Windows Store App Development: Resources

By Pete Brown, author of *Windows Store App Development*

The controls and their templates are made up of panels, shapes, and other controls, which are ultimately made up of other panels and shapes. You use the brushes to paint those controls as well as the text that's so prevalent in Windows 8 apps. Brushes and colors are commonly stored as resources in the application or in standard resource dictionary files. In this article, based on chapter 7 of [Windows Store App Development](), author Pete Brown explains resource management.

You may also be interested in…

Cascading Style Sheets (CSS) make it possible to reuse styles, colors, and more in HTML. In XAML, we've been defining all of our object properties at the object level. If you follow that approach, and you decide to change the color scheme for your app, you'll have a lot of manual searching and replacing to do. The same is true in HTML: If you define all your colors and styles locally, it makes reskinning the site much harder.

In XAML, reuse of colors and styles is handled through *resources*. Resources are reusable instances of types. Typically, resources are things like brushes or styles and templates. They can also be class instances, viewmodels, and more. Almost any class can be a resource, but because the objects in a resource dictionary must be sharable, most visual elements cannot. Sharing not only helps with reuse, but it can be a memory saver as well, because you have more control over the number of objects in use.

You can declare resources from code, but the more common approach is to declare them in markup. In this way, you can define reusable objects that exist within an element's scope, on a page, app-wide, or through the use of dictionaries, or any combination of those scopes.

All resources are added to a special type of lookup table called a *resource dictionary*. In some cases, this is transparent to you. In others, you explicitly create separate resource dictionary files. In all cases, when using resources from markup, you do so using the `StaticResource` markup extension.

In this introduction to resources, I'll focus on simple resources: brushes. First, we'll look at resources defined locally and on a single page. Then, we'll look at the common practice of defining app-wide resources. Finally, we'll look at the concept of a resource dictionary, something the built-in templates make extensive use of for standard control styles and colors.

Local and page resources

Resources may be used within the scope in which they're defined. If you define a resource at the `Grid` level, only elements inside the `Grid` will see it. If you define the resource at a `ListBox` level, only the children of the `ListBox` will see it.

Because of this, one common place to define a resource is at the `Page` or `User-Control` level. That way, the resource works for every element inside that `Page` or `UserControl`, including the `Page` or `UserControl` itself.

The following listing shows how to define and use a page-level resource and a gridlevel (local) resource in XAML.

Listing 1 Page and local resources defined and used in markup

```
<Page x:Class="ResourceExample.MainPage"
```

```
    IsTabStop="false"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ResourceExample"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Page.Resources>     #A
      <LinearGradientBrush x:Key="StandardGradient"      #B
                           StartPoint="0,0" EndPoint="1,1">
        <LinearGradientBrush.GradientStops>
          <GradientStop Offset="0" Color="White" />
          <GradientStop Offset="0.4" Color="Black" />
          <GradientStop Offset="0.6" Color="Black" />
          <GradientStop Offset="1" Color="White" />
        </LinearGradientBrush.GradientStops>
      </LinearGradientBrush>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
      <StackPanel x:Name="StackPanel1">
        <StackPanel.Resources>        #C
          <SolidColorBrush x:Key="ItemBackground"      #D
                           Color="White" />            #D
        </StackPanel.Resources>

        <Grid Height="300" Margin="20"
              Background="{StaticResource ItemBackground}">      #E
          <Rectangle Fill="{StaticResource StandardGradient}"    #F
                     Margin="50"/>
        </Grid>

        <Grid Height="300" Margin="20"
              Background="{StaticResource ItemBackground}">      #E
          <Rectangle Fill="Purple" Margin="50"/>
        </Grid>
      </StackPanel>
    </Grid>
  </Page>
```

#A Page-level resources
#B Resource StandardGradient
#C Local resources
#D Resource ItemBackground
#E Use local resource
#F Use page-level resource

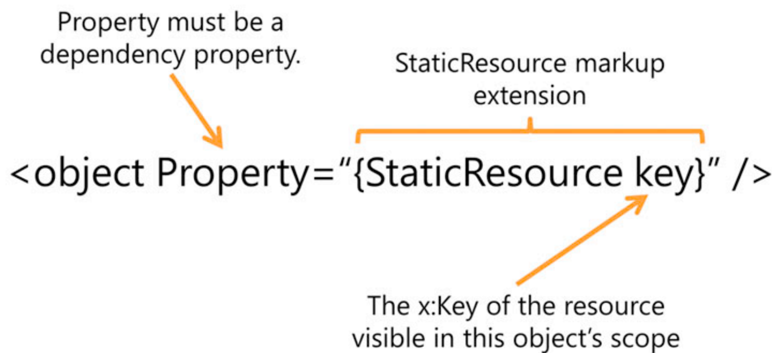


Figure 1 A breakdown of the components of the **`StaticResource`** markup extension as used to assign a property value. The extension is a helpful way of specifying the longer-form **`<object><object.Property> <StaticResource ResourceKey="key">`**… value.

All resources must have a key. That key must be unique within any combined scope so that no two resources visible to an item have the same key. That is, don't define a pagelevel resource and, say, an app-level resource with the same key.

Of course, the terms *local* and *page-level* are simply conveniences. In reality, resources simply have scope, and whether you consider something local really depends on which element's relationship you're looking at.

In this example, resources are defined both at the page level and at a level local to the `StackPanel` named `StackPanel1`. Only elements on the page may use the pagelevel resource `StandardGradient`. Similarly, only the `StackPanel` and child elements of it may use the `ItemBackground` resource.

When using a resource, always use the `StaticResource` markup extension. The format for using this extension is illustrated in figure 1.

Note that in figure 1 I specifically called out that the property must be a dependency property. This is important: Only dependency properties can derive their value from a resource or from an animation. If you run into an error using a resource with a property, double-check to make sure the property is a `DependencyProperty`.

When you want to access resources from code, the `StaticResource` markup extension doesn't come into play. Instead, you simply refer to the resource by its key and get back an appropriate type, as shown here.

Listing 2 Accessing page and local resources from code

```
public MainPage()
{
  this.InitializeComponent();

  var resource = this.Resources["StandardGradient"] as Brush;     #A

  var localRes = StackPanel1.Resources["ItemBackground"] as Brush;   #B
}
```

#A Page resource
#B StackPanel-level local resource

You can add resources from code, although that's very rarely done. Just make sure you add the resources before the visual tree with elements containing references to the resources is loaded. Typically, this is going to be in the constructor, before the `InitializeLayout` function call. The `Loaded` event (which fires after completion of `InitializeLayout` and page loading) is too late to add resources.

Putting resources on a page is a common approach to reusing styles, brushes, and more at only the page level. Resources at levels below that (local resources) are usually confined to data templates just because of a desire to generally keep resources together and easily identified.

Scoping of page and local resources is pretty easy to see. There's one additional scope beyond that, however, that may not be immediately obvious.

Application resources

By far, the most common resource management approach used by app developers is to put the resources in app.xaml. By defining the resources there, they're available to the entire app and may be used in any `Page` or `UserControl`.

Just by virtue of the fact that the resources are defined in app.xaml makes them app-global in scope. Should you decide to promote a page-level resource to an applevel resource, all you need to do is cut and paste the resource itself—everything else stays the same. The next listing shows some resources defined at the application level.

Listing 3 Application resources

```
<Application
  x:Class="ResourceExample.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ResourceExample">

  <Application.Resources>
    <ResourceDictionary>
```

```
      <SolidColorBrush x:Key="AppTextColor" Color="White" />        #A
      <SolidColorBrush x:Key="AppTextBackground" Color="Black" />   #A

      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>   #B
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

#A Regular resources
#B Standard resource dictionary

The `ResourceDictionary` tag is optional when you have only standard keyed resources. But as you'll see shortly, if you plan to merge in any other resource dictionaries, it's required.

Resource dictionaries outside of app.xaml don't start with the `Application` tag. Instead, they start directly with the `ResourceDictionary` tag. Once inside that tag, everything else is the same as it is in app.xaml.

Resource dictionaries

Just as you wouldn't want a single code file with thousands of lines of code, you don't want enormous XAML files. Once the number of resources gets large (where *large* is pretty subjective) you'll want to put them into one or more separate resource dictionaries. Doing so enables you to break them up by groupings that are logical to your application as well as to your team. You'll get more manageable source control and so on.

More important, resource dictionaries enable you to be more thoughtful about which pages merge in which resources. If, for example, you have a set of resources that's used in only 10% of the pages in the app, making those resources available appwide is a waste of memory. Not only that, but loading resources exacts a performance penalty—the more resources you have defined in (or merged into) app.xaml, the longer your app will take to load.

You can create a new resource dictionary by simply adding a new file using the Resource Dictionary template. But every project already has several resource dictionaries of interest. One resource file you may not immediately notice (because it's in the Windows SDK location, not the local project) is the ThemeResources.xaml file, a small portion of which is shown here.

Listing 4 An example resource dictionary

```
<ResourceDictionary
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ResourceDictionary.ThemeDictionaries>   #A
    <ResourceDictionary x:Key="Default">   #B
      <FontFamily
         x:Key="ContentControlThemeFontFamily">Segoe UI</FontFamily>
      <FontFamily
         x:Key="SymbolThemeFontFamily">Segoe UI Symbol</FontFamily>
      <x:Double x:Key="AppBarThemeMinHeight">68</x:Double>
      <Thickness x:Key="TextControlThemePadding">10,3,10,5</Thickness>
      <Thickness x:Key="ToggleButtonBorderThemeThickness">2</Thickness>
      <Thickness x:Key="ToolTipBorderThemeThickness">2</Thickness>
      <SolidColorBrush x:Key="AppBarBackgroundThemeBrush"
                       Color="#E5000000" />
      <SolidColorBrush x:Key="AppBarBorderThemeBrush"
                       Color="#E5000000" />
      <SolidColorBrush x:Key="AppBarItemBackgroundThemeBrush"
                       Color="Transparent" />
      <SolidColorBrush x:Key="AppBarItemDisabledForegroundThemeBrush"
                       Color="#66FFFFFF" />
      <SolidColorBrush x:Key="AppBarItemForegroundThemeBrush"
                       Color="#FFFFFFFF" />
      <SolidColorBrush x:Key="AppBarItemPointerOverBackgroundThemeBrush"
                       Color="#21FFFFFF" />
      <SolidColorBrush x:Key="AppBarItemPointerOverForegroundThemeBrush"
                       Color="#FFFFFFFF" />
      ...
    </ResourceDictionary>
    <ResourceDictionary x:Key="HighContrast">     #D
      ...
```

```
    </ResourceDictionary>
  </ResourceDictionary.ThemeDictionaries>
</ResourceDictionary>
```

#A Default theme
#B Theme support
#C Key system parameter
#D High-contrast theme

There are three important things to notice in this listing:

- Resource dictionaries can themselves have keys.
- Resources aren't limited to brushes.
- Most critical system UI parameters exist as resources.

Resource dictionaries can have keys specifically to support theming. Note that currently the only system themes supported are standard and high contrast.

The second thing to notice is that the resource files contain much more than just brushes. You'll see not only control styles but also simple types like `Font`, `double`, and `Thickness`. Resources can be used to standardize these across the app.

Finally, when designing your own custom controls and page layouts, it will help if you know the contents of the theme resources file. Make use of the built-in sizing and other constant resources whenever possible.

One unique aspect of stand-alone resource dictionaries is that they can be merged into other dictionaries, providing access to the resources defined therein.

> **THE STANDARD RESOURCES** In addition to the StandardStyles.xaml in the Common folder of every new project, there are two other important XAML resource dictionaries. First is the ThemeResources.xaml shown in listing 4. This handles dark and light theme color and style settings. The second is generic.xaml, which includes the Windows 8-style UI templates for the built-in controls. Both resource dictionaries can be found in your Program Files (x86) folder, under \Windows Kits\8.0\Include\winrt\xaml\design. These resources are for your education and to help designers; changing them will not necessarily alter the built-in styles at runtime. These files are updated with the Windows SDK.

Merging resource dictionaries

Resource dictionaries (whether stand-alone or the `Resources` property of any `FrameworkElement`) can include, or "pull in," other resource dictionaries. This is called merging.

Each resource dictionary must merge in any other resource dictionaries it relies on. It's not sufficient for the resource to simply be defined ahead of time; it must be merged in. Think of it like include files in C (if you're familiar with that). Look at each resource dictionary file individually and ensure that it has merged into it all the other resource dictionaries it requires. The XAML resource management system will ensure the same resources aren't physically stored or created multiple times.

The following listing shows the StandardStyles.xaml resource file merged into app.xaml. Every XAML project includes at least this set of merged-in styles to provide common colors, app bar buttons, and more.

Listing 5 The standard resource dictionary merged at the application level

```
<Application
  x:Class="ResourceExample.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:ResourceExample">

  <Application.Resources>
    <ResourceDictionary>
      <SolidColorBrush x:Key="AppTextColor" Color="White" />         #A
      <SolidColorBrush x:Key="AppTextBackground" Color="Black" />    #A

      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>    #B
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
```

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

#A Regular resources
#B Standard resource dictionary

In this example, there are two standard resources and one merged-in resource dictionary. The StandardStyles.xaml can't see the `AppTextColor` or `AppTextBackground` resource; if it needs them, they must be defined in that file, or they must be moved to a separate resource dictionary and merged into the StandardStyles.xaml. All of these resources here can be seen app-wide because they're defined in or merged into the resource dictionary in app.xaml.

Earlier, I mentioned that every resource incurs a load time penalty. So, when you create your final versions of your app, you should remove from StandardStyles.xaml any resources your app isn't using. To verify that you're removing the correct ones, comment them out and run through your tests. Once you're certain, remove the resources to cut down on file size. Should you need additional standard resources in the future, you can copy them in from another project.

Let's say that you're creating something highly custom, like a game. In that game, only a couple of opening screens use the standard styles—everything else is custom drawn on multiple other pages. In that case, you may want to merge the standard styles into only the first couple pages. The next listing shows how to merge in a resource dictionary at the page level.

Listing 6 Merging a resource dictionary at the page level

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


  <Page.Resources>     #A
    <ResourceDictionary>
      <LinearGradientBrush x:Key="StandardGradient"       #B
                           StartPoint="0,0" EndPoint="1,1">
        <LinearGradientBrush.GradientStops>
          <GradientStop Offset="0" Color="White" />
          <GradientStop Offset="0.4" Color="Black" />
          <GradientStop Offset="0.6" Color="Black" />
          <GradientStop Offset="1" Color="White" />
        </LinearGradientBrush.GradientStops>
      </LinearGradientBrush>

      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Common/StandardStyles.xaml"/>    #C
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Page.Resources>
...
</Page>
```

#A Page resources
#B Gradient brush resource
#C Merged-in resource dictionary

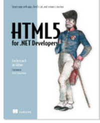
As before, I'm showing both standard resources as well as resource dictionaries together. The reason is, without seeing the syntax (that both must be in the `ResourceDictionary` tag), it can appear that you're unable to mix and match. I want to assure you that you can. You'll notice that the approach here is the same as used in app.xaml.

Regardless of where you define them, resources are a great way to centralize the definitions of colors, fonts, brushes, sizes, and much more in XAML.

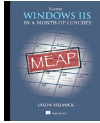
Summary

Brushes can be used to paint the vector graphics and provide borders and shading to most anything. One brush, the `ImageBrush`, crosses the line between vector and bitmapped images by making it possible to paint vector art using an image as the brush. The `Brush` is one of the most commonly used resources. Resources can be local, page scoped, application wide, or, through the use of merged-in resource dictionaries, any combination of those scopes. Once you end up with more than a few resources, consider moving them into separate resource dictionaries rather than just keeping everything in app.xaml.

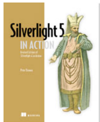
Here are some other Manning titles you might be interested in:



HTML5 for .NET Developers
Jim Jackson II and Ian Gilman



Learn Windows IIS in a Month of Lunches
Jason Helmick



Silverlight 5 in Action
Pete Brown

Last updated: June 6, 2013