

Programming for mobiles: theory, OS, frameworks, good practices

Design practices for building a better mobile applications

Mobile devices are on track to become the single most common way to access the Web within five years.

-in America, 25 percent of mobile Web users say they “never” or “infrequently” access the Web using a traditional PC

Why Mobile Programming Is Different

As you know, just about every mobile browser supports some form of HTML. Many, especially on high-end devices such as iPhones and Windows Phone 7, support the latest HTML, CSS and JavaScript standards and render pixel-perfect copies of what you’d see in a traditional PC browser.

Your cheapest option for supporting mobile browsers, then, is to do nothing.

Choosing this option leads to a very poor mobile browsing experience, for several reasons:

1. Mobile phone screens are small.

The most obvious difference between a PC and a mobile phone is display size. The average size of a PC monitor is 21 inches, with a display resolution no less than **1024 × 768**.

The typical smart phone has a display of 3.9 inches and a resolution less than **800 × 480**. Clearly, a mobile phone doesn't give you much space to work with. You need to use less text and better icons.

The Figure below illustrates an effective way of presenting clear icons that represent key functions.



Some mobile browsers, such as Opera Mini, handle desktop-width pages by **dynamically reformatting the page** layout and styles. The resulting appearance is rarely what your designer had in mind. Other mobile browsers, such as Safari for iPhone or Internet Explorer for Windows Phone 7, render desktop-width pages and then force the user to **zoom in and out** and pan around to read the text.

If you rely on icons to convey your features, functions and contents, your icons need to be **more concrete and less complex** than for a PC, and they must accurately represent the concept you're trying to express.

Your **text should be as legible and readable** as possible. Legibility refers to how easy characters and words can be identified. To make your text more legible, use a font size of at least 10 points and turn up the contrast.

Here's what you need to remember about display size:

- You don't have much room to work with, so minimize the number of screens and options, and minimize the amount of text.*
- The text you use should be legible and readable.*
- Replace text with icons when you can.*
- Your icons should look as realistic as possible.*
- The visuals should match the tasks, and the design should be straightforward.*

2. Input: How It's Used

The standard PC has several ways of inputting, but basically a keyboard and a mouse are used most often. A mobile device relies mostly on **fingers**, whether they are used to type on the screen keyboard, **tap icons or buttons, or make gestures, such as swiping or pinching.**

To compensate for these drawbacks, you can provide **sound or vibration for key presses** and minimize text entry.

Fingers comes in all sizes, so make your **buttons large enough** (35 pixels square) and visually separated from other buttons or objects to reduce error.

Find innovative ways to input data, such as **using photographs.**

You should also take advantage of the hardware. **Windows Phone has three dedicated buttons** (Back, Start and Search), and you can use them to reduce clutter on the interface and to minimize error.

3. Context: Where It's Used

PCs are used mostly in homes and offices—fairly predictable and stable places, with good lighting, limited distractions and a relatively focused user.

When developing for mobile users, throw out everything you know about PC users. Mobile users are a different breed. Mobile device users (for the most part) are on the move, and many are in a rush. They have a limited amount of time and have come to your application for a specific reason. They come with questions that need to be answered.

If your application is to succeed, it must help users get those answers quickly. **If users are consistently helped by your application, they will spend more time using it.**

Environmental issues that rarely occur at the desktop often appear in a mobile environment.

Distractions such as **noise** can draw the user's attention away from the device, or at least interfere with the detection of sounds from your application. **Sun glare** can obliterate the screen and make it difficult even to see your application much less use it. The **overload of information** can make it difficult for users to concentrate on your application, which can affect their cognitive processing.

One final note: Developers who simply miniaturize the PC experience for the mobile device are missing the point. We need to focus our attention on giving users what they want most from our applications— simply but elegantly.

4. Mobile data networks are often slow

Don't assume that your visitors have the same bandwidth as fixed-line broadband users. They may even be paying by the megabyte, so heavyweight sites won't be popular.



There are two main aspects to support mobile browsers:

1. Detecting which kind of device a given visitor is using.

ASP.NET has built-in support for browser detection. In the next section, we'll examine this mechanism.

2. Producing output that works well on the detected device.

we'll describe technical means to produce different outputs for different devices, but it's still up to you to design and implement different layouts and user workflows for mobiles.

Browser Detection

You can find out whether or not a visitor is using a mobile browser using (ASP.NET) the **Request.Browser.IsMobileDevice** Boolean property. ASP.NET determines what kind of browser is making a request and what capabilities that browser has (screen size, JavaScript support and so on) by comparing the incoming request userAgent header string against a series of regular expressions in XML files that describe common browsers.

The information about corresponding device capabilities is stored in a set of **.browser files** in the folder :

C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config\Browsers
(or your installation's equivalent).

For example, the **standard iphone.browser** file includes the code shown:

```
<browsers>
  <!-- Mozilla/5.0 (iPhone; U; CPU like Mac OS X; en) AppleWebKit/420+ (KHTML, like Gecko) Version/3.0
        Mobile/1A543a Safari/419.3 -->
  <gateway id="IPhone" parentID="Safari">
    <identification>
      <userAgent match="iPhone" />
    </identification>
    <capabilities>
      <capability name="mobileDeviceModel" value="IPhone" />
      <capability name="mobileDeviceManufacturer" value="Apple" />
      <capability name="isMobileDevice" value="true" />
      <capability name="canInitiateVoiceCall" value="true" />    </capabilities>
    </gateway> ... </browsers>
```

The following element defines the expression to be matched against incoming userAgent header strings:

```
<userAgent match="iPhone" />
```

Once the system finds a matching userAgent expression, the remainder of the XML data specifies the type and capabilities of that device.

(Unfortunately, this does not include common modern browsers such as Opera Mobile or the default browser for Google Android. *Request.Browser.IsMobileDevice* will incorrectly be set to false)

How to Enhance Browser Detection

You have two main options :

- 1.You can supply your own .browser files to represent newer devices.
- 2.You can use a third-party browser-detection library.

To take the first option, right-click on your project's name in the Visual Studio Solution Explorer and choose Add | Add ASP.NET Folder | App_Browsers. You can then add .browser files to that folder;

If you don't want to be responsible for tracking all of the hundreds of newly released mobile browsers and keeping your .browser files up-to-date, you can take the second option and use a third-party browser-detection library.

Currently, the one is **51degrees.Mobi Foundation**, an open source Library. This library does not use .browser files directly.

The easiest way to install **51degrees.Mobi Foundation** into either Web Forms or MVC projects is by using the NuGet package manager.

If you're running ASP.NET MVC 3, you already have NuGet. If not - use the VS Extension Manager (it's on the Tools menu) to install NuGet. Go to Tools | Library Package Manager | Package Manager and issue the command:

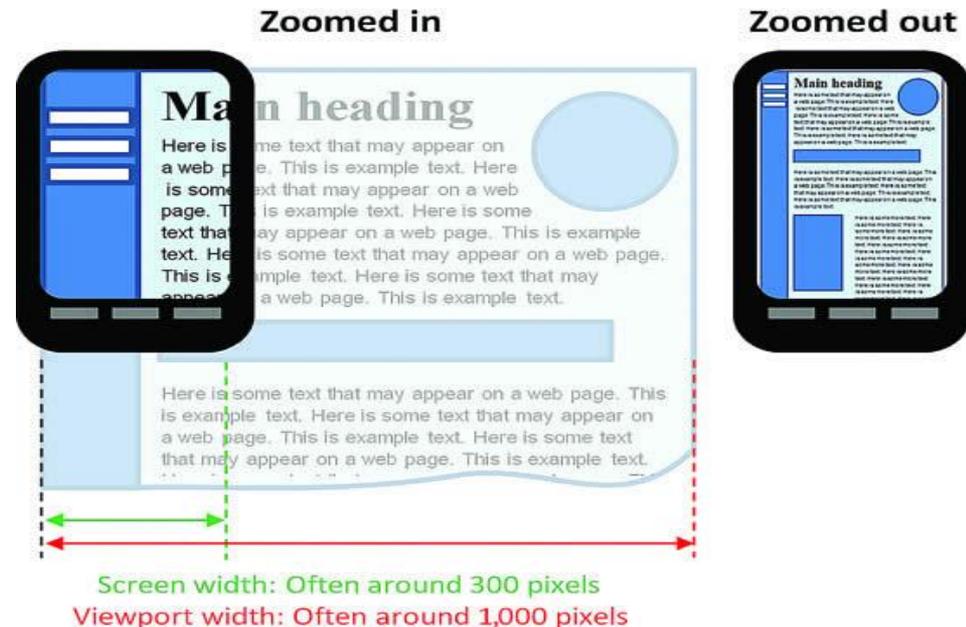
Install-Package 51Degrees.mobi

Styling for Mobiles

Now that you have an idea of how to detect mobile browsers reliably, We'll show a key way to control how pages are rendered by mobile browsers.

Many modern mobile browsers, including Safari for iOS and Internet Explorer for Windows Phone 7, try to make **rendered pages** look just as they do on a desktop browser. They know that most pages are designed for screens around 1,000 pixels wide and the designer has most likely not accounted for much smaller widths.

To solve this, they typically render the page onto a virtual canvas known as a “**viewport**,” usually around 1,000 virtual pixels wide. The browser can then scale the visual display of that virtual canvas arbitrarily, allowing the user to zoom in and out and pan around. This arrangement is illustrated:



Controlling the Viewport Width

If you've actually designed pages for the small screen, you'll want your pages to be laid out on a viewport that's the same width as the actual screen, so that it neatly fits horizontally with no zooming required.

Many of the most popular mobile browsers support a nonstandard “**viewport**” meta tag that lets you control the width of the virtual viewport. For example, if you add the following to your page's <head> section, the browser will lay out the page on a viewport 320 pixels wide:

```
<meta name="viewport" content="width=320"/>
```

This is usually a much better fit for mobile phones.

Keep in mind that some mobile devices have screens with much higher horizontal resolution. For example, the **iPhone 4 has 640 physical pixels per row**. However, it still makes sense to use a virtual viewport of around 320 pixels; otherwise, the resulting text will be too small to read without zooming in. If you want, you can let the virtual viewport vary in size according to the device being used, using the following syntax:

```
<meta name="viewport" content="width=device-width"/>
```

Note that some mobile devices won't give you a literal device width. They interpret “**device-width**” as meaning “the virtual viewport width that the manufacturer thinks gives the most pleasing result.” So, for example, iPhone 4 defines device-width as 320 pixels, despite its higher physical resolution.

Markup Recommendations

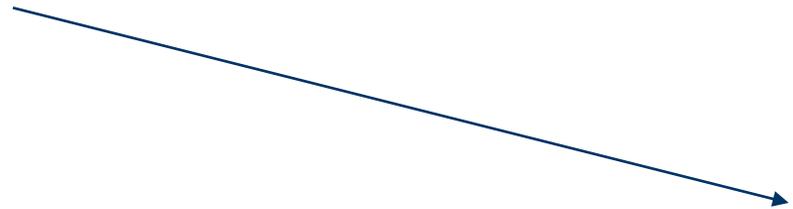
Whenever you're designing pages for mobile browsers:

- Use the viewport meta tag to make the viewport fit the horizontal width of the screen.
- Adjust your page layouts, CSS styles for this narrow width. If visitors don't need to zoom or scroll horizontally, your page feels more like a native application designed for their device—a far better experience.
- Make sure your links and buttons are large enough to be tapped imprecisely. Fingertips are much bigger than the tip of a mouse pointer.
- Minimize bandwidth requirements by not using very high-resolution images or massive JavaScript files

Architectural Options

You've seen how to detect mobile browsers, and some recommendations for markup that better suits them. Now we'll describe three straightforward options for structuring your application to produce different output for different browser types:

1. Showing and hiding sections of markup according to browser type.
2. Switching master pages according to browser type.
3. Presenting entirely different content according to browser type.



1. Showing and Hiding Markup

If you only need to include or exclude meta tags and CSS file references according to browser type, this is extremely simple. For example, in a Web Forms master page, you can add an “if” statement inside your <head> section:

```
<head runat="server">
  <title>My site</title>
  <link href="~/Styles/Site.css" rel="stylesheet" type="text/css" />

  <% if (Request.Browser.IsMobileDevice)
    { %>
    <meta name="viewport" content="width=device-width"/>
    <link href="~/Styles/MobileSite.css" rel="stylesheet" type="text/css" />
    <% } %>
</head>
```

For many sites this simple technique won't be sufficient, but there are two alternatives: switching the master page or presenting different content.

2. Switching Master Pages

You may be able to keep your existing content pages unchanged, and merely adapt the layout for the small screen using a different master page or layout. For example, if you're building a Web Forms application, you could define a standard page base class that switches its master page dynamically:

```
public class PageBase : Page
{
    protected override void OnPreInit(EventArgs e)
    {
        if (Request.Browser.IsMobileDevice)
            MasterPageFile = "~/Mobile.Master";
    }
}
```

Then, for any page whose layout should vary by device type, set its codebehind class to inherit from PageBase instead of the usual System.Web.UI.Page. You can then create a master page at /Mobile.Master whose layout and CSS styling are optimized for mobile devices.

3. Presenting Different Content

For some applications, you won't be able to adapt your desktop pages to suit mobile devices merely using different CSS or master pages and layouts because:

Your business requirements might be too demanding. If you want a truly slick mobile experience, you may need to display different (perhaps less) information to mobile devices, and possibly guide the user through different workflows. For example, your user-registration process may have fewer steps and collect less information for mobile visitors.

You may be working with legacy code that's not amenable to such change. For example, your existing markup may contain hardcoded element sizes and styles. Modifying this using CSS or a different master page might be impossible, or might just make things more complicated and less maintainable.

In either case, the ultimate solution is to use entirely separate logic and markup for different device types. The drawback is that you then **have two versions** to maintain, but the key benefit is that the behavior of the two can vary independently in any way you want. For Web Forms developers, the implementation is usually a set of mobile-specific ASPX pages, and for MVC developers, it usually means creating a new area for mobile-specific controllers and views. Either way, you'll need some logic to redirect incoming visitors to the correct page depending on their device type.



Getting Started with building (1) Windows Mobile solutions with Visual Studio and Windows Mobile SDK

-how to start to develop for Windows® phones by using the same tools that developers use for desktop development, for example:

Microsoft Visual Studio 2008;

and the Windows Mobile specific Microsoft .NET Compact Framework 3.5.

- .NET Compact Framework (.NET CF) is a subset of the full Microsoft .NET Framework, with all the functionality that you must have to deliver powerful enterprise applications for the Windows Mobile platform. Visual Studio 2008 has built-in support for both the .NET Compact Framework 2.0 and .NET Compact Framework 3.5.

Developers who develop desktop applications that use Visual Studio have all the necessary tools and knowledge to develop solutions for Windows Mobile platform.

Of course, there are differences between the desktop and mobile platform, and you have to approach each differently. However, the programming concepts remain the same.

You have to answer the following questions before you start to develop a Windows Mobile solution:

- What development tools do you have and want to use?
- What platform do you want to target, Smartphone or Pocket PC?
- What version of the .NET Compact Framework do you want to use?

Other business-related considerations can also influence your decisions:

- Do you want to create offline or online solution?
- How expensive is your connection?
- Do you want touch screen functionality?
- Are you targeting multiple form factors or a single platform?
- What are your security requirements?

History :

Visual Studio has supported Windows Mobile application development since Visual Studio 5.0.

Visual Studio 2005 supports development for Smartphone 2003, Pocket PC 2003 SE, and Windows CE. You can easily add support for Windows Mobile 5.0 (Smartphone, Pocket PC, and Pocket PC Phone Edition) and Windows Mobile 6 (Standard, Classic, and Professional) by installing SDKs for these platforms. SDKs include various tools that greatly improve your development experience, emulators, GPS, mobile operator emulator, and more.

Visual Studio 2008 brought some new features and functionalities, both in the IDE and in functionality.

The following list shows some new features in Visual Studio 2008:

- Support for unit testing, in Visual Studio Developer Edition and Visual Studio Team Suite
- Remote Performance Monitor
- Device Security Manager, part of Visual Studio 2008 IDE
- New Device Emulators and Device Emulator Manager 3.0
- New and redesigned New Project Wizard for creating new Windows Mobile projects

As its name suggests, .NET CF (compact framework) is smaller than the .NET Framework, and supplies a subset of the .NET Framework functionality. Therefore, most applications that you develop for .NET CF should run without any modifications on the desktop .NET Framework platform.

There are two main reasons for reduced functionality in .NET CF:

- unsupported desktop features and
- limited storage space on mobile devices.

some new controls that are added and improved:

- **MonthCalendar.** New in .NET CF 2.0.
- **DataGrid.** Greatly improved.
- **DateTimePicker.** New in .NET CF 2.0.
- **LinkLabel.** New in .NET CF 2.0.
- **Splitter.** New in .NET CF 2.0.
- **WebBrowser.** New in .NET CF 2.0.

Because Visual Studio 2005 and 2008 support application development for various resolutions and various orientations, .NET CF 2.0 introduced improvements in display and layout management.

When you change **Orientation from Portrait to Landscape or vice versa**, you want your application and interface to remain fully functional and available.

To support that requirement, the following improvements were made in .NET CF 2.0:

- **Automatic scrollbars.** If because of an orientation change, some controls remain outside the visible area, horizontal and vertical scrollbars are created automatically to provide access to all parts of user interface.
- **Tab order support.** Enables moving from one control to another by using the TAB key.

Working **with data** in .NET CF 2.0 resembles working with data in the .NET Framework. ADO.NET is fully supported. Working with Microsoft SQL Server Compact Edition (various versions), and working with XML, Web services, and files as data stores is supported. .NET CF supports datasets, data binding, and collections.

.NET Compact Framework 3.5 is a superset of .NET CF 2.0.

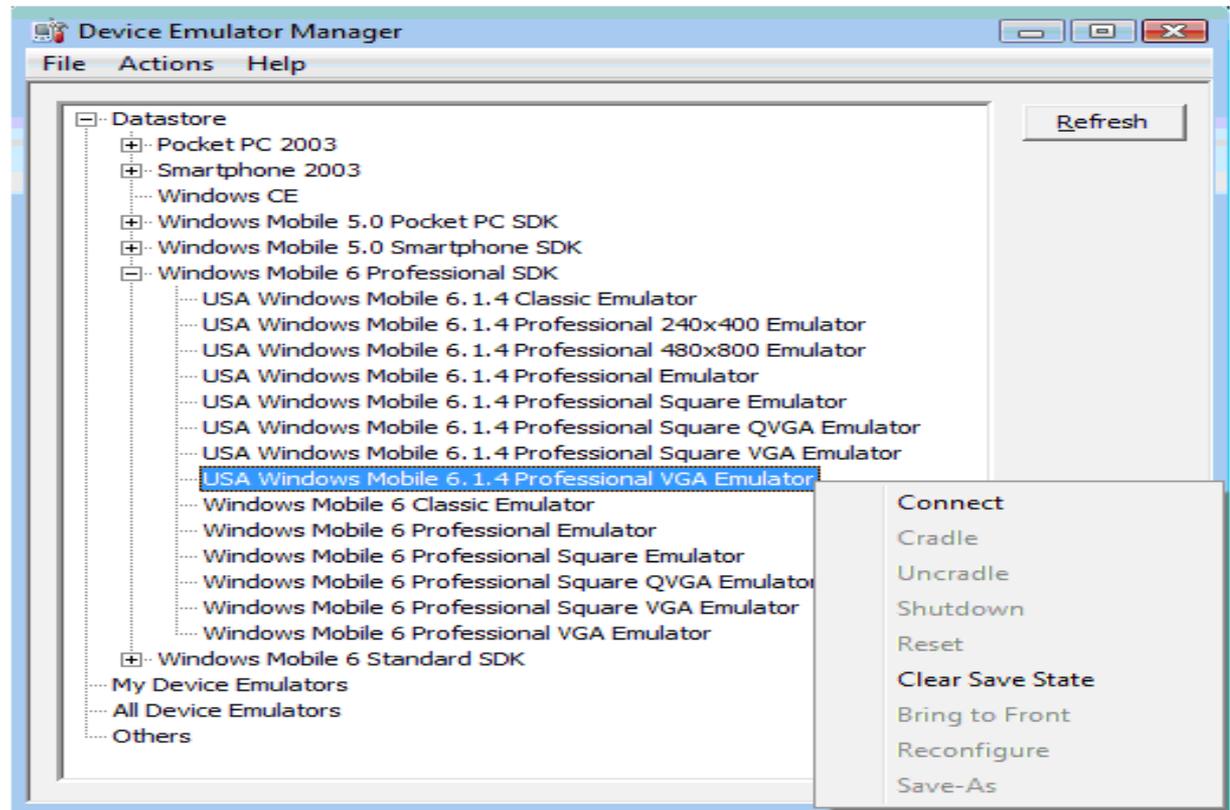
Windows Mobile SDK Tools

Windows Mobile SDK Tools are an addition to Visual Studio and .NET CF. They provide support and templates for developing applications targeting Windows Mobile 6 platforms.

Windows Mobile 6 SDK Tools are free downloads. They require you to have Visual Studio 2005 or 2008 and .NET CF installed. There are two separate installations for two platforms. One is Windows Mobile 6 Standard (previously called Smartphone) and the other one is Windows Mobile Professional, targeted for both Classic (previously called Pocket PC) and Professional (previously called Pocket PC Phone Edition).

Device Emulator Manager

Device Emulator Manager manages Device Emulators that are installed on the developer workstation



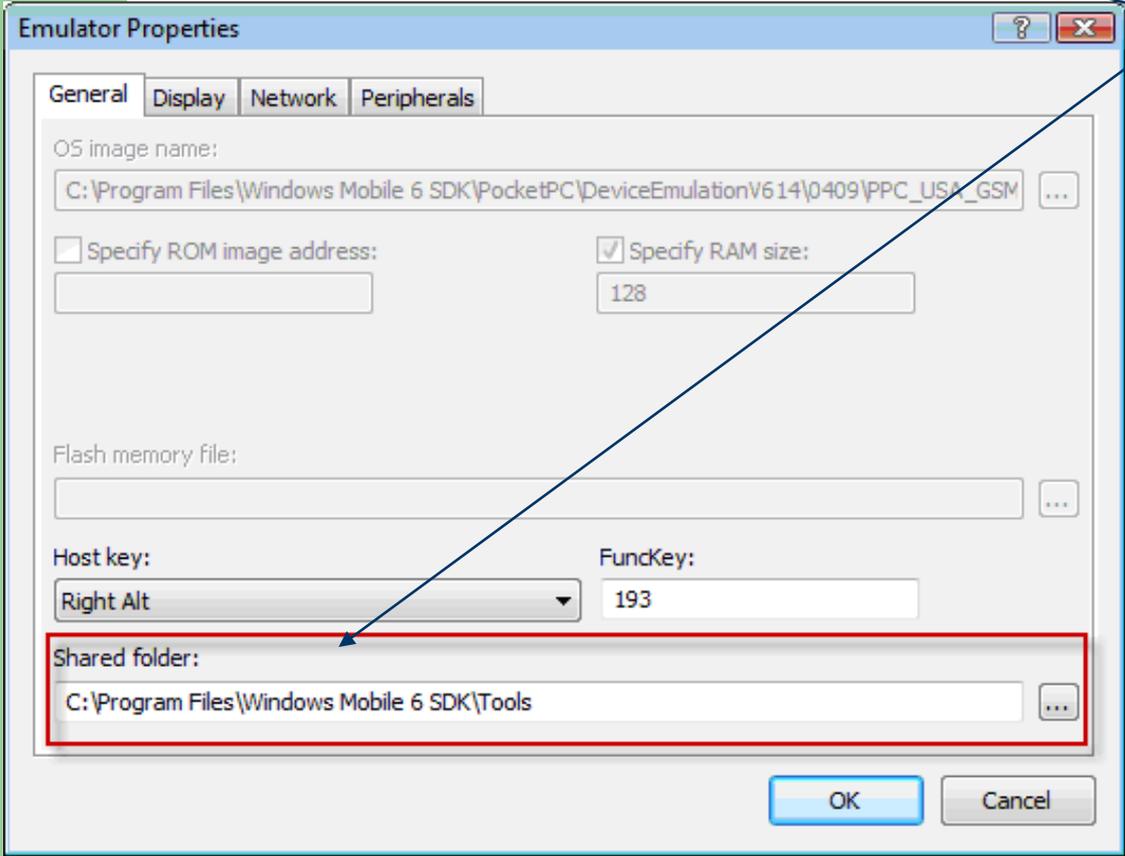
Device Emulators

You can start the Device Emulator from Microsoft Visual Studio IDE, manually or automatically when you deploy the application, or from Device Emulator Manager. When it is up and running, you can configure it to meet your needs. You can access configuration options on the Device Emulator menu.



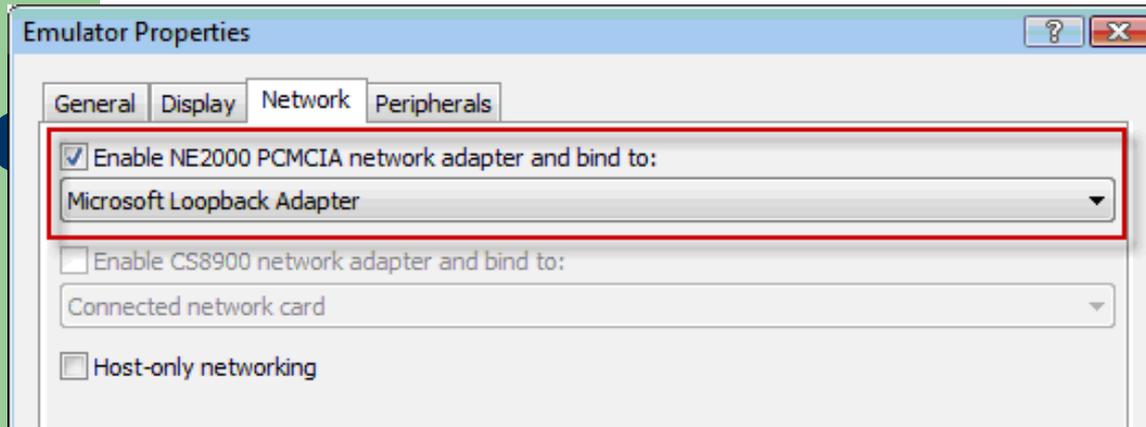
You frequently have to access host files and folders from the Device Emulator. The following illustration shows how to set up the location of a Shared folder in the Emulator configuration.

That gives you access to that folder from the Device Emulator. The Shared folder is displayed as a Storage Card in the Emulator File Explorer.

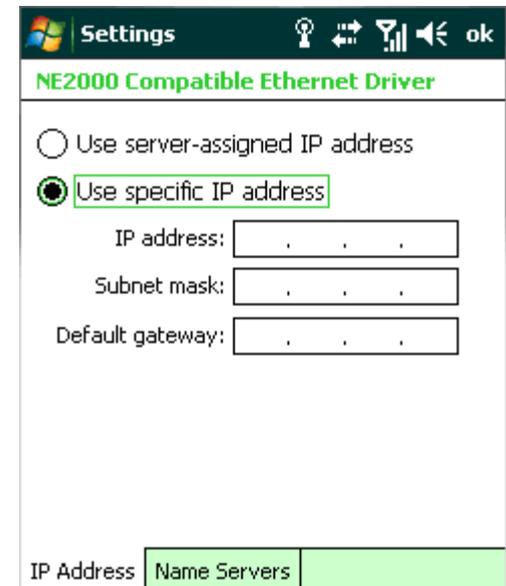


network connection in the Device Emulator

To simulate real-world environment, sometimes you must have a network connection in the Device Emulator. To do that, you have to set up the host network adapter that you want to connect to in the emulator.

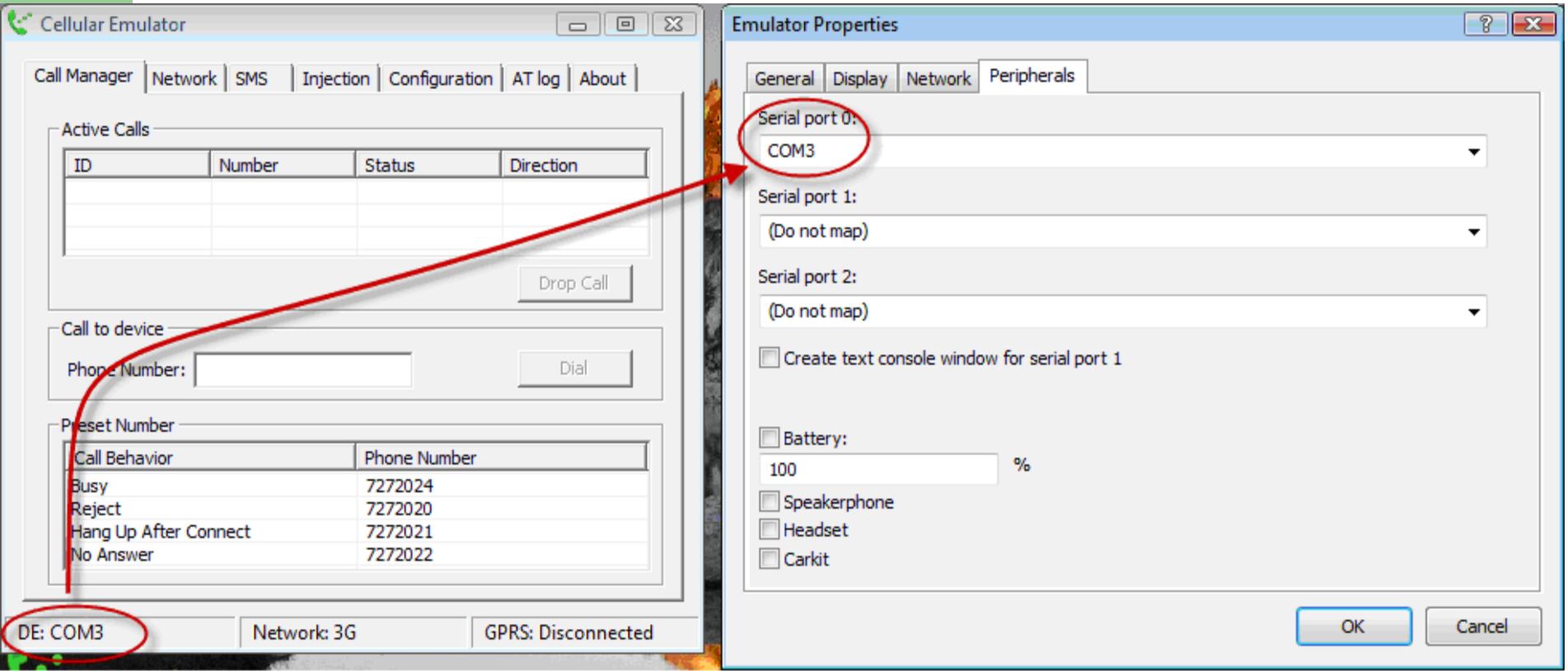


After the network adapter is set up, you have to set up the network adapter in the emulator :



Cellular Emulator

Cellular Emulator is part of Windows Mobile SDK Tool. It emulates the presence of mobile operator network that enables developers to test phone functionality in their applications from a Device Emulator. To configure Cellular Emulator to communicate with Device Emulator, you have to map its COM port to the physical port of Device Emulator.



When it is run and configured, the Cellular Emulator enables Windows Mobile Professional Device Emulator to send and receive phone calls and SMS messages, and data services, such as GPRS and 3G.

Cellular Emulator

Call Manager | Network | SMS | Injection | Configuration | AT log | About

Active Calls

ID	Number	Status	Direction
1	123456	Active	MT

Drop Call

Call to device

Phone Number: Dial

Preset Number

Call Behavior	Phone Number
Busy	7272024
Reject	7272020
Hang Up After Connect	7272021
No Answer	7272022

DE: COM3 | Network: 3G | GPRS: Disconnected

WM 6.1.4 Professional

File | Flash | Help

Phone 35638

Connected: 00:21

1 (234) 56

Speaker On Mute Hold

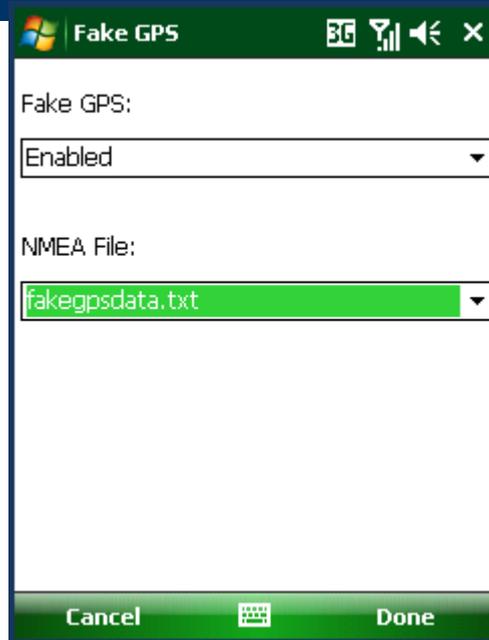
Note Contacts End

Keypad Menu

Fake GPS

Fake GPS enables you to use GPS functionalities from Device Emulators or physical devices that do not have GPS, or to test applications in scenarios where you do not have GPS coverage.

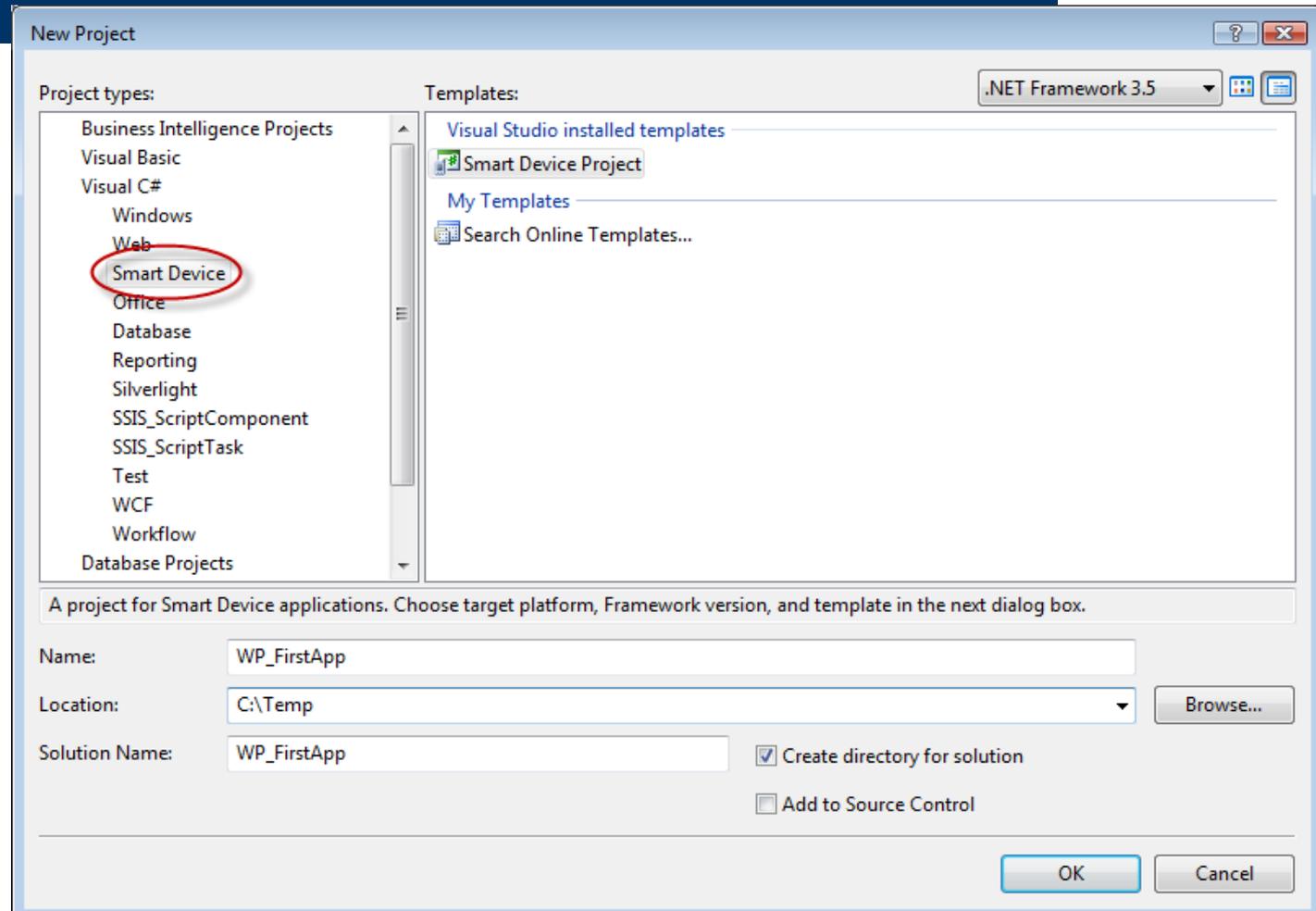
Fake GPS is a small application that you have to install with Device Emulator. When it is enabled, it runs like a service on the emulator. It randomly provides GPS coordinates from one of the files that are included to applications that use the assembly. The application behaves the same way it will with a physical GPS on a device and reads real GPS coordinates from a GPS device.

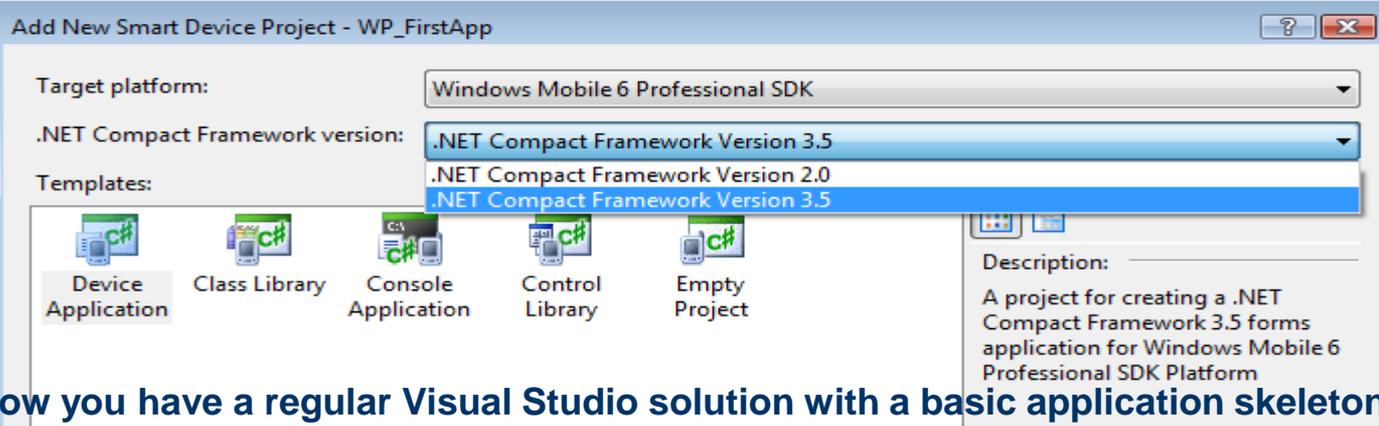


You have to install the following software products and tools to your development environment in order to get started:

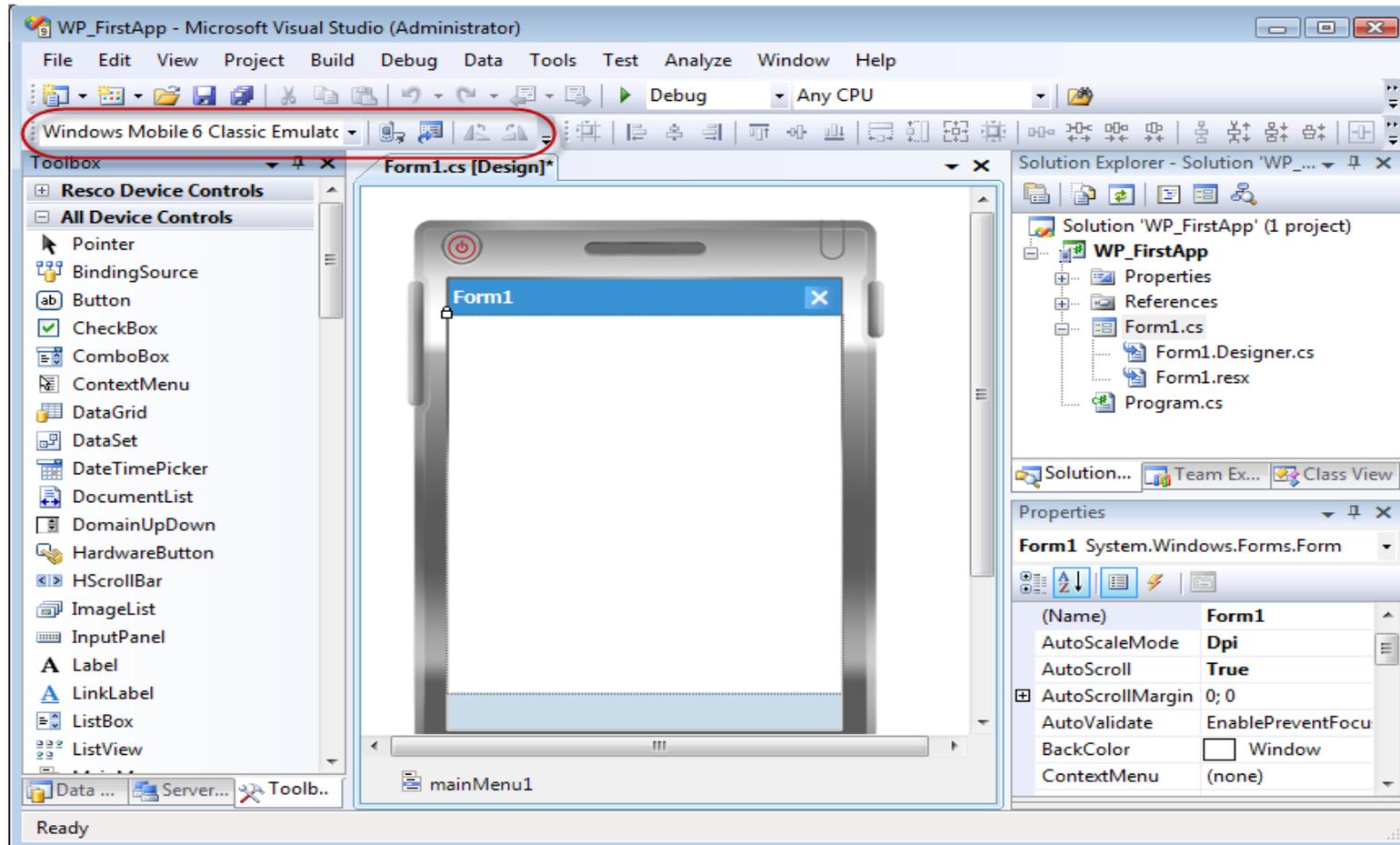
- Visual Studio 2005 or 2008
- Latest Visual Studio, .NET Framework and .NET Compact Framework updates
- Windows Mobile SDK Tools Standard or Professional, or both
- Latest emulators to be supported, not included in SDK Tools
- SQL Server to work with local data stores

You are now ready to start and develop your first Windows Mobile solution.





Now you have a regular Visual Studio solution with a basic application skeleton



Creating the user interface

When you design the user interface for mobile devices, you have to be aware of the limitations of the small form factor, and design the user interface accordingly.

One of the approaches that is very common in desktop applications, **scrolling**, is bad practice on

mobile devices. It is very user unfriendly to scroll up and down on a mobile device and should be avoided when it is possible. Instead, use different approaches for that, such as Tab controls, panels, or similar approaches.

Menus on Windows Mobile Devices are located on the bottom of the screen. They are created in the same manner as menus in desktop applications. However, it is good practice not to nest them more than three levels down, because they become unreadable after that. Mobile forms allow for two menu items that correspond to a left and a right soft key, Soft Key 1 and Soft Key 2. They can have nested menus as necessary. It is common practice to put the default action on the left soft key, and all actions and nested menus on the right soft key.

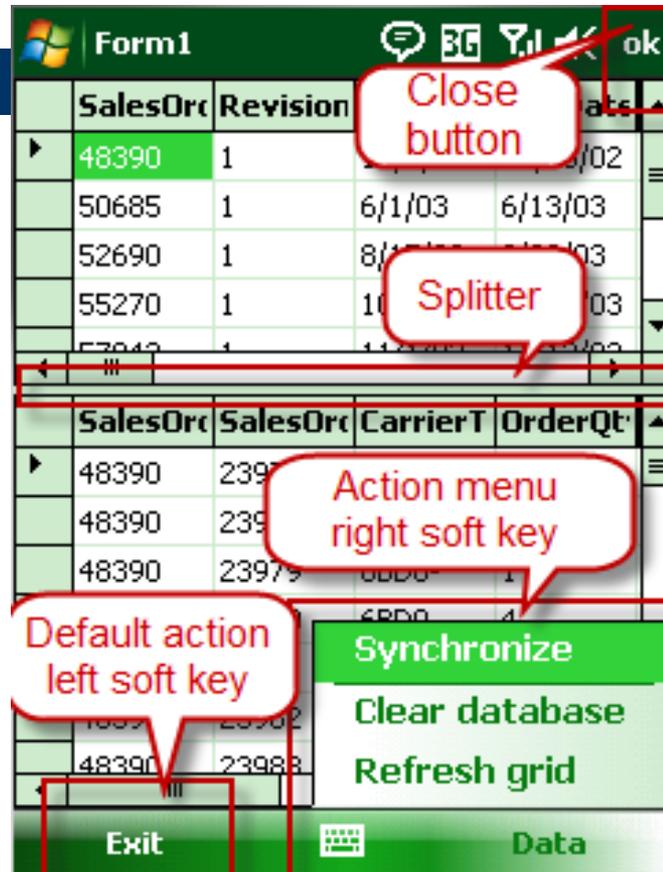
Windows Mobile 6 devices support **orientation change**. Therefore, applications that are written for a certain orientation become user unfriendly when the orientation is changed. Automatic scroll bars are supported, but as mentioned earlier, scrolling should be avoided when possible.

Controls available in .NET CF are all standard controls (text box, label, common dialog boxes, tab, grid, and more) plus many advanced controls (Web browser) from the .NET Framework.

In Windows Mobile forms, the **X** in the upper-right corner means minimize the window, instead of closing it. To close it, you have to set the MinimizeBox property of the form to False.

Then **ok** appears that means close the window.

Typical user interface for mobile:



Handling inputs

Compared to standard desktop input methods, keyboard and mouse, mobile devices handle input somewhat differently. **Keyboard** became a standard option in many mobile devices so that users can use it as an input device for mobile devices, but more common options are **Taps, Software Input Panel (SIP), and Hardware keys.**

Tap is to a mobile device, what the left mouse double-click is to the desktop. It is a common input

method on touch-screen devices. **Tap-and-hold** corresponds to right-click. It opens the shortcut menu for selected items. When tap is used on a control in a Windows form, it starts all corresponding events for that control: **MouseDown, MouseUp, MouseMove, Click, DoubleClick.**

From a programmatic perspective, it is the equivalent to a mouse click, and you can handle all these events in your application just as you handle them in desktop applications.

The **soft input panel** lets users accept keyboard input without a physical keyboard. SIP in combination with Tap provides a real keyboarding experience for users, invoking events associated with pressing keys on the keyboard: **KeyUp, KeyDown, and KeyPress.**

123	1	2	3	4	5	6	7	8	9	0	-	=	←
Tab	q	w	e	r	t	y	u	i	o	p	[]	
CAP	a	s	d	f	g	h	j	k	l	;	'		
Shift	z	x	c	v	b	n	m	,	.	/		↵	
Ctl	á	ü	`	\						↓	↑	←	→

The Input panel is a .NET CF control, and when you add it to the form, it can be programmatically controlled from the code. You can see it when the text box control receives the focus:

```
inputPanel1.Enabled = true;
```

Error handling

Error handling in .NET CF is performed exactly like in the .NET Framework, by using a **try....catch....finally** structure

Working with data

.NET CF supports working with various data sources. The concept is the same as the .NET Framework and most important, **ADO.NET** is supported in most of its functionality. Windows Mobile applications can use **SQL Server CE, XML, SQL Server, Web services** as data sources.

XML is a common format that most data sources can export data to, and import data from. Even though it can be used by most data sources, we do not recommend it as a data store, because it has a large overhead, it is not optimized, and is much slower than a relational database, because it has no indexes. However, if there is no other option, XML can be used as a data store on the device. SQL Server can directly be accessed from devices as a data store. It is a great concept if there is no need for an offline solution.

Web Services service are a good choice for transferring data in the n-tier environment when you do not have direct access to SQL Server (either for direct data access or for replication).

Deployment and packaging

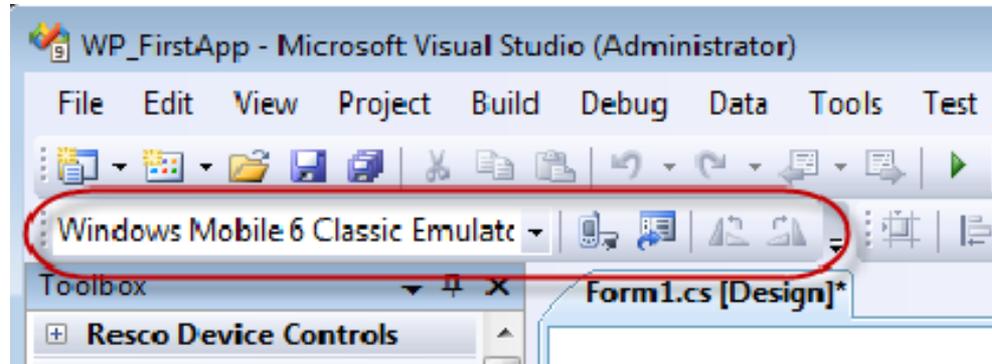
The last step in the project development cycle is to deploy the completed solution to the device and to package it for distribution and mass deployment.

There are several ways with which an application can be deployed to the device or emulator: Development IDE deployment, Device Installer, or Desktop Installer:

Development IDE installer

The easiest way to deploy the application is when the device is connected to the development workstation through ActiveSync or WMDC.

The first step is to select the deployment device or emulator.



When you select it, you can connect to the target device manually from the toolbar or the deployment process connects automatically if it is not connected already.

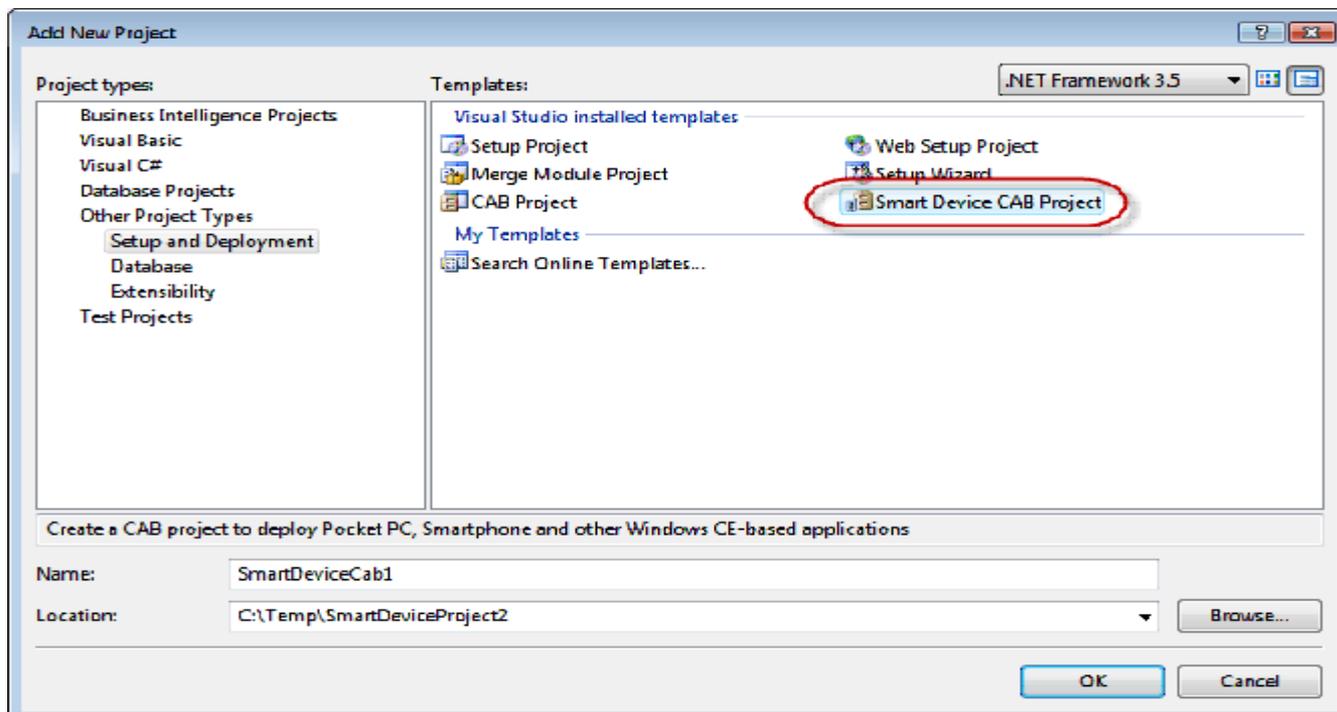
Device installer

It is not very likely that every device that is used in production arrives at the developer's desk for application deployment.

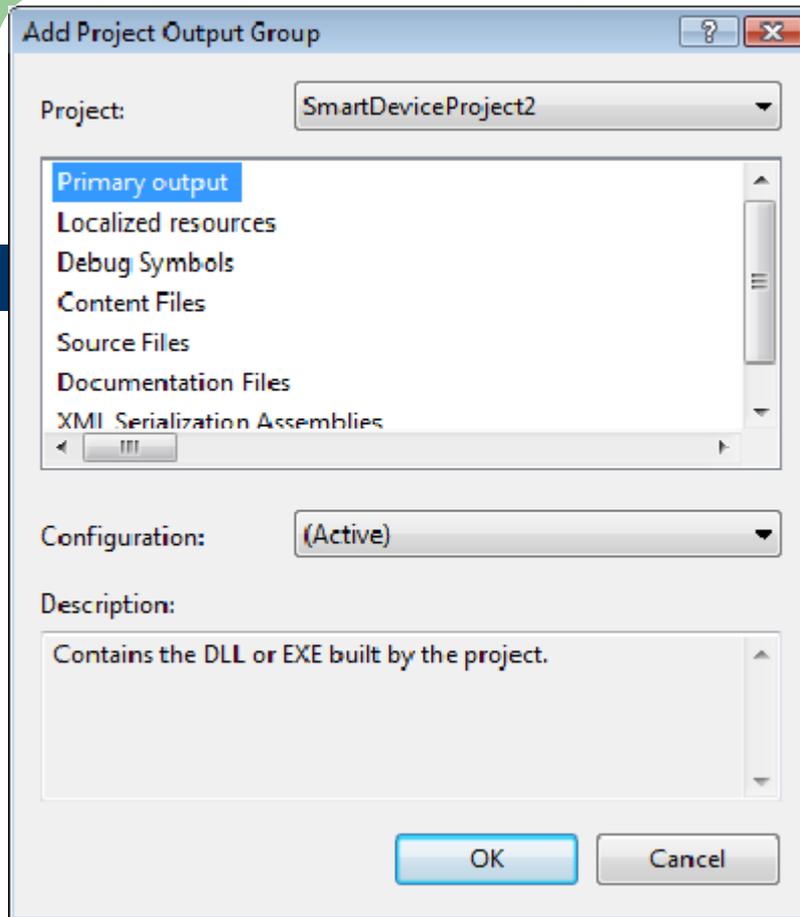
Device Installer is the method that enables applications to be easily deployed directly on the device, without any need for a connection between the device and the developer workstation. All assemblies and dependencies are packed in a single (or more) CAB file which is copied to the device and

executed there. The CAB execution on the device installs it with predefined parameters. The CAB file can be distributed through e-mail, a Web server, external cards, or by coping it through ActiveSync or WMDC.

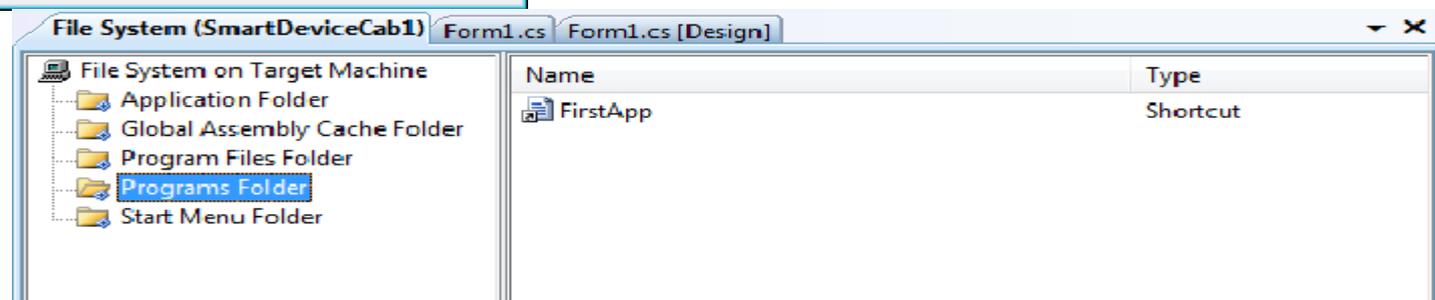
In the Add New Project dialog box, under Templates, there is a project template in Visual Studio to enables you to create CAB projects:



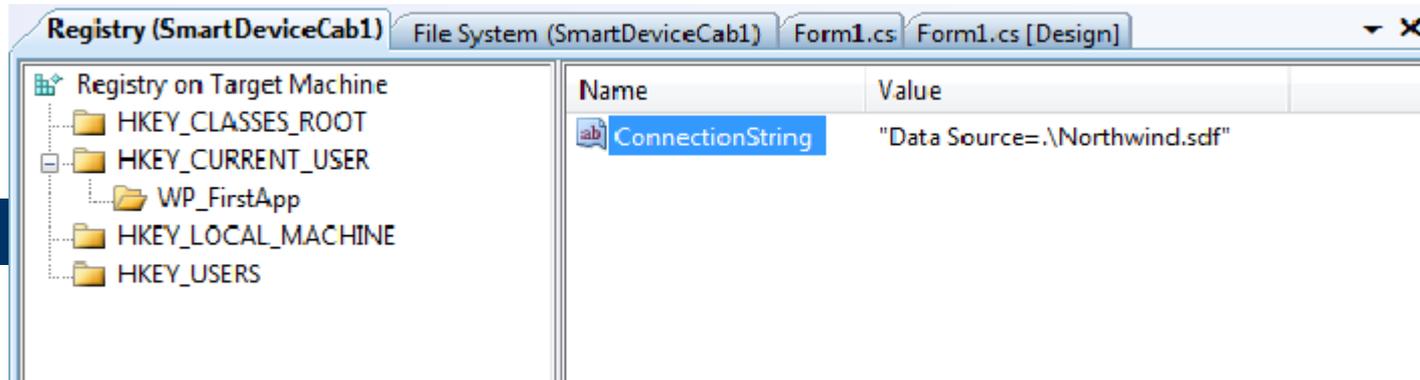
In the CAB project, you have to define project output, which is the application that you just finished



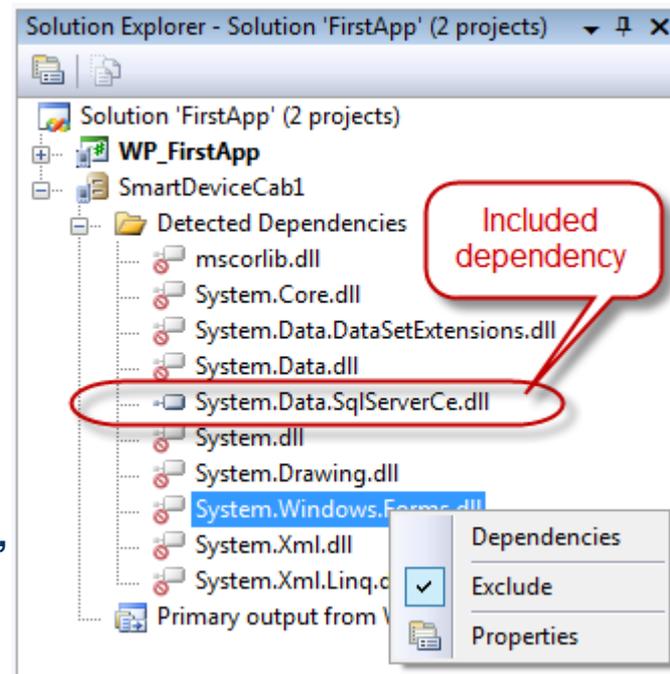
After the project output is set, it is automatically added to the application folder for installation



Through the Deployment project, you can also alter registry settings and add new keys for your application



Even though it is good practice to distribute and install dependencies (.NET CF, SQL Server CE) separately to keep the CAB file as small as possible, you can also include dependencies into the final CAB. By default, they are excluded:



When all parameters for the Deployment project are set, you can build it. The result of the building action is a redistributable, self-executing CAB file, ready to be distributed and deployed directly on devices.

Pocket outlook object model

The POOM API is part of the **Microsoft.WindowsMobile.PocketOutlook** library. Through POOM, you can access **Calendar, Contacts and Tasks** on the device.

All activities that use POOM functionality are based on the OutlookSession class. Therefore, in your applications, you must always have an instance of that object at the application level.

The following code example shows that an OutlookSession object is declared the first time that it appears and that an instance of it is created.

```
using Microsoft.WindowsMobile.PocketOutlook;
namespace WP_FirstApp
{
    public partial class Form1 : Form
    {
        private OutlookSession outlook;

        private void Main_Load(object sender, EventArgs e)
        {
            outlook = new OutlookSession();
        }
    }
}
```

The following code example shows that it is very easy to create a new task from the application.

```
Task task = new Task();
task.Subject = textBox1.text;
task.Body = textBox2.text;
outlook.Tasks.Items.Add(task);
```

New task will be added to Pocket Outlook **Tasks**, and can be managed either from code or from **Tasks** directly. Existing tasks can also be altered from the application.

You can also manage **Contacts** from the application, using the same parent object as for Tasks. When you create a new contact, all fields are optional. However, some basic set of information should be set, to recognize the contact later.

The following code example shows the code for creating a new contact.

```
    Contact con = outlook.Contacts.Items.AddNew();
    Con.FirstName = textBox1.text;
    Con.LastName = textBox2.text;
    Con.Update();
```

To select an existing contact, use the **ChooseContactDialog** object. This object can be used to access a specific contact or a specific property of a contact. ChooseContactDialog enables you to filter contacts to be shown to the user, by using the RestrictContacts property and setting the filter. The following code example shows how to use an instance of the ChooseContactDialog class to present a Contacts dialog box to the user and how to retrieve the business telephone number of the contact if a contact was selected by the user from the dialog box.

```
    ChooseContactDialog dialog = new ChooseContactDialog();
    dialog.RestrictContacts = "[Department] = \"Field\"";
    dialog.Title = "Assign To";
    if (dialog.ShowDialog() == DialogResult.OK)
    {
        str = dialog.SelectedContact.BusinessTelephoneNumber;
    }
```

E-mail

E-mail functionality is accessed through the **EmailMessage object**. The EmailMessage object contains all standard properties that e-mail messages usually have: Attachments, Bcc, BodyText, CC, Importance, Subject, To and more.

The following code example show how to create a new e-mail message, set properties, and send an e-mail message to the contact previously selected.

```
EmailMessage email = new EmailMessage();
email.To.Add(new Recipient(dialog.SelectedContact.Email1Address));
email.Subject = textBox1.text;
email.BodyText = textBox2.text;
email.Send("ActiveSync");
```

Handling SMS

Handling SMS resembles handling e-mail. Except for having fewer properties, the main difference is that when it calls a Send method, an SMS message is sent immediately, whereas an e-mail message waits for the next synchronization.

The following code example shows how to create a new SMS message and send it.

```
SmsMessage sms = new SmsMessage();
sms.To.Add(new Recipient("123456"));
sms.Body = "WP_FirstApp SMS message";
sms.Send();
```

SMS interception

SMS messages can easily be intercepted by the application by using the **MessageInterceptor** object.

The MessageInterceptor object is contained in the

Microsoft.WindowsMobile.PocketOutlook.MessageInterception namespace. Messages to be intercepted can be filtered by the message body, the message subject or sender.

SMS Interception can be used even if the application is currently not running.

The following example code shows how to implement SMS interception.

// SMS Interception Setup

```
smsInterceptor = new MessageInterceptor( InterceptionAction.NotifyAndDelete );
smsInterceptor.MessageCondition = new MessageCondition(
    MessageProperty.Sender,
    MessagePropertyComparisonType.Equal,          "+14254448851");
smsInterceptor.MessageReceived += new MessageInterceptorEventHandler
    (smsInterceptor_MessageReceived);
```

// SMS Interception event handler

```
private void smsInterceptor_MessageReceived(object sender, MessageInterceptorEventArgs e )
{
    if (e.Message.GetType() == typeof(SmsMessage))
    {
        SmsMessage message = (SmsMessage)e.Message;
    }
}
```

```
//do something with message
```

```
....
```

Telephony

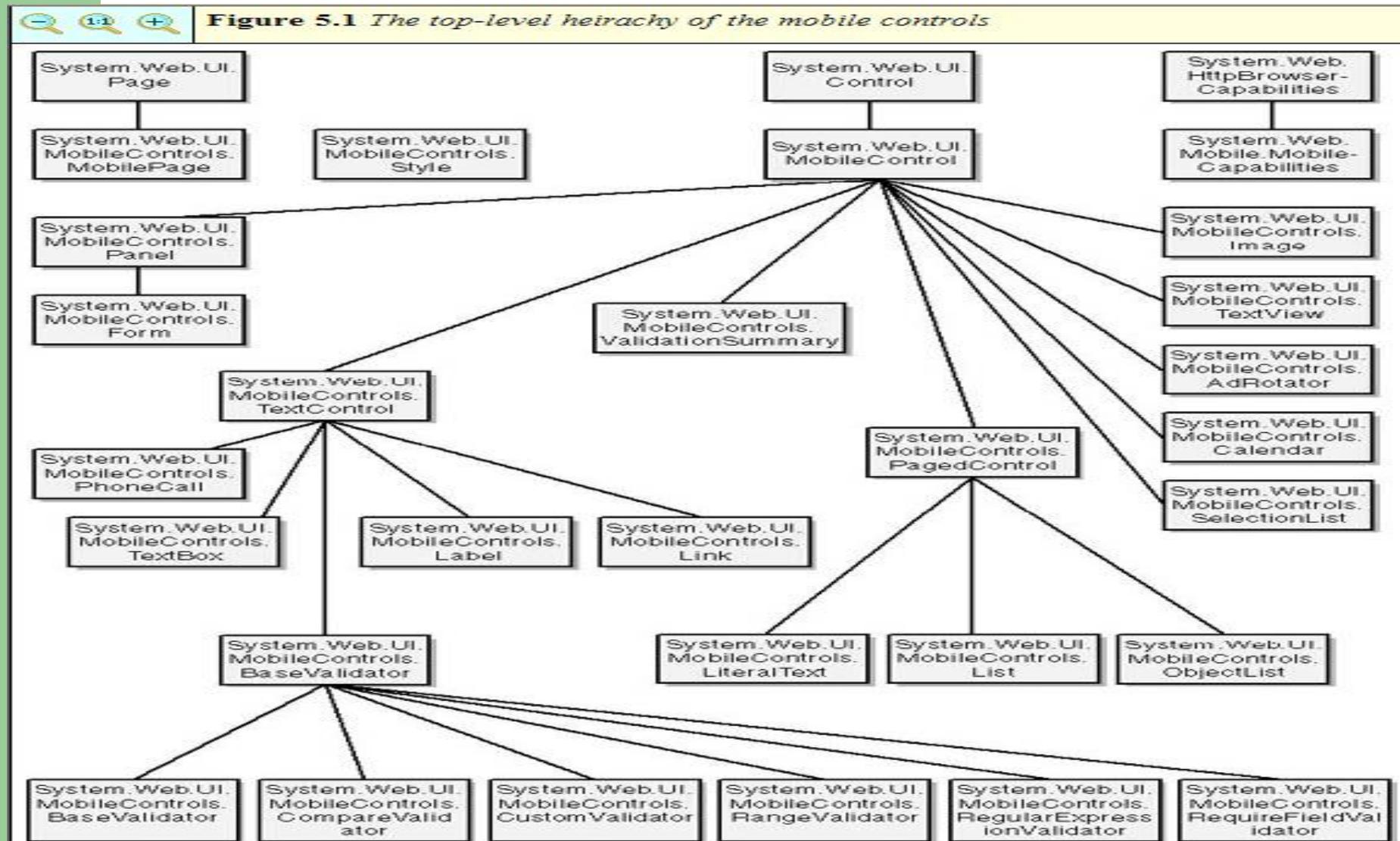
Making voice calls from applications enables you to use phone functionality directly from the application, without having to go to a device dialer. The Phone class is in the **Microsoft.WindowsMobile.Telephony library**. Use the Phone class to establish voice calls. The following example code shows how to establish voice calls by using the contact that you previously selected.

```
using Microsoft.WindowsMobile.Telephony;

private void MakeVoiceCall()
{
    Phone phone = new Phone();
    phone.Talk(dialog.SelectedContact.BusinessTelephoneNumber);
}
```

Mobile Internet Toolkit (.NET) Standard Controls

we'll describe some of the standard controls of the Mobile Internet Toolkit and demonstrate how to use the most important properties, methods, and events of the associated mobile control class.



A developer working in an object-oriented language such as C# or C++ defines program classes in code modules. The .aspx file is the place for defining a class that is specific to ASP.NET.

In mobile Web applications, this class must descend—directly or indirectly—from the `System.Web.UI.MobileControls.MobilePage` class.

A mobile Web Forms page declares itself as a descendant of the `MobilePage` class through the `@Page` directives at the top of the .aspx file, which you can write like this:



```
<%@ Page Inherits="System.Web.UI.MobileControls.MobilePage" Language="c#"%>  
<%@ Register TagPrefix="mobile" Namespace="System.Web.UI.MobileControls"  
                        Assembly="System.Web.Mobile" %>
```

If you have a code-behind module, this page must inherit from a class in the code-behind module, which must itself inherit from the `MobilePage` class.

In such instances, the `@Page` directive must specify the name of the code-behind module and the class within it:

```
<%@ Page Codebehind="Default.aspx.cs" Language="c#" Inherits="MyMobileWebForm"%>
```

So, the file Default.aspx :

```
<%@ Page CodeBehind="Default.aspx.cs" Language="c#" Inherits="MyMobileWebForm"
    AutoEventWireup="True"%>
<%@ Register TagPrefix="mobile" Namespace="System.Web.UI.MobileControls"
    Assembly="System.Web.Mobile" %>
<mobile:Form runat="server">
    <mobile:Label runat="server"/>
</mobile:Form>
```

means that methods such as Page_Load or Page_Init are “wired-up” automatically. Visual Studio .NET sets AutoEventWireup to False by default in the mobile Web Forms

And, the Code-behind module Default.aspx.cs:

```
using System;
public class MyMobileWebForm : System.Web.UI.MobileControls.MobilePage
{
    protected System.Web.UI.MobileControls.Label Label1;
    private void Page_Load(object sender, System.EventArgs e)
    {
        Label1.Text = "Hello World";
    }
}
```

pages it creates and automatically inserts the code needed to wire up the Page events.

The XML text in Default.aspx represents the server control syntax for a Label control contained within a Form control.

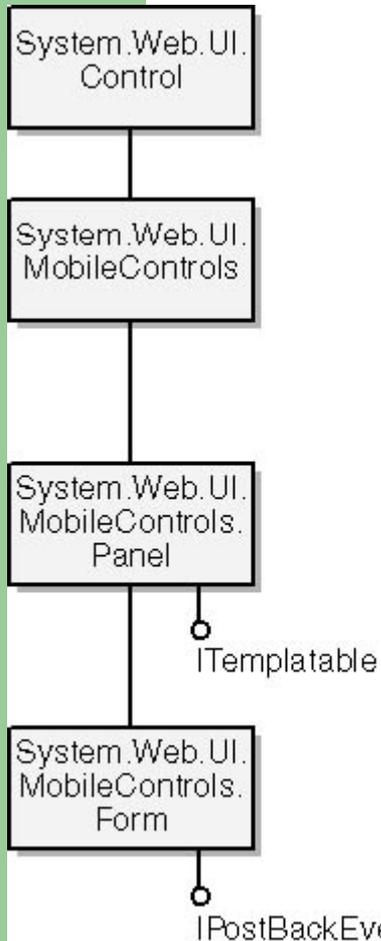
When this code compiles, the runtime actually creates an instance of the MobilePage-derived class, which contains an instance of the Form class, and in that, an instance of the Label class.

Container Controls

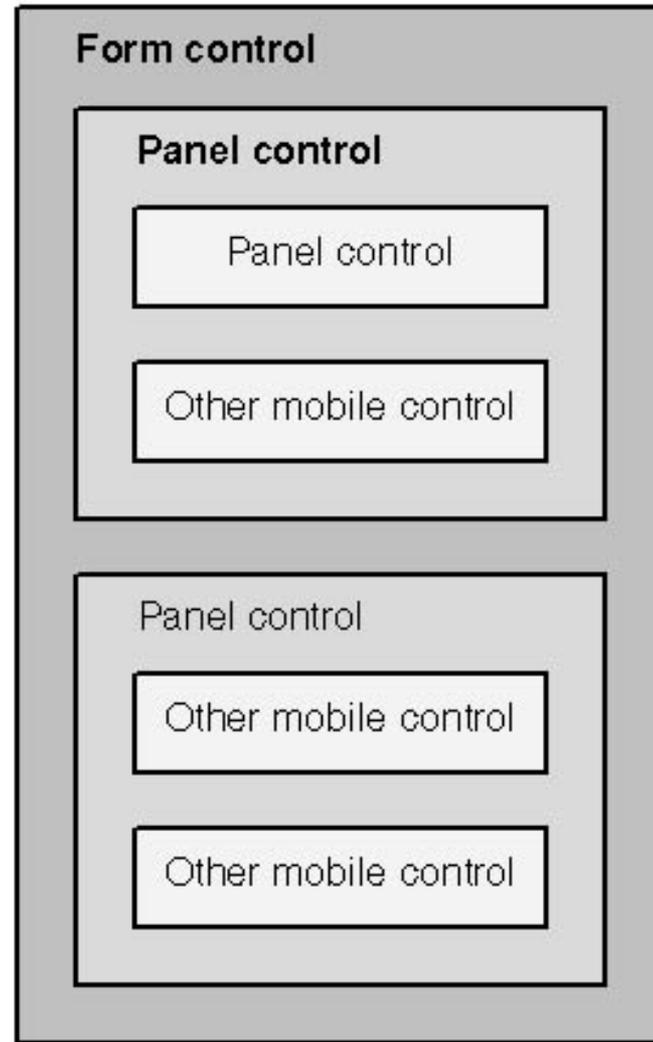
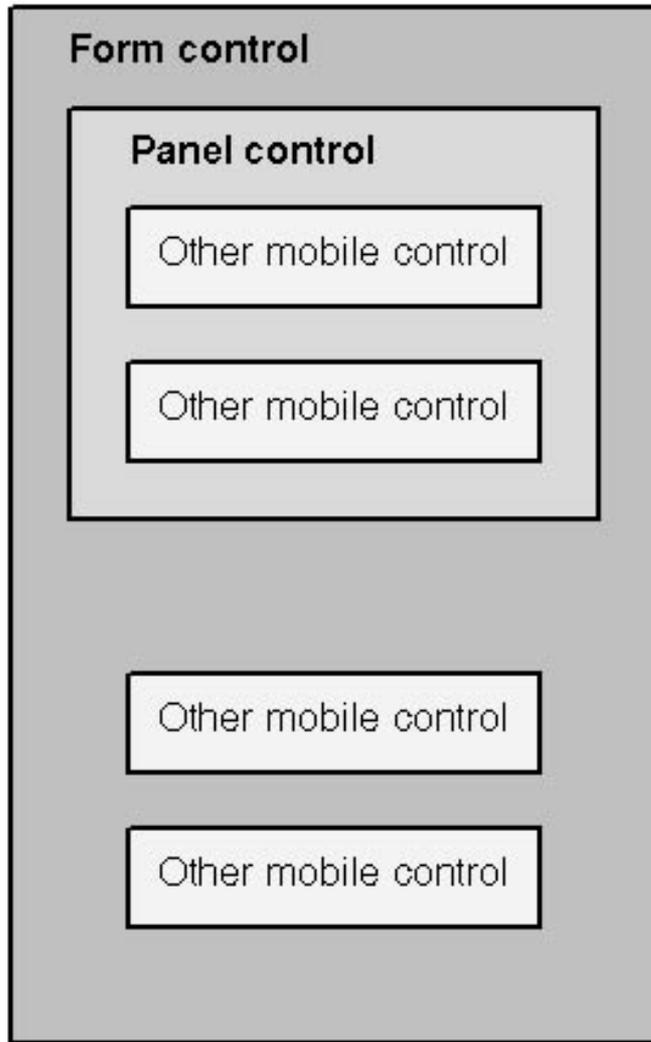
Container controls provide a powerful means of structuring your Web application.

All mobile Web Forms pages derive from the **MobilePage class**, which itself derives from the **ASP.NET Page class**. Therefore, every mobile Web Forms page is a valid MobilePage object. Each MobilePage object must contain one or more **Form controls**, which, as we've mentioned, you use to group controls into programmatically accessible objects.

A mobile Web Forms page can contain more than one Form control; however, you can't nest (or overlap) these controls. Each Form control can contain one or more mobile controls. These mobile controls can be of any type, except other Form controls or style sheets. You can include one or more Panel controls within a Form control; doing so will allow you to dictate the appearance of groups of controls. A Form control can contain zero or more Panel controls. Unlike Form controls, you can nest Panel controls. For example, Panel control A can contain Panel control B, and Panel control B can contain Panel control C.



Mobile Web Forms page



<mobile:Form

```
runat="server"  
id="id"  
Font-Name="fontName"  
Font-Bold="{NotSet | False | True}"  
Font-Italic="{NotSet | False | True}"
```

```
ForeColor="foregroundColor"  
BackColor="backgroundColor"  
Alignment="{NotSet | Left | Center | Right}"  
Visible="{True | False}"
```

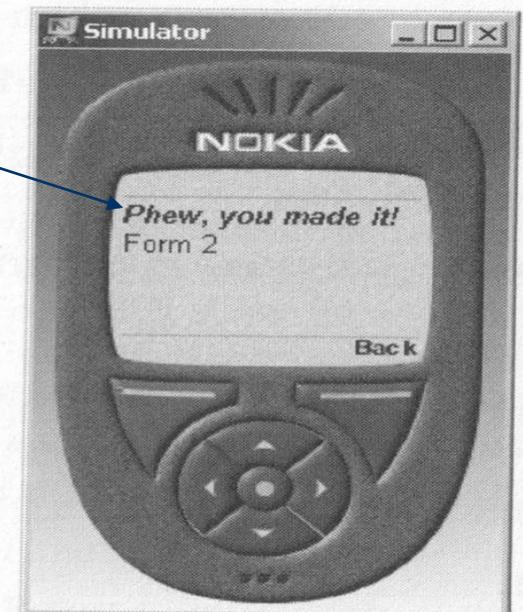
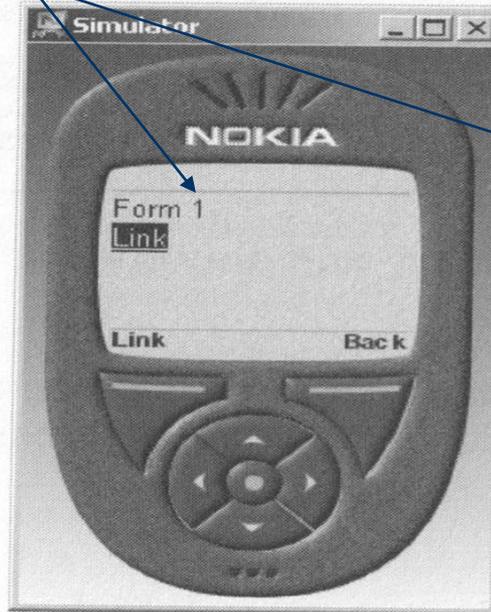
```
Action="url"  
Method="{Post | Get}"  
OnActivate="onActivateHandler"  
OnDeactivate="onDeactivateHandler"  
OnInit="onInitHandler"  
Paginate="{True | False}"  
PagerStyle-NextPageText="text"  
PagerStyle-PageLabel="text"  
PagerStyle-StyleReference="styleReference"  
Title="formTitle"> Child controls
```

</mobile:Form>

Source code for FormExample.aspx

```
<%@ Register TagPrefix="mobile" Namespace="System.Web.UI.MobileControls" Assembly="System.Web.Mobile" %>
<%@ Page language="c#" Inherits="System.Web.UI.MobileControls.MobilePage" %>
<mobile:Form id="Form1" runat="server">
  <mobile:Label id="Label1" runat="server"> Form 1 </mobile:Label>
  <mobile:Link id="Link1" runat="server" NavigateUrl="#Form2"> Link </mobile:Link>
```

```
</mobile:Form> <mobile:Form id="Form2" runat="server">
  <b> <i>Phew, you made it!</i> </b>
  <br>
  <mobile:Label id="Label2" runat="server"> Form 2 </mobile:Label>
</mobile:Form>
```



State Management in mobiles

When you build dynamic Web applications, you usually need a mechanism to store information between client requests and server responses. Unfortunately, Hypertext Transfer Protocol (HTTP) is effectively **stateless**, which means you must maintain state in some other way.

In the past, developers often used cookies to track—and thus identify—a user with a session ID and to reconcile information stored on the server to that user

Following are four significant methods ASP.NET offers for preserving state:

1. **Session state** Allows you to maintain the variables and objects for a client over multiple requests and responses.
2. **Hidden variables** Allow you to persist objects between server round-trips by posting the data to the client as hidden fields.
3. **View state** Allows you to maintain the values of a mobile Web Forms page on the server. The runtime stores this information in an instance of the StateBag class, which itself gets stored within the session. We'll discuss later (in the section “View State,”) the server then sends some information to the client.
4. **Application state** Allows you to maintain the variables and objects of an application over multiple requests by multiple clients.

1. Managing Session State

ASP.NET offers an updated and improved version of the Session object. This object allows you to perform the following tasks:

- Identify a user through a unique session ID.
- Store information specific to a user's session.
- Manage a session lifetime through event handler methods.
- Release session data after a specified timeout

In ASP.NET, the **Session object** is a generic term.

The **Session** property of the **System.Web.HttpApplication class** (the parent class of the Global.asax page) and the **Session** property of the **MobilePage class** (the parent class of your mobile Web Forms page) both give access to the Session object.

Typically, you'll manipulate the Session object either in

1. the code-behind module of your application's Global.asax file or
2. the code-behind module of your mobile Web Forms page.

Like mobile Web Forms pages, the Global.asax file supports a code-behind module. This module follows the naming convention `global.asax.extension`, where `extension` indicates the programming language used. For example, you'd name a C# code-behind module `Global.asax.cs`.

Following is a fragment of code that adds a string representing the user's start time to the Session object with a key of `UserStartTime` from within the Global.asax file.

A way of adding items to the Session object is shown: the use of `Add` method to define an entry with the key `HelpAccess`.

Global.asax.cs file for the SessionObjectExample project:

```
....  
namespace SessionState1  
{  
    public class Global : System.Web.HttpApplication  
    {  
        protected void Session_Start(Object sender, EventArgs e)  
        { Session["UserStartTime"]=DateTime.Now.ToLongTimeString();  
          Boolean HelpAccess=false;  
          Session.Add("HelpAccess",HelpAccess);  
        }  
    }  
}
```

Constructed for every new session

The Global.asax file that references the code-behind module consists of a single line containing just an @ Application directive:

```
<%@ Application Codebehind="Global.asax.cs" Inherits=" SessionObjectExample.Global" %>
```

MobileWebForm1.aspx of project *SessionObjectExample*:

```
<%@ Register TagPrefix="mobile" Namespace="System.Web.UI.MobileControls"
    Assembly="System.Web.Mobile" %>
<%@ Page language="c#" Codebehind="MobileWebForm1.aspx.cs"
    Inherits="MobileWebForm1" AutoEventWireup="true" %>
<mobile:Form id="Form1" runat="server">
    <mobile:Label id="Label1" runat="server"/>
    <mobile:Command id="Command1" runat="server">Go To Help</mobile:Command>
</mobile:Form> <mobile:Form id="Form2" runat="server">
    <mobile:Label id="Label2" runat="server">    This is a help page.    </mobile:Label>
    <mobile:Label id="Label3" runat="server"></mobile:Label> </mobile:Form>
```



MobileWebForm1.aspx.cs of project *SessionObjectExample*:

```
public class MobileWebForm1 : System.Web.UI.MobileControls.MobilePage
{
    protected System.Web.UI.MobileControls.Label Label1;
    protected System.Web.UI.MobileControls.Label Label3;
    protected System.Web.UI.MobileControls.Command Command1;

    protected System.Web.UI.MobileControls.Form Form1;
    protected System.Web.UI.MobileControls.Form Form2;
    private void Page_Load(object sender, System.EventArgs e)
    {
        Label1.Text = "Help accessed: "; //
        Label1.Text += Session["HelpAccess"].ToString();
        Command1.Click += new System.EventHandler(Command1_OnClick);
    }
    private void Command1_OnClick(object sender, System.EventArgs e)
    {
        //Switch to the Help form, set the flag in Session object
        Session["HelpAccess"] = true;
        Label3.Text = "Help accessed: ";
        Label3.Text += Session["HelpAccess"].ToString();
        ActiveForm = Form2;
    }
}
```

When this simple application executes, the `HelpAccess` flag in the `Session` object is initialized as `false` in `Global.asax.cs`. When the Web Form displays, the label on `Form1` displays a message to show that this is the case.

When the user clicks the Command control marked `Go To Help`, the `Command1_Click` event handler in the code-behind module executes, setting the `HelpAccess` flag to `true` and setting the text of `Label3` to reflect this new state.

2. Working with Cookies

Cookies provide an invaluable way for Web servers to identify wired clients such as HTML desktop browsers. However, the potential of cookies is limited with regard to applications for wireless clients. This is because many wireless devices, including some Wireless Application Protocol (WAP) and i-mode devices, don't support cookies.

If you know that your target devices support cookies or that a proxy supports them on the client's behalf, as is the case with some WAP gateways, cookies provide an excellent way to track and identify sessions.

3. Hidden Variables

Sometimes you might want to pass small amounts of information between Web pages without using session state. For example, suppose you need to collect information from a form that the user fills in. In HTML, you'd pass the information from one page to another using input tags with a type value of hidden.

The MobilePage class's **HiddenVariables** property provides this type of functionality. This property allows you to store variable name-value pairs, which the runtime then passes back and forth between the server and the client as hidden fields.

You should use HiddenVariables only for small amounts of information.

There are a number of reasons for this, including the following:

- Many mobile devices have limited bandwidth.
- WAP devices only support compiled deck sizes of up to approximately 1.4 KB. (A deck is the outermost element of a file of WML content. Each WML file must support exactly one deck.)

Example: ...HiddenVariables.Add(TextBoxName.ID, TextBoxName.Text);

4. View State

ASP.NET gives the user the impression that the runtime maintains pages over several server round-trips. The pages don't really exist over multiple requests and responses; instead, the runtime saves the properties of the page into a server control view state (an instance of the **StateBag class**). When the user makes a request, the runtime automatically reconstructs the page using the property values persisted in the StateBag instance.

For example, if you define a property in your code-behind class, that property isn't automatically saved and restored each time the page is torn down and then reconstructed on the next request. If you set this property in code on one request, you might want to persist this value across server round-trips. You could add the property to the session or even persist it by using hidden variables. However, if you use the ViewState property of the MobilePage to maintain the property's value, the runtime will automatically save and restore that value on your behalf.

```
...public String MyMessage    {
    get
    {          // Explicit cast to String
        return (String) ViewState["MyMessage"];
    }
    set
    {
        ViewState["MyMessage"]=value;
    } }

private void Command1_Click(object sender, System.EventArgs e)
{    // Consume the persisted property.
    Label1.Text=this.MyMessage;
}
}
```

When using the session with view state, you have two important considerations. First, sessions can expire, which means you can lose your view state information. The number of minutes allowed to elapse before a response is received from a client is set by the timeout attribute of the sessionState element in the application's Web.config file; 20 minutes is the default.

```
<sessionState
    mode="inProc"
    cookieless="true"
    timeout="20" />
```

Second, the page displayed on the client and the current state of the session information held on the server can fall out of sync. This can occur when a user uses a **Back** feature on the browser to return to a page viewed previously—for instance, by pressing a Back button. For example, imagine that a user goes to the first page of an application and then clicks a link to go to the second page. If the user then navigates backward to the first page, the user views the first page while the server holds session data for the application's second page. The Mobile Internet Toolkit overcomes this issue by maintaining a small history of view state information in the user's session.

You can configure the size of the view state history. The default history size is 6. To change the history size, use the sessionStateHistorySize attribute of the mobileControls element within the Web.config file, as the following code shows:

```
<configuration>
  <system.web>
    <mobileControls sessionStateHistorySize="10"/>
  </system.web>
</configuration>
```

5. Application State

In ASP.NET, an application is the total of all the files that the runtime can invoke or run within the scope of a virtual directory and all its subdirectories. At times, you might want to instantiate variables and objects that have scope at an application level rather than at a session level.

The **HttpApplicationState class** allows you to do this. The Application object is the generic term for the instance of this class for your application, which is exposed through the Application property of the System.Web.HttpApplication class (the parent class of the Global.asax page) and the Application property of the MobilePage class (the parent class of your mobile Web Forms page).

Using Application State in Global.asax

You define information that relates to application state in the **Global.asx file**, which always resides at the root of a virtual directory. For the moment, you'll use Global.asax to implement event handlers associated with application state, but remember that you can also use this file to store other information such as session state, as described earlier.

You define application state data within the code-behind module of Global.asax by writing code for the two event handler methods, Application_Start and Application_End.

Things to Consider When Using Application State

At times, you might wonder whether to use session state or application state.

First of all, information stored in application state is memory hungry.

In other words, the application holds all application state information in memory and doesn't release the memory, even when a user exits an application.

Second, all threads in a multithread application can access application data simultaneously, since ASP.NET doesn't automatically lock resources.

(2) Programming Windows Phone 7 essentials

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

Text-based controls and menus	Silverlight
Event-driven application	Silverlight
Interaction with Windows Phone controls such as Pivot and Panorama	Silverlight
Embedded video	Silverlight
Hosted HTML	Silverlight
Web browser compatibility	Silverlight
Vector graphics	Silverlight
Looping game framework	XNA
Highly performant, 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 4-point 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

When you write applications for Windows Phone 7, you should be familiar with some of the terminology.

Code named Metro design

Tile



Back

Start

Search



Status Bar

Application title

Page title

On-screen keyboard

Application Bar

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
Zune software
XNA Game Studio 4.0
Silverlight
.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

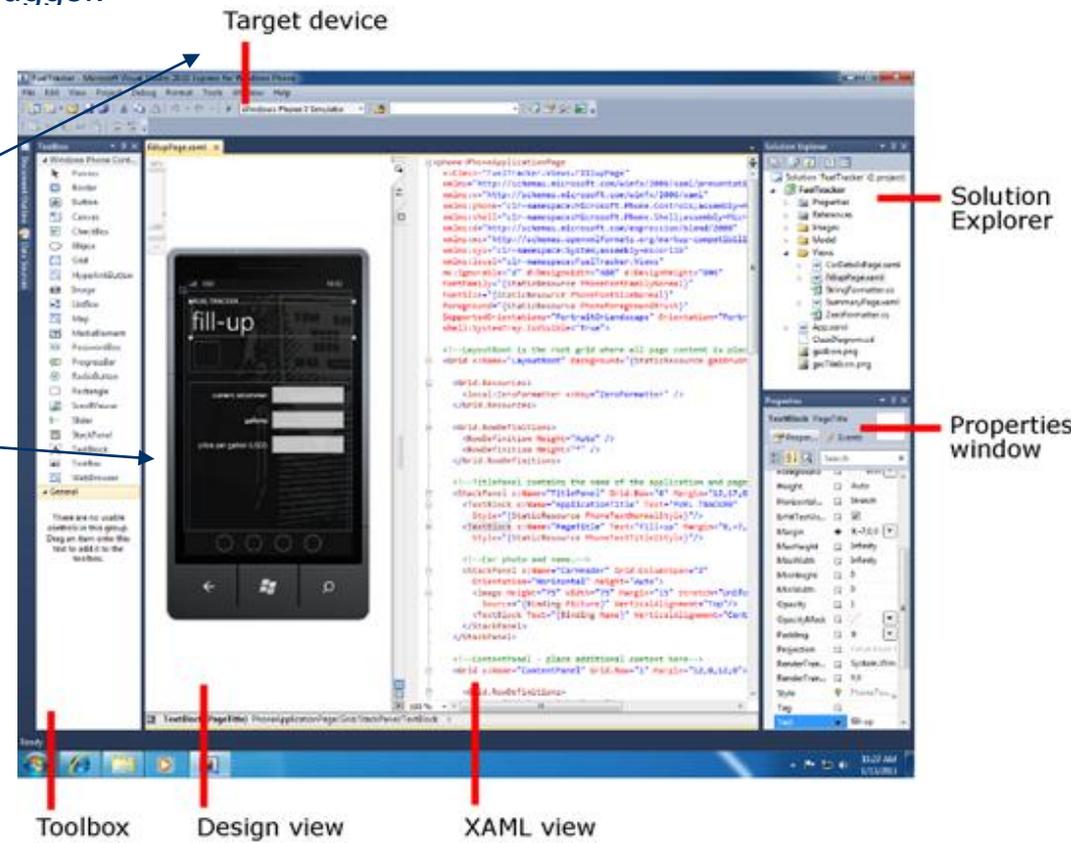
Expression Blend for Windows Phone is a design suite that allows you to create and add special visual features, such as gradients, animations, and transitions. For some tasks, Expression Blend is easier to use than Visual Studio

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.



Get Started Creating a Windows Phone 7 Application example

(SILVERLIGHT QUICKSTART FOR WINDOWS PHONE DEVELOPMENT)

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:

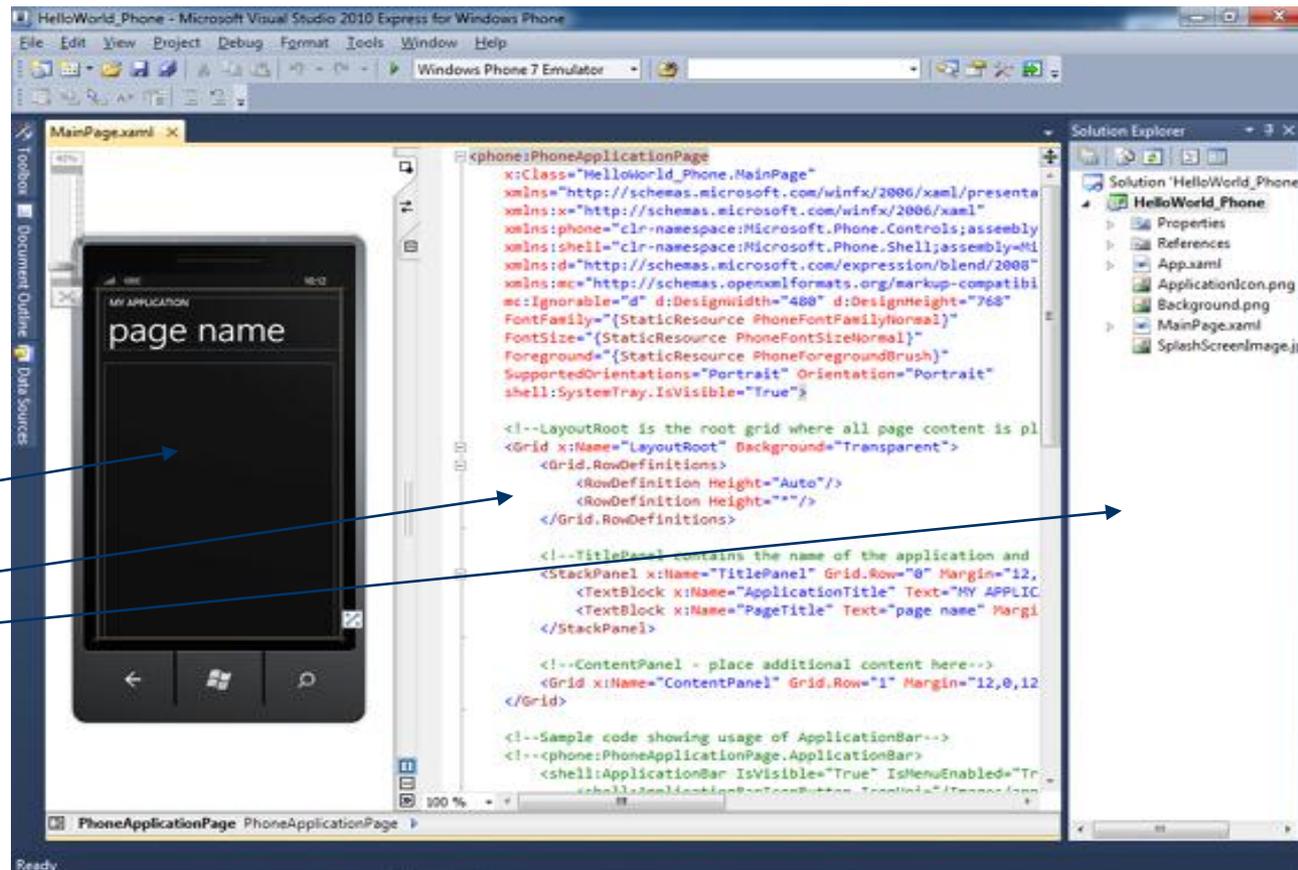
- On the Start menu, launch Microsoft Visual Studio 2010 Express for Windows Phone.
- On the File menu, click New Project

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.



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 7 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:



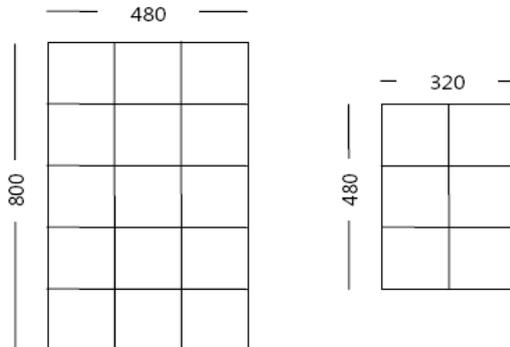
Back Programs can use this button for their own navigation needs, much like the Back button on a Web browser. From the home page of a program, the button causes the program to terminate.

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. In the future, screens of 320×480 pixels are also expected. 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 might be more adaptable.

Software:

In Solution Explorer, there are a number of project files.

You'll see two pairs of skeleton files: **App.xaml** and **App.xaml.cs**,

If you look at the **App.xaml.cs** file, you'll see a namespace definition that is the same as the project name and a class named *App* that derives from the Silverlight class *Application*

The files that you'll use are **MainPage.xaml** and **MainPage.xaml.cs**. **MainPage.xaml** defines the user interface for the application. XAML is an XML-based declarative language used to create and lay out UI elements. If you expand **MainPage.xaml**, you'll see a C# code-behind file named **MainPage.xaml.cs**. A code-behind file is joined with a XAML file through a partial class and contains the logic for the XAML file.

Separating the UI from the code allows you to create visible user interface elements in the declarative XAML markup and then use a separate code-behind file to respond to events and manipulate the objects you declare in XAML. This separation makes it easy for designers and developers to work together efficiently on the same projects.

In the standard Visual Studio toolbar under the program's menu, you'll see a drop-down list probably displaying "**Windows Phone 7 Emulator**." The other choice is "**Windows Phone 7 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 (excerpt) :

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

Silverlight Project: Helloworld_Phone File: App.xaml (excerpt) :

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** and sets that object to its own *RootVisual* property. 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 и MainPage.xaml.cs

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

Silverlight Project: Helloworld_Phone File: MainPage.xaml.cs (excerpt) :

```
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.

The other half of this *MainPage* class is defined in the **MainPage.xaml** file:

```
<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>
```

Иерархия:

PhoneApplicationFrame

PhoneApplicationPage

Grid named "LayoutRoot"

StackPanel named "TitlePanel"

TextBlock named ApplicationTitle

TextBlock named "PageTitle"

Grid named "ContentPanel"

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, and then **click Toolbox**. The Toolbox window appears.
- Resize or pin the Toolbox so that 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.

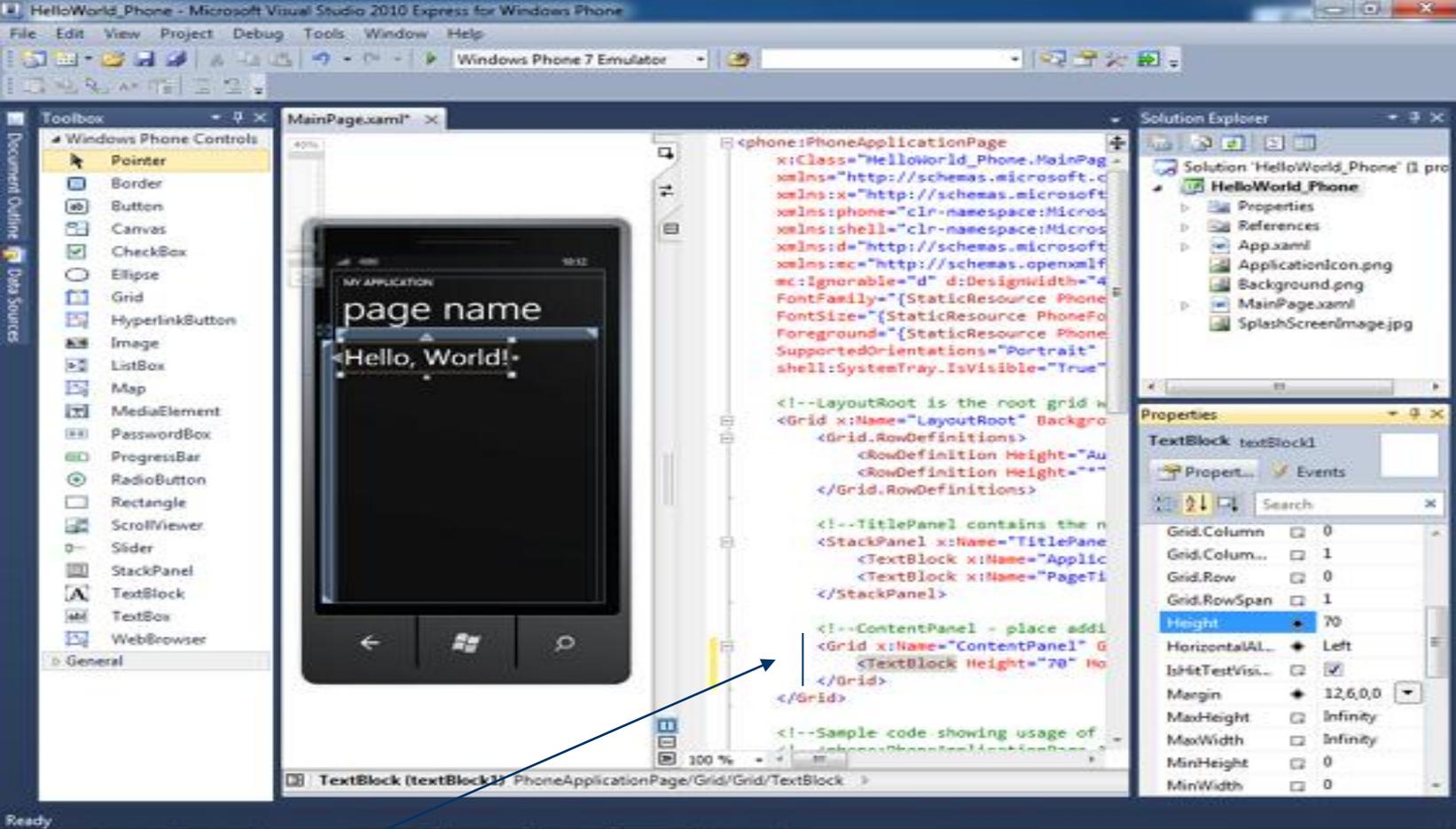
In XAML view, notice that a TextBlock element was added in the Grid content panel.

In Silverlight Project: Helloworld_Phone File: MainPage.xaml (excerpt)
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:

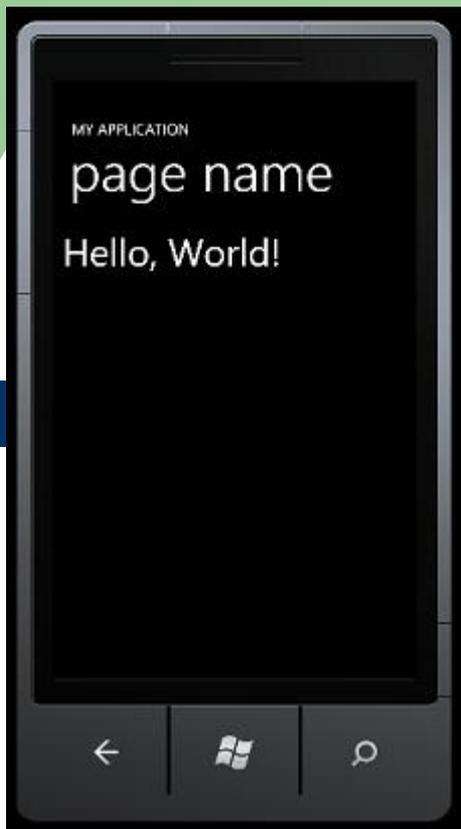
`<TextBlock ... Text="page name" ... />`

Running Your First Application

Now that you've created your first Silverlight application for Windows Phone, you need to run it. 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.

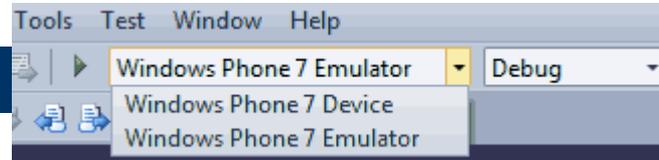
To start the application in debug mode, **press F5 or choose Debug->Start Debugging**



-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.

Enhancing the application: about Points and Pixels (1)

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

Points and Pixels (2)

The issue of font size becomes more complex when dealing with high-resolution screens found on devices such as Windows Phone 7. The 480 × 800 pixel display has a diagonal of 933 pixels. The phone I used for this book has a screen with about 3½” for a pixel density closer to

264 DPI. (Screen resolution is usually expressed as a multiple of 24.)

Roughly that’s 2,5 times the resolution of conventional video displays.

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

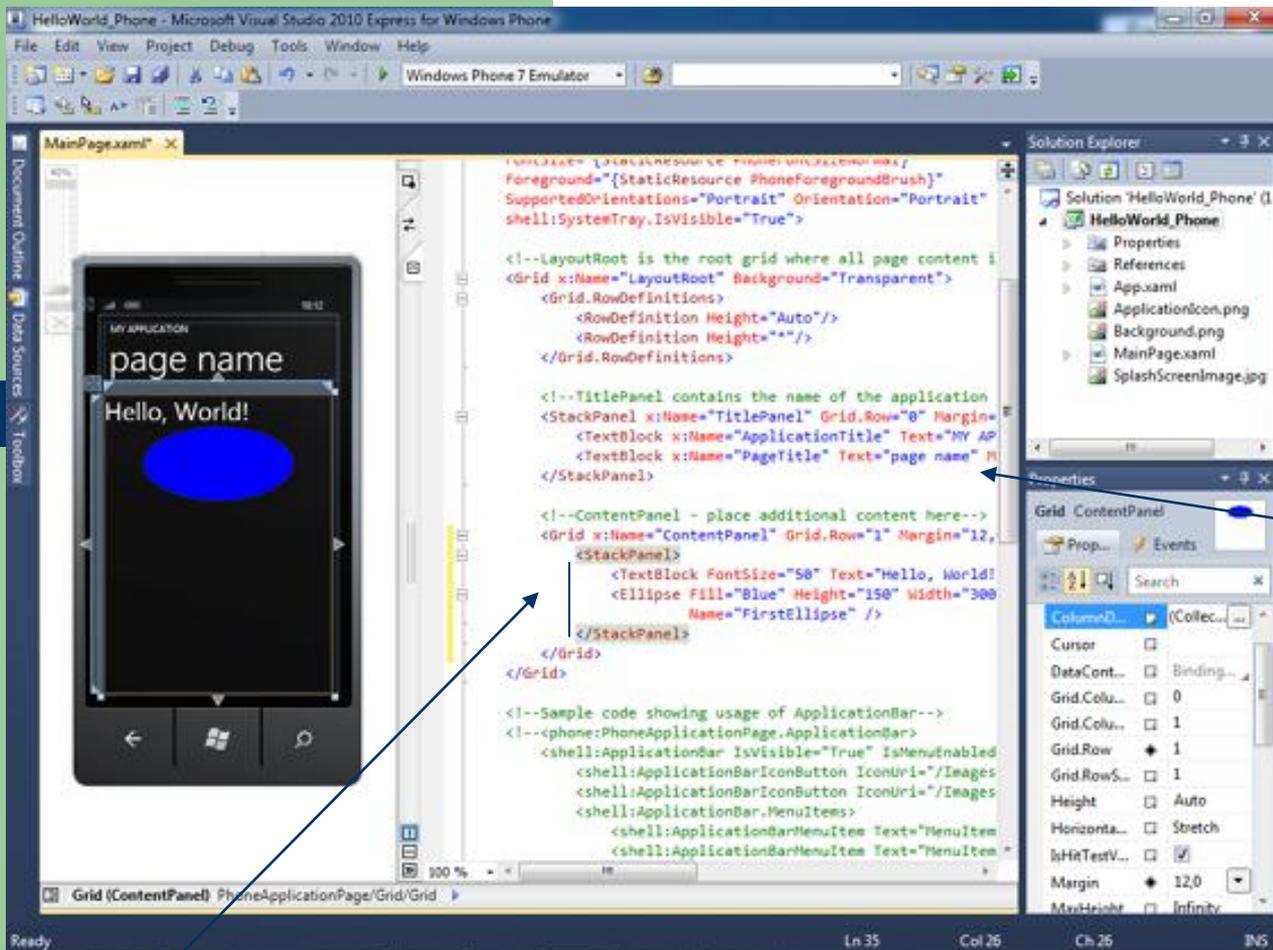
Canvas panels allow for more exact positioning of elements.

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. Controls are one way users can interact with Silverlight applications. 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"

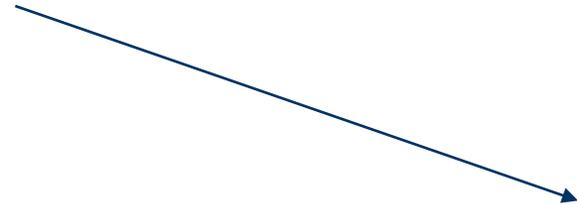
    Width="300"
    Content="Tap" />
```

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. Within the curly braces, add the following code to the event handler.



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

```
    if (FirstButton.Content as string == "TAP")
```

```
    {
```

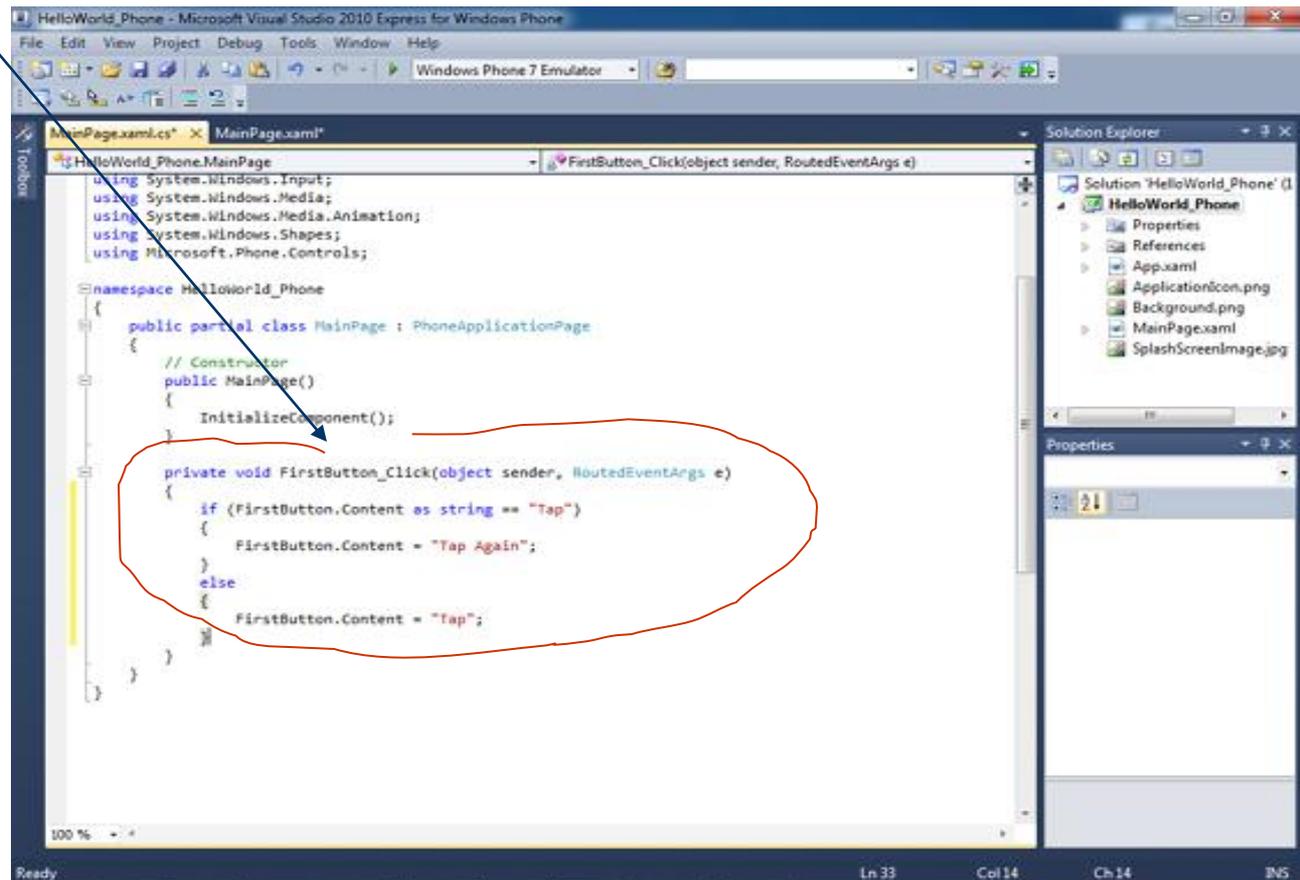
```
        FirstButton.Content = "TAP AGAIN";
```

```
    }
```

```
    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

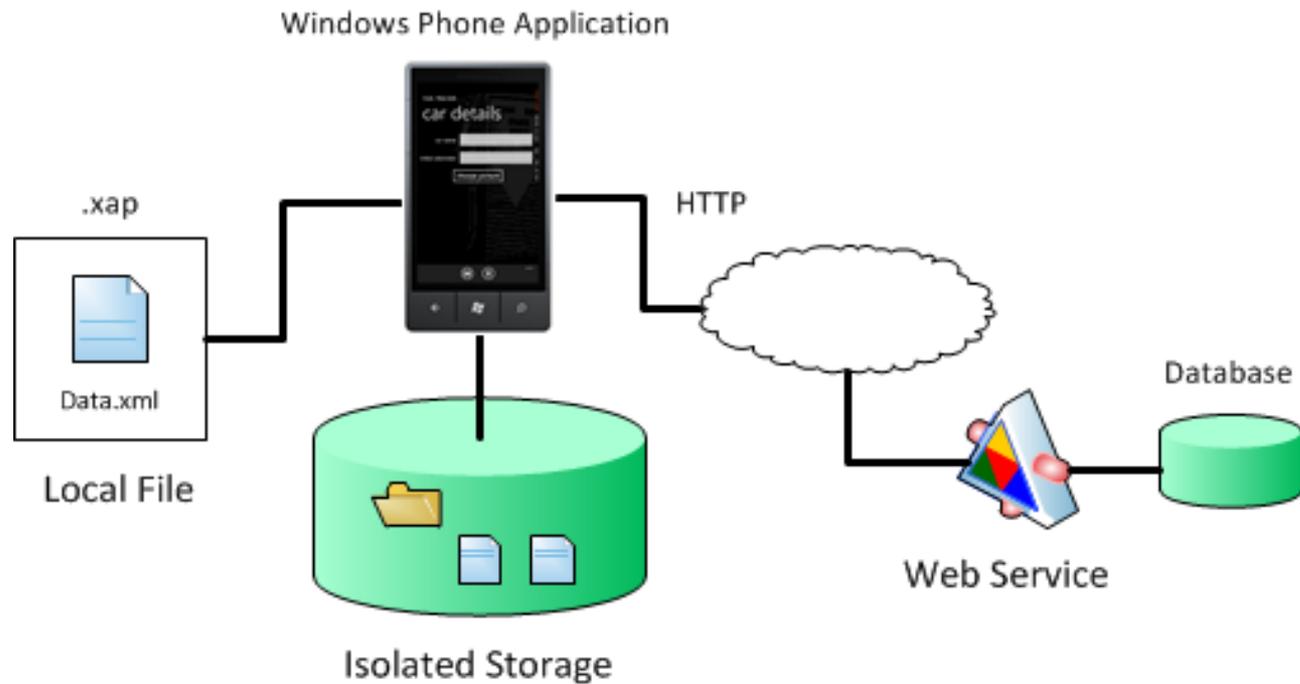
When you have finished your application, you will likely want to distribute it to the public as a free download or sell it. You do this by submitting your application to the

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 into your Windows Phone Applications

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't 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
<u>web service</u>	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.
<u>REST</u>	(Representational State Transfer Protocol) A protocol for exposing resources on the web for access by clients.
<u>POX</u>	(Plain Old XML) A term used to describe basic XML.
<u>JSON</u>	(JavaScript Object Notation) A lightweight format for exchanging data. It's designed to be human-readable, but also easily parsed by a computer.
<u>OData</u>	(Open Data Protocol) A web protocol for querying and updating data.
<u>SOAP</u>	(Simple Object Access Protocol) A lightweight protocol intended for exchanging structured information in a decentralized, distributed environment.

Web Service Technologies

There are several networking and Web service technologies that you can use to get data into your Silverlight for Windows Phone application. These technologies include the following:

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.

Keep in mind, however, that JSON serialization is limited on the Windows Phone. You can use these classes to build the request, ensuring the requests match the format expected by the service.

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, as described in the next section.

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. This means 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 applications, particularly since storage on the device is limited. The Windows Azure platform provides several data storage options for Windows Phone applications. 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 item. The target is the **Text** property of a text box control. The source is a simple music **Recording class**.

XAML

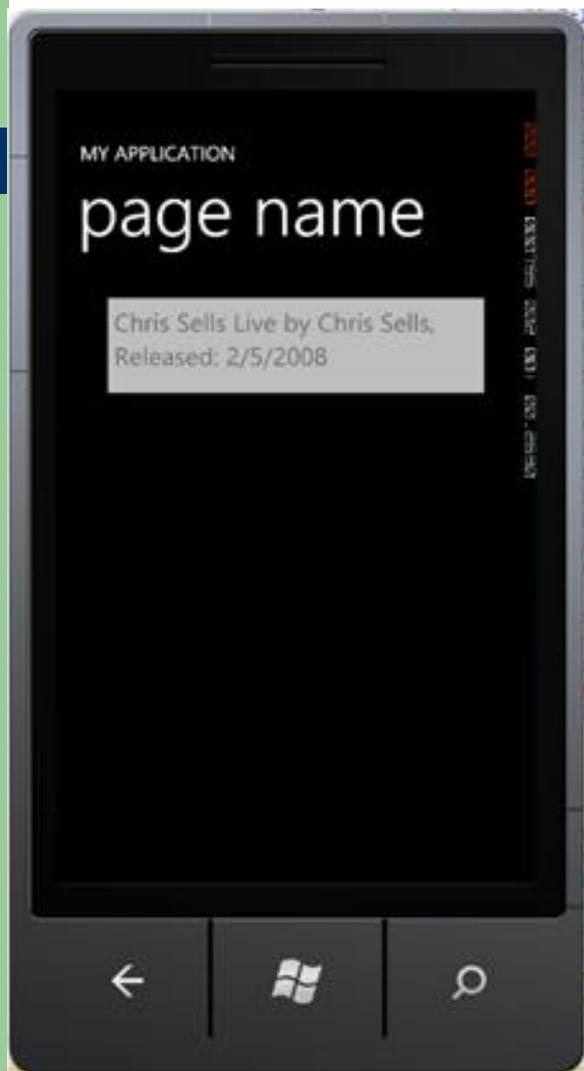
```
<GRID X:NAME="CONTENTPANEL" GRID.ROW="1" MARGIN="12,0,12,0">  
  <TextBox VerticalAlignment="Top" IsReadOnly="True" Margin="5"  
    TextWrapping="Wrap" Height="120" Width="400"  
    Text="{Binding}" x:Name="textBox1" />  
</Grid>
```

// CONSTRUCTOR

```
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

(3)

DB connection and code example for

Windows Phone SDK 7.1 – 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

Silverlight/XNA integration

Here’s a summary of the tasks required to build this application, from start to finish:

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.
- 6*. Incorporate an XNA game into the application.

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 that you build purely to create the database.

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 `DataContext`, which is defined in the custom Phone version of the `System.Data.Linq` assembly. 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, NVARCHAR 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—and using a standard Model-View-ViewModel (MVVM) approach—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, we create multiple instances of the Recipe class, corresponding to rows in the table; add all the rows in the collection to the DataContext; 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();
}
```

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 <code>WMAppManifest.xml</code> 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

Consume 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, with a couple of minor changes. In Mangolicious, we don't need to write to the database, so we can use it directly from the application install folder.

Thus the connection string is slightly different from the one in the helper application. Also, Mangolicious defines a 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 two 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 as a starting point. We need the `DataContext` field and the set of `List<T>` collection properties for the data tables. On top of that, 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 Panoramaltems. 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.

We don't define an explicit style for the databound TextBlocks, but that they all nonetheless use `TextWrapping=Wrap`. This is done by declaring a TextBlock style in the App.xaml.cs:

```
<Style TargetType="TextBlock" BasedOn="{StaticResource PhoneTextNormalStyle}">
<Setter Property="TextWrapping" Value="Wrap"/>
</Style>
```

The effect of this is that any TextBlock in the solution that doesn't explicitly define its own style

will implicitly use this one instead. Implicit styling is another new feature introduced in the Windows Phone SDK 7.1 as part of Silverlight 4.

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

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:



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 (¾ inch cubes) mango. **Topping:** 1 pint heavy cream, 1 tablespoon confectioner's sugar, 1 teaspoon lime zest, 2 limes cut into wheels.

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 first: define a tag into the current page, marking it as tiled, and second – to define a code – handler to PinUnpin_Click event defining the way and substitutions for un-pinning

Final marks about Mango

We've looked at how to develop applications against several of the new features in Windows Phone SDK 7.1:

local databases,
LINQ to SQL,
secondary tiles and deep linking,
Silverlight/XNA integration – not commented now.

The 7.1 release offers many more new features and enhancements to existing features. For further details, see the following links:

What's New in the Windows Phone SDK:

bit.ly/c2RmNr

Tiles:

bit.ly/oQlu15

Combining Silverlight and XNA:

bit.ly/p4RncQ

Local Database Overview for Windows Phone:

bit.ly/l23UQM

The final version of the Mangolicious application is available on the Windows Phone Marketplace at (note: Zune software is needed for access).

bit.ly/nuJcTA

Note that the sample uses the Silverlight for Windows Phone Toolkit – a free download, available at

bit.ly/qiHnTT

Working with sensors on Windows Phone 7

This include: camera, cell phone radio itself, Wi-Fi, GPS, a touchscreen, motion detectors and more.

Although the camera has always been in Windows Phone, the only API available in the original release was `CameraCaptureTask`. This class essentially spawns a child process that lets the user take a photo, and then returns that picture to the application.

The application can't control any part of this process, nor can it obtain the live video feed coming through the lens.

That deficiency has now been corrected with two sets of programming interfaces.

One set of APIs concerns the `Camera` and `PhotoCamera` classes. These classes allow an application to assemble an entire photo-taking UI, including flash options; live preview video feed; shutter key presses and half-presses; and focus detection.

The APIs we'll discuss here were inherited from the Silverlight 4 webcam interface. They let an application obtain live video and audio feeds from the phone's camera and microphone.

These feeds can be presented to the user, saved to a file or—and here it gets more interesting—manipulated or interpreted in some way.

Devices and Sources

The webcam interface consists of about a dozen classes defined in the `System.Windows.Media` namespace. You'll always begin with the static `CaptureDeviceConfiguration` class. You can simply call the `GetDefaultVideoCaptureDevice()` and `GetDefaultAudioCaptureDevice()` methods.

`CaptureDeviceConfiguration` methods return instances of `VideoCaptureDevice` and `AudioCaptureDevice`. For video, they return the device's name + formats (pixel dimensions of each frame of video, the color format and frames per second). For audio, the format specifies the number of channels, the bits per sample and the wave format, which is always Pulse Code Modulation (PCM).

If the `CaptureDeviceConfiguration.AllowedDeviceAccess` property is true, however, then the user has already given permission for this access.

The application must also create a `CaptureSource` object, which combines a video device and an audio device into a single stream of live video and audio. `CaptureSource` has two properties, named `VideoCaptureDevice` and `AudioCaptureDevice`, that you set to instances of `VideoCaptureDevice` and `AudioCaptureDevice` obtained from `CaptureDeviceConfiguration`. After creating a `CaptureSource` object, you can call the object's `Start()` and `Stop()` methods.

In addition, you can use the `CaptureImageAsync()` method of `CaptureSource` to obtain individual video frames in the form of `WriteableBitmap` objects.

The VideoBrush

The easiest **CaptureSource** option is the **VideoBrush**. As with any brush, you can use it to color element backgrounds or foregrounds.

Here is a program called **StraightVideo** that uses **VideoCaptureDevice**, **CaptureSource** and **VideoBrush** to display the live video feed coming through the default camera lens.

Following is a chunk of the **MainPage.xaml** file. Notice the use of landscape mode (which you'll want for video feeds), the definition of the **VideoBrush** on the **Background** property of the content **Grid** and the **Button** for obtaining user permission to access the camera:

```
<phone:PhoneApplicationPage
  x:Class="StraightVideo.MainPage"
  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"
  ...
  SupportedOrientations="Landscape" Orientation="LandscapeLeft" shell:SystemTray.IsVisible="True">
  <Grid x:Name="LayoutRoot" Background="Transparent">
    <Grid.RowDefinitions>      <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <StackPanel x:Name="TitlePanel" Grid.Row="0" Margin="12,17,0,28">
    <TextBlock x:Name="ApplicationTitle" Text="STRAIGHT VIDEO"      Style="{StaticResource PhoneTextNormalStyle}"/>
  </StackPanel>
  <Grid x:Name="ContentPanel" Grid.Row="1" Margin="12,0,12,0">
    <Grid.Background>
      <VideoBrush x:Name="videoBrush" />
    </Grid.Background>
    <Button Name="startButton" Content="start" HorizontalAlignment="Center" VerticalAlignment="Center"
      Click="OnStartButtonClick" />
  </Grid>
</Grid>
</phone:PhoneApplicationPage>
```

The following code shows much of [the codebehind file](#). The `CaptureSource` object is created in the page's constructor, but it's started and stopped in the navigation overrides. We call `SetSource` on the `VideoBrush` in `OnNavigatedTo`; otherwise the image was lost after a previous `Stop` call:

```
public partial class MainPage : PhoneApplicationPage
{ CaptureSource captureSource;

public MainPage()
{ InitializeComponent();
  captureSource = new CaptureSource
  VideoCaptureDevice = CaptureDeviceConfiguration.DefaultVideoCaptureDevice() }
protected override void OnNavigatedTo(NavigationEventArgs args)
{ if (captureSource != null && CaptureDeviceConfiguration.AllowedDeviceAccess)
  { videoBrush.SetSource(captureSource);
    captureSource.Start();
    startButton.Visibility = Visibility.Collapsed; }
  base.OnNavigatedTo(args);
}

protected override void OnNavigatedFrom(NavigationEventArgs args)
{ if (captureSource != null && captureSource.State == CaptureState.Started)
  { captureSource.Stop();
    startButton.Visibility = Visibility.Visible; }
  base.OnNavigatedFrom(args);
}

void OnStartButtonClick(object sender, RoutedEventArgs args)
{ if (captureSource != null && (CaptureDeviceConfiguration.AllowedDeviceAccess ||
  CaptureDeviceConfiguration.RequestDeviceAccess()))
  { videoBrush.SetSource(captureSource); captureSource.Start();
    startButton.Visibility = Visibility.Collapsed; }
}
}
```

You can run the previous program on the Windows Phone Emulator, but it's much more interesting on a real device. You'll notice that the rendering of the video feed is very responsive. Evidently the video feed is going directly to the video hardware. You'll also notice that because the video is rendered via a brush, the brush is stretched to the dimensions of the content Grid and the image is distorted.



Source and Sink

The alternative to using a VideoBrush is connecting a CaptureSource object to an AudioSink, VideoSink or FileSink object. The use of the word “sink” in these class names is in the sense of “receptacle” .

The FileSink class is the preferred method for saving video or audio streams to your application's isolated storage without any intervention on your part. If you need access to the actual video or audio bits in real time, you'll use VideoSink and AudioSink. These two classes are abstract. You derive a class from one or both of these abstract classes and override the OnCaptureStarted, OnCaptureStopped, OnFormatChange and OnSample methods.

The class that you derive from VideoSink or AudioSink will always get a call to OnFormatChange before the first call to OnSample. The information supplied with OnFormatChange indicates how the sample data is to be interpreted. For both VideoSink and AudioSink, the OnSample call provides timing information and an array of bytes. For AudioSink, these bytes represent PCM data. For VideoSink, these bytes are rows and columns of pixels for each frame of video.

You must use an instance of `WritableBitmap` to display the resultant video feed. That `WritableBitmap` can't be created until the `OnFormatChange` method is called in the `VideoSink` derivative, because that call indicates the size of the video frame. (It's usually 640x480 pixels on a phone but might be something else.)



A bit of exercise:

If you just need a real-time video feed with occasional frame captures, you can use the `CaptureImageAsync` method of `CaptureSource`.

The `VideoSink` derivative gets a video feed where each frame is probably 640x480 pixels in size or perhaps something else. You want to reference an equilateral triangle of image data from that frame as shown in Figure (a triangle had been selected because to better capture faces):



The image in that triangle can then be duplicated on a `WritableBitmap` multiple times with some rotation and flipping so the images are tiled and grouped into hexagons without any discontinuities:



When rendered on the phone, the height of the target `WriteableBitmap` will be the same as the phone's smaller dimension, or 480 pixels.

Each equilateral triangle thus has a side of 120 pixels. This means that the height of the triangle is 120 times the square root of 0.75, or about 104 pixels. In the program, I use 104 for the math but 105 for sizing the bitmap to make the loops simpler. The entire resultant image is 630 pixels wide.

We can change a little our program: the `OnFormatChange` method is now responsible for computing the members of an array. That array has as many members as the number of pixels in the `WriteableBitmap` (105 multiplied by 480, or 50,400) and stores an index into the rectangular area of the video image. Using this array, the transfer of pixels from the video image to the `WriteableBitmap` in the `OnSample` method is pretty much as fast as conceivably possible. The figure shows the result. You can even watch TV through it.



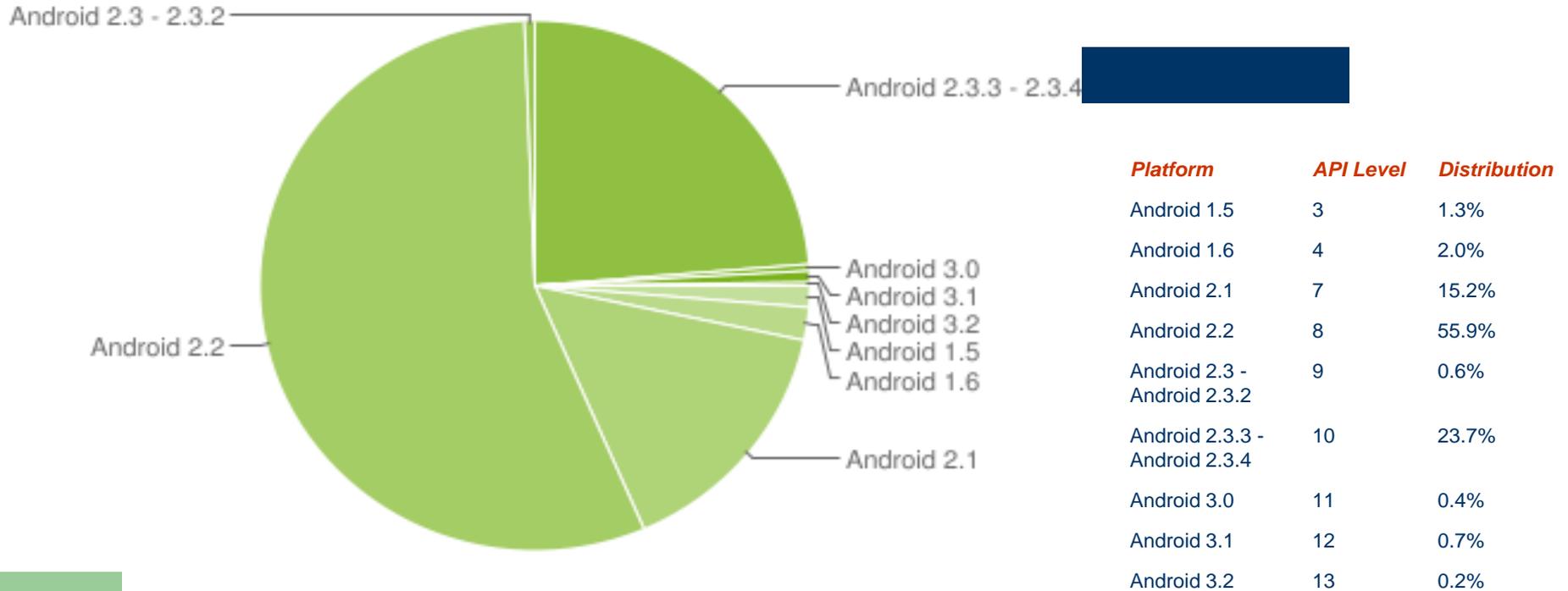
Android OS

What is Android?

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.

Current Distribution

The following pie chart and table is based on the number of Android devices that have accessed Android Market within a 14-day period ending on the data collection date noted below.



Data collected during a 14-day period ending on August 1, 2011

The following diagram shows the major components of the Android operating system.



Applications

Android will ship with a set of core applications including an email client, SMS program, calendar, maps, browser, contacts, and others. All applications are written using the Java programming language.

Application Framework

By providing an open development platform, Android offers developers the ability to build applications. Developers are free to take advantage of the device hardware, access location information, run background services, set alarms, add notifications to the status bar, and more. Developers have full access to the same framework APIs used by the core applications.

Underlying all applications is a set of services and systems, including:

- A rich and extensible set of **Views** that can be used to build an application, including lists, grids, text boxes, buttons, and even an embeddable web browser
- **Content Providers** that enable applications to access data from other applications (such as Contacts), or to share their own data
- A **Resource Manager**, providing access to non-code resources such as localized strings, graphics, and layout files
- A **Notification Manager** that enables all applications to display custom alerts in the status bar
- An **Activity Manager** that manages the lifecycle of applications and provides a common navigation backstack

Libraries

Android includes a set of C/C++ libraries used by various components of the Android system.

Android Runtime

Android includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language.

Every Android application runs in its own process, with its own instance of the Dalvik virtual machine. The Dalvik VM executes files in the Dalvik Executable (.dex) format which is optimized for minimal memory. The VM runs classes compiled by a Java language compiler that have been transformed into the .dex format by the included "dx" tool.

The Dalvik VM relies on the Linux kernel for underlying functionality such as threading and low-level memory management.

Linux Kernel

Android relies on Linux version 2.6 for core system services such as security, memory management, process management, network stack, and driver model. The kernel also acts as an abstraction layer between the hardware and the rest of the software stack.

Application Fundamentals

- Android applications are written in the **Java programming language**. The Android SDK tools compile the code—along with any data and resource files—into an *Android package*.
- The Android operating system is a **multi-user Linux system** in which each application is a different user.
- Each process has its **own virtual machine (VM)**, so an application's code runs in isolation from other applications.
- By default, every application runs in its **own Linux process**.

Application Components

Application components are the essential building blocks of an Android application. Each component is a different point through which the system can enter your application.

There are four different types of application components. Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is created and destroyed.

Here are the four types of application components:

Activities

An *activity* represents a single screen with a user interface. For example, an email application might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email application, each one is independent of the others. As such, a different application can start any one of these activities (if the email application allows it). For example, a camera application can start the activity in the email application that composes new mail, in order for the user to share a picture. An activity is implemented as a subclass of Activity.

Services

A *service* is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different application, or it might fetch data over the network without blocking user interaction with an activity. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. A service is implemented as a subclass of Service.

Content providers

A *content provider* manages a shared set of application data. You can store the data in the file system, an SQLite database, on the web, or any other persistent storage location your application can access. Through the content provider, other applications can query or even modify the data. For example, the Android system provides a content provider that manages the user's contact information.

Content providers are also useful for reading and writing data that is private to your application and not shared. For example, the [Note Pad](#) sample application uses a content provider to save notes.

A content provider is implemented as a subclass of [ContentProvider](#) and must implement a standard set of APIs that enable other applications to perform transactions.

Broadcast receivers

A *broadcast receiver* is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system—for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Applications can also initiate broadcasts—for example, to let other applications know that some data has been downloaded to the device and is available for them to use.

Although broadcast receivers don't display a user interface, they may [create a status bar notification](#) to alert the user when a broadcast event occurs. The broadcast receiver is just a "gateway" to other components and is intended to do a very minimal amount of work. For instance, it might initiate a service to perform some work based on the event.

A broadcast receiver is implemented as a subclass of [BroadcastReceiver](#) and each broadcast is delivered as an [Intent](#) object



A unique aspect of the Android system design is that any application can start another application's component. For example, if you want the user to capture a photo with the device camera, there's probably another application that does that and your application can use it, instead of developing an activity to capture a photo yourself. You don't need to incorporate or even link to the code from the camera application. Instead, you can simply start the activity in the camera application that captures a photo. When complete, the photo is even returned to your application so you can use it. To the user, it seems as if the camera is actually a part of your application.

When the system starts a component, it starts the process for that application (if it's not already running) and instantiates the classes needed for the component. For example, if your application starts the activity in the camera application that captures a photo, that activity runs in the process that belongs to the camera application, not in your application's process. Therefore, unlike applications on most other systems, Android applications don't have a single entry point (there's no `main()` function, for example).

Because the system runs each application in a separate process with file permissions that restrict access to other applications, your application cannot directly activate a component from another application. The Android system, however, can. So, to activate a component in another application, you must deliver a message to the system that specifies your *intent* to start a particular component. The system then activates the component for you.

Activating Components

Three of the four component types—activities, services, and broadcast receivers—are activated by an asynchronous message called an *intent*. Intents bind individual components to each other at runtime (you can think of them as the messengers that request an action from other components), whether the component belongs to your application or another.

An intent is created with an Intent object, which defines a message to activate either a specific component or a specific *type* of component—an intent can be either explicit or implicit, respectively.

For activities and services, an intent defines the action to perform (for example, to "view" or "send" something) and may specify the URI of the data to act on (among other things that the component being started might need to know). For example, an intent might convey a request for an activity to show an image or to open a web page. In some cases, you can start an activity to receive a result, in which case, the activity also returns the result in an Intent (for example, you can issue an intent to let the user pick a personal contact and have it returned to you—the return intent includes a URI pointing to the chosen contact).

For broadcast receivers, the intent simply defines the announcement being broadcast (for example, a broadcast to indicate the device battery is low includes only a known action string that indicates "battery is low").

The other component type, content provider, is not activated by intents. Rather, it is activated when targeted by a request from a ContentResolver. The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider doesn't need to and instead calls methods on the ContentResolver object. This leaves a layer of abstraction between the content provider and the component requesting information (for security).

There are separate methods for activating each type of component:

- *You can start an activity (or give it something new to do) by passing an Intent to startActivity() or startActivityForResult() (when you want the activity to return a result).*
 - *You can start a service (or give new instructions to an ongoing service) by passing an Intent to startService(). Or you can bind to the service by passing an Intent to bindService().*
 - *You can initiate a broadcast by passing an Intent to methods like sendBroadcast(), sendOrderedBroadcast(), or sendStickyBroadcast().*
 - *You can perform a query to a content provider by calling query() on a ContentResolver.*
-

The Manifest File

Before the Android system can start an application component, the system must know that the component exists by reading the application's `AndroidManifest.xml` file (the "manifest" file). Your application must declare all its components in this file, which must be at the root of the application project directory.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
  <application android:icon="@drawable/app_icon.png" ... >
    <activity android:name="com.example.project.ExampleActivity"
      android:label="@string/example_label" ... >
      </activity>
    ...
  </application>
</manifest
```

Declaring application requirements

There are a variety of devices powered by Android and not all of them provide the same features and capabilities. In order to prevent your application from being installed on devices that lack features needed by your application, it's important that you clearly define a profile for the types of devices your application supports by declaring device and software requirements in your manifest file.

Most of these declarations are informational only and the system does not read them, but external services such as Android Market do read them in order to provide filtering for users when they search for applications from their device.

For example, if your application requires a camera and uses APIs introduced in Android 2.1, you should declare these as requirements in your manifest file. That way, devices that do *not* have a camera and have an Android version *lower* than 2.1 cannot install your application from Android Market.

Here are some of the important device characteristics :

Screen size and density

Input configurations (hardware keyboard, a trackball, or a five-way navigation pad)

Device features (camera, a light sensor, bluetooth, a certain version of OpenGL, Touchscreen)

Platform Version

Developing applications process

Developing applications for Android devices is facilitated by a group of tools that are provided with the SDK. You can access these tools through an **Eclipse plugin** called ADT (Android Development Tools) or from the command line. Developing with Eclipse is the preferred method because it can directly invoke the tools that you need while developing applications.

The basic steps for developing applications with or without Eclipse are the same:

- 1. **Set up Android Virtual Devices or hardware devices.**

You need to create Android Virtual Devices (AVD) or connect hardware devices on which you will install your applications.

An Android Virtual Device (AVD) is an emulator configuration that lets you model an actual device by defining hardware and software options to be emulated by the Android Emulator. The easiest way to create an AVD is to use the graphical AVD Manager, which you launch from Eclipse

- 2. **Create an Android project.**

An Android project contains all source code and resource files for your application. It is built into an .apk package that you can install on Android devices.

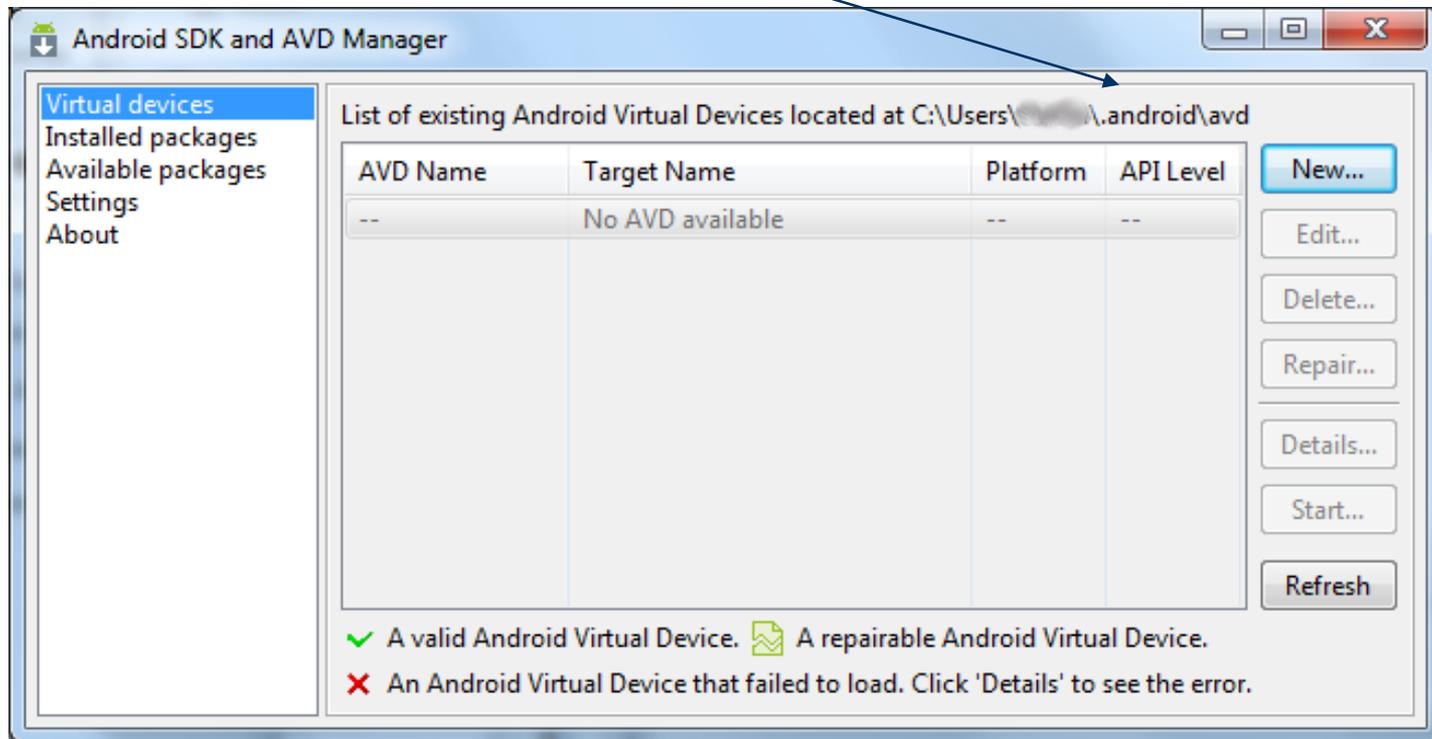
- 3. **Build and run your application.**

- 4. **Debug your application with the SDK debugging and logging tools.**

tools are provided with the Android SDK. Eclipse already comes packaged with a compatible debugger.

- 5. **Test your application with the Testing and Instrumentation framework.**

An **AVD** is a device configuration for the Android emulator that allows you to model different configurations of Android-powered devices. When you start the AVD Manager in Eclipse or run the **android** tool on the command line, you will see the AVD Manager as shown:



Publishing on Android Market

When you have a compiled .apk file that is signed with your private release key - Your application is now ready to be published publicly so users can install it.

You can publish your application and allow users to install it any way you choose, including from your own web server. This document provides information about publishing your Android application with **Android Market**.

About Android Market

Android Market is a service that makes it easy for users to find and download Android applications to their Android-powered devices, either from the Android Market application on their device or from the Android Market web site (market.android.com). As a developer, you can use Android Market to distribute your applications to users on all types of Android-powered devices, all around the world.

To publish your application on Android Market, you first need to register with the service using a Google account and agree to the terms of service. **Once you are registered, you can upload your application to the service whenever you want, update it as many times as you want, and then publish it when you are ready.** Once published, users can see your application, download it, and rate it.

To register as an Android Market developer and get started with publishing, visit the Android Market publisher site:

<http://market.android.com/publish>

Android SDK

First - set up your SDK.

<i>Platform</i>	<i>Package</i>	<i>Size</i>
Windows	<u>android-sdk_r12-windows.zip</u>	36486190 bytes
	<u>installer_r12-windows.exe</u> (Recommended)	36531492 bytes
Mac OS X (intel)	<u>android-sdk_r12-mac_x86.zip</u>	30231118 bytes
Linux (i386)	<u>android-sdk_r12-linux_x86.tgz</u>	30034243 bytes

Eclipse IDE

Android Development Tools (ADT) is a plugin for the Eclipse IDE that is designed to give you a powerful, integrated environment in which to build Android applications.

Android APIs Include more than 70 packages with more than 400 classes

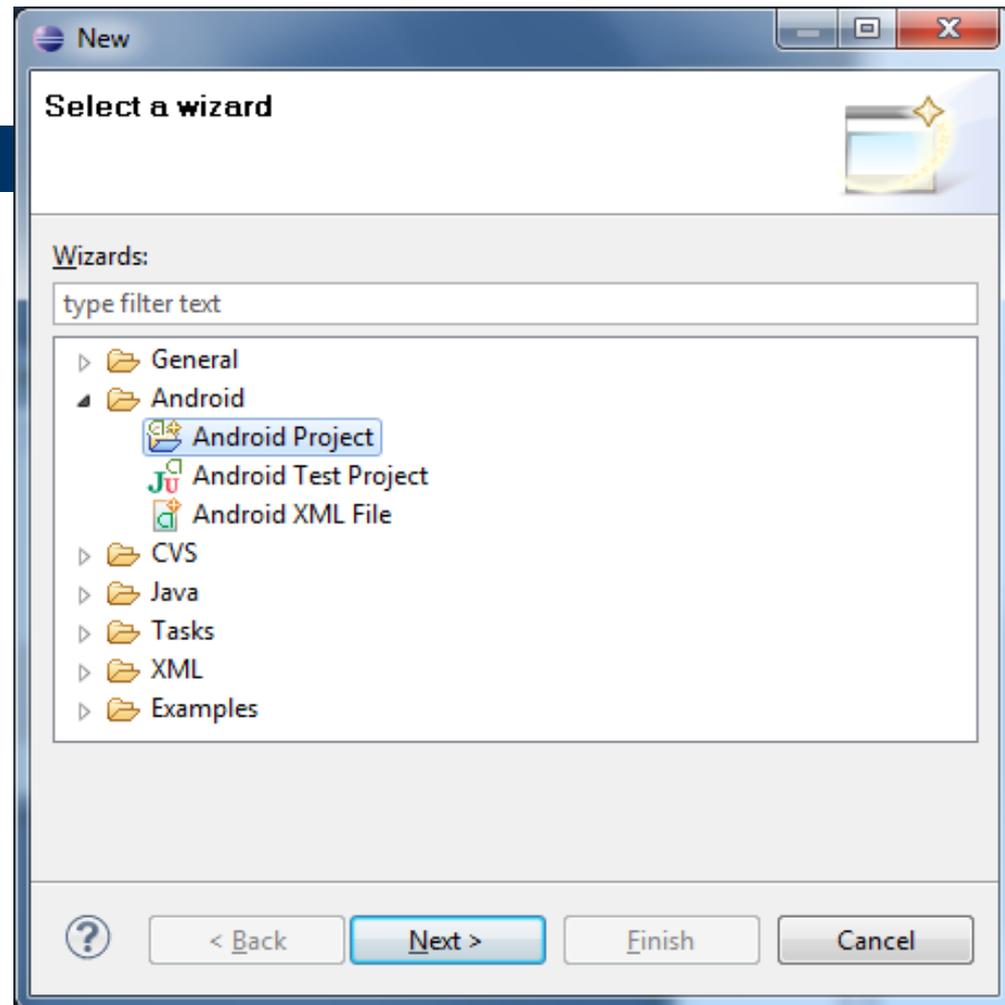
First application : Hello world

As a developer, you know that the first impression of a development framework is how easy it is to write "Hello, World." Well, on Android, it's pretty easy. It's particularly easy if you're using Eclipse as your IDE, because that plugin handles your project creation and management to greatly speed up your development cycles

1. you will **run your application in the Android Emulator**. Before you can launch the emulator, you must **create an Android Virtual Device (AVD)**. An AVD defines the system image and device settings used by the emulator

2. Create a New Android Project

After you've created an AVD you can move to the next step and start a new Android project in Eclipse



Your Android project is now ready. It should be visible in the Package Explorer on the left. Open the **HelloAndroid.java** file, located inside *HelloAndroid* → *src* → *com.example.helloandroid*

It should look like this:

```
package com.example.helloandroid;

import android.app.Activity;
import android.os.Bundle;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Notice that the **class is based on the Activity class**. An Activity is a single application entity that is used to perform actions. An application may have many separate activities, but the user interacts with them one at a time. The **onCreate()** method is called by the Android system when your Activity starts — it is where you should perform all initialization and UI setup.



An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

An application usually consists of multiple activities that are loosely bound to each other. Typically, one activity in an application is specified as the "main" activity, which is presented to the user when launching the application for the first time. Each activity can then start another activity in order to perform different actions. Each time a new activity starts, the previous activity is stopped, but the system preserves the activity in a stack (the "back stack"). When a new activity starts, it is pushed onto the back stack and takes user focus. The back stack abides to the basic "last in, first out" queue mechanism, so, when the user is done with the current activity and presses the BACK key, it is popped from the stack (and destroyed) and the previous activity resumes.

When an activity is stopped because a new activity starts, it is notified of this change in state through the activity's lifecycle callback methods. There are several callback methods that an activity might receive, due to a change in its state—whether the system is creating it, stopping it, resuming it, or destroying it—and each callback provides you the opportunity to perform specific work that's appropriate to that state change. For instance, when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted.

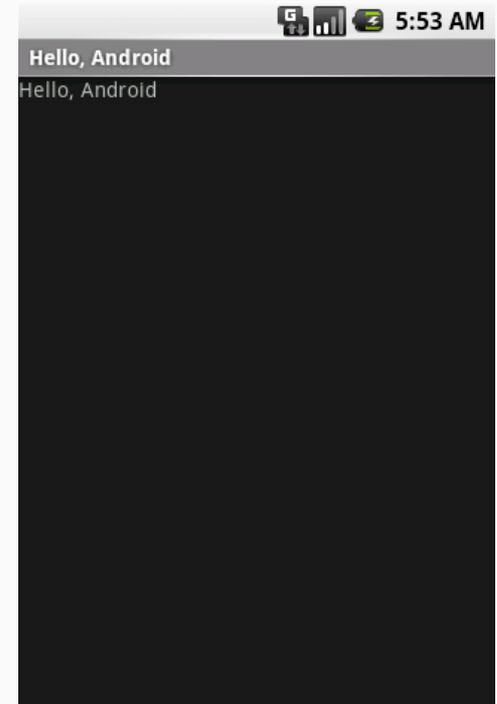
Construct the UI

Take a look at the revised code below and then make the same changes to your HelloAndroid class. The bold items are lines that have been added.

```
package com.example.helloandroid;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView tv = new TextView(this);
        tv.setText("Hello, Android");
        setContentView(tv);
    }
}
```



An Android user interface is composed of hierarchies of objects called **Views**. A View is a drawable object used as an element in your UI layout, such as a button, image, or (in this case) a text label. Each of these objects is a subclass of the View class and the subclass that handles text is TextView.

In this change, you create a TextView with the class constructor, which accepts an Android Context instance as its parameter. A **Context** is a handle to the system; it provides services like resolving resources, obtaining access to databases and preferences, and so on.

The Activity class inherits from Context, and because your HelloAndroid class is a subclass of Activity, it is also a Context. So, you can pass this as your Context reference to the TextView. Next, you define the text content with setText().

Finally, you pass the TextView to setContentView() in order **to display it as the content** for the Activity UI. If your Activity doesn't call this method, then no UI is present and the system will display a blank screen.

There it is — "Hello, World" in Android! The next step, of course, is to see it running.

Upgrade the UI to an XML Layout

The "Hello, World" example you just completed uses what is called a "programmatic" UI layout. This means that you constructed and built your application's UI directly in source code.

Android provides an alternate UI construction model: **XML-based layout files**. The easiest way to explain this concept is to show an example. Here's an XML layout file that is identical in behavior to the programmatically-constructed example:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/textview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/hello"/>
```

The general structure of an Android XML layout file is simple: it's a tree of XML elements, wherein each node is the name of a View class .

In the above XML example, there's just one View element: the **TextView**, which has a number of XML attributes.

These XML layout files belong in the **res/layout/ directory** of your project

The Eclipse plugin automatically creates one of these layout files for you: **main.xml**

you should almost always define your layout in an XML file instead of in your code. The following procedures will instruct you how to change your existing application to use an XML layout.

1. In the Eclipse Package Explorer, expand the `/res/layout/` folder and open `main.xml`. Replace the contents with the following XML:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/textview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/hello"/>
```

Save the file.

2. Inside the `res/values/` folder, open `strings.xml`. This is where you should save all default text strings for your user interface. If you're using Eclipse, then ADT will have started you with two strings, `hello` and `app_name`. Revise `hello` to something else. The entire file should now look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello, Android! I am a string resource!</string>
    <string name="app_name">Hello, Android</string>
</resources>
```

3. Now open and modify your **HelloAndroid class** and use the XML layout. Edit the file to look like this:

```
package com.example.helloandroid;

public class HelloAndroid extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```



When you make this change, type it by hand to try the code-completion feature. As you begin typing "R.layout.main" the plugin will offer you suggestions. You'll find that it helps in a lot of situations. Instead of passing setContentView() a View object, you give it a reference to the layout resource. The resource is identified as R.layout.main, which is actually a compiled object representation of the layout defined in /res/layout/main.xml. The Eclipse plugin automatically creates this reference for you inside the project's R.java class.

R class

In Eclipse, open the file named `R.java` (in the `gen/ [Generated Java Files]` folder). It should look like this:
`package com.example.helloandroid;`

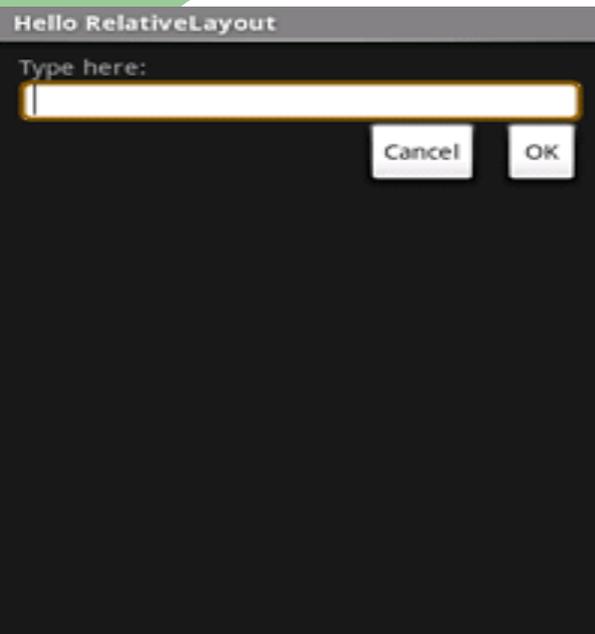
```
public final class R {  
    public static final class attr {  
    }  
    public static final class drawable {  
        public static final int icon=0x7f020000;  
    }  
    public static final class id {  
        public static final int textview=0x7f050000;  
    }  
    public static final class layout {  
        public static final int main=0x7f030000;  
    }  
    public static final class string {  
        public static final int app_name=0x7f040001;  
        public static final int hello=0x7f040000;  
    }  
}
```

The `R.java` file is an index into all the resources defined in the file.

You use this class in your source code as a sort of short-hand way to refer to resources you've included in your project.

examples”:

(example 1)



Relative Layout

RelativeLayout is a ViewGroup that displays child View elements in relative positions

A **ViewGroup** is a special view that can contain other views (called children.) The view group is the base **class** for layouts and views containers. This class also defines the ViewGroup.LayoutParams class which serves as the base class for layouts parameters.

RelativeLayout

A Layout where the positions of the children can be described in relation to each other or to the parent.

1. Start a new project named *HelloRelativeLayout*.
2. Open the res/layout/main.xml file and insert the following

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/label"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Type here:"/>
    <EditText
        android:id="@+id/entry"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:background="@android:drawable/editbox_background"
        android:layout_below="@id/label"/>
    <Button
        android:id="@+id/ok"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/entry"
        android:layout_alignParentRight="true"
        android:layout_marginLeft="10dip"
        android:text="OK" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/ok"
        android:layout_alignTop="@id/ok"
        android:text="Cancel" />
</RelativeLayout>
```

Notice each of the **android:layout_*** attributes, such as **layout_below**, **layout_alignParentRight**, and **layout_toLeftOf**.

When using a RelativeLayout, you can use these attributes to describe how you want to position each View. Each one of these attributes define a different kind of relative position. Some attributes use the resource ID of a sibling View to define its own relative position. For example, the last Button is defined to lie to the left-of and aligned-with-the-top-of the View identified by the ID ok (which is the previous Button).

All of the available layout attributes are defined in RelativeLayout.LayoutParams.

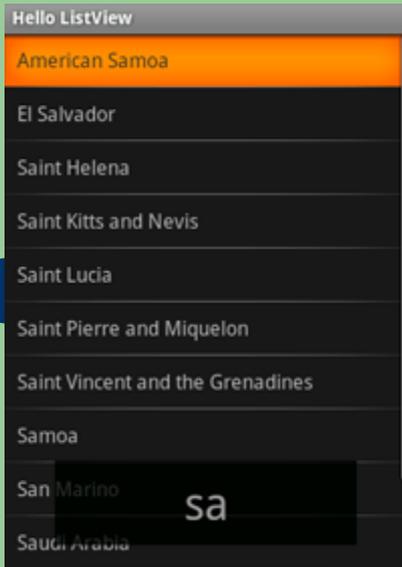
3. Make sure you load this layout in the onCreate() method:

```
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.main);  
}
```

The setContentView(int) method loads the layout file for the Activity, specified by the resource ID — R.layout.main refers to the res/layout/main.xml layout file.

4. Run the application

Example of views (2)



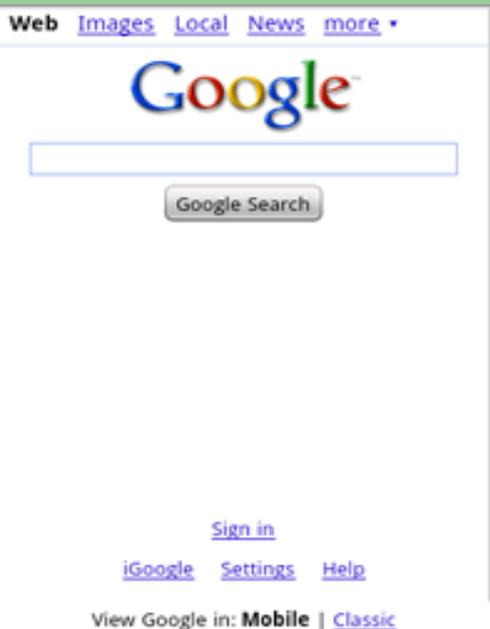
List View

List View is a ViewGroup that creates a list of scrollable items. The list items are automatically inserted to the list using a ListAdapter



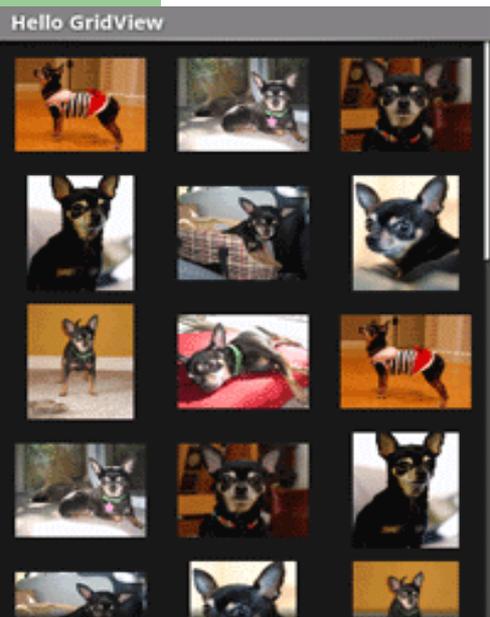
Google Map View

Using the Google Maps library, you can create your own map-viewing Activity



Web View

WebView allows you to create your own window for viewing web pages (or even develop a complete browser).



Grid View

GridView is a ViewGroup that displays items in a two-dimensional, scrollable grid. The grid items are automatically inserted to the layout using a ListAdapter