ComponentOne

BasicLibrary for UWP

ComponentOne, a division of GrapeCity
201 South Highland Avenue, Third Floor
Pittsburgh, PA 15206 USA

Website: http://www.componentone.com
Sales: sales@componentone.com
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Table of Contents

UWP Edition Basic Library

Get UI controls for data visualization, layout, input, and more inside **ComponentOne Studio UWP Edition**. Based on our popular Silverlight controls and designed to enhance the rich user experience of the UWP platform, these controls give you powerful and unique functionality to help you build better and more compelling Universal Windows apps.

The basic library includes the following controls:

- **CollectionView for UWP**

  Get a more powerful implementation of the **ICollectionView** interface with **CollectionView for UWP**. The C1CollectionView class delivers missing functionality like sorting, filtering, grouping, and modifying any collection. Use **C1CollectionView** as you would use **CollectionViewSource** in your Windows Store apps to instantly get more functionality without sacrifice.

- **Input for UWP**

  Provide smarter input for phone numbers, zip codes, percentages and more. With **Input for UWP** you get two controls for masked and numeric input. Quickly gather valid input while displaying formatted text automatically.

- **Layout Panels for UWP**

  Control the flow and positioning of the content in your Universal Windows apps with **Layout Panels for UWP**. Wrap content vertically or horizontally using C1WrapPanel. Dock content along the edges of the panel with C1DockPanel. Display content neatly in a grid using **C1UniformGrid**.

- **ListBox for UWP**

  Get two high performance controls for displaying lists of bound data with **ListBox for UWP**. Display lists with tile layouts or with optical zoom using the C1ListBox and C1TileListBox controls. These controls support UI virtualization so they are blazing-fast while able to display thousands of items with little-to-no loss of performance.

- **Menu for UWP**

  Add the classic "File" menu system to your Windows Store applications with **Menu for UWP**. The C1Menu control gives a real desktop look-and-feel with traditional looking menus that support deep nested items and vertical orientation.

- **RadialMenu for UWP**

  Add an attractive radial menu system to your Windows Store apps with **RadialMenu for UWP**. Modeled after popular Microsoft apps, the C1RadialMenu control gives you a unique and touch-friendly alternative to the traditional context menu.

- **TabControl for UWP**

  Organize and navigate content as tabs with **TabControl for UWP**. Tabs help utilize available space while letting the user see all available items to select. Tabs can be positioned to the top, bottom, left, or right of a page and support several different shapes and built-in features.

- **TreeView for UWP**

  Get a hierarchical view of your data items with **TreeView for UWP**. The familiar TreeView UI is now available for Windows 8 applications. Supports collapsible nodes, hierarchical templates, check box nodes, editing, and drag-and-drop operations.

- **DropDown for UWP**

  Create drop-downs for any type of custom input with **DropDown for UWP**. Make custom drop-down editors without having to mess with popups and flyouts. The C1DropDown control allows you to create that specialized drop-down for anything imaginable such as a color picker, an autocomplete textbox, or even a hierarchical combobox.

- **RangeSlider for UWP**

  Add smooth numeric data selection to your UWP applications with **RangeSlider for UWP**. It extends the basic slider control and provides two thumb elements instead of one, allowing users to select ranges instead of single values.

- **GridSplitter for UWP**

  Redistribute space between columns and rows with **GridSplitter for UWP**.

- **HeaderedContentControl for UWP**

  Add style to your UI, create customized layout and display blocks of content. This control is comprised of two elements: a header bar and a content panel. The header can be horizontal or vertical and the content panel can be located accordingly.

Getting Started with UWP Edition

Help with UWP Edition

Getting Started

For information on installing **ComponentOne Studio UWP Edition**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with Component Studio UWP Edition.

CollectionView for UWP

Get a more powerful implementation of the **ICollectionView** interface with **CollectionView for UWP**.
The C1CollectionView class delivers missing functionality like sorting, filtering, grouping, and modifying any collection. Use C1CollectionView as you would use **CollectionViewSource** in your Universal Windows apps to instantly get more functionality without sacrifice.

CollectionView for UWP Key Features

CollectionView for UWP includes the following key features:

- **Sorting, Filtering and Grouping**

  The C1CollectionView class and IC1CollectionView interface give you support for sorting, filtering, and grouping collections in UWP apps. The object model and functionality are virtually identical to those in the **ICollectionView** interface provided in WPF, Silverlight, and Windows Phone, so there is no learning curve.

- **More Powerful than CollectionViewSource**

  The standard **ICollectionView** interface and **CollectionViewSource** implementations in UWP are limited compared to WPF and Silverlight. For instance, the UWP implementation of **ICollectionView** does not support sorting, filtering or editing. The C1CollectionView class adds these missing elements so you can achieve the functionality you need.

- **Use with Any Control**

  C1CollectionView implements the **ICollectionView** interface completely, so any standard items control can use it as a data source. Many ComponentOne controls make use of the C1CollectionView class internally and automatically, but the interface is public so you can use it yourself with any control.

- **Editable Live Rowsets**

  C1CollectionView provides a live rowset. Any changes made to the data in **C1CollectionView** are automatically propagated to the underlying collection. **C1CollectionView** also adds and implements the **IEditableCollectionView** interface, cloned from WPF and Silverlight, so you get a better editing experience including pressing escape to cancel all changes made to the item being edited.

- **Familiar Object Model**

  The C1CollectionView interface is based on the WPF and Silverlight **ICollectionView**, so you will be familiar with sorting, filtering and grouping your collections. The **C1CollectionView** class is also compatible with WPF, Silverlight and Windows Phone versions so you can easily re-use your code.

Getting Started with C1CollectionView

C1CollectionView implements the IC1CollectionView interface which in turn implements the standard **ICollectionView** interface. Like the standard **CollectionView** class in WPF, C1CollectionView supports current item management, item selection, sorting, grouping, filtering and editing. If you can't find enough information about a topic in this documentation then you can easily search the web for **ICollectionView** practices in WPF and Silverlight to find very valuable information that can be used in UWP.

The **C1CollectionView** class can be found in the C1.UWP assembly.

To get started with C1CollectionView, instantiate it with an **IEnumerable** collection of your business objects:

Visual Basic

```
Dim customers As List(Of Customer) = Await GetCustomerData()
Dim view = New C1.Xaml.C1CollectionView(customers)
```

C#

```
List<Customer> customers = await GetCustomerData();
var view = new C1.Xaml.C1CollectionView(customers);
```

Then bind it to your favorite **ItemsControl** or data grid to start using **C1CollectionView**:

Visual Basic

```
C1FlexGrid1.ItemsSource = view
```

C#

```
c1FlexGrid1.ItemsSource = view;
```

C1CollectionView Versus CollectionViewSource

The standard **ICollectionView** interface and **CollectionViewSource** implementations in UWP are limited compared to WPF and Silverlight. For instance, the UWP implementation of **ICollectionView** does not support sorting, filtering or collection editing. The C1CollectionView class adds these missing elements so you can achieve that extra functionality you need. The IC1CollectionView interface is based on the WPF and Silverlight **ICollectionView**, so if you are familiar with those platforms, you will be familiar with sorting, filtering and grouping with **C1CollectionView**. If you are familiar with the **CollectionViewSource** class then moving to C1CollectionView is very easy.

Take a look at this example binding the **C1FlexGrid** control to a **CollectionViewSource** versus a **C1CollectionView** given the same underlying list of **Customer** objects.

CollectionViewSource:

Visual Basic

```
Dim customers As List(Of Customer) = Await GetCustomerData()
Dim view = New CollectionViewSource()
view.Source = customers
c1FlexGrid1.ItemsSource = view.View
```

C#

```
List<Customer> customers = await GetCustomerData();
var view = new CollectionViewSource();
view.Source = customers;
c1FlexGrid1.ItemsSource = view.View;
```

C1CollectionView:

Visual Basic

```
Dim customers As List(Of Customer) = Await GetCustomerData()
Dim view = New C1.Xaml.C1CollectionView(customers)
c1FlexGrid1.ItemsSource = view
```

C#

```
List<Customer> customers = await GetCustomerData();
var view = new C1.Xaml.C1CollectionView(customers);
c1FlexGrid1.ItemsSource = view;
```

If you are working in MVVM, simply expose a property of type **IC1CollectionView** on the view model and populate the collection within the view model. Then bind the **ItemsSource** property of **C1FlexGrid** (or whatever control you are using) to the property in XAML.

Visual Basic

```
''' <summary>
''' Gets the collection of customers.
''' </summary>
Public ReadOnly Property Customers() As C1.Xaml.IC1CollectionView
      Get
            Return view
      End Get
End Property
```

C#

```
/// <summary>
/// Gets the collection of customers.
/// </summary>
public C1.Xaml.IC1CollectionView Customers
{
    get { return view; }
}
```

Sorting C1CollectionView

You can sort items in your collection using C1CollectionView just as you would using a **CollectionView** implementation in any other platform. When you sort a C1CollectionView, the underlying data set is not affected.

For example you can sort by using the SortDescriptions property passing in a SortDescription object:

Visual Basic

```
Dim list = New System.Collections.ObjectModel.ObservableCollection(Of Customer)()
' create a C1CollectionView from the list
Dim _view As New C1.Xaml.C1CollectionView(list)
' sort customers by country
_view.SortDescriptions.Add(New C1.Xaml.SortDescription("Country",
C1.Xaml.ListSortDirection.Ascending))
```

C#

```
var list = new System.Collections.ObjectModel.ObservableCollection<Customer>();
// create a C1CollectionView from the list
C1.Xaml.C1CollectionView _view = new C1.Xaml.C1CollectionView(list);
// sort customers by country
_view.SortDescriptions.Add(new C1.Xaml.SortDescription("Country",
C1.Xaml.ListSortDirection.Ascending));
```

Where "Country" is the name of the property which you want to sort on. You can sort ascending (A-Z) or descending (Z-A) depending on the **ListSortDirection** parameter.

Sorting on more than one property or column

You can sort on more than one property by simply adding additional SortDescriptions. If you are sorting by multiple properties you should use the DeferRefresh method to defer the automatic refresh after each sort so that you only apply each sort once.

Visual Basic

```
' sort multiple properties using DeferRefresh so you only refresh once
Using _view.DeferRefresh()
      _view.SortDescriptions.Clear()
      _view.SortDescriptions.Add(New C1.Xaml.SortDescription("Country",
C1.Xaml.ListSortDirection.Ascending))
      _view.SortDescriptions.Add(New C1.Xaml.SortDescription("Name",
C1.Xaml.ListSortDirection.Ascending))
```

```
End Using
```

C#

```
// sort multiple properties using DeferRefresh so you only refresh once
using (_view.DeferRefresh())
{
    _view.SortDescriptions.Clear();
    _view.SortDescriptions.Add(new C1.Xaml.SortDescription("Country",
C1.Xaml.ListSortDirection.Ascending));
    _view.SortDescriptions.Add(new C1.Xaml.SortDescription("Name",
C1.Xaml.ListSortDirection.Ascending));
}
```

Note: If the collection implements **INotifyCollectionChanged** any changes to the data will be applied to the sort even after it's been set.

For more advanced sorting that can improve performance see the C1CollectionView.CustomSort property.

Filtering C1CollectionView

With C1CollectionView you can filter a collection to produce a new sub-set containing exactly those elements of the original collection for which a given predicate returns true. When you filter a **C1CollectionView**, the underlying data set is not affected. The Filter property gets or sets a callback used to determine if an item is suitable for inclusion in the view.

For example, you can set a predicate to the **Filter** property and this will cause the list to be filtered by that predicate.

Visual Basic

```
' create an observable list of customers
Dim list = New System.Collections.ObjectModel.ObservableCollection(Of Customer)()

' create a C1CollectionView from the list
_view = New C1.Xaml.C1CollectionView(list)

' filter by country = Austria. Customers not from Austria will be filtered out.
_view.Filter = Sub(item As Object)
Dim c As Customer = TryCast(item, Customer)
If c IsNot Nothing Then
      If c.Country.Equals("Austria") Then
            Return True
      End If
End If
Return False
End Sub
```

C#

```
// create an observable list of customers
var list = new System.Collections.ObjectModel.ObservableCollection<Customer>();

// create a C1CollectionView from the list
```

```
_view = new C1.Xaml.C1CollectionView(list);

// filter by country = Austria. Customers not from Austria will be filtered out.
_view.Filter = delegate(object item)
{
    Customer c = item as Customer;
    if (c != null)
    {
        if (c.Country.Equals("Austria"))
            return true;
    }
    return false;
};
```

Note: If the collection implements **INotifyCollectionChanged** any changes to the data will be applied to the filter even after it's been set.

For an advanced example of filtering, see the **Filter** sample for **FlexGrid for UWP**: http://our.componentone.com/samples/UWPxaml-filter.

Grouping C1CollectionView

Group your collection by a particular property using the GroupDescriptions property on C1CollectionView.

For example, to group the collection by the "Country" property:

Visual Basic

```
Dim list = New System.Collections.ObjectModel.ObservableCollection(Of Customer)()
' create a C1CollectionView from the list
_view = New C1.Xaml.C1CollectionView(list)

' group customers by country
_view.GroupDescriptions.Add(New C1.Xaml.PropertyGroupDescription("Country"))
```

C#

```
var list = new System.Collections.ObjectModel.ObservableCollection<Customer>();

// create a C1CollectionView from the list
_view = new C1.Xaml.C1CollectionView(list);

// group customers by country
_view.GroupDescriptions.Add(new C1.Xaml.PropertyGroupDescription("Country"));
```

Note: If the collection implements **INotifyCollectionChanged** any changes to the data will be applied to the grouping even after it's been set.

The **C1FlexGrid** control supports grouping on C1CollectionView for you. If you are grouping with any other control you will need to define a **GroupStyle** to control the appearance of each group.

For example, here is a **GroupStyle** defined for a standard **ListBox** control:

Markup

```
<ListBox
    Name="_listBox"
    ItemsSource="{Binding Customers}">
    <ListBox.GroupStyle>
        <GroupStyle>
            <GroupStyle.HeaderTemplate>
                <DataTemplate>
                    <TextBlock Text="{Binding}" />
                </DataTemplate>
            </GroupStyle.HeaderTemplate>
        </GroupStyle>
    </ListBox.GroupStyle>
</ListBox>
```

Custom Grouping

Consider a scenario where you want to group by a number. Unless you have a short list of clean integers, the group of unique values will be unmanageable. You would instead want to apply a custom grouping action that groups items into ranges like "between 0 and 100" and "over 5,000", etc. To do this you would perform a custom grouping passing an **IValueConverter** to the **PropertyGroupDescription** parameter.

For example, the following code will group our Customer collection by Country listing each group as a letter of the alphabet (such as: Countries: A, Countries: B, Countries: C and so on). The Countries: A group would include all items belonging to Algeria, Argentina, and Austria. Modify the previous code snippet to the following:

Visual Basic

```
_view.GroupDescriptions.Add(New C1.Xaml.PropertyGroupDescription("Country", New
GroupByCountryAtoZConverter()))
```

C#

```
_view.GroupDescriptions.Add(new C1.Xaml.PropertyGroupDescription("Country", new
GroupByCountryAtoZConverter()));
```

And add the following **GroupByCountryAtoZConverter** class to your project:

Visual Basic

```
Public Class GroupByCountryAtoZConverter
      Implements IValueConverter
      Public Function Convert(value As Object, targetType As Type, parameter As
Object, culture As String) As Object
            If value IsNot Nothing Then
                  Return value.ToString()(0)
            End If
            Return "Undefined"
      End Function

      Public Function ConvertBack(value As Object, targetType As Type, parameter As
Object, culture As String) As Object
```

```
            Throw New NotImplementedException()
        End Function
End Class
```

C#

```
public class GroupByCountryAtoZConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string
culture)
    {
        if (value != null)
        {
            return value.ToString()[0];
        }
        return "Undefined";
    }

    public object ConvertBack(object value, Type targetType, object parameter, string
culture)
    {
        throw new NotImplementedException();
    }
}
```

DropDown for UWP

Create drop-downs for any type of custom input with **DropDown for UWP**. The C1DropDown control helps you make custom drop-down editors without having to mess with popups and flyouts. Use it to create that specialized drop-down for anything imaginable such as a color picker, an autocomplete textbox, or even a hierarchical combobox.

DropDown for UWP Key Features

DropDown for UWP's key features include the following:

- **Can host any UI as drop-down content**

  The C1DropDown control gives you complete control to create specialized drop-down editors with ease. Fully design you own drop-down content, and configure which value to display in the header portion. For example, you could place a C1TreeView in the drop-down portion to create a hierarchical combobox that displays the selected node in the header.

- **Automatic closing and opening bounds detection**

  Configure the drop-down direction preference to above or below the header portion. If there is not enough room on the page, the **C1DropDown** control will automatically display in the other direction. With the AutoClose property you can determine if the drop-down automatically closes when the control loses focus.

C1DropDown Quick Start

The following quick start guide is intended to get you up and running with the C1DropDown control. In this quick start you'll start in Visual Studio and create a new project, add **C1DropDown** to your application, and customize the appearance and behavior of the control.

Step 1 of 3: Creating an Application with a C1DropDown Control

In this step, you'll create a UWP application in Visual Studio using **DropDown for UWP**.

1. In Visual Studio 2013 Select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Open **MainPage.xaml** if it isn't already open.
4. Edit your opening <Page> tag to add the following namespace:

Markup
```
xmlns:Xaml="using:C1.Xaml"
```

5. Locate the <Grid> </Grid> tags on your page and insert the following to define your Grid rows:

Markup
```
<Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition/>
        </Grid.RowDefinitions>
```

6. Directly below the row definitions, add a CheckBox using the following markup:

Markup
```
<CheckBox Content="AutoClose - determines if C1DropDown will automatically close
when focus is lost." IsChecked="{Binding ElementName=c1DropDown1,
Path=AutoClose, Mode=TwoWay}" Margin="10"/>
```

7. Use the following markup to add a C1DropDownButton control:

Markup
```
<Xaml:C1DropDownButton Grid.Row="1" Background="White" x:Name="c1DropDown1"
Padding="2" Width="150" HorizontalAlignment="Center" VerticalAlignment="Center"
> </Xaml:C1DropDownButton>
```

8. Next, we'll add some Header content to customize the C1DropDownButton.Header:

Markup
```
<Xaml:C1DropDownButton.Header>
  <Border x:Name="dropDownBorder" Background="White" />
</Xaml:C1DropDownButton.Header>
```

What You've Accomplished

In this step, you created a new Visual Studio Universal Windows application, added a **CheckBox** control, and added and began customizing a **C1DropDown** control.

Step 2 of 3: Adding Content to the C1DropDown Control

In this step, you'll add some content to the C1DropDown control you added in the preceding step.

1. Add the following markup directly below the <Xaml:C1DropDownButton.Header>. This will allow you to add any content to your C1DropDown control:

   Markup

   ```
   <Xaml:C1DropDownButton.Content>
   </Xaml:C1DropDownButton.Content>
   ```

2. Place your cursor between the <Xaml:C1DropDownButton.Content> </Xaml:C1DropDown.Content> tags.
3. Locate the C1TileListBox control in the Visual Studio ToolBox. Double-click the control to add it to the page.
4. Edit the opening <Xaml:C1TileListBox> tag so that it resembles the following:

   Markup

   ```
   <Xaml:C1TileListBox x:Name="colorListBox"
                                    Height="180"
                                    Orientation="Horizontal"
                                    ItemTapped="colorListBox_ItemTapped"
                                    SelectionMode="None"
                                    BorderBrush="{StaticResource
   ComboBoxPopupBorderThemeBrush}"
                                    BorderThickness="{StaticResource
   ComboBoxPopupBorderThemeThickness}"
                                    Background="{StaticResource
   ComboBoxPopupBackgroundThemeBrush}">
   ```

5. Place your cursor between the <Xaml:C1TileListBox> </Xaml:C1TileListBox> tags and add the following markup. This will allow you to change the background color of the C1DropDownButton control at runtime:

   Markup

   ```
   <Border Background="Black" BorderBrush="White" BorderThickness="1"/>
                       <Border Background="DarkGray"/>
                       <Border Background="White" BorderBrush="Black"
   BorderThickness="1"/>
                       <Border Background="DarkBlue" />
                       <Border Background="Blue" />
                       <Border Background="Cyan" />
                       <Border Background="Teal" />
                      <Border Background="Green" />
                       <Border Background="Lime" />
                       <Border Background="SaddleBrown"/>
                       <Border Background="Orange" />
                       <Border Background="Yellow" />
                       <Border Background="Maroon" />
   ```

```
                    <Border Background="Red" />
                    <Border Background="Magenta" />
```

6. Right-click your page and select **View Code** from the list. Add the following namespace at the top of the page:

C#
```
using C1.Xaml;
```

7. Add the following code to handle the **colorListBox_ItemTapped** event:

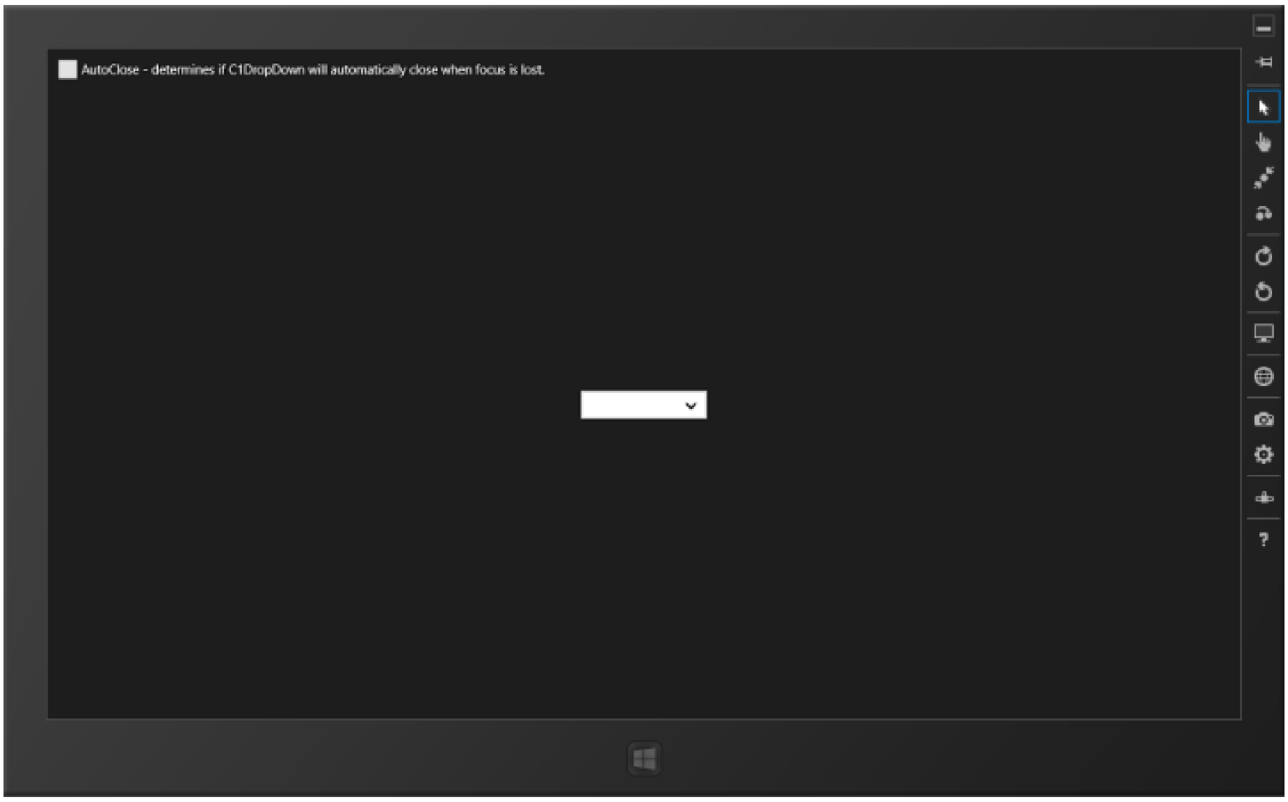
C#
```
private void colorListBox_ItemTapped(object sender, C1TappedEventArgs e)
        {
            C1ListBoxItem item = sender as C1ListBoxItem;
            if (item != null)
            {
                Border b = item.Content as Border;
                if (b != null)
                {
                    dropDownBorder.Background = b.Background;
                }
            }
            c1DropDown1.IsDropDownOpen = false;
        }
```

What You've Accomplished

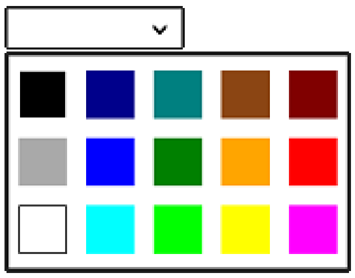
In this step, you added content to the **C1DropDownButton** control. In the next step, you'll run your application.

Step 3 of 3: Running the C1DropDown Application

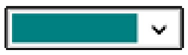
1. Press F5 or start debugging to run your application. It should resemble the following image:


AutoClose - determines if C1DropDown will automatically close when focus is lost.

2. When you tap or click the drop-down button, the C1DropDownButton control will resemble the following image:



3. And when you select a color from the C1TileListBox, the **C1DropDownButton** will resemble the following image:



What You've Accomplished

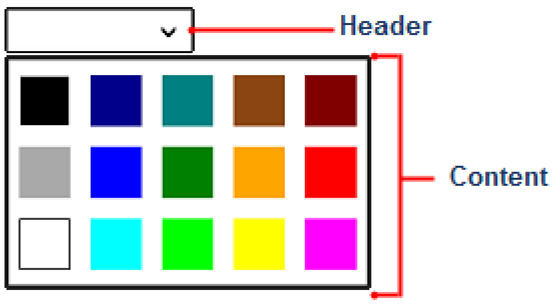
In this Quick Start, you created a new Universal Windows application, added a **C1DropDownButton** control to the application, and added a **C1TileListBox** to the content area of the **C1DropDownButton**.

Working with DropDown for UWP

DropDown for UWP includes the C1DropDown control, a simple drop-down box control that acts as a container, allowing you to add controls, images, and more to an interactive input box. When you add the C1DropDown control to a XAML window it exists as a fully function input control that can be customized and include added content.

C1DropDown Elements

The C1DropDown control consists of two parts: a header area and a content area. The header appears visible when the drop-down box is not open; the content is what is visible when the drop-down area is clicked. In the image below, the two sections are identified:



You can add content to neither, either, or both the header and content areas. You can customize the **C1DropDown** control in XAML by adding content to the header and content areas. For example, the following markup creates a drop-down control similar to the one pictured above:

Markup

```
<Xaml:C1DropDownButton Grid.Row="1" Background="White" x:Name="c1DropDown1"
Padding="2" Width="150" HorizontalAlignment="Center" VerticalAlignment="Center"
Xaml:C1NagScreen.Nag="True">
            <Xaml:C1DropDownButton.Header>
               <Border x:Name="dropDownBorder" Background="White" />
            </Xaml:C1DropDownButton.Header>
            <Xaml:C1DropDownButton.Content>
                <Xaml:C1TileListBox x:Name="colorListBox"
                                  Height="180"
                                  Orientation="Horizontal"
                                  ItemTapped="colorListBox_ItemTapped"
                                  SelectionMode="None"
                                  BorderBrush="{StaticResource
ComboBoxPopupBorderThemeBrush}"
                                  BorderThickness="{StaticResource
ComboBoxPopupBorderThemeThickness}"
                                  Background="{StaticResource
ComboBoxPopupBackgroundThemeBrush}">
                    <Border Background="Black" BorderBrush="White"
BorderThickness="1"/>
                    <Border Background="DarkGray"/>
                    <Border Background="White" BorderBrush="Black"
BorderThickness="1"/>
```

```
                    <Border Background="DarkBlue" />
                    <Border Background="Blue" />
                    <Border Background="Cyan" />

                    <Border Background="Teal" />
                    <Border Background="Green" />
                    <Border Background="Lime" />

                    <Border Background="SaddleBrown"/>
                    <Border Background="Orange" />
                    <Border Background="Yellow" />

                    <Border Background="Maroon" />
                    <Border Background="Red" />
                    <Border Background="Magenta" />
                </Xaml:C1TileListBox>
            </Xaml:C1DropDownButton.Content>
        </Xaml:C1DropDownButton>
```

Note that the <Xaml:C1DropDown.Header> and <Xaml:C1DropDown.Content> tags are used to define header and content. You can add controls and content within these tags.

C1DropDown Interaction

Users can interact with items in the drop-down box, or with the C1DropDown control itself at run time. By default users can interact with controls placed within the drop-down box. For example, if you place a button or drop-down box within the **C1DropDown** control, it will be clickable by users at run time.

You can control the **C1DropDown** control's drop-down direction using the DropDownDirection property. You can see **Drop-Down Box Direction** for more information. You can choose if the **C1DropDown** box appears with the drop-down box automatically open using the IsDropDownOpen property. You can also set whether or not the drop-down box automatically closes when the users click outside of it – this can be set using the AutoClose property.

Drop-Down Box Direction

By default, when the user clicks the C1DropDown control's drop-down arrow at run-time the color picker will appear below the control, and, if that is not possible, above the control. However, you can customize where you would like the color picker to appear by setting the DropDownDirection property.

You can set the **DropDownDirection** property to one of the following options:

Event	**Description**
BelowOrAbove (default)	Tries to open the drop-down box below the header. If it is not possible tries to open above it.
AboveOrBelow	Tries to open the drop-down box above the header. If it is not possible tries to open below it.
ForceBelow	Forces the drop-down box to open below the header.

ForceAbove	Forces the drop-down box to open above the header.

For more information and an example, see **Changing the Drop-Down Direction**.

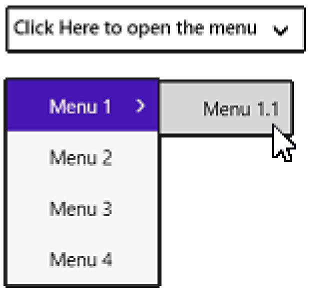
Additional Controls

In addition to the full-featured C1DropDown control, **DropDown for UWP** includes parts of the **C1DropDown** control and the C1DropDownButton that allow you to further customize your application.

C1DropDown is similar to a traditional drop-down control allowing you to choose from a selection of items and **C1DropDownButton** works like a drop-down control but looks like a button.

C1DropDownButton Elements

The C1DropDownButton control is similar to the C1DropDown control and consists of two parts: a header area and a content area. The header appears visible when the drop-down box is not open; the content is what is visible when the drop-down area is clicked. For example, in the below image the content area concludes a structured menu:



You can add content to neither, either, or both the header and content areas. You can customize the **C1DropDown** control in XAML by adding content to the header and content areas. For example the following markup creates a drop-down control similar to the one pictured above:

Markup

```
<Xaml:C1DropDownButton x:Name="ddl" Header="Click Here to open the menu"
HorizontalAlignment="Left" Xaml:C1NagScreen.Nag="True" Margin="451,301,0,391"
Grid.Row="1">
            <Xaml:C1MenuList>
                <Xaml:C1MenuItem Header="Menu 1">
                    <Xaml:C1MenuItem Header="Menu 1.1" />
                </Xaml:C1MenuItem>
                <Xaml:C1MenuItem Header="Menu 2" />
                <Xaml:C1MenuItem Header="Menu 3" />
                <Xaml:C1MenuItem Header="Menu 4" />
            </Xaml:C1MenuList>
        </Xaml:C1DropDownButton>
```

Note that the header text is defined in the <c1:C1DropDownButton> tag and the content is placed within the <c1:C1DropDownButton></c1:C1DropDownButton> tags.

DropDown for UWP Task-Based Help

The following task-based help topics assume that you are familiar with Visual Studio and know how to use the C1DropDown control in general. If you are unfamiliar with the **DropDown for UWP** product, please see the DropDown for UWP Quick Start first.

Each topic in this section provides a solution for specific tasks using the **DropDown for UWP** product. Most task-based help topics also assume that you have created a new Universal Windows application and added a **C1DropDown** control to the project – for information about creating the control, see **Creating a DropDown**.

Creating a DropDown

You can easily create a C1DropDown control at design time in XAML and in code. Note that if you create a **C1DropDown** control as in the following steps, it will appear empty. You will need to add items to the control. For an example, see **Adding Content to C1DropDown**.

In XAML

To create a **C1DropDown** control using XAML markup, complete the following steps:

1. In the Visual Studio Solution Explorer, right-click the **References** folder in the project files list. In the context menu choose **Add Reference**, select the **C1.UWP.dll** assembly, and click **OK**.
2. Add a XAML namespace to your project by adding xmlns:Xaml="using:C1.Xaml" to the initial <Page> tag. It will appear similar to the following:

Markup
```
<Page xmlns:Xaml="using:C1.Xaml"
    x:Class="DropDownTest.MainPage"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:DropDownTest"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
```

3. Set the name of Grid to LayoutRoot and add a <Xaml:C1DropDown> tag to your project within the <Grid> tag to create a **C1DropDown** control. The markup will appear similar to the following:

Markup
```
<Grid x:Name="LayoutRoot" Background="{ThemeResource
ApplicationPageBackgroundThemeBrush}">
     <Xaml:C1DropDown Name="c1dropdown1" />
</Grid>
```

This markup will create an empty C1DropDown control named "c1dropdown1" and set the control's size.

In Code

To create a C1DropDown control in code, complete the following steps:

1. In the Visual Studio Solution Explorer, right-click the **References** folder in the project files list. In the context menu choose **Add Reference**, select the **C1.Xaml.dll** assembly, and click OK.

2. Right-click within the MainPage.xaml window and select **View Code** to switch to Code view

3. Add the following import statement to the top of the page:

C#

```
using C1.Xaml;
```

4. Add code to the page's constructor to create the C1DropDown control. It will look similar to the following:

C#

```
C1DropDown c1dropdown1 = new C1DropDown();
c1dropdown1.Height = 30;
c1dropdown1.Width = 100;
LayoutRoot.Children.Add(c1dropdown1);
```

This code will create an empty **C1DropDown** control named "c1dropdown1", set the control's size, and add the control to the page.

What You've Accomplished

You've created a **C1DropDown** control. Note that when you create a **C1DropDown** control as in the above steps, it will appear empty. You can add items to the control that can be interacted with at run time. For an example, see **Adding Content to C1DropDown**.

Adding Content to C1DropDown

You can add any sort of arbitrary content to a C1DropDown control. This includes text, images, and other standard and 3rd-party controls. In this example, you'll add a **Button** control to a **C1DropDown** control, but you can customize the steps to add other types of content instead.

In XAML

For example, to add a **Button** control to the drop-down add <Button Height="30" Name="button1" Width="100">Hello World!</Button> within the <Xaml:C1DropDown> tag so that it appears similar to the following:

Markup

```
<Xaml:C1DropDown HorizontalAlignment="Center" VerticalAlignment="Top" Width="100">
  <Xaml:C1DropDown.Content>
    <Button Height="30" Name="button1" Width="100">Hello World!</Button>
  </Xaml:C1DropDown.Content>
</Xaml:C1DropDown>
```

In Code

For example, to add a **Button** control to the drop-down box, add code to the page's constructor so it appears like the following:

C#

```
public MainPage()
        {
            this.InitializeComponent();
            C1DropDown c1dropdown1 = new C1DropDown();
            c1dropdown1.Height = 30;
            c1dropdown1.Width = 100;
            LayoutRoot.Children.Add(c1dropdown1);

            Button c1button1 = new Button();
            c1button1.Content = "Hello World!";
            c1dropdown1.Content = c1button1;
        }
```

What You've Accomplished

You've added a button control to the **C1DropDown** control. Run the application and click the drop-down arrow. Observe that the Button control has been added to the drop-down box. Note that to add multiple items to the **C1DropDown** control, add a Grid or other panel to the **C1DropDown** control, and add items to that panel.

Changing the Drop-Down Direction

By default, when the user clicks the C1DropDown control's drop-down arrow at run-time the drop-down box will appear below the control, and if that is not possible, above the control. However, you can customize where you would like the color picker to appear. For more information about the drop-down arrow direction, see **Drop-Down Box Direction**.

To change the drop-down direction add DropDownDirection="ForceAbove" to the <Xaml:C1DropDown> tag so that it appears similar to the following:

In XAML

Markup

```
<Xaml:C1DropDown Height="30" Name="c1dropdown1" Width="100"
DropDownDirection="ForceAbove"/>
```

In Code

To change the drop-down box direction, add the following code to your project:

C#

```
c1dropdown1.DropDownDirection = DropDownDirection.ForceAbove;
```

Hiding the Drop-Down Arrow

By default, when you add the C1DropDown control to an application, the drop-down arrow, the **ToggleButton**, is visible. However, if you choose you can hide the drop-down arrow by setting the ShowButton property as in the following steps.

In XAML

To hide the drop-down arrow, add ShowButton="False" to the <Xaml:C1DropDown> tag so that it appears similar to the following:

Markup

```
<XamlC1DropDown Height="30" Name="c1dropdown1" Width="100" ShowButton="False" />
```

In Code

To hide the drop-down arrow, add the following code to your project:

C#
```
c1dropdown1.ShowButton = false;
```

Opening the Drop-Down on MouseOver

By default, the C1DropDown control's drop-down box only opens when users click on the drop-down arrow at run time. In this topic you'll set the drop-down box to open when users mouse over the control at run-time instead. Note that this topic assumes you have already added a **C1DropDown** control which contains content to the application.

Complete the following steps:

1. Click once on the **C1DropDown** control to select it.
2. Navigate to the Properties window and click on the **Events** lightning bolt button to view events associated with the control.
3. Double-click the box next to the **IsMouseOverChanged** item to switch to Code view and create the **C1DropDown_IsMouseOverChanged** event handler.
4. In Code view, add the following import statement to the top of the page:

   C#
   ```
   using C1.Xaml;
   ```

5. Add code to the **C1DropDown_IsMouseOver** event handler so that it looks like the following:

   C#
   ```
   private void C1DropDown_IsMouseOverChanged(object sender,
   PropertyChangedEventArgs<bool> e)
           {
               if (c1dropdown1.IsMouseOver == true)
               {
                   c1dropdown1.IsDropDownOpen = true;
               }
               else
               {
                   c1dropdown1.IsDropDownOpen = false;
               }
           }
   ```

What You've Accomplished

In this topic you added code so that the drop-down box opens when moused over at run time using the IsDropDownOpen property. Run the application and move the mouse over the control. Notice that the drop-down box opens. Move the mouse away from the control and observe that the drop-down box closes.

Creating a Hierarchical C1DropDown

You can easily create a C1DropDown application that contains a C1TreeView control, giving you a hierarchical drop-down.

Follow these steps to create a hierarchical **C1DropDown** control:

1. Edit the markup for your C1DropDown control so that it resembles the following:

   Markup
   ```
   <Xaml:C1DropDownButton x:Name="soccerCountries" HorizontalAlignment="Center"
   VerticalAlignment="Center" Padding="2" AutoClose="False" Width="150"
   DropDownWidth="200">
   ```

2. Add some C1DropDown.Header content to your control. This will add a **TextBlock** control to your **C1DropDown** header:

   Markup
   ```
   <Xaml:C1DropDownButton.Header>
                   <TextBlock x:Name="selection" Text="« Pick one »" Padding="7 0 0
   0" Foreground="Black" TextAlignment="left"/>
               </Xaml:C1DropDownButton.Header>
   ```

3. Place your cursor below the </Xaml:C1DropDownButton.Header> tag. Locate the **C1TreeView** control in your Visual Studio ToolBox and double-click to add it to your application. Edit the markup of the opening tag to reflect the following:

   Markup
   ```
   <Xaml:C1TreeView x:Name="treeSelection" Header="Soccer World Cups Winners"
   KeyDown="C1TreeView_KeyDown" ItemClick="C1TreeView_ItemClicked"
   AllowDragDrop="False" Padding="5" BorderBrush="{StaticResource
   ComboBoxPopupBorderThemeBrush}"           BorderThickness="{StaticResource
   ComboBoxPopupBorderThemeThickness}"            Background="{StaticResource
   ComboBoxPopupBackgroundThemeBrush}">
   ```

   Note that in the markup above you've added two Events: **KeyDown** and **ItemClick**.

4. Next, we'll add some content to the **C1TreeView** control:

   Markup
   ```
   <Xaml:C1TreeViewItem Header="South America">
      <Xaml:C1TreeViewItem Header="Argentina" />
      <Xaml:C1TreeViewItem Header="Brasil" />
      <Xaml:C1TreeViewItem Header="Uruguay" />
   </Xaml:C1TreeViewItem>
   <Xaml:C1TreeViewItem Header="Europe">
      <Xaml:C1TreeViewItem Header="England" />
      <Xaml:C1TreeViewItem Header="France" />
      <Xaml:C1TreeViewItem Header="Germany" />
      <Xaml:C1TreeViewItem Header="Italy" />
      <Xaml:C1TreeViewItem Header="Spain" />
   ```

```
</Xaml:C1TreeViewItem>
```

5. Now that you've finished creating the markup for your application, right-click the page and select **View Code** from the list. Your Code view will open.
6. Add the following import statement to the top of your page:

C#

```
using C1.Xaml;
```

7. Add the following **KeyDown** event after the closing bracket of the **InitializeComponent** method:

C#

```
private void C1TreeView_KeyDown(object sender, KeyRoutedEventArgs e)
        {
            if (e.Key == VirtualKey.Enter)
            {
                UpdateSelection();
                e.Handled = true;
            }
        }
```

8. The **UpdateSelection()** method comes next:

C#

```
private void UpdateSelection()
        {
            if (treeSelection.SelectedItem != null)
            {
                selection.Text = treeSelection.SelectedItem.Header.ToString();
            }
            else
            {
                selection.Text = "« Pick one »";
            }
            soccerCountries.IsDropDownOpen = false;
        }
```

9. Next, create the **C1TreeView_ItemClicked** event:

C#

```
private void C1TreeView_ItemClicked(object sender, SourcedEventArgs e)
        {
            UpdateSelection();
        }
```

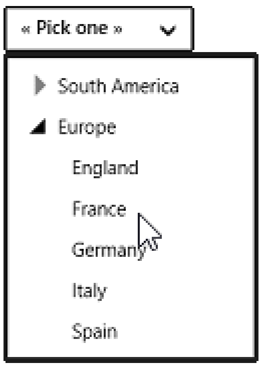
10. Last, add the **MouseLeftButtonDown** event:

C#

```
private void ContentControl_MouseLeftButtonDown(object sender, EventArgs e)
```

```
        {
            soccerCountries.IsDropDownOpen = true;
        }
```

11. Press F5 or start debugging to run your application. When you click the drop-down button, your **C1DropDown** control should resemble the following. Note that you can select one of the continents to reveal the available countries:



What You've Accomplished

In this topic, you created a hierarchical C1DropDown control using XAML markup and code.

GridSplitter for UWP

Redistribute space between columns and rows with the **GridSplitter for UWP**.

GridSplitter for UWP Quick Start

In this Quick Start, you create a new Universal Windows application in Visual Studio, set up a Grid, and add the C1GridSplitter control to your application.

Complete these steps:

1. Create a new Universal Windows application in Visual Studio:
    1. Select **File | New | Project** from the **File** menu. The **New Project** dialog box will open.
    2. Select Templates | Visual C# | Windows | Universal. From the templates list, select Blank App (Universal Windows).
    3. Give your project a **Name**, and select **OK**. Your new project will open.
2. Locate the **References** folder in the Solution Explorer. Right-click the folder and select **Add Reference**.
    1. In the **Reference Manager**, expand Universal Windows and select Extensions; you should see the UWP assemblies in the center pane.
    2. Select **C1.UWP.dll** from the center pane.
    3. Select **OK** to add the reference to your application.
3. Locate the **<Grid> </Grid>** tags in your **MainPage.xaml** file.
4. Add the following **Grid.RowDefinitions** and **Grid.ColumnDefinitions** to your application. The XAML markup should resemble the following:

XAML

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="100"/>
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="160"/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>

</Grid>
```

5. Place your cursor below the closing **</Grid.ColumnDefinition>** tag. Locate the **C1GridSplitter** control in your Visual Studio Toolbox, and add two **C1GridSplitter** controls to your application.
6. Edit your GridSplitter controls so that they resemble the following XAML markup:

XAML

```
<c1:C1GridSplitter Grid.Row="1" Grid.ColumnSpan="2" VerticalAlignment="Top"
Height="16" ShowsPreview="True" />
<c1:C1GridSplitter Grid.RowSpan="2" Grid.Column="1" HorizontalAlignment="Left"
Width="16" />
```

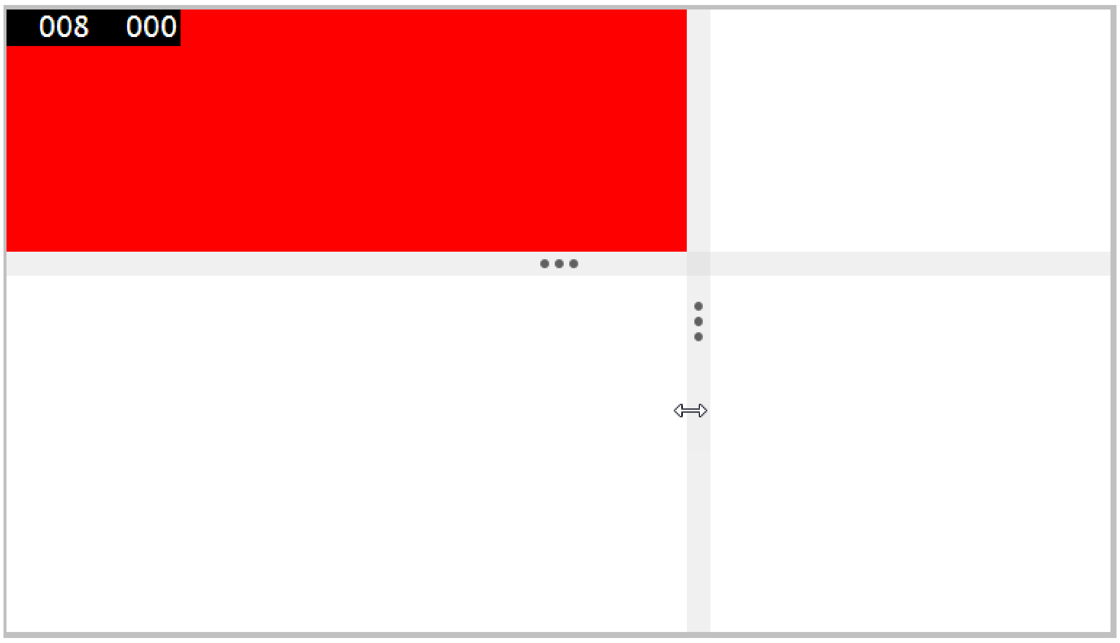
Note that you set the Grid Row and Column positions and the Vertical or Horizontal Alignment.

7. Directly below the second **C1GridSplitter** control, add a general **Rectangle** control from your Toolbox and set the **Fill** property to "Red":

XAML

```
<Rectangle Fill="Red"/>
```

8. Press **F5** or start debugging to run your application. Moving the horizontal and vertical GridSplitters will enlarge or minimize the red Rectangle:



Layout Panels for UWP

Control the flow and positioning of the content on your UWP app with **Layout Panels for UWP**. Wrap content vertically or horizontally using C1WrapPanel. Dock content along the edges of the panel with C1DockPanel. Display content in a grid using C1UniformGrid.

Layout Panels for UWP Features

Layout Panels for UWP includes the following key features:

- **Create Flowing Layouts**

  Create flow type layouts that wrap content vertically or horizontally using the C1WrapPanel control. This can be very useful for handling flow of items when the user rotates your app into portrait orientation.

- **Create Docked Layouts**

  Dock content along the top, left, right and bottom edges of the screen with the C1DockPanel control. Child elements are positioned in the dock panel in the order that they are declared in XAML.

- **Create Uniform Grid Layouts**

  Neatly display child elements in columns and rows with the C1UniformGrid control. By setting the **ColumnSpan** property you can span columns. Or, span rows by setting the **RowSpan** property. This resembles the built-in Microsoft Grid. Using **C1UniformGrid** you can also show or hide an entire column or row.

Layout Panels for UWP Quick Starts

A quick start has been created for each control in **Layout Panels for UWP**. In each quick start, you'll begin by creating a UWP application and then you will add styles for the controls. For **WrapPanel**, you will wrap several HyperlinkButtons. For **DockPanel**, you will create the panel with several elements inside, demonstrating the four docking options for C1DockPanel. For UniformGrid, you will create a grid with three columns and two empty cells in the first row.

Choose one of the following quick starts to get started:

- WrapPanel for UWP Quick Start
- DockPanel for UWP Quick Start
- UniformGrid for UWP Quick Start

WrapPanel for UWP Quick Start

The following quick start guide is intended to get you up and running with **WrapPanel for UWP**. In this quick start, you'll create a new project in Visual Studio, add styled HyperlinkButtons that can be wrapped, and change the orientation for the buttons.

Step 1 of 3: Creating an Application

In this step you'll begin in Visual Studio to create a UWP-style application using **WrapPanel for UWP**.

To set up your project, complete the following steps:

1. In Visual Studio, Select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Open **MainPage.xaml** if it isn't already open.

In the next step, you'll add, style, and wrap several **HyperlinkButtons**.

Step 2 of 3: Adding a C1WrapPanel to the Application

We're going to use simple **HyperlinkButtons** to show how content can be wrapped vertically or horizontally. This is the typical scenario to create a TagCloud view; very commonly used in Web applications.

Complete the following steps:

1. First, remove the **Grid** tags from your project.
2. Drag and drop **C1WrapPanel** control on the page. This adds the panel and a reference to the page.
3. Edit the <Xaml:C1WrapPanel> tags and add the **HyperlinkButtons** so the markup looks like the following:

Markup

```
 <Xaml:C1WrapPanel>
            <HyperlinkButton Content="Example Text" FontSize="25" />
            <HyperlinkButton Content="Longer sentences can be used for wrapping
scenarios." />
            <HyperlinkButton Content="Let's insert a break." />
            <HyperlinkButton Xaml:C1WrapPanel.BreakLine="After" Content="Break
After" />
            <HyperlinkButton Content="C1WrapPanel" />
            <HyperlinkButton Content="Wrap Vertically" />
            <HyperlinkButton Content="Wrap Horizontally" FontSize="20"  />
            <HyperlinkButton Xaml:C1WrapPanel.BreakLine="Before" Content="Break
Before" />
```

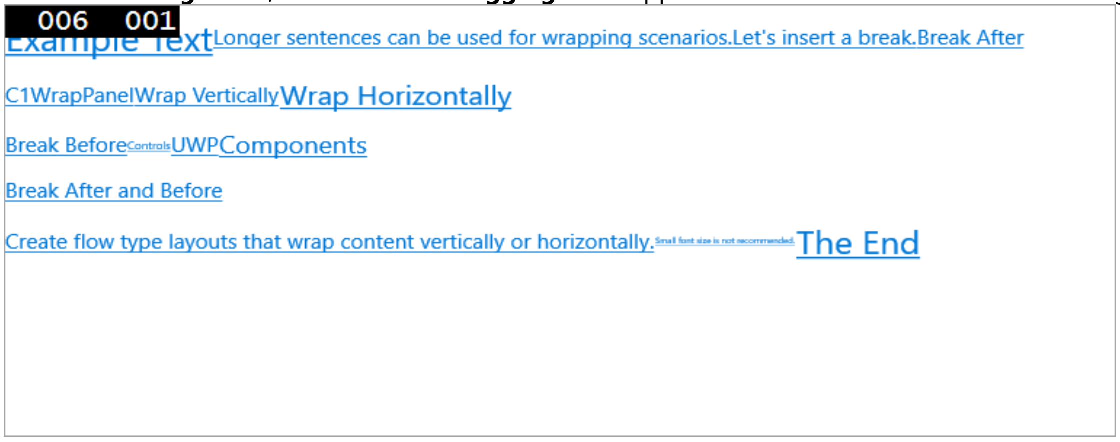
```
            <HyperlinkButton Content="Controls" FontSize="8" />
            <HyperlinkButton Content="UWP" />
            <HyperlinkButton Content="Components" FontSize="18" />
            <HyperlinkButton Xaml:C1WrapPanel.BreakLine="AfterAndBefore"
Content="Break After and Before" />
            <HyperlinkButton Content="Create flow type layouts that wrap content
vertically or horizontally." />
            <HyperlinkButton Content="Small font size is not recommended."
FontSize="6" />
            <HyperlinkButton Content="The End" FontSize="24" />
    </Xaml:C1WrapPanel>
```

In the next step, you'll run the application.

Step 3 of 3: Running the Application

Now you're ready to run the application. Complete the following steps:

1. From the **Debug** menu, select **Start Debugging**. Your application will look similar to the following:



006 001
Example TextLonger sentences can be used for wrapping scenarios.Let's insert a break.Break After

C1WrapPanelWrap VerticallyWrap Horizontally

Break BeforeControlsUWPComponents

Break After and Before

Create flow type layouts that wrap content vertically or horizontally.Small font size is not recommended.The End

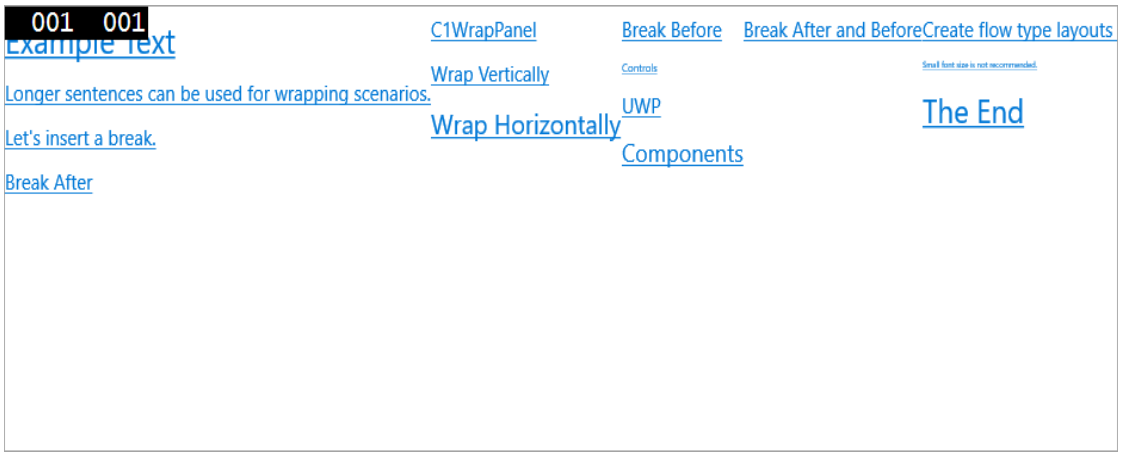
2. Click the **Stop Debugging** button to close the application.

3. Go back to **MainPage.xaml**. In the <Xaml:C1WrapPanel> tag, set the **Orientation** property to **Vertical**; the XAML will look like the following:

Markup

```
<Xaml:C1WrapPanel Orientation="Vertical">
```

4. Click **Start Debugging** again in the **Debug** menu. Your application will now look like this:



Notice how the buttons are stacked vertically.

Congratulations! You have successfully completed the **WrapPanel for UWP** quick start.

DockPanel for UWP Quick Start

The following quick start guide is intended to get you up and running with **DockPanel for UWP**. In this quick start, you'll create a new project in Visual Studio, and add elements docked on the top, bottom, left, and right of the C1DockPanel.

Step 1 of 3: Creating an Application

In this step you'll begin in Visual Studio to create a UWP-style application using **DockPanel for UWP**.

To set up your project, complete the following steps:

1. Select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Open **MainPage.xaml** if it isn't already open.

In the next step, you'll add and style **C1DockPanels**.

Step 2 of 3: Adding a C1DockPanel to the Application

In this step you'll add and style several **C1DockPanels**.

Complete the following steps:

1. First, remove the **Grid** tags from your project.
2. Drag and drop **C1DockPanel** control on the page. This adds the panel and a reference to the page.
3. Edit the <Xaml:C1DockPanel> tags to add docked borders on the left, right, top, and bottom of the screen so the markup looks like the following:

Markup

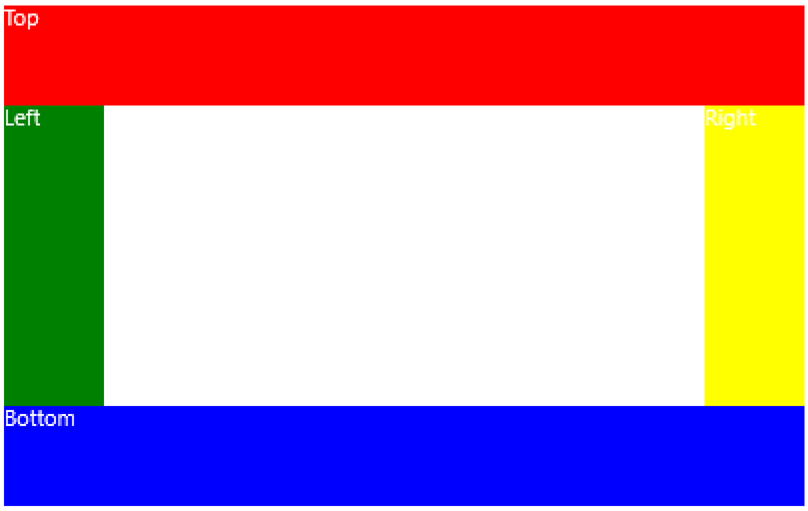
```
<Xaml:C1DockPanel Background="White" Width="400" Height="250">
```

```
    <Border Xaml:C1DockPanel.Dock="Top" Height="50" Background="Red">
       <TextBlock Text="Top" />
    </Border>
    <Border Xaml:C1DockPanel.Dock="Bottom" Height="50" Background="Blue">
       <TextBlock Text="Bottom" />
    </Border>
    <Border Xaml:C1DockPanel.Dock="Right" Width="50" Background="Yellow">
       <TextBlock Text="Right" />
    </Border>
    <Border Xaml:C1DockPanel.Dock="Left" Width="50" Background="Green">
       <TextBlock Text="Left" />
    </Border>
</Xaml:C1DockPanel>
```

In the next step, you'll run the application.

Step 3 of 3: Running the Application

Now you're ready to run the application. From the **Debug** menu, select **Start Debugging**. Your application will look similar to the following, with four borders docked on the top, right, left, and bottom of the C1DockPanel control:



Congratulations! You have successfully completed the **DockPanel for UWP** quick start.

UniformGrid for UWP Quick Start

The following quick start guide is intended to get you up and running with **UniformGrid for UWP**. In this quick start, you'll create a new project in Visual Studio, add a C1UniformGrid to your application, set the C1UniformGrid.Columns, C1UniformGrid.FirstColumn, and Width properties, and then run the application.

Step 1 of 3: Creating a UWP Application

In this step you'll begin in Visual Studio to create a UWP-style application using **UniformGrid for UWP**.

To set up your project, complete the following steps:

1. Select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Open **MainPage.xaml** if it isn't already open.

In the next step, you'll add, style, and wrap several border blocks.

Step 2 of 3: Adding the C1UniformGrid control to the Application

In this step you'll add a C1UniformGrid.

Complete the following steps:

1. First, remove the **Grid** tags from your project.
2. Drag and drop **C1UniformGrid** control on the page. This adds the panel and a reference to the page.
3. Edit the <Xaml:C1UniformGrid> tags to size the grid and add child elements to the grid (numbered cells in this case) so the markup looks like the following:

Markup

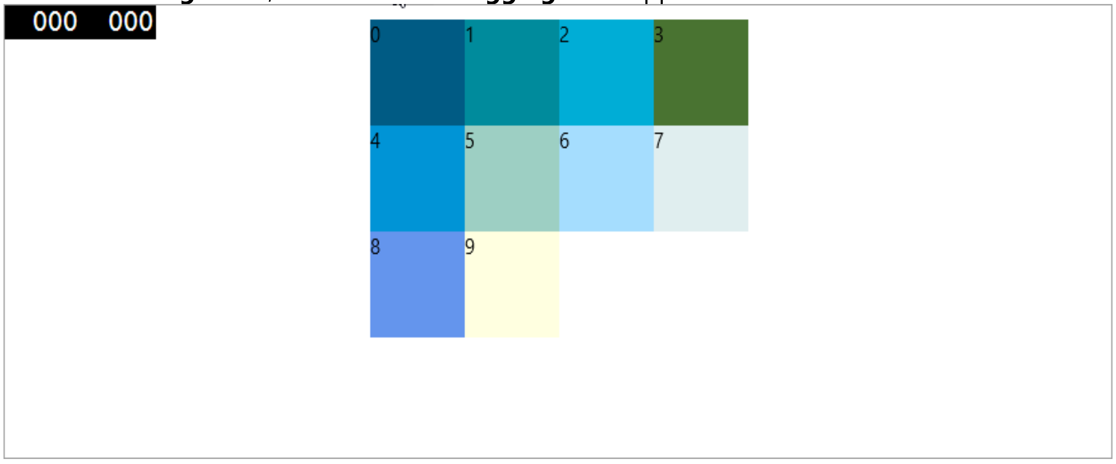
```
<Xaml:C1UniformGrid Width="300" Height="300">
            <Border Background="#FF005B84" >
                <TextBlock Text="0" />
            </Border>
            <Border Background="#FF008B9C" >
                <TextBlock Text="1" />
            </Border>
            <Border Background="#FF00ADD6" >
                <TextBlock Text="2" />
            </Border>
            <Border Background="#FF497331" >
                <TextBlock Text="3" />
            </Border>
            <Border Background="#FF0094D6" >
                <TextBlock Text="4" />
            </Border>
            <Border Background="#FF9DCFC3" >
                <TextBlock Text="5" />
            </Border>
            <Border Background="#FFA5DDFE" >
                <TextBlock Text="6" />
            </Border>
            <Border Background="#FFE0EEEF" >
                <TextBlock Text="7" />
            </Border>
            <Border Background="CornflowerBlue" >
                <TextBlock Text="8" />
            </Border>
            <Border Background="LightYellow" >
                <TextBlock Text="9" />
            </Border>
    </Xaml:C1UniformGrid>
```

In this step you added content to the C1UniformGrid panel. In the next step you'll run the application and edit some additional properties of the grid.

Step 3 of 3: Running the Application

Now you're ready to run the application. Complete the following steps:

1. From the **Debug** menu, select **Start Debugging**. Your application will look similar to the following:



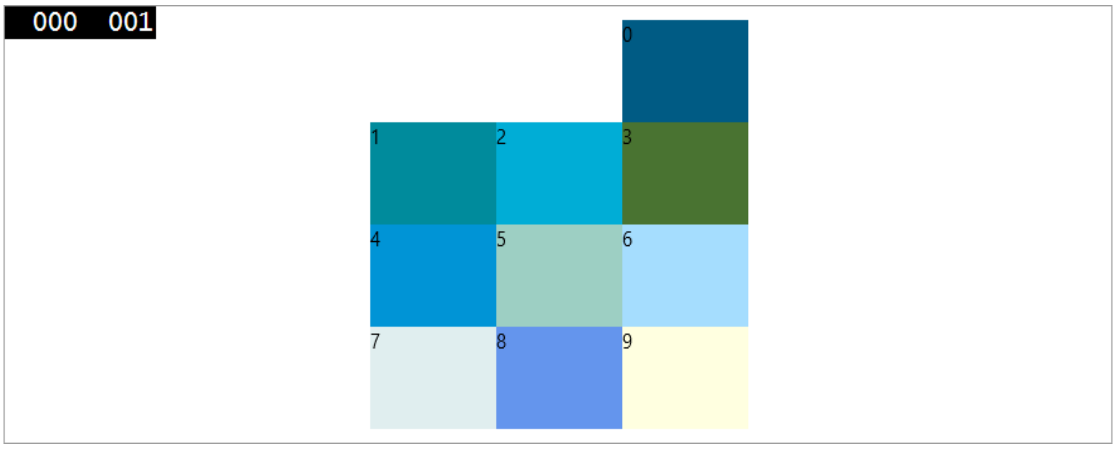
2. Click the **Stop Debugging** button to close the application.
3. Go back to **MainPage.xaml** and place your cursor in the <Xaml:C1UniformGrid> tag.
4. Set the C1UniformGrid.Columns and C1UniformGrid.FirstColumn properties using the following XAML markup:

Markup

```
<Xaml:C1UniformGrid Width="300" Height="300" Columns="3" FirstColumn="2" >
```

The C1UniformGrid.Columns will set the number of columns in the grid, the **Width** property will set the width, in pixels, and the C1UniformGrid.FirstColumn property will determine how many empty cells will appear in the first row.

5. From the **Debug** menu, select **Start Debugging**. Your application will look similar to the following:



Notice the two empty cells in the first row, as specified with the C1UniformGrid.FirstColumn property.

Congratulations! You have successfully completed the **UniformGrid for UWP** quick start.

Layout Panels for UWP Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET and know how to use the Layout Panels in general. If you are unfamiliar with the **Layout Panels for UWP** product, please see the Layout Panels for UWP Quick Starts first.

Each topic in this section provides a solution for specific tasks using the **Layout Panels for UWP** product.

Each task-based help topic also assumes that you have created a new UWP project.

Wrapping and Formatting Items with C1WrapPanel

You can wrap items using the **C1WrapPanel.BreakLine Attached** property. In this example, **HyperlinkButtons** are used. Complete the following steps:

1. In your project, drag a C1WrapPanel control from the Toolbox and place it before the closing </Grid> tag in the .xaml page.
2. Place your cursor in between the <Xaml:C1WrapPanel> tags and press ENTER.
3. Add the following XAML to wrap **HyperlinkButtons**:

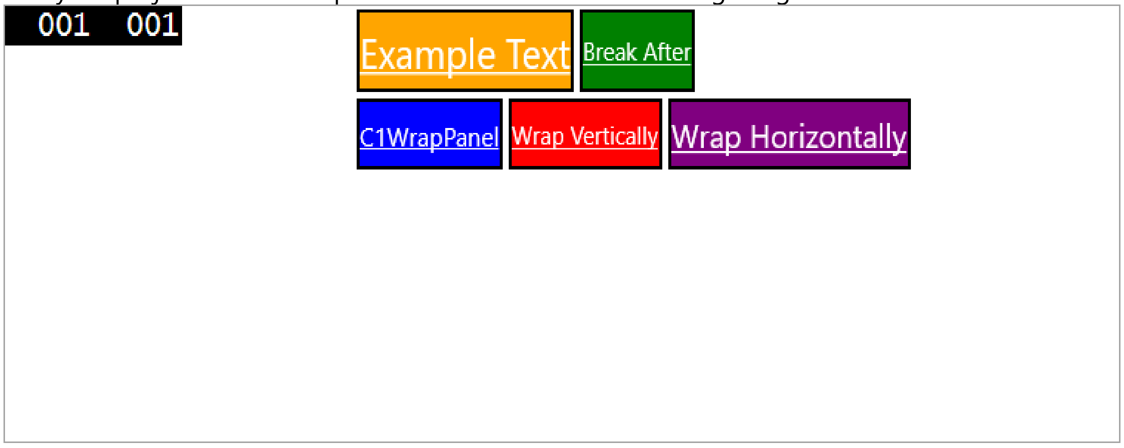
Markup

```
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Orange">
    <HyperlinkButton Foreground="White"  Content="Example Text" FontSize="25" />
</Border>
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Green"
Xaml:C1WrapPanel.BreakLine="After">
    <HyperlinkButton Foreground="White" Content="Break After" />
</Border>
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Blue">
    <HyperlinkButton Foreground="White" Content="C1WrapPanel" FontSize="16"/>
</Border>
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Red">
    <HyperlinkButton Foreground="White" Content="Wrap Vertically" />
</Border>
```

```
<Border Margin="2" BorderBrush="Black" BorderThickness="2"  Background="Purple">
    <HyperlinkButton Foreground="White" Content="Wrap Horizontally"
FontSize="20" />
</Border>
```

Notice the **C1WrapPanel.BreakLine** property is set to **After** for the second **HyperlinkButton**. This will add a break after the button.

4. Run your project. The C1WrapPanel will resemble the following image:



Notice there is a break after the second **HyperlinkButton**.

Wrapping Items Vertically with C1WrapPanel

By default, items are wrapped horizontally. However, in some cases you may need them to wrap vertically. You can set the C1WrapPanel.Orientation property to specify vertical wrapping. In this example, **HyperlinkButtons** are used. Complete the following steps:

1. In your project, drag a C1WrapPanel control from the Toolbox and place it before the closing </Grid> tag in the .xaml page.
2. In the <Xaml:C1WrapPanel> tag, set the **Orientation** property to **Vertical**; the XAML will look like the following:

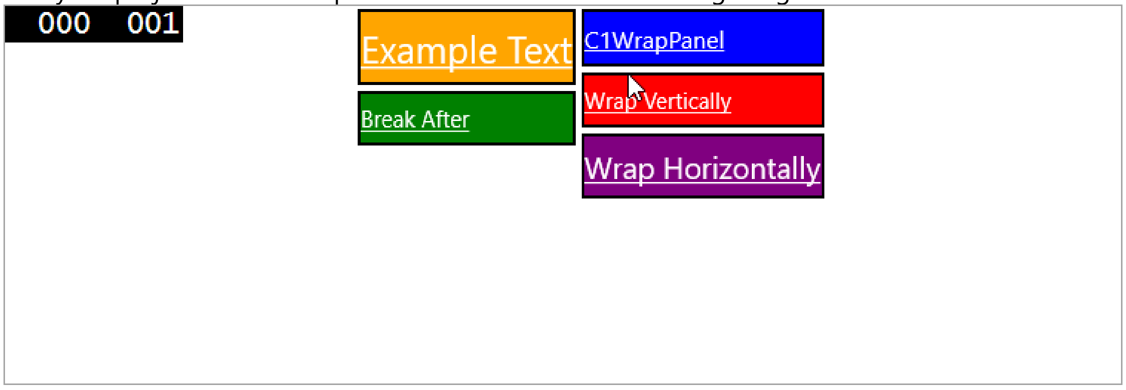
Markup
```
<Xaml:C1WrapPanel Orientation="Vertical">
```

3. Place your cursor in between the <Xaml:C1WrapPanel> tags and press ENTER.
4. Add the following XAML to wrap HyperlinkButtons:

Markup
```
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Orange">
    <HyperlinkButton Foreground="White" Content="Example Text" FontSize="25" />
</Border>
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Green"
Xaml:C1WrapPanel.BreakLine="After">
    <HyperlinkButton Foreground="White" Content="Break After" />
</Border>
```

```
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Blue">
    <HyperlinkButton Foreground="White" Content="C1WrapPanel" FontSize="16"/>
</Border>
<Border Margin="2" BorderBrush="Black" BorderThickness="2" Background="Red">
    <HyperlinkButton Foreground="White" Content="Wrap Vertically" />
</Border>
<Border Margin="2" BorderBrush="Black" BorderThickness="2"  Background="Purple">
    <HyperlinkButton Foreground="White" Content="UWP" FontSize="20"/>
</Border>
```

5. Run your project. The C1WrapPanel will resemble the following image:



ListBox for UWP

Get two high performance controls for displaying lists of bound data with **ListBox for UWP**. Display lists with tile layouts or with optical zoom using the C1ListBox and C1TileListBox controls. These controls support UI virtualization so they are blazing-fast while able to display thousands of items with little-to-no loss of performance.

ListBox for UWP Key Features

ListBox for UWP's key features include the following:

- **Horizontal or Vertical Orientation**

  The **ListBox** controls support both horizontal and vertical orientation, allowing for more layout scenarios.

- **Display Items as Tiles**

  The C1TileListBox can arrange its items in both rows and columns creating tile displays. Specify the size and template of each item and select the desired orientation.

- **Optical Zoom**

  The C1ListBox control supports optical zoom functionality so users can manipulate the size of the items intuitively through pinch gestures. The zooming transformation is smooth and fluid so the performance of your application is not sacrificed.

- **UI Virtualization**

  The **ListBox** controls support UI virtualization so they are blazing-fast while able to display thousands of items with virtually no loss of performance. You can determine how many items are rendered in each layout pass by setting the ViewportGap and ViewportPreviewGap properties. These properties can be adjusted depending on the scenario.

- **Preview State**

  In order to have the highest performance imaginable, the **ListBox** controls can render items outside the viewport in a preview state. Like the standard **ItemTemplate**, the **Preview** template defines the appearance of items when they are in a preview state, such as zoomed out or during fast scroll. The controls will then switch to the full item template when the items have stopped scrolling or zooming.

C1ListBox Quick Start

The following quick start guide is intended to get you up and running with the C1ListBox control. In this quick start you'll start in Visual Studio and create a new project, add C1ListBox to your application, and customize the appearance and behavior of the control.

Step 1 of 3: Creating an Application with a C1ListBox Control

In this step, you'll create a UWP application in Visual Studio using **ListBox for UWP**.

Complete the following steps:

1. In Visual Studio, select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Right-click the project name in the Solution Explorer and select **Add Reference**.
4. In the **Reference Manager** dialog box, expand Universal Windows and select Extensions; you should see the UWP assemblies in the center pane. Select the C1.UWP and C1.UWP.Tile assemblies and click OK.
5. Open **MainPage.xaml** if it isn't already open, and add the following markup within the <Page> tag:

   Markup
   ```
   xmlns:c1="using:C1.Xaml"
   xmlns:c1tile="using:C1.Xaml.Tile"
   ```

6. Place the cursor between the <Grid> and </Grid> tags, and click once.
7. Add the following <StackPanel> markup between the <Grid> and </Grid> tags to add a **StackPanel** containing a **TextBlock** and **ProgressBar**:

   Markup
   ```
   <StackPanel x:Name="loading" VerticalAlignment="Center">
       <TextBlock Text="Retrieving data from Flickr..." TextAlignment="Center"/>
       <ProgressBar IsIndeterminate="True" Width="200" Height="4"/>
   </StackPanel>
   ```

   The **TextBlock** and **ProgressBar** will indicate that the **C1ListBox** is loading.

8. Navigate to the Toolbox and double-click the C1ListBox icon to add the control to the grid. This will add the reference and XAML namespace automatically.
9. Edit the <Xaml:C1ListBox> tag to customize the control:

Markup

```
<c1:C1ListBox x:Name="listBox" ItemsSource="{Binding}" Background="Transparent"
Visibility="Collapsed" ItemWidth="800" ItemHeight="600"
RefreshWhileScrolling="False"></c1:C1ListBox>
```

   This gives the control a name and customizes the binding, background, visibility, size, and refreshing ability of the control.

10. Add the following markup between the <Xaml:C1ListBox> and </Xaml:C1ListBox> tags:

Markup

```
<c1:C1ListBox.PreviewItemTemplate>
  <DataTemplate>
    <Grid Background="Gray">
      <Image Source="{Binding Thumbnail}" Stretch="UniformToFill" />
    </Grid>
  </DataTemplate>
</c1:C1ListBox.PreviewItemTemplate>
<c1:C1ListBox.ItemTemplate>
  <DataTemplate>
    <Grid>
      <Image Source="{Binding Content}" Stretch="UniformToFill" />
      <TextBlock Text="{Binding Title}" Foreground="White" Margin="4 0 0 4"
VerticalAlignment="Bottom" />
    </Grid>
  </DataTemplate>
</c1:C1ListBox.ItemTemplate>
```

   This markup adds data templates for the **C1ListBox** control's content. Note that you'll complete binding the control in code.

What You've Accomplished

You've successfully created a UWP style application containing a **C1ListBox** control. In the next step, Step 2 of 3: Adding Data to the ListBox, you will add the data for **C1ListBox**.

Step 2 of 3: Adding Data to the ListBox

In the last step, you added the C1ListBox control to the application. In this step, you will add code to display images from a photo stream.

Complete the following steps to add data to the control programmatically:

1. Select the **Page**, navigate to the Properties window, click the lightning bolt **Events** button to view events, and scroll down and double-click the area next to the **Loaded** event.

   This will open the Code Editor and add the **Page_Loaded** event.

2. Add the following imports statements to the top of the page:

Visual Basic

```
Imports C1.Xaml
Imports System
Imports System.Collections.Generic
Imports System.Linq
Imports System.Net
Imports System.Xml.Linq
Imports Windows.UI.Popups
Imports Windows.UI.Xaml
Imports Windows.UI.Xaml.Controls
```

C#

```
using C1.Xaml;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Xml.Linq;
using Windows.UI.Popups;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
```

3. Add the following code inside the **Page_Loaded** event handler:

Visual Basic

```
LoadPhotos()
```

C#

```
LoadPhotos();
```

4. Add the following code below the **Page_Loaded** event within the **MainPage** class:

Visual Basic

```
Private Async Function LoadPhotos() As Task
        Dim flickrUrl = "http://api.flickr.com/services/feeds/photos_public.gne"
        Dim AtomNS = "http://www.w3.org/2005/Atom"

        Dim photos = New List(Of Photo)()
        Dim client = WebRequest.CreateHttp(New Uri(flickrUrl))
        Dim response = Await client.GetResponseAsync()

        Try
            '#Region "** parse data"
            Dim doc = XDocument.Load(response.GetResponseStream())
            For Each entry As XElement In doc.Descendants(XName.[Get]("entry",
AtomNS))
                Dim title = entry.Element(XName.[Get]("title", AtomNS)).Value
                Dim enclosure = entry.Elements(XName.[Get]("link",
AtomNS)).Where(Function(elem) elem.Attribute("rel").Value =
"enclosure").FirstOrDefault()
                Dim contentUri = enclosure.Attribute("href").Value
                photos.Add(New Photo() With { _
                    .Title = title, _
                    .Content = contentUri, _
                    .Thumbnail = contentUri.Replace("_b", "_m") _
                })
            Next
            '#End Region

            listBox.ItemsSource = photos
            loading.Visibility = Visibility.Collapsed
            listBox.Zoom = C1ZoomUnit.Fill
            listBox.Visibility = Visibility.Visible
        Catch
            Dim dialog = New MessageDialog("There was an error when attempting
to download data from Flickr.")
            async dialog.ShowAsync()
        End Try
    End Function
```

C#

```
private async void LoadPhotos()
        {
            var flickrUrl =
"http://api.flickr.com/services/feeds/photos_public.gne";
            var AtomNS = "http://www.w3.org/2005/Atom";

            var photos = new List<Photo>();
            var client = WebRequest.CreateHttp(new Uri(flickrUrl));
            var response = await client.GetResponseAsync();

            try
            {
                #region ** parse data
                var doc = XDocument.Load(response.GetResponseStream());
                foreach (var entry in doc.Descendants(XName.Get("entry",
AtomNS)))
                {
                    var title = entry.Element(XName.Get("title", AtomNS)).Value;

                    var enclosure = entry.Elements(XName.Get("link",
AtomNS)).Where(elem => elem.Attribute("rel").Value ==
"enclosure").FirstOrDefault();
                    var contentUri = enclosure.Attribute("href").Value;
                    photos.Add(new Photo() { Title = title, Content =
contentUri, Thumbnail = contentUri.Replace("_b","_m") });
                }
                #endregion

                listBox.ItemsSource = photos;
                loading.Visibility = Visibility.Collapsed;
                listBox.Zoom = C1ZoomUnit.Fill;
                listBox.Visibility = Visibility.Visible;
            }
            catch
            {
                var dialog = new MessageDialog("There was an error when
attempting to download data from Flickr.");
                async dialog.ShowAsync();
            }
        }
```

5. The code above pulls images from Flickr's public photo stream and binds the **C1ListBox** to the list of images.

4. Add the following code just below the **MainPage** class:

Visual Basic

```
Public Class Photo
        Public Property Title() As String
            Get
                Return m_Title
            End Get
            Set(value As String)
                m_Title = Value
            End Set
        End Property
        Private m_Title As String
        Public Property Thumbnail() As String
            Get
                Return m_Thumbnail
            End Get
            Set(value As String)
                m_Thumbnail = Value
            End Set
        End Property
        Private m_Thumbnail As String
        Public Property Content() As String
            Get
                Return m_Content
            End Get
            Set(value As String)
                m_Content = Value
            End Set
        End Property
        Private m_Content As String
    End Class
```

C#

```
public class Photo
    {
        public string Title { get; set; }
        public string Thumbnail { get; set; }
        public string Content { get; set; }
    }
```

What You've Accomplished

You have successfully added data to C1TileListBox. In the next step, Step 3 of 3: Running the ListBox Application, you'll examine the **ListBox for UWP** features.

Step 3 of 3: Running the ListBox Application

Now that you've created a UWP style application and customized the application's appearance and behavior, the only thing left to do is run your application. To run your application and observe **ListBox for UWP**'s run-time behavior, complete the following steps:

1. From the **Debug** menu, select **Start Debugging** to view how your application will appear at run time.

   The application will appear, displaying an image.

2. Use the scroll bar on the right of the control, to scroll through the image stream.

3. If you have touch capabilities, try pinching to zoom an image.

What You've Accomplished

Congratulations! You've completed the **ListBox for UWP** quick start and created an application using the **C1ListBox** control and viewed some of the run-time capabilities of your application.

C1TileListBox Quick Start

The following quick start guide is intended to get you up and running with the C1TileListBox control. In this quick start you'll start in Visual Studio and create a new project, add C1TileListBox to your application, and customize the appearance and behavior of the control.

Step 1 of 3: Creating an Application with a C1TileListBox Control

In this step, you'll create a UWP application in Visual Studio using **TileListBox for UWP**.

Complete the following steps:

1. In Visual Studio select **File | New | Project**.

2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.

3. Open **MainPage.xaml** if it isn't already open, place the cursor between the <Grid> and </Grid> tags, and click once.

4. Add the following <StackPanel> markup between the <Grid> and </Grid> tags to add a **StackPanel**

containing a **TextBlock** and **ProgressBar**:

Markup

```
<StackPanel x:Name="loading" VerticalAlignment="Center">
    <TextBlock Text="Retrieving data from YouTube..." TextAlignment="Center"/>
    <ProgressBar IsIndeterminate="True" Width="200" Height="4"/>
 </StackPanel>
```

The **TextBlock** and **ProgressBar** will indicate that the **C1TileListBox** is loading.

5. Navigate to the Toolbox and double-click the C1TileListBox icon to add the control to the grid. This will add the reference and XAML namespace automatically.
6. Edit the <Xaml:C1TileListBox> tag to customize the control:

Markup

```
<Xaml:C1TileListBox x:Name="tileListBox" ItemsSource="{Binding}"
Background="Transparent" Visibility="Collapsed" ItemWidth="800" ItemHeight="600"
RefreshWhileScrolling="False"></Xaml:C1TileListBox>
```

This gives the control a name and customizes the binding, background, visibility, size, and refreshing ability of the control.

7. Add the following markup between the **<Xaml:C1TileListBox>** and **</Xaml:C1TileListBox>** tags:

Markup

```
<Xaml:C1TileListBox.PreviewItemTemplate>
        <DataTemplate>
            <Grid Background="Gray"/>
        </DataTemplate>
            </Xaml:C1TileListBox.PreviewItemTemplate>
            <Xaml:C1TileListBox.ItemTemplate>
        <DataTemplate>
        <Grid Background="LightBlue">
            <Image Source="{Binding Thumbnail}" Stretch="UniformToFill"/>
            <TextBlock Text="{Binding Title}" Foreground="White" Margin="4 0 0
4" VerticalAlignment="Bottom"/>
            </Grid>
        </DataTemplate>
    </Xaml:C1TileListBox.ItemTemplate>
```

This markup adds data templates for the **C1TileListBox** control's content. Note that you'll complete binding the control in code.

✓ **What You've Accomplished**

You've successfully created a UWP style application containing a **C1TileListBox** control. In the next step, Step 2 of 3: Adding Data to the TileListBox, you will add the data for **C1TileListBox**.

Step 2 of 3: Adding Data to the TileListBox

In the last step, you added the C1TileListBox control to the application. In this step, you will add code to display images from a photo stream.

Complete the following steps to add data to the control programmatically:

1. Select the **Page**, navigate to the Properties window, click the lightning bolt **Events** button to view events, and scroll down and double-click the area next to the **Loaded** event.

   This will open the Code Editor and add the **Page_Loaded** event.

2. Add the following imports statements to the top of the page:

   Visual Basic

   ```
   Imports C1.Xaml
   Imports System
   Imports System.Collections.Generic
   Imports System.Linq
   Imports System.Net
   Imports System.Xml.Linq
   Imports Windows.UI.Popups
   Imports Windows.UI.Xaml
   Imports Windows.UI.Xaml.Controls
   ```

   C#

   ```
   using C1.Xaml;
   using System;
   using System.Collections.Generic;
   using System.Linq;
   using System.Net;
   using System.Xml.Linq;
   using Windows.UI.Popups;
   using Windows.UI.Xaml;
   using Windows.UI.Xaml.Controls;
   ```

3. Add the following code inside the **Page_Loaded** event handler:

   Visual Basic

   ```
   LoadVideos()
   ```

   C#

   ```
   LoadVideos();
   ```

4. Add the following code below the **Page_Loaded** event within the **MainPage** class:

   Visual Basic

   ```
   Private Async Function LoadVideos() As Task
   Dim youtubeUrl = "https://gdata.youtube.com/feeds/api/videos?q=windows+8&max-
   results=50"
   Dim AtomNS = "http://www.w3.org/2005/Atom"
   Dim MediaNS = "http://search.yahoo.com/mrss/"
   ```

```
Dim videos = New List(Of Video)()
Dim client = WebRequest.CreateHttp(New Uri(youtubeUrl))
Dim response = Await client.GetResponseAsync()

Try
   #Region "** parse you tube data"
   Dim doc = XDocument.Load(response.GetResponseStream())
   For Each entry As var In doc.Descendants(XName.[Get]("entry", AtomNS))
         Dim title = entry.Element(XName.[Get]("title", AtomNS)).Value
         Dim thumbnail = ""
         Dim group = entry.Element(XName.[Get]("group", MediaNS))
         Dim thumbnails = group.Elements(XName.[Get]("thumbnail", MediaNS))
         Dim thumbnailElem = thumbnails.FirstOrDefault()
         If thumbnailElem IsNot Nothing Then
               thumbnail = thumbnailElem.Attribute("url").Value
         End If
         Dim alternate = entry.Elements(XName.[Get]("link",
AtomNS)).Where(Function(elem) elem.Attribute("rel").Value =
"alternate").FirstOrDefault()
         Dim link = alternate.Attribute("href").Value
         videos.Add(New Video() With { _
               Key .Title = title, _
               Key .Link = link, _
               Key .Thumbnail = thumbnail _
         })
   Next
   #End Region

   tileListBox.ItemsSource = videos
   loading.Visibility = Visibility.Collapsed
   tileListBox.Visibility = Visibility.Visible
Catch
   Dim dialog = New MessageDialog("There was an error when attempting to
download data from you tube.")
   dialog.ShowAsync()
End Try
End Function
```

C#

```
private async void LoadVideos()
        {
            var youtubeUrl = "https://gdata.youtube.com/feeds/api/videos?
q=windows+8&max-results=50";
            var AtomNS = "http://www.w3.org/2005/Atom";
            var MediaNS = "http://search.yahoo.com/mrss/";

            var videos = new List<Video>();
            var client = WebRequest.CreateHttp(new Uri(youtubeUrl));
            var response = await client.GetResponseAsync();

            try
            {
                #region ** parse you tube data
                var doc = XDocument.Load(response.GetResponseStream());
                foreach (var entry in doc.Descendants(XName.Get("entry",
AtomNS)))
                {
                    var title = entry.Element(XName.Get("title", AtomNS)).Value;
                    var thumbnail = "";
                    var group = entry.Element(XName.Get("group", MediaNS));
                    var thumbnails = group.Elements(XName.Get("thumbnail",
MediaNS));
                    var thumbnailElem = thumbnails.FirstOrDefault();
                    if (thumbnailElem != null)
                        thumbnail = thumbnailElem.Attribute("url").Value;
                    var alternate = entry.Elements(XName.Get("link",
AtomNS)).Where(elem => elem.Attribute("rel").Value ==
"alternate").FirstOrDefault();
                    var link = alternate.Attribute("href").Value;
                    videos.Add(new Video() { Title = title, Link = link,
Thumbnail = thumbnail });
                }
                #endregion

                tileListBox.ItemsSource = videos;
                loading.Visibility = Visibility.Collapsed;
                tileListBox.Visibility = Visibility.Visible;
            }
            catch
            {
                var dialog = new MessageDialog("There was an error when
attempting to download data from you tube.");
                dialog.ShowAsync();
            }
}
```

5. The code above pulls images from YouTube and binds the **C1TileListBox** to the list of videos.
6. Add the following code below the code you just added within the **MainPage** class:

Visual Basic

```
Private Sub tileListBox_ItemClick(sender As Object, e As EventArgs)
   Dim video = TryCast(TryCast(sender, C1ListBoxItem).Content, Video)
   NavigateUrl(video.Link)
End Sub
#Region "** public properties"

Public Property Orientation() As Orientation
   Get
         Return tileListBox.Orientation
   End Get
```

```
    Set
          tileListBox.Orientation = value
    End Set
End Property

Public Property ItemWidth() As Double
    Get
          Return tileListBox.ItemWidth
    End Get
    Set
          tileListBox.ItemWidth = value
    End Set
End Property

Public Property ItemHeight() As Double
    Get
          Return tileListBox.ItemHeight
    End Get
    Set
          tileListBox.ItemHeight = value
    End Set
End Property
```

C#

```
private void tileListBox_ItemClick(object sender, EventArgs e)
        {
            var video = (sender as C1ListBoxItem).Content as Video;
            NavigateUrl(video.Link);
        }


        #region ** public properties

        public Orientation Orientation
        {
            get
            {
                return tileListBox.Orientation;
            }
            set
            {
                tileListBox.Orientation = value;
            }
        }

        public double ItemWidth
        {
            get
            {
                return tileListBox.ItemWidth;
            }
            set
            {
                tileListBox.ItemWidth = value;
            }
        }

        public double ItemHeight
        {
            get
            {
                return tileListBox.ItemHeight;
            }
            set
            {
                tileListBox.ItemHeight = value;
            }
}
```

7. Add the following code just below the **MainPage** class:

Visual Basic

```
Public Class Video
   Public Property Title() As String
         Get
               Return m_Title
         End Get
         Set
               m_Title = Value
         End Set
   End Property
   Private m_Title As String
   Public Property Thumbnail() As String
         Get
               Return m_Thumbnail
         End Get
         Set
               m_Thumbnail = Value
         End Set
   End Property
   Private m_Thumbnail As String
   Public Property Link() As String
         Get
               Return m_Link
         End Get
         Set
               m_Link = Value
         End Set
   End Property
   Private m_Link As String
End Class
```

C#

```
public class Video
    {
        public string Title { get; set; }
        public string Thumbnail { get; set; }
        public string Link { get; set; }
    }
```

✓ **What You've Accomplished**

You have successfully added data to **C1TileTileListBox**. In the next step, Step 3 of 3: Running the TileListBox Application, you'll examine the **TileListBox for UWP** features.

Step 3 of 3: Running the TileListBox Application

Now that you've created a UWP style application and customized the application's appearance and behavior, the only thing left to do is run your application. To run your application and observe **TileListBox for UWP**'s run-time behavior, complete the following steps:

1. From the **Debug** menu, select **Start Debugging** to view how your application will appear at run time.

   The application will appear, displaying an image.

2. Use the scroll bar on the right of the control, to scroll through the video images.

If your device have touch capabilities, try pinching to zoom an image.

✓ **What You've Accomplished**

Congratulations! You've completed the **TileListBox for UWP** quick start and created an application using the C1TileListBox control and viewed some of the run-time capabilities of your application.

Top Tips

These tips will help you maximize your performance while using any of the **ListBox** controls.

- **Use PreviewTemplate**

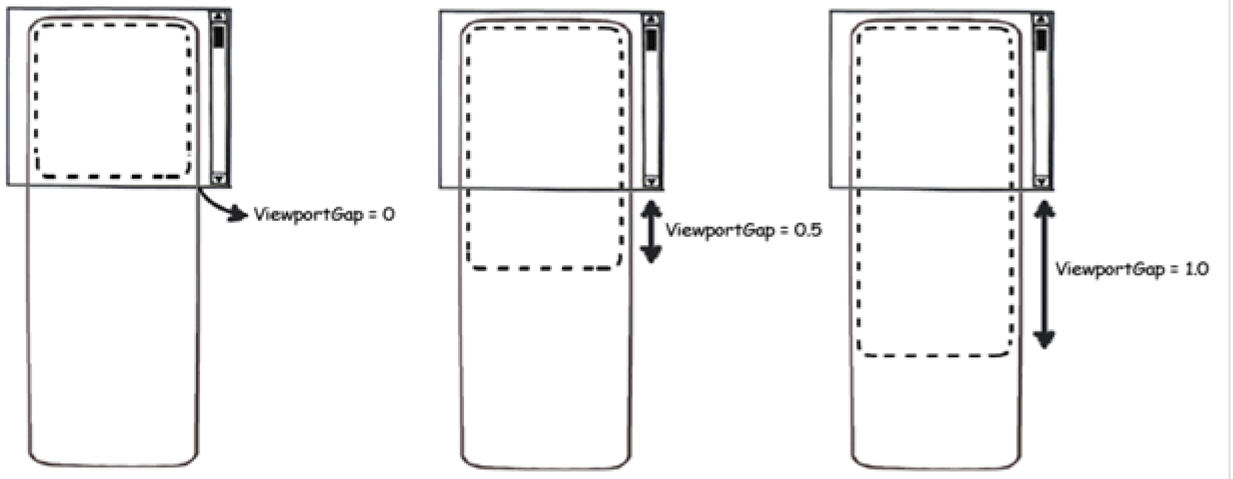
  In order to avoid making the layout heavier, the **PreviewTemplate** can be used to render a lighter template during preview states, such as while zooming and scrolling fast. For example you could display a thumbnail image in the **PreviewTemplate** and display the larger image in the full **ItemTemplate**.

C#

```
<Xaml:C1ListBox x:Name="listBox"
                  ItemsSource="{Binding}"
                  RefreshWhileScrolling="False">
    <Xaml:C1ListBox.PreviewItemTemplate>
        <DataTemplate>
            <Grid Background="Gray">
                <Image Source="{Binding Thumbnail}" Stretch="UniformToFill"/>
            </Grid>
        </DataTemplate>
    </Xaml:C1ListBox.PreviewItemTemplate>
    <Xaml:C1ListBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <Image Source="{Binding Content}" Stretch="UniformToFill"/>
            </Grid>
        </DataTemplate>
    </Xaml:C1ListBox.ItemTemplate>
</Xaml:C1ListBox>
```

- **Adjust the ViewportGap and ViewportPreviewGap Properties**

  These coefficient values determine what size of items outside the viewport to render in advance. The larger the value the more quickly off-screen items will appear to render, but the slower the control will take on each layout pass. For example, if set to 0.5 the view port will be enlarged to take up a half screen more at both sides of the original view port.



- **Set RefreshWhileScrolling to False**

  Determines whether the view port is refreshed while scrolling. If set to false items will appear blank or say "Loading" while the user scrolls real fast until they stop and allow items to render.

Working with ListBox for UWP

ListBox for UWP includes the C1ListBox and C1TileListBox controls. **C1ListBox** is similar to the standard **ListBox** control but it includes additional functionality, such as zooming. **C1TileListBox** allows you to create tiled lists of items. Display lists with tile layouts or with optical zoom using the **C1ListBox** and **C1TileListBox** controls.

Basic Properties

ListBox for UWP includes several properties that allow you to set the functionality of the control. Some of the more important properties are listed below.

The following properties let you customize the C1ListBox control:

Property	Description
C1ListViewer.ActualMaxZoom	Gets the actual maximum zoom.
C1ListViewer.ActualMinZoom	Gets the actual minimum zoom.
C1ListViewer.ActualZoom	Gets the actual zoom.
C1ListViewer.IsScrolling	Gets a value indicating whether the list is scrolling.
C1ListViewer.IsZooming	Gets a value indicating whether this list is zooming.
C1ListViewer.ItemHeight	Gets or sets the height of each item.
Items	Gets the collection used to generate the content of the control. (Inherited from ItemsControl.)
ItemsPanel	Gets or sets the template that defines the panel that controls the layout of items. (Inherited from ItemsControl.)
ItemsSource	Gets or sets a collection used to generate the content of the ItemsControl. (Inherited from ItemsControl.)
ItemTemplate	Template applied to all the items of the list. (Inherited from C1ItemsControl.)
ItemTemplateSelector	Template selector used to specify different templates applied to items of the same type. (Inherited from C1ItemsControl.)
C1ListViewer.ItemWidth	Gets or sets the width of each item.
C1ListViewer.MaxZoom	Gets or sets the maximum zoom available.
C1ListViewer.MinZoom	Gets or sets the minimum zoom available.
C1ListViewer.Orientation	Gets or sets the orientation in which the list is displayed.
C1ListViewer.Panel	Gets the panel associated with this items control.
C1ListViewer.PreviewItemTemplate	Gets or sets the template used for previewing an item.
C1ListViewer.RefreshWhileScrolling	Gets or sets a value indicating whether the viewport must be refreshed while the scroll is running.
C1ListViewer.ScrollViewer	Gets the scroll viewer template part belonging to this items control.
C1ListViewer.ViewportGap	Gets or sets a coeficient which will determine in each layout pass the size of the viewport. If zero is specified the size of the viewport will be equal to the scrollviewer viewport. If 0.5 is specified the size of the viewport will be enlarged to take up half screen more at both sides of the original viewport.
C1ListViewer.ViewportPreviewGap	Gets or sets a coeficient which will determine in each layout pass the size of the viewport to render items in

	preview mode.
C1ListViewer.Zoom	Gets or set the zoom applied to this list.
C1ListViewer.ZoomMode	Gets or sets whether the zoom is enabled or disabled.

The following properties let you customize the C1ListBoxItem:

Description	
C1ListViewerItem.PreviewContent	Gets or sets the content of the preview state.
C1ListViewerItem.PreviewContentTemplate	Gets or sets the DataTemplate used when in preview state.
C1ListViewerItem.State	Gets or sets the state of the item, which can be Preview or Full.

Optical Zoom

The **ListBox for UWP** controls support optical zoom functionality so users can manipulate the size of the items intuitively through pinch gestures. The zooming transformation is smooth and fluid so the performance of your application is not sacrificed.

You can customize the zoom using the ZoomMode and Zoom properties. The **ZoomMode** property gets or sets whether the zoom is enabled or disabled. The **Zoom** property gets or sets the **Zoom** value applied to the control. The ZoomChanged event is triggered when the zoom value of the control is changed.

UI Virtualization

The **ListBox** controls support UI virtualization so they are blazing-fast while able to display thousands of items with virtually no loss of performance. You can determine how many items are rendered in each layout pass by setting the ViewportGap and ViewportPreviewGap properties. These properties can be adjusted depending on the scenario.

The **ViewportGap** property gets or sets a coefficient which will determine in each layout pass the size of the viewport. If zero is specified, the size of the viewport will be equal to the scrollviewer viewport. If 0.5 is specified, the size of the viewport will be enlarged to take up half screen more at both sides of the original viewport.

The **ViewportPreviewGap** property gets or sets a coefficient which will determine in each layout pass the size of the viewport to render items in preview mode.

Orientation

The **ListBox** controls support both horizontal and vertical orientation, allowing for more layout scenarios. To set the orientation of the control, set the Orientation property to **Horizontal** or **Vertical**.

Preview State

In order to have the highest performance imaginable, the **ListBox** controls can render items outside the viewport in a preview state. Like the standard **ItemTemplate**, the **Preview** template defines the appearance of items when they are in a preview state, such as zoomed out or during fast scroll. The controls will then switch to the full item template when the items have stopped scrolling or zooming.

Input for UWP

Provide smarter input for phone numbers, zip codes, percentages and more. With **Input for UWP** you get two controls for masked and numeric input. Quickly gather valid input while displaying formatted text automatically.

Input for UWP includes the following controls:

- **NumericBox for UWP**

  NumericBox for UWP provides a text box for displaying and editing formatted numeric values such as currencies and percentages. The control comes complete with smart input and increment buttons.

- **MaskedTextBox for UWP**

  Provide input mask validation with **MaskedTextBox for UWP**. The C1MaskedTextBox control provides a text box with a mask that prevents users from entering invalid characters.

The following topics will get you started with **Input for UWP**.

Input for UWP Key Features

Input for UWP allows you to create customized, rich applications. Make the most of **Input for UWP** by taking advantage of the following key features:

- **Standard Format Strings**

  Provide instant formatting on user input using the Mask or **Format** properties. The C1MaskedTextBox and C1NumericBox controls support the standard formatting strings defined by Microsoft and uses the same syntax as the classic Windows Forms controls. The Format property enables you to use the familiar .NET format strings to display numbers in several formats with support for decimal places. Supported formats include fixed-point (F), number (N), general (G), currency (C), exponential (E), hexadecimal (X), and percent (P).

- **Include Prompts and Literals**

  Choose whether or not to show prompt characters and literals in the C1MaskedTextBox control by simply setting one property. The prompt character indicates to the user that text can be entered (such as _ or *). Literals are non-mask characters that will appear as themselves within the mask (such as / or -).

- **Numeric Range**

  With the C1NumericBox control you can restrict input to a specific numeric range by setting the Minimum and Maximum properties.

- **Watermark Support**

  Using the Watermark property you can provide contextual clues of what value users should enter. The watermark is displayed in the control while no text has been entered.

Input for UWP Quick Starts

The following quick start guides will get you up and running with **Input for UWP**.

NumericBox for UWP Quick Start

In this quick start you'll create an application that includes five C1NumericBox controls. The controls will function as a lock and when the correct code number has been entered in each, the controls will become locked and inactive and a button will appear directing users to a Web site.

Step 1 of 4: Creating an Application with NumericBox Control

In this step you'll create a UWP-style application using **NumericBox for UWP**. When you add a C1NumericBox control to your application, you'll have a complete, functional numeric editor. You can further customize the control to your application.

To set up your project, complete the following steps:

1. Select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Open MainPage.xaml if it isn't already open, place the cursor between the <Grid> and </Grid> tags, and click once.
4. Navigate to the Toolbox and double-click the **StackPanel** icon to add the panel to **MainPage.xaml**.
5. Add x:Name="sp1" Width="Auto" Height="Auto" Orientation="Vertical" HorizontalAlignment="Center" VerticalAlignment="Center" to the <StackPanel> tag so that it appears similar to the following:

   Markup

   ```
   <StackPanel x:Name="sp1" Width="Auto" Height="Auto" Orientation="Vertical"
   HorizontalAlignment="Center" VerticalAlignment="Center"></StackPanel>
   ```

   Elements in the panel will now appear centered and vertically positioned.
6. In the XAML window of the project, place the cursor between the <StackPanel> and </StackPanel> tags.
7. Navigate to the Visual Studio Toolbox and double-click the standard **TextBlock** control to it to your project.
8. Name the **TextBlock** and add content to it by adding x:Name="tb1" Text="Enter Combination" Margin="5" FontSize="24" to the <TextBlock> tag so that it appears similar to the following:

   Markup

   ```
   <TextBlock x:Name="tb1" Text="Enter Combination" Margin="5" FontSize="24"/>
   ```
9. Navigate to the Toolbox and double-click the **StackPanel** icon to add the panel to the existing **StackPanel** just below the **TextBlock**.
10. Add x:Name="sp2" Width="Auto" Height="Auto" Orientation="Vertical" HorizontalAlignment="Center" VerticalAlignment="Center" to the <StackPanel> tag so that it appears similar to the following:

    Markup

    ```
    <StackPanel x:Name="sp2" Width="Auto" Height="Auto" Orientation="Vertical"
    ```

```
HorizontalAlignment="Center" VerticalAlignment="Center"></StackPanel>
```

Elements in the panel will now appear centered and horizontally positioned.

11. Place the cursor between the first </StackPanel> tag and the second </StackPanel> tag and add the following markup to create a second label:

Markup

```
<TextBlock x:Name="tb2" Text="Invalid Combination" Foreground="Red" Margin="5"
FontSize="18"/>
```

12. Place the cursor between the <TextBlock> tag and the second </StackPanel> tag and add the following markup to create a hidden button:

Markup

```
<Button x:Name="btn1" Content="Enter" Height="60" Visibility="Collapsed"
Click="btn1_Click" Margin="5"></Button>
```

You will add the **btn1_Click** event handler later in code.

You've successfully created a UWP-style application, set up the application's user interface, and added controls to the application. In the next step you'll add **C1NumericBox** controls and complete setting up the application.

Step 2 of 4: Adding C1NumericBox Controls

In the previous step you created a new UWP-style project and added five controls to the application. In this step you'll continue by adding C1NumericBox controls to customize the application.

Complete the following steps:

1. In the XAML window of the project, place the cursor between the <StackPanel x:Name="sp2"> and </StackPanel> tags.
2. Navigate to the Toolbox and double-click the **C1NumericBox** icon to add the control to the **StackPanel**. The XAML markup will now look similar to the following:

Markup

```
<Xaml:C1NumericBox></Xaml:C1NumericBox>
```

Note that the C1.Xaml namespace and <Xaml:C1NumericBox></Xaml:C1NumericBox> tags have been added to the project.

3. Give your control a name by adding x:Name="c1nb1" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1NumericBox x:Name="c1nb1">
```

By giving it a unique identifier, you'll be able to access the control in code.

4. Add a margins by adding Margin="5" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1NumericBox x:Name="c1nb1" Margin="2">
```

Controls will now appear spaced on the page.

5. Set your control's limits by adding Minimum="0" Maximum="9" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1NumericBox x:Name="c1nb1" Margin="2" Minimum="0" Maximum="9">
```

The C1NumericBox.Minimum and C1NumericBox.Maximum properties will set the minimum and maximum values that are allowed in the control. Users will not be able to enter values outside of that range providing built-in data validation.

6. Add ValueChanged="c1nb1_ValueChanged" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1NumericBox x:Name="c1nb1" Margin="2" Minimum="0" Maximum="9"
ValueChanged="c1nb1_ValueChanged">
```

You will add code for the **c1nb1_ValueChanged** event handler in a later step.

7. Add the following XAML just below the existing <Xaml:C1NumericBox x:Name="c1nb1"> </Xaml:C1NumericBox> tags:

Markup

```
<Xaml:C1NumericBox x:Name="c1nb2" Minimum="0" Maximum="9" Margin="5"
ValueChanged="c1nb2_ValueChanged"></Xaml:C1NumericBox>
<Xaml:C1NumericBox x:Name="c1nb3" Minimum="0" Maximum="9" Margin="5"
ValueChanged="c1nb3_ValueChanged"></Xaml:C1NumericBox>
<Xaml:C1NumericBox x:Name="c1nb4" Minimum="0" Maximum="9" Margin="5"
ValueChanged="c1nb4_ValueChanged"></Xaml:C1NumericBox>
<Xaml:C1NumericBox x:Name="c1nb5" Minimum="0" Maximum="9" Margin="5"
ValueChanged="c1nb5_ValueChanged"></Xaml:C1NumericBox>
```

This will add four additional C1NumericBox controls so that you have a total of five controls on the page.

You've successfully added **C1NumericBox** controls to the application and customized those controls. In the next step you'll add code to the application.

Step 3 of 4: Adding Code to the Application

In the previous steps you set up the application's user interface and added **C1NumberBox**, **TextBlock**, and **Button** controls to your application. In this step you'll add code to your application to finalize it.

Complete the following steps:

1. Select **View | Code** to switch to Code view.

2. Add the following imports statements to the top of the page:

Visual Basic

```
Imports Windows.UI.Xaml.Media
Imports Windows.UI.Xaml.Navigation
Imports Windows.UI
Imports C1.Xaml
```

C#

```
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;
using Windows.UI;
using C1.Xaml;
```

3. Initialize the following global variables just inside the **MainPage** class:

Visual Basic

```
Dim nb1 As Integer = 5
Dim nb2 As Integer = 2
Dim nb3 As Integer = 3
Dim nb4 As Integer = 7
Dim nb5 As Integer = 9
```

C#

```
int nb1 = 5;
int nb2 = 2;
int nb3 = 3;
int nb4 = 7;
int nb5 = 9;
```

These numbers will be used as the correct 'code' in the application. When the user enters the correct combination of numbers at run time the button will appear.

4. Add code to the **Button1_Click** event handler so that it appears like the following:

Visual Basic

```
Private Sub btn1_Click(ByVal sender As System.Object, ByVal e As
System.Windows.RoutedEventArgs) Handles btn1.Click
    DefaultLaunch()
End Sub
```

C#

```
private void btn1_Click(object sender, RoutedEventArgs e)
{
    DefaultLaunch();
}
```

5. Add the following code just below the **Button1_Click** event handler:

Visual Basic

```
async Sub DefaultLaunch()
   ' The URI to launch
   Dim uri As New Uri("www.componentone.com")
   ' Launch the URI
   Dim success = await Windows.System.Launcher.LaunchUriAsync(uri)

   If success Then
      ' URI launched
   Else
      ' URI launch failed
   End If
End Sub
```

C#

```
async void DefaultLaunch()
{
    // The URI to launch
    string uriToLaunch = @"http://www.componentone.com/";
    var uri = new Uri(uriToLaunch);
    // Launch the URI
    var success = await Windows.System.Launcher.LaunchUriAsync(uri);

    if (success)
    {
        // URI launched
    }
    else
    {
        // URI launch failed
    }
}
```

When the button is pressed at run time it will open the ComponentOne Web site.

6. Next add the following custom **NBValidation** event to your code:

Visual Basic

```
Private Sub NBValidation()
    If Me.c1nb1.Value = nb1 And Me.c1nb2.Value = nb2 And Me.c1nb3.Value = nb3
And Me.c1nb4.Value = nb4 And Me.c1nb5.Value = nb5 Then
        Me.tb2.Foreground = New SolidColorBrush(Colors.Green)
        Me.tb2.Text = "Combination Valid"
        Me.c1nb1.IsReadOnly = True
        Me.c1nb2.IsReadOnly = True
        Me.c1nb3.IsReadOnly = True
        Me.c1nb4.IsReadOnly = True
        Me.c1nb5.IsReadOnly = True
        Me.btn1.Visibility = Visibility.Visible
    End If
End Sub
```

C#

```
private void NBValidation()
{
    if (this.c1nb1.Value == nb1 & this.c1nb2.Value == nb2 & this.c1nb3.Value ==
nb3 & this.c1nb4.Value == nb4 & this.c1nb5.Value == nb5)
    {
        this.tb2.Foreground = new SolidColorBrush(Colors.Green);
        this.tb2.Text = "Combination Valid";
        this.c1nb1.IsReadOnly = true;
        this.c1nb2.IsReadOnly = true;
        this.c1nb3.IsReadOnly = true;
        this.c1nb4.IsReadOnly = true;
        this.c1nb5.IsReadOnly = true;
        this.btn1.Visibility = Visibility.Visible;
    }
}
```

When the user enters the correct numbers (as indicated in step 3 above) the C1NumericBox controls will be set to read only and will no longer be editable, the text of the label below the controls will change to indicate the correct code has been entered, and a button will appear allowing users to enter the ComponentOne Web site.

7. Add **C1NumericBox_ValueChanged** event handlers to initialize **NBValidation**. The code will look like the following:

Visual Basic

```
Private Sub c1nb1_ValueChanged(ByVal sender As System.Object, ByVal e As
C1.Xaml.PropertyChangedEventArgs(Of System.Double)) Handles c1nb1.ValueChanged
    NBValidation()
End Sub
Private Sub c1nb2_ValueChanged(ByVal sender As System.Object, ByVal e As
C1.Xaml.PropertyChangedEventArgs(Of System.Double)) Handles c1nb2.ValueChanged
    NBValidation()
End Sub
Private Sub c1nb3_ValueChanged(ByVal sender As System.Object, ByVal e As
C1.Xaml.PropertyChangedEventArgs(Of System.Double)) Handles c1nb3.ValueChanged
    NBValidation()
End Sub
Private Sub c1nb4_ValueChanged(ByVal sender As System.Object, ByVal e As
C1.Xaml.PropertyChangedEventArgs(Of System.Double)) Handles c1nb4.ValueChanged
    NBValidation()
End Sub
Private Sub c1nb5_ValueChanged(ByVal sender As System.Object, ByVal e As
C1.Xaml.PropertyChangedEventArgs(Of System.Double)) Handles c1nb5.ValueChanged
    NBValidation()
End Sub
```

C#

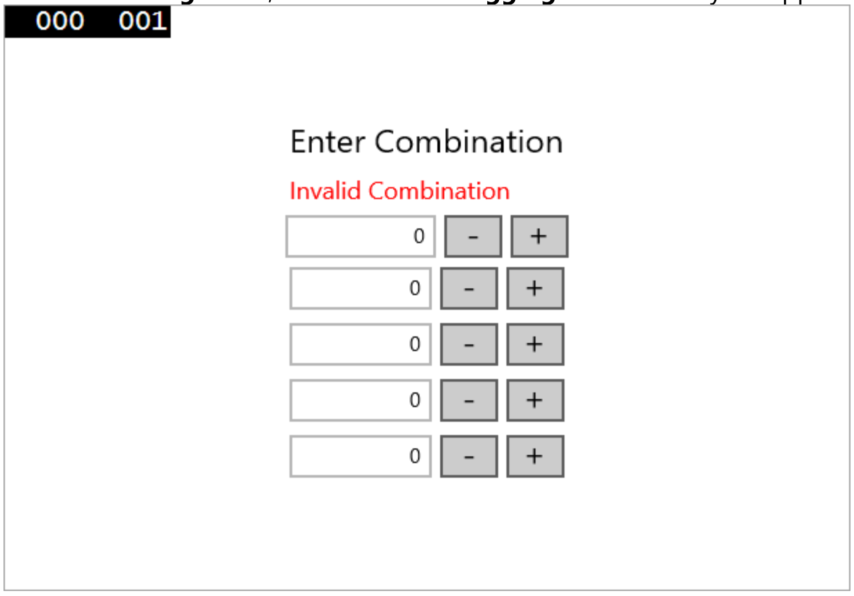
```
private void c1nb1_ValueChanged(object sender, PropertyChangedEventArgs<double>
e)
{
    NBValidation();
}
private void c1nb2_ValueChanged(object sender, PropertyChangedEventArgs<double>
e)
{
    NBValidation();
}
private void c1nb3_ValueChanged(object sender, PropertyChangedEventArgs<double>
e)
{
    NBValidation();
}
private void c1nb4_ValueChanged(object sender, PropertyChangedEventArgs<double>
e)
{
    NBValidation();
}
private void c1nb5_ValueChanged(object sender, PropertyChangedEventArgs<double>
e)
{
    NBValidation();
}
```

In this step you completed adding code to your application. In the next step you'll run the application and observe run-time interactions.

Step 4 of 4: Running the Application

Now that you've created a UWP-style application and customized the application's appearance and behavior, the only thing left to do is run your application. To run your application and observe **NumericBox for UWP**'s run-time behavior, complete the following steps:

1. From the **Debug** menu, select **Start Debugging** to view how your application will appear at run time.



2. Click the **+** button in the first C1NumericBox control until **5** is displayed. Note that the number increased by 1

each time you click the button, this is because the C1NumericBox.Increment property is set to **1** by default.

3. Click inside the second **C1NumericBox**, highlight the "0" value, and type "2" to replace it.
4. Try clicking the **-** button in the third **C1NumericBox** control and notice that the number does not change. This is because the C1NumericBox.Minimum property was set to **0** and so the control will not accept values less than zero. Click the **+** button until **3** is displayed.
5. In the fourth **C1NumericBox** control, place the cursor in front of the **0** and click. Enter "5" so that "50" is displayed.
6. Click inside the last **C1NumericBox** control. Notice that the **50** inside the fourth **C1NumericBox** was reset to **9**. That's because the C1NumericBox.Maximum property was set to **9** so the control will not accept values greater than nine.
7. Enter **9** in the last **C1NumericBox** control.
8. Click the **-** button of the fourth **C1NumericBox** control twice so **7** is displayed. Note that the text of the second Label changed and the button is now visible.
9. Try typing inside a **C1NumericBox** control or clicking its **+** or **-** buttons, notice that you cannot. That is because the C1NumericBox.IsReadOnly property was set to **True** when the correct number sequence was entered and the controls are now locked from editing.
10. Click the now-visible **Enter** button to navigate to the ComponentOne Web site.

Congratulations! You've completed the **NumericBox for UWP** quick start and created a **NumericBox for UWP** application, customized the appearance and behavior of the controls, and viewed some of the run-time capabilities of your application.

MaskedTextBox for UWP Quick Start

The following quick start guide is intended to get you up and running with **MaskedTextBox for UWP**. In this quick start you'll create a new application in Visual Studio, add **MaskedTextBox** controls, and customize the appearance and behavior of the controls in the application.

You will create a simple form using several C1MaskedTextBox controls that will demonstrate the difference between the **Text** and **Value** properties. The controls will include various masks and different appearance and behavior settings so that you can explore the possibilities of using **MaskedTextBox for UWP**.

Step 1 of 4: Setting up the Application

In this step you'll begin in Visual Studio to create a UWP-style application using **MaskedTextBox for UWP**. When you add a C1MaskedTextBox control to your application, you'll have a complete, functional input editor. You can further customize the control to your application.

To set up your project and add C1MaskedTextBox controls to your application, complete the following steps:

1. In Visual Studio select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Open MainPage.xaml if it isn't already open, place the cursor between the <Grid> and </Grid> tags, and click once.

4. Navigate to the Toolbox and double-click the **StackPanel** icon to add it to the page.

5. Add x:Name="sp1" Width="Auto" Height="Auto" Orientation="Vertical" HorizontalAlignment="Center" VerticalAlignment="Center" to the <StackPanel> tag so that it appears similar to the following:

   Markup

   ```
   <StackPanel x:Name="sp1" Width="Auto" Height="Auto" Orientation="Vertical"
   HorizontalAlignment="Center" VerticalAlignment="Center"></StackPanel>
   ```

   Elements in the panel will now appear centered and vertically positioned.

You've successfully created a UWP-style application. In the next step you'll add and customize **TextBlock** and **C1MakedTextBox** controls and complete setting up the application.

Step 2 of 4: Customizing the Application

In the previous step you created a new UWP-style project and added a **StackPanel** to the application. In this step you'll continue by adding and customizing **TextBlock** and C1MaskedTextBox controls.

Complete the following steps:

1. In the XAML window of the project, place the cursor between the <StackPanel x:Name="sp1"> and </StackPanel> tags.

2. Add the following markup within the StackPanel to add two standard TextBlock controls:

   Markup

   ```
   <TextBlock Margin="2,2,2,10" Name="tb1" Text="Employee Information" />
   <TextBlock FontSize="16" Margin="2,2,2,0" Text="Employee ID" />
   ```

3. Place the cursor just below the makrup you just added , navigate to the Toolbox and double-click the C1MaskedTextBox icon to add the control to the **StackPanel**. This will add the reference and XAML namespace automatically. The XAML markup resembles the following:

   Markup

   ```
   <Xaml:C1MaskedTextBox x:Name="c1MaskedTextBox" Text="C1MaskedTextBox"/>
   ```

4. Inside the Grid, initialize the **C1MaskedTextBox** control and give it a name by adding Name="c1mtb1" VerticalAlignment="Top" Margin="2" Mask="000-00-0000" TextChanged="c1mtb1_TextChanged" to the <Xaml:C1MaskedTextBox/> tag so that it appears similar to the following:

   Markup

   ```
   <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
       <Xaml:C1MaskedTextBox Name="c1mtb1" VerticalAlignment="Top" Margin="2"
   Mask="000-00-0000" TextChanged="c1mtb1_TextChanged"/>
   </Grid>
   ```

   Notice that this markup adds a name, sets the margin and alignment, and sets a mask for the content of the box. Note that you'll add code for the event handler you added in a later step.

5. Place the cursor just after the <Xaml:C1MaskedTextBox> tag and add the following XAML to add additional **C1MaskedTextBox** and **TextBlock** controls to the **StackPanel**:

Markup

```
<TextBlock x:Name="tb2" FontSize="16" Margin="2" />
<TextBlock FontSize="16" Margin="2,2,2,0" Text="Name"/>
<Xaml:C1MaskedTextBox Name="c1mtb2" VerticalAlignment="Top" Margin="2"
TextChanged="c1mtb2_TextChanged"></Xaml:C1MaskedTextBox>
<TextBlock x:Name="tb3" FontSize="16" Margin="2"/>
<TextBlock FontSize="16" Margin="2"  Text="Hire Date"/>
<Xaml:C1MaskedTextBox Name="c1mtb3" VerticalAlignment="Top" Margin="2"
Mask="00/00/0000" TextChanged="c1mtb3_TextChanged"></Xaml:C1MaskedTextBox>
<TextBlock x:Name="tb4" FontSize="16" Margin="2"/>
<TextBlock FontSize="16" Margin="2,2,2,0" Text="Phone Number"/>
<Xaml:C1MaskedTextBox Name="c1mtb4" VerticalAlignment="Top" Margin="2" Mask="
(999) 000-0000" TextChanged="c1mtb4_TextChanged"></Xaml:C1MaskedTextBox>
<TextBlock x:Name="tb5" FontSize="16" Margin="2"/>
```

You've successfully set up your application's user interface. In the next step you'll add code to your application.

Step 3 of 4: Adding Code to the Application

In the previous steps you set up the application's user interface and added controls to your application. In this step you'll add code to your application to add additional functionality.

Complete the following steps:

1. Select **View | Code** to switch to Code view.
2. In Code view, add the following import statement to the top of the page:

Visual Basic

```
Imports C1.Xaml
```

C#

```
using C1.Xaml;
```

3. Add the following **C1MaskedTextBox_TextChanged** event handlers to the project:

Visual Basic

```
Private Sub c1mtb1_TextChanged(ByVal sender As System.Object, ByVal e As
System.Windows.Controls.TextChangedEventArgs) Handles c1mtb1.TextChanged
    Me.tb2.Text = "Mask: " & Me.c1mtb1.Mask & "  Value: " & Me.c1mtb1.Value & "
Text: " & Me.c1mtb1.Text
End Sub
Private Sub c1mtb2_TextChanged(ByVal sender As System.Object, ByVal e As
System.Windows.Controls.TextChangedEventArgs) Handles c1mtb2.TextChanged
    Me.tb3.Text = "Mask: " & Me.c1mtb2.Mask & "  Value: " & Me.c1mtb2.Value & "
Text: " & Me.c1mtb2.Text
End Sub
Private Sub c1mtb3_TextChanged(ByVal sender As System.Object, ByVal e As
System.Windows.Controls.TextChangedEventArgs) Handles c1mtb3.TextChanged
```

```
    Me.tb4.Text = "Mask: " & Me.c1mtb3.Mask & "  Value: " & Me.c1mtb3.Value & "
Text: " & Me.c1mtb3.Text
End Sub
Private Sub c1mtb4_TextChanged(ByVal sender As System.Object, ByVal e As
System.Windows.Controls.TextChangedEventArgs) Handles c1mtb4.TextChanged
    Me.tb5.Text = "Mask: " & Me.c1mtb4.Mask & "  Value: " & Me.c1mtb4.Value & "
Text: " & Me.c1mtb4.Text
End Sub
```

C#

```
private void c1mtb1_TextChanged(object sender, TextChangedEventArgs e)
{
    this.tb2.Text = "Mask: " + this.c1mtb1.Mask + " Value: " + this.c1mtb1.Value
+ " Text: " + this.c1mtb1.Text;
}
private void c1mtb2_TextChanged(object sender, TextChangedEventArgs e)
{
    this.tb3.Text = "Mask: " + this.c1mtb2.Mask + " Value: " + this.c1mtb2.Value
+ " Text: " + this.c1mtb2.Text;
}
private void c1mtb3_TextChanged(object sender, TextChangedEventArgs e)
{
    this.tb4.Text = "Mask: " + this.c1mtb3.Mask + " Value: " + this.c1mtb3.Value
+ " Text: " + this.c1mtb3.Text;
}
private void c1mtb4_TextChanged(object sender, TextChangedEventArgs e)
{
    this.tb5.Text = "Mask: " + this.c1mtb4.Mask + " Value: " + this.c1mtb4.Value
+ " Text: " + this.c1mtb4.Text;
}
```

4. Add code to the page's constructor so that it appears like the following:

Visual Basic

```
Public Sub New()
    InitializeComponent()
    Me.c1mtb1_TextChanged(Nothing, Nothing)
    Me.c1mtb2_TextChanged(Nothing, Nothing)
    Me.c1mtb3_TextChanged(Nothing, Nothing)
    Me.c1mtb4_TextChanged(Nothing, Nothing)
End Sub
```

C#

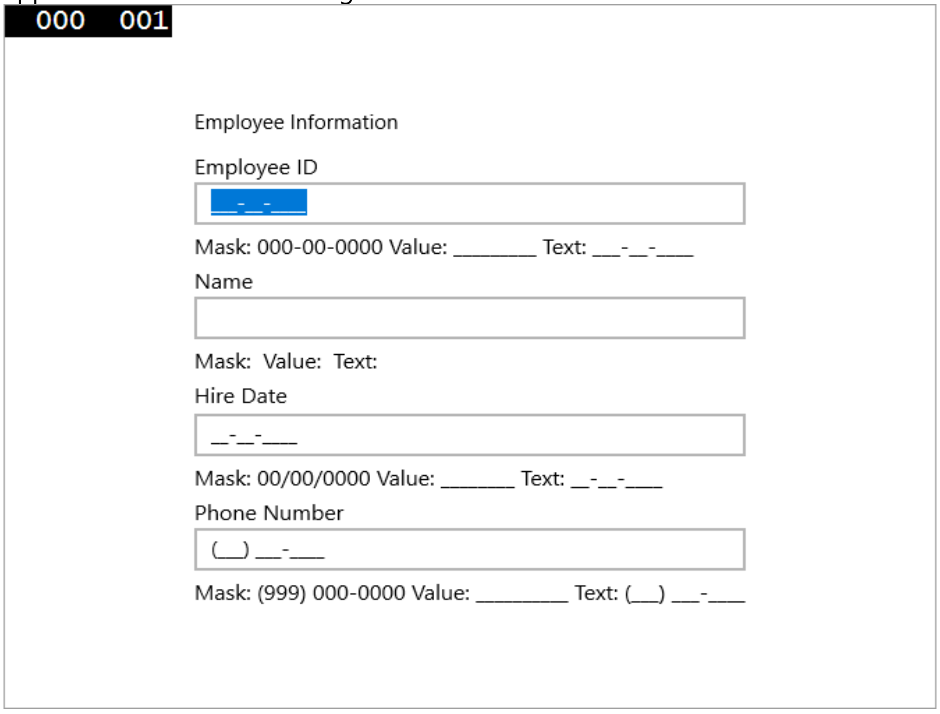
```
public MainPage()
{
    InitializeComponent();
    this.c1mtb1_TextChanged(null, null);
    this.c1mtb2_TextChanged(null, null);
    this.c1mtb3_TextChanged(null, null);
    this.c1mtb4_TextChanged(null, null);
}
```

In this step you completed adding code to your application. In the next step you'll run the application and observe run-time interactions.

Step 4 of 4: Running the Application

Now that you've created a UWP-style application and customized the application's appearance and behavior, the only thing left to do is run your application. To run your application and observe **MaskedTextBox for UWP**'s run-time behavior, complete the following steps:

1. From the **Debug** menu, select **Start Debugging** to view how your application will appear at run time. It will appear similar to the following:



000 001

Employee Information

Employee ID

___-__-____

Mask: 000-00-0000 Value: ________ Text: ___-__-____

Name

Mask: Value: Text:

Hire Date

__-__-____

Mask: 00/00/0000 Value: _______ Text: __-__-____

Phone Number

(___) ___-____

Mask: (999) 000-0000 Value: _________ Text: (___) ___-____

2. Enter a number in the first **C1MaskedTextBox** control.

   The label below the control displays the mask, current value, and current text.

3. Enter a string in the second **C1MaskedTextBox** control.

   Notice that there was no mask added to this control – if you chose, you could type numbers or other characters in the control.

4. Try entering a string in the third **C1MaskedTextBox** control. Notice that you cannot – that is because the C1MaskedTextBox.Mask property was set to only accept numbers. Enter a numeric value instead – notice that this does work.

5. Enter numbers in each of the remaining controls.

   Notice that the C1MaskedTextBox.Value property displayed under each C1MaskedTextBox control does not include literal characters, while the **Text** property does.

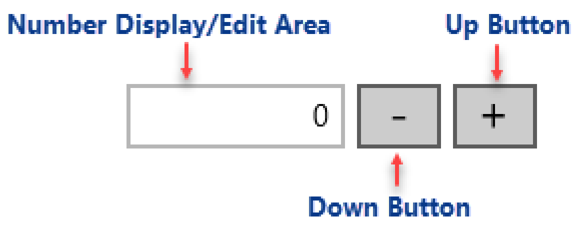
Congratulations! You've completed the **MaskedTextBox for UWP** quick start and created a **MaskedTextBox for UWP** application, customized the appearance and behavior of the controls, and viewed some of the run-time capabilities of your application.

Working with Input for UWP

The following topics will introduce some of the features and functionality found in the C1NumericBox and C1MaskedTextBox controls.

Working with C1NumericBox

NumericBox for UWP includes the C1NumericBox control, a simple control which provides numeric input and editing. When you add the C1NumericBox control to a XAML window, it exists as a completely functional numeric editor. By default, the control's interface looks similar to the following image:



It consists of the following elements:

- **+ and - Buttons**

  The **+**(Up) and **-**(Down) buttons allow users to change the value displayed in the control. Each time a button is pressed the C1NumericBox.Value changes by the amount indicated by the C1NumericBox.Increment property (by default 1). By default the **+** and **-** buttons are visible; to hide the buttons set the C1NumericBox.ShowButtons property to **False**.

- **Number Display/Edit Area**

  The current **C1NumericBox.Value** is displayed in the number display/editing area. Users can type in the box to change the **C1NumericBox.Value** property. By default users can edit this number; to lock the control from editing set C1NumericBox.IsReadOnly to **True**.

Number Formatting

You can change how the number displayed in the C1NumericBox control will appear by setting the C1NumericBox.Format property. **NumericBox for UWP** supports the standard number formatting strings defined by Microsoft. For more information, see MSDN.

The **C1NumericBox.Format** string consists of a letter or a letter and number combination defining the format. By default, the **C1NumericBox.Format** property is set to "F0". The letter indicates the format type, here "F" for fixed-point, and the number indicates the number of decimal places, here none.

The following formats are available:

Format Specifier	Name	Description
C or c	Currency	The number is converted to a string that represents a currency amount. The conversion is controlled by the currency format information of the current

		NumberFormatInfo object. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default currency precision given by the current NumberFormatInfo object is used.
D or d	Decimal	This format is supported only for integral types. The number is converted to a string of decimal digits (0-9), prefixed by a minus sign if the number is negative. The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier. The following example formats an Int32 value with the Decimal format specifier.
E or e	Scientific (exponential)	The number is converted to a string of the form "-d.ddd…E+ddd" or "-d.ddd…e+ddd", where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. One digit always precedes the decimal point. The precision specifier indicates the desired number of digits after the decimal point. If the precision specifier is omitted, a default of six digits after the decimal point is used. The case of the format specifier indicates whether to prefix the exponent with an 'E' or an 'e'. The exponent always consists of a plus or minus sign and a minimum of three digits. The exponent is padded with zeros to meet this minimum, if required.
F or f	Fixed-point	The number is converted to a string of the form "-ddd.ddd…" where each 'd' indicates a digit (0-9). The string starts with a minus sign if the number is negative. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision is given by the NumberDecimalDigits property of the current NumberFormatInfo object.
G or g	General	The number is converted to the most compact of either fixed-point or scientific notation, depending on the type of the number and whether a precision specifier is present. If the precision specifier is omitted or zero, the type of the number determines the default precision, as indicated by the following list. · Byte or SByte: 3 · Int16 or UInt16: 5 · Int32 or UInt32: 10

		· Int64: 19 · UInt64: 20 · Single: 7 · Double: 15 · Decimal: 29 Fixed-point notation is used if the exponent that would result from expressing the number in scientific notation is greater than -5 and less than the precision specifier; otherwise, scientific notation is used. The result contains a decimal point if required and trailing zeroes are omitted. If the precision specifier is present and the number of significant digits in the result exceeds the specified precision, then the excess trailing digits are removed by rounding. The exception to the preceding rule is if the number is a Decimal and the precision specifier is omitted. In that case, fixed-point notation is always used and trailing zeroes are preserved. If scientific notation is used, the exponent in the result is prefixed with 'E' if the format specifier is 'G', or 'e' if the format specifier is 'g'. The exponent contains a minimum of two digits. This differs from the format for scientific notation produced by the 'E' or 'e' format specifier, which includes a minimum of three digits in the exponent.
N or n	Number	The number is converted to a string of the form "-d,ddd,ddd.ddd…", where '-' indicates a negative number symbol if required, 'd' indicates a digit (0-9), ',' indicates a thousand separator between number groups, and '.' indicates a decimal point symbol. The actual negative number pattern, number group size, thousand separator, and decimal separator are specified by the NumberNegativePattern, NumberGroupSizes, NumberGroupSeparator, and NumberDecimalSeparator properties, respectively, of the current NumberFormatInfo object. The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision is given by the NumberDecimalDigits property of the current NumberFormatInfo object.
P or p	Percent	The number is converted to a string that represents a percent as defined by the NumberFormatInfo.PercentNegativePattern property if the number is negative, or the NumberFormatInfo.PercentPositivePattern property if the number is positive. The converted number is multiplied by 100 in order to be presented as a percentage.

		The precision specifier indicates the desired number of decimal places. If the precision specifier is omitted, the default numeric precision given by the current NumberFormatInfo object is used.
R or r	Round-trip	This format is supported only for the Single and Double types. The round-trip specifier guarantees that a numeric value converted to a string will be parsed back into the same numeric value. When a numeric value is formatted using this specifier, it is first tested using the general format, with 15 spaces of precision for a **Double** and 7 spaces of precision for a **Single**. If the value is successfully parsed back to the same numeric value, it is formatted using the general format specifier. However, if the value is not successfully parsed back to the same numeric value, then the value is formatted using 17 digits of precision for a **Double** and 9 digits of precision for a **Single**. Although a precision specifier can be present, it is ignored. Round trips are given precedence over precision when using this specifier.
X or x	Hexadecimal	This format is supported only for integral types. The number is converted to a string of hexadecimal digits. The case of the format specifier indicates whether to use uppercase or lowercase characters for the hexadecimal digits greater than 9. For example, use 'X' to produce "ABCDEF", and 'x' to produce "abcdef". The precision specifier indicates the minimum number of digits desired in the resulting string. If required, the number is padded with zeros to its left to produce the number of digits given by the precision specifier.
Any other single character	(Unknown specifier)	(An unknown specifier throws a FormatException at runtime.)

Input Validation

You can use the C1NumericBox.Minimum and C1NumericBox.Maximum properties to set a numeric range that users are limited to at run time. If the **C1NumericBox.Minimum** and **C1NumericBox.Maximum** properties are set, users will not be able to pick a number larger than the **C1NumericBox.Minimum** or smaller than the **C1NumericBox.Maximum**.

When setting the C1NumericBox.Minimum and C1NumericBox.Maximum properties, the C1NumericBox.Minimum should be smaller than the C1NumericBox.Maximum. Also be sure to set the C1NumericBox.Value property to a number within the C1NumericBox.Minimum and C1NumericBox.Maximum range.

You can also choose a mode for range validation using the C1NumericBox.RangeValidationMode property. This property controls when the entered number is validated. You can set **C1NumericBox.RangeValidationMode** to one of the following options:

Option	Description

Always	This mode does not allow users to enter out of range values.
AlwaysTruncate	This mode does not allow users to enter out of range values. The value will be truncated if the limits are exceeded.
OnLostFocus	This mode truncates the value when the control loses focus.

Working with C1MaskTextBox

MaskedTextBox for UWP includes the C1MaskedTextBox control, a simple control which provides a text box with a mask that automatically validates entered input. When you add the C1MaskedTextBox control to a XAML window, it exists as a completely functional text box which you can further customize with a mask.

The C1MaskedTextBox control appears like a text box and includes a basic text input area which can be customized.

Mask Formatting

You can provide input validation and format how the content displayed in the C1MaskedTextBox control will appear by setting the C1MaskedTextBox.Mask property. **MaskedTextBox for UWP** supports the standard number formatting strings defined by Microsoft and the **C1MaskedTextBox.Mask** property uses the same syntax as the standard **MaskedTextBox** control in WinForms. This makes it easier to re-use masks across applications and platforms.

By default, the **C1MaskedTextBox.Mask** property is not set and no input mask is applied. When a mask is applied, the C1MaskedTextBox.Mask string should consist of one or more of the masking elements. Other elements that may be displayed in the control are literals and prompts which may also be used if allowed by the C1MaskedTextBox.TextMaskFormat property.

The following table lists some example masks:

Mask	Behavior
00/00/0000	A date (day, numeric month, year) in international date format. The "/" character is a logical date separator, and will appear to the user as the date separator appropriate to the application's current culture.
00->L<LL-0000	A date (day, month abbreviation, and year) in United States format in which the three-letter month abbreviation is displayed with an initial uppercase letter followed by two lowercase letters.
(999)-000-0000	United States phone number, area code optional. If users do not want to enter the optional characters, they can either enter spaces or place the mouse pointer directly at the position in the mask represented by the first 0.
$999,999.00	A currency value in the range of 0 to 999999. The currency, thousandth, and decimal characters will be replaced at run time with their culture-specific equivalents.

You can set the C1MaskedTextBox.TextMaskFormat property to one of the following elements to define what is included in the mask:

Option	Description
IncludePrompt	Return text input by the user as well as any instances of the prompt

	character.
IncludeLiterals	Return text input by the user as well as any literal characters defined in the mask.
IncludePromptAndLiterals	Return text input by the user as well as any literal characters defined in the mask and any instances of the prompt character.
ExcludePromptAndLiterals	Return only text input by the user.

The following topics detail mask, literal, and prompt elements that can be used or displayed.

Mask Elements

MaskedTextBox for UWP supports the standard number formatting strings defined by Microsoft.
The C1MaskedTextBox.Mask string should consist of one or more of the masking elements as detailed in the following table:

Element	Description
0	Digit, required. This element will accept any single digit between 0 and 9.
9	Digit or space, optional.
#	Digit or space, optional. If this position is blank in the mask, it will be rendered as a space in the **Text** property. Plus (+) and minus (-) signs are allowed.
L	Letter, required. Restricts input to the ASCII letters a-z and A-Z. This mask element is equivalent to [a-zA-Z] in regular expressions.
?	Letter, optional. Restricts input to the ASCII letters a-z and A-Z. This mask element is equivalent to [a-zA-Z]? in regular expressions.
&	Character, required.
C	Character, optional. Any non-control character.
A	Alphanumeric, optional.
a	Alphanumeric, optional.
.	Decimal placeholder. The actual display character used will be the decimal symbol appropriate to the format provider.
,	Thousands placeholder. The actual display character used will be the thousands placeholder appropriate to the format provider.
:	Time separator. The actual display character used will be the time symbol appropriate to the format provider.
/	Date separator. The actual display character used will be the date symbol appropriate to the format provider.
$	Currency symbol. The actual character displayed will be the currency symbol appropriate to the format provider.
<	Shift down. Converts all characters that follow to lowercase.
>	Shift up. Converts all characters that follow to uppercase.

|	Disable a previous shift up or shift down.
\	Escape. Escapes a mask character, turning it into a literal. "\\" is the escape sequence for a backslash.
All other characters	Literals. All non-mask elements will appear as themselves within C1MaskedTextBox. Literals always occupy a static position in the mask at run time, and cannot be moved or deleted by the user.

The decimal (.), thousandths (,), time (:), date (/), and currency ($) symbols default to displaying those symbols as defined by the application's culture.

Literals

In addition to the mask elements defined in the Mask Formatting topic, other characters can be included in the mask. These characters are *literals*. Literals are non-mask elements that will appear as themselves within **C1MaskedTextBox**. Literals always occupy a static position in the mask at run time, and cannot be moved or deleted by the user.

For example, if the C1MaskedTextBox.Mask property has been set to "(999)-000-0000" to define a phone number, the mask characters include the "9" and "0" elements. The remaining characters, the dashes and parentheses, are literals. These characters will appear as they in the C1MaskedTextBox control.

Note that the C1MaskedTextBox.TextMaskFormat property must be set to **IncludeLiterals** or **IncludePromptAndLiterals** for literals to be used. If you do not want literals to be used, set C1MaskedTextBox.TextMaskFormat to **IncludePrompt** or **ExcludePromptAndLiterals**.

Prompts

You can choose to include prompt characters in the C1MaskedTextBox control. The prompt character defined that text that will appear in the control to prompt the user to enter text. The prompt character indicates to the user that text can be entered, and can be used to detail the type of text allowed. By default the underline "_" character is used.

Note that the C1MaskedTextBox.TextMaskFormat property must be set to IncludePrompt or IncludePromptAndLiterals for prompt characters to be used. If you do not want prompt characters to be used, set **C1MaskedTextBox.TextMaskFormat** to IncludeLiterals or ExcludePromptAndLiterals.

Watermark

Using the Watermark property you can provide contextual clues of what value users should enter in a C1MaskedTextBox control. The watermark is displayed in the control while not text has been entered. To add a watermark, add the text Watermark="Watermark Text" to the <Xaml:C1MaskedTextBox> tag in the XAML markup for any **C1MaskedTextBox** control.

So, for example, enter Watermark="Enter Text" to the <Xaml:C1MaskedTextBox> tag so that appears similar to the following:

Markup

```
<Xaml:C1MaskedTextBox Height="23" Width="120" Name="C1MaskedTextBox1"
Watermark="Enter Text" />
```

If you click within the control and enter text, you will notice that the watermark disappears.

Input for UWP Task-Based Help

The following task-based help topics assume that you are familiar with programming with UWP and know how to use the C1NumericBox and C1MaskedTextBox controls in general. Each topic in this section provides a solution for specific tasks using the **Input for UWP** product.

Each task-based help topic also assumes that you have created a new UWP project.

C1NumericBox Task-Based Help

Each topic in this section provides a solution for specific tasks using the C1NumericBox control.

Setting the Start Value

The C1NumericBox.Value property determines the currently selected number. By default the C1NumericBox control starts with its **C1NumericBox.Value** set to **0** but you can customize this number.

At Design Time

To set the **C1NumericBox.Value** property, complete the following steps:

1. Click the C1NumericBox control once to select it.
2. Navigate to the Properties tab, and enter a number, for example "123", in the text box next to the **C1NumericBox.Value** property.

This will set the **C1NumericBox.Value** property to the number you chose.

In XAML

For example, to set the **C1NumericBox.Value** property addValue="123" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup
```
<Xaml:C1NumericBox x:Name="C1NumericBox1" Value="123"></Xaml:C1NumericBox>
```

In Code

For example, to set the **C1NumericBox.Value** property add the following code to your project:

Visual Basic
```
C1NumericBox1.Value = 123
```

C#
```
c1NumericBox1.Value = 123;
```

Run your project and observe:

Initially **123** (or the number you chose) will appear in the control.

Setting the Increment Value

The C1NumericBox.Increment property determines by how much the C1NumericBox.Value property changes when the **Up** or **Down** button is pressed at run time. By default the C1NumericBox control starts with its **C1NumericBox.Increment** set to **1** but you can customize this number.

At Design Time

To set the **C1NumericBox.Increment** property, complete the following steps:

1. Click the C1NumericBox control once to select it.
2. Navigate to the Properties tab, and enter a number, for example "20", in the text box next to the C1NumericBox.Increment property.

This will set the **C1NumericBox.Increment** property to the number you chose.

In XAML

For example, to set the C1NumericBox.Increment property to **20** addIncrement="20" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1NumericBox x:Name="C1NumericBox1" Increment="20"></Xaml:C1NumericBox>
```

In Code

For example, to set the **C1NumericBox.Increment** property to **20** add the following code to your project:

Visual Basic

```
C1NumericBox1.Increment = 20
```

C#

```
c1NumericBox1.Increment = 20;
```

Run your project and observe:

Press the **Up** and then the **Down** button a few times. Notice that the **C1NumericBox.Value** changes in steps of 20. You can still edit the value directly by selecting the text box and entering a number that falls between that step.

Setting the Minimum and Maximum Values

You can use the C1NumericBox.Minimum and C1NumericBox.Maximum properties to set a numeric range that users are limited to at run time. If the **C1NumericBox.Minimum** and **C1NumericBox.Maximum** properties are set, users will not be able to pick a number larger than the **C1NumericBox.Minimum** or smaller than the **C1NumericBox.Maximum**.

When setting the **C1NumericBox.Minimum** and **C1NumericBox.Maximum** properties, the **C1NumericBox.Minimum** should be smaller than the **C1NumericBox.Maximum**. Also be sure to set the **C1NumericBox.Value** property to a number within the **C1NumericBox.Minimum** and **C1NumericBox.Maximum** range. In the following example, the default value **0** falls within the range chosen.

At Design Time

To set the **C1NumericBox.Minimum** and **C1NumericBox.Maximum**, complete the following steps:

1. Click the **C1NumericBox** control once to select it.

2. Navigate to the Properties tab, and enter a number, for example **500**, next to the **C1NumericBox.Maximum** property.
3. In the Properties tab, enter a number, for example **-500**, next to the **C1NumericBox.Minimum** property.

This will set **C1NumericBox.Minimum** and **C1NumericBox.Maximum** values.

In XAML

To set the **C1NumericBox.Minimum** and **C1NumericBox.Maximum** in XAML addMaximum="500" Minimum="-500" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1NumericBox x:Name="C1NumericBox1" Maximum="500" Minimum="-500">
</Xaml:C1NumericBox>
```

In Code

To set the **C1NumericBox.Minimum** and **C1NumericBox.Maximum** add the following code to your project:

Visual Basic

```
C1NumericBox1.Minimum = -500
C1NumericBox1.Maximum = 500
```

C#

```
c1NumericBox1.Minimum = -500;
c1NumericBox1.Maximum = 500;
```

Run your project and observe:

Users will be limited to the selected range at run time.

Hiding the Up and Down Buttons

By default buttons are visible in the C1NumericBox control to allow users to increment and decrement the value in the box by one step. You can choose to hide the **Up** and **Down** buttons in the **C1NumericBox** control at run time. To hide the **Up** and **Down** buttons you can set the C1NumericBox.ShowButtons property to **False**.

At Design Time

To hide the **Up** and **Down** buttons, complete the following steps:

1. Click the **C1NumericBox** control once to select it.
2. Navigate to the Properties tab, and uncheck the **C1NumericBox.ShowButtons** check box.

This will set the **C1NumericBox.ShowButtons** property to **False**.

In XAML

To hide the **Up** and **Down** buttons in XAML addShowButtons="False" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1NumericBox x:Name="C1NumericBox1" Width="40" Height="25" ShowButtons="False">
</Xaml:C1NumericBox>
```

In Code

To hide the **Up** and **Down** buttons add the following code to your project:

Visual Basic
```
C1NumericBox1.ShowButtons = False
```

C#
```
c1NumericBox1.ShowButtons = false;
```

Run your project and observe:

The **Up** and **Down** buttons will not be visible.

Locking the Control from Editing

By default the C1NumericBox control's C1NumericBox.Value property is editable by users at run time. If you want to lock the control from being edited, you can set the C1NumericBox.IsReadOnly property to **True**.

At Design Time

To lock the **C1NumericBox** control from run-time editing, complete the following steps:

1. Click the **C1NumericBox** control once to select it.
2. Navigate to the Properties tab, and check the **C1NumericBox.IsReadOnly** check box.

This will set the **C1NumericBox.IsReadOnly** property to **False**.

In XAML

To lock the **C1NumericBox** control from run-time editing in XAML add IsReadOnly="True" to the <Xaml:C1NumericBox> tag so that it appears similar to the following:

Markup
```
<Xaml:C1NumericBox x:Name="C1NumericBox1" IsReadOnly="True"></Xaml:C1NumericBox>
```

In Code

To lock the **C1NumericBox** control from run-time editing add the following code to your project:

Visual Basic
```
C1NumericBox1.IsReadOnly = True
```

C#
```
c1NumericBox1.IsReadOnly = true;
```

Run your project and observe:

The control is locked from editing; notice that the **Up** and **Down** buttons are grayed out and inactive.

C1MaskedTextBox Task-Based Help

Each topic in this section provides a solution for specific tasks using the C1MaskedTextBox control.

Setting the Value

The C1MaskedTextBox.Value property determines the currently visible text. By default the C1MaskedTextBox control starts with its **C1MaskedTextBox.Value** not set but you can customize this.

At Design Time

To set the **C1MaskedTextBox.Value** property, complete the following steps:

1. Click the C1MaskedTextBox control once to select it.
2. Navigate to the Properties tab and enter a number, for example "123", in the text box next to the **C1MaskedTextBox.Value** property.

This will set the **C1MaskedTextBox.Value** property to the number you chose.

In XAML

To set the C1MaskedTextBox.Value property add Value="123" to the <Xaml:C1MaskedTextBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1MaskedTextBox Height="23" Width="120" Name="C1MaskedTextBox1" Value="123">
</Xaml:C1MaskedTextBox>
```

In Code

To set the **C1MaskedTextBox.Value** property, add the following code to your project:

Visual Basic

```
C1MaskedTextBox1.Value = "123"
```

C#

```
c1MaskedTextBox1.Value = "123";
```

Run your project and observe:

Initially **123** (or the number you chose) will appear in the control.

Adding a Mask for Currency

You can easily add a mask for currency values using the C1MaskedTextBox.Mask property. By default the C1MaskedTextBox control starts with its C1MaskedTextBox.Mask not set but you can customize this. For more details about mask characters, see Mask Elements.

At Design Time

To set the **C1MaskedTextBox.Mask** property, complete the following steps:

1. Click the C1MaskedTextBox control once to select it.

2. Navigate to the Properties tab and enter "$999,999.00" in the text box next to the **C1MaskedTextBox.Mask** property.

This will set the **C1MaskedTextBox.Mask** property to the number you chose.

In XAML

To set the **C1MaskedTextBox.Mask** property add Mask="$999,999.00" to the <Xaml:C1MaskedTextBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1MaskedTextBox Height="23" Width="120" Name="C1MaskedTextBox1"
Mask="$999,999.00"></Xaml:C1MaskedTextBox>
```

In Code

To set the **C1MaskedTextBox.Mask** property add the following code to your project:

Visual Basic

```
C1MaskedTextBox1.Mask = "$999,999.00"
```

C#

```
c1MaskedTextBox1.Mask = "$999,999.00";
```

Run your project and observe:

The mask will appear in the control. Enter a number; notice that the mask is filled.

Changing the Prompt Character

The C1MaskedTextBox.PromptChar property sets the characters that are used to prompt users in the **C1MaskedTextBox** control. By default the **C1MaskedTextBox.PromptChar** property is set to an underline character ("_") but you can customize this. For more details about the **C1MaskedTextBox.PromptChar** property, see Prompts.

At Design Time

To set the **C1MaskedTextBox.PromptChar** property, complete the following steps:

1. Click the C1MaskedTextBox control once to select it.
2. Navigate to the Properties tab and enter "0000" in the text box next to the **C1MaskedTextBox.Mask** property to set a mask.
3. In the properties window, enter "#" (the pound character) in the text box next to the **C1MaskedTextBox.PromptChar** property

In XAML

To set the C1MaskedTextBox.PromptChar property add Mask="0000" PromptChar="#" to the <Xaml:C1MaskedTextBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1MaskedTextBox Height="23" Name="C1MaskedTextBox1" Width="120" Mask="0000"
PromptChar="#"></Xaml:C1MaskedTextBox>
```

In Code

To set the **C1MaskedTextBox.PromptChar** property add the following code to your project:

Visual Basic

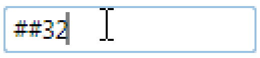
```
Dim x As Char = "#"c
C1MaskedTextBox1.Mask = "0000"
C1MaskedTextBox1.PromptChar = x
```

C#

```
char x = '#';
this.c1MaskedTextBox1.Mask = "0000";
this.c1MaskedTextBox1.PromptChar = x;
```

Run your project and observe:

The pound character will appear as the prompt in the control. In the following image, the number 32 was entered in the control:



Changing Font Type and Size

You can change the appearance of the text in the grid by using the text properties.

At Design Time

To change the font of the grid to Arial 10pt, complete the following:

1. Click the C1MaskedTextBox control once to select it.
2. Navigate to the Properties tab, and set **FontFamily** property to "Arial" (or a font of your choice).
3. In the Properties window, set the **FontSize** property to **10**.

This will set the control's font size and style.

In XAML

For example, to change the font of the control to Arial 10pt in XAML add FontFamily="Arial" FontSize="10" to the <Xaml:C1MaskedTextBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1MaskedTextBox Height="23" Name="C1MaskedTextBox1" Width="120" FontSize="10"
FontFamily="Arial"></Xaml:C1MaskedTextBox>
```

In Code

For example, to change the font of the grid to Arial 10pt add the following code to your project:

Visual Basic

```
C1MaskedTextBox1.FontSize = 10
C1MaskedTextBox1.FontFamily = New System.Windows.Media.FontFamily("Arial")
```

C#

```
c1MaskedTextBox1.FontSize = 10;
c1MaskedTextBox1.FontFamily = new System.Windows.Media.FontFamily("Arial");
```

Run your project and observe:

The control's content will appear in Arial 10pt font.

Locking the Control from Editing

By default the C1MaskedTextBox control's C1MaskedTextBox.Value property is editable by users at run time. If you want to lock the control from being edited, you can set the **IsReadOnly** property to **True**.

At Design Time

To lock the **C1MaskedTextBox** control from run-time editing, complete the following steps:

1. Click the C1MaskedTextBox control once to select it.
2. Navigate to the Properties tab and check the **IsReadOnly** check box.

   This will set the **IsReadOnly** property to **False**.

In XAML

To lock the **C1MaskedTextBox** control from run-time editing in XAML add IsReadOnly="True" to the <Xaml:C1MaskedTextBox> tag so that it appears similar to the following:

Markup

```
<Xaml:C1MaskedTextBox Height="23" Name="C1MaskedTextBox1" Width="120"
IsReadOnly="True"></Xaml:C1MaskedTextBox>
```

In Code

To lock the **C1MaskedTextBox** control from run-time editing add the following code to your project:

Visual Basic

```
C1MaskedTextBox1.IsReadOnly = True
```

C#

```
c1MaskedTextBox1.IsReadOnly = true;
```

Run your project and observe:

The control is locked from editing. Try to click the cursor within the control – notice that the text insertion point (the blinking vertical line) will not appear in the control.

Menu for UWP

Add touch-friendly context menus and classic "File" menu systems to your Universal Windows apps with **Menu for UWP**. The C1Menu and C1ContextMenu controls give you a real desktop look-and-feel with traditional looking menus that support deep nested items and vertical orientation.

- **C1ContextMenu**

  The C1ContextMenu provides a pop-up menu that provides frequently used commands that are associated with the selected object.

- **C1Menu**

  The C1Menu is a control that allows hierarchical organization of elements associated with event handlers.

Menu for UWP Key Features

Menu for UWP allows you to create customized, rich applications with the C1Menu and C1ContextMenu controls. Make the most of **Menu for UWP** by taking advantage of the following key features:

- **Display Holding Indicator**

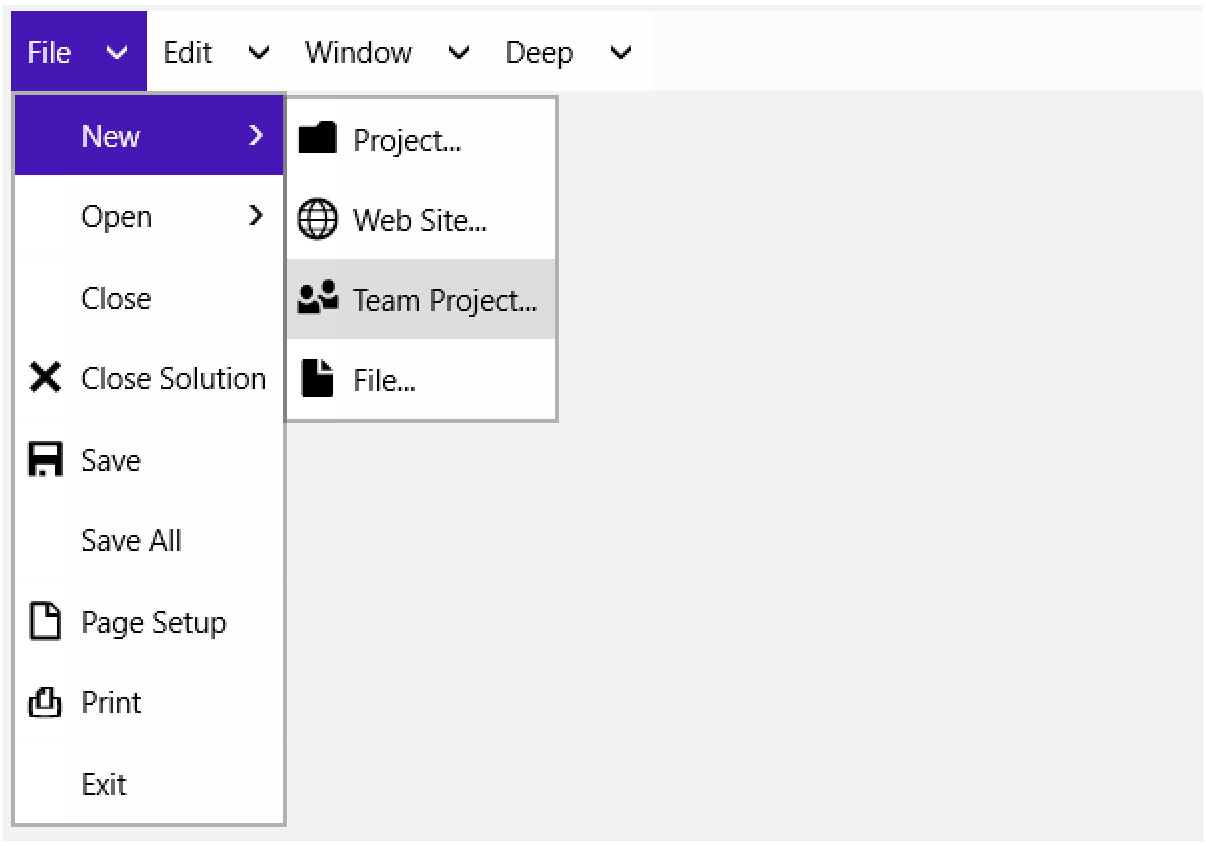
  With the **C1ContextMenu** control, a holding indicator can display when the user holds down on the parent control. This mimics the Windows desktop context menu experience on touch devices.

- **Page Boundaries Detection**

  Drop-down menus are positioned automatically and always stay within the page bounds. Long menus will show scroll buttons to indicate there are more menu items out of view.

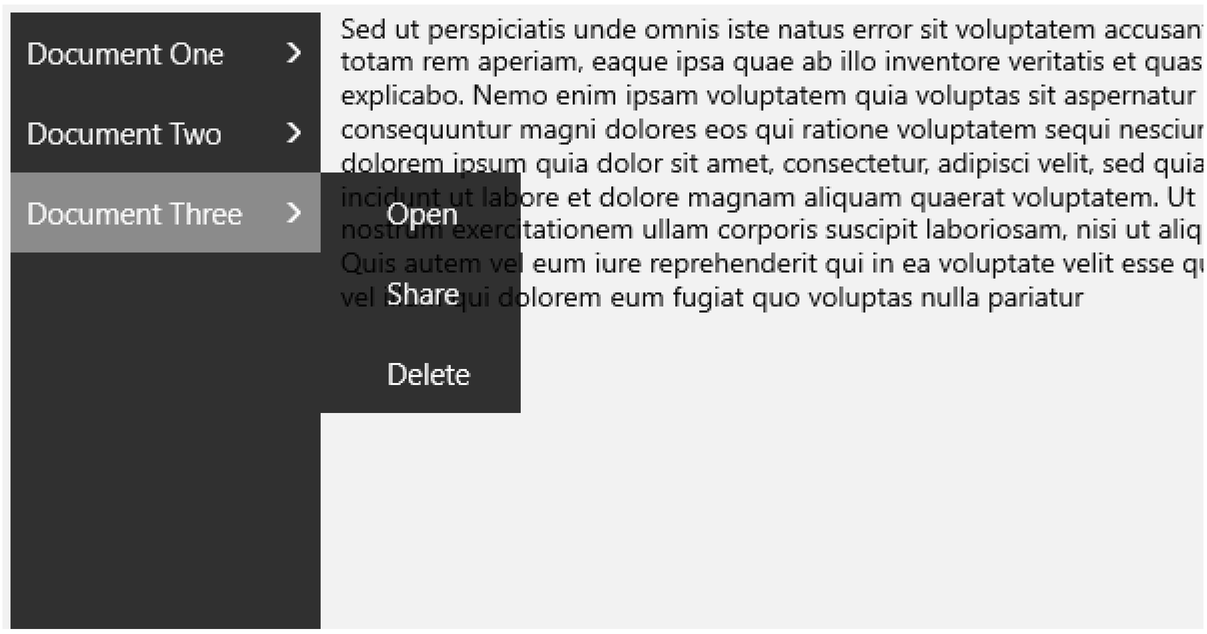
- **Icons and Custom Content**

  Display icons or any custom content for each menu item.



- **Horizontal or Vertical Orientation**

Set the Orientation to **Horizontal** or **Vertical**. Use the **C1Menu** control with the **C1DockPanel** control to dock it to any edge of the page.



- **Checked Items**

  C1MenuItem can be checked to show toggled state of features.

- **Easily Change Colors with ClearStyle**

  The **C1Menu** control supports **ClearStyle** technology which allows you to easily change control brushes without having to override templates. By just setting a few brush properties in Visual Studio you can quickly style each part of the control.

Menu for UWP Quick Start

The following quick start guide is intended to get you up and running with **Menu for UWP for UWP**. In this quick start, you'll create a new project with the C1Menu and C1ContextMenu controls. You will also add menu items, submenus, and a context menu to the control.

Step 1 of 4: Creating a Universal Windows Application

In this step, you'll create a Universal Windows application using the C1Menu and C1ContextMenu controls.

Complete the following steps:

1. In Visual Studio select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Right-click the project name in the Solution Explorer and select **Add Reference**.
4. In the **Reference Manager** dialog box, expand Universal Windows and select Extensions; you should see the UWP assemblies in the center pane. Select **C1.UWP** and click OK.

5. Open the **MainPage.xaml.cs** (or **MainPage.xaml.vb**) code file and add the following reference to the top of the page:

Visual Basic

```
Imports C1.Xaml
```

C#

```
using C1.Xaml;
```

6. Open **MainPage.xaml** and add the following markup within the <Page> tag:

Markup

```
xmlns:C1="using:C1.Xaml"
```

7. Add the following markup within the <Page> and </Page> tags:

Markup

```
<Page.Resources>
        <Style TargetType="Image" x:Key="MenuIcon">
            <Setter Property="Width" Value="16"/>
            <Setter Property="Height" Value="16"/>
            <Setter Property="Margin" Value="5 0 0 0"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="TextIconStyle">
            <Setter Property="FontSize" Value="20" />
            <Setter Property="FontFamily" Value="Segoe UI Symbol" />
            <Setter Property="FontWeight" Value="Normal" />
            <Setter Property="Foreground" Value="{StaticResource
AppBarItemForegroundThemeBrush}" />
            <Setter Property="HorizontalAlignment" Value="Center" />
            <Setter Property="VerticalAlignment" Value="Center" />
            <Setter Property="Margin" Value="5,-1,0,0"/>
        </Style>
        <Style TargetType="C1:C1MenuItem">
            <Setter Property="ItemContainerTransitions">
                <Setter.Value>
                    <TransitionCollection>
                        <RepositionThemeTransition/>
                        <EntranceThemeTransition/>
                    </TransitionCollection>
                </Setter.Value>
            </Setter>
            <Setter Property="ItemContainerTransitions">
                <Setter.Value>
                    <TransitionCollection>
                        <RepositionThemeTransition/>
                        <EntranceThemeTransition/>
                    </TransitionCollection>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>
```

This markup adds style resources.

8. Add the following markup within the <Grid> and </Grid> tags:

Markup
```
<Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition />
        </Grid.RowDefinitions>
```

This markup adds row definitions to the grid.

You have completed the first step of the **Menu for UWP** quick start. In this step, you created a Universal Windows project. In the next step, you will add tabs and tab pages to the control.

Step 2 of 4: Adding C1Menu to the Application

In the last step, you created a Universal Windows application. In this step, you will add a C1Menu control.

1. Place the cursor between the <Grid> and </Grid> tags in **MainPage.xaml**, and click once.
2. Add markup for a **C1DockPanel** control:

Markup
```
<C1:C1DockPanel Grid.Row="1" LastChildFill="False"></C1:C1DockPanel>
```

You'll add the **C1Menu** within this control.

3. Add markup for a **TextBlock** just after the </C1:C1DockPanel> tag:

Markup
```
<TextBlock x:Name="txt" Foreground="Red" Text="" FontSize="16"
VerticalAlignment="Bottom" HorizontalAlignment="Center" Margin="10" />
```

The name of any menu item you select will be displayed in the **TextBlock** at run time.

4. Add the following markup within the <C1:C1DockPanel> tags:

Markup
```
<C1:C1Menu x:Name="VisualStudioMenu" C1:C1DockPanel.Dock="Top"
DetectBoundaries="True" MinWidth="200" ItemClick="Menu_ItemClick">
    <C1:C1Menu.ItemContainerTransitions>
        <TransitionCollection>
            <EntranceThemeTransition/>
        </TransitionCollection>
    </C1:C1Menu.ItemContainerTransitions>
</C1:C1Menu>
```

This adds a **C1Menu** control.

5. Add the following markup just before the `</c1:C1Menu>` tag:

Markup

```
<C1:C1MenuItem Header="File">
<C1:C1MenuItem Header="New">
    <C1:C1MenuItem Header="Project..." IsCheckable="True" IsChecked="True" >
        <C1:C1MenuItem.Icon>
            <TextBlock Text="&#xE188;" Style="{StaticResource TextIconStyle}" />
        </C1:C1MenuItem.Icon>
    </C1:C1MenuItem>
    <C1:C1MenuItem Header="Web Site..." >
        <C1:C1MenuItem.Icon>
            <TextBlock Text="&#xE12B;" Style="{StaticResource TextIconStyle}" />
        </C1:C1MenuItem.Icon>
    </C1:C1MenuItem>
    <C1:C1MenuItem Header="Team Project..." >
        <C1:C1MenuItem.Icon>
            <TextBlock Text="&#xE125;" Style="{StaticResource TextIconStyle}" />
        </C1:C1MenuItem.Icon>
    </C1:C1MenuItem>
    <C1:C1MenuItem Header="File..." >
        <C1:C1MenuItem.Icon>
            <TextBlock Text="&#xE132;" Style="{StaticResource TextIconStyle}" />
        </C1:C1MenuItem.Icon>
    </C1:C1MenuItem>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Open">
    <C1:C1MenuItem Header="Project..." >
        <C1:C1MenuItem.Icon>
            <TextBlock Text="&#xE19C;" Style="{StaticResource TextIconStyle}" />
        </C1:C1MenuItem.Icon>
    </C1:C1MenuItem>
    <C1:C1MenuItem Header="Web Site..." >
        <C1:C1MenuItem.Icon>
            <TextBlock Text="&#xE12B;" Style="{StaticResource TextIconStyle}" />
        </C1:C1MenuItem.Icon>
    </C1:C1MenuItem>
    <C1:C1MenuItem Header="File..." >
        <C1:C1MenuItem.Icon>
            <TextBlock Text="&#xE19C;" Style="{StaticResource TextIconStyle}" />
        </C1:C1MenuItem.Icon>
    </C1:C1MenuItem>
</C1:C1MenuItem>
<C1:C1Separator />
<C1:C1MenuItem Header="Close" />
```

```
<C1:C1MenuItem Header="Close Solution">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE10A;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1Separator />
<C1:C1MenuItem Header="Save" >
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE105;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Save All" />
<C1:C1Separator />
<C1:C1MenuItem Header="Page Setup">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE160;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Print">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#x2399;" Style="{StaticResource TextIconStyle}"
FontWeight="ExtraBold" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1Separator />
<C1:C1MenuItem Header="Exit"/>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Edit">
<C1:C1MenuItem Header="Undo">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE10E;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Redo">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE10D;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1Separator />
<C1:C1MenuItem Header="Cut">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE16B;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Copy">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE16F;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Paste">
    <C1:C1MenuItem.Icon>
```

```
        <TextBlock Text="&#xE16D;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Delete" Width="100">
    <C1:C1MenuItem.Icon>
        <TextBlock Text="&#xE106;" Style="{StaticResource TextIconStyle}" />
    </C1:C1MenuItem.Icon>
</C1:C1MenuItem>
</C1:C1MenuItem>
<C1:C1MenuItem Header="Window">
<C1:C1MenuItem Header="Document 1" IsCheckable="True" IsChecked="True" />
<C1:C1MenuItem Header="Document 2" IsCheckable="True" IsChecked="True" />
<C1:C1MenuItem Header="Document 3" IsCheckable="True" />
<C1:C1MenuItem Header="Document 4" IsCheckable="True" />
<C1:C1Separator />
<C1:C1MenuItem Header="Exclusive 1" GroupName="Exclusives" IsCheckable="True"
IsChecked="True" />
<C1:C1MenuItem Header="Exclusive 2" GroupName="Exclusives" IsCheckable="True"
IsChecked="True" />
<C1:C1MenuItem Header="Exclusive 3" GroupName="Exclusives" IsCheckable="True" />
</C1:C1MenuItem>
<C1:C1MenuItem Header="Deep">
<C1:C1MenuItem Header="Deep1">
    <C1:C1MenuItem Header="Deep2">
        <C1:C1MenuItem Header="Deep3">
            <C1:C1MenuItem Header="Deep4">
                <C1:C1MenuItem Header="Deep5">
                </C1:C1MenuItem>
            </C1:C1MenuItem>
        </C1:C1MenuItem>
    </C1:C1MenuItem>
</C1:C1MenuItem>
</C1:C1MenuItem>
```

The above markup adds markup for menu structure.

6. Open the **MainPage.xaml.cs** (or **MainPage.xaml.vb**) page and add the following **Click** event handler to the project:

Visual Basic

```
Private Sub Menu_ItemClick(sender As Object, e As C1.Xaml.SourcedEventArgs)
   txt.Text = "Item Clicked: " & DirectCast(e.Source,
C1.Xaml.C1MenuItem).Header.ToString()
End Sub
```

C#

```
private void Menu_ItemClick(object sender, C1.Xaml.SourcedEventArgs e)
        {
            txt.Text = "Item Clicked: " +
((C1.Xaml.C1MenuItem)e.Source).Header.ToString();
        }
```

In this step, you added a C1Menu control. In the next step, you will add a C1ContextMenu control to the application.

Step 3 of 4: Adding a C1ContextMenu to the C1Menu Control

In the last step, you added submenus to the C1Menu control's menu items. In this step, you will add a C1ContextMenu control to the C1Menu control. This context menu will have one item that, when clicked, will add submenu items to the C1Menu control's top-level "Added Items" top-level menu item that you created in Step 2.

Complete the following steps:

1. In XAML view, place the following XAML markup right before the `</C1:C1Menu>` tag:

Markup

```
<C1:C1ContextMenuService.ContextMenu>
    <C1:C1ContextMenu x:Name="contextMenu" ItemClick="Menu_ItemClick">
        <C1:C1ContextMenu.ItemContainerTransitions>
            <TransitionCollection>
                <EntranceThemeTransition FromVerticalOffset="10"
FromHorizontalOffset="0" IsStaggeringEnabled="False"/>
            </TransitionCollection>
        </C1:C1ContextMenu.ItemContainerTransitions>
        <C1:C1MenuItem Header="Add">
            <C1:C1MenuItem.ItemContainerTransitions>
                <TransitionCollection>
                    <EntranceThemeTransition FromVerticalOffset="10"
FromHorizontalOffset="0" IsStaggeringEnabled="False"/>
                </TransitionCollection>
            </C1:C1MenuItem.ItemContainerTransitions>
            <C1:C1MenuItem Header="New Item">
                <C1:C1MenuItem.Icon>
                    <TextBlock Text="&#xE1DA;" Style="{StaticResource
TextIconStyle}" />
                </C1:C1MenuItem.Icon>
            </C1:C1MenuItem>
            <C1:C1MenuItem Header="Existing Item"/>
            <C1:C1MenuItem Header="Folder"/>
            <C1:C1Separator />
            <C1:C1MenuItem Header="Class"/>
        </C1:C1MenuItem>
        <C1:C1Separator />
        <C1:C1MenuItem Header="Exclude From Project"/>
        <C1:C1Separator />
        <C1:C1MenuItem Header="Cut">
            <C1:C1MenuItem.Icon>
                <TextBlock Text="&#xE16B;" Style="{StaticResource
TextIconStyle}" />
            </C1:C1MenuItem.Icon>
        </C1:C1MenuItem>
        <C1:C1MenuItem Header="Copy">
```

```
            <C1:C1MenuItem.Icon>
                <TextBlock Text="&#xE16F;" Style="{StaticResource
TextIconStyle}" />
            </C1:C1MenuItem.Icon>
        </C1:C1MenuItem>
        <C1:C1MenuItem Header="Paste">
            <C1:C1MenuItem.Icon>
                <TextBlock Text="&#xE16D;" Style="{StaticResource
TextIconStyle}" />
            </C1:C1MenuItem.Icon>
        </C1:C1MenuItem>
        <C1:C1MenuItem Header="Delete">
            <C1:C1MenuItem.Icon>
                <TextBlock Text="&#xE107;" Style="{StaticResource
TextIconStyle}" />
            </C1:C1MenuItem.Icon>
        </C1:C1MenuItem>
        <C1:C1MenuItem Header="Rename"/>
        <C1:C1Separator/>
        <C1:C1MenuItem Header="Properties"/>
    </C1:C1ContextMenu>
</C1:C1ContextMenuService.ContextMenu>
```

The above markup adds a C1ContextMenu control to the C1Menu control using the **C1ContextMenuService** helper class. Note that the C1ContextMenu control contains one C1MenuItem that is attached to the **Click** event named "Menu_ItemClick".

2. Add `x:Name="AddedItems"` to the `<c1:C1MenuItem Header="Added Items"/>` tag. This gives the item a unique identifier so that you can call it in code.

In this step, you added a C1ContextMenu control to the C1Menu control. In the next step, you will run the project and see the result of the **Menu for UWP** quick start.

Step 4 of 4: Running the Project

Now that you have created a Universal Windows project with a C1Menu control, the only thing left to do is run the project and observe the results of your work.

Complete the following steps:

1. Select **Debug | Start Debugging** to run the project.
2. Click **File** and observe that a submenu appears.
3. Hover your cursor over **New** and observe that another submenu appears.
4. Click **Window** and click **Document 4**.

   The **Window** submenu closes and the name of the item appears in the text box.
5. Right-click the menu and notice that the context menu appears.
6. Select and item in the context menu and observe that the name of the item appears in the TextBlock on the screen.

Congratulations! You have completed the **Menu for UWP Quick Start**.

Radial Menu for UWP

Add an attractive radial menu system to your Universal Windows apps with **RadialMenu for UWP**. Modeled after popular Microsoft apps, the C1RadialMenu control gives you a unique and touch-friendly alternative to the traditional context menu.

Radial Menu for UWP Key Features

RadialMenu for UWP allow you to create customized, rich applications. Make the most of **RadialMenu for UWP** by taking advantage of the following key features:

- **Nested Menus**

  The radial menus can be nested to any depth that you desire, and you can add as many items to the radial menu as you need to add. The C1RadialMenu control will automatically create the sectors based on the number of items your control contains.

- **Flexible Positioning**

  You can specify the exact position of each item within the C1RadialMenu control and you can even specify the angle at which the items begin.

- **Automatic Selection**

  Each menu item can contain any number of submenu items, and the C1RadialMenu control will show a selected item for each submenu. You can specify which submenu item is selected or allow the control to automatically select items based on the user's previous actions. So a frequently selected menu item that is not the default will be displayed on the main menu, allowing faster selection.

- **Automatic Collapsing**

  By default, the C1RadialMenu control and its submenus will remain open even when a user clicks outside the radial menu. However, you can change this behavior by enabling the automatic collapsing feature. This will allow users to close a radial menu by clicking outside the control's boundaries.

- **Checkable Menu Items**

  You can make any C1RadialMenuItem a checkable menu item by setting its IsCheckable property to True. In the C1RadialMenu, a checked item is marked similarly to a highlighted item instead of with a typical check mark.

- **Easily Change Colors with ClearStyle**

  The C1RadialMenu control supports **ComponentOne's ClearStyle** technology that allows you to easily change control brushes without having to override templates. By just setting a few brush properties in Visual Studio you can quickly style each part of the control.

Radial Menu for UWP Quick Start

The following quick start guide is intended to get you up and running with **RadialMenu for UWP**. In this quick start, you'll use Visual Studio to create a new project with the C1RadialMenu control.

Step 1 of 3: Creating a C1RadialMenu Application

In this step, you'll create a Universal Windows application using the C1RadialMenu control.

Complete the following steps:

1. In Visual Studio select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Right-click the project name in the Solution Explorer and select **Add Reference**.
4. In the **Reference Manager** dialog box, expand Universal Windows and select Extensions; you should see the UWP assemblies in the center pane. Select **C1.UWP** and click OK.
5. Open **MainPage.xaml** if it isn't already open, and add the following markup within the <Page> tag:

   Markup

   ```
   xmlns:Xaml="using:C1.Xaml"
   ```

   This adds the required references in the project.

6. Add the following markup within the <Page> and </Page> tags:

   Markup

   ```
   <Page.Resources>
       <Style TargetType="TextBlock" x:Key="TextIconStyle">
         <Setter Property="Margin" Value="-10" />
         <Setter Property="FontSize" Value="20" />
         <Setter Property="FontFamily" Value="Segoe UI Symbol" />
         <Setter Property="FontWeight" Value="Normal" />
         <Setter Property="Foreground" Value="#333333" />
         <Setter Property="HorizontalAlignment" Value="Center" />
         <Setter Property="VerticalAlignment" Value="Center" />
       </Style>
     </Page.Resources>
   ```

   This markup adds style resources that define the layout of our menu items content. We will add the menu items to C1RadialMenu in the next step. Each of our menu items will contain an Image and a TextBlock label

7. Add the following markup within the <Grid> and </Grid> tags:

   Markup

   ```
   <Border Background="LemonChiffon" MinHeight="40" BorderBrush="#969696"
   BorderThickness="1" Padding="5" HorizontalAlignment="Stretch"
   VerticalAlignment="Stretch">
   </Border >
   ```

   This markup adds border definition to the application.

You have completed the first step of the **RadialMenu for UWP** quick start. In this step, you created a Universal Windows project. In the next step, you will add a RadialMenu control and menu items.

Step 2 of 3: Adding RadialMenu Items to the Control

In the last step, you created a UWP project. In this step, you will add a C1RadialMenu control.

1. Place the cursor between the <Grid> and </Grid> tags in **MainPage.xaml**, and click once.
2. Add markup for a C1ContextMenuService.ContextMenu attached property:

Markup

```
<Xaml:C1ContextMenuService.ContextMenu>
</Xaml:C1ContextMenuService.ContextMenu>
```

The C1ContextMenuService enables C1RadialMenu to act as the context menu for any control. It will automatically appear when you right-click or right-tap on the parent control. In this case, the parent control is a Grid. Next, You'll add the C1RadialMenu within this control.

Note: If you prefer to show the C1RadialMenu control programmatically, you can skip C1ContextMenuService and simply call the C1RadialMenu.Show method in code.

3. Add the following markup within the <Xaml:C1ContextMenuService.ContextMenu > tags:

Markup

```
<Xaml:C1RadialMenu x:Name="contextMenu" Offset="-130,0"
ItemClick="contextMenu_ItemClick" ItemOpened="contextMenu_ItemOpened" >
 </Xaml:C1RadialMenu>
```

This adds a C1RadialMenu control.

4. Add the following markup within the `<Xaml:C1RadialMenu> </Xaml:C1RadialMenu>` tags:

Markup

```
<Xaml:C1RadialMenuItem Header="UndoRedo" SelectedIndex="0"
ShowSelectedItem="True" Command="{Binding UndoCommand, ElementName=page}">
       <Xaml:C1RadialMenuItem Header="Undo" >
          <Xaml:C1RadialMenuItem.Icon>
            <TextBlock Text="&#xE10E;" Style="{StaticResource TextIconStyle}" />
          </Xaml:C1RadialMenuItem.Icon>
        </Xaml:C1RadialMenuItem>
        <Xaml:C1RadialMenuItem Header="Redo" >
          <Xaml:C1RadialMenuItem.Icon>
            <TextBlock Text="&#xE10D;" Style="{StaticResource TextIconStyle}" />
          </Xaml:C1RadialMenuItem.Icon>
        </Xaml:C1RadialMenuItem>
        <Xaml:C1RadialMenuItem ToolTip="Clear Text Formatting" DisplayIndex="7">
        <Xaml:C1RadialMenuItem.Header>
                        <TextBlock TextWrapping="Wrap" MaxWidth="50"
TextAlignment="Center">Clear Format</TextBlock>
                    </Xaml:C1RadialMenuItem.Header>
                </Xaml:C1RadialMenuItem>
            </Xaml:C1RadialMenuItem>
            <Xaml:C1RadialMenuItem AutoSelect="True" ShowSelectedItem="True"
Header="Clipboard" SectorCount="8">
                <Xaml:C1RadialMenuItem Header="Cut">
                    <Xaml:C1RadialMenuItem.Icon>
```

```
                        <TextBlock Text="&#xE16B;" Style="{StaticResource
TextIconStyle}" />
                    </Xaml:C1RadialMenuItem.Icon>
                </Xaml:C1RadialMenuItem>
                <Xaml:C1RadialMenuItem Header="Copy">
                    <Xaml:C1RadialMenuItem.Icon>
                        <TextBlock Text="&#xE16F;" Style="{StaticResource
TextIconStyle}" />
                    </Xaml:C1RadialMenuItem.Icon>
                </Xaml:C1RadialMenuItem>
                <Xaml:C1RadialMenuItem Header="Paste">
                    <Xaml:C1RadialMenuItem.Icon>
                        <TextBlock Text="&#xE16D;" Style="{StaticResource
TextIconStyle}" />
                    </Xaml:C1RadialMenuItem.Icon>
                </Xaml:C1RadialMenuItem>
            </Xaml:C1RadialMenuItem>
            <Xaml:C1RadialMenuItem Header="Add" SectorCount="6">
                <Xaml:C1RadialMenuItem.Icon>
                    <TextBlock Text="&#xE109;" Style="{StaticResource
TextIconStyle}" />
                </Xaml:C1RadialMenuItem.Icon>
                <Xaml:C1RadialMenuItem Header="New" ToolTip="New Item">
                    <Xaml:C1RadialMenuItem.Icon>
                        <TextBlock Text="&#xE1DA;" Style="{StaticResource
TextIconStyle}" />
                    </Xaml:C1RadialMenuItem.Icon>
                </Xaml:C1RadialMenuItem>
                <Xaml:C1RadialMenuItem Header="Existing" ToolTip="Existing
Item">
                    <Xaml:C1RadialMenuItem.Icon>
                        <TextBlock Text="&#xE132;" Style="{StaticResource
TextIconStyle}" />
                    </Xaml:C1RadialMenuItem.Icon>
                </Xaml:C1RadialMenuItem>
                <Xaml:C1RadialMenuItem Header="Folder">
                    <Xaml:C1RadialMenuItem.Icon>
                        <TextBlock Text="&#xE188;" Style="{StaticResource
TextIconStyle}" />
                    </Xaml:C1RadialMenuItem.Icon>
                </Xaml:C1RadialMenuItem>
                <Xaml:C1RadialMenuItem Header="Class">
                    <Xaml:C1RadialMenuItem.Icon>
                        <TextBlock Text="&#xE1D3;" Style="{StaticResource
TextIconStyle}" />
                    </Xaml:C1RadialMenuItem.Icon>
                </Xaml:C1RadialMenuItem>
            </Xaml:C1RadialMenuItem>
            <Xaml:C1RadialMenuItem Header="Exclude" ToolTip="Exclude From
Project" >
                <Xaml:C1RadialMenuItem.Icon>
```

```
                    <TextBlock Text="&#xE10A;" Style="{StaticResource
TextIconStyle}" />
                </Xaml:C1RadialMenuItem.Icon>
            </Xaml:C1RadialMenuItem>
            <Xaml:C1RadialMenuItem Header="Delete">
                <Xaml:C1RadialMenuItem.Icon>
                    <TextBlock Text="&#xE107;" Style="{StaticResource
TextIconStyle}" />
                </Xaml:C1RadialMenuItem.Icon>
            </Xaml:C1RadialMenuItem>
            <Xaml:C1RadialMenuItem Header="Rename">
                <Xaml:C1RadialMenuItem.Icon>
                    <TextBlock Text="&#xE13E;" Style="{StaticResource
TextIconStyle}" />
                </Xaml:C1RadialMenuItem.Icon>
            </Xaml:C1RadialMenuItem>
            <Xaml:C1RadialMenuItem Header="Properties">
                <Xaml:C1RadialMenuItem.Icon>
                    <TextBlock Text="&#xE115;" Style="{StaticResource
TextIconStyle}" />
                </Xaml:C1RadialMenuItem.Icon>
            </Xaml:C1RadialMenuItem>
```

The above markup adds menu items to the RadialMenu.

5. Add the following markup directly below the </Xaml:C1ContextMenuService.ContextMenu> tags:

Markup

```
<TextBlock Text="Press the context-menu button over this text (right-click in
Windows)." Foreground="Sienna" FontSize="16" TextWrapping="Wrap"
HorizontalAlignment="Center" VerticalAlignment="Center"/>
</Border>
<TextBlock x:Name="txt" Foreground="Red" Text="" FontSize="16"
VerticalAlignment="Bottom" HorizontalAlignment="Center" Margin="10" />
```

This will add two general TextBlock controls to the application, one before the </Border> tag, and one after it. The first TextBlock will contain the directions for how to display the C1RadialMenu. The second TextBlock will display the C1RadialMenuItem you have tapped or clicked, or opened.

6. Open the **MainPage.xaml.cs** page and add the following **Click** event handlers to the project:

C#

```
private void contextMenu_ItemClick(object sender, C1.Xaml.SourcedEventArgs e)
        {
            txt.Text = "Item Clicked: " +
((C1.Xaml.C1RadialMenuItem)e.Source).Header.ToString();
        }
        private void contextMenu_ItemOpened(object sender,
C1.Xaml.SourcedEventArgs e)
        {
            txt.Text = "Item Opened: " +
((C1.Xaml.C1RadialMenuItem)e.Source).Header.ToString();
        }
```

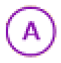
In this step, you added a C1RadialMenu control. In the next step, you will run the project and see the result of the **RadialMenu for UWP** quick start.

Step 3 of 3: Running the Project

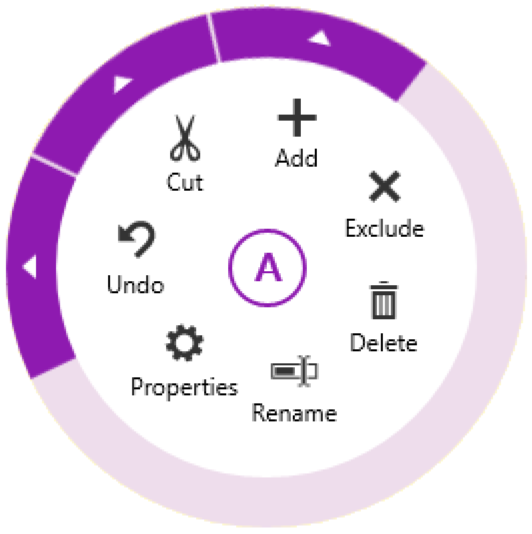
Now that you have created a Universal Windows project with a C1RadialMenu control, the only thing left to do is run the project and observe the results of your work.

Complete the following steps:

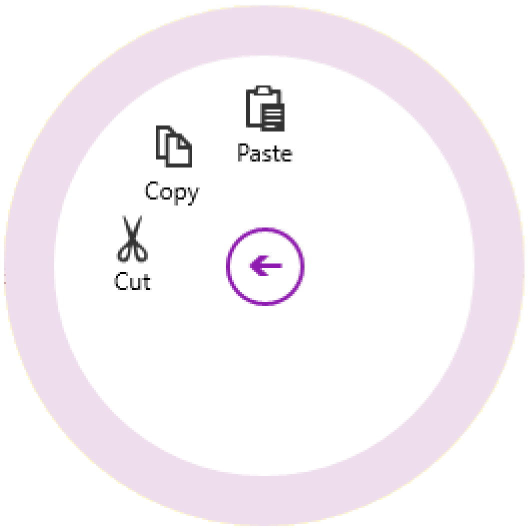
1. Select **Debug | Start Debugging** to run the project. When you right tap or right click on the page, the Navigation Button will appear as in the following image:



2. Tap or click the Navigation Button to display the radial menu:



3. Tap the **ExpandArea** above the **Cut** C1RadialMenuItem and observe that the **Clipboard** menu appears:



4. To go back to the main radial menu, tap the purple arrow in the center of the C1RadialMenu control.

Congratulations! You have completed the **RadialMenu for UWP** quick start.

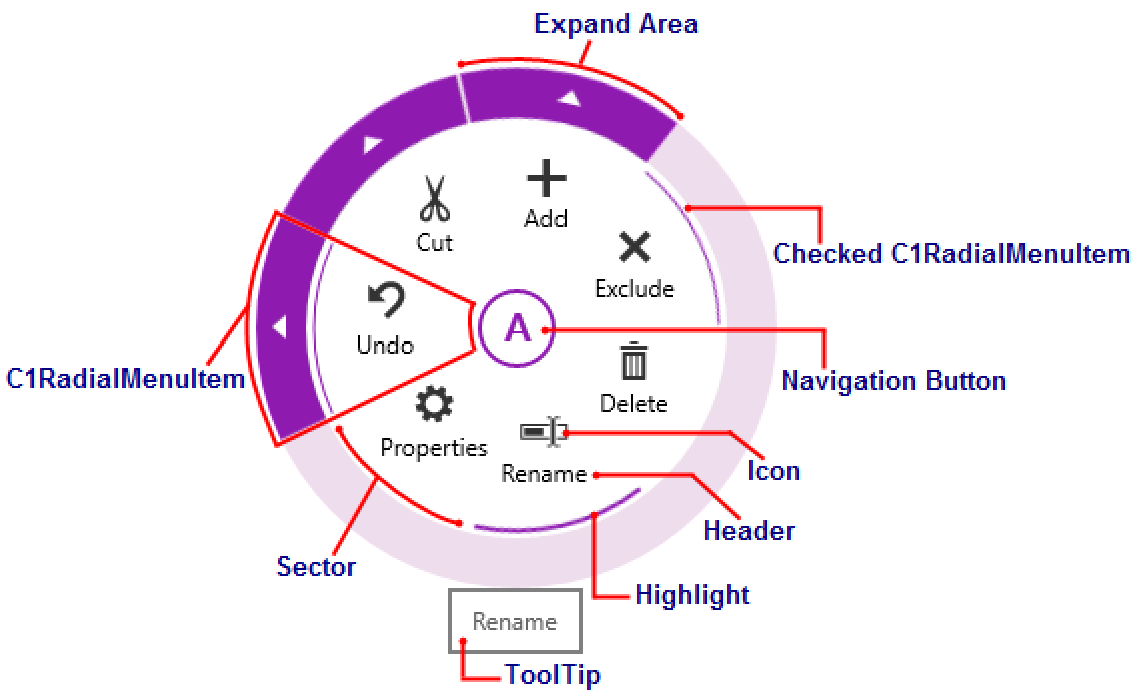
WorkingwithC1Radial Menu

The following topics outline the basics of working with the C1RadialMenu control. In this section, you will find outlines of the controls' elements and descriptions of some of the controls' most popular features.

C1RadialMenu Elements

The C1RadialMenu is a control that allows radial organization of elements. The radial menus can be nested to any depth that you desire, and you can add as many items to the radial menu as you need to add.

The following image diagrams the elements of the C1RadialMenu control.



The elements of the C1RadialMenu control can be described as follows:

- **C1RadialMenu**

  The main radial menu is a circular, top-level context menu control. It is comprised of first-level C1RadialMenuItems and can hold any feature available to C1RadialMenuItems.

- **C1RadialMenuItem**

  Radial menu items are represented by the C1RadialMenuItem class. C1RadialMenuItems can have a Header and an Icon, and you can also set them as Checkable. Each radial menu item might be associated with a Command, which is invoked when the button is pressed. A C1RadialMenuItem that contains child items will also have an **Expand Area**.

- **Sectors**

  The C1RadialMenu is split into sectors. The C1RadialMenu control will automatically create the sectors based on the number of C1RadialMenuItem your control contains, but you can customize the number of sectors using the SectorCount property.

- **Navigation Button**

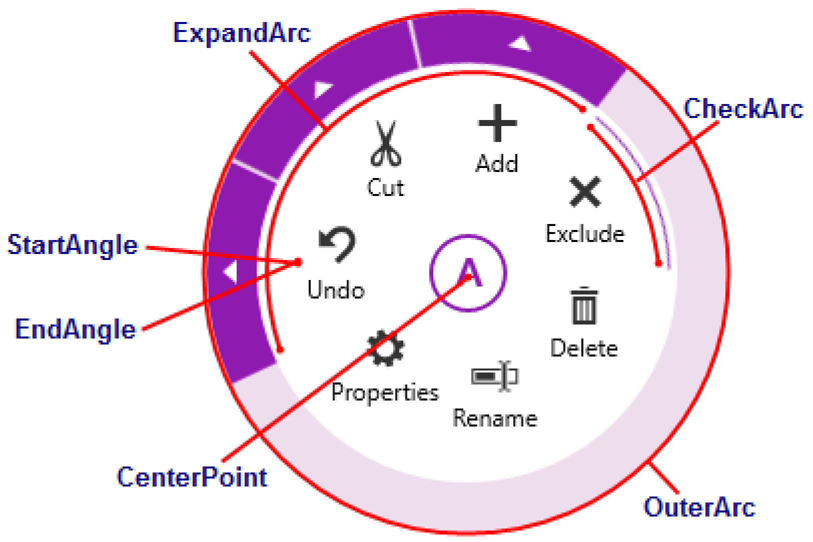
  When a user right taps the application, the Navigation Button will appear. Tap the Navigation Button to show or hide the C1RadialMenu. This button will turn into a back arrow when a user navigates to a submenu.

- **Icon**

  Using the C1RadialMenuItem.Icon property, you can customize the icons that appear on your C1RadialMenu.

C1RadialMenuItem and C1RadialPanel Elements

There are a few properties that are used to draw the geometrical paths in the C1RadialMenuItem default control template. By default, these properties are set by the C1RadialPanel, but you can use them to create custom control templates by setting them explicitly in the C1RadialPanel.



C1RadialMenuItem Elements

The C1RadialMenuItem elements are set by the C1RadialPanel by default. They draw the geometric paths in the C1RadialMenuItem control. The default control template uses them, and you can also use them in custom control templates.

- **CenterPoint**

  The CenterPoint property gets the coordinates of the circle's center. This can be used to draw the circle sector in XAML, representing the current C1RadialMenuItem.

- **CheckArc**

  The CheckArc property gets the definition of the check arc segment.

- **ExpandArc**

  The ExpandArc property gets the definition of the expand area arc segment.

- **OuterArc**

  The OuterArc property gets the definition of the outer arc segment. This can be used to draw the circle sector in XAML, representing the current C1RadialMenuItem.

C1RadialPanel Elements

The C1RadialPanel is created in the default C1RadialMenu style. It's used by the C1RadialMenuItemsPresenter as an **ItemsPanel**. The C1RadialPanel makes use of all 360 degrees available to it so that radial menu items are arranged within the circle. If you need to change the StartAngle or EndAngle, then you can use XAML like the following to redefine the default style and use custom settings:

Markup
```
<Setter Property="ItemsPanel">
   <Setter.Value>
     <ItemsPanelTemplate>
       <c1:C1RadialPanel StartAngle="-180" EndAngle="180" />
     </ItemsPanelTemplate>
   </Setter.Value>
 </Setter>
```

- **StartAngle**

  The StartAngle defines the place where the first C1RadialMenuItem should be located. In the sample above, -180 corresponds to 9 o'clock on a clock face. This is the default StartAngle.

- **EndAngle**

  The EndAngle defines the place where the last C1RadialMenuItem should be located. In the sample above, 180 corresponds to 9 o'clock on a clock face. This is the default EndAngle. Starting and ending the sweep of the angle at a similar point ensures that the menu items will be arranged properly around the C1RadialMenu.

C1RadialMenu Features

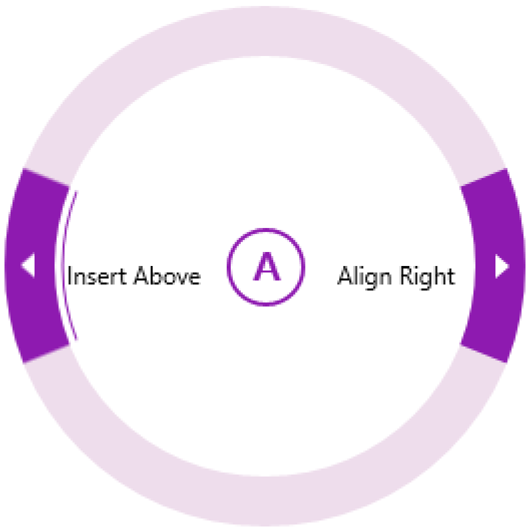
Automatic Collapsing

By default, the C1RadialMenu control and its submenus will remain open even when a user clicks outside of the radial menu. The only way users can close the radial menu is if they click the Navigation Button in the center of the C1RadialMenu control. However, you can change this behavior by enabling the automatic collapsing feature, which will allow users to close a radial menu by clicking outside of the control's boundaries. To turn on automatic collapsing, set the AutoCollapse property to **True**.

Checkable Radial Menu Items

You can make any C1RadialMenuItem a checkable RadialMenu item by setting its IsCheckable property to **True**.

In the C1RadialMenu, a checked item is marked similarly to a highlighted item instead of with a typical check mark. You can see the **Insert Above** option checked in the image below. Note that the check is thinner than the highlight you can see in the C1RadialMenu Elements topic:



You can create a group of mutually exclusive checkable items by setting the GroupName property of each item you wish to add to the group. For example, the XAML below will create a group of four mutually exclusive checkable items.

Markup

```
 <Xaml:C1RadialMenu SectorCount="8" >
  <Xaml:C1RadialMenuItem Header="Insert" SectorCount="8" AutoSelect="True"
ShowSelectedItem="True" IsCheckable="True" >
    <Xaml:C1RadialMenuItem Header="Insert Left" IsCheckable="True"
GroupName="MutuallyExclusiveGroup"/>
    <Xaml:C1RadialMenuItem Header="Insert Above" DisplayIndex="2" IsCheckable="True"
GroupName="MutuallyExclusiveGroup"/>
    <Xaml:C1RadialMenuItem Header="Insert Right" DisplayIndex="4" IsCheckable="True"
GroupName="MutuallyExclusiveGroup"/>
    <Xaml:C1RadialMenuItem Header="Insert Below" DisplayIndex="6" IsCheckable="True"
GroupName="MutuallyExclusiveGroup" />
  </Xaml:C1RadialMenuItem>
</Xaml:C1RadialMenu>
```

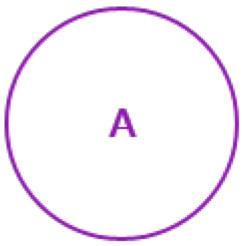
A mutually exclusive group will only allow one C1RadialMenuItem to be checked at one time.

Navigation Button

The C1RadialMenu's **Navigation Button** is the first thing a user will see when they right tap the application. You can customize the **Navigation Button** with the C1RadialMenu's Icon property and NavigationButtonRelativeSize property. The NavigationButtonRelativeSize property allows you to set the Navigation Button's size relative to the size of the C1RadialMenu. By default, this property is set to 0.15.

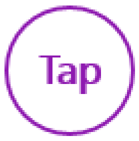
For example, this is the default Navigation Button, with the NavigationButtonRelativeSize set to 0.15:

Here's the Navigation Button when you set the NavigationButtonRelativeSize property to 0.45:



You can also change the Icon property. By default, the Icon property is set to "A".

Here's the default Navigation Button with the Icon property changed, and the NavigationButtonRelativeSize property set to 0.25:



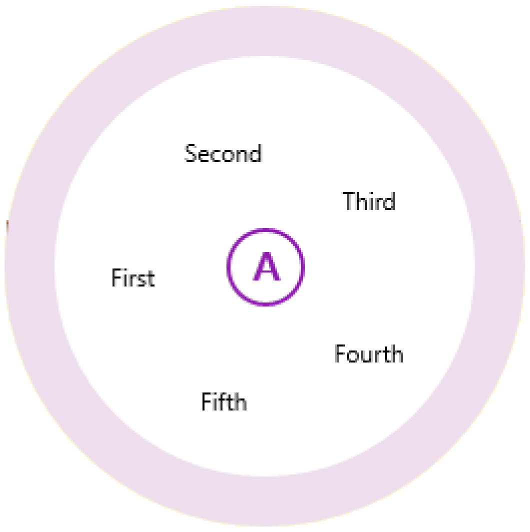
Nested Submenus

The C1RadialMenu control can hold submenus. These submenus are created when C1RadialMenuItems are nested within the tags of other C1RadialMenuItems. For example, placing the following XAML markup

Markup

```
<c1:C1RadialMenuItem Header="First">
     <c1:C1RadialMenuItem Header="Second">
          <c1:C1RadialMenuItem Header="Third">
               <c1:C1RadialMenuItem Header="Fourth">
                    <c1:C1RadialMenuItem Header="Fifth"/>
</c1:C1RadialMenuItem>
```

between the opening and closing tags of a C1RadialMenu would create the following:

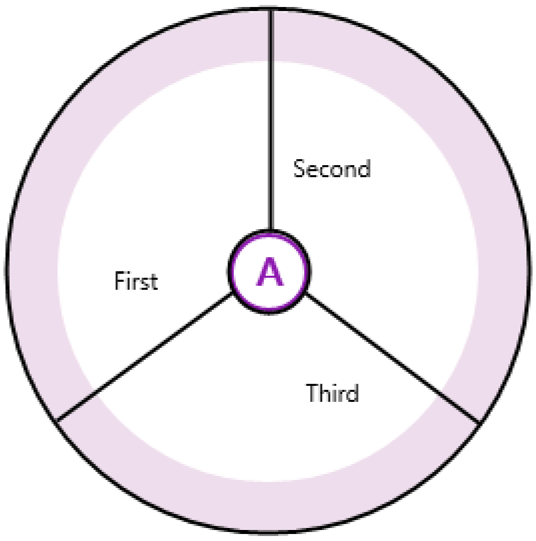


You can have as many nested radial menus as you want, although it's best not to have more than two or three submenus in a hierarchy for usability purposes.

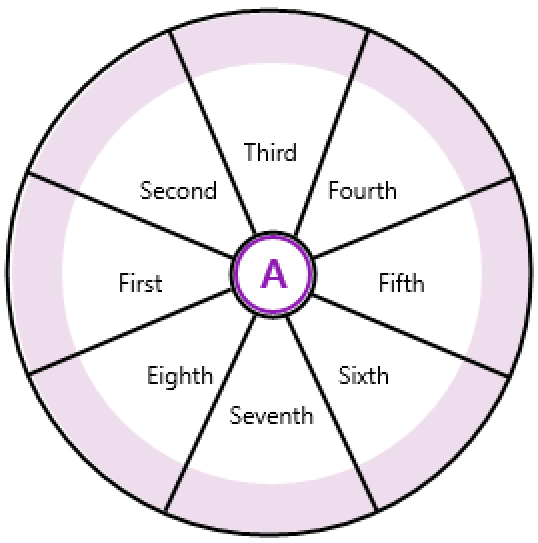
For task-based help on creating a nested submenu for a C1RadialMenu control, see Creating a Submenu.

Positioning Items

By default, C1RadialMenuItems will be positioned to equally fill the 360 degree menu. If, for example, a C1RadialMenu has three items, they would be positioned so that each item takes up a third of the C1RadialMenu:

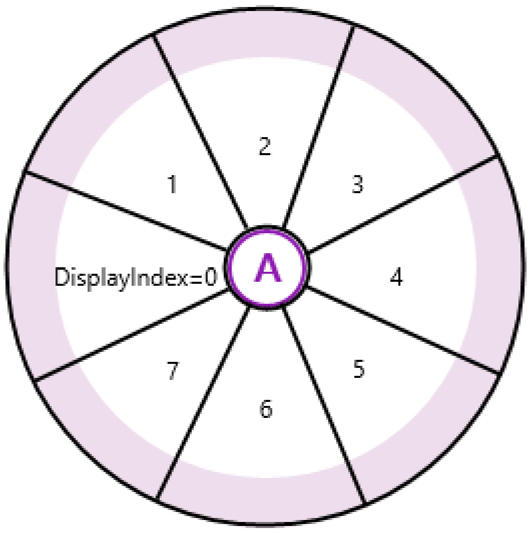


But if you have a C1RadialMenu with 8 items, it will resemble the following image:



Note that each item takes up an eighth of the radial menu. Each C1RadialMenuItem is positioned inside a sector. You can customize the number of sectors with the SectorCount property. This property can be set for both the main C1RadialMenu and for any C1RadialMenuItem that contains child items.

The SectorCount property, if used with the DisplayIndex property, allows you to fully customize C1RadialMenuItem positioning. The DisplayIndex property uses zero-based indexing to define how the C1RadialMenuItems are displayed. For example, in a C1RadialMenu with the SectorCount set to "8", the display indices would be as follows:



Note that the indices begin at the center left side of the C1RadialMenu and continue around the menu in a clockwise direction.

For example, the following C1RadialMenu will display just two C1RadialMenuItems at index 3 and index 4:

Markup

```
<Xaml:C1RadialMenu SectorCount=8>
     <Xaml:C1RadialMenuItem DisplayIndex="3" />
```

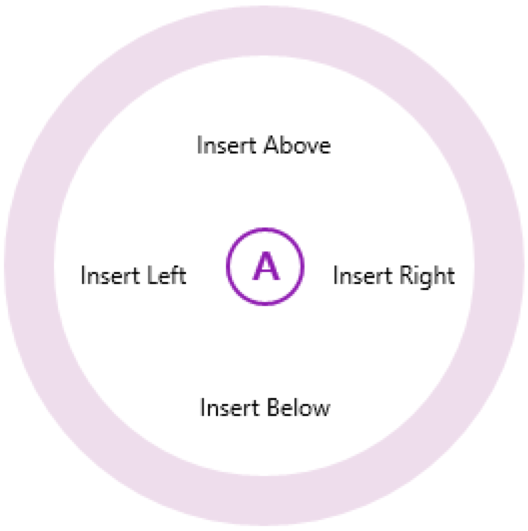
```
    <Xaml:C1RadialMenuItem DisplayIndex="4" />
</Xaml:C1RadialMenu>
```

You can also use the SectorCount property and the DisplayIndex property to display items at four different positions:

Markup

```
<Xaml:C1RadialMenu SectorCount="8">
    <Xaml:C1RadialMenuItem Header="Insert Left"/>
    <Xaml:C1RadialMenuItem Header="Insert Above" DisplayIndex="2" />
    <Xaml:C1RadialMenuItem Header="Insert Right" DisplayIndex="4" />
    <Xaml:C1RadialMenuItem Header="Insert Below" DisplayIndex="6" />
 </Xaml:C1RadialMenu>
```

The markup above will create a C1RadialMenu control like the one in the following image:



Selecting Items

C1RadialMenuItems can have any number of child items. You can control which item is shown as the selected item by setting the SelectedIndex property, but by default, the C1RadialMenuItem will show the first child item in its collection as the selected item. You can also set the AutoSelect property to true to have the same effect.

The C1RadialMenu control supports selection memory, as well, through the ShowSelectedItem property. This property allows you to show the last selected item rather than the preset one. You must set the AutoSelect property to True when you are using the ShowSelectedItem property.

So, if you have a user who right-aligns text frequently, it would be best to show the item that was selected last rather than the default align-left option or the align-center option. You can set the ShowSelectedItem property to True so that it will always show the item that a user selected last, rather than the preset item or the first selected item.

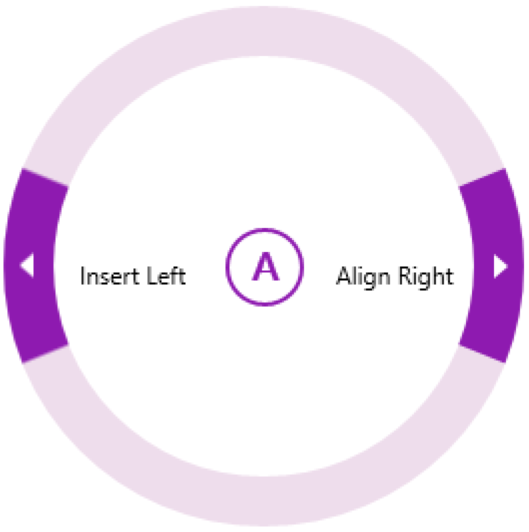
Here's the markup to create a simple C1RadialMenu with two submenus:

Markup

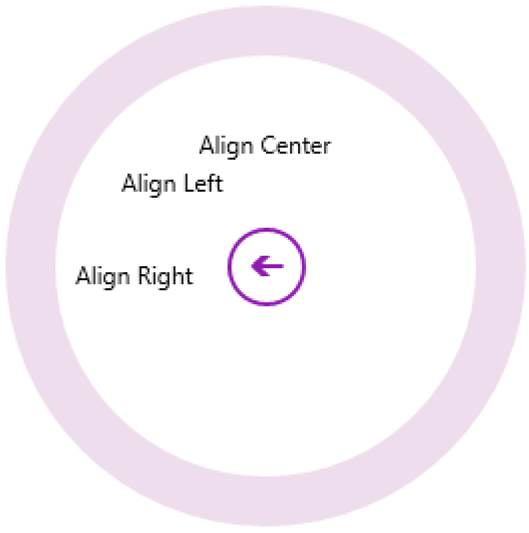
```
<Xaml:C1RadialMenu SectorCount="8" >
  <Xaml:C1RadialMenuItem Header="Insert" SectorCount="8" AutoSelect="True"
ShowSelectedItem="True" >
```

```
    <Xaml:C1RadialMenuItem Header="Insert Left" />
    <Xaml:C1RadialMenuItem Header="Insert Above" DisplayIndex="2" />
    <Xaml:C1RadialMenuItem Header="Insert Right" DisplayIndex="4" />
    <Xaml:C1RadialMenuItem Header="Insert Below" DisplayIndex="6" />
  </Xaml:C1RadialMenuItem>
  <Xaml:C1RadialMenuItem Header="Align" AutoSelect="True" SectorCount="8"
DisplayIndex="4" ShowSelectedItem="True">
    <Xaml:C1RadialMenuItem Header="Align Right" />
    <Xaml:C1RadialMenuItem Header="Align Left" />
    <Xaml:C1RadialMenuItem Header="Align Center" />
  </Xaml:C1RadialMenuItem>
</Xaml:C1RadialMenu>
```

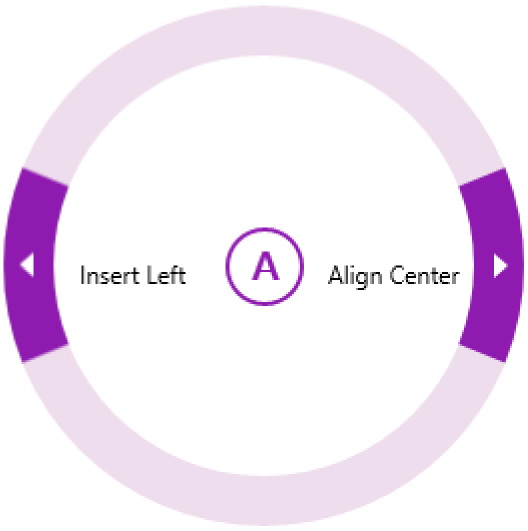
When you run the application, it will resemble the following image:



When you tap the **Expand Area** above the **Align Right** option, you will see the following menu:



Tap **Align Center**, and then the **Back** arrow. Your main menu will now resemble the following image:



Note that **Align Right** has been replaced by the last-selected **Align Center**.

Radial Menu for UWP Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET and know how to use the C1RadialMenu control in general. If you are unfamiliar with the **Radial Menu for UWP** product, please see the Radial Menu for UWP Quick Start first.

Each topic in this section provides a solution for specific tasks using the C1RadialMenu control.

Each task-based help topic also assumes that you have created a new Universal Windows project.

Creating Radial Menus

Creating a Top-Level Menu

In this topic, you will learn how to create a top-level radial menu for the C1RadialMenu control.

Complete the following steps:

1. Place the following XAML between the <Grid> and </Grid> tags:

Markup

```
<Xaml:C1ContextMenuService.ContextMenu>
     <Xaml:C1RadialMenu >
    </Xaml:C1RadialMenu>
 </Xaml:C1ContextMenuService.ContextMenu>
```

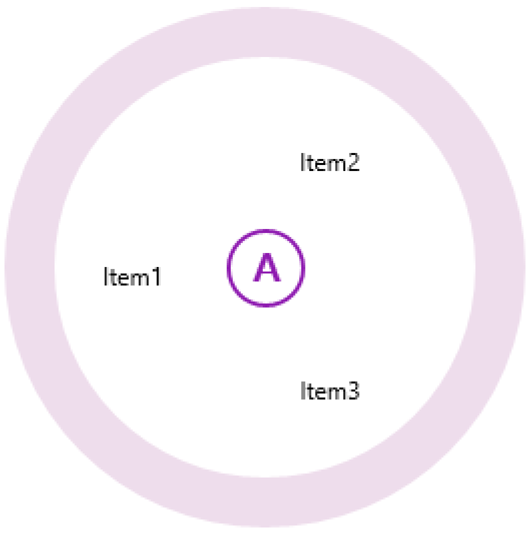
2. Place the following XAML between the `<Xaml:C1RadialMenu>` and `</Xaml:C1RadialMenu>` tags:

Markup
```
<Xaml:C1RadialMenuItem Header="RadialMenuItem1" />
<Xaml:C1RadialMenuItem Header="RadialMenuItem2" />
<Xaml:C1RadialMenuItem Header="RadialMenuItem3" />
```

3. Run the program and observe the following:

- Right tap or right-click the page. In this case, this is the element to which the C1RadialMenu is attached. Observe that the Navigation Button appears.
- Tap or click the Navigation Button to open the radial menu. Observe that the C1RadialMenu contains three C1RadialMenuItems.

This topic illustrates the following:



Creating a Submenu

In this topic, you will create a submenu that's attached to one of a C1RadialMenu's items. This topic assumes that you have created a top-level radial menu (see Creating a Top-Level Menu) with at least one C1RadialMenuItem.

Complete the following steps:

1. Place the following XAML between the `<Xaml:C1RadialMenu>` and `</Xaml:C1RadialMenu>` tags.

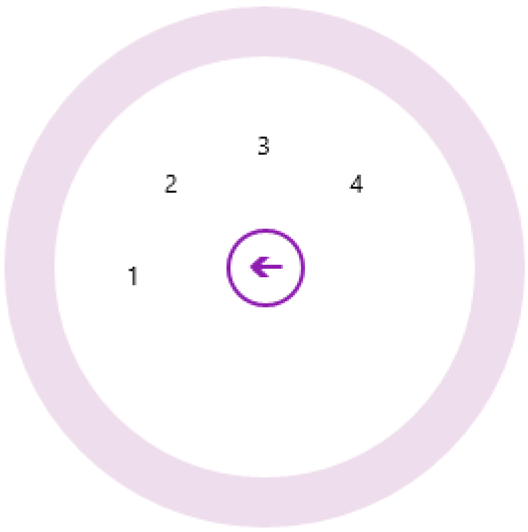
Markup
```
<Xaml:C1RadialMenuItem Header="Tap Here" >
  <Xaml:C1RadialMenuItem Header="Item 1" />
  <Xaml:C1RadialMenuItem Header="Item 2" />
  <Xaml:C1RadialMenuItem Header="Item 3" />
</Xaml:C1RadialMenuItem>
```

2. Run the program.

- Right tap or right-click the page to display the Navigation Button. Tap or click the Navigation Button to display the main **C1RadialMenu**.
- Tap the **Tap Here** item and observe that the submenu you created appears.

This Topic Illustrates the Following:

This is the **C1RadialMenu** control after you tap the **Tap Here** item:



Creating a Color Picker Menu

The C1RadialColorItem allows you to create a color picker using the C1RadialMenu control. In this topic, you'll create a C1RadialMenu application and add C1RadialColorItems to the C1RadialMenu control. You'll use both XAML markup and code to create the application.

Complete the following steps:

1. Locate the <Grid> </Grid> tags on your page and replace them with the following markup to create the framework for your application:

Example Title

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Border Background="LemonChiffon" MinHeight="40" BorderBrush="#969696"
                BorderThickness="1" Padding="5" HorizontalAlignment="Stretch"
VerticalAlignment="Stretch">
            <Xaml:C1ContextMenuService.ContextMenu>

            </Xaml:C1ContextMenuService.ContextMenu>

           <Xaml:C1ListViewer x:Name="text" Foreground="Sienna"
HorizontalAlignment="Stretch" VerticalAlignment="Center" Height="75"
ZoomMode="Disabled" Xaml:C1NagScreen.Nag="True">
                <TextBlock Text="Touch: hold down for a few seconds until the
indicator displays.&#10;Keyboard: press the context-menu button over this
text.&#10;Mouse: right-click over this text."
                           FontSize="16" TextWrapping="Wrap" />
            </Xaml:C1ListViewer>
        </Border>
        <TextBlock x:Name="txt" Foreground="Red" Text="" FontSize="16"
VerticalAlignment="Bottom" HorizontalAlignment="Center" Margin="10" />
    </Grid>
```

2. Locate the <Xaml:C1ContextMenuService.ContextMenu> </Xaml:C1ContextMenuService.ContextMenu> tags within the markup you just added. Place your cursor between the tags.
3. Locate the **C1RadialMenu** control in the Visual Studio ToolBox and double-click it to add it to your application.
4. Edit the opening <C1RadialMenu> tag so that it resembles the following:

Markup

```
<Xaml:C1RadialMenu x:Name="contextMenu" Offset="-130,0"
ItemClick="contextMenu_ItemClick" >
```

5. Add the following markup between the <C1RadialMenu> </C1RadialMenu> tags to add three C1RadialColorItems to your application:

Markup

```
<Xaml:C1RadialColorItem x:Name="rainbowItem" ShowSelectedItem="True" AutoSelect="True"
IsSelectable="False" >
</Xaml:C1RadialColorItem>
<Xaml:C1RadialColorItem x:Name="greenItem" SelectedIndex="5" ShowSelectedItem="True"
AutoSelect="True" IsSelectable="False" >
</Xaml:C1RadialColorItem>
<Xaml:C1RadialColorItem x:Name="blueItem" SelectedIndex="0" ShowSelectedItem="True"
AutoSelect="True" IsSelectable="False" >
</Xaml:C1RadialColorItem>
```

6. Select the **C1RadialColorItem** named "rainbowItem" and insert the following markup between the opening and closing tags. This will add **SolidColorBrush** submenu items:

Markup

```
<SolidColorBrush Color="Red"/>
<SolidColorBrush Color="Orange"/>
<SolidColorBrush Color="Yellow"/>
<SolidColorBrush Color="Green"/>
<SolidColorBrush Color="Blue"/>
<SolidColorBrush Color="Indigo"/>
<SolidColorBrush Color="Violet"/>
```

7. Select the "greenItem" C1RadialColorItem and insert the following markup between the opening and closing tags. This will add C1RadialColorItem submenu items in shades of green:

Markup

```
<Xaml:C1RadialColorItem x:Name="greenItem" SelectedIndex="5"
                                      ShowSelectedItem="True" AutoSelect="True"
IsSelectable="False" Xaml:C1NagScreen.Nag="True">
                        <Xaml:C1RadialColorItem ToolTip="Lime" Brush="#FF92D050"
GroupName="Green"/>
                        <Xaml:C1RadialColorItem ToolTip="Light Green"
Brush="#FFC6EFCE" GroupName="Green"/>
                        <Xaml:C1RadialColorItem ToolTip="Green" Brush="#FF00FF00"
GroupName="Green"/>
```

```
                        <Xaml:C1RadialColorItem ToolTip="Dark Green" Brush="#FF1D421E"
GroupName="Green"/>
                        <Xaml:C1RadialColorItem ToolTip="Dark Green" Brush="#FF1D5A2D"
GroupName="Green"/>
                        <Xaml:C1RadialColorItem ToolTip="Dark Green" Brush="Green"
GroupName="Green"/>
                        <Xaml:C1RadialColorItem ToolTip="Dark Green" Brush="#FF008000"
GroupName="Green"/>
                        <Xaml:C1RadialColorItem ToolTip="Dark Green" Brush="#FF00B050"
GroupName="Green"/>
                    </Xaml:C1RadialColorItem>
```

8. Select the "blueItem" **C1RadialColorItem** and insert the following markup between the opening and closing tags. This will add **C1RadialColorItem** submenu items in shades of blue:

Markup

```
<Xaml:C1RadialColorItem x:Name="blueItem" SelectedIndex="0"
                                      ShowSelectedItem="True" AutoSelect="True"
IsSelectable="False" Xaml:C1NagScreen.Nag="True">
                        <Xaml:C1RadialColorItem ToolTip="Blue" Brush="Blue"
GroupName="Blue"/>
                        <Xaml:C1RadialColorItem ToolTip="Slate Blue"
Brush="MediumSlateBlue" GroupName="Blue"/>
                        <Xaml:C1RadialColorItem ToolTip="Turquoise" Brush="Turquoise"
GroupName="Blue"/>
                        <Xaml:C1RadialColorItem ToolTip="Aqua" Brush="Aqua"
GroupName="Blue"/>
                        <Xaml:C1RadialColorItem ToolTip="Sky Blue" Brush="SkyBlue"
GroupName="Blue"/>
                        <Xaml:C1RadialColorItem ToolTip="Purple" Brush="#FFAC38AC"
AccentBrush="#FF801C80" GroupName="Blue"/>
                        <Xaml:C1RadialColorItem ToolTip="Dark Purple" Brush="Purple"
GroupName="Blue"/>
                        <Xaml:C1RadialColorItem ToolTip="Dark Blue" Brush="DarkBlue"
GroupName="Blue"/>
                    </Xaml:C1RadialColorItem>
```

9. Right-click the page and select **View Code** from the list.
10. Add the following using statement to the top of the page:

C#

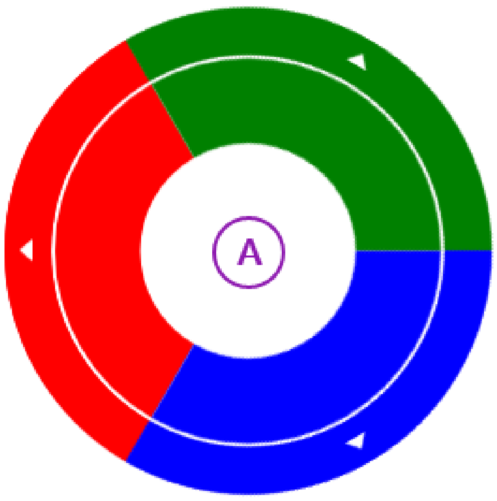
```
 using C1.Xaml;
```

11. Add the following **ItemClick** event to the page. This will allow the text you added to your main page to change color when you select one of the colors from the **C1RadialMenu**:

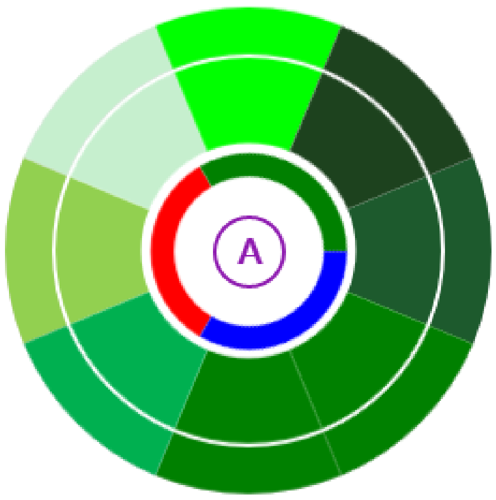
C#

```
private void contextMenu_ItemClick(object sender, SourcedEventArgs e)
        {
            C1RadialMenuItem item = e.Source as C1RadialMenuItem;
            if (item is C1RadialColorItem)
            {
                this.text.Foreground = ((C1RadialColorItem)item).Brush;
                txt.Text = "Item Clicked: " +
((C1RadialColorItem)item).Color.ToString();
            }
            else
            {
                txt.Text = "Item Clicked: " + (item.Header ?? item.Name).ToString();
            }
        }
```

12. Press F5 or start debugging to run your application. When you right-click or right-tap the application and open the **C1RadialMenu**, it should resemble the following image:



13. When you select the green submenu, your **C1RadialMenu** will resemble the following image:



Note that you can return to the main menu by selecting one of the interior colors.

14. Select a color. Note that the text on your application page has changed color.

Creating a Numeric Radial Menu

The C1RadialMenu control allows you to create a radial numeric slider menu. This is especially useful for applications where you might want users to select a font size for an application. You'll use both XAML markup and code to create the application.

1. Locate the <Grid> </Grid> tags on your page and replace them with the following markup to create the framework for your application:

Markup

```
<Page.Resources>
        <Style TargetType="TextBlock" x:Key="TextIconStyle">
            <Setter Property="Margin" Value="-10" />
            <Setter Property="FontSize" Value="20" />
            <Setter Property="FontFamily" Value="Segoe UI Symbol" />
            <Setter Property="FontWeight" Value="Normal" />
            <Setter Property="Foreground" Value="#333333" />
            <Setter Property="HorizontalAlignment" Value="Center" />
            <Setter Property="VerticalAlignment" Value="Center" />
        </Style>
        <Style TargetType="Image" x:Key="MenuIcon">
            <Setter Property="Width" Value="16"/>
            <Setter Property="Height" Value="16"/>
            <Setter Property="Margin" Value="0"/>
        </Style>
    </Page.Resources>

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <Xaml:C1ContextMenuService.ContextMenu>

        </Xaml:C1ContextMenuService.ContextMenu>

        <Xaml:C1ListViewer x:Name="text" Foreground="Sienna"
HorizontalAlignment="Stretch" VerticalAlignment="Center" Height="75"
ZoomMode="Disabled" FontSize="16" Xaml:C1NagScreen.Nag="True">
            <TextBlock Text="Touch: hold down for a few seconds until the
indicator displays.&#10;Keyboard: press the context-menu button over this
text.&#10;Mouse: right-click over this text."
                           TextWrapping="Wrap" />
        </Xaml:C1ListViewer>

        <TextBlock x:Name="txt" Foreground="Red" Text="" FontSize="16"
VerticalAlignment="Bottom" HorizontalAlignment="Center" Margin="10" />

    </Grid>
```

2. Locate the <Xaml:C1ContextMenuService.ContextMenu> </Xaml:C1ContextMenuService.ContextMenu> tags within the markup you just added. Place your cursor between the tags.
3. Locate the **C1RadialMenu** control in the Visual Studio ToolBox and double-click it to add it to your application.
4. Edit the opening <C1RadialMenu> tag so that it resembles the following:

Markup

```
<Xaml:C1RadialMenu Xaml:C1NagScreen.Nag="True" x:Name="contextMenu" Offset="-
130,0" Opened="contextMenu_Opened" AccentBrush="ForestGreen"
                                 ItemClick="contextMenu_ItemClick"
ItemOpened="contextMenu_ItemOpened" BorderBrush="#FFC6DEC4">
```

5. Add the following markup between the <C1RadialMenu> </C1RadialMenu> tags to add one **C1RadialNumericItem**s to your application. Note that you are also adding several subitems and a custom item icon:

Markup

```
<Xaml:C1RadialNumericItem Header="Font Size" Minimum="9" Maximum="72"
MarkStartAngle="-128" MarkEndAngle="231" x:Name="fontSize" Value="11">
      <Xaml:C1RadialNumericItem.Icon>
        <TextBlock Style="{StaticResource TextIconStyle}" FontFamily="Segoe UI"
FontSize="18">
           <Run Text="A"/>
           <Run Text="{Binding Value, ElementName=fontSize}"
Typography.Variants="Superscript"/>
        </TextBlock>
     </Xaml:C1RadialNumericItem.Icon>
    <x:Double>9</x:Double>
    <x:Double>11</x:Double>
    <x:Double>13</x:Double>
    <x:Double>16</x:Double>
    <x:Double>20</x:Double>
    <x:Double>36</x:Double>
    <x:Double>72</x:Double>
</Xaml:C1RadialNumericItem>
```

6. Right-click the page and select **View Code** from the list.

7. Add the following using statement to the top of the page:

C#

```
using C1.Xaml;
```

8. Add the following **ItemClick** event to the page. This will allow the size of the text in the TextBox control to change size depending on the item selected:

C#

```
private void contextMenu_ItemClick(object sender, SourcedEventArgs e)
        {
            C1RadialMenuItem item = e.Source as C1RadialMenuItem;

if (item is C1RadialNumericItem)
            {
                txt.FontSize = ((C1RadialNumericItem)item).Value;
                txt.Text = "Item Clicked: " +
((C1RadialNumericItem)item).Value.ToString();
```

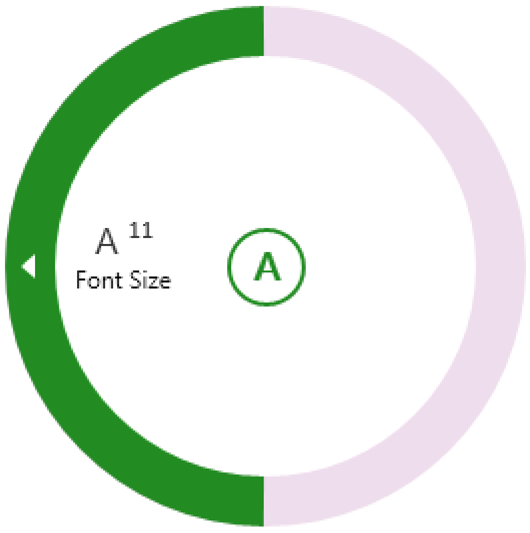
```
            }
            else
            {
                txt.Text = "Item Clicked: " + (item.Header ??
item.Name).ToString();
            }
        }
```

9. Then, add two more events. This code will control the ItemOpened and Opened events:

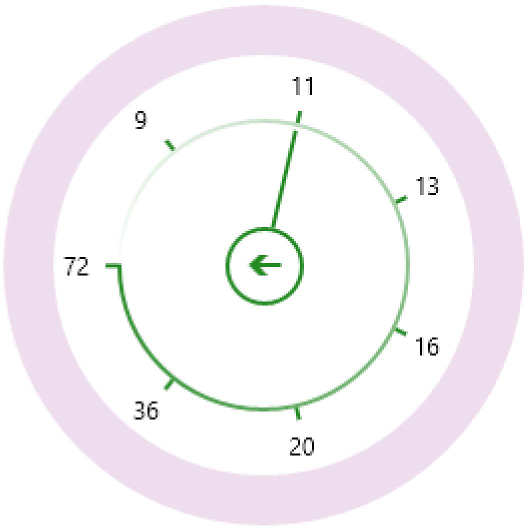
C#

```
private void contextMenu_ItemOpened(object sender, SourcedEventArgs e)
        {
            C1RadialMenuItem item = e.Source as C1RadialMenuItem;
            txt.Text = "Item Opened: " + (item.Header ?? item.Name).ToString();
        }
        private void contextMenu_Opened(object sender, EventArgs e)
        {
            // expand menu immediately, as in this sample we don't have
underlying editable controls
            contextMenu.Expand();
        }
```

10. Press F5 or start debugging to run your application. Your **C1RadialMenu** control should resemble the following:



11. When you click on the Font Size option, the **C1NumericRadialItem**s will be displayed:



Working with Checkable Radial Menu Items

In the following topics, you will learn how to create standalone and mutually exclusive checkable radial menu items.

Creating a Checkable C1RadialMenuItem

In this topic, you will create a checkable C1RadialMenuItem that can be selected or cleared by a user. In order to complete this topic, you must have a C1RadialMenu control that holds at least one item or a C1RadialMenu control with at least one submenu.

In XAML

Complete the following steps:

1. Locate the `<Xaml:C1RadialMenuItem>` tag for the **C1RadialMenuItem** you wish to make checkable and then add `IsCheckable="True"` to the tag so that the XAML resembles the following:

   Markup
   ```
   <Xaml:C1RadialMenuItem Header="C1RadialMenuItem" IsCheckable="True"/>
   ```

2. Run the project.

In Code

Complete the following steps:

1. In Source view, locate the `<Xaml:C1RadialMenuItem>` tag for the item you wish to make checkable and add `Name="CheckableRadialMenuItem"` to it. This will give the item a unique identifier that you can use in code.

2. Enter Code View and add the following code beneath the **InitializeComponent()** method:

   Visual Basic
   ```
   CheckableRadialMenuItem.IsCheckable = True
   ```

C#

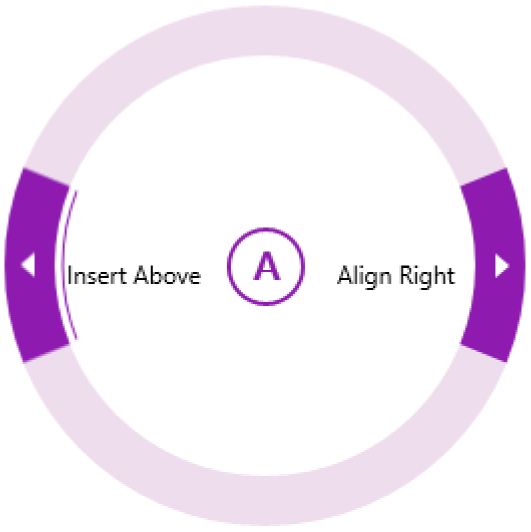
```
CheckableRadialMenuItem.IsCheckable = true;
```

3. Run the program

This Topic Illustrates the Following:

Once the program is run, open the **C1RadialMenu**. To add a check mark, click the **C1RadialMenuItem**.

The graphic below illustrates a checkable **C1RadialMenuItem**.



Creating Mutually Exclusive Checkable Radial Menu Items

In this topic, you learn how to create a list of checkable C1RadialMenuItems that are grouped together so that only one item can be checked at a time.

In XAML

Complete the following steps:

1. Add `IsCheckable="True"` and `GroupName="CheckableGroup"` to the `<c1:C1RadialMenuItem>` tag of each C1RadialMenuItem you wish to add to the group of mutually exclusive checkable items.
2. Run the program and click the first item in the group. Observe that the **C1RadialMenuItem** is highlighted. Now click the second item in the group and observe that the highlight is removed from the first item and then added to the second item.

In Code

1. Complete the following steps:
2. Set the **Name** property of each **C1RadialMenuItem** you wish to add to the group of mutually exclusive checkable items.
3. Open the **MainPage.xaml.cs** page.
4. Set the IsCheckable and GroupName property of each **C1RadialMenuItem**, replacing "ItemName" with the value of the **C1RadialMenuItem**'s **Name** property.

Visual Basic

```
ItemName.IsCheckable = True
```

```
C#
ItemName.IsCheckable = true;
```

4. Run the program and tap the first item in the group.

- Observe that a check that resembles a thinner highlight is added to the **C1RadialMenuItem**.
- Now tap the second item in the group and observe that the check is removed from the first item and then added to the second item.

Customizing the C1RadialMenu's Appearance

You can quickly and easily customize the C1RadialMenu's appearance using just a few properties.

Complete the following steps:

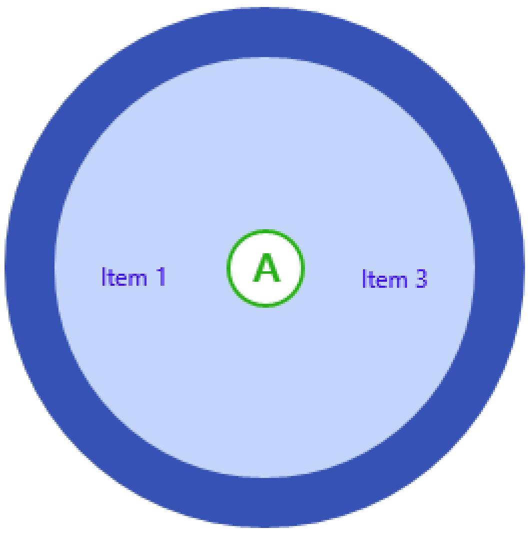
1. Add the following properties to the <Xaml: C1RadialMenu> tag:

- AccentBrush="#FF28B01A"
- Background="#FFC3D5FB"
- BorderBrush="#FF3652B4"
- Foreground="#FF4210EE"

2. Run your application.

This Topic Illustrates the Following:

When you run your application, it should resemble the following image:



Enabling Automatic Menu Collapsing

Automatic menu collapsing allows users to collapse the C1RadialMenu by clicking outside of the menu area. To enable automatic menu collapsing, set the AutoCollapse property to **True**.

In XAML

Complete the following steps:

1. Add `AutoCollapse="True"` to the `<Xaml:C1RadialMenu>` tag.
2. Run the program.

In Code

Complete the following steps:

1. In Source view locate the `<Xaml:C1RadialMenuItem>` tag for the item you wish to make checkable and add `Name="C1RadialMenu1"` to it. This will give the item a unique identifier that you can use in code.
2. Enter Code view and add the following code beneath the **InitializeComponent()** method:

Visual Basic

```
C1RadialMenu1.AutoCollapse = True
```

C#

```
C1RadialMenu1.AutoCollapse = true;
```

3. Run the program.

This Topic Illustrates the Following:

After you've run the project, right tap the page to view the Navigation Button. Tap the Navigation Button to open the **C1RadialMenu**. With the radial menu open, click outside of the menu and observe that the **C1RadialMenu** closes.

Adding a Separator Between Radial Menu Items

In this topic, you will learn how to add separators between radial menu items. The separator appears as a blank sector between two C1RadialMenuItems. There are two ways to include a separator between C1RadialMenuItems, either by placing an empty C1RadialMenuItem between the existing menu items, or by setting the **C1RadialMenu** SectorCount property and the DisplayIndex of each C1RadialMenuItem.

In XAML

Complete one of the following:

- **Inserting a C1RadialMenuItem**

  To add a separator by placing an empty **C1RadialMenuItem** between other menu items, place `<Xaml:C1RadialMenuItem />` between two `<Xaml:C1RadialMenuItem>` tags:

Markup

```
<Xaml:C1RadialMenuSectorCount="8">
        <Xaml:C1RadialMenuItemHeader="Item 1"/>
        <Xaml: C1RadialMenuItem/>
        <Xaml:C1RadialMenuItemHeader="Item 2" />
        <Xaml: C1RadialMenuItem/>
        <Xaml:C1RadialMenuItemHeader="Item 3" />
      </Xaml:C1RadialMenu>
```

- **Setting the SectorCount and the DisplayIndex**

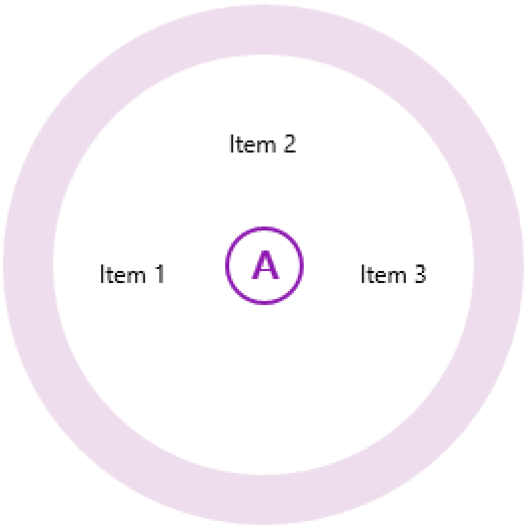
  If you want to add separators to your application by setting the **C1RadialMenu** SectorCount and the DisplayIndex of each **C1RadialMenuItem**, your XAML markup will resemble the following:

Markup

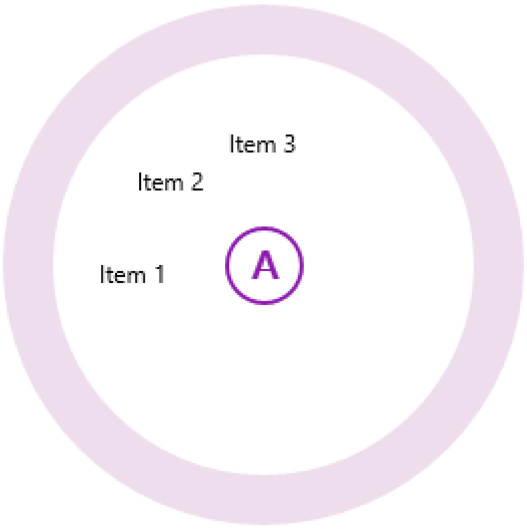
```
<Xaml:C1RadialMenu SectorCount="8" >
     <Xaml:C1RadialMenuItem Header="Item 1" />
     <Xaml:C1RadialMenuItem Name="C1RadialMenuItem1" Header="Item 2"
                            DisplayIndex="2" />
     <Xaml:C1RadialMenuItem Header="Item 3" DisplayIndex="4" />
</Xaml:C1RadialMenu>
```

This Topic Illustrates the Following:

Either markup sample will result in the following image:



For contrast, the same markup with no separators will result in a **C1RadialMenu** that resembles the following image:



Adding an Icon to a Radial Menu Item

In this step, you will learn how to add an icon to a C1RadialMenuItem|tag=C1RadialMenuItem_Class.

In XAML

Complete the following steps:

1. Add an icon image to your Universal Windows project. A 12x12 pixel image is best.
2. Add the following XAML markup between the `<Xaml:C1RadialMenuItem>` and `</Xaml:C1RadialMenuItem>` tags, replacing the value of the Source property with your image's name:

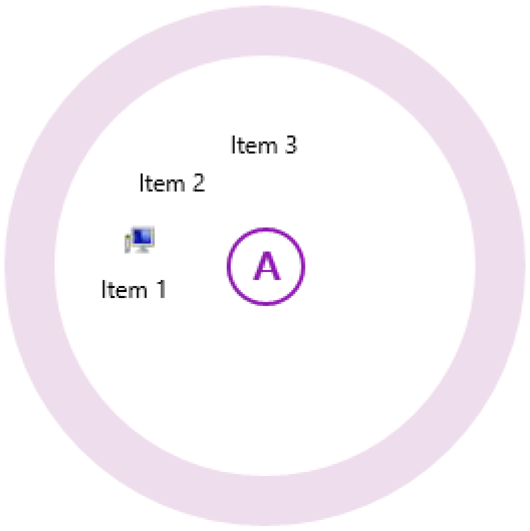
Markup

```
<Xaml:C1RadialMenuItem.Icon>
     <Image Source="YourImage.png" Height="12" Width="12" Margin="5,0,0,0"/>
</Xaml:C1RadialMenuItem.Icon>
```

3. Run the project.

This Topic Illustrates the Following:

The following image depicts a **C1RadialMenuItem** with a 12x12 pixel icon.



RangeSlider for UWP

Add smooth numeric data selection to your UWP applications. **RangeSlider for UWP** extends the basic slider control and provides two thumb elements instead of one, allowing users to select ranges instead of single values.

Make the most of **RangeSlider for UWP** by taking advantage of the following key features:

- **Horizontal or Vertical Orientation**

  Change the orientation with one simple property. Create vertical or horizontal range sliders.

- **Set Min and Max Values**

  Control the minimum and maximum values of the range slider.

- **Customizable Thumbs**

Customize the thumbs of C1RangeSlider to create custom zooming controls.

RangeSlider for UWP Quick Start

Step 1 of 4: Setting Up the Application

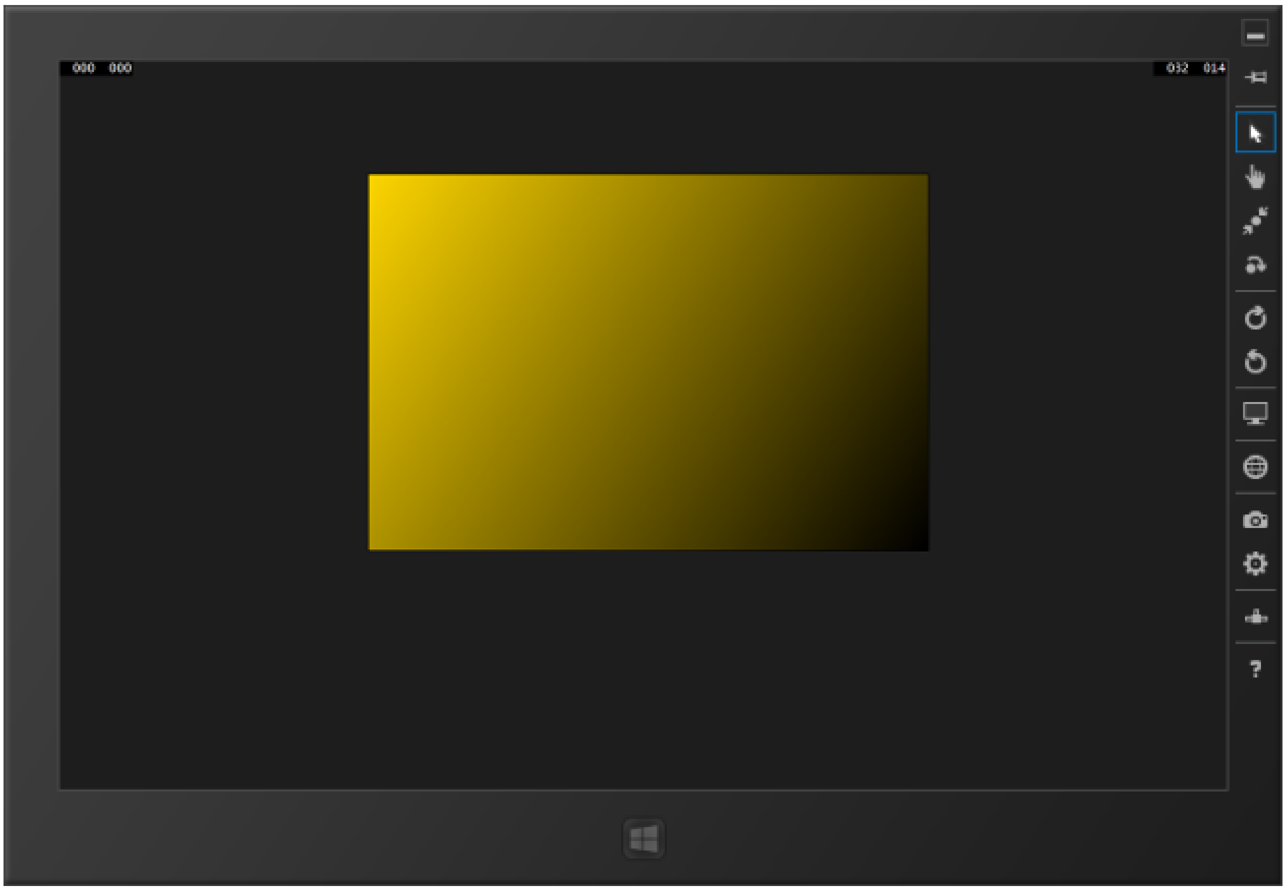
In this step, you will create a new Universal Windows application using **RangeSlider for UWP**.

1. In Visual Studio, select **File | New Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal.
3. From the templates list, select **Blank App (Universal Windows)** and give your project a **Name**, and select **OK**.
4. Add the following assembly to your application by right-clicking the **References** folder and selecting **Add Reference**:
    - C1.UWP.dll
5. In **Design** view, click once within the **Grid** in your application.
6. Navigate to the **Toolbox** and double-click the **Rectangle** icon to add the standard control to the **Grid**.
7. In the **Design** pane, move and resize **rectangle1** to fill the center of the Grid.
8. Switch to XAML view and add a **Fill** to the **<Rectangle>** tag so it appears similar to the following:

XAML

```
<Rectangle Name="rectangle1">
            <Rectangle.Fill>
                <LinearGradientBrush x:Name="colors">
                    <GradientStop x:Name="goldcol" Color="Gold" Offset="0" />
                    <GradientStop x:Name="blackcol" Color="Black" Offset="1" />
                </LinearGradientBrush>
            </Rectangle.Fill>
</Rectangle>
```

9. Run your application now and observe that it looks similar to the following:



You've successfully created a Universal Windows application and customized the Rectangle control. In the next step you'll add and customize the C1RangeSlider control.

Step 2 of 4: Adding a C1RangeSlider Control

In the previous step you created a new Universal Windows project and added a Rectangle control with a gradient to the application. In this step you'll continue by adding a C1RangeSlider control that will control the gradient fill in the **Rectangle**.

Complete the following steps:

1. In the XAML window of the project, place the cursor between the **</Rectangle>** and **</Grid>** tags and click once.
2. Navigate to the **Toolbox** and double-click the **C1RangeSlider** icon to add the control to the application on top of the **Rectangle**.
3. Give your control a name by adding **x:Name="c1rs1"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

   XAML
   ```
   <Xaml:C1RangeSlider x:Name="c1rs1" />
   ```

   By giving it a unique identifier, you'll be able to access the control in code.

4. Add a margin by adding **Margin="50"** to the **<c1:C1RangeSlider>** tag so that it appears similar to the following:

XAML
```
<Xaml:C1RangeSlider x:Name="c1rs1" Margin="50">
```

This will set each edge the same distance away from the grid's border.

5. Set the Orientation property to Vertical by adding **Orientation="Vertical"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

   XAML
   ```
   <Xaml:C1RangeSlider x:Name="c1rs1" Margin="50" Orientation="Vertical">
   ```

   By default **Orientation** is Horizontal and the control appears across the page.

6. Set the UpperValue property to 1 by adding **UpperValue="1"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

   XAML
   ```
   <Xaml:C1RangeSlider x:Name="c1rs1" Margin="50" Orientation="Vertical"
   UpperValue="1">
   ```

   The upper thumb will now begin at 1.

7. Set the Maximum property to 1 by adding **Maximum="1"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

   XAML
   ```
   <Xaml:C1RangeSlider x:Name="c1rs1" Margin="50" Orientation="Vertical"
   UpperValue="1" Maximum="1">
   ```

   Users will now not be able to select a value greater than 1.

8. Set the ValueChange property to 0.1 by adding **ValueChange="0.1"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

   XAML
   ```
   <Xaml:C1RangeSlider x:Name="c1rs1" Margin="50" Orientation="Vertical"
   UpperValue="1" Maximum="1" ValueChange="0.1">
   ```

   When you click on the slider track at run time, the slider thumb will now move by 0.1 units.

9. Set the **Opacity** property to "0.8" by adding **Opacity="0.8"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

   XAML
   ```
   <Xaml:C1RangeSlider x:Name="c1rs1" Margin="50" Orientation="Vertical"
   UpperValue="1" Maximum="1" ValueChange="0.1" Opacity="0.8">
   ```

   By default this property is set to 1 and the control appears completely opaque. Changing this to a lower number will make the control appear slightly transparent.

10. Indicate event handlers by adding **LowerValueChanged="c1rs1_LowerValueChanged" UpperValueChanged="c1rs1_UpperValueChanged"** to the **<Xaml:C1RangeSlider>** tag so that it appears

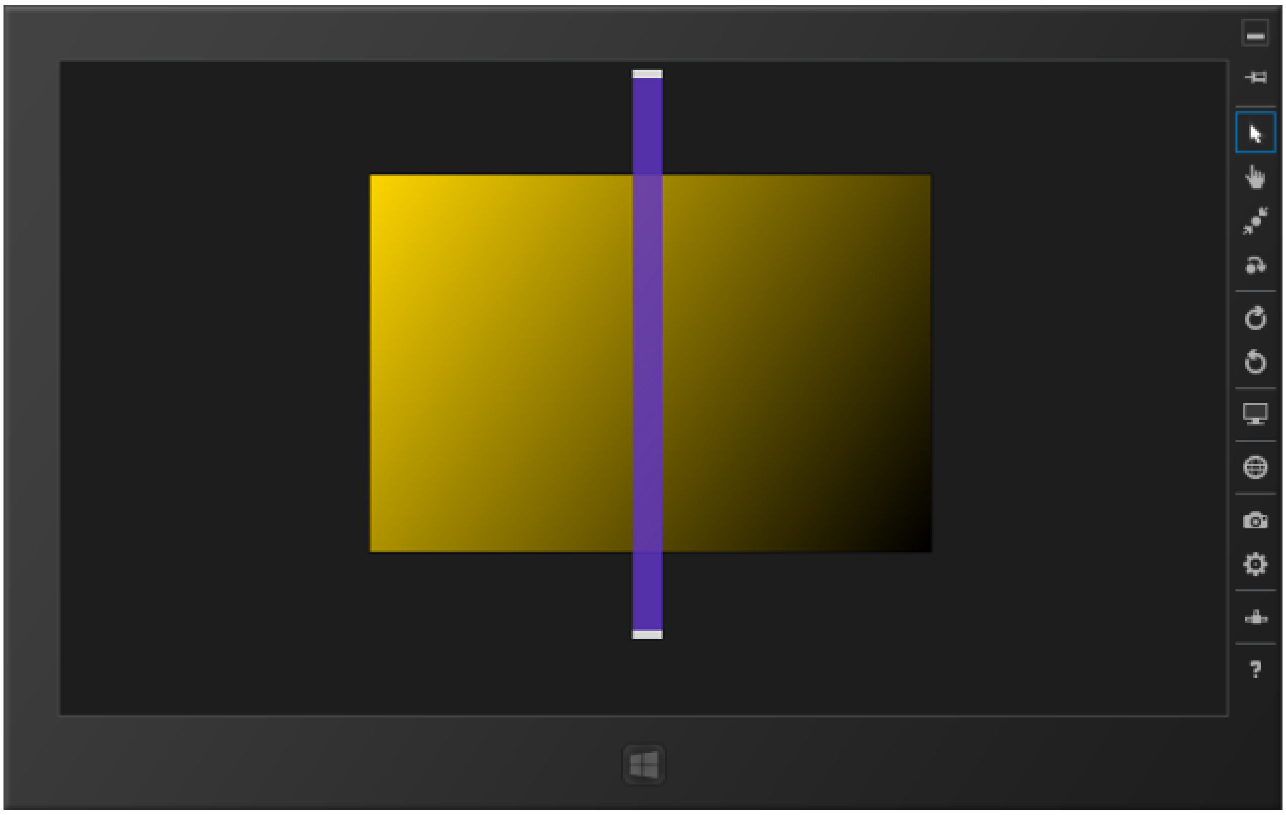
similar to the following:

XAML

```
<Xaml:C1RangeSlider x:Name="c1rs1" HorizontalAlignment="Left"
Margin="671,10,0,90" Orientation="Vertical" UpperValue="1" Maximum="1"
ValueChange="0.1" Opacity="0.8" LowerValueChanged="c1rs1_LowerValueChanged"
UpperValueChanged="c1rs1_UpperValueChanged" />
```

You'll add code for these event handlers in a later step.

11. Run your application now and observe that it looks similar to the following:



You've successfully set up your application's user interface, but right now the slider will do nothing if you move it. In the next step you'll add code to your application to add functionality.

Step 3 of 4: Adding Code to the Application

In the previous steps you set up the application's user interface and added controls to your application. In this step you'll add code to your application to finalize it.

Complete the following steps:

1. Select **View | Code** to switch to **Code** view.
2. In **Code** view, add the following import statement to the top of the page:

C#

```
using C1.Xaml;
```

3. Create a **MainPage_Loaded** event handler and add it below the page constructor:

C#

```
private void MainPage_Loaded(object sender, RoutedEventArgs e)
        {
            UpdateGradient();
        }
```

4. Add the following code just after the **MainPage_Loaded** event handler to update the gradient values:

C#

```
private void UpdateGradient()
        {
            if (c1rs1 != null)
            {
                this.goldcol.Offset = this.c1rs1.LowerValue;
                this.blackcol.Offset = this.c1rs1.UpperValue;
            }
        }
```

5. Add code to the **c1rs1_LowerValueChanged** event handler so that it appears like the following:

C#

```
private void c1rs1_LowerValueChanged(object sender, EventArgs e)
        {
            UpdateGradient();
        }
```

6. Add code to the **c1rs1_UpperValueChanged** event handler so that it appears like the following:

C#

```
c1rs1_UpperValueChanged(object sender, EventArgs e)
{
    UpdateGradient();
}
```

In this step you completed adding code to your application. In the next step you'll run the application and observe run-time interactions.

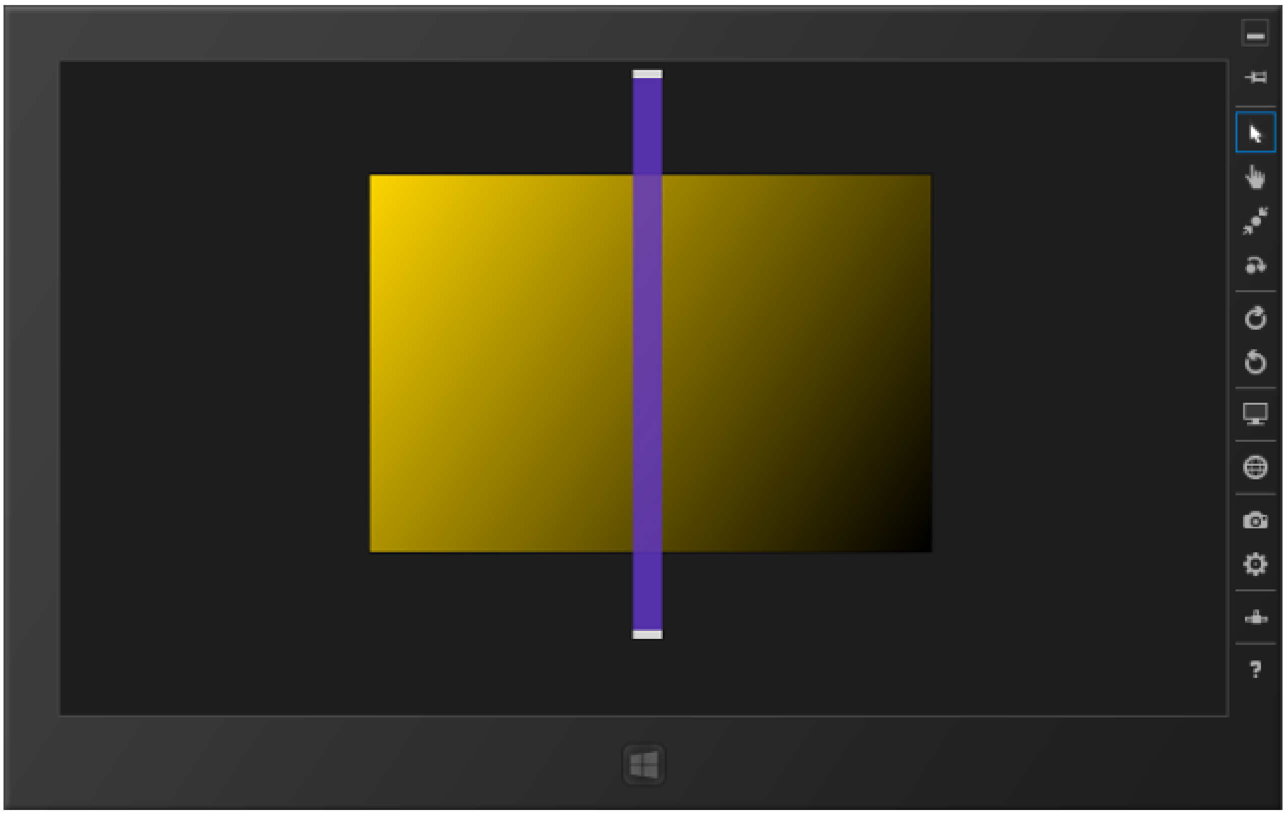
Step 4 of 4: Running the Application

Now that you've created a UWP application and customized the application's appearance and behavior, the only thing left to do is run your application.

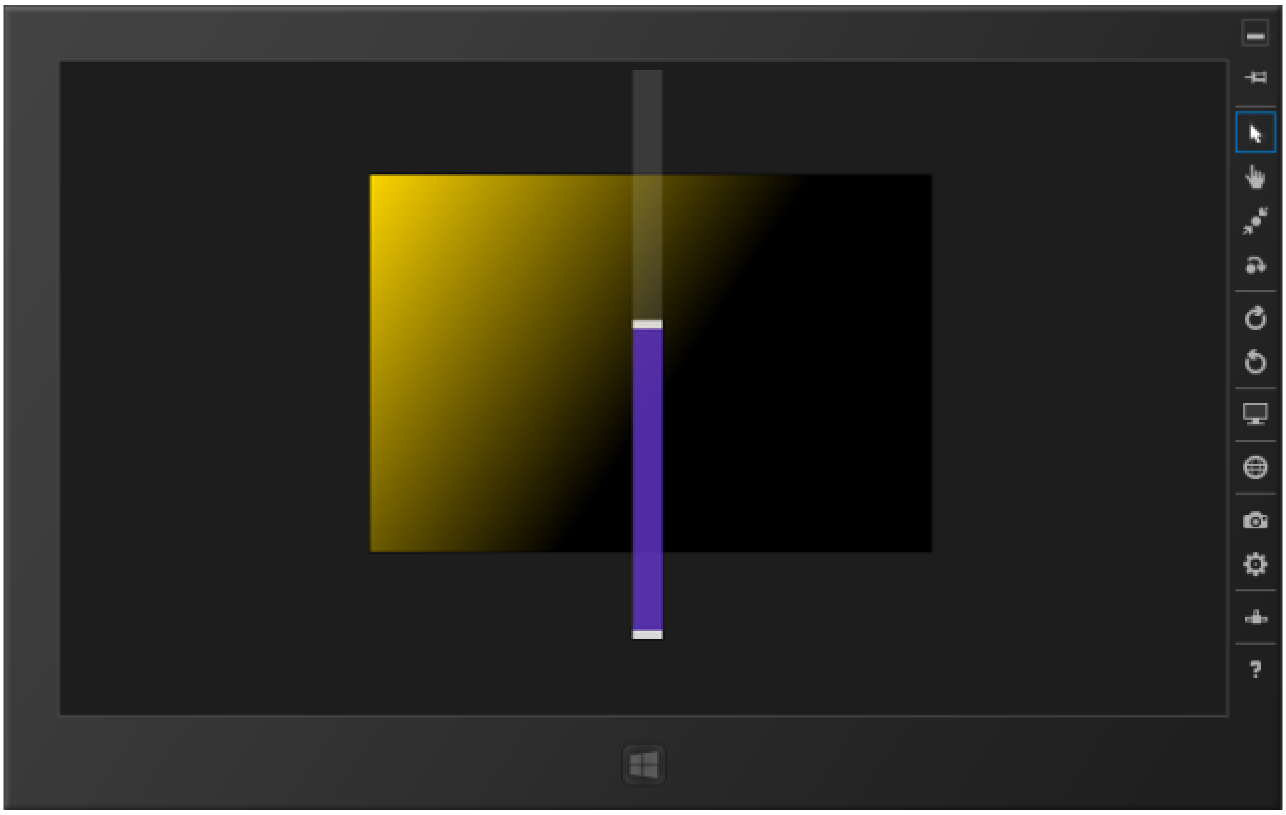
To run your application and observe **RangeSlider for UWP**'s run-time behavior, complete the following steps:

1. Select **Debug | Start Debugging** from the menu, or press **F5**.

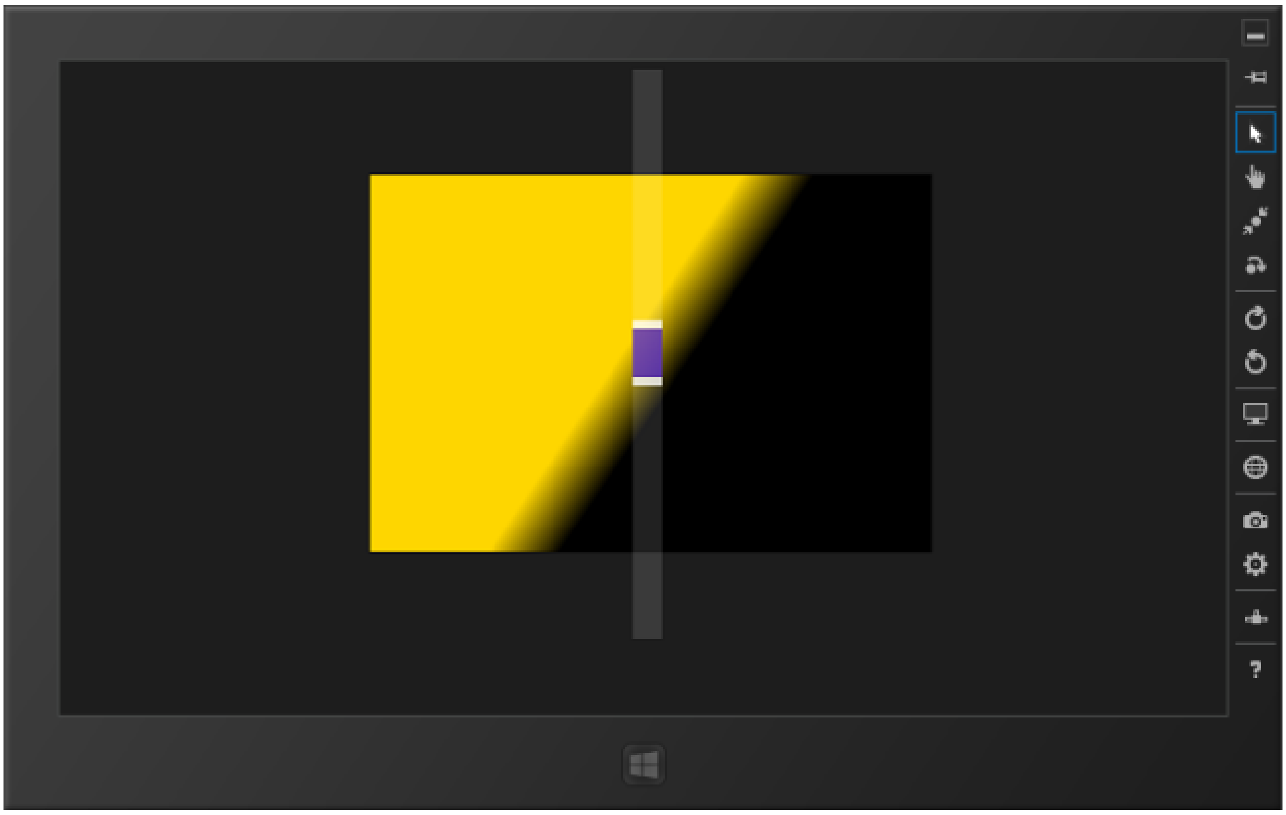
   The application will appear similar to the following:



2. Move the top slider thumb down. Notice that the gradient's appearance changes:



3. Move the bottom thumb up, notice that the gradient effect appears less diffused:

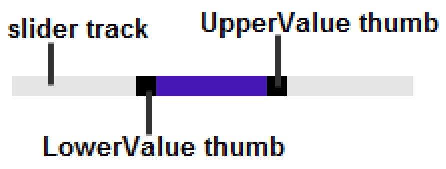


Congratulations!

You've completed the **RangeSlider for UWP** quick start and created a **RangeSlider for UWP** application, customized the appearance and behavior of the controls, and viewed some of the run-time capabilities of your application.

RangeSlider Elements

RangeSlider for UWP includes the C1RangeSlider control, a simple input control that moves beyond the typical slider and includes two thumbs for selecting a range of values. When you add the **C1RangeSlider** control to a XAML window, it exists as a completely functional slider control which you can further customize. The control's interface looks similar to the following image:



RangeSlider Features

Minimum and Maximum

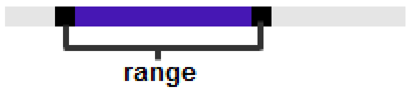
The Minimum and Maximum properties set the possible range of values allowable in the C1RangeSlider control. There are some restrictions on setting values on the **RangeSlider** control:

- The **LowerValue** thumb, the thumb with the smaller number, cannot be set to a value lower than the set **Minimum**.
- The **UpperValue** thumb, the thumb with the higher number, cannot be set to a value higher than the set **Maximum**.

By default, the **Minimum** property is set to 0, and the **Maximum** property is set to 100.

Thumb Values and Range

The **C1RangeSlider**'s value range is determined by the difference between the UpperValue property and the LowerValue property. These two properties are connected to the thumbs used to select the range of values:



Setting the Thumb Values

The C1RangeSlider control includes two thumbs for selecting a range of values. The **UpperValue** and the **LowerValue** thumbs move along the slider track. By default, the **UpperValue** property is set to 100 and the **LowerValue** property is set to 0.

Using one of the following methods, you can customize the thumb values.

In XAML

To set the **UpperValue** and **LowerValue** properties add **UpperValue="90" LowerValue="10"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

XAML
```
<Xaml:C1RangeSlider Name="C1RangeSlider1" Width="26" Height="18" UpperValue="90"
LowerValue="10" />
```

In Code

For example, to set the **UpperValue** and **LowerValue** properties add the following code to your project:

Visual Basic
```
Me.C1RangeSlider1.LowerValue = 10
Me.C1RangeSlider1.UpperValue = 90
```

C#
```
this.c1RangeSlider1.LowerValue = 10;
this.c1RangeSlider1.UpperValue = 90;
```

At Design Time

To set the **UpperValue** and **LowerValue** properties at Design time, complete the following steps:

1. Click the **C1RangeSlider** control once to select it.
2. Navigate to the **Properties** window, and enter a number, for example "10", in the text box next to the **LowerValue** property.
3. Still in the **Properties** window, enter a number, for example "90", in the text box next to the **UpperValue** property.

This will set the **UpperValue** and **LowerValue** properties to the values you chose.

Setting the ValueChange Property

The ValueChange property determines by what value the **UpperValue** and the **LowerValue** thumbs move along the slider track when the track is clicked, note that if the tack is clicked between the **UpperValue** and **LowerValue** thumbs (in the range) the thumbs will not move.

The **UpperValue** property cannot be less than the Maximum property and the **LowerValue** cannot be less than the Minimum property.

Using any of the following methods, when you click the track of the **C1RangeSlider** control, the closest thumb will move by 5 units.

In XAML

To set the **ValueChange** property add **ValueChange="5"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

XAML

```
<Xaml:C1RangeSlider Name="C1RangeSlider1" Height="18" Width="26" ValueChange="5" />
```

In Code

To set the **ValueChange** property add the following code to your project:

Visual Basic

```
Me.C1RangeSlider1.ValueChange = 5
```

C#

```
this.c1RangeSlider1.ValueChange = 5;
```

At Design Time

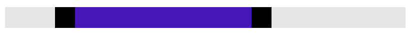
To set the **ValueChange** property at Design time, complete the following steps:

1. Click the **C1RangeSlider** control once to select it.
2. Navigate to the **Properties** window and enter a number, for example "5", in the text box next to the **ValueChange** property.

This will set the ValueChange property to the value you chose.

RangeSlider Orientation

C1RangeSlider includes the ability to orient the control either horizontally or vertically using the Orientation property. By default the control initially appears with a horizontal orientation when added to the application, as in the following image:



You can easily change the orientation from the Properties window, in XAML, and in code using the **Orientation** property:

In XAML

To set the **Orientation** property to Vertical add **Orientation="Vertical"** to the **<Xaml:C1RangeSlider>** tag so that it appears similar to the following:

XAML
```
<Xaml:C1RangeSlider Name="C1RangeSlider1" Width="26" Orientation="Vertical" />
```

In Code

To set the **Orientation** property to **Vertical**, add the following code to your project:

Visual Basic
```
Me.C1RangeSlider1.Orientation = Orientation.Vertical
```

C#
```
this.c1RangeSlider1.Orientation = Orientation.Vertical;
```

At Design Time

To set the **Orientation** property to **Vertical** at Design time, complete the following steps:

1. Click the **C1RangeSlider** control once to select it.
2. Navigate to the Properties window and locate the **Orientation** property.
3. Click the drop-down arrow next to the **Orientation** property and choose **Vertical**.

This will change the **Orientation** property so that the control appears vertically.

Run the application and observe:

The **C1RangeSlider** control will be shown vertically:

TabControl for UWP

Organize and navigate content as tabs with **TabControl for UWP**. Tabs help utilize available space while letting the user see all available items to select. Tabs can be positioned to the top, bottom, left, or right of a page and support several different shapes and built-in features.

TabControl for UWP Key Features

TabControl for UWP allows you to create customized, rich applications. Make the most of **TabControl for UWP** by taking advantage of the following key features:

- **Modern Tab Styles**

  You can modify the shape of the tab headers using one of the four built-in shapes: Rounded, Rectangle, Ribbon and Sloped. Or use no shape outline for a clean, modern look.

- **Position Tabs to Any Edge**

  Tabs can be positioned to the top, bottom, left or right. Just set the TabStripPlacement property.

- **Overlap Tabs**

  The overlap between tab items headers can be customized to show jagged tabs, for example (like the Documents tab in Microsoft Visual Studio). Just set the TabStripOverlap property. Define if the tab items are overlapped with the right-most in the back or the left-most in the back using the TabStripOverlapDirection property. The selected item is always on top.

- **Closeable Tabs**

  Control whether the user can close tabs and where to show the close button. Display the close button inside each tab item or in a global location outside the tab strip, just like Visual Studio does in its Documents tab.

- **Menu Tabs**

  You can show all the items in a menu (like the Documents tab in Visual Studio). This is useful when the items don't fit in the available space, so the end-user can quickly access all the elements.

- **Scrollable Elements**

  If the tab items cannot fit in the current available space, the **C1TabControl** shows Next/Previous buttons just like Microsoft Internet Explorer.

TabControl for UWP Quick Start

The following quick start guide is intended to get you up and running with **TabControl for UWP**. In this quick start, you'll create a new project with the C1TabControl control. You will also customize the C1TabControl control, add tabs pages filled with content, and then observe some of the run-time features of the control.

Step 1 of 3: Creating a C1TabControl Application

In this step, you'll create a Universal Windows application using **TabControl for UWP**.

Complete the following steps:

1. In Visual Studio select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**.
3. Right-click the project name in the Solution Explorer and select **Add Reference**.
4. In the **Reference Manager** dialog box, expand Universal Windows and select Extensions; you should see the UWP assemblies in the center pane. Select **C1.UWP** and click OK.
5. Open **MainPage.xaml** if it isn't already open, and add the following markup within the <Page> tag:

   Markup

   ```
   xmlns:c1="using:C1.Xaml"
   ```

   This adds required references to the project.

You have completed the first step of the **TabControl for UWP** quick start. In this step, you created a Universal Windows project. In the next step, you will add tabs and tab pages to the control.

Step 2 of 3: Adding C1TabControl to the Application

In the last step, you created a Universal Windows project. In this step, you'll add a C1TabControl to the application.

Complete the following steps:

1. Place the cursor between the <Grid> and </Grid> tags in **MainPage.xaml**, and click once.
2. Add the following markup within the <Grid> and </Grid> tags:

   Markup

   ```
   <c1:C1TabControl x:Name="tabControl" TabStripOverlapDirection="Left"
   TabStripOverlap="8" TabItemShape="Sloped" TabItemClose="InEachTab"
   TabStripPlacement="Bottom" TabStripMenuVisibility="Visible">
   </c1:C1TabControl>
   ```

This markup adds a **C1TabControl** to the project. This sets the following properties:

- The TabStripOverlapDirection property is set to **Left** and TabStripOverlap is set to **8** indicating the overlapping allowed between the tabs.

- The TabItemClose property is set to **InEachTab**. This will add close buttons to each tab.
- The TabItemShape property is set to Sloped. This will change the shape of the tab items so that they resemble tabs on an office folder.
- The TabStripMenuVisibility property is set to **Visible**.
- The TabStripPlacement property is set to **Bottom**. This will place the tabstrip at the bottom of the control.

  You'll also need to add **C1TabItem**s within the **C1TabControl**.

3. Add the following markup within the <c1:C1TabControl> and </c1:C1TabControl> tags:

Markup

```
<c1:C1TabItem Header="Notes" CanUserPin="True">
    <RichEditBox HorizontalAlignment="Left" Height="680" VerticalAlignment="Top"
Width="1330"/>
</c1:C1TabItem>
<c1:C1TabItem Header="Tasks">
    <RichEditBox HorizontalAlignment="Left" Height="680" VerticalAlignment="Top"
Width="1330"/>
</c1:C1TabItem>
<c1:C1TabItem Header="Reminders">
    <RichEditBox HorizontalAlignment="Left" Height="680" VerticalAlignment="Top"
Width="1330"/>
</c1:C1TabItem>
<c1:C1TabItem Header="Topics" CanUserPin="True">
    <RichEditBox HorizontalAlignment="Left" Height="680" VerticalAlignment="Top"
Width="1330"/>
</c1:C1TabItem>
```

   This adds C1TabItems, each containing a **RichEditBox**. Note that in two of the **C1TabItem**s, **CanUserPin** is set to **True** allowing the tabs to be pinned at run time.

In this step, you added and customized the C1TabControl control. In the next step, you will run the program and observe what you've accomplished during this quick start.

Step 3 of 3: Running the Project

In the previous steps, you created a project with a C1TabControl control, added tab pages to the control, and modified the control's appearance and behaviors. In this step, you will run the program and observe all of the changes you made to the **C1TabControl** control.

Complete the following steps:

1. Select **Debug | Start Debugging** to run the project. Observe that the C1TabControl control's tabstrip runs along the bottom of the control and features sloped tabs.
2. Enter content in the first tab's **RichEditBox**.
3. Click the second tab, notice that the focus changes to this tab.
4. Click the **Close** button on the tab and observe that the tab closes.

Congratulations!

You have completed all four steps of the **TabControl for UWP** quick start. In this quick start, you created a project with a fully customized C1TabControl.

TreeView for UWP

Get a hierarchical view of your data items with **TreeView for UWP**. The familiar TreeView UI is now available for Windows 8 applications. Supports collapsible nodes, hierarchical templates, check box nodes, editing, and drag-and-drop operations (mouse only in Beta version).

TreeView for UWP Key Features

TreeView for UWP allows you to create customized, rich applications. Make the most of **TreeView for UWP** by taking advantage of the following key features:

- **Customizable Nodes**

  Node headers are content elements, so they can host any type of element. Add images, check boxes, or whatever your application requires. Provide editing functionality using the customizable EditTemplate property.

- **Show Connecting Lines**

  Show connected lines in C1TreeView by simply setting the ShowLines property. This gives the appearance of a classic windows treeview. Adjust the appearance of the lines with several simple properties (**LineThickness**/**LineStroke**).

- **Hierarchical Templates**

  You can use different templates for different node types without having to subclass the C1TreeViewItem class.

- **Drag-and-drop Nodes**

  C1TreeView supports drag-and-drop operations within the tree. Simply set the AllowDragDrop property to true and users will be able to reorder nodes within the tree by dragging them with the mouse. Note that drag-and-drop functionality is only supported via keyboard and mouse in the beta version.

TreeView for UWP Quick Start

The following quick start guide is intended to get you up and running with **TreeView for UWP**. In this quick start, you'll start in Visual Studio to create a new project, add a C1TreeView control to your application, and then add content to the C1TreeView control's content area.

Step 1 of 3: Creating an Application with a C1TreeView Control

In this step, you'll begin in Visual Studio to create a UWP application using **TreeView for UWP**.

Complete the following steps:

1. Select **File | New | Project**.
2. In the **New Project** dialog box, select Templates | Visual C# | Windows | Universal. From the templates list, select **Blank App (Universal Windows)**. Enter a **Name** and click **OK** to create your project.
3. Open **MainPage.xaml** if it isn't already open, place the cursor between the <Grid> and </Grid> tags, and click once.
4. Navigate to the Toolbox and double-click the **C1TreeView** icon to add the treeview control to **MainPage.xaml**. The XAML markup will now look similar to the following:

Markup

```
<Page xmlns:Xaml="using:C1.Xaml" x:Class="C1TreeViewQuickStart.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:C1TreeViewQuickStart"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Xaml:C1TreeView />
    </Grid>
</Page>
```

Note that the C1.Xaml namespace and <Xaml:C1TreeView /> tag have been added to the project.

5. Give your grid a name by adding x:Name="Tree" to the <Xaml:C1TreeView> tag and add a closing tag so that it appears similar to the following:

Markup

```
<Xaml:C1TreeView x:Name="Tree"></Xaml:C1TreeView>
```

By giving the control a unique identifier, you'll be able to access the **C1TreeView** control in code.

You've successfully created a Universal Windows application containing a C1TreeView control. In the next step, you will customize the appearance and behavior of the **C1TreeView** control.

Step 2 of 3: Adding C1TreeView Items toC1TreeView

This lesson will show you how to add static C1TreeView items to the C1TreeView control. To add static C1TreeViewItems to the **C1TreeView** control in the XAML, complete the following steps:

1. Add the **C1TreeViewItem** to create the top-level node called "Book List". Within the <Xaml:C1TreeViewItem> tag add Header="Book List". This will create a top-level node that at run time. The XAML markup appears as follows:

Markup

```
<Xaml:C1TreeViewItem Header="Book List"></Xaml:C1TreeViewItem>
```

2. Add two child **C1TreeViewItems** below the <Xaml:C1TreeViewItem> tag to create two child nodes beneath the Book List node and add Header="Language Books". In the second child node add Header="Security Books". The XAML markup appears as follows:

Markup

```
<Xaml:C1TreeViewItem Header="Language Books"/>
<Xaml:C1TreeViewItem Header="Security Books"/>
```

3. Add another <Xaml:C1TreeViewItem> tag to create a new top level node that will hold two child nodes. The XAML markup appears as follows:

Markup

```
<Xaml:C1TreeViewItem Header="Classic Books">
<Xaml:C1TreeViewItem Header="Catch-22"/>
<Xaml:C1TreeViewItem Header="The Great Gatsby"/>
```

4. Add two closing <Xaml:C1TreeViewItem> tags to close the **Book List** node and **Classic Books** node. You should have all of the following XAML markup now included in your MainPage.xaml:

Markup

```
<Page xmlns:Xaml="using:C1.Xaml" x:Class="C1TreeViewQuickStart.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:C1TreeViewQuickStart"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="d">
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
<Xaml:C1TreeView x:Name="Tree">
         <Xaml:C1TreeViewItem Header="Book List">
                <Xaml:C1TreeViewItem Header="Language Books"/>
                <Xaml:C1TreeViewItem Header="Security Books"/>
                <Xaml:C1TreeViewItem Header="Classic Books">
                    <Xaml:C1TreeViewItem Header="Catch-22"/>
                    <Xaml:C1TreeViewItem Header="The Great Gatsby"/>
                </Xaml:C1TreeViewItem>
         </Xaml:C1TreeViewItem>
</Xaml:C1TreeView>
    </Grid>
</Page>
```

5. Run the project and notice that the Book node is not expanded. You can expand it by clicking on the arrow image.

In this step, you added several **C1TreeViewItems** to the **C1TreeView** control. In the next step, you will customize the behavior and appearance of the **C1TreeView** control.

Step 3 of 3: Customizing TreeView'sAppearance and Behavior

In the previous step you worked in Visual Studio to create **C1TreeViewItems** in XAML markup. In this step you'll customize the C1TreeView control's appearance and behavior in using XAML code.

To customize **TreeView for UWP**, complete the following steps:

1. Place your cursor within the <Xaml:C1TreeView> tag. Within the <Xaml:C1TreeView> tag add SelectionMode="Extended". This will create a top-level node that you will be able to select multiple tree items by holding the shift and control keys. The XAML markup appears as follows:

Markup

```
<Xaml:C1TreeView x:Name="Tree" SelectionMode="Extended">
```

2. Place your cursor within the first <Xaml:C1TreeViewItem> tag. Within the <Xaml:C1TreeViewItem> add

IsExpanded="True" IsSelected="True". This will create a top-level node that appears selected and expanded at run time. The XAML markup appears as follows:

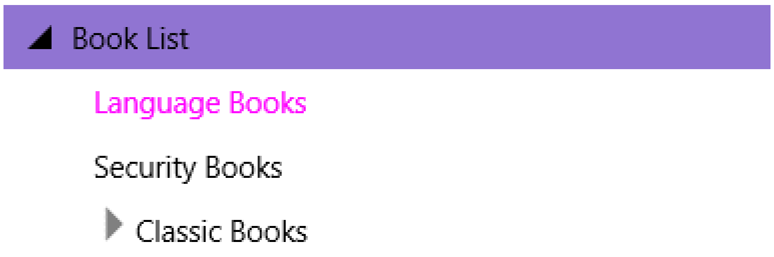
Markup
```
<Xaml:C1TreeViewItem Header="Book List" IsExpanded="True" IsSelected="True">
```

3. Locate the tag that reads <Xaml:C1TreeViewItem Header="Language Books">. Add Foreground="Fuchsia" within the <Xaml:C1TreeViewItem Header="Language Books"> tag. The "Classic Books" tree item text will now appear fuchsia-colored. The XAML markup will resemble the following:

Markup
```
<Xaml:C1TreeViewItem Header="Language Books" Foreground="Fuchsia"/>
```

4. From the **Debug** menu, select **Start Debugging** to view how your application will appear at run time.

   It will appear similar to the following image:



Observe the following behavioral and appearance changes for **C1TreeView**:

- The **C1TreeView** appears expanded.
- The first **C1TreeViewItem** appears selected.
- The second **C1TreeViewItem** has fuchsia-colored text.
- You can select multiple **C1TreeViewItem**s holding down the CTRL or SHIFT keys while selecting items.

Congratulations!

You've successfully completed the **TreeView for UWP** quick start. In this quick start, you created and customized a **TreeView for UWP** application, added static **C1TreeViewItem**s, and observed several of the control's run-time behavior.

C1TreeView Structure

The C1TreeView class is a **StackPanel** with two elements:

- A header that represents the actual node, with a button to collapse and expand the children.
- A body that is another **StackPanel** and contains other nodes.

You can add images to a node by grabbing its first child (the header), casting that to a **StackPanel**, and inserting an image element at whatever position you prefer. For example:

Visual Basic
```
Dim nodeHeader As StackPanel = TryCast(TreeNode.Children(0), StackPanel)
nodeHeader.Children.Insert(0, myImage)
```

C#

```
StackPanel nodeHeader = TreeNode.Children[0] as StackPanel;
nodeHeader.Children.Insert(0, myImage);
```

TreeView Creation

C1TreeViewItems can be added to the C1TreeView control as static items defined either in the XAML markup or in the code behind or can be defined on your page or user control by using any of the following methods:

- Static creation using XAML syntax or programmatically through the code behind file
- Dynamic creation using a constructor to create new instances of the **C1TreeViewItem** class.
- Data source creation through binding **C1TreeView** to a **SiteMapDataSource**, **XMLDataSource**, or an **AccessDataSource**.

Static TreeView Creation

Each node in the Tree is represented by a name/value pair, defined by the text and value properties of treenode, respectively. The text of a node is rendered, whereas the value of a node is not rendered and is typically used as additional data for handling postback events.

A static menu is the simplest way to create the treeview structure.

To display static C1TreeViewItems using XAML syntax, first nest opening and closing <Xaml:C1TreeViewItem> tags between opening and closing tags of the C1TreeView control. Next, create the treeview structure by nesting <Xaml:C1TreeViewItem> elements between opening and closing <Xaml:C1TreeViewItem> tags. Each <Xaml:C1TreeViewItem> element represents a node in the control and maps to a C1TreeViewItem object.

Declarative syntax can be used to define the **C1TreeViewItem**s inline on your page.

For example:

Markup

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Xaml:C1TreeView x:Name="Tree">
        <Xaml:C1TreeViewItem Header="Book List" IsExpanded="True" IsSelected="True">
            <Xaml:C1TreeViewItem Header="Language Books"/>
            <Xaml:C1TreeViewItem Header="Security Books"/>
            <Xaml:C1TreeViewItem Header="Classic Books">
                <Xaml:C1TreeViewItem Header="Catch-22"/>
                <Xaml:C1TreeViewItem Header="The Great Gatsby"/>
            </Xaml:C1TreeViewItem>
        </Xaml:C1TreeViewItem>
    </Xaml:C1TreeView>
</Grid>
```

Dynamic TreeView Creation

Dynamic treeviews can be created on the server side or client side. When creating a dynamic treeview on the server side, use a constructor to dynamically create a new instance of the C1TreeView class. For example:

Visual Basic

```
Namespace TreeViewQuickStart
    Public Partial Class MainPage
        Inherits UserControl
        Public Sub New()
            InitializeComponent()

            InitializeTreeView()
        End Sub
        Private Sub InitializeTreeView()

            ' Remove items that were added at design time

            Tree.Items.Clear()

            Dim booklist As New C1TreeViewItem()
            booklist.Header = "Book List"
            Tree.Items.Add(booklist)

            ' Adding child items
            Dim language As New C1TreeViewItem()
            language.Header = "Language Books"
            booklist.Items.Add(language)

            ' Adding child items
            Dim security As New C1TreeViewItem()
            security.Header = "Security Books"
            booklist.Items.Add(security)

            ' Adding child items
            Dim classic As New C1TreeViewItem()
            classic.Header = "Classic Books"
            booklist.Items.Add(classic)

            ' Adding child items
            Dim subclassic As New C1TreeViewItem()
            subclassic.Header = "Catch-22"
            classic.Items.Add(subclassic)
            Dim subclassic2 As New C1TreeViewItem()
            subclassic2.Header = "The Great Gatsby"
            classic.Items.Add(subclassic2)
        End Sub
    End Class
End Namespace
```

C#

```
namespace TreeViewQuickStart
{
    public partial class MainPage : UserControl
    {
        public MainPage()
        {
            InitializeComponent();
            InitializeTreeView();

        }
       void InitializeTreeView()
       {

            // Remove items that were added at design time

            Tree.Items.Clear();

            C1TreeViewItem booklist = new C1TreeViewItem();
            booklist.Header = "Book List";
           Tree.Items.Add(booklist);
           // Adding child items
          C1TreeViewItem language = new C1TreeViewItem();
          language.Header = "Language Books";
          booklist.Items.Add( language );

         // Adding child items
         C1TreeViewItem security = new C1TreeViewItem();
         security.Header = "Security Books";
         booklist.Items.Add(security);

        // Adding child items
        C1TreeViewItem classic = new C1TreeViewItem();
        classic.Header = "Classic Books";
        booklist.Items.Add(classic);

       // Adding child items
       C1TreeViewItem subclassic = new C1TreeViewItem();
       subclassic.Header = "Catch-22";
       classic.Items.Add(subclassic);
       C1TreeViewItem subclassic2 = new C1TreeViewItem();
       subclassic2.Header = "The Great Gatsby";
       classic.Items.Add(subclassic2);
}
    }
}
```

Data Source TreeView Creation

TreeView items can be created from a hierarchal datasource control such as an **XMLDataSource** or **SiteMapDataSource**. This allows you to update the treeview items without having to edit code.

When using multi-level data as **ItemsSource** for the C1TreeView, you need to specify a C1HierarchicalDataTemplate for the items.

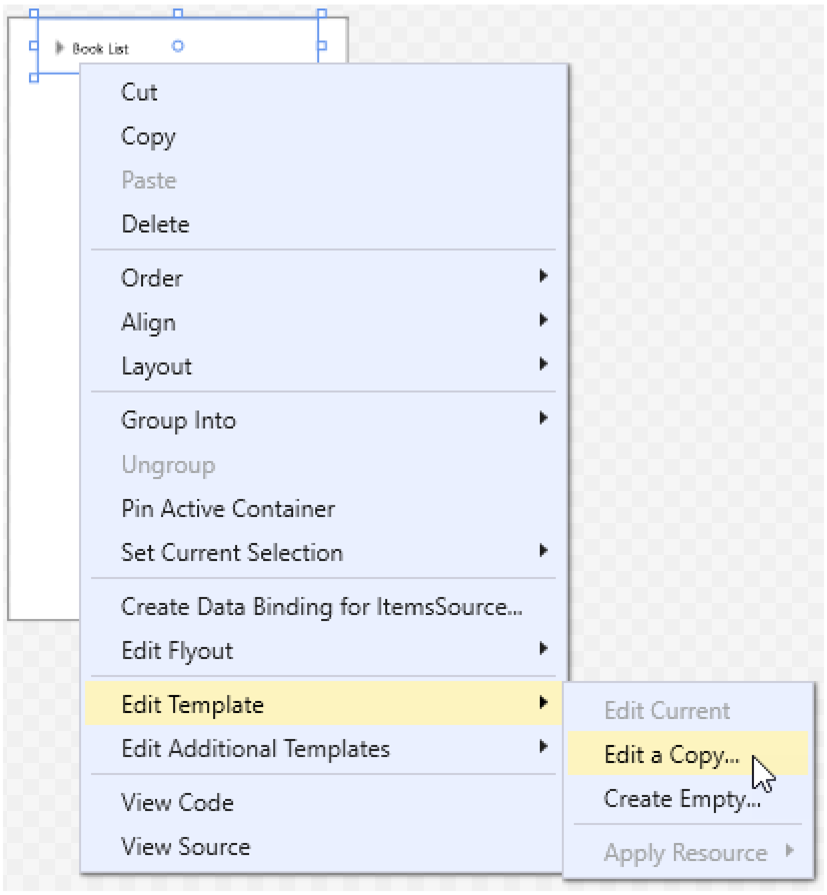
The template will tell the C1TreeView where to find the next level of data; this is done through the ItemsSource property of the C1HierarchicalDataTemplate.

C1TreeView Templates

One of the main advantages to using a UWP control is that controls are "lookless" with a fully customizable user interface. Just as you design your own user interface (UI), or look and feel, for UWP applications, you can provide your own UI for data managed by **TreeView for UWP**. Extensible Application Markup Language (XAML; pronounced "Zammel"), an XML-based declarative language, offers a simple approach to designing your UI without having to write code.

Accessing Templates

You can access templates in Microsoft Expression Blend by selecting the C1TreeView control and, in the menu, selecting **Edit Template**. Select **Edit a Copy** to create an editable copy of the current template or **Create Empty**, to create a new blank template.



Once you've created a new template, the template will appear in the **Objects and Timeline** window. Note that you can use the Template property to customize the template.

Note: If you create a new template through the menu, the template will automatically be linked to that template's property. If you manually create a template in XAML you will have to link the appropriate template property to the template you've created.

Additional Templates

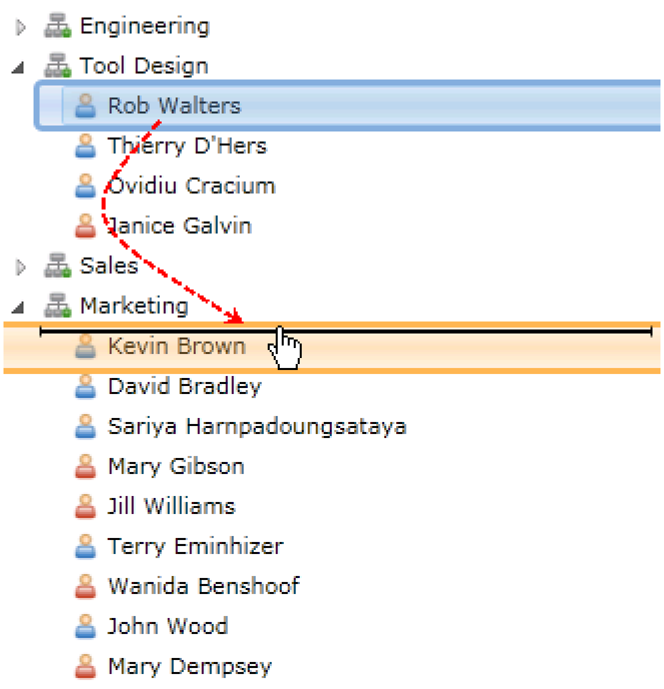
In addition to the default template, the C1TreeView control includes a few additional templates. These additional templates can also be accessed in Microsoft Expression Blend – in Blend select the C1TreeView control and, in the menu, select **Edit Additional Templates**. Choose a template, and select **Create Empty**:

TreeView Behavior

Drag-and-Drop Nodes

You can drag-and-drop **C1TreeViewItems** on nodes, in between nodes, or from one tree to another tree when the C1TreeView.AllowDragDrop property is set to **True**.

The following image shows a C1TreeViewItem being dragged from one C1TreeView to another C1TreeView. An arrow or vertical line can be used as a visual cue to show you where the **C1TreeViewItem** is going to be dropped when either the C1TreeView.DragDropArrowMarker or C1TreeView.DragDropLineMarker properties are applied.



Load on Demand

Instead of fully populating each node when the application starts, you can use a technique called delayed loading, where nodes are populated on demand when the user expands them. This allows the application to load faster and use resources more efficiently

To implement the delayed loading nodes, use the following code:

Visual Basic

```
Public Sub New()
      InitializeComponent()
      ' No changes here.
      ' ...
      ' Initialize the C1TreeView
      InitializeTreeView()
End Sub
Private Sub InitializeTreeView()
      ' Remove items that were added at design time
      _tv.Items.Clear()

      ' Scan every type in the assembly
      For Each t As Type In _tv.[GetType]().Assembly.GetTypes()
            If t.IsPublic AndAlso Not t.IsSpecialName AndAlso Not t.IsAbstract Then
                  ' Add node for this type
                  Dim node As New C1TreeViewItem()
                  node.Header = t.Name
                  node.FontWeight = FontWeights.Bold
                  _tv.Items.Add(node)
                  ' Add subnodes for properties, events, and methods
                  node.Items.Add(CreateMemberNode("Properties", t, MemberTypes.
[Property]))
                  node.Items.Add(CreateMemberNode("Events", t, MemberTypes.[Event]))
                  node.Items.Add(CreateMemberNode("Methods", t, MemberTypes.Method))
            End If
      Next
End Sub
Private Function CreateMemberNode(header As String, memberTypes As MemberTypes) As
C1TreeViewItem
      ' Create the node
      Dim node As New C1TreeViewItem()
      node.Header = header
      node.Foreground = New SolidColorBrush(Colors.DarkGray)
      ' Hook up event hander to populate the node before expanding it
      AddHandler node.Expanding, AddressOf node_Expanding
      ' Save information needed to populate the node
      node.Tag = memberTypes
      ' Add a dummy node so this node can be expanded
      node.Items.Add(New C1TreeViewItem())
      node.IsExpanded = False
      ' Finish
      Return node
End Function
'populate the node
Private Sub node_Expanding(sender As Object, e As RoutedEventArgs)
      ' Get the node that fired the event
      Dim node As C1TreeViewItem = TryCast(sender, C1TreeViewItem)
      ' Unhook event handler (we'll populate the node and be done with it)
      RemoveHandler node.Expanding, AddressOf node_Expanding
      ' Remove dummy node
      node.Items.Clear()
      ' Populate the node
      Dim type As Type = DirectCast(node.Parent.Tag, Type)
      Dim memberTypes As MemberTypes = DirectCast(node.Tag, MemberTypes)
      Dim bf As BindingFlags = BindingFlags.[Public] Or BindingFlags.Instance
      For Each mi As MemberInfo In type.GetMembers(bf)
            If mi.MemberType = memberTypes Then
                  If Not mi.Name.StartsWith("get_") AndAlso Not
mi.Name.StartsWith("set_") Then
                        Dim item As New C1TreeViewItem()
                        item.Header = mi.Name
                        item.FontSize = 12
                        node.Items.Add(item)
                  End If
            End If
      Next
End Sub
```

C#

```
public Page()
{
    InitializeComponent();
    // No changes here.
    // ...
    // Initialize the C1TreeView
    InitializeTreeView();
}
void InitializeTreeView()
{
  // Remove items that were added at design time
  _tv.Items.Clear();

   // Scan every type in the assembly
  foreach (Type t in _tv.GetType().Assembly.GetTypes())
  {
    if (t.IsPublic && !t.IsSpecialName && !t.IsAbstract)
    {
      // Add node for this type
      C1TreeViewItem node = new C1TreeViewItem();
      node.Header = t.Name;
      node.FontWeight = FontWeights.Bold;
      _tv.Items.Add(node);
      // Add subnodes for properties, events, and methods
      node.Items.Add(CreateMemberNode("Properties", t, MemberTypes.Property));
      node.Items.Add(CreateMemberNode("Events", t, MemberTypes.Event));
      node.Items.Add(CreateMemberNode("Methods", t, MemberTypes.Method));
    }
  }
}
C1TreeViewItem CreateMemberNode(string header, MemberTypes memberTypes)
{
  // Create the node
  C1TreeViewItem node = new C1TreeViewItem();
  node.Header = header;
  node.Foreground = new SolidColorBrush(Colors.DarkGray);
   // Hook up event hander to populate the node before expanding it
  node.Expanding += node_Expanding;
   // Save information needed to populate the node
  node.Tag = memberTypes;
   // Add a dummy node so this node can be expanded
  node.Items.Add(new C1TreeViewItem());
  node.IsExpanded = false;
   // Finish
  return node;
}
//populate the node
void node_Expanding(object sender, RoutedEventArgs e)
{
  // Get the node that fired the event
  C1TreeViewItem node = sender as C1TreeViewItem;
  // Unhook event handler (we'll populate the node and be done with it)
  node.Expanding -= node_Expanding;
   // Remove dummy node
  node.Items.Clear();
   // Populate the node
  Type type = (Type)node.Parent.Tag;
  MemberTypes memberTypes = (MemberTypes)node.Tag;
   BindingFlags bf = BindingFlags.Public | BindingFlags.Instance;
  foreach (MemberInfo mi in type.GetMembers(bf))
  {
    if (mi.MemberType == memberTypes)
    {
      if (!mi.Name.StartsWith("get_") && !mi.Name.StartsWith("set_"))
      {
        C1TreeViewItem item = new C1TreeViewItem();
        item.Header = mi.Name;
        item.FontSize = 12;
        node.Items.Add(item);
      }
    }
  }
}
```

This implementation hooks up an event hander for the **Expanding** event, so we can populate the node when the user tries to expand it. We also save the information we will need to populate the node in the **Tag** property. Finally, we add a dummy child node so the user will be able to expand this node and trigger the Expanding event that will populate the node.

Note that instead of using the **Tag** property, we could also have derived a custom class from **C1TreeViewItem** and built all the delay-load logic into that class. This would be a more elegant approach, but unfortunately UWP doesn't support template inheritance. If you derive a class from a class that has a template (such as **Button** or **C1TreeViewItem**), the template is not inherited, and you have to provide a template yourself or your derived class will be just an empty control.

Node Selection

When you click on a node at run time it is automatically marked as selected. Clicking a node will raise the C1TreeView.SelectionChanged event to provide custom functionality. To have the nodes marked as selected without clicking them you can enable the C1TreeViewItem.IsSelected property.

When the user selects a new item, the C1TreeView fires the C1TreeView.SelectionChanged event. You can then retrieve the item that was selected using the C1TreeView.SelectedItem property.

There're several ways to do this. One is to assign additional data to the Tag property of each C1TreeViewItem as you create them. Later, you can inspect the **Tag** property to retrieve the information. For example:

Visual Basic

```
' Create a node and assign some data to its Tag property
Dim item As New C1TreeViewItem()
item.Header = "Beverages"
item.Tag = beveragesID
```

C#

```
// Create a node and assign some data to its Tag property
C1TreeViewItem item = new C1TreeViewItem();
item.Header = "Beverages";
item.Tag = beveragesID;
```

Later, use the information in whatever way you see fit:

Visual Basic

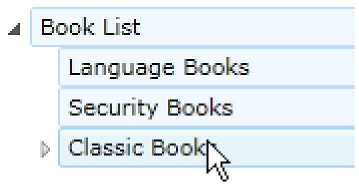
```
Dim item As C1TreeViewItem = _tv.SelectedItem
    ' Handle beverages node
If TypeOf item.Tag Is Integer AndAlso CInt(item.Tag) = beveragesID Then
End If
```

C#

```
C1TreeViewItem item = _tv.SelectedItem;
if (item.Tag is int && (int)item.Tag == beveragesID)
{
    // Handle beverages node
}
```

If the C1TreeView.SelectionMode property is set to Multiple then multiple nodes can be selected at one time by holding down the control key while mouse clicking multiple nodes. To unselect a node, click on it again.

The nodes are marked as selected in the following C1TreeView:



Node Navigation

C1TreeView supports mouse and keyboard navigation.

Navigating C1TreeViewItem**s using the mouse**

The following table describes the actions and corresponding mouse commands when navigating through the **C1TreeViewItems**:

Action	**Mouse Command**
Expand a node	Click on the plus sign at the left of the node's name.

Collapse a node	Click on the minus sign at the left of the node's name.
Select a node	Click on the node's name.

Navigating C1TreeViewItems using the keyboard

The following table describes the actions and their associated keys to use when navigating through **C1TreeViewItem**s:

Action	Keyboard Command
Expand a node	+ KEY
Collapse a node	- KEY
Move up a node	UP ARROW KEY
Move down a down	DOWN ARROW KEY
Select multiple nodes	MOUSE + CTRL KEY

TreeView for UWP Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET and know how to use the C1TreeView control in general. If you are unfamiliar with the **TreeView for UWP** product, please see the TreeView for UWP Quick Start first.

Each topic in this section provides a solution for specific tasks using the **TreeView for UWP** product.

Each task-based help topic also assumes that you have created a new Windows Store project.

Adding C1TreeViewItems using Code

To add static **C1TreeView** items to the C1TreeView control in the code behind file, edit the code in your application so it appears like the following:

Visual Basic

```
Imports C1.Xaml
Class MainPage
    Public Sub New()
        InitializeComponent()
        InitializeTreeView()
    End Sub
    Private Sub InitializeTreeView()
        ' Remove items that were added at design time
        Tree.Items.Clear()
        Dim booklist As New C1TreeViewItem()
        booklist.Header = "Book List"
        Tree.Items.Add(booklist)

        ' Adding child items
        Dim language As New C1TreeViewItem()
        language.Header = "Language Books"
        booklist.Items.Add(language)

        ' Adding child items
        Dim security As New C1TreeViewItem()
        security.Header = "Security Books"
        booklist.Items.Add(security)

        ' Adding child items
        Dim classic As New C1TreeViewItem()
        classic.Header = "Classic Books"
        booklist.Items.Add(classic)

        ' Adding child items
        Dim subclassic As New C1TreeViewItem()
        subclassic.Header = "Catch-22"
        classic.Items.Add(subclassic)
        Dim subclassic2 As New C1TreeViewItem()
        subclassic2.Header = "The Great Gatsby"
        classic.Items.Add(subclassic2)
    End Sub
End Class
```

C#

```
using C1.Xaml;
public MainPage()
{
    InitializeComponent();
    InitializeTreeView();
}
void InitializeTreeView()
{
    // Remove items that were added at design time
    Tree.Items.Clear();
    C1TreeViewItem booklist = new C1TreeViewItem();
    booklist.Header = "Book List";
    Tree.Items.Add(booklist);

   // Adding child items
    C1TreeViewItem language = new C1TreeViewItem();
    language.Header = "Language Books";
    booklist.Items.Add( language );

    // Adding child items
    C1TreeViewItem security = new C1TreeViewItem();
    security.Header = "Security Books";
    booklist.Items.Add(security);

    // Adding child items
    C1TreeViewItem classic = new C1TreeViewItem();
    classic.Header = "Classic Books";
    booklist.Items.Add(classic);
    // Adding child items
    C1TreeViewItem subclassic = new C1TreeViewItem();
    subclassic.Header = "Catch-22";
    classic.Items.Add(subclassic);
    C1TreeViewItem subclassic2 = new C1TreeViewItem();
    subclassic2.Header = "The Great Gatsby";
    classic.Items.Add(subclassic2);
}
```

Getting the Text or Value of the SelectedItem in a TreeView

The Header property will return the values contained in your **C1TreeViewItem**s, you can get the string value using the following code:

Visual Basic

```
Dim item As C1TreeViewItem = _tree.SelectedItem
_textBlock.Text = item.Header.ToString()
```

C#

```
C1TreeViewItem item = _tree.SelectedItem;
_textBlock.Text = item.Header.ToString();
```

What You've Accomplished

Run your application and observe that the **Header** property will return the values contained in your **C1TreeViewItem**s.

Adding Check Boxes to the TreeView

You can easily add check boxes to the C1TreeView control, check boxes can appear before text and allow users to select a tree view item. The following XAML markup adds check boxes to the **C1TreeView**:

Markup

```
<Xaml:C1TreeView Name="C1TreeView1" Height="300" Width="200" >
    <Xaml:C1TreeViewItem IsExpanded="True" Margin="10">
        <Xaml:C1TreeViewItem.Header>
            <CheckBox>
                <CheckBox.Content>
                    <TextBlock Text="Desktop" />
                </CheckBox.Content>
            </CheckBox>
        </Xaml:C1TreeViewItem.Header>
    <Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem.Header>
            <CheckBox>
                <CheckBox.Content>
                    <TextBlock Text="User" />
                </CheckBox.Content>
            </CheckBox>
        </Xaml:C1TreeViewItem.Header>
    </Xaml:C1TreeViewItem>
    <Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem.Header>
            <CheckBox>
                <CheckBox.Content>
                    <TextBlock Text="Public" />
                </CheckBox.Content>
            </CheckBox>
        </Xaml:C1TreeViewItem.Header>
        <Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem.Header>
                <CheckBox>
                    <CheckBox.Content>
                    <TextBlock Text="Favorites" />
                    </CheckBox.Content>
                </CheckBox>
            </Xaml:C1TreeViewItem.Header>
    </Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem.Header>
                <CheckBox>
                    <CheckBox.Content>
                    <TextBlock Text="Public Downloads" />
                    </CheckBox.Content>
                </CheckBox>
            </Xaml:C1TreeViewItem.Header>
    </Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem.Header>
                <CheckBox>
                    <CheckBox.Content>
                    <TextBlock Text="Public Music" />
                    </CheckBox.Content>
                </CheckBox>
            </Xaml:C1TreeViewItem.Header>
    </Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem.Header>
                <CheckBox>
                    <CheckBox.Content>
                    <TextBlock Text="Public Pictures" />
                    </CheckBox.Content>
                </CheckBox>
            </Xaml:C1TreeViewItem.Header>
    </Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem.Header>
                <CheckBox>
                    <CheckBox.Content>
                    <TextBlock Text="Public Videos" />
                    </CheckBox.Content>
                </CheckBox>
            </Xaml:C1TreeViewItem.Header>
    </Xaml:C1TreeViewItem>
            </Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem IsExpanded="True">
        <Xaml:C1TreeViewItem.Header>
            <CheckBox>
                <CheckBox.Content>
                    <TextBlock Text="Computer" />
                </CheckBox.Content>
            </CheckBox>
        </Xaml:C1TreeViewItem.Header>
                <Xaml:C1TreeViewItem IsExpanded="True">
            <Xaml:C1TreeViewItem.Header>
                <CheckBox>
                    <CheckBox.Content>
                    <TextBlock Text="Local Disk (C:)" />
                    </CheckBox.Content>
                </CheckBox>
            </Xaml:C1TreeViewItem.Header>
            <Xaml:C1TreeViewItem>
                <Xaml:C1TreeViewItem.Header>
                    <CheckBox>
                        <CheckBox.Content>
                        <TextBlock Text="Program Files" />
                        </CheckBox.Content>
                    </CheckBox>
                </Xaml:C1TreeViewItem.Header>
        </Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem>
                <Xaml:C1TreeViewItem.Header>
                    <CheckBox>
                        <CheckBox.Content>
                            <TextBlock Text="Users" />
                        </CheckBox.Content>
                    </CheckBox>
                </Xaml:C1TreeViewItem.Header>
        </Xaml:C1TreeViewItem>
            <Xaml:C1TreeViewItem>
                <Xaml:C1TreeViewItem.Header>
                    <CheckBox>
                        <CheckBox.Content>
                            <TextBlock Text="Windows" />
                        </CheckBox.Content>
                    </CheckBox>
                </Xaml:C1TreeViewItem.Header>
        </Xaml:C1TreeViewItem>
    </Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem>
```

```
            <Xaml:C1TreeViewItem.Header>
                <CheckBox>
                    <CheckBox.Content>
                        <TextBlock Text="DVD Drive (D:)" />
                    </CheckBox.Content>
                </CheckBox>
            </Xaml:C1TreeViewItem.Header>
        </Xaml:C1TreeViewItem>
    </Xaml:C1TreeViewItem>
    <Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem.Header>
            <CheckBox>
                <CheckBox.Content>
                    <TextBlock Text="Network" />
                </CheckBox.Content>
            </CheckBox>
        </Xaml:C1TreeViewItem.Header>
            </Xaml:C1TreeViewItem>
    <Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem.Header>
            <CheckBox>
                <CheckBox.Content>
                    <TextBlock Text="Control Panel" />
                </CheckBox.Content>
            </CheckBox>
        </Xaml:C1TreeViewItem.Header>
            </Xaml:C1TreeViewItem>
    <Xaml:C1TreeViewItem>
        <Xaml:C1TreeViewItem.Header>
            <CheckBox>
                <CheckBox.Content>
                    <TextBlock Text="Recycle Bin" />
                </CheckBox.Content>
            </CheckBox>
        </Xaml:C1TreeViewItem.Header>
        </Xaml:C1TreeViewItem>
    </Xaml:C1TreeViewItem>
</Xaml:C1TreeView>
```

Enabling Drag-and-Drop

C1TreeView supports drag-and-drop behavior. For more information see the Drag-and-Drop Nodes topic. To enable drag-and-drop behavior, set the C1TreeView.AllowDragDrop property to **True**:

Visual Basic

```
C1TreeView.AllowDragDrop = True
```

C#

```
C1TreeView.AllowDragDrop = true;
```

To enable visual drag-and-drop indicators you can set the C1TreeView.DragDropArrowMarker and C1TreeView.DragDropLineMarker properties.