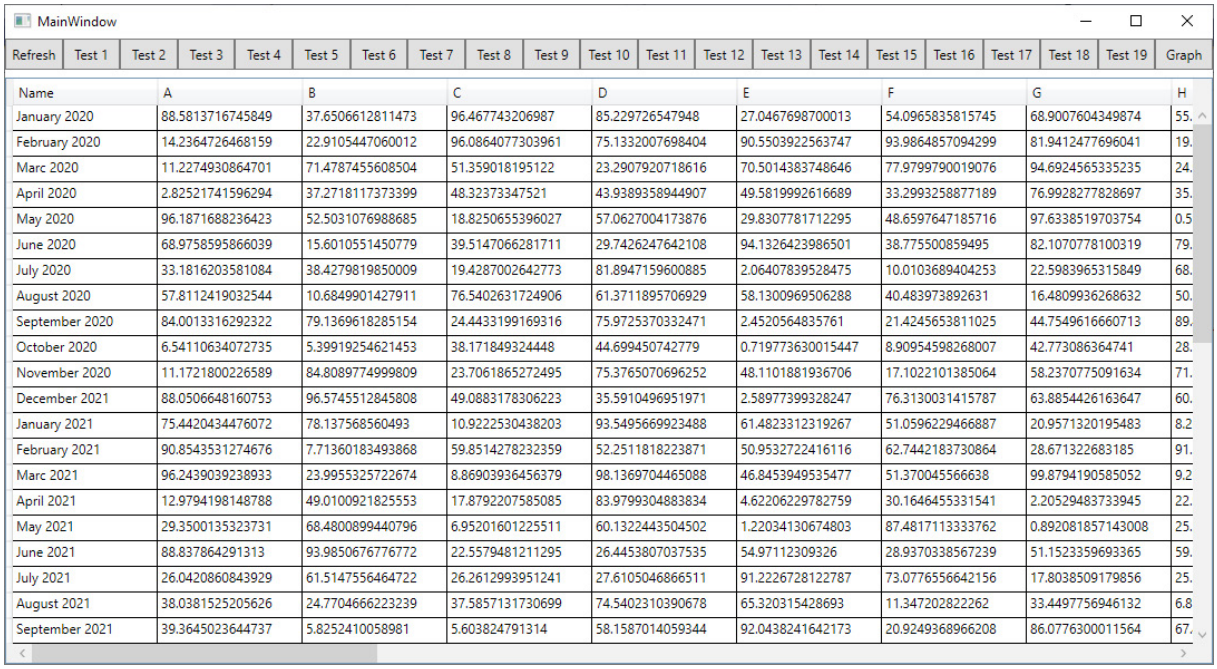
1. *MSLCPBDrawer*, which merges LINE, CURVE, PLOT and VBAR graphs with the same y axis
2. *MDLCPBDrawer*, which merges LINE, CURVE, PLOT and VBAR graphs with different y axis
3. *MSVBAR3DDrawer*, which merges VBAR3D graphs with the same y axis
4. *MDVBAR3DDrawer*, which merges VBAR3D graphs with different y axis
5. *MSHBARDrawer*, which merges HBAR graphs with the same y axis
6. *MDHBARDrawer*, which merges HBAR graphs with different y axis
7. *MSHBAR3DDrawer*, which merges HBAR3D graphs with the same y axis
8. *MDHBAR3DDrawer*, which merges HBAR3D graphs with different y axis
9. *MSLCPDrawer*, which merges LINE, CURVE and PLOT graphs with the same y axis if when stack of bar charts are selected
10. *MSLCPDrawer*, which merges LINE, CURVE and PLOT graphs with the different y axis if when stack of bar charts are selected
11. *SVBARDrawer*, which merges VBAR graphs
12. *SHBARDrawer*, which stacks HBAR graphs
13. *SVBAR3DDrawer*, which stacks VBAR3D graphs
14. *SHBAR3DDrawer*, which stacks HBAR3D graphs

If a multi-valued graph is to be merged and a graph can not be merged with others, it is drawn as a *SingleValues* graph using the corresponding drawer.

The result of all these is that there is a lot of code related to merge of graphs, but the work consists primarily of writing the respective *Drawer* classes, all of which look similar. All the code is in a name space *Tools*, but I will not show the code here, but it is a good exercise to study the code. You should note that they are only a few comments in the code as the classes are similar to each other and I thus had to write the same comments in many places.

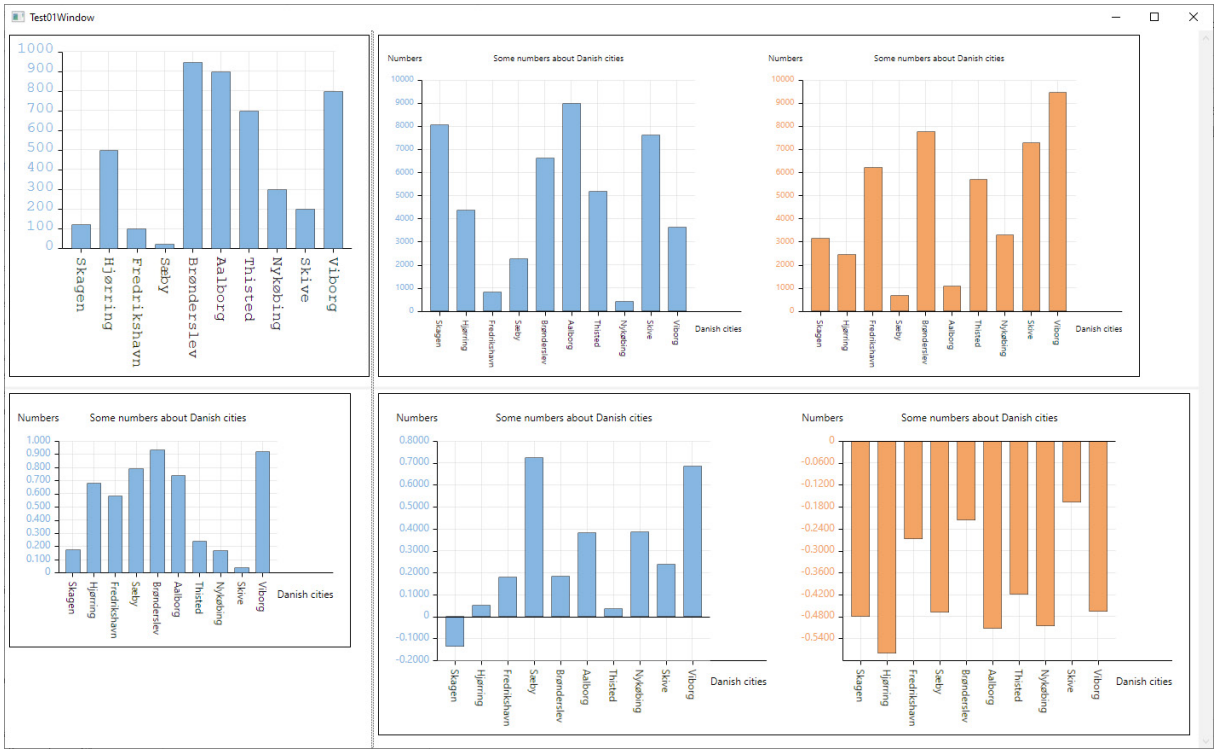
13.2 THE TEST PROGRAM

Then there is the test program that opens a window with a *DataGrid* containing numbers - near the first column, which contains texts to be used as labels for graphs:

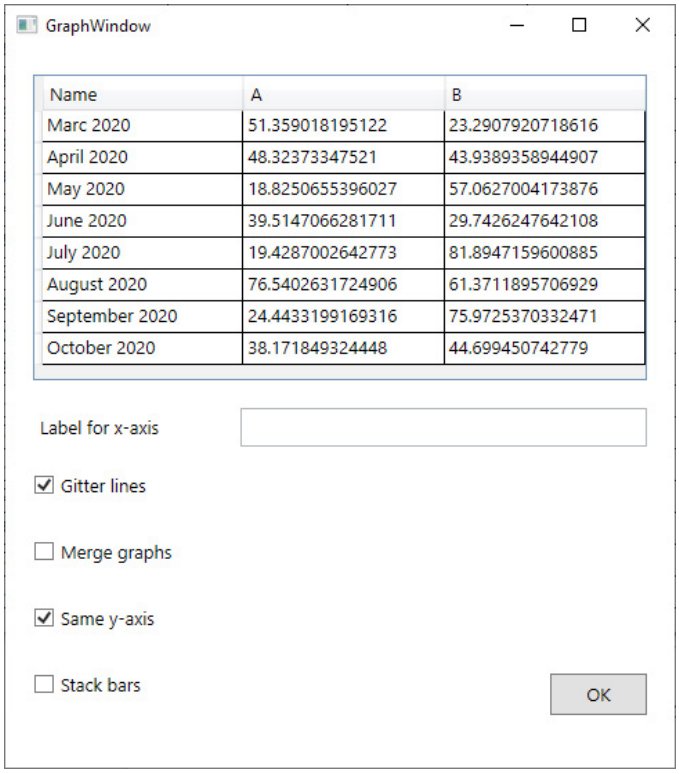


Name	A	B	C	D	E	F	G	H
January 2020	88.5813716745849	37.6506612811473	96.467743206987	85.229726547948	27.0467698700013	54.0965835815745	68.9007604349874	55.
February 2020	14.2364726468159	22.9105447060012	96.0864077303961	75.1332007698404	90.5503922563747	93.9864857094299	81.9412477696041	19.
Marc 2020	11.2274930864701	71.4787455608504	51.359018195122	23.2907920718616	70.5014383748646	77.9799790019076	94.6924565335235	24.
April 2020	2.82521741596294	37.2718117373399	48.32373347521	43.9389358944907	49.5819992616689	33.2993258877189	76.9928277828697	35.
May 2020	96.1871688236423	52.5031076988685	18.8250655396027	57.0627004173876	29.8307781712295	48.6597647185716	97.6338519703754	0.5
June 2020	68.9758595866039	15.6010551450779	39.5147066281711	29.7426247642108	94.1326423986501	38.775500859495	82.1070778100319	79.
July 2020	33.1816203581084	38.4279819850009	19.4287002642773	81.8947159600885	2.06407839528475	10.0103689404253	22.5983965315849	68.
August 2020	57.8112419032544	10.6849901427911	76.5402631724906	61.3711895706929	58.1300969506288	40.483973892631	16.4809936268632	50.
September 2020	84.0013316292322	79.1369618285154	24.4433199169316	75.9725370332471	2.4520564835761	21.4245653811025	44.7549616660713	89.
October 2020	6.54110634072735	5.39919254621453	38.171849324448	44.699450742779	0.719773630015447	8.90954598268007	42.773086364741	28.
November 2020	11.1721800226589	84.8089774999809	23.7061865272495	75.3765070696252	48.1101881936706	17.1022101385064	58.2370775091634	71.
December 2021	88.0506648160753	96.5745512845808	49.0883178306223	35.5910496951971	2.58977399328247	76.3130031415787	63.8854426163647	60.
January 2021	75.4420434476072	78.137568560493	10.9222530438203	93.5495669923488	61.4823312319267	51.0596229466887	20.9571320195483	8.2
February 2021	90.8543531274676	7.71360183493868	59.8514278232359	52.2511818223871	50.9532722416116	62.7442183730864	28.671322683185	91.
Marc 2021	96.2439039238933	23.9955325722674	8.86903936456379	98.1369704465088	46.8453949535477	51.370045566638	99.8794190585052	9.2
April 2021	12.9794198148788	49.0100921825553	17.8792207585085	83.9799304883834	4.62206229782759	30.1646455331541	2.20529483733945	22.
May 2021	29.3500135323731	68.4800899440796	6.95201601225511	60.1322443504502	1.22034130674803	87.4817113333762	0.892081857143008	25.
June 2021	88.837864291313	93.9850676776772	22.5579481211295	26.4453807037535	54.97112309326	28.9370338567239	51.1523359693365	59.
July 2021	26.0420860843929	61.5147556464722	26.2612993951241	27.6105046866511	91.2226728122787	73.0776556642156	17.8038509179856	25.
August 2021	38.0381525205626	24.7704666223239	37.5857131730699	74.5402310390678	65.320315428693	11.347202822262	33.4497756946132	6.8
September 2021	39.3645023644737	5.8252410058981	5.603824791314	58.1587014059344	92.0438241642173	20.9249368966208	86.0776300011564	67.

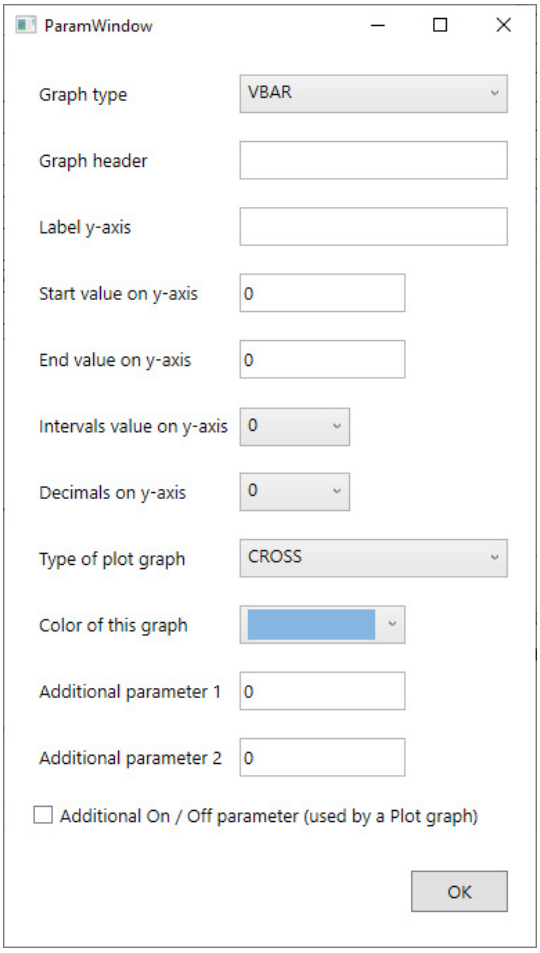
The text in first column are used as labels for independent axis, while the numbers are used as values for graphs. All numbers are random generated, and are when the program starts between 1 and 100. At the top there is a toolbar with buttons. Clicking on the first button allows you to specify an interval for the table's numbers, and the program will then fill the table with random numbers within this range. Next, 19 test buttons follow which open windows I have used for testing in connection with the development of the library. The windows look likes each others, but shows different charts, and they could be programmed with less code than is the case, but to make it simple (and the maintenance of the program is not important) a dialog box has been created for each window. All windows are hard coded and does not use the numbers in the table. As example the below is the result when you click on *Test 1*:



The most interesting is the button to the far right. If you select a rectangular area in the table (which does not include the left column) and clicking the button the program opens a window showing the selected numbers and where you can select values for a *MultiValues* defining a chart:

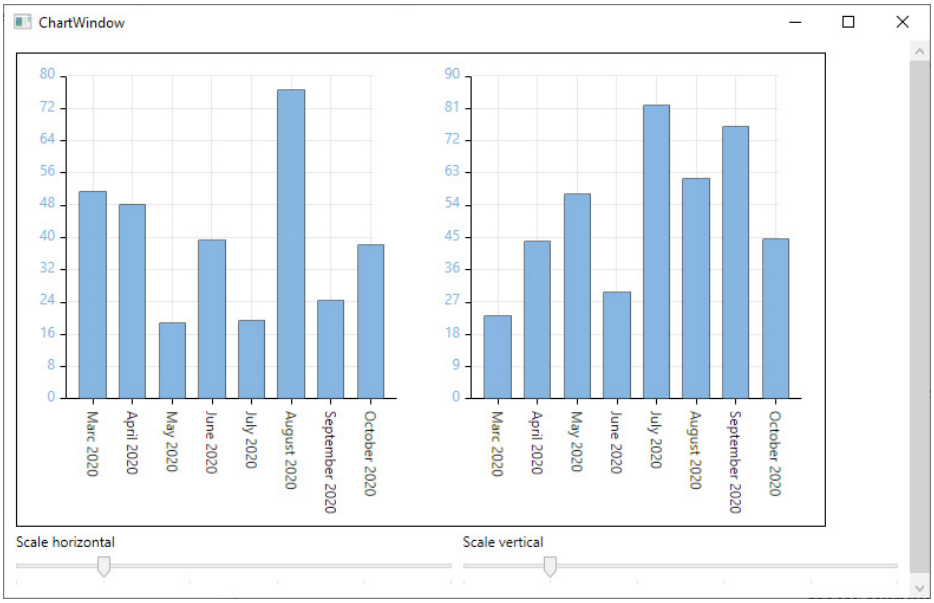


Above is selected two columns with numbers, and the result will be a multi-valued function with two dependent value for each of the 8 independent values. If you double click on a column header you opens another dialog box to enter values for single-value function and then a *SingleValues* object:



You can thus use the program to define different settings for the individual graphs and test the effect.

If you in the window *GraphWindow* click on *OK* and do not change any think you get the following window:



The sliders below the graph are used to scale the graph.

I should not show the code for the test program. There is a lot of it, but most of it is below the graph.

13.3 LACKS AND THINGS THAT COULD BE IMPROVED

The graphs can not display all numbers with a reasonable result, and some of the graphs, for example, are mistaken for negative numbers. For example, it does not make sense to stack bar charts if the numbers can be negative. The conclusion is that several of the graphs have preconditions, which the drawing functions with advantage could validate and possibly raise an exception if the graph can not be drawn correctly.

The actual component *Chart* can raise a single event if the user clicks on a graph. The component is drawn based on an object of type *MultiValues*, and the component could be extended as it fires an event if a value for a property is changed in the *MultiValues* data structure. In practice, it would typically mean that the component should be redrawn, and it could also be of significance to the layout controls which contains the component.

Another question is how easy it is to expand the library with a new graph. If it is a whole new graph for a single-valued function, it's simple. In this case, the work consists primarily of writing a new *Drawer* derived from *SingleDrawer*, adding a new graph type to *GraphType* and expanding the class *DrawingFactory*. In addition, the *MergeDrawer* class must be expanded (the method *Merge()*) to know the new type. If graphs of the new type can also be merged,

a *Drawer* derived from *MultiDrawer* must also be written, and even more extensive is if the new graph type can be merged with existing graphs, which can both cause existing *Drawer* types to be changed as well as possible new *Drawer* classes to be written. The conclusion is that if the library is expanded with new graphs, which can be merged with existing graphs, the work can be quite extensive.

It may be a good practice to improve the library corresponding to the above and optionally expand with a new graph type.