One of the most basic language constructs of C# is the class member known as the property. All of us very early on in our first encounters with C# learned the general routine of defining a property. The property is often backed by a private field and includes set and get accessors that reference the private field and do something with a new value:

```csharp
public class MyClass
{
    double quality;

    public double Quality
    {
        set
        {
            quality = value;
            // Do something with the new value
        }
        get
        {
            return quality;
        }
    }
}
```

Properties are sometimes referred to as smart fields. Syntactically, code that accesses a property resembles code that accesses a field. Yet the property can execute some of its own code when the property is accessed.

Properties are also like methods. Indeed, C# code is compiled into intermediate language that implements a property such as Quality with a pair of methods named set_Quality and get_Quality. Yet despite the close functional resemblance between properties and a pair of set and get methods, the property syntax reveals itself to be much more suitable when moving from code to markup. It’s hard to imagine XAML built on an underlying API that is missing properties.

So you may be surprised to learn that Xamarin.Forms implements an enhanced property definition that builds upon C# properties. Or maybe you won’t be surprised. If you already have experience with Microsoft’s XAML-based platforms, you’ll encounter some familiar concepts in this chapter.

The property definition shown above is known as a CLR property because it’s supported by the .NET common language runtime. The enhanced property definition in Xamarin.Forms builds upon the CLR property and is called a bindable property, encapsulated by the BindableProperty class and supported by the BindableObject class.
The Xamarin.Forms class hierarchy

Before exploring the details of the important `BindableObject` class, let’s first discover how `BindableObject` fits into the overall Xamarin.Forms architecture by constructing a class hierarchy.

In an object-oriented programming framework such as Xamarin.Forms, a class hierarchy can often reveal important inner structures of the environment. The class hierarchy shows how various classes relate to one another and the properties, methods, and events that they share, including how bindable properties are supported.

You can construct such a class hierarchy by laboriously going through the online documentation and taking note of what classes derive from what other classes. Or you can write a Xamarin.Forms program to do the work for you and display the class hierarchy on the phone. Such a program makes use of .NET reflection to obtain all the public classes, structures, and enumerations in the `Xamarin.Forms.Core` and `Xamarin.Forms.Xaml` assemblies and arrange them in a tree. The `ClassHierarchy` application demonstrates this technique.

As usual, the `ClassHierarchy` project contains a class that derives from `ContentPage`, named `ClassHierarchyPage`, but it also contains two additional classes, named `TypeInfo` and `ClassAndSubclasses`.

The program creates one `TypeInfo` instance for every public class (and structure and enumeration) in the `Xamarin.Forms.Core` and `Xamarin.Forms.Xaml` assemblies, plus any .NET class that serves as a base class for any Xamarin.Forms class, with the exception of `Object`. (These .NET classes are `Attribute`, `Delegate`, `Enum`, `EventArgs`, `Exception`, `MulticastDelegate`, and `ValueType`.) The `TypeInfo` constructor requires a `Type` object identifying a type but also obtains some other information:

```csharp
class TypeInfo
{
    bool isBaseGenericType;
    Type baseGenericTypeDef;

    public TypeInfo(Type type, bool isXamarinForms)
    {
        Type = type;
        IsXamarinForms = isXamarinForms;
        TypeInfo typeInfo = type.GetTypeInfo();
        BaseType = typeInfo.BaseType;

        if (BaseType != null)
        {
            TypeInfo baseTypeInfo = BaseType.GetTypeInfo();
            isBaseGenericType = baseTypeInfo.IsGenericType;

            if (isBaseGenericType)
            {
                baseGenericTypeDef = baseTypeInfo.GetGenericTypeDefinition();
            }
        }
    }
}
```
A very important part of this class is the IsDerivedDirectlyFrom method, which will return true if passed an argument that is this type’s base type. This determination is complicated if generic classes are involved, and that issue largely accounts for the complexity of the class.

The ClassAndSubclasses class is considerably shorter:

```csharp
class ClassAndSubclasses
{
    public ClassAndSubclasses(Type parent, bool isXamarinForms)
    {
        Type = parent;
        IsXamarinForms = isXamarinForms;
        Subclasses = new List<ClassAndSubclasses>();
    }

    public Type Type { get; }
    public bool IsXamarinForms { get; }
    public List<ClassAndSubclasses> Subclasses { get; }
}
```

The program creates one instance of this class for every Type displayed in the class hierarchy, including Object, so the program creates one more ClassAndSubclasses instance than the number of TypeInformation instances. The ClassAndSubclasses instance associated with Object contains a collection of all the classes that derive directly from Object, and each of those ClassAndSubclasses instances contains a collection of all the classes that derive from that one, and so forth for the remainder of the hierarchy tree.
The ClassHierarchyPage class consists of a XAML file and a code-behind file, but the XAML file contains little more than a scrollable StackLayout ready for some Label elements:

```xml
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ClassHierarchy.ClassHierarchyPage">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                   iOS="5, 20, 0, 0"
                   Android="5, 0, 0, 0"
                   WinPhone="5, 0, 0, 0" />
    </ContentPage.Padding>

    <ScrollView>
        <StackLayout x:Name="stackLayout"
                     Spacing="0" />
    </ScrollView>
</ContentPage>
```

The code-behind file obtains references to the two Xamarin.Forms Assembly objects and then accumulates all the public classes, structures, and enumerations in the classList collection. It then checks for the necessity of including any base classes from the .NET assemblies, sorts the result, and then calls two recursive methods, AddChildrenToParent and AddItemToStackLayout:

```csharp
public partial class ClassHierarchyPage : ContentPage
{
    public ClassHierarchyPage()
    {
        InitializeComponent();

        List<TypeInformation> classList = new List<TypeInformation>();

        GetPublicTypes(typeof(View).GetTypeInfo().Assembly, classList);

        // Get types in Xamarin.Forms.Xaml assembly.
        GetPublicTypes(typeof(Extensions).GetTypeInfo().Assembly, classList);

        // Ensure that all classes have a base type in the list.
        // (i.e., add Attribute, ValueType, Enum, EventArgs, etc.)
        int index = 0;

        // Watch out! Loops through expanding classList!
        do
        {
            // Get a child type from the list.
            TypeInformation childType = classList[index];

            if (childType.Type != typeof(Object))
            {
                bool hasBaseType = false;
```
// Loop through the list looking for a base type.
foreach (TypeInformation parentType in classList)
{
    if (childType.IsDerivedDirectlyFrom(parentType.Type))
    {
        hasBaseType = true;
    }
}

// If there's no base type, add it.
if (!hasBaseType && childType.BaseType != typeof(Object))
{
    classList.Add(new TypeInformation(childType.BaseType, false));
}
}
}

index++;
}
while (index < classList.Count);

// Now sort the list.
classList.Sort((t1, t2) =>
{
    return String.Compare(t1.Type.Name, t2.Type.Name);
});

// Start the display with System.Object.
ClassAndSubclasses rootClass = new ClassAndSubclasses(typeof(Object), false);

// Recursive method to build the hierarchy tree.
AddChildrenToParent(rootClass, classList);

// Recursive method for adding items to StackLayout.
AddItemToStackLayout(rootClass, 0);

void GetPublicTypes(Assembly assembly, List<TypeInformation> classList)
{
    // Loop through all the types.
    foreach (Type type in assembly.ExportedTypes)
    {
        TypeInfo typeInfo = type.GetTypeInfo();

        // Public types only but exclude interfaces.
        if (typeInfo.IsPublic && !typeInfo.IsInterface)
        {
            // Add type to list.
            classList.Add(new TypeInformation(type, true));
        }
    }
}

void AddChildrenToParent(ClassAndSubclasses parentClass, List<TypeInformation> classList)
{
    foreach (TypeInformation typeInformation in classList)
    {
        if (typeInformation.IsDerivedDirectlyFrom(parentClass.Type))
        {
            ClassAndSubclasses subClass =
                new ClassAndSubclasses(typeInformation.Type,
                                        typeInformation.IsXamarinForms);
            parentClass.Subclasses.Add(subClass);
            AddChildrenToParent(subClass,classList);
        }
    }
}

void AddItemToStackLayout(ClassAndSubclasses parentClass, int level)
{
    // If assembly is not Xamarin.Forms, display full name.
    string name = parentClass.IsXamarinForms ? parentClass.Type.Name :
                    parentClass.Type.FullName;

    TypeInfo typeInfo = parentClass.Type.GetTypeInfo();

    // If generic, display angle brackets and parameters.
    if (typeInfo.IsGenericType)
    {
        Type[] parameters = typeInfo.GetGenericArguments();
        name = name.Substring(0, name.Length - 2);
        name += "<";

        for (int i = 0; i < parameters.Length; i++)
        {
            name += parameters[i].Name;
            if (i < parameters.Length - 1)
            {
                name += ", ";
            }
        }
        name += ">";
    }

    // Create Label and add to StackLayout.
    Label label = new Label
    {
        Text = String.Format("{0}{1}", new string( ' ', 4 * level), name),
        TextColor = parentClass.Type.GetTypeInfo().IsAbstract ?
            Color.Accent : Color.Default
    };

    stackLayout.Children.Add(label);

    // Now display nested types.
    foreach (ClassAndSubclasses subclass in parentClass.Subclasses)
    {
        AddItemToStackLayout(subclass, level + 1);
    }
The recursive AddChildrenToParent method assembles the linked list of ClassAndSubclasses instances from the flat classList collection. The AddItemToStackLayout method is also recursive because it is responsible for adding the ClassesAndSubclasses linked list to the StackLayout object by creating a Label view for each class, with a little blank space at the beginning for the proper indentation. The method displays the Xamarin.Forms types with just the class names, but the .NET types include the fully qualified name to distinguish them. The method uses the platform accent color for classes that are not instantiable because they are abstract or static.

Overall, you'll see that the Xamarin.Forms visual elements have the following general hierarchy:

```
System.Object
   BindableObject
      Element
         VisualElement
            View
               ... Layout
                  ... Layout<T>
                     ... Page
...```

Aside from Object, all the classes in this abbreviated class hierarchy are implemented in the Xamarin.Forms.Core.dll assembly and associated with a namespace of Xamarin.Forms.

Let’s examine some of these major classes in detail.

As the name of the BindableObject class implies, the primary function of this class is to support data binding—the linking of two properties of two objects so that they maintain the same value. But BindableObject also supports styles and the DynamicResource markup extension as well. It does this in two ways: through BindableObject property definitions in the form of BindableProperty objects and also by implementing the .NET INotifyPropertyChanged interface. All of this will be discussed in much more detail in this chapter and future chapters.

Let’s continue down the hierarchy: as you’ve seen, user-interface objects in Xamarin.Forms are often arranged on the page in a parent-child hierarchy, and the Element class includes support for parent and child relationships.

VisualElement is an exceptionally important class in Xamarin.Forms. A visual element is anything in Xamarin.Forms that occupies an area on the screen. The VisualElement class defines 28 public properties related to size, location, background color, and other visual and functional characteristics, such as IsEnabled and IsVisible.

In Xamarin.Forms the word view is often used to refer to individual visual objects such as buttons, sliders, and text-entry boxes, but you can see that the View class is the parent to the layout classes as well. Interestingly, View adds only three public members to what it inherits from VisualElement. These are HorizontalOptions and VerticalOptions—which make sense because these properties don’t apply to pages—and GestureRecognizers to support touch input.

The descendants of Layout are capable of having children views. A child view appears on the screen visually within the boundaries of its parent. Classes that derive from Layout can have only one child of type View, but the generic Layout<T> class defines a Children property, which is a collection of multiple child views, including other layouts. You’ve already seen the StackLayout, which arranges its children in a horizontal or vertical stack. Although the Layout class derives from View, layouts are so important in Xamarin.Forms that they are often considered a category in themselves.

ClassHierarchy lists all the public classes, structures, and enumerations defined in the Xamarin.Forms.Core and Xamarin.Forms.Xaml assemblies, but it does not list interfaces. Those are important as well, but you’ll just have to explore them on your own. (Or enhance the program to list them.)

Nor does ClassHierarchy list the many public classes that help implement Xamarin.Forms on the various platforms. In the final chapter of this book, you’ll see a version that does.
A peek into BindableObject and BindableProperty

The existence of classes named BindableObject and BindableProperty is likely to be a little confusing at first. Keep in mind that BindableObject is much like Object in that it serves as a base class to a large chunk of the Xamarin.Forms API, and particularly to Element and hence VisualElement.

BindableObject provides support for objects of type BindableProperty. A BindableProperty object extends a CLR property. The best insights into bindable properties come when you create a few of your own—as you’ll be doing before the end of this chapter—but you can also glean some understanding by exploring the existing bindable properties.

Toward the beginning of Chapter 7, “XAML vs. code,” two buttons were created with many of the same property settings, except that the properties of one button were set in code using the C# 3.0 object initialization syntax and the other button was instantiated and initialized in XAML.

Here’s a similar (but code-only) program named PropertySettings that also creates and initializes two buttons in two different ways. The properties of the first Label are set the old-fashioned way, while the properties of the second Label are set with a more verbose technique:

```csharp
public class PropertySettingsPage : ContentPage
{
    public PropertySettingsPage()
    {
        Label label1 = new Label();
        label1.Text = "Text with CLR properties";
        label1.IsVisible = true;
        label1.Opacity = 0.75;
        label1.HorizontalTextAlignment = TextAlignment.Center;
        label1.VerticalOptions = LayoutOptions.CenterAndExpand;
        label1.TextColor = Color.Blue;
        label1.BackgroundColor = Color.FromRgb(255, 128, 128);
        label1.FontSize = Device.GetNamedSize(NamedSize.Medium, new Label());
        label1.FontAttributes = FontAttributes.Bold | FontAttributes.Italic;

        Label label2 = new Label();
        label2.SetValue(Label.TextProperty, "Text with bindable properties");
        label2.SetValue(Label.IsVisibleProperty, true);
        label2.SetValue(Label.OpacityProperty, 0.75);
        label2.SetValue(Label.HorizontalAlignmentProperty, TextAlignment.Center);
        label2.SetValue(Label.VerticalOptionsProperty, LayoutOptions.CenterAndExpand);
        label2.SetValue(Label.TextColorProperty, Color.Blue);
        label2.SetValue(Label.BackgroundColorProperty, Color.FromRgb(255, 128, 128));
        label2.SetValue(Label.FontAttributesProperty, FontAttributes.Bold | FontAttributes.Italic);

        Content = new StackLayout
        {
```
Children =
{
    label1,
    label2
};
}

These two ways to set properties are entirely consistent:

Yet the alternative syntax seems very odd. For example:

```
label2.SetValue(Label.TextProperty, "Text with bindable properties");
```

What is that `SetValue` method? `SetValue` is defined by `BindableObject`, from which every visual object derives. `BindableObject` also defines a `GetValue` method.

That first argument to `SetValue` has the name `Label.TextProperty`, which indicates that `TextProperty` is static, but despite its name, it’s not a property at all. It’s a static `field` of the `Label` class. `TextProperty` is also read-only, and it’s defined in the `Label` class something like this:

```
public static readonly BindableProperty TextProperty;
```

That’s an object of type `BindableProperty`. Of course, it may seem a little disturbing that a field is named `TextProperty`, but there it is. Because it’s static, however, it exists independently of any `Label` objects that might or might not exist.

If you look in the documentation of the `Label` class, you’ll see that it defines 10 properties, including `Text, TextColor, FontSize, FontAttributes,` and others. You’ll also see 10 corresponding
public static read-only fields of type BindableProperty with the names TextProperty, TextColorProperty, FontSizeProperty, FontAttributesProperty, and so forth.

These properties and fields are closely related. Indeed, internal to the Label class, the Text CLR property is defined like this to reference the corresponding TextProperty object:

```csharp
public string Text
{
    set { SetValue(Label.TextProperty, value); }
    get { return (string)GetValue(Label.TextProperty); }
}
```

So you see why it is that your application calling SetValue with a Label.TextProperty argument is exactly equivalent to setting the Text property directly, and perhaps just a tinier bit faster!

The internal definition of the Text property in Label isn’t secret information. This is standard code. Although any class can define a BindableProperty object, only a class that derives fromBindableObject can call the SetValue and GetValue methods that actually implement the property in the class. Casting is required for the GetValue method because it’s defined as returning object.

All the real work involved with maintaining the Text property is going on in those SetValue and GetValue calls. The BindableObject and BindableProperty objects effectively extend the functionality of standard CLR properties to provide systematic ways to:

- Define properties
- Give properties default values
- Store their current values
- Provide mechanisms for validating property values
- Maintain consistency among related properties in a single class
- Respond to property changes
- Trigger notifications when a property is about to change and has changed
- Support data binding
- Support styles
- Support dynamic resources

The close relationship of a property named Text with a BindableProperty named TextProperty is reflected in the way that programmers speak about these properties: Sometimes a programmer says that the Text property is “backed by” a BindableProperty named TextProperty because TextProperty provides infrastructure support for Text. But a common shortcut is to say that Text is itself a “bindable property,” and generally no one will be confused.
Not every Xamarin.Forms property is a bindable property. Neither the Content property of ContentPage nor the Children property of Layout<T> is a bindable property. Of the 28 properties defined by VisualElement, 26 are backed by bindable properties, but the Bounds property and the Resources properties are not.

The Span class used in connection with FormattedString does not derive from BindableObject. Therefore, Span does not inherit SetValue and GetValue methods, and it cannot implement BindableProperty objects.

This means that the Text property of Label is backed by a bindable property, but the Text property of Span is not. Does it make a difference?

Of course it makes a difference! If you recall the DynamicVsStatic program in the previous chapter, you discovered that DynamicResource worked on the Text property of Label but not the Text property of Span. Can it be that DynamicResource works only with bindable properties?

This supposition is pretty much confirmed by the definition of the following public method defined by Element:

```csharp
public void SetDynamicResource(BindableProperty property, string key);
```

This is how a dictionary key is associated with a particular property of an element when that property is the target of a DynamicResource markup extension.

This SetDynamicResource method also allows you to set a dynamic resource link on a property in code. Here’s the page class from a code-only version of DynamicVsStatic called DynamicVsStatic-Code. It’s somewhat simplified to exclude the use of a FormattedString and Span object, but otherwise it pretty accurately mimics how the previous XAML file is parsed and, in particular, how the Text properties of the Label elements are set by the XAML parser:

```csharp
public class DynamicVsStaticCodePage : ContentPage
{
    public DynamicVsStaticCodePage()
    {
        Padding = new Thickness(5, 0);

        // Create resource dictionary and add item.
        Resources = new ResourceDictionary
        {
            { "currentDateTime", "Not actually a DateTime" }
        };

        Content = new StackLayout
        {
            Children =
            {
                new Label
                {
                    Text = "StaticResource on Label.Text:",
                    VerticalOptions = LayoutOptions.EndAndExpand,
                },
            },
        };
    }
}
```
The Text property of the second Label is set directly from the dictionary entry and makes the use of the dictionary seem a little pointless in this context. But the Text property of the last Label is bound to the dictionary key through a call to SetDynamicResource, which allows the property to be updated when the dictionary contents change:
Consider this: What would the signature of this `SetDynamicResource` method be if it could not refer to a property using the `BindableProperty` object? It’s easy to reference a property value in method calls, but not the property itself. There are a couple of ways, such as the `PropertyInfo` class in the `System.Reflection` namespace or the LINQ Expression object. But the `BindableProperty` object is designed specifically for this purpose, as well as the essential job of handling the underlying link between the property and the dictionary key.

Similarly, when we explore styles in the next chapter, you’ll encounter a `Setter` class used in connection with styles. `Setter` defines a property named `Property` of type `BindableProperty`, which mandates that any property targeted by a style must be backed by a bindable property. This allows a style to be defined prior to the elements targeted by the style.

Likewise for data bindings. The `BindableObject` class defines a `SetBinding` method that is very similar to the `SetDynamicResource` method defined on `Element`:

```csharp
public void SetBinding(BindableProperty targetProperty, BindingBase binding);
```

Again, notice the type of the first argument. Any property targeted by a data binding must be backed by a bindable property.

For these reasons, whenever you create a custom view and need to define public properties, your default inclination should be to define them as bindable properties. Only if after careful consideration you conclude that it is not necessary or appropriate for the property to be targeted by a style or a data binding should you retreat and define an ordinary CLR property instead.
So whenever you create a class that derives from `BindableObject`, one of the first pieces of code you should be typing in that class begins “public static readonly BindableProperty”—perhaps the most characteristic sequence of four words in all of Xamarin.Forms programming.

**Defining bindable properties**

Suppose you’d like an enhanced `Label` class that lets you specify font sizes in units of points. Let’s call this class `AltLabel` for “alternative Label.” It derives from `Label` and includes a new property named `PointSize`.

Should `PointSize` be backed by a bindable property? Of course! (Although the real advantages of doing so won’t be demonstrated until upcoming chapters.)

The code-only `AltLabel` class is included in the `Xamarin.FormsBook.Toolkit` library, so it’s accessible to multiple applications. The new `PointSize` property is implemented with a `BindableProperty` object named `PointSizeProperty` and a CLR property named `PointSize` that references `PointSizeProperty`:

```csharp
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty ... ;
    ...
    public double PointSize
    {
        set { SetValue(PointSizeProperty, value); }
        get { return (double)GetValue(PointSizeProperty); }
    }
    ...
}
```

Both the field and the property definition must be public.

Because `PointSizeProperty` is defined as static and readonly, it must be assigned either in a static constructor or right in the field definition, after which it cannot be changed. Generally, a `BindableProperty` object is assigned in the field definition by using the static `BindableProperty.Create` method. Four arguments are required (shown here with the argument names):

- `propertyName` The text name of the property (in this case “PointSize”)
- `returnType` The type of the property (a `double` in this example)
- `declaringType` The type of the class defining the property (`AltLabel`)
- `defaultValue` A default value (let’s say 8 points)

The second and third arguments are generally defined with `typeof` expressions. Here’s the assignment statement with these four arguments passed to `BindableProperty.Create`:
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty =
        BindableProperty.Create("PointSize", typeof(double),
                                typeof(AltLabel), 8.0);  

    ...  
}

Notice that the default value is specified as 8.0 rather than just 8. Because BindableProperty.Create is designed to handle properties of any type, the defaultValue parameter is defined as object. When the C# compiler encounters just an 8 as that argument, it will assume that the 8 is an int and pass an int to the method. The problem won’t be revealed until run time, however, when the BindableProperty.Create method will be expecting the default value to be of type double and respond by raising a TypeInitializationException.

You must be explicit about the type of the value you’re specifying as the default. Not doing so is a very common error in defining bindable properties. A very common error.

BindableProperty.Create also has six optional arguments. Here they are with the argument names and their purpose:

- defaultBindingMode Used in connection with data binding
- validateValue A callback to check for a valid value
- propertyChanged A callback to indicate when the property has changed
- propertyChanging A callback to indicate when the property is about to change
- coerceValue A callback to coerce a set value to another value (for example, to restrict the values to a range)
- defaultValueCreator A callback to create a default value. This is generally used to instantiate a default object that can’t be shared among all instances of the class; for example, a collection object such as List or Dictionary.

Do not perform any validation, coercion, or property-changed handling in the CLR property. The CLR property should be restricted to SetValue and GetValue calls. Everything else should be done in the callbacks provided by the bindable property infrastructure.

It is very rare that a particular call to BindableProperty.Create would need all of these optional arguments. For that reason, these optional arguments are commonly indicated with the named argument feature introduced in C# 4.0. To specify a particular optional argument, use the argument name followed by a colon. For example:
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty =
        BindableProperty.Create("PointSize", typeof(double),
            typeof(AltLabel), 8.0, propertyChanged: OnPointSizeChanged);

    ...}

Without a doubt, propertyChanged is the most important of the optional arguments because the class uses this callback to be notified when the property changes, either directly from a call to SetValue or through the CLR property.

In this example, the property-changed handler is called OnPointSizeChanged. It will be called only when the property truly changes and not when it’s simply set to the same value. However, because OnPointSizeChanged is referenced from a static field, the method itself must also be static. Here’s what it looks like:

public class AltLabel : Label
{
    ...
    static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
    {
        ...
    }
    ...
}

This seems a little odd. We might have multiple AltLabel instances in a program, yet whenever the PointSize property changes in any one of these instances, this same static method is called. How does the method know exactly which AltLabel instance has changed?

The method can tell which instance’s property has changed because that instance is always the first argument to the property-changed handler. Although that first argument is defined as a BindableObject, in this case it’s actually of type AltLabel and indicates which AltLabel instance’s property has changed. This means that you can safely cast the first argument to an AltLabel instance:

static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
{
    AltLabel altLabel = (AltLabel)bindable;
    ...
}

You can then reference anything in the particular instance of AltLabel whose property has changed. The second and third arguments are actually of type double for this example and indicate the previous value and the new value.

Often it’s convenient for this static method to call an instance method with the arguments converted to their actual types:
public class AltLabel : Label
{
    ...
    static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
    {
        ((AltLabel)bindable).OnPointSizeChanged((double)oldValue, (double)newValue);
    }
    
    void OnPointSizeChanged(double oldValue, double newValue)
    {
        ...
    }
}

The instance method can then make use of any instance properties or methods of the underlying base class as it would normally.

For this class, this OnPointSizeChanged method needs to set the FontSize property based on the new point size and a conversion factor. In addition, the constructor needs to initialize the FontSize property based on the default PointSize value. This is done through a simple SetLabelFontSize method. Here’s the final complete class:

```csharp
public class AltLabel : Label
{
    public static readonly BindableProperty PointSizeProperty =
        BindableProperty.Create("PointSize", typeof(double), typeof(AltLabel), 8.0, propertyChanged: OnPointSizeChanged);

    public AltLabel()
    {
        SetLabelFontSize((double)PointSizeProperty.DefaultValue);
    }

    public double PointSize
    {
        set { SetValue(PointSizeProperty, value); }
        get { return (double)GetValue(PointSizeProperty); }
    }

    static void OnPointSizeChanged(BindableObject bindable, object oldValue, object newValue)
    {
        ((AltLabel)bindable).OnPointSizeChanged((double)oldValue, (double)newValue);
    }

    void OnPointSizeChanged(double oldValue, double newValue)
    {
        SetLabelFontSize(newValue);
    }

    void SetLabelFontSize(double pointSize)
```
{  
    FontSize = 160 * pointSize / 72;  
}

It is also possible for the instance OnPointSizeChanged property to access the PointSize property directly rather than use newValue. By the time the property-changed handler is called, the underlying property value has already been changed. However, you don't have direct access to that underlying value, as you do when a private field backs a CLR property. That underlying value is private to Binda-
bleObject and accessible only through the GetValue call.

Of course, nothing prevents code that's using AltLabel from setting the FontSize property and overriding the PointSize setting, but let's hope such code is aware of that. Here's some code that is—a program called PointSizedText, which uses AltLabel to display point sizes from 4 through 12:

```csharp
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            x:Class="PointSizedText.PointSizedTextPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                   iOS="5, 20, 0, 0"
                   Android="5, 0, 0, 0"
                   WinPhone="5, 0, 0, 0" />
    </ContentPage.Padding>

    <StackLayout x:Name="stackLayout">
        <toolkit:AltLabel Text="Text of 4 points" PointSize="4" />
        <toolkit:AltLabel Text="Text of 5 points" PointSize="5" />
        <toolkit:AltLabel Text="Text of 6 points" PointSize="6" />
        <toolkit:AltLabel Text="Text of 7 points" PointSize="7" />
        <toolkit:AltLabel Text="Text of 8 points" PointSize="8" />
        <toolkit:AltLabel Text="Text of 9 points" PointSize="9" />
        <toolkit:AltLabel Text="Text of 10 points" PointSize="10" />
        <toolkit:AltLabel Text="Text of 11 points" PointSize="11" />
        <toolkit:AltLabel Text="Text of 12 points" PointSize="12" />
    </StackLayout>
</ContentPage>
```

And here are the screenshots:
The read-only bindable property

Suppose you’re working with an application in which it’s convenient to know the number of words in the text that is displayed by a Label element. Perhaps you’d like to build that facility right into a class that derives from Label. Let’s call this new class CountedLabel.

By now, your first thought should be to define a BindableProperty object named WordCount-Property and a corresponding CLR property named WordCount.

But wait: It only makes sense for this WordCount property to be set from within the CountedLabel class. That means the WordCount CLR property should not have a public set accessor. It should be defined this way:

```csharp
public int WordCount
{
    private set { SetValue(WordCountProperty, value); }
    get { return (double)GetValue(WordCountProperty); }
}
```

The get accessor is still public, but the set accessor is private. Is that sufficient?

Not exactly. Despite the private set accessor in the CLR property, code external to CountedLabel can still call SetValue with the CountedLabel.WordCountProperty bindable property object. That type of property setting should be prohibited as well. But how can that work if the WordCountProperty object is public?

The solution is to make a read-only bindable property by using the BindableProperty.CreateReadOnly method. The
Xamarin.Forms API itself defines several read-only bindable properties—for example, the Width and Height properties defined by VisualElement.

Here’s how you can make one of your own:

The first step is to call `BindableProperty.CreateReadOnly` with the same arguments as for `BindableProperty.Create`. However, the `CreateReadOnly` method returns an object of `BindablePropertyKey` rather than `BindableProperty`. Define this object as static and readonly, as with the `BindableProperty`, but make it be private to the class:

```csharp
public class CountedLabel : Label
{
    static readonly BindablePropertyKey WordCountKey =
        BindableProperty.CreateReadOnly("WordCount", typeof(int),
                                        typeof(CountedLabel), 0);
    ...
}
```

Don’t think of this `BindablePropertyKey` object as an encryption key or anything like that. It’s much simpler—really just an object that is private to the class.

The second step is to make a public `BindableProperty` object by using the `BindableProperty` property of the `BindablePropertyKey`:

```csharp
public class CountedLabel : Label
{
    public static readonly BindableProperty WordCountProperty = WordCountKeyBindableProperty;
    ...
}
```

This `BindableProperty` object is public, but it’s a special kind of `BindableProperty`: It cannot be used in a `SetValue` call. Attempting to do so will raise an `InvalidOperationException`.

However, there is an overload of the `SetValue` method that accepts a `BindablePropertyKey` object. The CLR set accessor can call `SetValue` using this object, but this set accessor must be private to prevent the property from being set outside the class:

```csharp
public class CountedLabel : Label
{
    public int WordCount
    {
        private set { SetValue(WordCountKey, value); }
        get { return (int)GetValue(WordCountProperty); }
    }
    ...
}
```

The `WordCount` property can now be set from within the `CountedLabel` class. But when should the
class set it? This CountedLabel class derives from Label, but it needs to detect when the Text property has changed so that it can count up the words.

Does Label have a TextChanged event? No it does not. However, BindableObject implements the INotifyPropertyChanged interface. This is a very important .NET interface, particularly for applications that implement the Model-View-ViewModel (MVVM) architecture. In Chapter 18 you’ll see how to use it in your own data classes.

The INotifyPropertyChanged interface is defined in the System.ComponentModel namespace like so:

```csharp
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

Every class that derives from BindableObject automatically fires this PropertyChanged event whenever any property backed by a BindableProperty changes. The PropertyChangedEventArgs object that accompanies this event includes a property named PropertyName of type string that identifies the property that has changed.

So all that’s necessary is for CountedLabel to attach a handler for the PropertyChanged event and check for a property name of “Text”. From there it can use whatever technique it wants for calculating a word count. The complete CountedLabel class uses a lambda function on the PropertyChanged event. The handler calls Split to break the string into words and see how many pieces result. The Split method splits the text based on spaces, dashes, and em dashes (Unicode \u2014):

```csharp
public class CountedLabel : Label
{
    static readonly BindablePropertyKey WordCountKey =
        BindableProperty.CreateReadOnly("WordCount", typeof(int), typeof(CountedLabel), 0);

    public static readonly BindableProperty WordCountProperty = WordCountKeyBindableProperty;

    public CountedLabel()
    {
        // Set the WordCount property when the Text property changes.
        PropertyChanged += (object sender, PropertyChangedEventArgs args) =>
        {
            if (args.PropertyName == "Text")
            {
                if (String.IsNullOrEmpty(Text))
                {
                    WordCount = 0;
                }
                else
                {
                    WordCount = Text.Split(' ', '-', '\u2014').Length;
                }
            }
        }
    }
}
```

Because this call of the Split method splits the text at blank characters, dashes, and em dashes, you might assume that CountedLabel will be demonstrated with text that contains some dashes and em dashes. This is true. The BaskervillesCount program is a variation of the Baskervilles program from Chapter 3, but here the paragraph of text is displayed with a CountedLabel, and a regular Label is included to display the word count:

```xml
x:Class="BaskervillesCount.BaskervillesCountPage"
Padding="5, 0">
  <StackLayout>
    <toolkit:CountedLabel x:Name="countedLabel"
      VerticalOptions="CenterAndExpand"
      Text="Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the stick which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a Penang lawyer. Just under the head was a broad silver band, nearly an inch across, To James Mortimer, M.R.C.S., from his friends of the C.C.H., was engraved upon it, with the date 1884. It was just such a stick as the old-fashioned family practitioner used to carry; dignified, solid, and reassuring."
    />
    <Label x:Name="wordCountLabel"
      Text="???
```
That regular Label is set in the code-behind file:

```csharp
public partial class BaskervillesCountPage : ContentPage
{
    public BaskervillesCountPage()
    {
        InitializeComponent();

        int wordCount = countedLabel.WordCount;
        wordCountLabel.Text = wordCount + " words";
    }
}
```

The word count that it calculates is based on the assumption that all hyphens in the text separate two words and that “hearth-rug” and “bulbous-headed” should be counted as two words each. That’s not always true, of course, but word counts are not quite as algorithmically simple as this code might imply:

How would the program be structured if the text changed dynamically while the program was running? In that case, it would be necessary to update the word count whenever the WordCount property of the CountedLabel object changed. You could attach a PropertyChanged handler on the CountedLabel object and check for the property named “WordCount”.

Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the stick which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer," just under the head was a broad silver band, nearly an inch across. "To James Moriarty, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry—dignified, solid, and reassuring.

Mr. Sheriuck Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the sticks which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer," just under the head was a broad silver band, nearly an inch across. "To James Moriarty, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry—dignified, solid, and reassuring.

Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those not infrequent occasions when he was up all night, was seated at the breakfast table. I stood upon the hearth-rug and picked up the stick which our visitor had left behind him the night before. It was a fine, thick piece of wood, bulbous-headed, of the sort which is known as a "Penang lawyer," just under the head was a broad silver band, nearly an inch across. "To James Moriarty, M.R.C.S., from his friends of the C.C.H.," was engraved upon it, with the date "1884." It was just such a stick as the old-fashioned family practitioner used to carry—dignified, solid, and reassuring.

120 words

120 words

120 words
However, exercise caution if you try to set such an event handler from XAML—for example, like so:

```xml
<toolkit:CountedLabel x:Name="countedLabel"
    VerticalOptions="CenterAndExpand"
    PropertyChanged="OnCountedLabelPropertyChanged"
    Text="..." />
```

You’ll probably want to code the event handler in the code-behind file like this:

```csharp
void OnCountedLabelPropertyChanged(object sender,
    PropertyChangedEventArgs args)
{
    wordCountLabel.Text = countedLabel.WordCount + " words";
}
```

That handler will fire when the Text property is set by the XAML parser, but the event handler is trying to set the Text property of the second Label, which hasn’t been instantiated yet, which means that the wordCountLabel field is still set to null. This is an issue that will come up again in Chapter 15 when working with interactive controls, but it will be pretty much solved when we work with data binding in Chapter 16.

There is another variation of a bindable property coming up in Chapter 14 on the AbsoluteLayout: this is the attached bindable property, and it is very useful in implementing certain types of layouts, as you’ll also discover in Chapter 26, “Custom layouts.”

Meanwhile, let’s look at one of the most important applications of bindable properties: styles.