Programming for mobiles

theory;
different OS;
frameworks;
good practices.

prof. O. Nakov, Ph.D

Windows Phone

(2)

Types of Applications

The Windows Phone application platform provides two frameworks for developing applications:

Silverlight

The Silverlight framework supports an event-driven, XAML-based application development.

XNA

The XNA Framework supports loop-based games.

The following table lists some of the criteria that you can use to determine whether you should use Silverlight or the XNA Framework for your Windows Phone application.

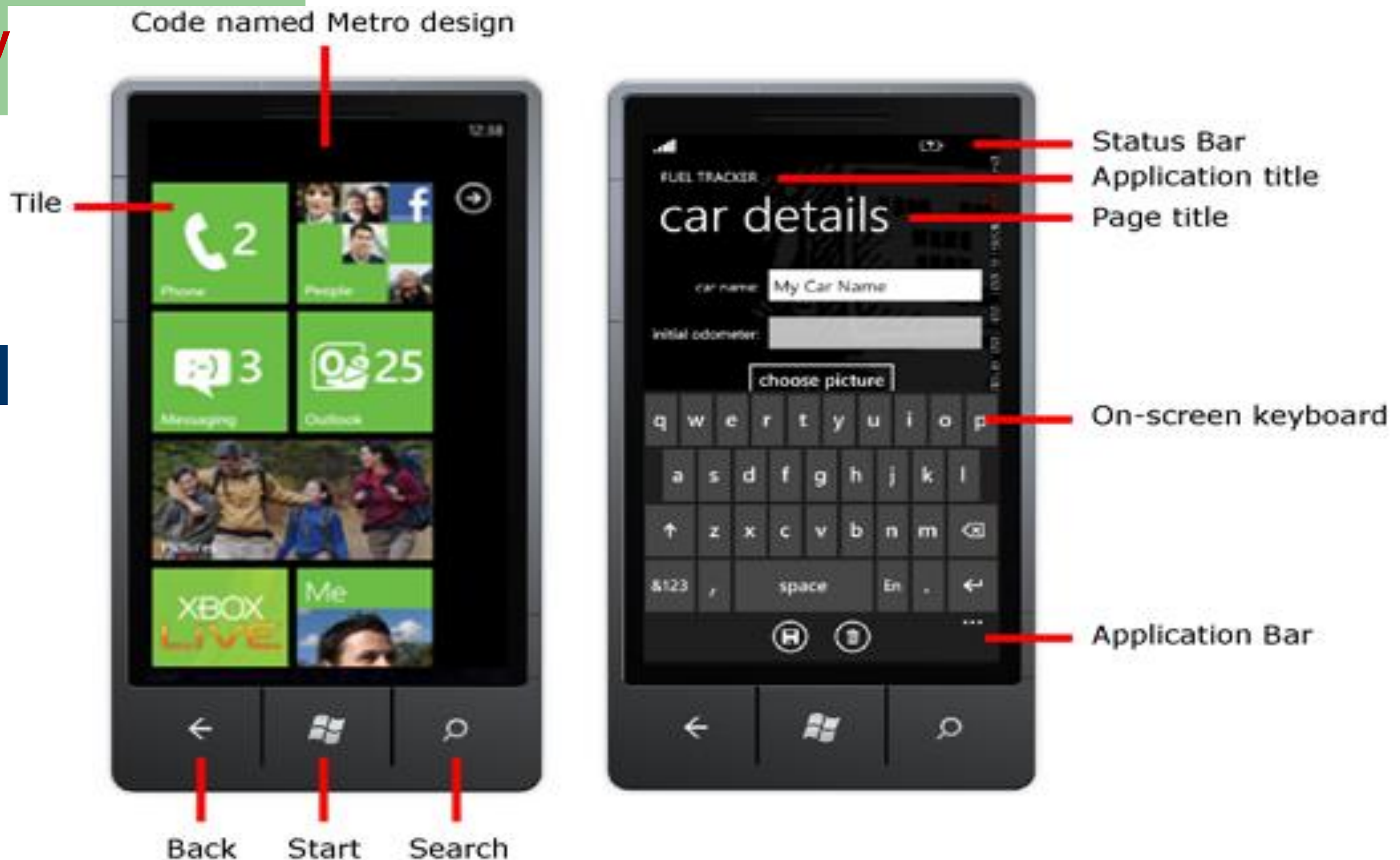
Criteria	Framework
Text-based controls and menus	Silverlight
Event-driven application	Silverlight
Interaction with Windows Phone controls such as Panorama	Silverlight
Embedded video	Silverlight
Hosted HTML	Silverlight
Web browser compatibility	Silverlight
Vector graphics	Silverlight
Looping game framework	XNA
Visually complex applications	XNA
3D games	XNA
Advanced art assets such as textures, effects, and terrains	XNAX

Hardware

Windows Phone 7 have a minimum hardware requirement that make it easier for developers to write applications.
Each Windows Phone 7 contains the following hardware elements:

WVGA (800 x 480) format display.
Capacitive multi-touch screen.
DirectX 9 hardware acceleration.
Sensors for A-GPS, accelerometer, compass, light, and proximity.
Digital camera.
Start, Search, and Back buttons.
Support for data connectivity using cellular networks and Wi-Fi.
256 MB (or more) of RAM and 8 GB (or more) of flash storage.

Terminology



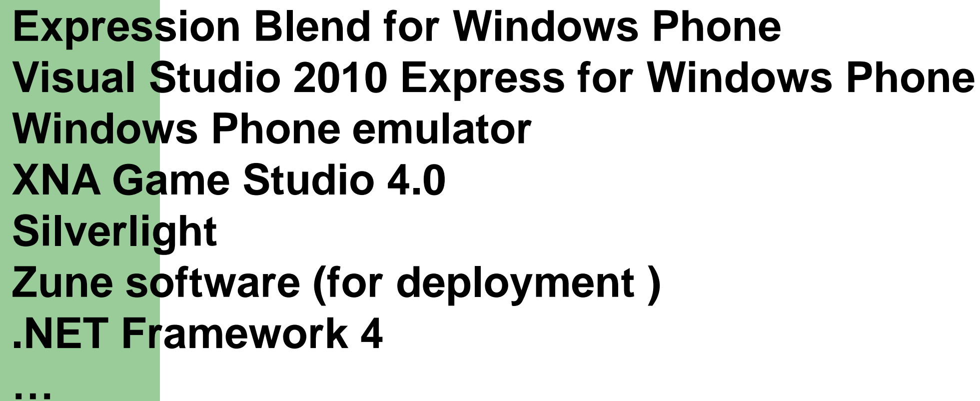
Code named Metro design: The user interface (UI) used in Windows Phone. You should follow this design in your applications so that they integrate with the operating system and other applications. The design provides a modern UI that is easy to use, while minimizing power consumption on the phone.

Tile: A representation of an application that appears in the start screen. A tile can be designed to be dynamic and display information to the user.

Status Bar: Indicates status of phone operations, such as signal strength. Not necessarily application specific.

Tools for Creating Applications

When you install the Windows Phone Developer Tools, you get the following free tools and components.



Expression Blend for Windows Phone
Visual Studio 2010 Express for Windows Phone
Windows Phone emulator
XNA Game Studio 4.0
Silverlight
Zune software (for deployment)
.NET Framework 4
…

If you already have Visual Studio 2010 (Professional or Ultimate) installed, then you can use Visual Studio 2010 for development after installing the Windows Phone Developer Tools.

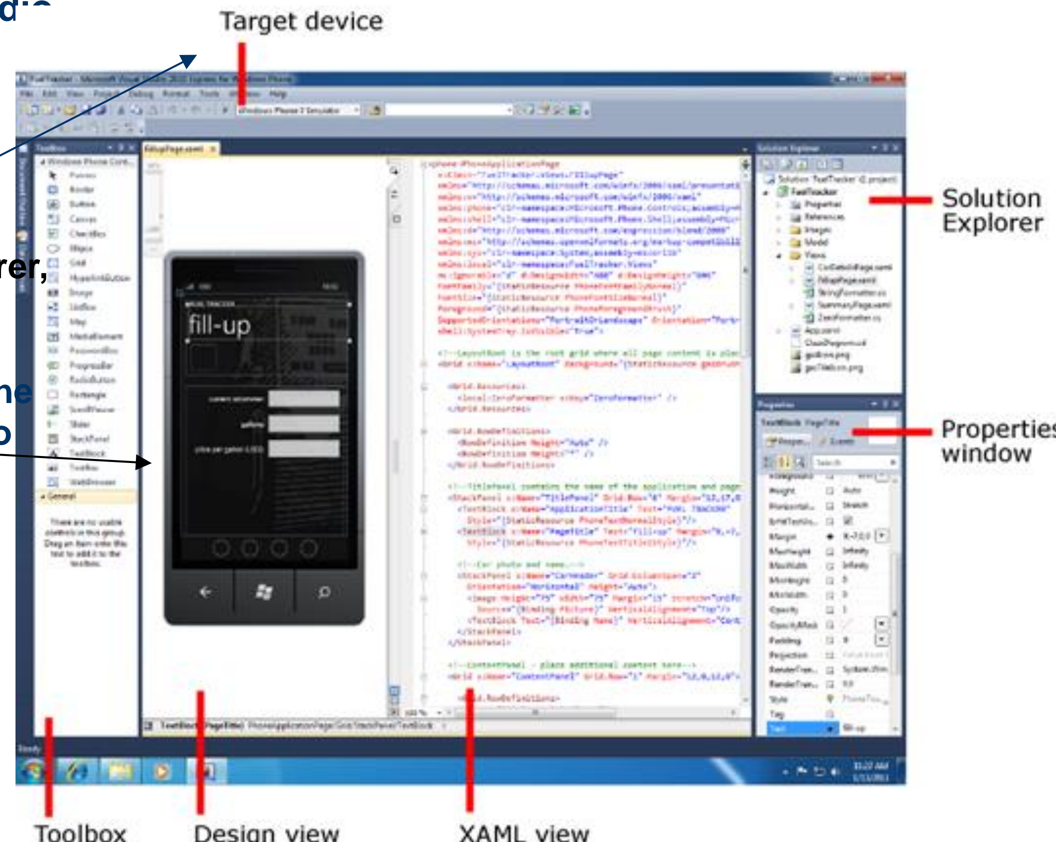
Expression Blend for Windows Phone

Expression Blend for Windows Phone is a design suite that allows you to create and add special visual features, such as gradients, animations, and transitions. For some tasks, Expression Blend is easier to use than Visual Studio

Visual Studio 2010 Express for Windows Phone

Visual Studio 2010 Express for Windows Phone includes a drag-and-drop designer that emulates the appearance of the phone, a code editor, and a debugger.

The following illustration shows the Visual Studio 2010 Express environment for the phone:



The designer for Windows Phone contains the **Toolbox**, **Design view**, **XAML view**, **Solution Explorer**, and the **Properties window** similar to the Visual Studio designer. Two key differences are that the design surface looks like a Windows Phone, and the addition of the Target device, which enables you to choose whether you debug your application on a device or the emulator.

Creating a Windows Phone 7 Application

(SILVERLIGHT QUICKSTART FOR WINDOWS PHONE DEVELOPMENT)

A link to file:
File_1 First program

1. Creating A New Project

After you've installed the Windows Phone Developer Tools, the easiest way to create your first application is to use Visual Studio:

- On the Start menu, launch Microsoft Visual Studio 2010 Express for Windows Phone.

- On the File menu, click New Project

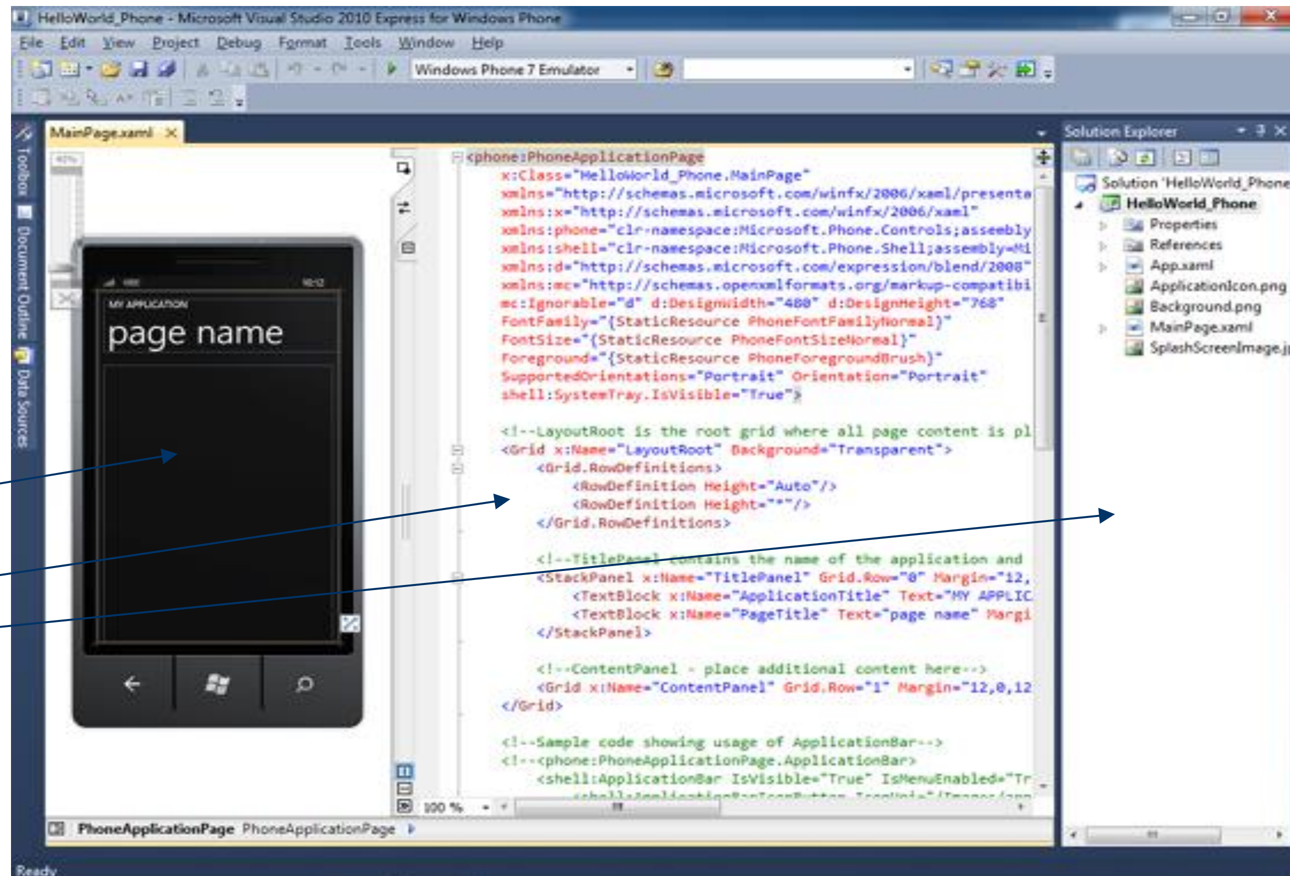
2. select Silverlight for Windows Phone.

-select the Windows Phone Application template.

-Name the project HelloWorld_Phone

A new Silverlight for Windows Phone project is created and opened in the designer.

On the left is the Design view, in the middle is the XAML view, and on the right is Solution Explorer.



A link to file:

File_1 First program

A link to file:

File_2 XAML

A link to file:

File_3 Creation an Application

****

For programmers, Windows Phone 7 supports two popular and modern programming platforms: Silverlight and XNA.

Silverlight has already given Web programmers power to develop sophisticated user interfaces with a mix of traditional controls, high-quality text, vector graphics, media, animation, and data binding that run on multiple platforms and browsers.

Windows Phone extends Silverlight to mobile devices.

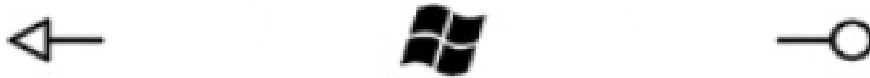
XNA—the three letters stand for something like "XNA is Not an Acronym"—is Microsoft's game platform supporting both 2D sprite-based and 3D graphics with a traditional game-loop architecture. Although XNA is mostly associated with writing games for the Xbox 360 console, developers can also use XNA to target the PC itself, as well as Microsoft's classy audio player, the Zune HD.

******

Generally you'll choose Silverlight for writing programs you might classify as applications or utilities. These programs are built from a combination of markup and code. The markup is the Extensible Application Markup Language, or XAML and pronounced "zammel." The XAML mostly defines a layout of user-interface controls and panels. Code-behind files can also perform some initialization and logic, but are generally relegated to handling events from the controls

The 'hardware'

The front of the phone consists of a multi-touch display and three hardware buttons generally positioned in a row below the display. From left to right, these buttons are called Back, Start, and Search:

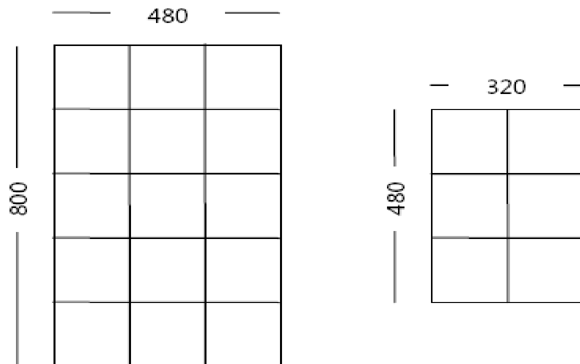


Back **Programs can use this button for their own navigation needs, much like the Back button on a Web browser. From the home page of a program, the button causes the program to terminate.**

Start **This button takes the user to the start screen of the phone.**

Search **The operating system uses this button to initiate a search feature.**

Screen: **The initial releases of Windows Phone 7 devices have a display size of 480 × 800 pixels. Screens of 320 × 480 pixels are also available. We will generally refer to these two sizes as the "large" screen and the "small" screen. The greatest common denominator of the horizontal and vertical dimensions of both screens is 160, so you can visualize the two screens as multiples of 160-pixel squares:**



Of course, phones can be rotated to put the screen into landscape mode. Some programs might require the phone to be held in a certain orientation; others are more adaptable.

The Software:

The main files are: **App.xaml** and **App.xaml.cs**,

MainPage.xaml defines the user interface for the application.
A C# code-behind file is named MainPage.xaml.cs.

In the standard Visual Studio toolbar under the program's menu, you'll see a drop-down list probably displaying "Windows Phone .. Emulator." The other choice is "Windows Phone .. Device." This is how you deploy your program to either the emulator or an actual phone connected to your computer via USB.

Silverlight Project: Helloworld_Phone File: **App.xaml.cs :**

```
namespace Helloworld_Phone
{ public  class App : Application {
public App()
{ …
  InitializeComponent();
… } …
}
```

Silverlight Project: Helloworld_Phone File: **App.xaml :**

An URI of MS where all Silverlight declarations are

Declaration of XAML elements

```
<Application
  x:Class="Helloworld_Phone.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
xmlns:phone="clr-namespace:Microsoft.Phone.Controls;assembly=Microsoft.Phone"
xmlns:shell="clr-namespace:Microsoft.Phone.Shell;assembly=Microsoft.Phone">
        …
</Application>
```

developers often use the **App.xaml** file for storing *resources* that are used throughout the application. These resources might include color schemes, gradient brushes, styles, and so forth.

The root element is ***Application***, which is the Silverlight class that the *App* class derives from. The root element contains XML namespace declarations.

When a program is run, the ***App* class** declares an

object of type ***PhoneApplicationFrame*** This frame is 480 pixels wide and 800 pixels tall and occupies the entire display surface of the phone. The

***PhoneApplicationFrame* object** then behaves somewhat like a web browser by navigating to

an object called *MainPage*.

****

the phone emulator is on the desktop and you'll see the opening screen, followed soon by this little do-nothing Silverlight program as it is deployed and run on the emulator.

The phone emulator has a little floating menu at the upper right that comes into view when you move the mouse to that location. You can change orientation through this menu, or change the emulator size

Don't exit the emulator itself by clicking the X at the top of the floating menu! Keeping the emulator running will make subsequent deployments go much faster.

****

If you have a Windows Phone 7 device, you'll need to register for the marketplace at the Windows Phone 7 portal, *http://developer.windowsphone.com*. After you're approved, you'll to connect the phone to your PC and run the Zune desktop software. You can unlock the phone for development by running the Windows Phone Developer Registration program and entering your Windows Live ID. You can then deploy programs to the phone from Visual Studio.

MainPage е вторият основен клас на всяка Silverlight програма и се структурира от 2 файла:
MainPage.xaml
MainPage.xaml.cs

За по-малки програми те са и 2-та файла на проекта

Silverlight Project: Helloworld_Phone File: MainPage.xaml.cs :

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using Microsoft.Phone.Controls;

namespace Helloworld_Phone
{ public partial class MainPage : PhoneApplicationPage
{
// Constructor
public MainPage()
            { InitializeComponent(); }

} }
```

we see another *partial* class definition. This one defines a class named *MainPage* that derives from the Silverlight class *PhoneApplicationPage*. This is the class that defines the visuals you'll actually see on the screen when you run the SilverlightHelloPhone program.

Иерархия:

PhoneApplicationFrame
  PhoneApplicationPage
    Grid named "LayoutRoot"
      StackPanel named "TitlePanel"
        TextBlock named ApplicationTitle"
        TextBlock named "PageTitle"
      Grid named "ContentPanel"

```
<phone:PhoneApplicationPage x:Class="Helloworld_Phone.MainPage"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" ..
…

<!--LayoutRoot is the root grid where all page content is placed-->
 <Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>
                <RowDefinition Height="Auto"/>
                <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
<!--TitlePanel contains the name of the application and page title-->
   <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
      <TextBlock x:Name="ApplicationTitle" Text="MY APPLICATION"
             Style="{StaticResource PhoneTextNormalStyle}"/>
      <TextBlock x:Name="PageTitle" Text="page name" Margin="9,-7,0,0"
             Style="{StaticResource PhoneTextTitle1Style}"/>
    </StackPanel>
<!--ContentPanel - place additional content here-->
   <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0"> </Grid>
  </Grid>
</phone:PhoneApplicationPage>
```

Now to
edit the .xaml file:

Adding a TextBlock

Next you'll add a simple TextBlock that displays the message "Hello, World!"
- If MainPage.xaml isn't already open, double-click MainPage.xaml in Solution Explorer.
- On the View menu, click Other Windows → click Toolbox. The Toolbox window appears.
- Resize or pin the Toolbox -you can see both the Toolbox and the phone in Design view.

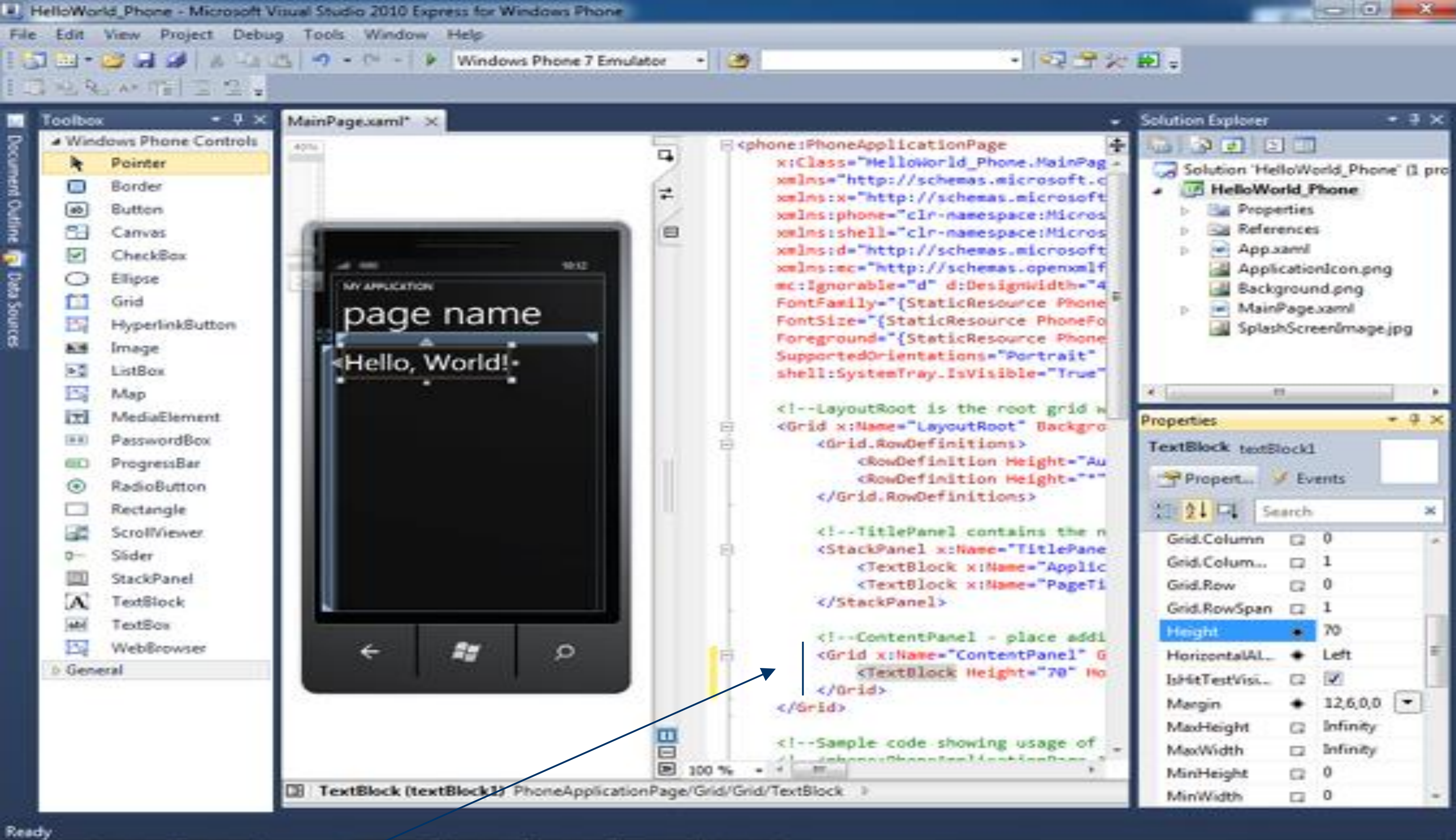
- From the Toolbox, drag a TextBlock on to the main panel of the phone.
In XAML view, notice that *a TextBlock element was added* in the Grid content panel.

MainPage.xaml now include:

```
<Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
<TextBlock Text="Hello, World" HorizontalAlignment="Center"
VerticalAlignment="Center" />
</Grid>
```

-On the View menu, click Other Windows, and then click Properties Window.
The Properties window appears.

Design view updates and should look like the following :



While you're editing MainPage.xaml you might also want to fix the other *TextBlock* elements. Change :

<TextBlock … Text="MY APPLICATION" … />
to
<TextBlock … Text="any new text…" … />
and
<TextBlock … Text="page title" … />
to:
<TextBlock … Text="page name" … />

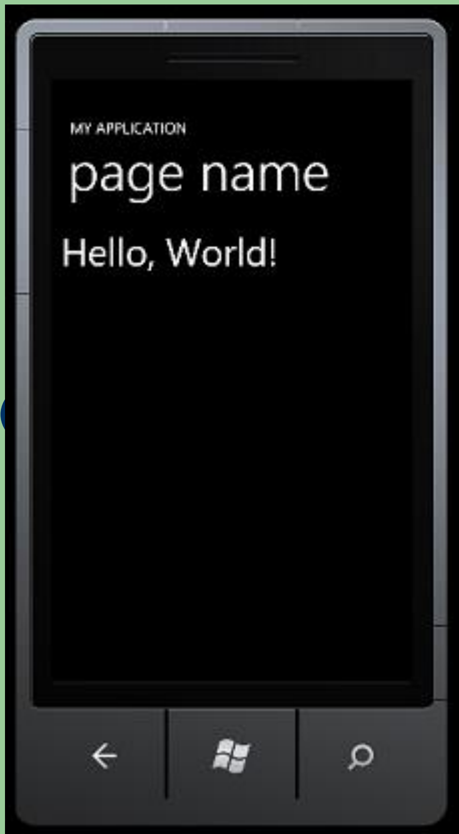
Running the Application

You'll use the built-in Windows Phone emulator, which mimics a Windows Phone device. Using the Windows Phone emulator, you can test and debug your application quickly on the desktop without having to deploy the application to the device.

To start the emulator, you simply need to start a debug session for the application. Visual Studio will launch the emulator and load the application onto it.

To start the application in debug mode,

press F5 or choose Debug->Start Debugging



Remark:

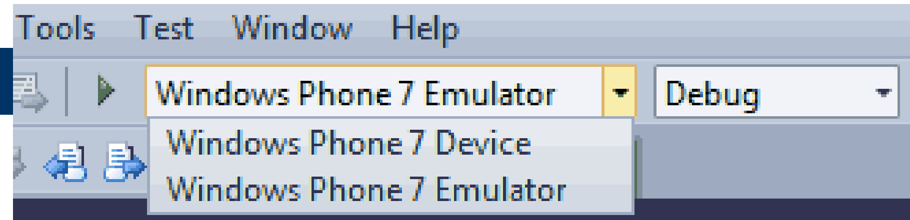
-To run your application on a Windows Phone, you must unlock the device by using the *Windows Phone Developer Registration tool*. This tool is located in the Start Menu under Windows Phone Developer Tools. In addition, you must have a paid App Hub account.

If you don't already have an App Hub account, register now on App Hub, the official Windows Phone developer portal.

- **Start the Zune software on your computer.**
- **Connect the phone to your computer.**
- **Launch the Windows Phone Developer Registration tool, then enter the Windows Live ID credentials associated with your App Hub account**

- In the registration wizard, enter the required identifying information about your phone.
-
- Your phone is unlocked and ready to receive application deployments in Visual Studio.

- In Visual Studio, deploying to the phone is as simple as selecting



"Windows Phone 7 Device" (instead of the emulator) in the deployment target.

FOR A SUCCESSFUL DEPLOYMENT, THE PHONE MUST BE CONNECTED TO THE COMPUTER WITH ITS SCREEN UNLOCKED, AND THE

ZUNE SOFTWARE MUST BE RUNNING.

Points and Pixels

Another property of the *TextBlock* that you can easily change is *FontSize*:

FontSize="36"

All dimensions in Silverlight are in units of pixels, and the *FontSize* is no exception. When you specify 36, you get a font that from the top of its ascenders to the bottom of its descenders measures approximately 36 pixels.

But fonts are never this simple. The resultant *TextBlock* will actually have a height more like 48 pixels—about 33% higher than the *FontSize* would imply. This additional space (called *leading*) prevents successive lines of text from jamming against each other.
Traditionally, font sizes are expressed in units of points. A point is very close to 1/72nd inch.

How do you convert between pixels and points? Obviously you can't except for a particular output device. On a 600 dots-per-inch (DPI) printer, for example, the 72-point font will be 600 pixels tall.

Desktop video displays in common use today usually have a resolution somewhere in the region of 100 DPI.
By default, Microsoft Windows assumes that video displays have a resolution of 96 DPI. Under that assumption, font sizes and pixels are related by the following formulas:

points = 3/4 × pixels
pixels = 4/3 × points

The issue of font size becomes more complex when dealing with high-resolution screens .
The 480 × 800 pixel display has a diagonal of 933 pixels. A phone that has a screen of about 3½" for example, has a diagonal of 264 DPI.
(Screen resolution is usually expressed as a multiple of 24.)

The XAP is a ZIP

If you navigate to the \bin\Debug directory of the Visual Studio project for Helloworld_Phone, you'll find a file named Helloworld_Phone.xap. This is commonly referred to as a
XAP file, pronounced "zap."
This is the file that is deployed to the phone or phone emulator.

The XAP file is a package of other files, in the very popular compression format known as ZIP. If you rename Helloworld_Phone.xap to Helloworld_Phone.zip, you can look inside. You'll see:

-several bitmap files that are part of the project;
- an XML file;
- a XAML file, and
-a Helloworld_Phone.dll file, which is the compiled binary of your program.

Adding Graphics

In Silverlight, you can add graphics by using **Shape** classes. You can create simple shapes, such as **Rectangles**, or more complex shapes, such as **Polygons**. **Brushes** are used to color or paint objects in Silverlight.

You'll start by adding a **StackPanel** around the **TextBlock**. A **Panel** is a container that is used to group and lay out UI elements. Each application should have at least one **Panel**. A **StackPanel** lays out each element one after the other, either vertically or horizontally, depending on the **Orientation**. **Grid** and

Canvas panels allow for more exact positioning of elements.

The shape you'll create is an **Ellipse**. The **Ellipse** will appear after the **TextBlock** in the **StackPanel**. You'll specify the **Height** and **Width** of the **Ellipse** as well as the **Fill**. For the **Fill**, you must specify a **Brush** to paint the **Ellipse**.

Instead of using Design view, this time you'll work in XAML view:

- Close the Toolbox window.

- In XAML view, locate the TextBlock that you added.

- Replace the TextBlock element with the following XAML.

```
<StackPanel>
    <TextBlock FONTSIZE="50" TEXT="HELLO, WORLD!" />
    <Ellipse Fill="Blue" Height="150" Width="300"
                                  Name="FirstEllipse" />
</StackPanel>
```

Press F5 to run the application:

Adding a Button

The next thing you'll add to your application is a button Control. Silverlight has a rich control library that includes a Button, a TextBox, ListBox, and many more.

There are two parts to adding a Button. The first part is to add a Button element to the XAML. The second part is to add some logic for handling events generated by user interaction, such as clicking the Button.

-In XAML view, add the following XAML after the <Ellipse /> tag.

```
<BUTTON HEIGHT="150"
      Name="FirstBUTTON"

                    Width="300"
                          Content="Tap" />
```

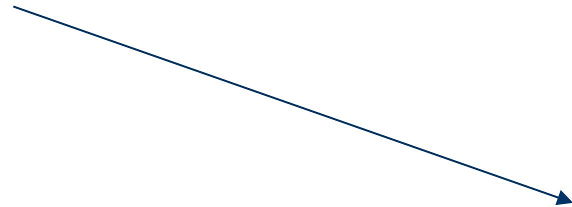
Visual Studio can create the event handlers for you.

- In Design View, select the Button.

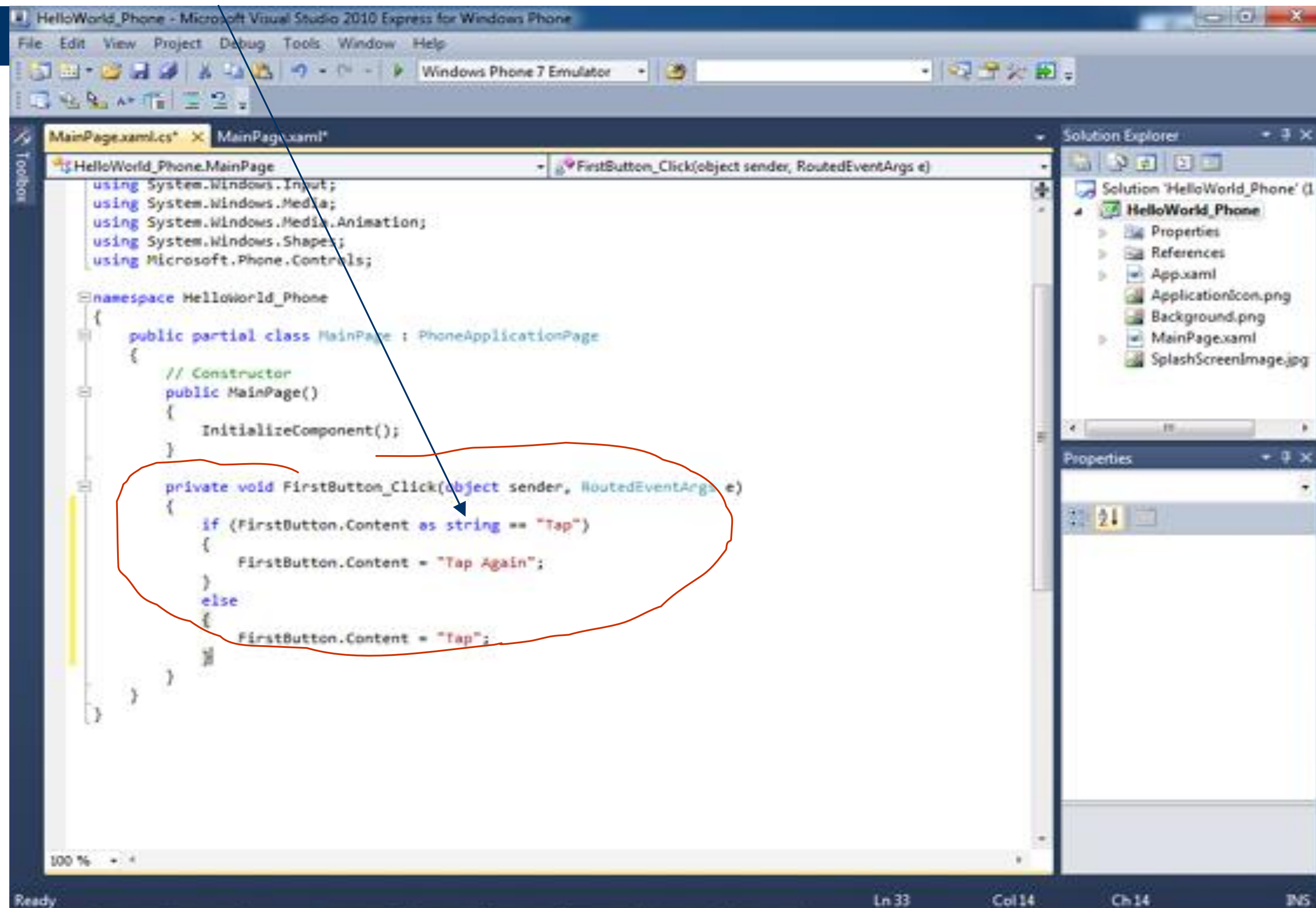
-In the Properties window, click the Events tab. A list of events for the Button appears. Select the needed events

The code-behind file MainPage.xaml.cs opens and you should see the FirstButton_Click event handler.

Add the following code to the event handler.



```
private void FirstButton_Click( object sender, RoutedEventAgs e)

if (FirstButton.Content as string == "TAP")
    {
    FirstButton.Content = "TAP AGAIN";
    }
else        { FirstButton.Content = "TAP"; } }
```



In the XAML for the Button, notice that a Click attribute was added

```
<StackPanel>
  <TextBlock FONTSIZE="50" TEXT="HELLO, WORLD!" />
  <Ellipse Fill="Blue" Height="150" Width="300"  Name="FirstEllipse" />
  <Button Height="150"  Width="300"  Content="Tap"  Name="FirstBuTTON"
                                     Click="FirstButton_CLICK" />
</StackPanel
```

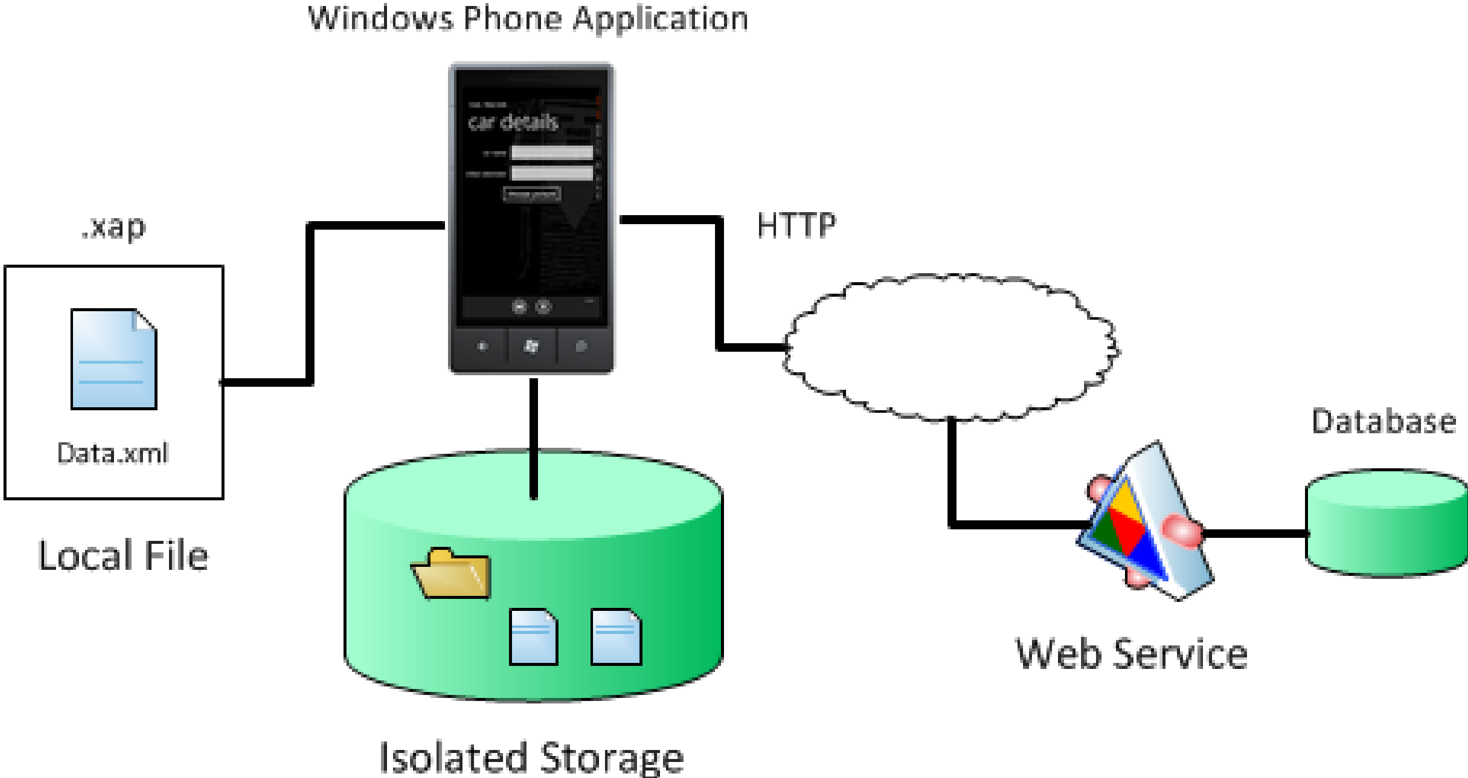
Publishing to the Marketplace

When you have finished your application, you will likely want to distribute it to the public as a free download or sell it. You do this by submitting your application to the Windows Phone Marketplace

You submit your application for publication through the App Hub, where it goes through a certification process to ensure that it meets the requirements. When the application is certified, marketplace pages are generated for display on the phone and in the Zune software.

Getting data

Sources of Data



Local Files

You can use files, such as text and XML files.
Local files can be compiled as resource files or content files.

Resource Files

Resource files are embedded in the project package (.xap). The advantage of a resource file is that the file will always be available to the application. However, your application may take longer to start when you use resource files.
You can access resource files by using the Application.GetResourceStream().

You typically use resource files when you have the following conditions:
- You aren' concerned about application startup time.
- You don't need to update the resource file after it's compiled into an assembly.
- You want to simplify application distribution complexity by reducing the number of file dependencies.

Content Files

For performance reasons, content files are preferred over resource files for Windows Phone 7 applications. Content files are included in the application package (.xap) without embedding them in the project assembly. Although they aren't compiled into an assembly, assemblies are compiled with metadata that establishes an association with each content file.
You access a content file relative to the application package file. For an example, use XElement.Load() method to access a content file.

Isolated Storage

If you need to store and retrieve user-specific information, you can use isolated storage.

In Silverlight for Windows Phone applications there's no direct access to the operating system's file system. However, you can use isolated storage to store and retrieve data locally on the user's device.

There are two ways to use isolated storage:

-to save or retrieve data as key/value pairs use the IsolatedStorageSettings class.
-to save or retrieve files by using the IsolatedStorageFile class.

Web Service Terminology

Working with **web services** can be a little confusing because of the different types of services,
formats,
and technologies.
The following are some terms related to web services.

Term	Description
web service	Units of application logic that provide data and services to other applications. Applications access web services using standard web protocols and data formats such as HTTP, XML, and SOAP, independent of how each web service is implemented.
REST	(Representational State Transfer Protocol) . A protocol for exposing resources on the web for access by clients.
POX	(Plain Old XML) A term used to describe basic XML.
JSON	(JavaScript Object Notation) A lightweight format for exchanging data. It's designed to be human-readable, but also easily parsed by a computer.
OData	(Open Data Protocol) A web protocol for querying and updating data.
SOAP	(Simple Object Access Protocol) A lightweight protocol intended for exchanging structured information in a decentralized, distributed environment.

Web Service Technologies

There are a several networking and Web service technologies that you can use to get data into your Silverlight for Windows Phone application:

HTTP classes
WCF services
WCF Data Services (OData services)
Windows Azure Services

HTTP Classes

You can access web services or resources on a network server directly from a Silverlight for Windows Phone application by using the HttpWebRequest/HttpWebResponse or WebClient classes in the System.Net namespace.
***These classes provide the functionality required to send requests to any web service available over the HTTP protocol*.**

Silverlight does not support the ability to host HTTP-based services, so these classes are useful when the phone application is using an existing web service. You typically use these classes if the HTTP service is hosted by a third-party and not within your control, and the service response is XML or JSON.

However, if you're building the service yourself, based on an existing data model, Silverlight offers more productive end-to-end solutions that can be built using WCF.

WCF Services

Windows Communication Foundation (WCF) is a framework for **building and accessing** web services.
WCF enables you to **expose a class as a service and exchange objects** between Silverlight and that service.
In a Silverlight for Windows Phone application you can use the
SLsvcUtil.exe tool or the Add Service Reference feature of Visual Studio

to generate a local proxy class for the service.
The proxy class enables you to access the service as though it's a local class. WCF services support a breadth of protocols (including HTTP and TCP) and a variety of formats, such as SOAP, XML.

WCF Data Services (OData services)

WCF Data Services, formerly known as ADO.NET Data services, is a framework to access data from your existing data model in the style of representational state transfer (REST) resources.

WCF Data Services handles all of the HTTP communication, serialization, and other tasks you traditionally have when you attempt to expose your data model as a service. So, applications can access this data through the standard HTTP protocol to execute queries, and even to create, update, and delete data in a data service, either in the same domain or across domains.

The OData functionality for Windows Phone is provided by the OData Client library.

Windows Azure Storage Services

You can use Windows Azure to store and retrieve data for use in your Windows Phone apps, particularly since storage on the device is limited.

Windows Azure storage services provide persistent, durable storage in the cloud and can scale elastically to meet increasing or decreasing demand.

The way you access Windows Azure storage is very similar to the way you access a web service.

Data Binding a Control to an Item. Example.

The following shows an example of binding a control to a single data item.
The target is the Text property of a text box control.
The source is from a music information class : Recording .

XAML

```
<GRID X:NAME="CONTENTPANEL" GRID.ROW="1" MARGIN="12,0,12,0">

    <TextBox VerticalAlignment="Top" IsReadOnly="True" Margin="5"

                TextWrapping="Wrap" Height="120" Width="400"

                Text="{Binding}" x:Name="textBox1" />

</GriD>
```

// CODE – BEHIND file

```
public MainPage()
{InitializeComponent();
// Set the data context to a new recording
  textBox1.DataContext = new Recording("Chris Sells", "Chris Sells Live",
                                     new DateTime(2008, 2, 5)); }
```

```
public class Recording                          // A SIMPLE BUSINESS OBJECT
{ public  Recording() { }
  public Recording(string artistName, string cdName, DateTime release)
   {    Name = cdName;
        ReleaseDate = release;
        Artist = artistName;
   }
  public string Artist { get; set; }
  public string Name { get; set; }
  public DateTime ReleaseDate { get; set; }
// Override the ToString method.
 public override string ToString()
   { return Name + " by " + Artist + ", Released: " + ReleaseDate.ToSortDateString(); }
   }
```

When you run the app, it will look something like this:

To display a music recording in a text box, the control's Text property is set to a Binding by using a markup extension.

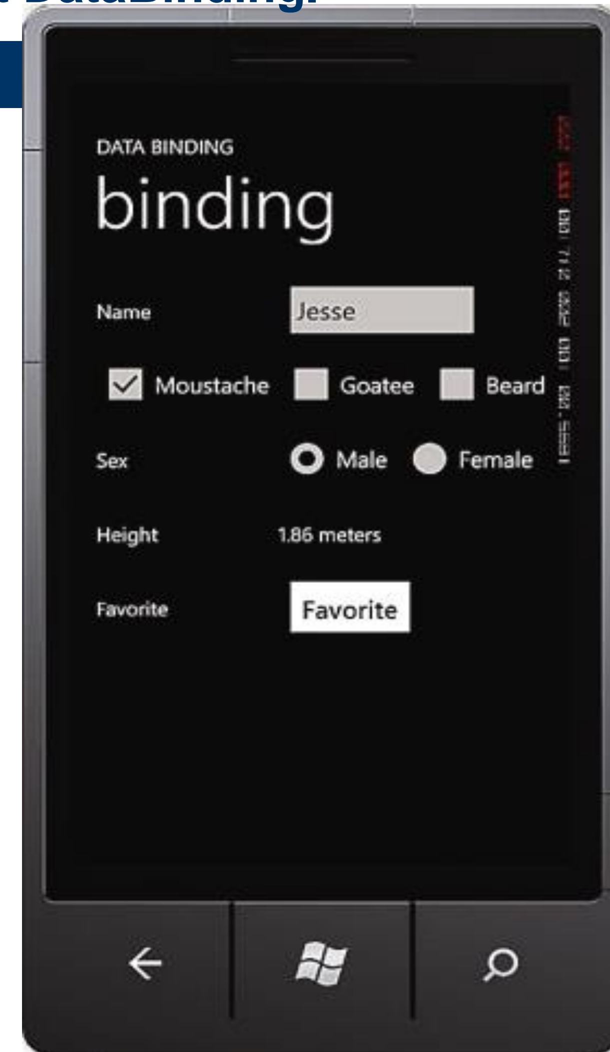
In this example, the binding mode is BindingMode.OneWay by default, which means that data are retrieved from the source, but changes are not propagated back to the source

A more complex data-binding example

Windows Phone Data Binding

(MSDN magazine, April, 2012)

1. Let us create Windows Phone application and name it DataBinding.



Begin by creating the class that will serve as the data to which you'll be binding (also known as the DataContext).

Right-click on the project → Add | New | Class and name the class Person.cs :

```
public class Person
{
public enum Sex
        {
        Male,
        Female,
        }
public string Name { get; set; }
public bool Moustache { get; set; }
public bool Goatee { get; set; }
public bool Beard { get; set; }
public Sex WhichSex { get; set; }
public double Height { get; set; }
public DateTime BirthDate { get; set; }
public bool Favorite { get; set; }
}
```

2. Creating the Form

The next task is to create the form you'll use to bind the data. Right-click on the project and select **"Open in Expression Blend."**

As a rule, create UI in Expression Blend and write code in Visual Studio.

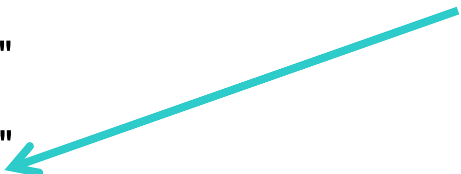
Create six rows and two columns (according the preview) in the content grid, and drag on the appropriate input controls:

```
<Grid
  x:Name="ContentPanel
  " Grid.Row="1“
   Margin="24,0,0,0">
     <Grid.ColumnDefinitions>
        <ColumnDefinition
               …..

          <TextBox
          x:Name="Name"
          TextWrapping="Wrap"
          d:LayoutOverrides="Height"
          Grid.Column="1"
          HorizontalAlignment="Left"
          Width="200"
          VerticalAlignment="Center"
          Text="{Binding Name}" />
          …
```

Binding

Each of the text-entry fields now has its value set using the Binding syntax. For example, to tell the TextBox to bind, identify which of its attributes will require the data—in this case, the Text attribute—and use the binding syntax, as shown earlier.

For example, this XAML states that the Text for the TextBox will be obtained from a public property named Name:



The object that contains the bindable property is known as the DataContext. It can be just about anything, but in this case you're going to create an instance of the Person class, and then you're going to set that Person object to be the DataContext for the entire view.
Note that you can set a DataContext for a container, in this case the page, and all the view controls inside that container will share that DataContext.
You're free to assign other DataContexts to one or more individual controls.

You can instantiate the Person in the Loaded event handler of the codebehind page. The Loaded event is called once the page is loaded and the controls are initialized:

```
private Person _currentPerson;
private Random _rand = new Random();
public MainPage()
{
InitializeComponent();
Loaded += MainPage_Loaded;
}
void MainPage_Loaded( object sender, RoutedEventArgs e )
{
_currentPerson = new Person
{
Beard = false,
Favorite = true,
Goatee = false,
Height = 1.86,
Moustache = true,
Name = "Ognian",
WhichSex = Person.Sex.Male
};
}
```

Now you can set the DataContext for every control in the ContentPanel to be the _currentPerson object you just instantiated (in the Loaded event handler):

```
ContentPanel.DataContext = _currentPerson;
```

Once it knows its DataContext, the TextBox can resolve the Name property and obtain the value ("Ognian") to display. The same is true for all the other controls, each bound to a property in the new Person object.

To explain the relationship between the binding and the display, let’s create a number of Person objects and display them one by one. To do this, modify MainPage.xaml.cs to create a list of (randomly created) Persons and then iterate through the list with a new “Next” button on the UI, which you should add to the bottom row:

```
<Button
  Name="Next"
  Content="Next"
  Grid.Row="5“

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

Here’s the modified code to interact with the Next button:

```
void MainPage_Loaded( object sender, RoutedEventArgs e )
{
  SetDataContext();
  Next.Click += Next_Click;
}

private void SetDataContext()
{  ContentPanel.DataContext = GeneratePerson();}

void Next_Click( object sender, RoutedEventArgs e )
            {  SetDataContext();}
```

Two-Way Binding

What if the user interacts with the UI and changes a value directly (for example, types a new name into the Name TextBox)? You'll (usually) want that change pushed back to the underlying data (the DataContext object). To do that, you'll use two-way binding.

To modify the program to use two-way binding on the Name property, find the Name binding and modify it to this:

```
<TextBox
  x:Name="Name"
  TextWrapping="Wrap"
  d:LayoutOverrides="Height"
  Grid.Column="1"
  HorizontalAlignment="Left"
  Width="200"
  VerticalAlignment="Center"
  Text="{Binding Name, Mode=TwoWay}" />
```

Data manipulation and display

* for additional information about the topic

MSDN magazine, November 2011
Full code at:http://archive.msdn.microsoft.com/mag201111MangoApp

A real DB application

(using Windows Phone SDK 7.1 tecnology – named Mango)

“Mango” is the internal code name for the Windows Phone SDK 7.1 release.

We’ll examine Mangolicious, a Windows Phone SDK 7.1 application about mangoes. The application provides a range of mango recipes, cocktails and facts, but the real purpose is to explore some of the big new features in the 7.1 release, specifically:

Local database and LINQ to SQL
Secondary Tiles and deep linking

Here’s a summary of the tasks required to build this application:

1. Create the basic solution in Visual Studio.
2. Independently create the database for the recipe, cocktail and fact data.
3. Update the application to consume the database and expose it for data binding.
4. Create the various UI pages and data bind them.
5. Set up the ‘’Secondary Tiles’’ feature to allow the user to pin Recipe items to the phone’s Start page.

Create the Solution

For this application, we'll use the
Windows Phone Silverlight and XNA Application **template** in Visual Studio.

Create the Database and DataContext Class

The Windows Phone SDK 7.1 release introduces support for **local databases.** That is, an application can store data in a **local database file (.sdf)** on the phone. We recommend to create the database in code, either as part of the application itself or via a separate helper application.
For the Mangolicious application, we have only static data, and we can populate the database in advance. To do this, we'll create a separate database-creator helper application, starting with the simple Windows Phone Application template. To create the database in code, we need a class derived from standard **DataContext** class.
This same **DataContext class** can be used both in the helper application that **creates** the database and the main application that **consumes** the database. In the helper application, we must specify the database location to be in isolated storage, because that's the only location we can write to from a phone application. The class also contains a **set of Table fields** for each database table:

```
public class MangoDataContext : DataContext
{
  public MangoDataContext()       : base("Data Source=isostore:/Mangolicious.sdf") { }

  public Table<Recipe> Recipes;
  public Table<Fact> Facts;
  public Table<Cocktail> Cocktails;             }
```

There's a 1:1 mapping between Table classes in the code and tables in the database. The **Column properties** map to the columns in the table in the database, and include the database schema properties such as the data type and size (INT, and so on), whether the column may be null, whether it's a key column and so on.

We define Table classes for all the other tables in the database in the same way, as shown:

```
[Table]
public class Recipe
{  private int id;
  [Column( IsPrimaryKey = true, IsDbGenerated = true,  DbType = "INT NOT NULL Identity",
                CanBeNull = false,  AutoSync = AutoSync.OnInsert)]
  public int ID
  {              get { return id; }
                 set     { if (id != value)  { id = value;  }    }
  }

  private string name;
  [Column(DbType = "NVARCHAR(32)")]
  public string Name
  {    get { return name; }
       set { if (name != value) { name = value;  }    }
  }
    ... additional column definitions omitted for brevity
}
```

Still, in the helper application we now need

a ViewModel class to mediate between the View (the UI) and the Model (the data) using the DataContext class.

The **ViewModel** has a DataContext field and a set of collections for the table data (Recipes, Facts and Cocktails). The data is static, so simple List<T> collections are sufficient here. For the same reason, we only need *get* property accessors, not *set* modifiers:

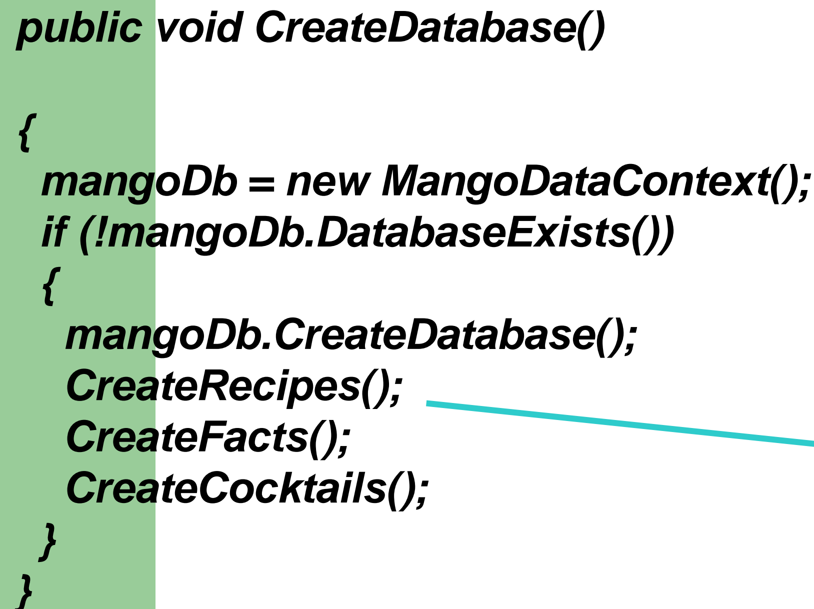
```
public class MainViewModel
{
  private MangoDataContext mangoDb;

  private List<Recipe> recipes;
  public List<Recipe> Recipes
  {
    get
    {
      if (recipes == null)
      {
        recipes = new List<Recipe>();
      }
    return recipes;
    }
  }

    ... additional table collections omitted for brevity
}
```

We also expose a public method—which we can invoke from the UI—to actually **create the database** and all the data. In this method, we create the database itself if it doesn't already exist and then create each table in turn, populating each one with static data.

For example, to create the Recipe table:

1. we create multiple instances of the Recipe class, corresponding to rows in the table;
2. add all the rows in the collection to the DataContext;
3. and finally commit the data to the database.

The same pattern is used for the Facts and Cocktails tables:



```
public void CreateDatabase()

{
  mangoDb = new MangoDataContext();
  if (!mangoDb.DatabaseExists())
  {
    mangoDb.CreateDatabase();
    CreateRecipes();
    CreateFacts();
    CreateCocktails();
  }
}
```

```
private void CreateRecipes()
{
  Recipes.Add(new Recipe
  (
    ID = 1,
    Name = "key mango pie",
    Photo = "Images/Recipes/MangoPie.jpg",
    Ingredients = "2 cans sweetened condensed milk, ¾ cup fresh key lime juice, ¼ cup mango purée,
                    2 eggs, ¾ cup chopped mango.",
    Instructions = "Mix graham cracker crumbs, sugar and butter until well distributed. Press into a
            inch pie pan. Bake for 20 minutes. Make filling by whisking condensed milk, lime juice, mango
            purée and egg together until blended well. Stir in fresh mango. Pour filling into cooled
            crust and bake for 15 minutes.",
    Season = "summer"
  ));

    ... additional Recipe instances omitted for brevity

  mangoDb.Recipes.InsertAllOnSubmit<Recipe>(Recipes);
  mangoDb.SubmitChanges();
}
```

1

2

3

At a suitable point in the helper application—perhaps in a button click handler—we can then invoke this **CreateDatabase method.** When we run the helper (either in the emulator or on a physical device), the **database file will be created in the application's isolated storage**.
The final task is to extract that file to the desktop so we can use it in the main application.
To do this, we'll use the **Isolated Storage Explorer tool**, a command-line tool that ships with the Windows Phone SDK 7.1.
Here's the command to take a snapshot of isolated storage from the emulator to the desktop:

```
"C:\Program Files\Microsoft SDKs\Windows Phone\v7.1\Tools\IsolatedStorageExplorerTool\ISETool"
ts xd {e0e7e3d7-c24b-498e-b88d-d7c2d4077a3b} C:\Temp\IsoDump
```

This command assumes the tool is installed in a standard location. The parameters are explained:

Parameter	Description
ts	"Take snapshot" (the command to download from isolated storage to the desktop).
xd	Short for XDE (that is, the emulator).
{e0e7e3d7-c24b-498e-b88d-d7c2d4077a3b}	The ProductID for the helper application. This is listed in the WMAppManifest.xml and is different for each application.
C:\Temp\IsoDump	Any valid path on the desktop where you want to copy the snapshot to.

Having extracted the **SDF file to the desktop**, we've finished with the helper application and can turn our attention to the Mangolicious application that will consume this database

Application consuming the Database

In the Mangolicious application, we **add the SDF file to the project** and also **add the same custom DataContext class to the solution**.

In Mangolicious, we don't need to write to the database, so we can use it directly.

Also, Mangolicious defines **new SeasonalHighlights table** in code. There's no corresponding SeasonalHighlight table in the database. Instead, this code table pulls data from two underlying database tables (Recipes and Cocktails) and is used to populate the Seasonal Highlights panorama.

These changes are the only differences in the **DataContext class** between the database-creation helper application and the Mangolicious database-consuming application:

```
public class MangoDataContext : DataContext
{
  public MangoDataContext()
    : base("Data Source=appdata:/Mangolicious.sdf; File Mode=read only;") { }

  public Table<Recipe> Recipes;
  public Table<Fact> Facts;
  public Table<Cocktail> Cocktails;
  public Table<SeasonalHighlight> SeasonalHighlights;
}
```

The **Mangolicious application** also needs a **ViewModel class**, and **we can use the ViewModel class from the helper application.**

We need the DataContext field and the set of List<T> collection properties for the data tables.

We'll add a string property to record the current season, computed in the constructor:

```
public MainViewModel()
{
  season = String.Empty;
  int currentMonth = DateTime.Now.Month;
  if (currentMonth >= 3 && currentMonth <= 5) season = "spring";
  else if (currentMonth >= 6 && currentMonth <= 8) season = "summer";
  else if (currentMonth >= 9 && currentMonth <= 11) season = "autumn";
  else if (currentMonth == 12 || currentMonth == 1 || currentMonth == 2)
    season = "winter";
}
```

The critical method in the ViewModel is the **LoadData method.** Here, we initialize the database and perform **LINQ-to-SQL queries** to load the data via the DataContext into in-memory collections. We could preload all three tables at this point, but we want to optimize startup performance by delaying the loading of data unless and until the relevant page is actually visited.

The only data we *must* load at startup is the data for the SeasonalHighlight table, because this is displayed on the main page. For this, we have two queries to select only rows from the Recipes and Cocktails tables that match the current season, and add the combined row sets to the collection:

```
public void LoadData()
{ mangoDb = new MangoDataContext();
  if (!mangoDb.DatabaseExists())  {  mangoDb.CreateDatabase();  }

  var seasonalRecipes = from r in mangoDb.Recipes
                        where r.Season == season
                        select new { r.ID, r.Name, r.Photo };
  var seasonalCocktails = from c in mangoDb.Cocktails
                          where c.Season == season
                          select new { c.ID, c.Name, c.Photo };

  seasonalHighlights = new List<SeasonalHighlight>();
  foreach (var v in seasonalRecipes)
  {    seasonalHighlights.Add(new SeasonalHighlight {
     ID = v.ID, Name = v.Name, Photo = v.Photo, SourceTable="Recipes" });  }
  foreach (var v in seasonalCocktails)
  {    seasonalHighlights.Add(new SeasonalHighlight {
     ID = v.ID, Name = v.Name, Photo = v.Photo, SourceTable = "Cocktails" });  }

  isDataLoaded = true;
}
```

We can use similar LINQ-to-SQL queries to build separate LoadFacts, LoadRecipes and LoadCocktails methods that can be used after startup to load their respective data on demand.

Create the UI

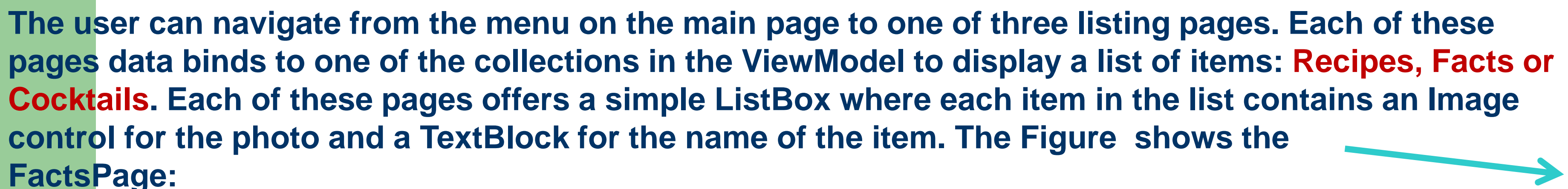
The main page consists of a **Panorama with three PanoramaItems.**
The first item consists of a ListBox that offers a main menu for the application. When the user selects one of the ListBox items, we navigate to the corresponding page—that is, the collection page for either Recipes, Facts and Cocktails.
Just before navigating, we make sure to load the corresponding data into the Recipes, Facts or Cocktails collections:

```
switch (CategoryList.SelectedIndex)
{
  case 0:
    App.ViewModel.LoadRecipes();
    NavigationService.Navigate(    new Uri("/RecipesPage.xaml", UriKind.Relative));
    break;

... additional cases omitted for brevity
}
```

When the user selects an item from the Seasonal Highlights list in the UI, we examine the selected item to see whether it's a Recipe or a Cocktail, and then navigate to the individual Recipe or Cocktail page, passing in the item ID as part of the navigation query string:

```
SeasonalHighlight selectedItem =
  (SeasonalHighlight)SeasonalList.SelectedItem;
String navigationString = String.Empty;
if (selectedItem.SourceTable == "Recipes")
{
  App.ViewModel.LoadRecipes();
  navigationString = String.Format("/RecipePage.xaml?ID={0}", selectedItem.ID);
}
else if (selectedItem.SourceTable == "Cocktails")
{
  App.ViewModel.LoadCocktails();
  navigationString = String.Format("/CocktailPage.xaml?ID={0}", selectedItem.ID);
}
NavigationService.Navigate(
  new System.Uri(navigationString, UriKind.Relative));
```

The user can navigate from the menu on the main page to one of three listing pages. Each of these pages data binds to one of the collections in the ViewModel to display a list of items: Recipes, Facts or Cocktails. Each of these pages offers a simple ListBox where each item in the list contains an Image control for the photo and a TextBlock for the name of the item. The Figure shows the FactsPage:



Mangolicious

fun facts

mango mania

mango, a love story

mango is good for you

the venerated mango

joie de mango

Fun Facts, One of the Collection List Pages

When the user selects an individual item from the Recipes, Facts or Cocktails lists, we navigate to the individual Recipe, Fact or Cocktail page, passing down the ID of the individual item in the navigation query string.
Again, these pages are almost identical across the three types, each one offering an Image and some text below.

The codebehind for each of these pages is simple. In the OnNavigatedTo() override, we extract theindividual item ID from the query string, find that item from the ViewModel collection and data bind to it.

Mangolicious

key mango pie

INGREDIENTS

Crust: 1 ½ cups fine graham cracker crumbs, ½ cup granulated cane sugar, 4 tablespoons (½ stick) melted butter. Filling: 2 cans sweetened condensed milk, ¾ cup fresh key lime juice (if not possible regular lime juice will work), ¼ cup mango purée, 2 whole eggs plus 2 egg yolks, ¾ cup chopped (¼ inch cubes) mango. Topping: 1 pint heavy cream, 1 tablespoon confectioner's sugar, 1 teaspoon lime zest, 2 limes cut into wheels.

The code for the RecipePage is a little more complex than the others—the additional code in this page is all related to the HyperlinkButton positioned at the top-right-hand corner of the Page:

When the user clicks **the "pin" HyperlinkButton** on the individual Recipe page, we pin that item as a tile on the phone's Start page. The act of pinning takes the user to the Start page and deactivates the application. When a tile is pinned in this way, it animates periodically, flipping between front and back, as shown:

Subsequently, the user may tap this pinned tile, which navigates directly to that item within the application. When he reaches the page, the "pin" button will now have an "unpin" image. If he unpins the page, it will be removed from the Start page, and the application continues.

This to be done, the programmer must :

1. define a tag into the current page, marking it as tiled;
2. define a code – handler to **PinUnpin_Click()** event defining the way and substitutions for un-pining