Quick answers to common problems

Xamarin Studio for Android Programming: A C# Cookbook

Over 50 hands-on recipes to help you get grips with Xamarin Studio and C# programming to develop market-ready Android applications

Mathieu Nayrolles

[PACKT] PUBLISHING

Xamarin Studio for Android Programming: A C# Cookbook

Over 50 hands-on recipes to help you get grips with Xamarin Studio and C# programming to develop market-ready Android applications

Mathieu Nayrolles

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Xamarin Studio for Android Programming: A C# Cookbook

First published: December 2015

Production reference: 1181215

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-140-6

www.packtpub.com

Credits

Author
Mathieu Nayrolles

Reviewers
Simen Hasselknippe
Douglas Mckechie
Sabry Moulana
Lucian Torje

Commissioning Editor
Sam Wood

Acquisition Editor
Vinay Argekar

Content Development Editor
Rohit Singh

Technical Editor
Abhishek Kotian

Copy Editor
Pranjali Chury

Project Coordinator
Mary Alex

Proofreader
Safis Editing

Indexer
Monica Ajmera Mehta

Graphics
Disha Haria

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Mathieu Nayrolles was born in France and lived in a small village in Côte d'Azur for almost 15 years. He started his studying computer science in France and continued in Montréal, Canada, where he now lives with his wife. He holds a master's degree in software engineering from eXia.Cesi and another in computer science from UQAM. He is currently a PhD student in electrical and computer engineering at Concordia University, Montréal, Canada, under the supervision of Dr. Wahab Hamou-Lhadj.

As well as his academic journey, Mathieu has also worked for worldwide companies such as Eurocopter and Saint-Gobain where he learned how important good technical resources are.

You can discover some of his work through his books *Instant Magento Performance*s and *Magento Performance Optimization: How to and Mastering Apache*. You can also check out his blog (`https://math.co.de/`) or latest realizations: `bumper-app.com`, `mindup.io` and `toolwatch.io`.

Follow him on Twitter at `@MathieuNls` for more information.

I would like to thank the reviewers of this book who did an amazing job and greatly improved its quality. I would also like to thank everyone at Packt Publishing who supported me throughout the writing of this book.

About the Reviewers

Douglas McKechie has had a passion for programming since an early age. He started out by writing QBASIC programs from books in the local library and then creating a replica of the game *Chip's Challenge* in Visual Basic 4.

Now, with over a decade of experience in the IT industry as a web application, database, and mobile developer, he has worked on some interesting and complex projects for both government and private sector organizations and is currently employed by one of New Zealand's top tech companies.

In his spare time, he likes to play his bass guitar, build things out of wood, play computer games, listen to vinyl records, and he dreams of building a large space ship out of Lego.

He is the creator of Winwheel.js, a feature-packed JavaScript library for creating spinning prize wheels on the HTML5 canvas. You can find more about this at `http://www.dougtesting.net`.

Sabry Moulana is a full stack developer with over 2 years of experience in the software development industry. He holds a degree in Software Engineering with First Class Honors from the University of Westminster. He currently works as a team lead at 99X Technology based in Sri Lanka. As a developer, his poisons include, but are not restricted to, JavaScript (Vanilla and all), Hybrid Application Development, and the .NET framework

---

I would like to thank Packt Publishing for giving me this opportunity. I'd also like to apologize if I have exceeded a few deadlines. Finally, I would like to thank the author for the hard work he has put into producing such an interesting and informative book.

Lucian Torje is a mobile app developer based in Cluj-Napoca, Romania. He started programming after watching Star Trek as a child and imagining that he would build androids someday. In his spare time, he loves watching sci-fi movies, reading (mostly sci-fi and philosophy books), and programming.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Xamarin is the leading company in cross-platform application development. This company was created by the same people who brought us Mono, MonoTouch, and Mono for Android, which were the very first cross-platform implementations of the Microsoft CLI (Common Language Structure) and CLS (Common Language Specification). Having a cross-platform CLI and CLS, which is often called .NET, allows us to develop a shared code base in C# to create a Windows Mobile, iOS, and Android application. As of last year, Xamarin has over half a million developers around the world. This success of Xamarin can be explained in many ways. First, they don't have much serious competition in the cross-platform mobile app development, they have a consequent developer base for Mono products and it works like a charm. Indeed, you do not need to have any knowledge about developing mobile applications to give it more than a try. Moreover, they provide a high-end IDE (integrated development environment), in which you can go from displaying a few words in a console to publishing a fully-fledged application in the applications store.

What this book covers

Chapter 1, *Getting Started*, talks about Xamarin Studio itself and the Android emulator needed to run our first Hello World application.

Chapter 2, *Mastering the Life and Death of Android Apps*, provides an in-depth and methodical approach to learning about activities' lifecycles and how to manage the different states of an Android Activity.

Chapter 3, *Building a GUI*, helps us familiarize ourselves with the Android GUI and prepares us to tackle every situation with no less than 19 different components.

Chapter 4, *Using Android Resources*, helps us make things look better by covering how to use splash screen, manage multiscreen, play songs or videos, and display our application icon.

Chapter 5, *Using On-Phone Data*, takes a leap forward in order to persist and access user data between different launches of our applications. To do so, you will learn about user preferences, file writing/reading, SQLite, and LinQ.

Chapter 6, *Populating Your GUI with Data*, completes the loop between the on-phone data of the previous chapter and the GUI of *Chapter 4*, *Using Android Resources*, by covering how to populate GUI elements with data.

Chapter 7, *Using Android Services*, shows how to create, bound to, and identify running Android services in order to pursue our application processes even when the users are not looking at them.

Chapter 8, *Mastering Intents – A Walkthrough*, covers how to switch back on forth between applications with Android Intents to access a contact number or an address on the map, for example.

Chapter 9, *Playing with Advanced Graphics*, will push forward the creation of advanced graphics, such as 2D graphics, in our application and animation. We will also take advantage of the Android API made available by Xamarin to use the camera.

Chapter 10, *Taking Advantage of the Android Platform*, brings the last piece to our GUI building skills by mastering Android fragments that represent the behavior or a portion of the user interface in an Activity.

Chapter 11, *Using Hardware Interactions*, guides us to interact with our phone's hardware, such as NFC, Bluetooth, Accelerometer, and GPS.

Chapter 12, *Debugging and Testing*, leverages the debugging and testing capacities of Xamarin Studio and the Android emulator.

Chapter 13, Monetizing and Publishing Your Applications, prepares us to build our final application and publish it to the Android Play Store.

Appendix, Mono – The Underlying Technology, exposes the key concepts about the technology that makes Xamarin possible.

What you need for this book

To follow the recipes in this book you with ease must have a good understanding of C# programming, as the basic C# code covered in this book will not be explained in detail. However, no knowledge in mobile development or Android is required.

Who this book is for

This book is for C# developers who want to be able to create cross-platform mobile applications and, more particularly, Android applications.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it, How it works, There's more, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it...

This section contains the steps required to follow the recipe.

How it works...

This section usually consists of a detailed explanation of what happened in the previous section.

There's more...

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
ISharedPreferences userPreferences = GetSharedPreferences
("OnPhoneData", FileCreationMode.Private);
ISharedPreferencesEditor userPreferencesEditor =
userPreferences.Edit();
count = userPreferences.GetInt ("some_counter", count);
userPreferencesEditor.PutInt ("some_counter", count++);
Console.WriteLine (count);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
[default]
exten => s,1,Dial(Zap/1|30)
exten => s,2,Voicemail(u100)
exten => s,102,Voicemail(b100)
exten => i,1,Voicemail(s0)
```

Any command-line input or output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `http://www.packtpub.com/sites/default/files/downloads/1234OT_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Getting Started

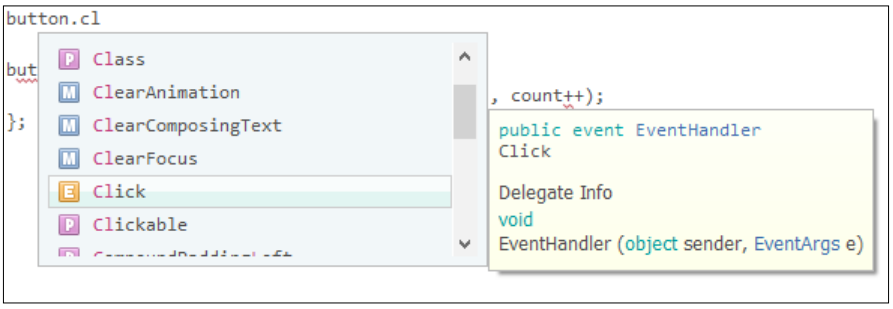
In this first chapter, we will learn how to install Xamarin Studio and the Xamarin plugin for Microsoft Visual Studio and have a quick tour of both. Then we will move towards the creation of our first Hello World application and get it running on the emulator. We will cover the following topics:

- Installing the Xamarin Suite
- Building a HelloWorld App!

Introduction

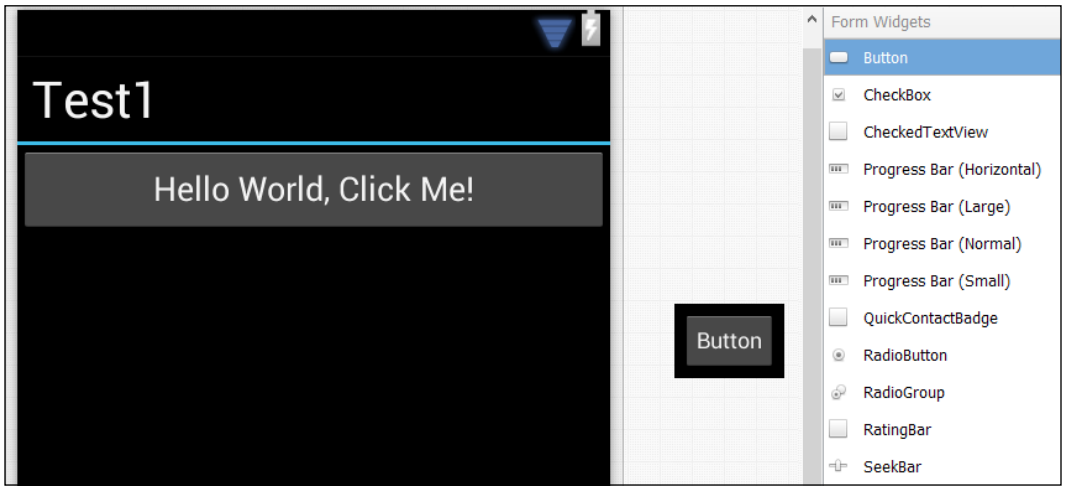
Xamarin Studio is a cross-platform integrated development environment. It is works across platforms in two ways: the IDE is available on Mac OS and Windows (no Linux support announced), and it allows the development of software for Mac OS, iOS, and Android.

While its design can remind you of Xcode, the Apple IDE is only available to Mac owners, Xamarin Studio offers stunning features. Indeed, this IDE allows users to discover the new API easily through a powerful code completion enhanced by the ability to quickly learn about the method and required types. The following screenshot provides an overview of this code completion and the relevant documentation:

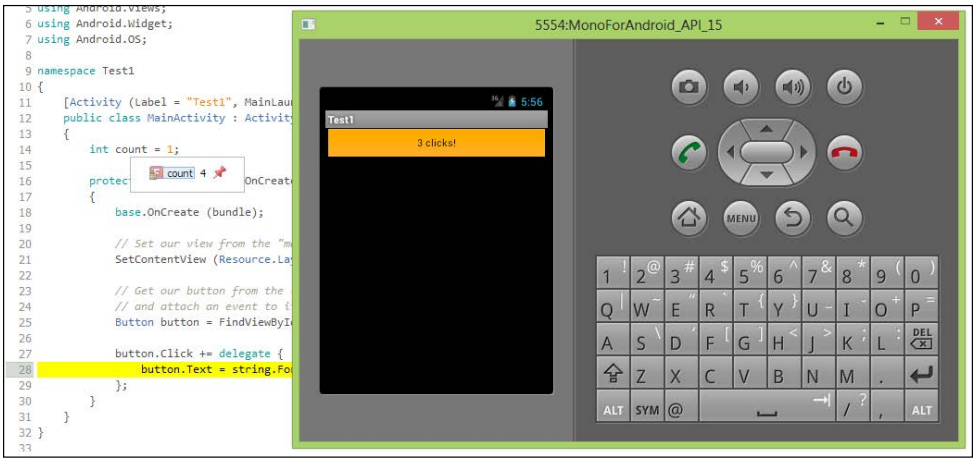


While being trained as a programmer, it seems that I have not developed any artistic capabilities and certainly won't in future. This is quite a problem, and you've certainly already faced an awkward situation like this: *You: "Look at these amazing features!" Project Manager/Client: "Meh. It is despicable; the colors don't match with each other. Could you try this in light blue?*".

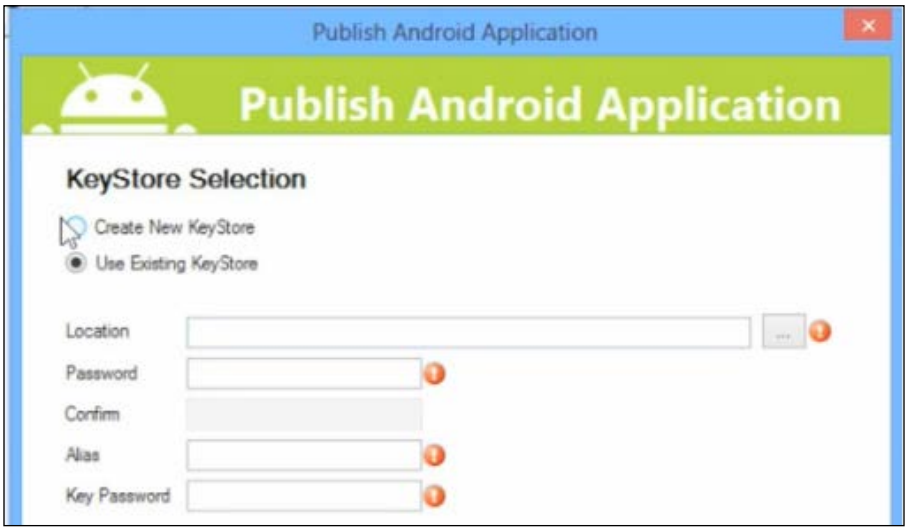
Xamarin Studio, just like any modern IDE, allows the generation of user interfaces in a **WYSIWYG** (**What You See Is What You Get**) interface. In this way, it is a tool not only for programmers but also for designers. Using Xamarin Studio, you will build classy and beautiful apps, while Visual Studio and Netbeans barely provide functioning interfaces. The following screenshot shows the user interface menus and an example of such user interfaces:



Every programmer experienced with modern IDEs such as Visual Studio, XCode, and Eclipse generally uses, on a daily basis, a step-by-step debugger. A debugger allows you to set breakpoints to stop code execution and then walk inside the code, instruction by instruction, while inspecting the values of variables. The step-by-step debugger included in Xamarin Studio is really simple to use yet efficient. The best part about it is that it allows debugging in the emulator (a virtual Android phone on your computer; we'll come back to this later) or on an Android device, live. An example implying the introspection of a variable counting the number of clicks on a button is shown in following screenshot:



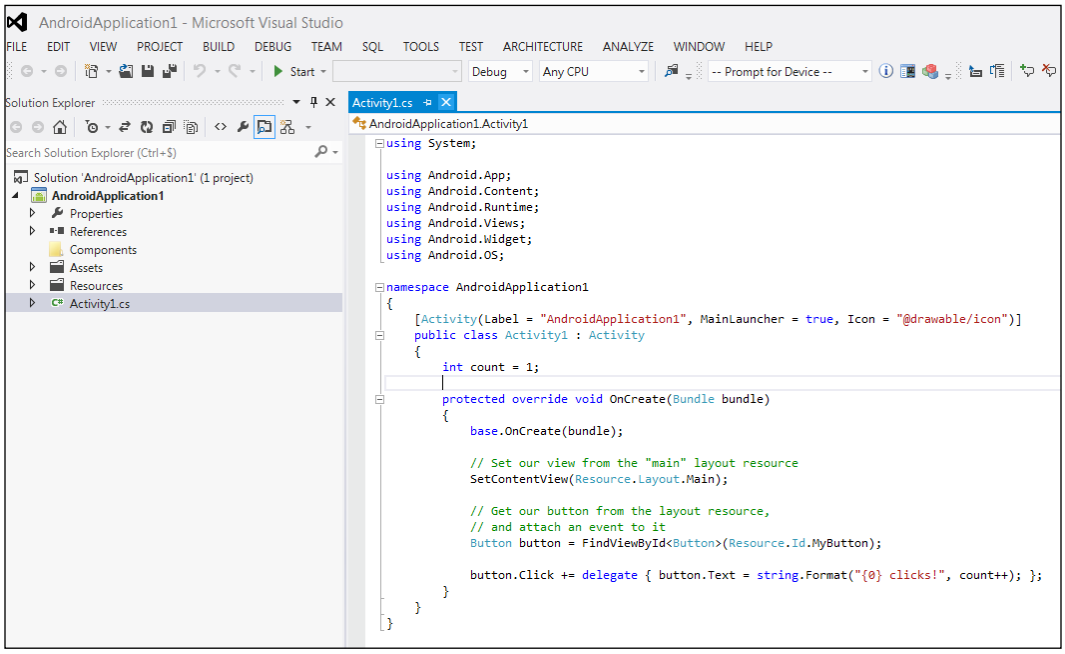
Among a long list of other really useful and entertaining features, Xamarin Studio also excels in creating a highway to your customers' mobile devices. Indeed, publishing apps to the Play Store (the Android market) has never been simpler. The packaging, deployment and shipment to the Play Store processes are smoothly integrated inside Xamarin Studio as shown by the follwoing screenshot:



A quick tour of Visual Studio

Visual Studio is the Microsoft IDE used to develop in any of the various languages that comprise the Microsoft ecosystem. It can serve as an IDE for pretty much anything in the Microsoft bosom, such as console and graphical interfaces, Windows Presentation Foundation apps, websites, web applications, web services, Windows Mobile, and Windows CE. Honestly, Visual Studio is an excellent IDE that could prove very useful after practicing a little; thus, it comes with a price ranging from free to $13,299 depending on the version. Nevertheless, the Xamarin Company has built a Visual Studio plugin that could definitively fit your needs to develop an Android app if you own Visual Studio and are used to it. Xamarin also costs from $999 to $1899 per platform, but it does have a free version that offers most of the features covered in this book.

After talking about the financial aspect, let's focus on features. It turns out that all Android-focused features are exactly the same and have to be used in the same ways in almost the same menus. Choosing between Visual Studio versus Xamarin Studio will, in the end, be a personal choice that we leave to the reader's discretion. However, Visual Studio consumes a lot more resources than Xamarin Studio. Therefore, if you are not equipped with good hardware, you should definitely go for Xamarin Studio.



In the rest of this book, screenshots and paths will be based on Xamarin Studio, which we believe is most used by newcomers to Android development (who use Xamarin and C#). We will indicate Visual Studio paths and options when they are different from those in Xamarin Studio.

Installing the Xamarin suite

In order to develop Android applications using C#, we have to set up our Android development environment, and more specifically, we have to install Xamarin Suite. This first recipe will show you how.

Getting ready

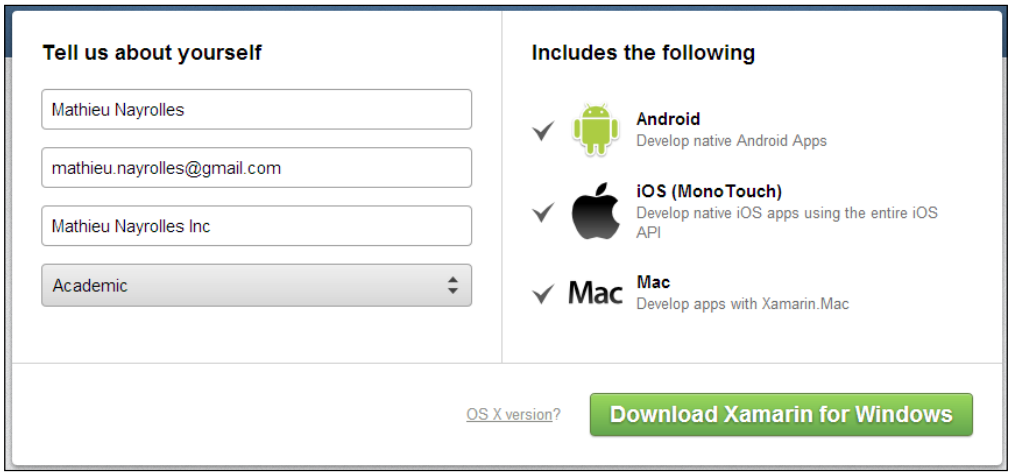
Xamarin Studio is available for the Mac and Windows operating systems and does not have a set of minimum hardware requirement to match. However, the technologies embedded in Xamarin Studio, such as the Android SDK, do have minimum requirements as follows:

- Operating system:
  - Windows XP (32-bit), Vista (32- or 64-bit), 7 (32- or 64-bit), 8 (32- or 64-bit)
  - Mac OS X 10.4.8 or later (x86 only—Intel chips)
- Hardware requirements:
  - 600 MB of available space and an additional 100MB per platform (iOS, Android, Windows)
  - This is not official, but I recommend an i3-equivalent processor and 4 GB of RAM to run Xamarin Studio and one emulator without any trouble.
  - Again, this is not official. In case of industrial needs, the debugging available on a real Android phone, while not mandatory, will accelerate your production. The phone cost is definitely worth it in terms of development time.
- Software requirements:
  - In case you intend to use the Visual Studio plugin inside Xamarin Studio Suite, you must own a license for Visual Studio 2010 or 2012 in a non-express edition.

How to do it...

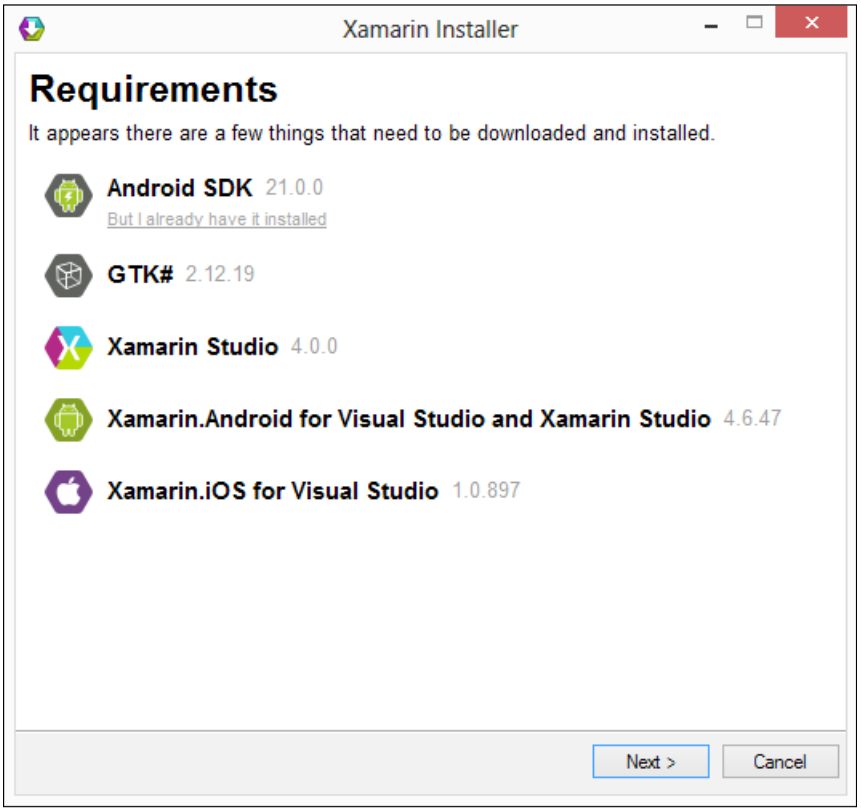
In order to install Xamarin Studio, please follow the given steps:

1. Browse to `http://xamarin.com/download`, enter your information, and choose your operating system.



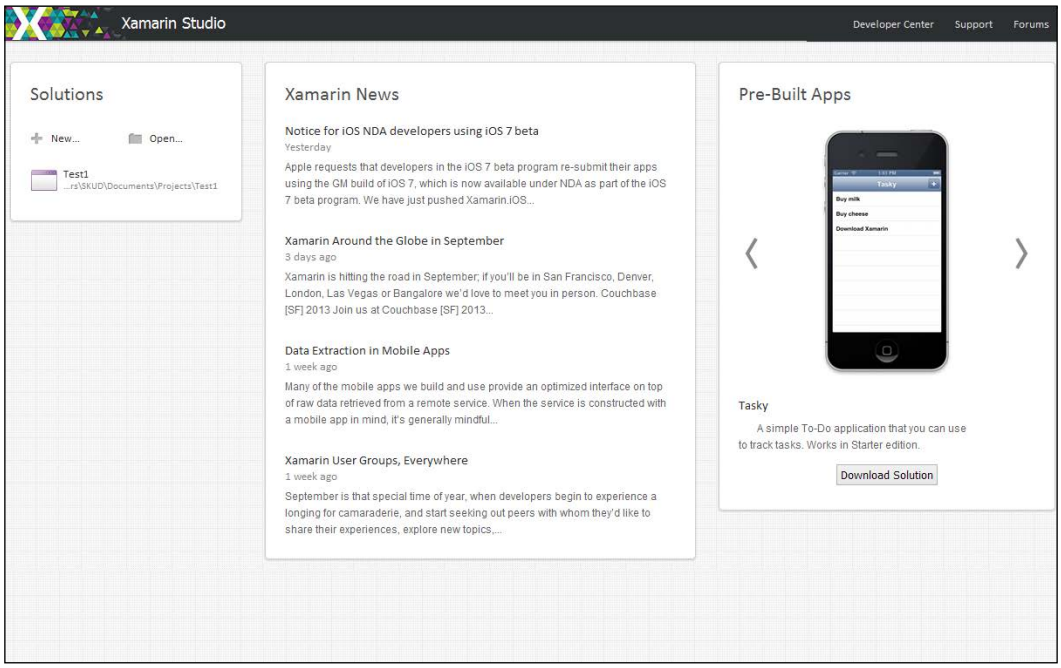
2. Execute the `XamarinInstaller.exe` or `XamarinInstaller.dmg` file that you subsequently receive and accept the terms and conditions by clicking on **Next**.
3. Choose what you want to do with your Xamarin tool suite: Android or iOS or both.
4. Accept the directory in which the Android SDK will be downloaded and installed.

5. For now, if you have checked everything, as we did, we should view the requirements of Xamarin Installer. The title is a bit misleading. In reality, this is not a list of requirements but of software that will be downloaded and installed during the Xamarin Suite installation.



6. After a long period of downloading, the various software will install themselves on your computer.

7. The installation is completed. You can now open Xamarin Studio.



The displayed screenshots of these seven steps have been taken during installation on a Windows 8 system. However, all the Windows installations are exactly the same. For Mac, the only differences are graphical.

How it works...

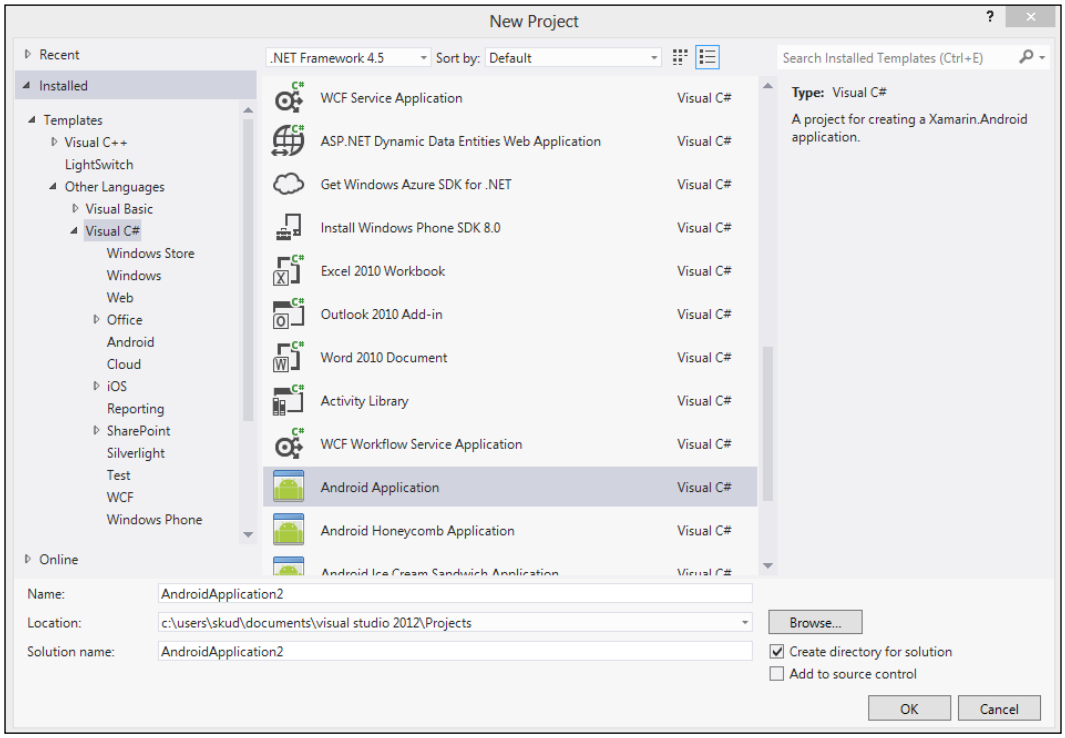
Xamarin is based on the Mono Project (refer to the *Appendix*), which is an Open Source implementation of the **CLR** (**Common Language Runtime**), but that does not explain how Xamarin manages to create Android apps. Indeed, the Mono parts explain how we can execute C# based applications on many platforms but not how these apps run on our Android devices.

The Xamarin compiler, which is responsible for the transformation of C#.Net into Android-understandable code, is a very powerful tool. Indeed, it will compile and link all your C#.Net code—using proprietary technologies and processes—directly into an APK file. **APK** (**Android Application Package**) is the required format to deploy and install applications onto Android devices. The deployed APK on the targeted Android device will take advantage of the **Just In Time** (**JIT**) compilation. This compilation is a hybrid approach between the interpreted language, where an interpreter translates the language to the underlying machine each time we need it, and the compiled approach, where the whole code is compiled into a machine. The interpreted approach loads at the speed of light but comes with poor runtime performance, while the compiled approach loads very slowly but offers better performance. Hence, the JIT strategy consists of translating code continuously, just as an interpreter does, but it saves the translated code with a cache mechanism to avoid performance degradation. In other words, JIT offers the best of both worlds. Last but not least, it runs natively on Android devices.

If you intend to use the Visual Studio plugin, here are the steps to verify that it has been properly installed:

1. Run Visual Studio.

2. In the **File** menu search for **New Project** and select **Visual C#** in the **Other languages** list. Finally, you can select and create a new Android Application Project.



After clicking on the **OK** button, Visual Studio will open the Android Project perspective and a new Android solution.

There's more...

Xamarin comes in two different versions: Business and Enterprise. The prices are $999, and $1899 per platform (Android, iOS, and Mac OS) and per developer, respectively. It's quite an investment, especially if you are on your own—understood to be "not sponsored by a company that handles license-related fees"—so it's mandatory to reduce your needs and buy the adapted version.

The starter (free) version will allow you to build very small apps that contain no more than 32 KB of compiled code. In other words, you will have the taste of Xamarin, but your applications will stay very simple—forever. A less constraining limitation related to the free version is the impossibility of invoking native third-party libraries such as P/Invoke. We can definitely build a world-class app without third-party libraries; it just takes longer. The indie version ($299/platform/dev) only takes down the size limitation and will therefore meet the needs of the majority. The two latest and more expensive versions will mainly provide support from the Xamarin team. The Business version offers e-mail support, Visual Studio support, and in-house deployment, while the Enterprise version will add $500 worth of ready-to-use components for Xamarin Studio, a one-day SLA, hotfixes, a technical kick-off session, and code troubleshooting provided by Xamarin engineers.

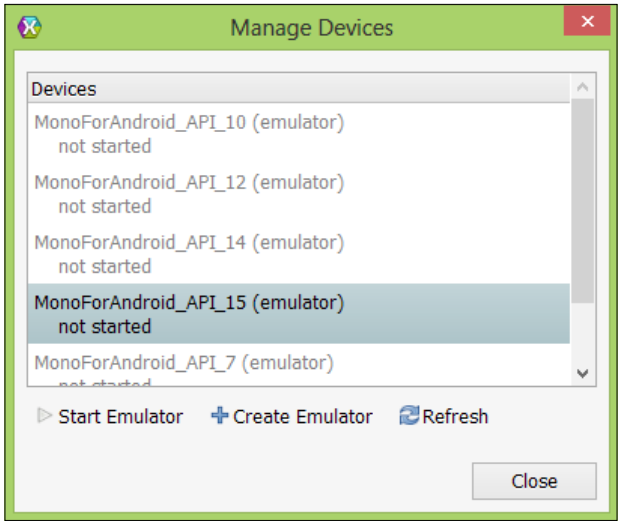
While many possibilities are offered to you, the Starter Edition should be enough to follow this book. In case you want to determine whether or not a feature is worth its price, go for the trial edition.

Testing the simulator

For both systems (Xamarin Studio and Visual Studio), we should determine whether or not the simulator is well configured. If so, there is a good chance that our whole environment is as well.

In Xamarin Studio:

1. Go to the **Emulator** menu via the **Project** menu | **Android Device Target** | **Manage Devices**. The following window appears:

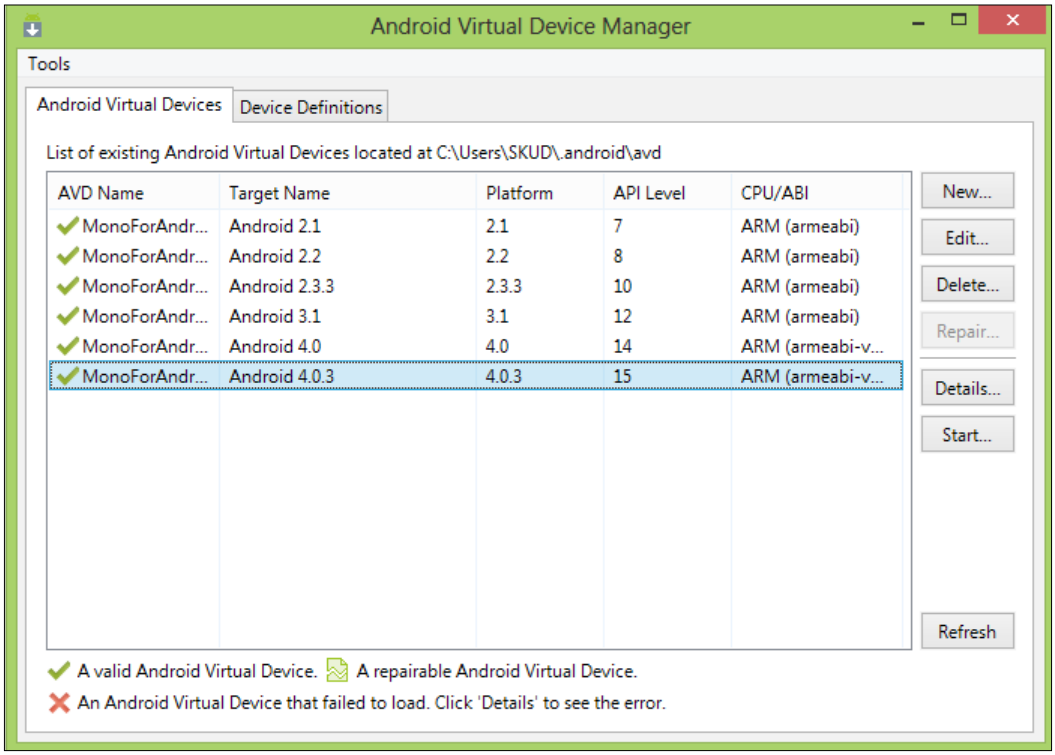


The different versions of MonoForAndroid—which could be referred to as MonoDroid informally—target the different versions of Android. The newer (higher) versions are made for the latest Android releases.

2. Select the higher version (API_21, which targets Android 4.0.3, at the time of writing—November 2015) and click on **Start Simulator**. After a while, especially on the first start, the simulator will show up. It can be used exactly as an Android Phone.

In Visual Studio:

1. Access the **Simulator** menu via the **Tool** menu and then start **Android Emulator Manager**. The following window appears:

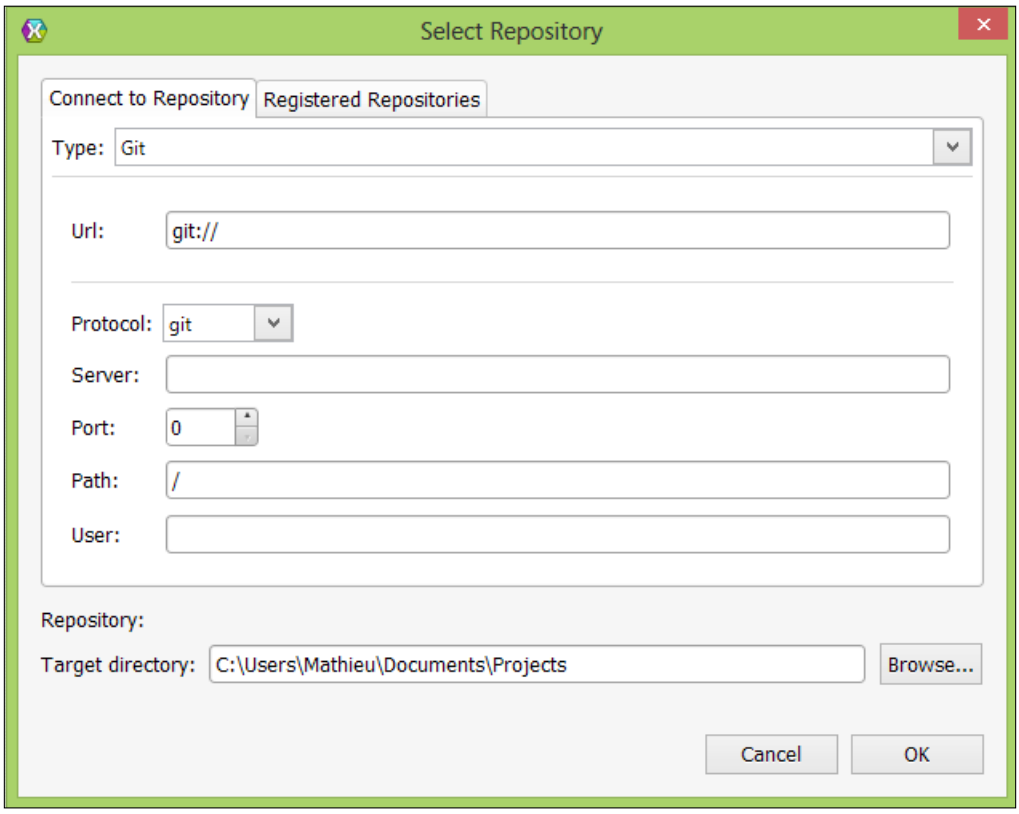


2. Select the higher version and click on **Start**. As with Xamarin Studio, the simulator can be used as an Android Phone.

After pressing the **Start** button, the Android emulator will start. This operation may take a few moments depending on your hardware. Nevertheless, after a moment, the emulator will provide a user experience really close to that of a real Android device.

Connecting Xamarin Studio and Visual Studio to a versioning control system

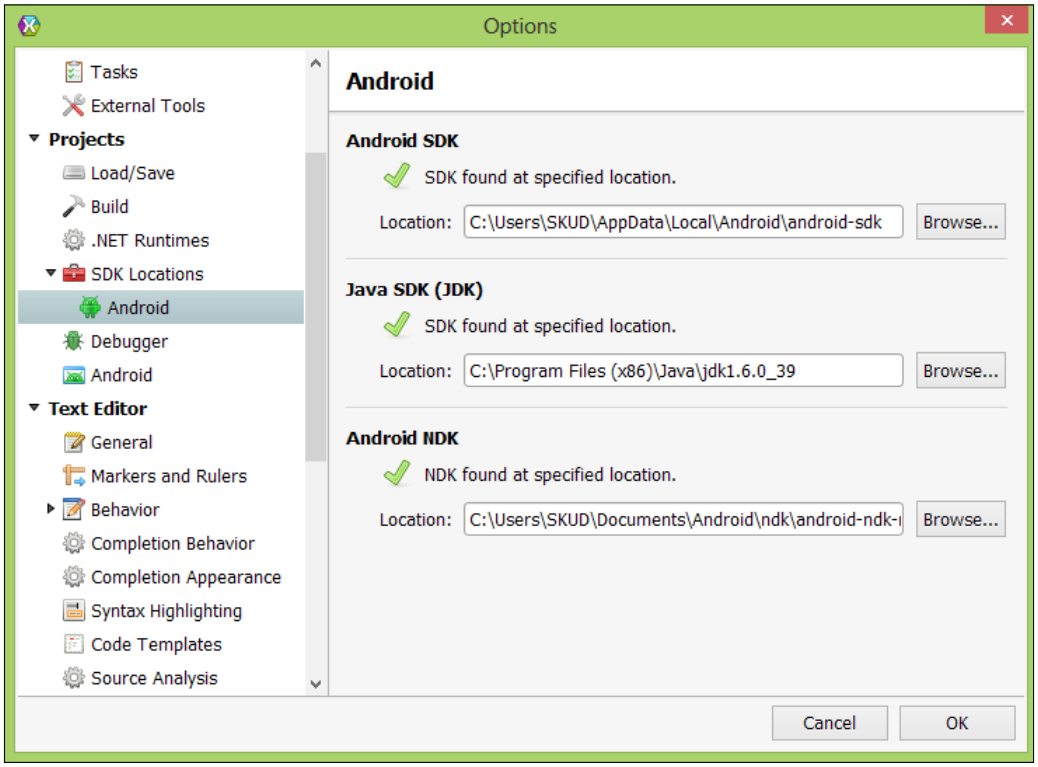
A good habit you should always have is to work with a versioning control system, no matter the size of the project. The integration of git/svn into Xamarin Studio comes off without a hitch. You just have to browse the **Version Control** menu and then press the **Checkout** button.



Unfortunately, Visual Studio does not offer any embedded support for `git` or `svn`, we recommend giving `libgit2` and `visualsvn`, respectively, a try.

Using another Android SDK

If you already are into Android development you might need, in a particular configuration such as maintaining an application on enterprise devices that are not up-to-date (for security reasons), to use a different Android SDK than the one installed with Xamarin Studio. To do so, browse the **Tool** menu in Xamarin Studio and then press the **Option** button. In the new window that comes up, search for **Android** under **SDK Locations** in the **Projects** section. The same menu exists in Visual Studio under **Tools** | **Options** | **Xamarin** | **Android Settings**.



Pay attention while modifying these parameters. Indeed, any misconfiguration will prevent your Xamarin Studio from running any Android applications.

See also

- See also the next recipe for the very first hands-on tutorial on Xamarin Studio.

Building a Hello World App!

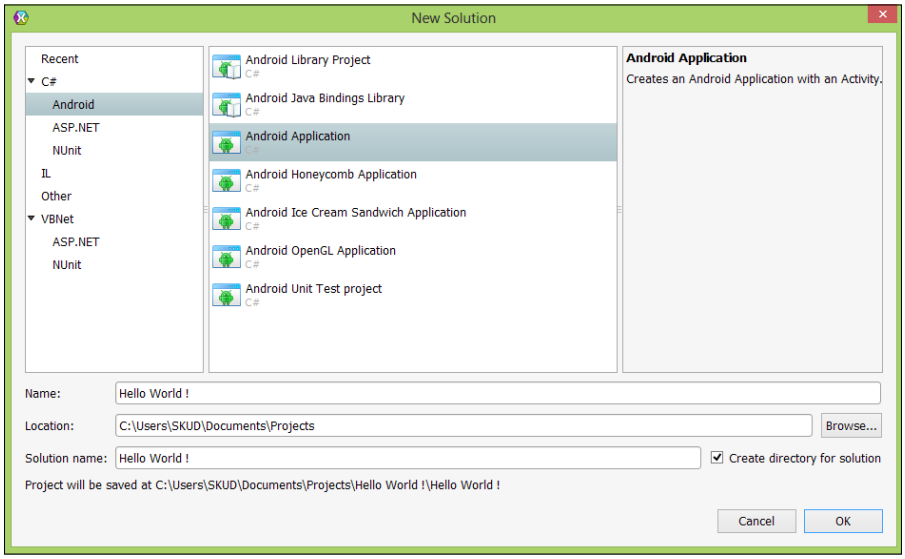
It's now time to get our hands dirty and write some simple code by ourselves to output the famous Hello World message. In this recipe, we will create a new Android project in Xamarin Studio. This project will lead to an Android app that outputs **Hello World!** after pressing on a **Say Hello!** button. This first application will run in the simulator and on a real Android Device. Moreover, this application will allow us to discover the architecture of an Android project, the same as its specificities.

Getting ready

For this recipe, we will need to have successfully installed the Xamarin Studio tool suite and run the emulator at least once in order to confirm that everything is working well.

How to do it...

1. Run Xamarin Studio, browse the **File** menu, and then click on **New Solution**. In the new window that comes up, select **Android Application** under **C#** | **Android**.



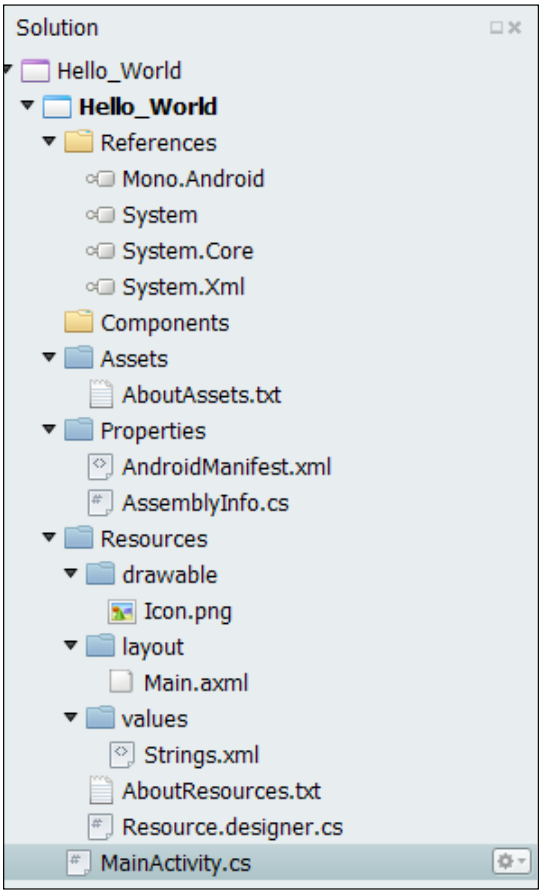
2. Make sure that **Create directory for solution** is checked.
3. Name your project `Hello_World` and press **OK**.

Note that the project name only accepts the letters A-Z, the digits 0-9 and the hyphen (-), underscore (_), and period (.) characters. This is the reason for the underscore between "Hello" and "World".

Now that you have a new project named **Hello_World**, it's appropriate to take some time to explain the files and folders comprising the architecture of an Android project. The project you have just created is already filled with a large number of files and folders.

- By order of appearance, the first folder is named `References` and contains four libraries `Mono.Android`, `System`, `System.core`, and `System.XML`. This folder will contain all libraries that you used in your project in the same way as a pure .Net project.
- The next folder, named `Components`, contains components that you previously developed for Xamarin or downloaded from the rich Xamarin database. Most of the components directly available from the Xamarin Component Store are free and ready-to-use components that will add commonly needed functionality to your project, ranging from interacting with Facebook to including Easter eggs in your applications.
- The two following folders are `Assets` and `Resources`, just like in native Android apps. They are pretty similar however, in the sense that they contain files that are not code, for example, images, songs, XML files, and pretty much anything your application will need. So why do we have two folders if they have the same purposes? In reality, the external files placed in the `Assets` folder will be easily accessible at runtime by using the Asset Manager (we'll come back to this later), while for the ones contained in the `Resources` folder, you will have to declare and maintain a list of resource IDs that you might use at runtime. In general use, we will put all images, sounds, icons, and other external files in the `Resources` folder, and the `Assets` folder will be privileged for dictionaries and XML storage.
- The next folder in line is named `Properties` and contains two files: `AndroidManifest.xml` and `AssymblyInfo.cs`. They are responsible for the Android version and permission your application targets, and your project information (such as the version and build number), respectively.

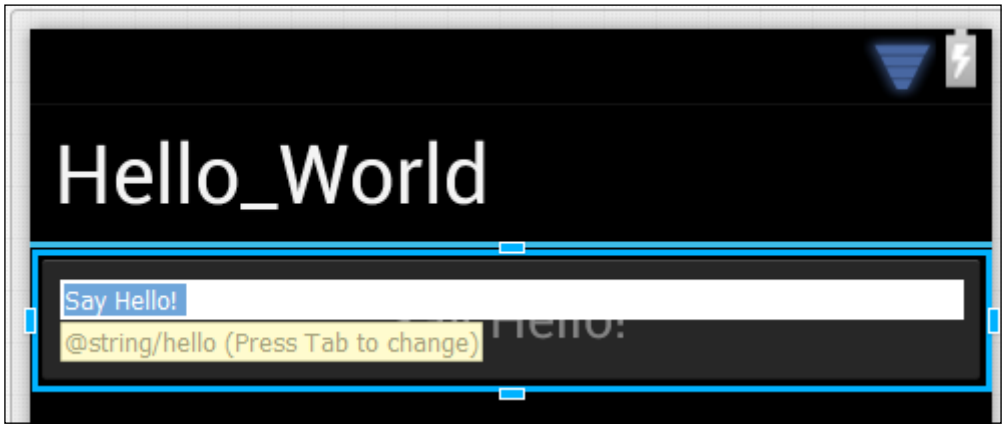
- Finally, the `MainActivity.cs` file is the main class of our application; it contains our very first Android apps lines of code.



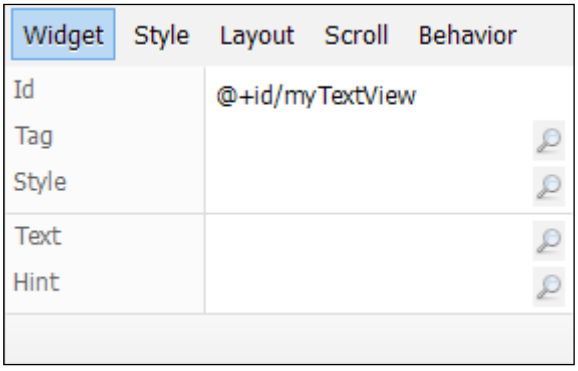
4. Open the `Main.axml` file in the `Layout` folder under `Resources`. A new file containing the graphical interface of your Hello World! app should appear.

The `.axml` file is, in fact, an XML file that allows code completion in Xamarin Studio and Visual Studio for graphical element. It also defines the position and properties of the graphical element.

5. Double-click on the button on the graphical interface and change the text to **Say Hello!**



6. From the **ToolBox** pane, on the right of both Xamarin Studio and Visual Studio, locate **Text (Large)** in the **Widgets** section.
7. Drag and drop the **Text (Large)** widget below the **Say Hello** button.
8. Double-click on the **Text (Large)** widget you just inserted and delete the text.
9. With the **Text (Large)** widget selected, locate the **Properties** pane in the bottom-right corner of the IDE and change the **Id** tag style from `@+id/textView1` to `@+id/myTextView`.



The **Id** tag style is the unique identifier of the graphic element. You will use these IDs in the C# code in order to manipulate the graphical elements. Therefore, you must give them meaningful names.

10. Return to the `MainActivity.cs` file and locate `Button button = FindViewById<Button> (Resource.Id.myButton);`. Add these lines of code below it:

```
// Get our TextView from the layout resource, and attach an
event to it
  TextView view = FindViewById<TextView>
(Resource.Id.myTextView);
```

11. Locate the following lines of code:

```
button.Click += delegate {
  button.Text = string.Format ("{0} clicks!", count++);
};
```

    Modify them as follows:

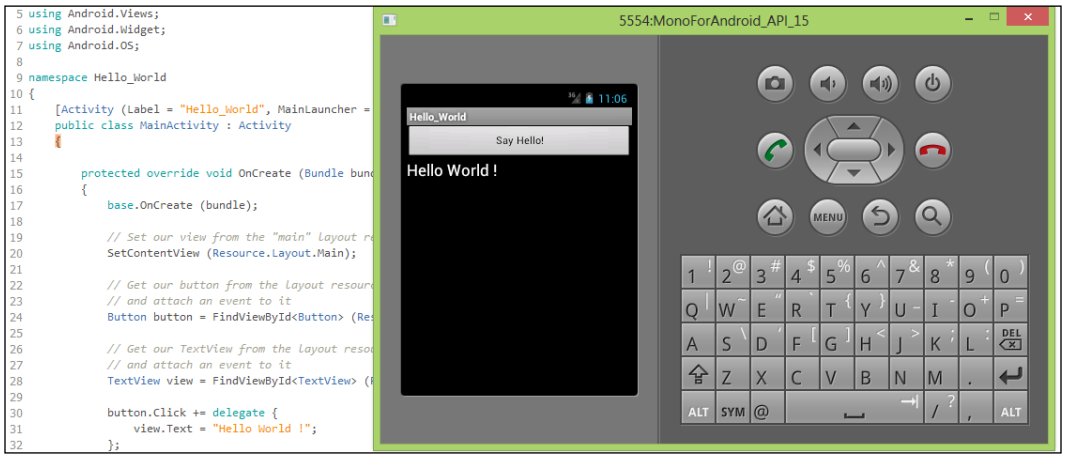
```
button.Click += delegate {
  view.Text = "Hello World !";
};
```

12. Delete the code `int count = 1;` in the 14th line.
13. Browse the **Project** menu, then click on **Manage Devices...** under the **Android Device Target** submenu.
14. Locate the `MonoForAndroid_API_15` emulator, select it, and click on **Start Emulator**.

You will always need to have an emulator started prior to launching your project as the project requires a device or an emulator to execute on.

15. Run your `Hello_World` project by pressing the  button in the top-left corner of Xamarin Studio. After a moment, your application will be pushed on the emulator and executed.

16. Here we go! Our very first Android application is now running on the Android emulator. You can press the **Say Hello!** button and **Hello World!** will appear.



How it works...

The How it works section of our Hello World recipe could appear rudimentary if you already have some graphical user interface development experience with C#. However, mastering the basics is never superfluous.

For this code example, and all the following one throughout this book, we will display the whole piece of code, add some references to it, and then explain the code using the references.

```
using System;
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;                                    [1]
using Android.Widget;
using Android.OS;

namespace Hello_World {                                 [2]

  [Activity (Label = "Hello_World", MainLauncher = true)]
  public class MainActivity : Activity {
    protected override void OnCreate (Bundle bundle) {
      base.OnCreate (bundle);

      // Set our view from the "main" layout resource
```

```
        SetContentView (Resource.Layout.Main);      [3]

        // Get our button from the layout resource,
        // and attach an event to it
        Button button = FindViewById<Button>
        (Resource.Id.myButton);                     [4]

        // Get our TextView from the layout resource,
        // and attach an event to it
        TextView view = FindViewById<TextView>
        (Resource.Id.myTextView);                   [5]

        button.Click += delegate {                  [6]
          view.Text = "Hello World !";
        };
      }
    }
  }
```

Here is an explanation of `Hello World MainActivity.cs`:

- **[1]**: Most of the C# applications and in general .Net applications files start by using directives. Using directives, enlist all the namespaces that your application will use on a frequent basis, saving you time. For example, the `TextView` class is inside the namespace `Android.Widget`, and without the corresponding directive, you will have to type `Android.Widget.TextView` each time you want to create such objects.
- **[2]**: C# also enables the creation of programmer-defined namespaces. Each class in that namespace will be visible without any need to specify using `Hello_World`. Moreover, you could refer to your `Hello_World` namespace in other namespaces and have access to your classes.
- **[3]**: Our layout is defined by the `Main.axml` file in the `Resources/Layout` folder. We therefore need to bind this file to the screen. To do so, we call the `SetContentView()` method.
- **[4-5]**: In the fourth and fifth instruction sets, we create a `Button` and a `TextView` instance by using the `FindViewById` constructor. This generic class enables the creation of graphical elements by using their IDs. These IDs are user defined at the creation of these elements.
- **[6]**: `delegate` is a method signature type, and it's the keystone of event programming in C#. Here we are facing an inline `delegate` instance, which introduces an anonymous method in reality. This method (`view.Text = "Hello World !";`) will be executed on the `button.Click` event by means of the `+=` which add this new behavior at the end of the existing ones.

There's more...

Let's have a look at the following sections in order to learn about testing on a physical device and checkout this book's free source code.

Testing on a physical device

Testing your applications on physical devices will save you a lot of time, as the simulators are rather slow. You can see the complete procedure to do so, depending on your system, at `http://developer.xamarin.com/guides/android/getting_started/installation/set_up_device_for_development/`.

Source code

This book comes with a lot of code examples that are freely available from Github. The code for this first chapter can be found here:

`https://github.com/MathieuNls/mastering-xamarin-studio/`

For the following chapters, adapt the following URL:

`https://github.com/MathieuNls/mastering-xamarin-studio/tree/master/chap1`.

In order to access all the codes given in the book, replace the end of the URL from chap1 to chap2 for Chapter 2 and so on.

See also

- The next chapter introduces Android Activities Lifecycle. It will show you how your application lives and dies.

2

Mastering the Life and Death of Android Apps

The previous chapter introduced us to the fundamentals of our programming tool, such as IDEs and virtual machines, and allowed us to create a Hello World app. This second chapter is fundamental because you will learn everything about Android activities and Android life cycles, which are mandatory to create Android apps. At the end of this chapter, we will have a solid understanding of how Android apps are born and their life and death. Moreover, mastering the concepts exposed by this chapter will result in better stability and avoid crashes and resource bloat, while a poor understanding can even introduce OS instability.

In this chapter, we will cover the following recipes:

- Understanding Android activities
- Practicing the activities' lifecycles
- Going through the state saving management

Introduction

Android activities are the main building blocks of Android applications. The official Android activity definition, coming from the official documentation (`http://developer.android.com`) states the following:

> *An activity is a single, focused thing that the user can do. Almost all activities interact with the user[...].*

In this chapter, we will focus on understanding them in order to master their lifecycle.

Understanding Android activities

In this recipe, we will focus on understanding Android activities with short and focused examples.

How to do it...

1. Open any Android application you find on the store, for example, Mindup (`https://play.google.com/store/apps/details?id=io.mindup.mindup&hl=en`).
2. Use the menu to change the page.
3. Press the **Return** button of your phone, which will redirect you to the first screen you were at.
4. Use the burger button of your phone; you'll be redirected to the home screen of your phone.
5. Use the burger button again and return to the application. You'll find the application as it was before you left.

This simple and expected behavior is induced and monitored by Android activities, which are the building blocks of any Android application.

How it works...

Being the main building block of Android apps does not ensure simplicity. Indeed, activities are a very unusual programming block. Let's compare Android activities with the C# traditional application. Every C# program in the world has the following lines of code:

```
class Program {
  static void Main(string[] args) {
  }
}
```

These lines will be—more or less—the same in every program. Moreover, these lines represent the starting point of your program. C# is not the only language that builds around a `static()` method for launching programs. Actually, most of the languages used, such as Java, C, C++, and so on work this way. On the contrary, in an Android app, the starting point of the application can be any registered activity. In our Hello World example from *Chapter 1*, *Getting Started*, our unique activity is named "Hello World" and is represented by the `MainActivity` class:

```
[Activity (Label = "Hello_World", MainLauncher = true)]
public class MainActivity : Activity
```

The `Activity()` method does have a label and a `MainLauncher` attribute that informs the Android systems about the main starting point. If we remember the code from the `MainActivity` class, there was no `static void Main` method but only one method named `OnCreate`:

```
protected override void  onCreate (Bundle bundle)
```

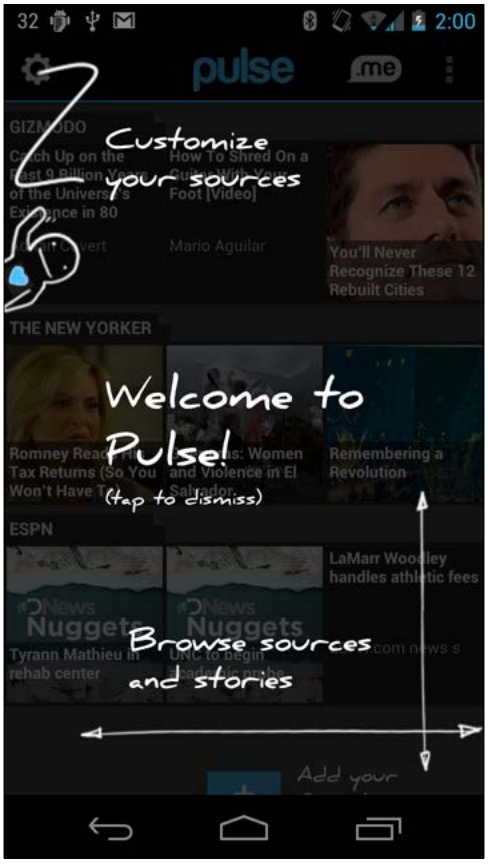
Even if the Android architecture allows programmers to create many activities, in practice, a large majority of Android applications have only one activity. Despite the official definition proposed by the Android team, which we discussed earlier, activities often drift from a *single, focused thing* to a bunch of functionalities that handle the whole application. Therefore, most of the Android applications have only one activity, and this activity is therefore the starting point of the applications. So you might ask why Android creators have implemented this functionality that allows us to create many entry points in the same program. If an application is forced to abort for any reason, such as a crash, the OS may try to restart the latest activities as they were before the crash. In other words, you can build applications that provide many functionalities divided into several activities so that if one of the activities crashes, it can be restored by the OS without affecting the rest of the application. Moreover, the activities fragmentation serves another purpose, that is, an OS can pause activities if they become inactive and awake them (unpause) at the user's demand.

Since the beginning of this section, we defined activities as the main building blocks of Android applications and as starting points that can be *created, paused*, *and even restored or restarted*. Therefore, activities have *states*. Activities also have *lifecycles* that we must master.

Activities can have four different states throughout their lives, and the OS, as we have seen earlier, will use these states to orchestrate the allocations of physical resources between the different applications running on the phone or Android device. These four states are officially named *running*, *paused*, *stopped*, and *killed*.

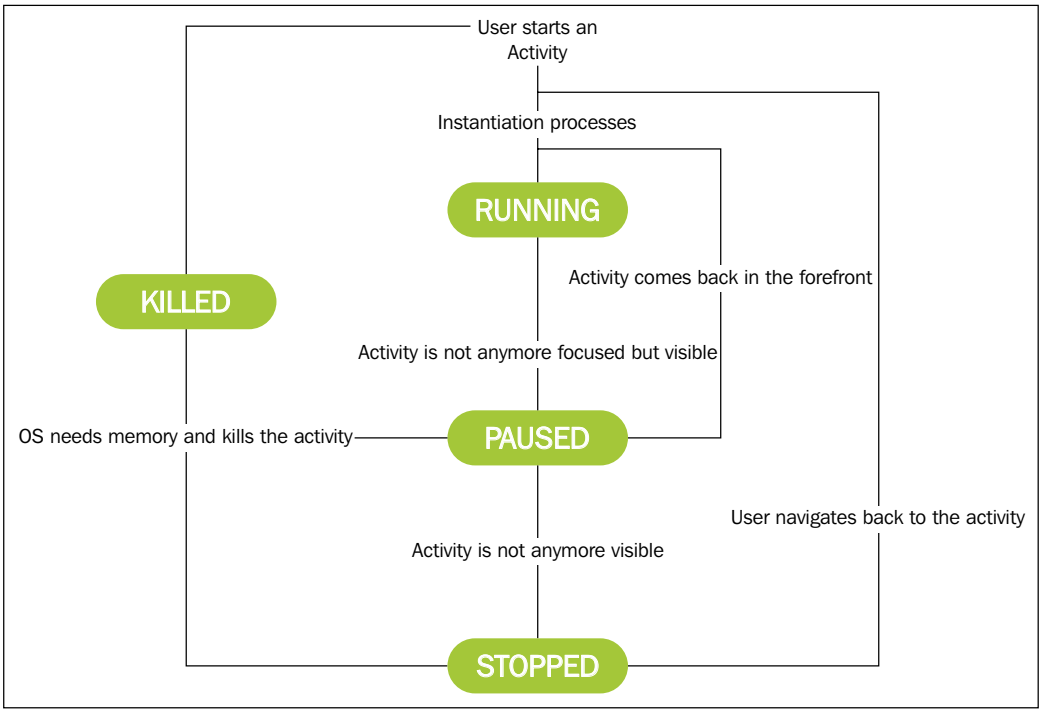
The *Running* state qualifies the application that is in the forefront, that is, the application that is currently being used by the user.

When a *Running* activity loses the focus but is still visible, the application is paused. The *Paused* state can be tricky to see, but it typically occurs when a translucent menu shows up with "How to Use" instructions. Because a picture is worth a thousand words, the following image shows the **Welcome Semitransparent** tutorial of the Pulse application, which is one of the best news reader apps out there:



This is a perfect example of a paused application. The main activity that is handling the Pulse application is paused because it was being used in the forefront but lost the focus while remaining visible and completely alive.

The next state is known as **stopped** and occurs when an application is not visible anymore. In this particular state, the application holds its inner data (filled in fields and so on) but a stopped application will often be killed by the OS in order to free some memory. Finally, when an activity is in the paused or stopped state, the OS can terminate it by gently asking it to close or, in a more brutal and instable way, by killing their underlying processes. If the user wants them in the forefront again, they have to be completely restarted from scratch. The following figure shows the actions that lead to the different states:



We can see that when a user starts an activity, some instantiation processes have to be done (we will get back to that in the next recipes), and then the activity is entered in the running state. When the activity is no longer focused but still visible, it's paused; however, it will be stopped if the activity is not visible at all.

The `paused()` method will be invoked even if the activity goes straight from visible to not visible.

If the activity is paused and comes back to the forefront or the user navigates back to a stopped Activity, the activity will be in the running state. Note that the change in state from paused to running requires less instantiation work than from stopped to running. Also, it is possible that the paused and stopped activities may have been killed by the OS to free some resources. In that case, the whole instantiation process has to be executed again, just as we did for the first launch.

There's more...

As always, when programming, there are exceptions to simple statement sequences. Android programming comes with its bunch of exceptions too. The configuration changes occur, for example, when the phone is rotated. In such cases, the application must adapt itself to the new screen orientation and might want to display additional functionalities, for example, the virtual keyboard. If we have to follow the state sequence we just discussed, adapting an activity will cause it to pass by all states: `onPause`, `onStop` and `onDestroy` followed by the `onStart`. Because activities are the main mechanism for interacting with the user, we must keep them responsive. In order to achieve this, Android exposes a special API, which bypasses classical states and operates a quick deconstruction/reconstruction cycle of the activity.

See also.

The next recipe *Practicing the activities' lifecycles* for a hands-on tutorial on Android activities' life cycles.

Practicing the activities' lifecycles

The Android's `Activity` class exposes seven methods for managing the different states. These seven methods are in protected visibility, meaning that anyone specializing the `Activity` class will be able to override these methods. If we take a look at our `MainActivity` declaration again, we can see that `MainActivity` specializes the `Activity` class using the `:` symbol:

```
[Activity (Label = "Hello_World", MainLauncher = true)]
public class MainActivity : Activity
```

The inheritance mechanism is also referred to as specialization. It is a strong object-oriented mechanism that establishes an "Is-a" relationship. Classes inherit attributes, methods, and behavior of the classes they specialize. The classes that are an outcome of this inheritance are referred to as derived classes or subclasses. Moreover, subclasses can add new behaviors to the inherited ones and even *override* or redefine inherited behaviors. While playing around with activities lifecyles, we redefine the base behavior contained in the `Activity` class.

By means of inheritance mechanisms, we can specialize and customize the behavior of these seven methods to fit our needs while implementing a new activity. If we pay attention to our very first and autogenerated statement of the `MainActivity` class, we notice that the `OnCreate()` method is overridden. In this override—and all other states-related method overrides—the new behavior is added after a call to the `base()` method.

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);
  […]
}
```

A call to the base will execute the behavior of the superclass. In our case, the `Activity` superclass contains mandatory code to handle states, and we are only agreeing this code with specific statements to fit our needs.

Getting ready

In order to follow this recipe, you must have Xamarin Studio or Visual Studio with the Xamarin Studio up and running. Moreover, you should have a thorough understanding of inheritance mechanisms in object-oriented programming. How do we determine whether our understanding is good enough? Basically, the two previous tips about inheritance and base calls should have been only reminders to us. If you are not comfortable with this OOP fundamental, we warmly encourage you to look for literature about inheritance.

How to do it...

In this section, we will see all the methods related to the states of activities. Let's take a look at the following steps:

1. Implement the `OnCreate()` method in your `MainActivity` class:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);
  SetContentView (Resource.Layout.MainActivity);
  var helpButton = FindViewById<ImageButton>
  (Resource.Id.loginQuestion);
```

```
    helpButton.Click += (sender, e) => {
      var builder = new AlertDialog.Builder (this)
      .SetTitle ("Need Help?")
      .SetMessage ("Here What you Should Do")
      .SetPositiveButton ("Ok", (innerSender, innere) =>
      { });
      var dialog = builder.Create ();
      dialog.Show ();
    };
  }
```

2. Implement the `OnStart()` method in your `MainActivity` class:

```
protected override void OnStart() {
  base.OnStart ();
  TextView aTextView = FindViewById<TextView>
  (Resource.Id.myTextView);
  aTextView.Text = "Hello !";
}
```

3. Implement the `OnResume()` method in your `MainActivity` class:

```
protected override void OnResume() {
  base.OnResume ();
  // Some init code
  aTextView.Text = BluetoothAdapter.DefaultAdapter.Address;
}
```

4. Implement the `OnPause()` method in your `MainActivity` class:

```
protected override void OnPause() {
  base.OnPause ();
  // Save data to persistent storage
  // Desallocate big objects
  // Free Hardware like Bluetooth
}
```

5. Implement the `OnStop()` method in your `MainActivity` class:

```
protected override void OnStop() {
  base.OnStop ();
  StopService (new Intent (this, typeof(Service)));
}
```

6. Implement the `OnRestart()` method in your `MainActivity` class:

```
protected override void OnRestart() {
  base.OnRestart ();
}
```

7. Implement the `OnDestroy()` method in your `MainActivity` class:

```
protected override void OnDestroy() {
  base.OnDestroy ();
  if (!IsStopped) {
    StopService (new Intent (this, typeof(Service)));
  }
}
```

How it works...

The seven methods of the activity class are: `OnCreate()`, `OnStart()`, `OnResume()`, `OnPause()`, `OnStop()`, `OnDestroy()`, and `OnRestart()`. These method invocations affect the states as described by the next figure. An activity passes through `OnCreate()`, `OnStart()`, and `OnResume()` in order to acquire the *Running* status. Methods `OnPause()`, `OnStop()`, and `OnDestroy()`, will lead to the *Pause*, *Stopped*, and *Destroyed* states, respectively. If the user navigates back to an activity that was stopped, the `OnRestart()` method is called, which in turn calls the `OnStart()` and `OnResume()` methods in order to get the activity in the *Running* state again. Finally, we notice that if the OS is in need of memory and has *Killed* the activity, the overall instantiation process has to be done again.

Now, we will discuss each method in detail:

OnCreate

The `onCreate()` method is the first one in getting an activity started. This method will always be overridden for initializing your graphical interfaces (as we did for the Hello World example), inner variable, and other more complex actions, such as starting an embedded database. The signature of the `OnCreate` method is `protected override void OnCreate (Bundle bundle)`.

In the code sample given in the *How to do it...* section, we override the `OnCreate()` method. The first statement in the method invokes the `OnCreate()` method of the `Activity` class. As a reminder, we must call the base method in the method we override in order to keep our Android application working. In the second statement, `SetContentView (Resource.Layout.MainActivity)`, we load the graphical interface related to the current activity; we will talk in depth about graphical interfaces in the *Chapter 3*, *Building a GUI*. The `OnCreate()` method being the first one to be called, is the most suitable method to bootstrap the graphical interface. In the following `FindView` statement, we get a reference to a button on our graphical interface using the `FindViewById<ImageButton> (Resource.Id.loginQuestion);` method. Again, these methods too will be extensively discussed in the next chapter. Using the acquired reference, we add some behavior to the `Click` event of the `helpButton` instance. We create an `AlertDialog` instance, with **Need help** as the title and **Here What You Should Do?** as text. Also, this dialog alert contains an **OK** button. Finally, we create the `AlertDialog` instance and display it using the `show()` method.

As shown by our little example in the *How to do it...* section, the `onCreate()` method is the best place to set up the behavior of your graphical interface or other operations that need to be done at the startup of the activity.

The `Bundle` parameter passed to the `onCreate()` method is a data structure that contains the last known state of the activity, which is the same as the variable values. In other words, a not `null` bundle means that the application is restarted (using the `onRestart()` method) and should therefore restore the previous data. This mechanism is not simple to reproduce on the virtual device because the activity has to be killed by the OS to be triggered. Indeed, if we press the **Home** button, the activity will be stopped, and in case we navigate back to the application, the `onCreate()` method will not be called; we will resume the activity using the `onRestart()` and `onStart()` methods. However, the following code sample shows how to use the bundle structure to recover data:

```
if (bundle != null) {
  aTextView = bundle.GetString ("aString");
}
```

Just like the `GetString()` method, the `Bundle` class offers 40 other `Get` methods. Each `Get` method will work for a specific data type, therefore, we have the `GetBoolean()`, `GetCharArray()`, or `GetByte()` methods returning a Boolean, char [], and byte, respectively.

Android will automatically call the `OnStart()` method right after the completion of the `OnCreate()` method.

OnStart: In the `OnCreate()` method code sample, we created a reference to a graphical element, which is a `TextView`, and we set the `Text` variable to "`Hello !`". While we stated that the `OnCreate()` method is the best one to set up our graphical element, the `OnStart()` method can be a good choice too. However, this method should be used in priority to refresh the value of the graphical interface like we did here with the `TextView` text. As a concrete example, in the step counter, which is a prebuilt application provided by Xamarin (`https://xamarin.com/prebuilt/step-counter`) to simply count the steps made by the user, the `OnStart()` method will be used to refresh the bars and curves that display the steps according to the *How to do it...* section, when they were made while the application wasn't displayed.

The `OnStart()` method will most likely not be overridden. Indeed, as seen before, this method is called automatically after the `OnStart()` method and the `OnRestart()` method. Moreover, this method will automatically call the `OnResume()` method. Therefore, if an application's operation is to be carried out, in general use, we will add our specific statements in the `OnStart()` or `OnResume()` overrides. The only case in which it can be pertinent to add behavior to the base `OnStart()` method is for refreshing the values right before the activity becomes visible to the user. The following code sample as seen in the *How to do it...* section, exposes the `OnStart()` prototype and changes the text of a `TextView` element. In methods that handle states, it is important that you always call the base method first. If you don't, the application will throw an exception.

OnResume: The code of the `OnResume()` method shows the modification of the address of a `TextView` element with the address of the Bluetooth device embedded in the phone (we will come back to this later in the book when we talk about interaction with hardware). The `OnResume()` method is called right after the `OnStart()` method is completed. Just like the `OnStart()` method, this method should be used to refresh the value of the graphical interface according to the events that might have taken place while the application wasn't displayed.

As soon as the activity is ready to interact with the end user (that is, as soon as the activity is visible), the `OnResume()` method is called. The `OnResume()` method is also called right after the `OnRestart()` method, which intervenes with a *Stopped* application, and the `OnPause()` method, which occurs after an activity is paused. With its strategic position in the stack call, the `OnResume()` method is the best one in which to start playing animations and accessing devices (NFC, Camera, GPS, and so on).

There is an important point to keep in mind about the `OnResume()` method. The activity is ready to interact with the end user when this method is called, however, this does not mean that the user can interact with the activity. In fact, the activity can be behind some keyguard menus (such as password, resume games, and so on).

OnPause: The code sample of the `OnPause()` method shows the prototype of the `OnPause()` method. The `OnPause()` method can be seen as the opposite of the `OnResume()` method. While the `OnResume()` method will be called when the application is called back, the `OnPause()` method will be called every time the application is set in the background. An example of what can be done in this method includes the use of the following: `Save data to persistent storage`, `Desallocate big objects or free Hardware like Bluetooth`.

As we would see in the next figure exposing all activities, the `OnPause()` method is called when the activity is still visible but partially obstructed by another activity. The `OnPause()` method gives us an opportunity—that you must take in order to keep the whole system stable—to release hardware, such as the Bluetooth device, or to save data to persistent storage for accessing it later through the bundle mechanisms seen in the `OnCreate()` method. It's also the perfect moment to destroy objects that consume a lot of resources—thus avoiding getting killed by the OS.

OnStop: The `OnStop()` method will be called if the activity isn't visible to the user anymore. In other words, this method will be executed if the user presses the **home** or **back** button. A classic use of the `OnStop` activity is to undo our application from services or even stop services. We will see how to create and interact with services in *Chapter 6*, *Populating Your GUI with Data* and *Chapter 7*, *Using Android Services*. Nevertheless, the `OnStop()` method should be used to deallocate and free any resources that consume a lot of computational power/battery or resources that may be useful to other applications on the device.

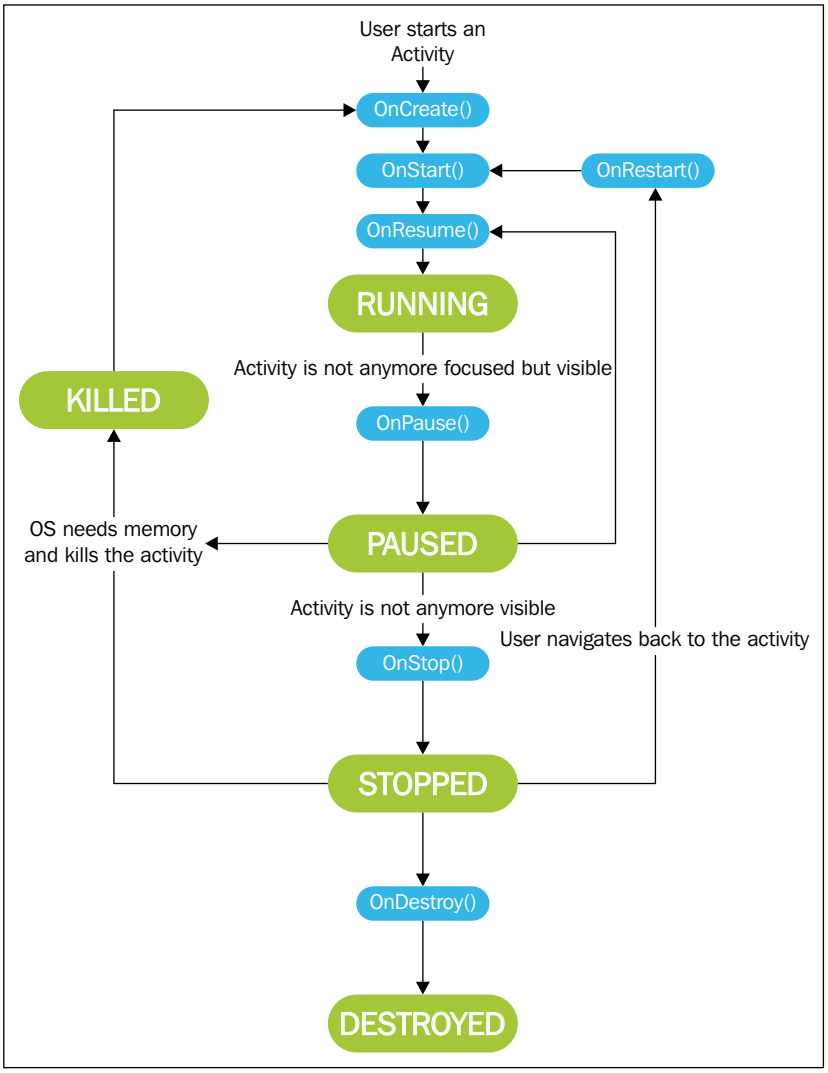
In a very critical memory situation, the `OnStop()` method can be bypassed by the OS, leading to a direct call of the `OnDestroy()` method after the `OnPause()` method. Therefore, it's a terrible idea to add some of your business logic in here. Moreover, the `OnPause()` method will be called, no matter how low the memory may be; thus, your de-allocation and hardware release must be done in the `OnPause()` method.

OnRestart: The `OnRestart()` method is not likely to be overridden as the `OnCreate()` method will be called right after this one, and we initialize everything on the `OnCreate()` method. Consequently, there is no typical use of the `OnRestart()` method nor best practices for it.

The `OnRestart()` method is called when the user browses back to an activity that was previously *Stopped* and the `OnStop()` method was executed. Because the `OnStart()` method will always be called right after the `OnRestart()` method, we initialize most of the data needed in the `OnStart()` method. A call to the `OnRestart()` method means that the application has not been killed by the OS, there is no typical statement for the `OnRestart()` method. Actually, this method will only be overridden in some exceptional cases that are beyond the scope of this book.

OnDestroy: Similar to the `OnStop()` method, the `OnDestroy()` method must be used for freed resources and the stopped services/threads that your application has launched. If the `OnStop()` method is overridden, then you should override the `OnDestroy()` method, and both the methods should have the same behavior. This is due to the fact that the Android operating system, when in need of memory, can kill your application by directly invoking the `OnDestroy()` method and skipping the `OnStop()` method. In the majority of cases, you'll find the same code duplicated on both methods. You can see an example of such duplication in the Step Counter prebuilt application from Xamarin, which we spoke about while presenting the `OnStart()` method. However, using the previous code sample of the *How to do it...* section, we can ensure that the code is only run once. This code will check whether the application has been through the `OnStop()` method, which has to set the Boolean `isStopped` to `true`—and execute the code once. However, the duplication is still present. Nevertheless, you can extract the logic of closing your application to a private method that the `OnStop()` and `OnDestroy()` methods will call.

The `OnDestroy()` method is the end of the lifecycle. Therefore, no method will be invoked after the `OnDestroy()` method, and the activity will be completely removed from the memory. If the user browses back to the activity, then the entire process of building an activity has to be done again. The `OnDestroy()` method is executed after the `OnStop()` method and, as we saw earlier, there is a possibility—in desperate attempts by the OS to acquire memory—that the `OnStop()` method will be skipped. This is also true for the `OnDestroy()` method. However, unlike the `OnStop()` method, there is some business logic that can be well-suited to an `OnDestroy()` override. For example, your activities can start any background threads that you wish to continue even when the application is not visible (*Paused* or *Stopped*). In such a configuration, the `OnDestroy()` method is the right place to kill these processes.



There's more...

Events beyond the life cycle methods

Using the API discovery and Intellisense from Xamarin Studio or Xamarin Plugin for Visual Studio, you will be able to discover more than 50 other methods to handle events. These methods range from `OnMenuSelected()` to `OnWindowsDetached()`. We will use some of them later in this book, more specifically while building advanced graphical interfaces.

See also

The complete list of Android events at `http://developer.android.com/guide/topics/ui/ui-events.html`.

Going through state-saving management

If you paid attention to the `OnCreate()` method, you certainly must have noticed that we can restore data from a data structure named `bundle` when activities are put in the forefront after being killed or destroyed by the OS. However, at this time, we have not yet provided the mechanism to be used in order to save this data inside the `Bundle` data structure. In this recipe, we will show you how to manage the saving/loading methods and properties of the bundle.

Getting ready

Just as the earlier recipes of this chapter, the only requirement is to have Xamarin Studio or the Xamarin plugin for Visual Studio up and running and a successful understanding of the concepts covered in previous recipes.

How to do it...

The first method we have to implement is the `OnSaveInstanceState()` method, which, as a reminder, will be triggered right before the death of an activity. Let's consider a simple application, composed of four `Text` labels and four `TextView` elements asking the user to enter their name, address, phone number, and e-mail address. Taking into account that filling in this kind of form is really a pain with a virtual keyboard, it will be a good thing to restore already typed information if the user has to quit the application for any reason earlier. The following screenshot presents the application we just described. The application that save preferences:

1. Add the following code sample to your `MainActivity` class (created in the previous recipe), in order to save the form's information into the bundle. As you can see, the Bundle accepts new entries in the form of key-value pairs. Therefore, we put four new keys `_email`, `_name`, `_address`, and `_phone` containing the values of the e-mail address, name, address, and phone number fields, respectively. As always, we also call the base implementation:

```
protected override void OnSaveInstanceState(Bundle bundle)
{
  bundle.PutString ("_email", FindViewById<TextView>
  (Resource.Id.mail).Text);
  bundle.PutString ("_name", FindViewById<TextView>
  (Resource.Id.name).Text);
  bundle.PutString ("_address", FindViewById<TextView>
  (Resource.Id.address).Text);
  bundle.PutString ("_phone", FindViewById<TextView>
  (Resource.Id.phone).Text);

  Log.Debug(GetType ().FullName, "OnSaveInstanceState
  Invoked");
  base.OnSaveInstanceState (bundle);
}
```

   The preceding code sample contains the logging method. The logging methods can be separated on four levels. By level priority, these levels are debug, info, warn, and error, and they have to be used with a tag for the log. Here, we choose the class full name and the log itself. The log will be printed in the Xamarin Studio or the Visual Studio consoles. We encourage you to write such statements in order to identify the method invocation without using the step-by-step debugger, thus speeding up your development.

2. Press the **Home** button of the virtual device, the following line appears in the Application Output Console:

```
[Mail.MainActivity] OnSaveInstanceState Invoked
```

   Meaning that the `OnSaveInstanceState()` method has been executed and therefore, the field's text has been saved into the Bundle.

3. Write the code enabling the restore of all our fields. For such a restore, the `OnCreate()` method throughout a conditional event linked to the Bundle's state could be the most suitable place to do it.

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);
  // Set our view from the "main" layout resource
  SetContentView (Resource.Layout.Main);
```

```
        if (bundle != null) {
          FindViewById<TextView> (Resource.Id.mail).Text =
          bundle.GetString ("_email");
          FindViewById<TextView> (Resource.Id.name).Text =
          bundle.GetString ("_name");
          FindViewById<TextView> (Resource.Id.address).Text =
          bundle.GetString ("_address");
          FindViewById<TextView> (Resource.Id.phone).Text =
          bundle.GetString ("_phone");
          Log.Debug(GetType ().FullName, "Bundle was not null.
          Restore complete.");
        }
      }
```

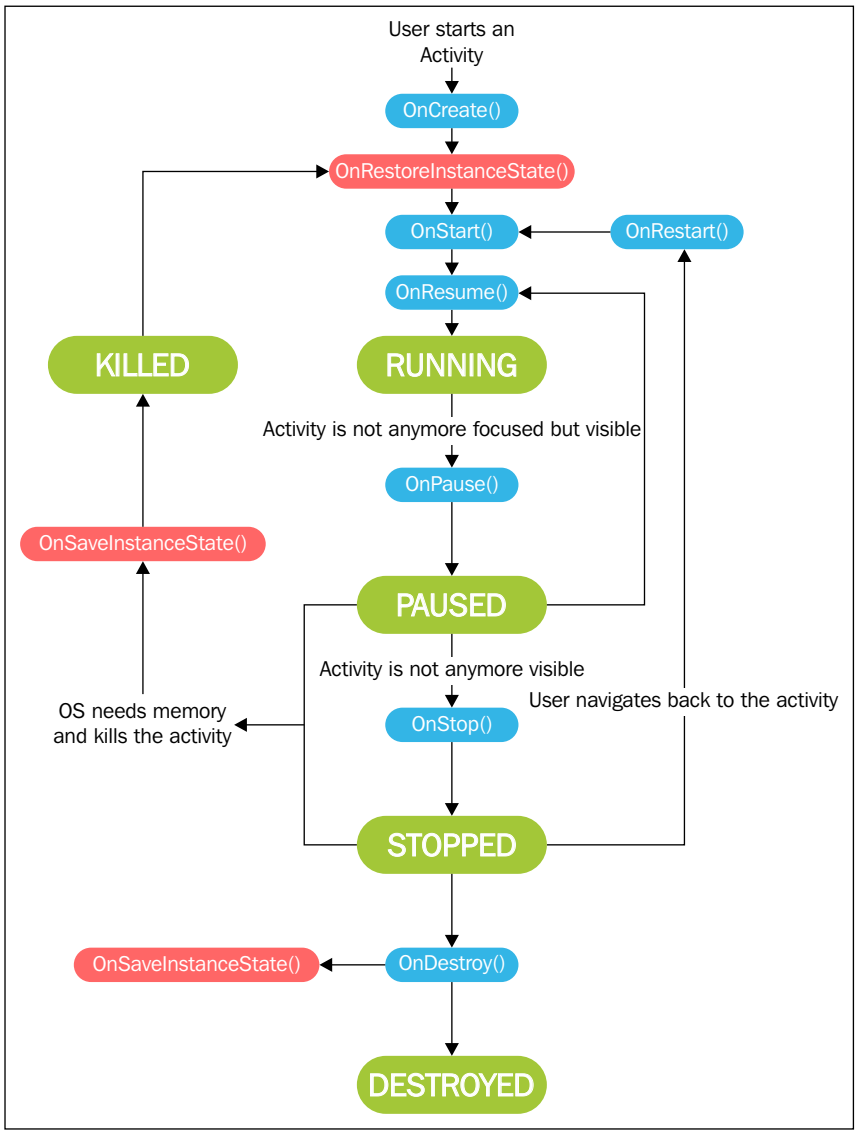
In the preceding code, we show how to get back our values from the Bundle. We will use the `Get()` method with the key in a parameter. Thus, we affect the field's text with their corresponding values. This code could also take place in an override of the `OnRestoreInstanceState()` method as shown by the following code:

```
protected override void OnRestoreInstanceState
(Bundle bundle) {
  // Restore your data here
}
```

How it works...

The management of the Bundle is handled by two new `On` methods. These methods are `OnSaveInstanceState()` and `OnRestoreInstanceState()`, which are invoked before the destruction of an activity and after the `OnCreate()` execution, respectively.

The following screenshot presents the state management process with these two new methods for saving a state.



As you can see, the `OnRestoreInstanceState()` method is called right after the end of the `OnCreate()` method, and the `OnSaveInstanceState()` method can be called at two different times. Indeed, the `OnSaveInstanceState()` method will be invoked during the planned destruction of an activity using the `OnDestroy()` method, and also when the OS kills your paused or stopped activities. Therefore, if these two methods are well overridden and correctly manage the save in/load from Bundle, they can restart like a charm even if the system kills your activities without any warnings.

There's more...

For a while, Android devices own two different buttons which seem to have the same behavior, these are the **Back** button and the **Home** button.

The difference between these two buttons is, in reality, quite simple. The **Home** button tells the OS that the user needs to use another application, while using the **Return** button closes the application, meaning that the user is done with it. Therefore, the **Home** button will put activities in the background, that is, in the *Stopped* state, and the **Return** button will activate the destruction of activities by the system.

See also

Refer to *Chapter 4*, *Using Android Resources*, to learn how to prevent activity being stored by the state management mechanism.

3
Building a GUI

In the previous chapters, you learned some fundamentals about Xamarin and its underlying platform. Then, we deepened our understanding of activities life cycles in order to deliver applications free of workflow problems that could lead to instability for both the application and the phone. The next logical step is to use this knowledge to provide a nice graphical interface for our users. Therefore, in this third chapter, you will learn how to take advantage of the built-in GUI builder of Xamarin Studio. We will cover the following topics:

- The multiscreen applications
- Manipulating basic form elements
- Handling rotations
- Choosing the right layout
- Customizing components

Introduction

This chapter is our very first hands-on chapter. As we have deployed a simple, text-only, "Hello World" application, our users expect more, much more. Here comes a really handy graphical user interface builder of Xamarin. Moreover, we only built one screen application, and if you want to build complex applications providing a wide range of services, you will need multiple screens. As we saw in *Chapter 1*, *Getting Started* and *Chapter 2*, *Mastering the Life and Death of Android Apps*, this chapter will describe an Android application architecture and all its components, such as activities, services, and intents, which are very loosely coupled to each other. This particularity offers a special challenge for a multiscreen application.

The multiscreen application

Our goal in this subsection is to have two screens and two activities. The first activity will contain a simple button to access the second activity, while the second activity will have a button to return to the first activity.

Getting ready

In order to follow this recipe, you must have Xamarin Studio or Visual Studio and running. We have created a new project dedicated to experiencing graphical interfaces. We have named this project `GraphicalInterfaces`.

How to do it...

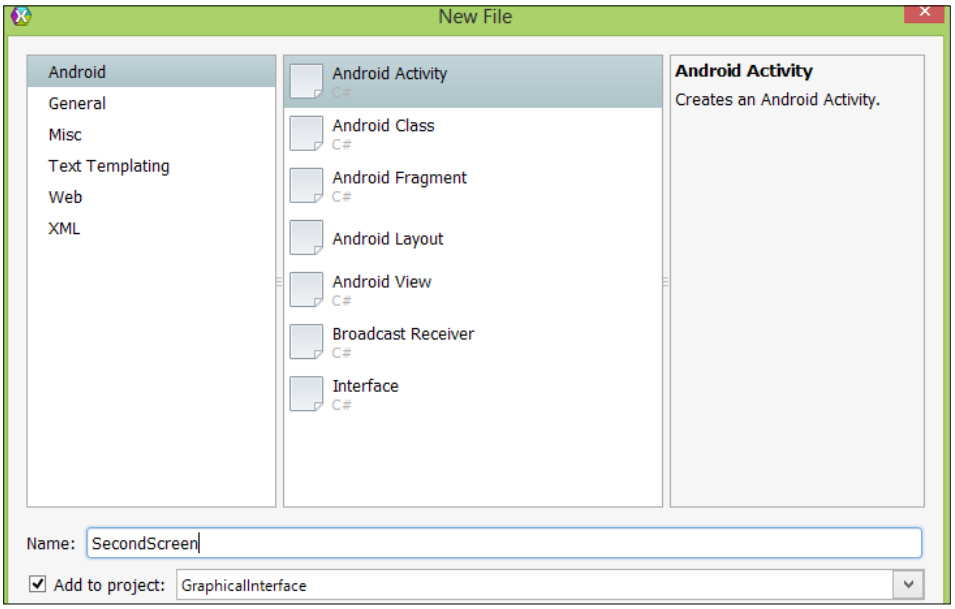
The first thing is to have a `GraphicalInterface` project created, as stated in the *Getting ready* section. A new project created with Xamarin will have a default activity and some code in it:

1. **Create a second activity**:

   The first step is to create the second activity that we wish to invoke from the first one. To do so, use the *Ctrl + N* shortcut, and create a new activity named `SecondScreen`, as shown in in the following figure. The wizard will create a new activity place in the `.cs` file. This new activity will have the following declaration:

```
[Activity (Label = "SecondScreen")]
  public class SecondScreen : Activity
```

As you can see, this one does not have the `MainLaucher` attribute set to `true`:



2. **Create an event handler invoking the StartActivity**:

   In the second step, we will create an adapted event handler to invoke the `StartActivity` method. The lines exposed by the following code samples shows how to create this specific event handler, and should be placed in the `MainActivity.cs` file in place of the default code:

```
button.Click += (sender, e) => {
  StartActivity(typeof(SecondScreen));
};

button.Click += delegate {
  button.Text = string.Format ("{0} clicks!", count++);
};
```

   The only thing to notice in these code samples is the `typeof()` method that returns the `SecondScreen` class—a type of `SecondScreen` class. As a reminder, the `StartActivity()` method takes a `Type` variable in the argument. At the end of this step, we can launch our application, press the on-screen button, and start the second activity. However, we will not have a way back to the first activity.

3. **Create a layout for the second activity**:

   Once again, use the *Ctrl* + *N* shortcut to create a new file. This time, choose `AndroidLayout` and name this file `SecondScreen`. The created file has the `.axml` extension and will be opened by Xamarin Studio right after its creation. Once you see the graphical interface of the second screen, drag and drop a button on it, as you did in *Chapter 1*, *Getting Started*. Then add the following code in the `SecondScreen.cs` file:

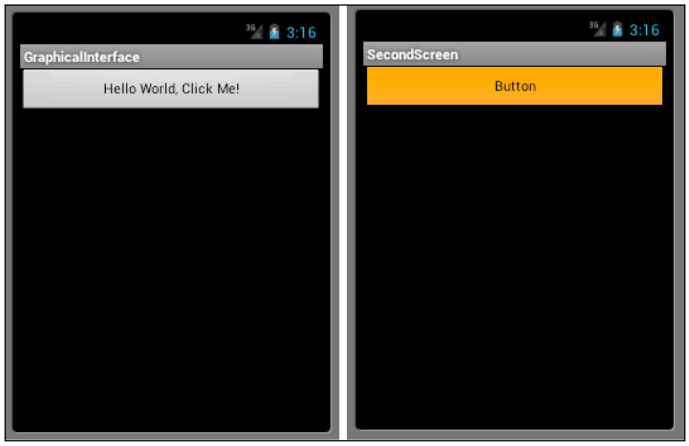
```
Button button = FindViewById<Button>
(Resource.Id.mySecondButton);

button.Click += (sender, e) => {
  StartActivity(typeof(MainActivity));
};
```

A graphic element is accessible by all the activities/classes of your project. Therefore, pay attention to your variable names. For example, in the second activity, if you refer to the first activity's button, then your code will be compiled and deployed. However, you will a have null pointer exception while you try to instantiate the second activity. A lot of bugs of this kind will appear during the development of a mobile application. Therefore, we can only insist on the necessity of testing your code as often as possible on real devices.

Don't hesitate to use the powerful intellisense of Xamarin. To activate it, press *Ctrl* + Spacebar. For example, with the preceding code sample, using the intellisense right after typing the `+=` variable will result in a set of proposition here the `Event Handler` prototype is present. Select it and watch the code autocomplete for you.

4. **Test it!**:

   Compile and deploy the application on the emulator. Press the buttons and watch the screens (activities) appear and disappear, shown as follows:



How it works...

You may think of Android activities communicating with each other as web pages on a web server. Each web page is an activity and may contain links to other activities hosted on the same webserver. Therefore, we have to programmatically create this link between activities. The method's prototype for doing so is shown here:

```
StartActivity (Type type)
```

For instance, the `StartActivity()` method, which takes the type of the other activity that we wish to run, is the equivalent of our HTML hyperlinks between webpages. To fit our objectives for this subsection, have buttons for switching between activities. We will need to call the `StartActivity()` method inside a new event handler. This newly created event handler will be crafted from the button reference. An event handler has to follow the following form:

```
button.Click += (sender, e) => {};
```

Analyzing this code leads to a simple conclusion, and a behavior place between the { } tag will be added using the `+=` variable to the current behavior that was triggered while clicking the button. This newly added behavior has a `Sender` argument and the event arguments' `a`.

To sum up, we will have an event handler placed on a button on the first activity. This event handler will invoke the `StartActivity()` method. The latest thing to take care of while tackling multiscreen applications is the `MainLauncher` argument in the activity declaration. As shown by the following code sample, our main activity has this argument set to `true`, meaning that our application will start from here:

```
[Activity (Label = "GraphicalInterface", MainLauncher = true)]
  public class MainActivity : Activity
```

There's more...

Communicating data between activities

We manage to create a multi-activity application that offers a multiscreen. It's quite appreciable, and your users will probably enjoy it. However, as a programmer, how to communicate data between these activities should bother you. Let's play with the dual screen application that asks the name of the user on the first screen and displays it on the second one:

1. Add the `TextField` and `PersonName` fields to the `MainActivity` class, and add `TextField` in the second activity.
2. Have a reference to the newly created activity.

In our first event handler, we just launched the second activity. However, we can keep a reference to it by using the following code:

```
button.Click  += (sender, e) => {
  var second = new Intent(this, typeof(SecondScreen));
  StartActivity(second);
};
```

Intents are a high-level abstractions of an operation that is to be executed. And we can use it, as exposed here, with the `StartActivity()` method to create new activities:

1. **Pass and retrieve the data**:

   Intents have, similar to the Bundle mechanism shown in *Chapter 2*, *Mastering the Life and Death of Android Apps*, access to a key-value map. We can populate this map with the `PutExtra()` method and access its values with the `GetTypeExtra()` methods. Here's how we can do so:

   ```
   //Pushing data
   second.PutExtra("name", FindViewById<TextView>
   (Resource.Id.name).Text);

   // Retrieving data
   FindViewById<TextView> (Resource.Id.nameAnwser).Text =
   Intent.GetStringExtra ("name") ?? "Error occurs";
   ```

   The pushing phase is very simple; we just add a new entry to the map. The retrieval is also simple, but it may require some explanation. As you can see, we use the `GetStringExtra()` method, meaning that we expect a `String` value at this key. Also, you can retrieve different types using `GetXExtra()` method and replace `X` by the a type (that is, `GetLongExtra()` method). We can even pass arrays. Also, we use the C# null-coalescing (`??`) operator. If you encounter this on this operator for the first time, its only purpose is to define a default value for values that may be `null`. In our case, if no data is passed, **Error occurs** will be printed, preventing a null pointer exception and an application crash.

2. **Test it**:

   Once you are done with the required change of adding the different fields and pushing and retrieving data on the first and second screens, compile and deploy your application:

The inheritance mechanism, also referred to as specialization, is a strong object-oriented mechanism that establishes the `Is-a` relationships. Classes inherit the attributes, methods, and behaviors of classes they specialize in. Outcome classes are referred to as derived classes or subclasses. Moreover, subclasses can add new behaviors to the inherited ones and even override or redefine inherited behaviors. While playing around with activities lifestyles we redefine the base behavior contained in the `Activity` class.

By means of inheritance mechanisms, we can specialize and customize the behavior of these seven methods to fit our needs while implementing a new activity. If we pay attention to our very first and autogenerated statement of the `MainActivity` class, we notice that the `OnCreate()` method has been overridden. In this override, the new behavior is added after a call to the `base` method:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);
  […]
}
```

A call to the base will execute the behavior of the previous superclass. In our case, the `Activity` superclass contains mandatory code to handle states. We are only agreeing with this code with specific statements to fit our needs.

See also

See also the next recipe to learn how to use the multiscreen application populated with form elements.

Using form elements

For sure, you will need most of the form elements that Xamarin Studio provides. Form elements will give you a very easy and quick way to get a request and retrieve data from users—as we have done several times with the `TextView` elements.

Getting ready

Create a new Android project named `GUIForm`.

How to do it...

1. **CheckBox**:

   A checkbox is either checked or not, and it contains an associated label describing the choice that the user is about to make:

   ```
   CheckBox cb = FindViewById<CheckBox>
   (Resource.Id.checkBox1);
   ```

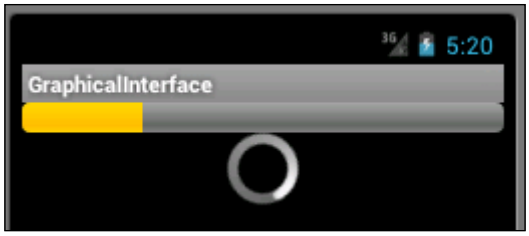
2. Obviously, checkboxes support the method in order to know whether it's checked or not and to change the associated text. Respectively, these methods are as follows:

   ```
   cb.Text = "My Describing Text";
   Boolean isMyCheckBoxChecked = cb.Checked;
   ```

   As other elements contain text, checkboxes also come with a set of methods regarding the size, color, and event—`TextChanged`, `TextColor`, and `TextChanged`, respectively.

3. **ProgressBar**:

   There are two types of ProgressBar; the horizontal one that progresses classically until it reaches 100%, and the circular one that keeps turning around until you cut it. The following figure shows both types on top of each other. The horizontal one is at the top and the declaring code is exposed by the following code sample:




```
ProgressBar pb = FindViewById<ProgressBar>
(Resource.Id.progressBar1);
```

4. The code for creating a reference to ProgressBars is the same for both types of this particular form element. How can the Android platform display it differently? If you look inside the `.axml` file corresponding to our activity on "source mode", you will see the following lines:

```
<ProgressBar
  style="?android:attr/progressBarStyleHorizontal"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
android:id="@+id/progressBar1" />
```

The source mode refers to the view where you actually see the XML code, which is behind the graphical interface. You can switch to this mode in the bottom-right corner of the window.

5. As you will have certainly noticed, there is a style property that can be filled by four different values:

```
?android:attr/progressBarStyleHorizontal
?android:attr/textAppearanceLarge
?android:attr/textAppearanceNormal
?android:attr/textAppearanceSmall
```

   The first one corresponds to the horizontal one, whereas the last three define the different sizes of the circular one. Finally, you can set the value of a progress bar using the `.Progress()` method, as shown in the following code sample:

```
pb.Progress = 25;
```

6. **RadioButton and Groups**:

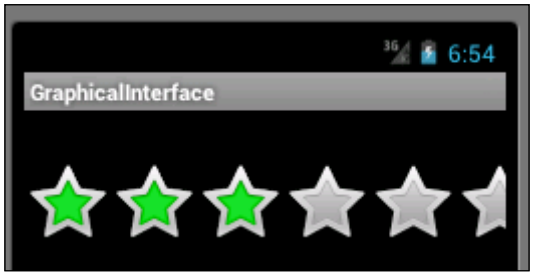
   RadioButton is basically the same as Checkbox, expects that, as same as for HTML, only on radio button of a group can be selected at once. We can create groups of RadioButtons, also known as RadioGroups, where we can pick only one option. Every single RadioButton, as for CheckBox, owns a method to know whether it's checked. Moreover, the RadioGroup also provides a method that returns the ID of the checked RadioButton. The four methods create a reference to RadioButton and RadioGroups and retrieve the checked values, as exposed here:

   ```
   //RADIO BUTTON
   RadioButton rb = FindViewById (Resource.Id.radioButton1);
   Boolean isChecked = rb.Checked;
   //RADIOGROUP
   RadioGroup rg = FindViewById (Resource.Id.radioGroup1);
   int selectedId = rg.CheckedRadioButtonId;
   ```

   Once again, the trick behind the RadioGroup cannot be seen in the classical and graphical view. If we take a look at the XML behind the scenes, we will be able to see that the RadioGroup is just another form of markup containing a set of RadioButtons.

7. **RatingBars**:

   With RatingBars, we have a very curious and useful component. It's simply a form element composed of stars that customers use to rate whatever you have for them to rate:

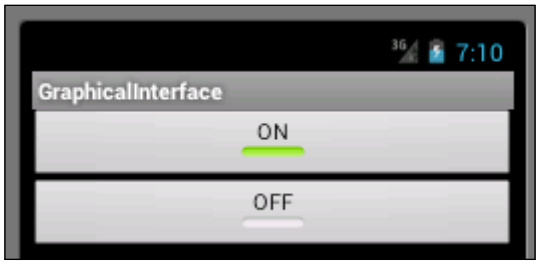
   As we can logically expect, RatingBar has a method to obtain the number of stars that have been checked:

   ```
   RatingBar rb = FindViewById (Resource.Id.ratingBar1);
   int nbStrars = rb.NumStars;
   ```

8. **ToggleButton**:

   A ToggleButton is similar to an interrupter. It has only two possible values: on or off. While the outcomes of the `.Checked()` method will always be `true` or `false`, the text, however, can be modified to fit your needs using the `TextOn()` and `TextOff()` methods:

```
ToggleButton tb = FindViewById<ToggleButton>
(Resource.Id.toggleButton1);
Boolean isCheked = tb.Checked;
//Modify the text
tb.TextOn = "Some Option:On";
tb.TextOff = "Some Option:Off";
```



9. **AutoCompleteTextView**:

   There is one particular element that comes in handy for selecting states or countries. Indeed, this component is parametrizable to display preset values according to the user entries. In the following example, we will create `AutoCompleteTextView`, which provides an autocomplete on Canadian states.

   1. Drag and drop an AutoComplete element onto your GUI.
   2. Instantiate a static array of strings in the `MainActivie.cs` file, shown as follows:

```
static string[] CANADIANS_STATES = new string[] {
  "British Columbia", "Alberta", "Saskatchewan",
  "Manitoba",
  "Ontario", "Quebec", "New Brunswick", "Prince Edward
  Island","Nova Scotia", "Newfoundland and Labrador",
"Yukon", "Northwest Territories",Nunavut"};
```

3. Create a `states_list.xml` file under the `Resources/Layout` folder:

```
<TextView xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:padding="10dp"
  android:textSize="16sp"
  android:textColor="#000">
</TextView>
```

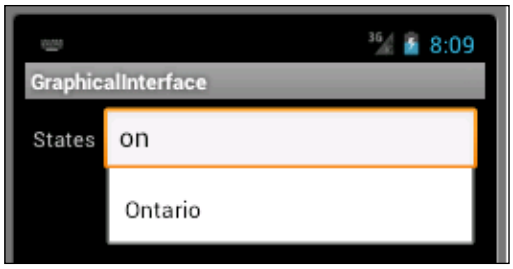
10. **Declare an ArrayAdapter**:

    An ArrayAdapter will be used to associate the string in the `CANADIANS_STATES` parameter to user entries in order to propose his/her matching cases:

```
AutoCompleteTextView actv =
FindViewById<AutoCompleteTextView>
(Resource.Id.autoCompleteTextView1);
var adapter = new ArrayAdapter<String>
(this, Resource.Layout.states_list, CANADIANS_STATES);
actv.Adapter = adapter;
```

An array adapter, just as `FindViewById`, is generic, meaning that you can use Autocompletion and ArrayAdapter for any type of data.

11. Build and deploy your code. You should have the following:



How it works...

There is not much to understand about form elements. Basically, we will always use the same method; to get a reference. We will always get a reference to the form element of interest by using the `FindViewById` generic method and after this, by using the specific methods of each element:

```
RadioGroup rg = FindViewById<RadioGroup>
(Resource.Id.radioGroup1);
```

As we already saw, we will pass buttons and TextView through other common form elements, such as CheckBox, ProgressBar, RadioButton and Group, RatingBar, ToggleButton, and finally the AutoCompleteTextView. For each of these form elements, we will give the code used to create the reference and some insight on a few useful methods associated with this element.

There's more...

In this recipe, we did not cover all the form elements you may want to use. However, you now have the basis to use the rest of them: Spinner, Date Picker, Chronometer, and so on.

See also

See also *Chapter 6*, *Populating your GUI with Data*, to populate data inside your form elements, and *Chapter 5*, *Using On-Phone Data*, for other graphical elements.

Using rotation

Modern mobile devices are basically meant to be rotated. There is actually two concrete and fairly different ways to handle it. Indeed, we can rotate the graphical interface of our activities because the user has rotated his/her phone, or we can do it programmatically to fit our needs. Thus, we will see both ways of handling rotation.

Getting ready

Once again, we will reuse the code from the `AutoCompletTextView` project.

How to do it...

1. **Layout oriented**:

   If you are only using built-in elements, they will all have a default look and feel for the landscape mode. Therefore, if you rotate the example used in the previous `AutoCompleteTextView` example, you will see something looking like the following screenshot:

If you only have access to an Android emulator for developing your applications, you cannot physically rotate it. However, by using the *Ctrl + F11* keys, you can force it to rotate.

The thing is that you can create a new folder named `Layout-land` under the `Resource` folder and create another `.axml` file inside it. To summarize, you will have two different `Main.axml` files for `MainActivity.axml`. One will be under the `Layout` folder and the other will be under the `Layout-land` folder. When the phone is in the portrait mode, the `.axml` file under layout will be used, whereas the one under the `Layout-land` folder will be used when the phone is rotated.

Copy the corresponding `.axml` file under the `Layout-land` folder that you just created under the `Resource` folder, and modify something in it similar to the text value with something remarkable, such as a `land` prefix:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:padding="5dp"
  android:id="@+id/linearLayout1">
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="States-land"
  android:id="@+id/textView1"/>
  <AutoCompleteTextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/autoCompleteTextView1"
  android:layout_marginLeft="5dp" />
</LinearLayout>
```

When rotating your phone or your emulator, you will see the text change, as shown in the following screenshot:

2. **Programmatically oriented**:

   Clearly, there is a simple way to get things done programmatically. First of all, you can test the current orientation using the following code:

```
TextView tv = FindViewById<TextView>
(Resource.Id.textView1);

var phoneMode = WindowManager.DefaultDisplay.Rotation;
if (phoneMode  == SurfaceOrientation.Rotation0) {
  tv.Text = "Not in Landscape";
}
else if (phoneMode == SurfaceOrientation.Rotation180) {
  tv.Text = "In Landscape";
}
```

   You can even force the phone into landscape (or the portrait mode) by using the `phoneMode` variable with the desired variable as follows:

```
surfaceOrientation = SurfaceOrientation.Rotation180;
```

There's more...

Do not restart my activities while rotating

Android is known to have the not-so-good habit of restarting activities when the phone gets rotated. In general manners, we do not want our activities to be restarted in order to save the state of our activities without saving it, making the activity look responsive even during rotation. The only way do this is to redefine the declaration of the activity with the following code:

```
[Activity (Label = "GraphicalInterface", MainLauncher = true,
ConfigurationChanges=Android.Content.PM.ConfigChanges.Orientation)
]
```

Once you have completed this, you must redefine the method: `OnConfigurationChanged()`:

```
public override void OnConfigurationChanged
(Android.Content.Res.Configuration newConfig) {
  base.OnConfigurationChanged (newConfig);

  TextView tv = FindViewById<TextView> (Resource.Id.textView1);
  if (newConfig.Orientation ==
  Android.Content.Res.Orientation.Portrait) {
    tv.Text = "Not in Landscape";
  }
  else if (newConfig.Orientation ==
  Android.Content.Res.Orientation.Landscape) {
```

```
        tv.Text = "In Landscape";
      }

    }
```

Notice that the way of detecting the orientation of the phone slightly differs in this redefinition. Also, as mentioned when studying Active Lifestyles, we must always call the base implementation in order to avoid fatal failures.

Saving states during rotation

As usual, we can save states during rotation using code that overrides the `OnSaveInstanceState()` method (shown in the following code sample) and calls the `bundle.GetInt ("someValue");` method to retrieve data:

```
protected override void OnSaveInstanceState (Bundle outState) {
  outState.PutInt ("SomeKey", SomeValue);
  base.OnSaveInstanceState (outState);
}
```

See also

See also the next recipe to choose a fit layout for your applications.

Adding layouts

There are four different and preprogrammed layouts that you can use in Xamarin Studio. Layout means a special ordering of elements inside the activity. These layouts are Relative, Linear, Table, and Tabbed Layouts. In this section, we will learn how to take advantage of each of them.

Getting ready

Create a new Android project named `Layout` for experiencing them.

How to do it...

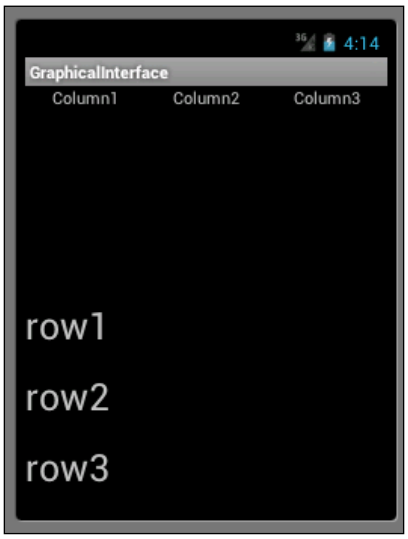
1. **LinearLayout**:

   Let's now look at how to add new elements in it to see how it looks. The next code sample shows three linear layouts in order to display three columns followed by three lines. The first linear layout contains the two other ones. It's the default one. After this, the second one owns `android: orientation="horizontal"` while the second one has the same argument, but is set to `vertical`, allowing us to create the following spatial organizations:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:minWidth="25px"
  android:minHeight="25px">
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1">
  <TextView
    android:text="Column1"
    android:gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_weight="1"/>
  <TextView
    android:text="Column2"
    android:gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_weight="1"/>
  <TextView
    android:text="Column3"
    android:gravity="center_horizontal"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_weight="1"/>
  </LinearLayout>
  <LinearLayout
```

```
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1">
  <TextView
    android:text="row1"
    android:textSize="15pt"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"/>
  <TextView
    android:text="row2"
    android:textSize="15pt"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"/>
  <TextView
    android:text="row3"
    android:textSize="15pt"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"/>
  </LinearLayout>
</LinearLayout>
```

The following screenshot shows the results of an activity displayed with the code that we just saw:

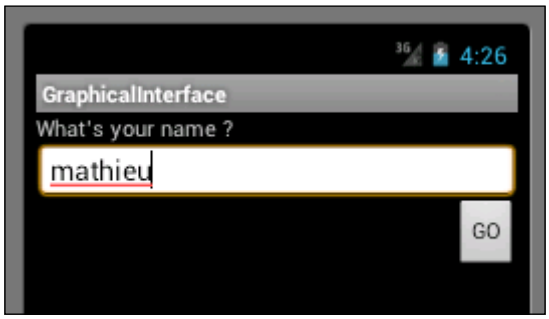


2. **RelativeLayout**:

   RelativeLayout, as its name suggests, ordinates elements in relative positions, meaning that for each element, you choose the alignment that they should take. For example, if we use `LinearLayout` containing `TextView`, followed by a button, the button will automatically get aligned to the left-hand side border of the screen. Using `RelativeLayout`, we can align the button to the right-hand side part of the element at the top of it:

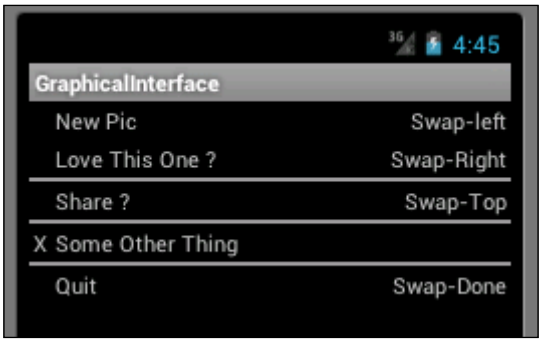
```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:id="@+id/textview1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="What's your name ?"/>
  <EditText
    android:id="@+id/editText1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background=
    "@android:drawable/editbox_background"
    android:layout_below="@id/textview1"/>
  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/editText1"
    android:layout_alignParentRight="true"
    android:layout_marginLeft="10dip"
  android:text="GO" />
</RelativeLayout>
```

The highlighted lines of the code show the `tdonateent` attributes that we can use to ordinate the element. We can use `android:layout_below=` to specify which element must be below which other one using the latter's ID. Moreover, we can use `android: layout_alignParentRight="true"` or `android: layout_alignParentLeft ="true"` to set up the alignment. The following screenshot shows how the activity is displayed:



3. **TableLayout**:

   This layout is heavily used by Android programmers. Indeed, it separates the activity displayed into a table row of a table just as the HTML table does. In fact, the skin of the separation is a clean separator for anything you have to differentiate in your graphical interface. Finally, inside a table row, elements own `android: layout_column`, allowing them to select in which column the element belongs. This creates a graphical interface as shown in the following screenshot, which has three different columns and five rows:



In addition to the `android: layout_column` attribute, we can see that an element without it will automatically be placed at the right-hand side if declared after an element owning this attribute. However, if it does not have the column attribute and is the first to be declared on the column row, then it will take its place in the first column.

The rows and columns numbering starts at 0. Therefore, `android:layout_column="1"` will place the element in the second column.

In the following code sample, we can also see the markup named `<View>`, containing a height and color. These elements are the white lines separating the first two lines from the others:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:stretchColumns="1">
  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="New Pic"
    android:padding="3dip" />
    <TextView
      android:text="Swap-left"
      android:gravity="right"
    android:padding="3dip" />
  </TableRow>
  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="Love This One ?"
    android:padding="3dip" />
    <TextView
      android:text="Swap-Right"
      android:gravity="right"
    android:padding="3dip" />
  </TableRow>

   <View
    android:layout_height="2dip"
  android:background="#FF909090" />

  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="Share ?"
    android:padding="3dip" />
```

```
    <TextView
      android:text="Swap-Top"
      android:gravity="right"
      android:padding="3dip" />
  </TableRow>

  <View
    android:layout_height="2dip"
  android:background="#FF909090" />

  <TableRow>
    <TextView
      android:text="X"
    android:padding="3dip" />
    <TextView
      android:text="Some Other Thing"
    android:padding="3dip" />
  </TableRow>

  <View
    android:layout_height="2dip"
    android:background="#FF909090" />

  <TableRow>
    <TextView
      android:layout_column="1"
      android:text="Quit"
    android:padding="3dip" />
    <TextView
      android:text="Swap-Done"
      android:gravity="right"
      android:padding="3dip" />
  </TableRow>
</TableLayout>
```

The preceding code sample shows the entire table layout code.

4. **TabbedLayout**:

   The last layout that we want you to keep in mind is the most complicated one, but it is also one of the most used. It's a layout enabling you to have a tab in your application. TabbedLayout is not as intuitive as the three other layouts we have seen so far. Indeed, the declaration of this layout is made by a `<TabHost>` markup, defining your activity as Tab capable. Then, you will have a LinearLayout containing two mandatory elements: `TabWidget` and `FrameLayout`. At last, your code will contain the markups of the following code sample. Also, the ID of `TabHost`, `TabWidget`, and `FrameLayout` must have the values of `@android:id/tabhost`, `@android:id/tabs`, and `@android:id/tabcontent`, respectively:

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:id="@android:id/tabhost">
  <LinearLayout>
    <TabWidget
      android:id="@android:id/tabs"/>
    <FrameLayout
      android:id="@android:id/tabcontent"/>
  </LinearLayout>
</TabHost>
```

5. If you deploy this activity on your emulator, you will be pretty disappointed, as it does nothing except display a black screen. Indeed, since we just defined the possibility of using tabs, we will now have to implement them. The first thing is to modify the declaration of your `MainActivity` class inside the `MainActivity.cs` file, as shown in the following code sample, where the `MainActivity` class is now a subtype of `TabActivity` instead of `Activity`:

```
-[Activity (Label = "Layout", MainLauncher = true)]
-  public class MainActivity : Activity

+[Activity (Label = "Layout", MainLauncher = true)]
+  public class MainActivity  : TabActivity
```

6. Each tab is a separate activity in reality. Therefore, we have to create as many new activities as the desired amount of tabs. Moreover, the tabs are identified by a logo with some text, and we have to specify the logo we want to use. We will create a two-tabbed layout with a megaphone and mic.

7. Since each tab is a separate activity, we have to create them. Create two activities—a mic and megaphone that contains the following code:

```
protected override void OnCreate (Bundle bundle) {
  TextView textview = new TextView (this);
  textview.Text = "Mic tab";
  SetContentView (textview);
}
```

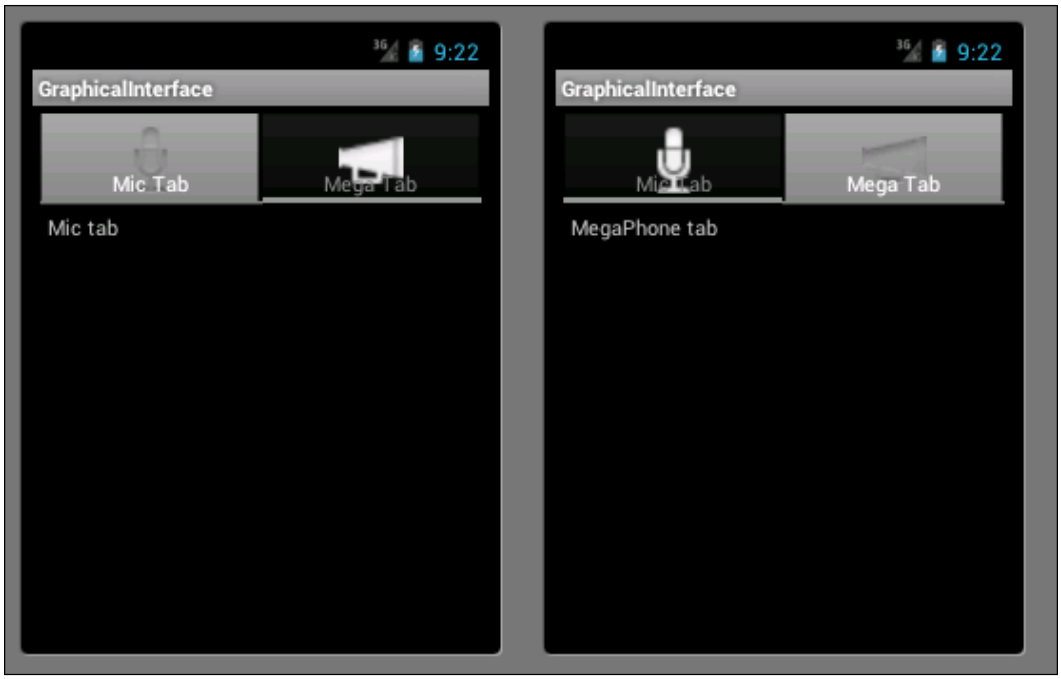
8. The corresponding `.axml` files, `mic.axml` and `megaphone.axml`, contain the reference to the icon. Note that icons must be placed under the `drawable` folder and referred to without the extension (`.png`, `.jpg`,...):

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android=
"http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/micOn"
    android:state_selected="true"/>
  <item android:drawable="@drawable/micOff"/>
</selector>
```

9. Finally, you can programmatically create these tabs in the `MainActivity` class using the following code sample. It first creates an Intent using the type of the Mic Activity, just as we did earlier between activities in this chapter. Then, it asks for a new Tab and fills this tab with appropriate properties, such as the layout or title. You will also have to do this for the second tab—the megaphone:

```
var intent = new Intent(this, typeof(Mic));
intent.AddFlags(ActivityFlags.NewTask);

var spec = TabHost.NewTabSpec("mic");
var drawable = Resources.GetDrawable(Resource.Layout.Mic);
spec.SetIndicator("Mic Tab", drawable);
spec.SetContent(intent);

TabHost.AddTab(spec);
```

10. It's done! To summarize the TabbedLayout, you have to make your activity implements `TabActivity` instead of `Activity`, and then create as many activities as you need—one for each tab. Then, for each activity, specify a layout. Finally, instantiate these tabs in the main activity. The following screenshot shows the TabbedLayout in action:



How it works...

Layouts provide a special environment to work in. When you select a layout, your elements will be put on the screen according to the layout that you pick. Therefore, you have to know the advantages of each of them to make a sound choice. Despite their importance, it's fairly simple to understand what they do, and how to implement them. Indeed, to declare a specific layout for our applications, we use, as usual, the `.axml` file related to the targeted activity.

In order to identify which layout is currently used for your activity, you can open the source view of the `.axml` file, and check the second line. The following code shows the code for the LinearLayout, which is the default one:

```
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:minWidth="25px"
  android:minHeight="25px">
</LinearLayout>
```

If you create your elements manually, place them inside the `Layout` markup. Also, we can create many layouts in the same activity.

See also

See also the following recipe to customize the components inside your layouts.

Customizing components

Using Xamarin Studio for Android, you can customize how your components look.

Getting ready

Create a new Android project, `CustomizingComponents`.

How to do it...

As an example, we will perform the transformation of a button. First, create your three images corresponding to the button being pressed, focused, and normal, and then add them to the `Drawable` resources. Then, create a new layout named `customizedButton.axml` under the `Resource/Layout` folder. This file will look like the following code sample that defines the images corresponding to the three states:

```
<?xml version="1.0" encoding="utf-8"?>
<selector>
  <item android:drawable="@drawable/pressed_state"
    android:state_pressed="true" />
  <item android:drawable="@drawable/focused_state "
    android:state_focused="true" />
  <item android:drawable="@drawable/normal_state " />
</selector>
```

To use this new customize button in your activities, you just have to add the background attribute to an existing button. As a value, this background attribute should have the layout that we created for customized buttons. Hence, the **Go!** button that we used in RelativeLayout will now have the shape shown by the following code sample and will be placed in your activity `layout.axml` file:

```
<Button
  android:id="@+id/button1"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_below="@id/editText1"
  android:layout_alignParentRight="true"
  android:layout_marginLeft="10dip"
  android:background="@layout/customizedButton"
android:text="GO" />
```

How it works...

While hanging around with the tabbed layout, we saw that subactivities understand activities that are embedded in a tabbed one, and they own a `.xaml` file containing a `<selector>` tag instead of classical layout ones. Using the `selector` tag, you can, for example, override the way the button looks by creating a new `.xaml` file composed of `android:drawable` and `android:state` respectively, for the image of the button and the state in which this image applies.

There's more...

Customizing other components

Once again, we did not see how to customize every component discovered in the `Form` element. However, you now have the basis to customize them all.

See also

See also the next chapter to use new components capable of playing audio and video.

4
Using Android Resources

In this chapter, we will cover the following recipes:

- Creating a SplashScreen
- Using an icon for your application
- Playing a song
- Playing a movie

Introduction

Just like *Chapter 3*, *Building a GUI*, this chapter is a hands-on guide. Indeed, we will once again pass through many code samples and see their corresponding screenshots.

This chapter will enhance your graphical interfaces by letting you play with images, splash screen, and playing songs. You certainly know that most of the applications in the Android market do have these features. While offering the same basic functionalities that other apps provide can be a good enough reason to master these aspects of building graphical interfaces, there is another—and much better—reason: building your brand.

There is an incredible number of applications out here. According to `AppBrain.com` statistics, there are 877,743 applications in the Android market at the time of writing (December 2015) and 78% of them are likely to be useful, meaning that they offer good features and are downloaded by users. If you want your apps to be successful in this jungle, you'd better build a strong brand image for yourself. While we will talk about monetizing our application and pushing it to the market in the last two chapters of this book, the first step is to make your application recognizable by customers using a logo, a splash screen, and playing audio.

Creating a SplashScreen

The splash screen of your application is, without a doubt, one of the images you should work hard on. The splash screen will appear every time your application is launched and will be displayed while your application is loading in the background. Usually, starting a full-fledged application takes a couple of seconds and you would want to inform your users that the application is starting by displaying a good-looking splash screen instead of a black screen. By definition, splash screens cover the entire screen and welcome your customers.

Getting ready

In order to follow this recipe, you must have Xamarin Studio or Visual Studio with the Xamarin Studio up and running. Then, create a new project named `UsingRessources` for experiencing the recipes in this chapter.

How to do it...

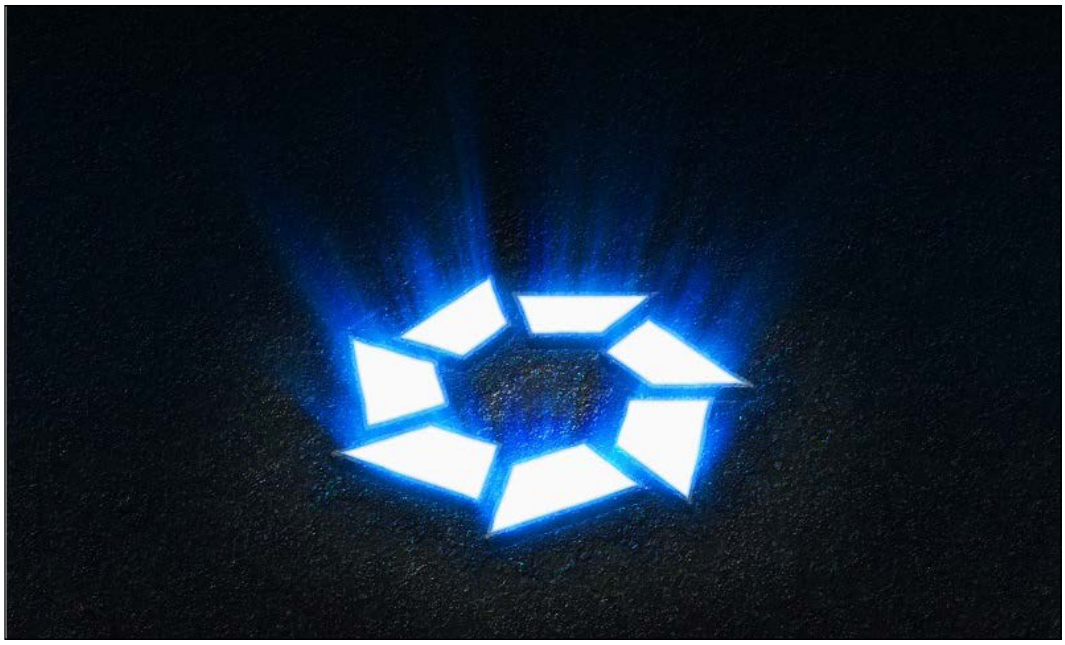
In order to display the splash screen, we will have to create an activity and a theme that we will apply to it. This can be done by following these steps:

1. Create a `mySplashScreen.Theme` instance in a file named `Style.xml` under the `Ressources/Values` folder.
2. The following code presents the content of the `Style.xml` file. As you can see, we have given a unique name to our style and defined two distinct items: `windowsBackround` and `WindowsNoTitle`. These two items specify the picture we want to display and an option to not display the activity title, respectively:

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
  <style name="mySplashScreen.Theme"
  parent="android:Theme">
    <item name="android:windowBackground">
    @drawable/mySplashPicture</item>
    <item name="android:windowNoTitle">true</item>
  </style>
</resources>
```

Add your splash screen image under the `Resources/drawable` folder.

We will use the following image, which is a work by **SinhalaTik** (`http://www.sinhalatik.blogspot.ca`) as our splash screen:



We have resized it to 800x480 which is a resolution that will be well accepted by most of the Android devices. Moreover, the Android engine will do its best to resize it well.

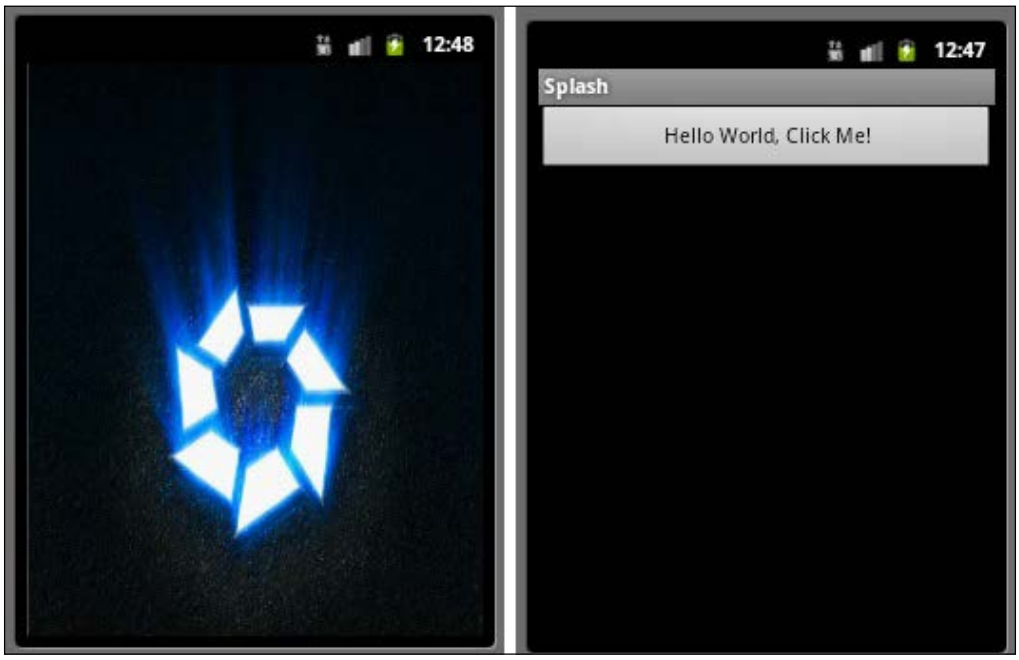
3. Finally, the third and last step consists of creating the SplashScreen activity, which has a reference to our theme:

```
[Activity(Theme = "@style/mySplashScreen.Theme",
MainLauncher = true, NoHistory = true)]
  public class SplashActivity : Activity {
    protected override void OnCreate (Bundle bundle) {
      base.OnCreate (bundle);
      // Some heavy business logic.
      StartActivity(typeof(MainActivity));
    }
  }
```

The `MainLauncher=true` option means that the current activity is the entry point of our application. Beside the `MainLaucher` instance, there are two new options: `Theme` and `NoHistory`. Obviously, the `Theme` parameter refers to the theme we just created for specifying the image that must be displayed as a splash screen. However, the `NoHistory` parameter deserves some explanation. An Android user can, using the back button, browse their activity history in the same way as the previous button on your Internet browser. Specifying `NoHistory = true` will prevent this activity from appearing in the history. Therefore, when users press the back button, they will not be able to see the splash screen again.

1. Remove the `MainLauncher` attribute from other activities.
2. Lastly, you should remove all other `MainLauncher=true` attributes from other activities. This way, users will always land on your Splash screen while starting your application.
3. Run your application on the emulator.

The following activity orchestration should appear:



As you can see in the screenshot, the splash screen activity appears first without the activity title. Then, when the business logic is completed, the second activity appears.

How it works...

Displaying a splash screen involves new kinds of object that are assignable to an activity: Android theme and style. We require these new concepts because we want the splash screen to be displayed even before the application starts loading. Indeed, there is no reason to use a splash screen if it only appears when the application is fully loaded. Using Android themes and styles will allow the Android engine to display whatever we define inside them, let's say a welcoming picture serving as a splash screen as soon as the user clicks on the application. Concretely, a style is a set of properties specifying the look and feel of a graphical element, such as TextView, and a theme refers to a style applied to an entire activity instead of a specific element.

Let's see what the style looks like and how to create and use styles. First, using a style will allow you to reuse the layout that you have defined in XML. For example, consider the following `TextView` layout from *Chapter 3*, *Building a GUI*:

```
<TextView
  android:text= "row1"
  android:textSize= "15pt"
  android:layout_width= "fill_parent"
  android:layout_height= "wrap_content"
  android:layout_weight= "1"
  android:text= "@string/row1"/>
```

This could be reduced to the following code snippet:

```
<TextView
  android:style= "@style/myStyle"
  android:text= "@string/row1"/>
```

This allows you to use only one style to set the look and feel of TextViews.

The concrete style file is another XML file which has to be created under the `Ressources/values` folder of your Android application. This file must have a `<resource>` XML root, which contains a set of `<item>` tags defining graphical properties. The following code shows the style file corresponding to the `TextView` element for *Chapter 3*, *Building a GUI*:.

```
<?xml version="1.0" encoding="UTF-8" ?>
<resources>
  <style name="myStyle" parent=
  "@android:style/TextAppearance.Medium">
    <item name="android:textSize">15pt</item>
    <item name="android:layout_width">fill_parent</item>
    <item name="android:layout_height">wrap_content</item>
    <item name="android:layout_weight">1</item>
  </style>
</resources>
```

As you can see, the style markup has a name attribute that must be unique and will identify this style. We will refer to this unique name to apply this style to an element with `@style/myStyle`, as shown earlier. Also, style can have a parent, meaning that style still supports inheritance in the same way as objects do. Here, the parent is a platform style named `TextAppearance.Medium`, but it also could be another style that you have previously defined. As for inheritance in object programming, the child style will have all the properties defined by the parent file, and we can refine them.

An XML file can contain many style definitions. You just have to specify the `<style></style>` tags one after the others.

The theme concept isn't physical as it is just referring to a style applied to the whole application or to an activity. Applying it is not rocket science. Indeed, you just have to specify it in the `AndroidManifest.xml` file by inserting the following tag for the whole application:

```
<application android:theme="@style/myStyle">
```

In case you only want the theme to be applied to one activity, the activity definition does have a `Theme` attribute, as shown by the following code:

```
[Activity (Theme = "@style/myStyle", Label = "Ressources",
MainLauncher = true)]
```

There's more...

Although Android performs extremely well resizing and scaling your images, you definitively should provide different image resolutions in order to adapt your applications to the screens they are displayed on. Several parameters should be taken into account. Some of them include the screen density, orientation, resolution, and **Density-Independent Pixel** (**DP**). While the orientation and resolution should sound familiar to you, the density and the density-independent pixel are trickier. The screen density refers to the number of pixels inside a given area. It also called **DPI** (**Dots Per Inch**). Using the Android platform, you have access to four different groups of density: low, medium, high, and extra high. The density-independent pixel is another concept that allows you to express a layout position or dimension in a density-independent manner. Therefore, you should use the density-independent pixels to build your layouts. The formula to convert from dp to pixel is *px= dp x (dpi / 160)*. Hence, in a 580 dpi screen, 1 dp equals 3 physical pixels. Once again, the screens are divided into four different categories: xlarge (960dp x 720xp), large (640dp x 480dp), normal (470dp x 320dp), and small (426dp x 320 dp). All of these values are on at least basis.

To take advantage of the screen categories proposed by the Android platform, you just have to create a different layout folder according to the categories you want to support. The folder must be named as follows:

- `Resources/layout/main_activity.xml width`
- `Resources/layout-sw600dp/main_activity`
- `Resources/layout-sw720dp/main_activity.xml`

In the preceding example, we have three different layouts. The first one will be picked for a screen smaller than 600 dp, the second one for screens ranging from 600 dp to 720 dp, and the last one for anything bigger than 720 dp. The `sw` attribute stands for smallest width. Instead of the `sw` attribute, you can use `w` for available width or `h` for available height.

See also

Refer to the complete Google guide to support multiscreen applications at `http://developer.android.com/guide/practices/screens_support.html` for more information about good practices.

Using an icon for your application

According to Google's Our Mobile Planet data (`http://services.google.com/fh/files/blogs/our_mobile_planet_us_en.pdf`), an average Android downloads 26 applications in addition to the already-installed ones. So, we can easily assume that your application will be hidden somewhere in the other 40-50 applications of your users. How can we catch the eyes on both the Android market and the applications' menus? This can be done using a recognizable icon. This recipe will show you how to change the default icon on Android.

Getting ready

For this recipe, we will continue to use the project created for the previous recipes.

How to do it...

Using the following steps, we will see how to use an icon in our application:

1. Find or create a logo representing your application/company.

   We will use the following icon:

2. Place your logo under the `Resources/drawable` folder.
3. Modify your `AndroidManifest.xml` file under `Properties` folder as follows:

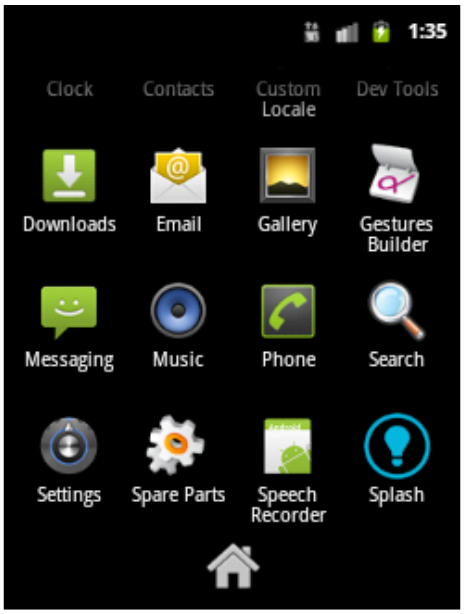
```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
"http://schemas.android.com/apk/res/android"
android:versionCode="1" android:versionName="1.0"
package="Splash.Splash">
  <uses-sdk/>
  <application android:icon="@drawable/icon_name"
  android:label="Splash"></application>
</manifest>
```

   All attributes but `android:icon` should already be in the file.

4. Deploy your application on the emulator again.

If you are confronted with an error involving "**not found resource in drawable**", try to clean/rebuild your project after checking that there is no misspelling in the icon's name.

The application will start as usual, however, if you browse back by pressing the **Home** button or the **Back** button, you will be able to access the application menu and see the new logo, as shown in the following screenshot:



You may have to restart your emulator to see the new logo as all files are not transferred each time. You may also try to uninstall the application on the phone.

How it works...

There is not much to understand about icons. Icons are images with a 48 x 48 pixel size. The default icon of your application will look like this:

This image is stored under the `Resources/drawable/Icon.png` folder. This picture is used both for the Android Play Store and the application menu on the phone.

There's more...

Here is some additional information recommended by Google.

Google Play Store icons

The required size of Google Play Store icons—the ones that the users will see while browsing through the market—is 512x512 px, due to the ability of users to browse through the market using a laptop and not only a phone. This logo will be demanded by Google while submitting your application to the market.

Keeping this in mind, I encourage you to design your icons using vectors capable software, such as **Abobe PhotoShop**.

Designing trick

In order to get a bigger visual impact, your icons should be a distinct three-dimensional silhouette. Also, your icons should have a little perspective, as if viewed from above. This way, users will be able to see depth in your icons and therefore, they will be highlighted compared to the background and other icons around.

See also

Iconography is a wide domain, you can find more on this at `http://developer.android.com/design/style/iconography.html`.

Playing a song

There are an infinite number of occasions in which you would want your applications to play sound. In this section, you will learn how to play a song from our application in a user-interactive way (by means of clicking on a button) or programmative way.

Getting ready

Once again, we will reuse the project we created in the first recipe of this chapter.

How to do it...

Now, we will see how to play a song :

1. Add the following `using` attribute to your `MainActivity.cs` file:

   ```
   using Android.Media;
   ```

2. Create a class variable named `_myPlayer` in the `MediaPlayer` class:

   ```
   MediaPlayer _myPlayer;
   ```

3. Create a subfolder named `raw` under the `Resources` folder.
4. Place the sound you wish to play inside the newly created folder.

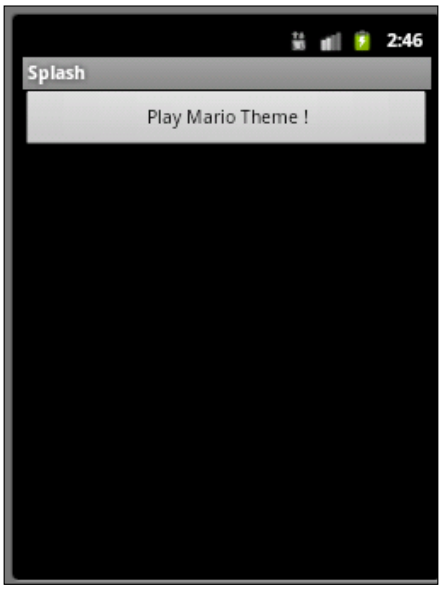
   We will use the Mario theme, free of rights for noncommercial use, downloaded from `http://mp3skull.com`.

5. Add the following lines at the end of the `OnCreate()` method of your `MainActivity` class:

   ```
   _myPlayer = MediaPlayer.Create (this, Resource.Raw.mario);

   Button button = FindViewById<Button>
   (Resource.Id.myButton);

   button.Click += delegate {
     _myPlayer.Start();
   };
   ```

   In the previous code sample, the first line creates an instance of the `MediaPlayer` using `this` as context and `Resource.Raw.mario` as the file to play with this `MediaPlayer`. The rest of the code is simple, we just acquired a reference to the button and created a behavior for the `OnClick` event of the button. In this event, we call the `Start()` method of the `_myPlayer()` variable.

6. Run your application and click on the button as shown on the following screenshot:



You should hear the Mario theme playing right after you clicked the button, and this will happen even if you running the application on the emulator.

How it works...

Playing sound (and video) is an activity handled by the `MediaPlayer` class of the Android platform. This class involves some serious implementations and a multitude of states in the same way as activities. However, as an Android applications developer (and not as an Android platform developer), we only require a little background on this.

The Android multimedia framework includes—through the `MediaPlayer` class—a support for playing a very large variety of media, such MP3s, from the filesystem or from the Internet. Also, you can only play a song on the current sound device, which can be the phone's speakers, headset, or even a Bluetooth-enabled speaker. In other words, even if there are many sound outputs available on the phone, the current default set by the user is the one where your sound will be played. However, you cannot play sound during a call.

There's more...

There are two ways to play sound, which are discussed in the following sections.

Playing sound that is not stored locally

You may want to play sounds that are not stored locally, that is, in the `raw` folder, but anywhere else on the phone, such as on an SD card. To do this, you have to use the following code sample:

```
Uri myUri = new Uri ("uriString");

_myPlayer = new MediaPlayer();

_myPlayer.SetAudioStreamType (AudioManager.UseDefaultStreamType);
_myPlayer.SetDataSource(myUri);
_myPlayer.Prepare();
```

The first line defines a URI for the target file to be played. The next three lines set the `StreamType` and the `Uri` parameter and prepare the `MediaPlayer` class. The `Prepare()` method is the method that prepares the player for playback in a synchronous manner, meaning that this instruction blocks the program until the player is ready to play, that is, until the player has loaded the file. You can also call the `PrepareAsync()` method, which returns immediately and performs the loading in an asynchronous way.

Playing online audio

Using a code very similar to the one required to play sounds stored somewhere on the phone, we can play sounds from the Internet.

Basically, we just have to replace the `Uri` attribute with an HTTP address, as follows:

```
String url = "http://myWebsite/mario.mp3";

_myPlayer = new MediaPlayer();

_myPlayer.SetAudioStreamType (AudioManager.UseDefaultStreamType);
_myPlayer.SetDataSource(url);
_myPlayer.Prepare();
```

Also, you must request the permission to access the Internet with your application. This is done in the manifest by adding a `<uses-permission>` tag for your application as shown by the following code sample:

```
<application android:icon="@drawable/Icon" android:label="Splash">
  <uses-permission android:name="android.permission.INTERNET" />
</application>
```

See also

Refer to the *Playing a movie* recipe for playing a video.

Playing a movie

As the final recipe of this chapter, we will see how to play a movie with your Android application. Playing video, unlike playing audio, involves some special `View` elements for displaying it to the users.

Getting ready

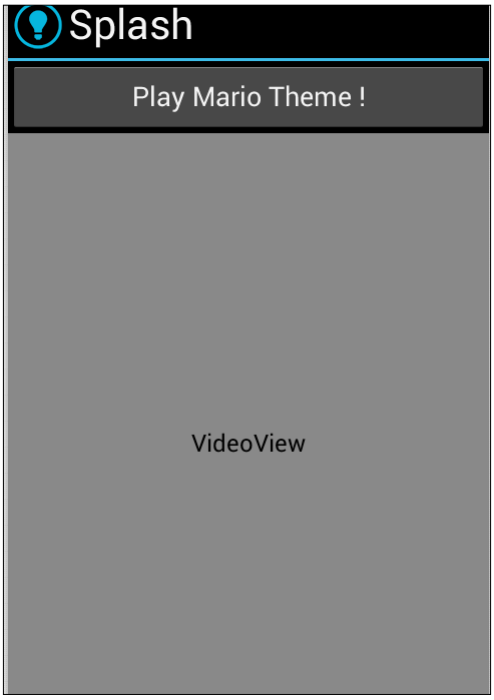
For the last time, we will reuse the same project, and more specifically, we will play a video of Mario under the button for playing the Mario theme seen in the previous recipe.

How to do it...

1. Add the following code to your `Main.axml` file under the `Layout` file:

```
<VideoView android:id="@+id/myVideoView"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
</VideoView>
```

2. As a result, the content of your `Main.axml` file should look as follows:



3. Add the following code to the `MainActivity.cs` class in the `OnCreate()` method:

```
var videoView = FindViewById<VideoView>
(Resource.Id.SampleVideoView);

var uri = Android.Net.Uri.Parse ("url of your video");

videoView.SetVideoURI (uri);
```

4. Finally, invoke the `videoView.Start();` method.

Note that playing an Internet-based video, even short one, will take a very long time as the video needs to be fully loaded while using this technique.

How it works...

As discussed in the introduction to this recipe, playing a video should involve a special view to display it to users. This special view is named the `VideoView` element and should be used in the same way as the simple `TextView` element that we saw earlier in this book:

```
<VideoView android:id="@+id/myVideoView"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
</VideoView>
```

As you can see in the previous code sample, you can apply the same parameters to the `VideoView` element as the `TextView` element, such as layout-based options.

The `VideoView` element, like the `MediaPlayer` class for audio, has a method to set the video URI, named `SetVideoURI`, and another one to start the video, named `Start()`.

See also

Chapter 10, *Taking Advantage of the Android Platform*, will expand on what you learned here about playing a video, notably by demonstrating how to use the camera and record video.

5
Using On-Phone Data

In the last three chapters we focused our efforts and energy on building a usable and, even more important, a marketable application. Nevertheless, users are still unable to really interact with our application. Indeed, they cannot create a profile, save preferences, or any other data. To address this shortfall, the Android platform allows the utilization of SQLite as an embedded, light, and powerful database. Moreover, in this chapter we will also take advantage of the LinQ mechanisms of the .Net framework to interact with this data.

In this chapter, we will pass through the following recipes:

- Storing preferences
- Simple file reading/writing
- Serializing and deserializing objects into files
- Using the SQLite database

Introduction

As we stated in the preamble of this chapter, data is important to your users. However, unlike normal desktop applications where a database is—more or less—always the right answer for assessing data storage: phones are different, they have low storage capacities. Thus, the amount of space users will allow you to use on their phones is ridiculously low because they cannot afford to lose 1 GB for each application they install.

Consequently, as programmers, we have to adapt our applications to this special environment. If you aim to reduce the data footprint of your applications to the minimum, you must master the weapons the Android platform offers you.

The storage mechanisms in Android are oriented around three different methods.

- The first one is the persistence of preferences, a.k.a user profiles or similar. The good news here is that Android has a built-in mechanism for storing key-value relationships. Therefore, if you are looking to store `Application_theme = blue`, drop the database thing and turn on the built-in mechanism we will see during the first recipe in this chapter.
- The second one is text-based files. We can do a lot with these good old text files. Indeed, we can store notes that users are taking or web pages they want to cache. Also, if we use a little more than plain text files, let's say, XML or JSON files, we can take advantage of **Serialization**. Serialization is a mechanism allowing the persistence of C# objects into a file in order to re-use them later or to send them to a server/other users. In the second recipe in this chapter, we will see a simple note-saving mechanism and the .NET library to use Serialization.
- Finally, our last option: a database. In the last recipe of this chapter, we will learn how to use the SQLite engine as our database and how to retrieve data using the **Microsoft ActiveX Data Objects** (**ADO**).

Storing preferences

As said before, the Android platform has a built-in mechanism for storing simple key-value pairs. A good example of such a pair is `theme=blue`. Therefore, we will not reinvent the wheel on this or use advanced mechanisms such as database or programmer-handled files. In this recipe, we will simply use the built-in mechanisms and learn how they work.

Getting ready

In order to follow this recipe you must have Xamarin Studio or Visual Studio with the Xamarin Studio up and running. Then, create a new project named `OnPhoneData` for experiencing the recipes in this chapter.

How to do it...

In order to store user preferences using the mechanisms Android's engines have planted for us, we will create an object typed by the `ISharedPreferences` interface. This object will be assigned by the `GetSharedPreferences()` method, which takes two arguments. The first one is determining the applications related to the preference you want to store, and the second one is determining the file creation mode. This object is only for storing preferences, therefore we have to create a reference to a `ISharedPreferencesEditor` by invoking the `.Edit()` method on the `ISharedPreferences` object.

The following code sample shows these two objects. It also shows how to increment a counter each time a user runs the application, and how to store the value of this incremented counter into the user preferences:

```
ISharedPreferences userPreferences = GetSharedPreferences
("OnPhoneData", FileCreationMode.Private);
ISharedPreferencesEditor userPreferencesEditor =
userPreferences.Edit();
count = userPreferences.GetInt ("some_counter", count);
userPreferencesEditor.PutInt ("some_counter", count++);
Console.WriteLine (count);
```

As you can see, the `ISharedPreferences` object is used to retrieve the value from the table of preferences belonging to the user and the `ISharedPreferencesEditor` object is used for updating the so-called counter.

In order to display the splash screen, we will have to create an activity and a theme that we will apply to it.

How it works...

As you can imagine, there is no magic in how this simple mechanism works. Indeed, the `ISharedPreferences` object-related classes only provide a general framework that allows you to store and retrieve persistent key-value pairs. Note that you can only store data related to a primitive type such as int, float, string, and so on. You cannot persist objects, even objects that are defined by the framework.

There's more...

Share preferences

One of the things that I have found handy is to share preferences across several applications. Indeed, the parameter for retrieving preferences is not automatically assumed by the Android platform. Consequently, you can choose a determined name for your preference's key-value table and re-use this table for another one of your applications.

The end-users will have their preferences propagated on every one of your applications installed on their phone.

Simple file reading/writing

One of the oldest and simplest ways to persist data in every system and on every language is to create a file, write our data into it, and eventually read that data at a determined time. Despite the perceived simplicity of reading and writing files, there are some obstacles to overcome in Android for security reasons. Indeed, applications are not allowed to write to/read from every part of the phone. Moreover, the file could be formatted in different ways. In this recipe, we will focus on how to read and write simple files.

Getting ready

For this recipe we will continue to use the project created in the previous recipe.

How to do it...

In the introduction of this recipe we state that, for security reasons, applications aren't allowed to write/read whatever they wish:

1. To assess this security blockage, we simply have to warn end-users about our intention of playing around with their filesystem. If they accept that, then the problem doesn't exist any more. The question now is how to gently warn my end-users? The response lies in the `AndroidManifest.xml` file, which is an XML file under the `properties` folder of your Xamarin project. This file, among other things, is responsible for presenting the permissions your application might require. In order to read/write files, two permissions are needed. These two permissions are presented by the following code, which is a fragment of the `AndroidManifest.xml` file:

```
<uses-permission android:name=
"android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name
="android.permission.READ_OWNER_DATA"
```

2. As installing the application grants end-users the permission to read/write, the code is actually pretty simple, as shown in the following code snippet:

```
string path = System.Environment.GetFolderPath
(System.Environment.SpecialFolder.Personal);
string filename = Path.Combine(path, "myfile.txt");
using (var streamWriter = new StreamWriter(filename, true))
{
  streamWriter.WriteLine(DateTime.UtcNow);
}
using (var streamReader = new StreamReader(filename)) {
  string content = streamReader.ReadToEnd();
```

```
        TextView textView = FindViewById<TextView>
        (Resource.Id.textView1);
        Console.WriteLine (content);
    }
```

In the upper code, we first retrieve the path in which we will create our files. This is done by invoking the `GetFolderPath()` method of the `Environment` class.

Note that we must fully qualify *Environment*, that is, write `System.Environment` in order to use it. Indeed, there is a name collision between the class `Environment` under the system and another `Environment` class under Android OS. Therefore, the only way to compile and deploy our project is to specify which `Environment` classes you intend to use.

After retrieving the path to the file, we create a string corresponding to the path, plus the filename using the `Path.Combine()` method, which takes the former two as parameters. We are able to create a StreamWriter. In our first `using` block, we simply use the `WriteLine()` method of the StreamWriter. As an argument for the `WriteLine()` method, we use the current date. At the end of this first `using` block we have successfully written today's date in the file. The remaining `using` block serves us by reading that file using the `ReadToEnd()` method of a newly created StreamReader. The read file is then prompted into the console by means of the `Console WriteLine` function. As proof, and seeing as data is really persisted, we compile and deploy our application five times. On the fifth run, the console shows the following outputs:

```
Forwarding debugger port 8937
Forwarding console port 8938
Detecting existing process
[monodroid-gc] GREF GC Threshold: 1800
Loaded assembly:
/data/data/OnPhoneData.OnPhoneData/files/.__override__/OnPhoneData.
dll
Loaded assembly: Mono.Android.dll [External]
Loaded assembly: System.Core.dll [External]
Loaded assembly: MonoDroidConstructors [External]
11/13/2013 8:55:36 PM
11/13/2013 8:57:01 PM
11/13/2013 8:57:43 PM
11/13/2013 8:59:27 PM
11/13/2013 9:00:03 PM
[gralloc_goldfish] Emulator without GPU emulation detected.
```

As you can see, we have the date printed out five times at `8:55:36`, `8:57:01`, `8:57:43`, `8:59:27`, and `9:00:03`.

How it works...

The theory behind file read and write is almost the same as on classical computers, but with a little difference. Therefore, we assume that the reader has mastered them.

Files you create using the FileWriter shown in the previous example are, by default, saved in the internal storage, that is, the phone's memory. Also, the file you are creating is completely and irrevocably private to your application, meaning that neither the user nor other applications can access these files. Finally, all the files your applications have created during this time and have been installed on the end-user phone, will be deleted at the same time as the application is uninstalled.

There's more...

Write on external storage

By default, files are written on the internal memory of the phone, however, there is a way to write on external storage. The external devices are, classically, SD cards that are inserted into the phone. To retrieve the path of such external storage, you will have to use the `Environment` class provided by the `Andoid.OS` namespace, as shown by the following code:

```
Android.OS.Environment.GetExternalStoragePublicDirectory ();
```

Starting from there, you can write files to the external device.

As the method `GetExternalStoragePublicDirectory` name suggests, the directory in which you will write is public. Therefore, in this case, files are accessible to end-users and other applications. You should not store critical data on external devices as they can be unmounted and files can be externally altered.

Other file types

There are plenty of file types that you could find utility for such as JSON and XML. In this case I suggest you do not reinvent the wheel and, instead, use wonderful components/modules, such as `Newtonsoft.Json` (`http://www.nuget.org/packages/Newtonsoft.Json`), which are available in the Xamarin library.

See also

Have a look at the `Json.net` component online at: `http://components.xamarin.com/view/json.net/` to understand more about the `Json.net` component.

Serializing and deserializing objects into files

In this recipe, we will learn about serialization. Serialization is the process of transforming data structures or—in our case—objects into files that can be read later by the same, or another, computer (phone) in order to create an exact replica of the object that was serialized. The possibilities are endless with serialization because you can use it for storage and communication purposes. End-users can actually send (serialized) objects to each other. You might have found the previous recipe trivial, and you are right, it is trivial. However, we can use the file to store serialized objects, which can be used for almost every purpose. Serialization will happen in two separate classes. Indeed, we first have to create a C# class that we wish to serialize and obviously deserialize. Second, we will effectively play with the serialization in the `MainActivity` class.

Getting ready

For this recipe we will continue to use the project created in the previous recipe.

How to do it...

1. Create a new C# class and name it `personal`:

   The following code sample shows the newly created `person` class. There are several particularities in this class declaration that must not be ignored.

```
using System;
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;
using System.Xml.Serialization;

namespace OnPhoneData {
  [XmlRoot("Person")]
  [Serializable]
  public class Person {
    #region IXmlSerializable Members
    public string Name { get; set; }
    #endregion
  }
}
```

The first one is using `System.Xml.Serialization`; without which serialization isn't possible at all. Indeed, this contains everything we may need for serialization purposes. The second particularity is the presence of `[XmlRoot("Person")]` and `[Serializable]` annotations. As serialization takes place in the XML file, you have to define the root name of this XML file. Therefore, the first annotation defines it. The second one enables the serialization for this particular class. While the first annotation is optional, without the second one, all attempts in serializing the `person` class will fail. Finally, last but not least, the `#region IXmlSerializable Members` and `#endregion` encapsulate the class' attributes you want to persist.

2. Serialize the `person` objects:

   In the following code sample, which takes place in the `MainActivity` class' `OnCreate()` method, we use the `MemoryStream()` method as StreamWriter. This allows us to get rid of file-writing permissions in case we just want to send the persisted file to another client/server. Then, we create an `XmlSerializer` instance, which takes into an argument the type of the class we wish to serialize. In our case, the method `typeof(Person)` is used as an argument. We now have an `XmlSerializer` instance capable of serializing `person` objects. We will not describe the two statements following the creation of the serializer as they are pure and very simple object manipulations in C#. The last statement, however, deserves some explanation. The `XmlSerializer` class offers the `Serialize()` method, which requires a StreamWriter and an object to serialize. We give them both with the `ms` and `Person` variable.

```
using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;
using System.Xml.Serialization;
using System.IO;

[Activity (Label = "OnPhoneData", MainLauncher = true)]
public class MainActivity : Activity {
  protected override void OnCreate (Bundle bundle) {
    base.OnCreate (bundle);
    SetContentView (Resource.Layout.Main);
    using (var ms = new MemoryStream()) {
      XmlSerializer mySerializer = new XmlSerializer
      (typeof(Person));
      var Person = new Person ();
      Person.Name = "Mathieu";
      mySerializer.Serialize (ms, Person);
    }
  }
}
```

At this point, you can compile and deploy your application to the emulator. However, nothing will tell you if the serialization went well. Obviously, Xamarin Studio will, for sure, tell you if something went wrong. Therefore, we will seize the opportunity to instantly deserialize our object and print the result on some component of the graphical interface.

3. Deserialize objects:

   In order to deserialize our newly serialized object, we have to add the following code after our serialization, but when it is still in use. The following code sample shows the complete code and highlights the parts related to the deserialization:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);
  SetContentView (Resource.Layout.Main);
  using (var ms = new MemoryStream()) {
    XmlSerializer mySerializer = new XmlSerializer
    (typeof(SomeData));
    var someData = new Person ();
    someData.Name = "Mathieu";
    mySerializer.Serialize (ms, someData);
    ms.Position = 0;
    Person deserialize =
    (Person)mySerializer.Deserialize (ms);
    Button button = FindViewById<Button>
    (Resource.Id.myButton);
    button.Text = deserialize.Name;
  }
}
```

As we are not using a real StreamWriter, we have to specifically set the position at which the `MemoryStream` instance should be, in order to read the correct data. Forgetting this statement will certainly result in a Null Exception while reading. Then, using the same `XmlSerializer` object and our StreamReader set at the position `0`, we invoke the `Deserialize()` method. This method returns an object, therefore, we have to cast the return into a `Person` object. Finally, as we did many times in *Chapter 1*, *Getting Started* and in *Chapter 6*, *Populating Your GUI with Data*, we retrieve a reference to a button and set the button's text with the `Name` value of the newly affected `Person` object.

The following screenshot shows us the result of the serializing/deserializing process we saw in the previous recipe:

As shown in the preceding screenshot, the button does have **Mathieu** as text. This value comes from an object that has been created, populated, serialized, and finally deserialized.

How it works...

For serialization, generated files are XML based and therefore human-readable and modifiable. Indeed, the previous sample will result in the following file:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Mathieu</Name>
</Person>
```

As you can see, this is very simple, and in case you have to, you could modify files corresponding to serialized objects. Here's the JSON version of the same serialization:

```
{
  "Person": {
    "-xmlns:xsi": "http://www.w3.org/2001/XMLSchema-instance",
    "-xmlns:xsd": "http://www.w3.org/2001/XMLSchema",
    "Name": "Mathieu"
  }
}
```

Using the SQLite database

SQLite is a **Relational DataBase Management System** (**RDBMS**) that has the benefit of being very small (~655KB) and written in C. These two assets make SQLite very portable and our first choice as programmers wishing to use **Object Relational Mapping** (**ORM**) embedded in a phone. Also SQLite respects the **Atomicity, Consistency, Isolation, and Durability** (**ACID**) principles, which ensure we have consistent data at any time.

Getting ready

Despite the fact that SQLite is the favorite RDBMS of thousands of programmers when it comes to using a database of fun, the SQLite .NET library isn't part of the standard library that comes with Xamarin. Therefore, we have to add it to our projects. You can add it in two different ways, such as the following:

- The first one is to download it from the Github account of the Xamarin team and then add the C# class to your project. Here's the URL for the SQLite .Net class: `https://github.com/praeclarum/sqlite-net/blob/master/src/SQLite.cs`.
- The alternative is to download the component in the component store. The component store is accessible through Xamarin Studio and Visual Studio in order to download and install components. Components are functionalities that you can use in your project. For example, there are components for Facebook integration and others. However, pay attention to which component you download and whether you are on the standard edition of Xamarin. Indeed, some of them will prevent your application from compiling because they make your application too large and prevent the compilation.

How to do it...

In the same flavor as serialization, we will have to interact with SQLite in two separate states. The first one consists of annotating a C# object in order to be able to create the corresponding table into the SQLite database. Then, we will be able to concretely interact with the database, that is, create (`insert`), retrieve (`select`), update, and delete information (rows).

Create a simple object. We can reuse the `Person` class we created for the serialization, but we have to remove all annotations referencing the serialization. The following code exposes the `Person` class refactored to be used with SQLite:

```
public class Person {
  [PrimaryKey, AutoIncrement]
  public int Id { get; set;}
  public string Name { get; set; }
}
```

It is mandatory to use the SQLlite namespace in which the SQLite class is defined: using SQLite;

We add an `ID` variable to a `Person` object. This `ID` variable is an integer and serves as a primary key for this object. In a relational database, primary keys refer to a key that uniquely defines a row. In our case, the primary key will range from 0 to 2,147,483,647 (2,147,483,647 being the max value of a unit). Moreover, this primary key will be incremented automatically, meaning that we must not affect a value to this field. The database will generate it while inserting the associated row:

1. Before inserting data to our database, we have to create it. We can do it using the following piece of code:

```
string folder = System.Environment.GetFolderPath
(System.Environment.SpecialFolder.Personal);
var db = new SQLiteConnection (System. IO. Path. Combine
(folder, "myDb. db"));
```

   The first line of the preceding code retrieves the personal folder of the user. Then, in the second statement, we allocate the variable `db` with a new `SQLiteConnection` object. The `SQLiteConnection` object's constructor takes the full path towards our database. As mentioned earlier in this chapter, we use the `Path.Combine` function to combine the personal folder and the name of the database. A very interesting thing about creating a `SQLiteConnection` object is that if the database, that is, `myDb.db` exists, than a simple connection to it will be created. In the instance when the database is not yet created, basically, the first time the end-user runs the application, the `SQLiteConnection` object will create the so-called database.

2. Even if the previous step creates our database, we still have to create the table in which we will insert our people. Creating the table is very straightforward. In fact, we can do it in one statement:

```
db.CreateTable<Person> ();
```

   There is only one statement, however, it is a tricky one. You have to invoke this method in the same way as you would create a generic class in C#. The generic argument must be the class you want to copy in the database. Copy means creating a database representation of this class (and its attributes) in the database.

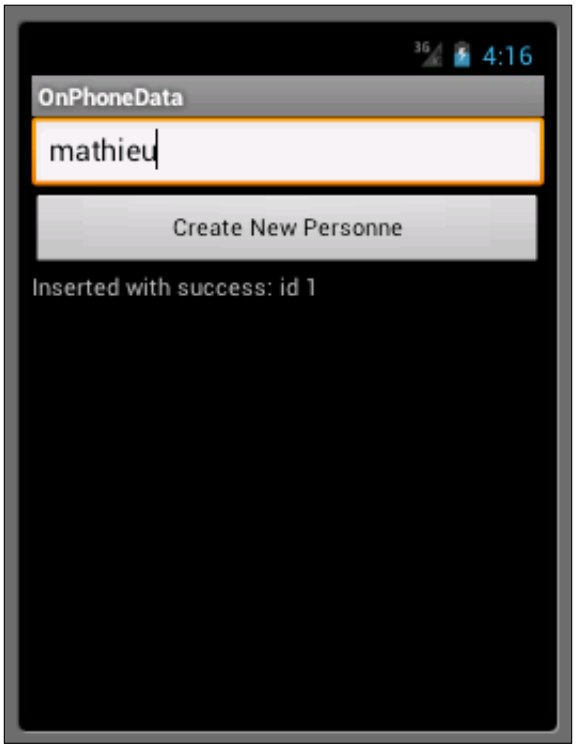
3. Add a new record to the database:

   In this step, we will create a new person using a very basic graphical form. This form will be composed of only three elements, an `EditText` for entering the person's name, a `Button` to insert the person in the database and, finally, a `TextField` that will display whether or not the insert is a success along with the generated ID of the person. Let's consider the following code:

```
button.Click += delegate {
```

```
    Person Person = new Person { Name = FindViewById<EditText>
    (Resource.Id.editText1).Text };

    int id = db.Insert(Person);
    FindViewById<TextView> (Resource.Id.textView2).Text = "Inserted
    with success: id "+id;

};
```

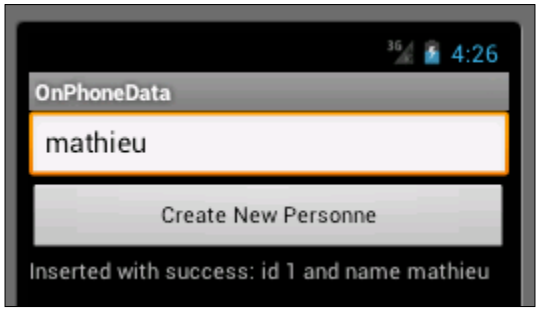
In the preceding piece of code, we add a new behavior to the `Click` event of the button using the `+= delegate`. This new behavior starts with the creation of a new `Person` object which has, as a name, the text typed in the `EditText` field. Then we use the `db` object, which was affected by the `new SQLiteConnection (System.IO.Path.Combine (folder, "myDb.db"));` during the previous step. More specifically, we will invoke the `Insert()` method on this object to the newly created person in an argument. Finally, we will update the text of the `TextView` element with the ID of the person generated by the database. The following screenshot shows us the result:



4. The preceding screenshot presents this simple application right after clicking on the **Create New Personne** screenshot.
5. Retrieve a row from the database.
6. Logically, the next step towards mastering SQLite on Xamarin is retrieving some data. In order to meet this goal, you will use the `Get` function proposed by the `SQLiteConnection` object. The `Get()` method must be invoked in the same way as the `CreateTable()` method with the targeted class between `<>`. The following code shows how:

```
Person personFromTheDatabase = db. Get<Person>(id);
FindViewById<TextView> (Resource.Id.textView2). Text =
"Inserted with success: id "+ personFromTheDatabase. Id+"
and the name "  + personFromTheDatabase. Name;
```

   In the previous piece of code, the first statement retrieves the row identified by the primary key ID. Then, in the second statement, we update the text of the `TextView` element with the name and ID of the retrieved person. If we insert this code right after the insertion of the person we saw in the last step, we have the following screen:



7. As you can see, the field is now composed of the ID and the name of the person. It would be trivial if we had directly affected the text from the `Person` object, however, we first persist this person and then retrieve it by its ID against the database in order to create the text field. It's also noteworthy that we can select all the table records by not specifying any ideas. The following code retrieves all the records of the `Person` table:

```
var listOfPerson = db.Get<Person>();
```

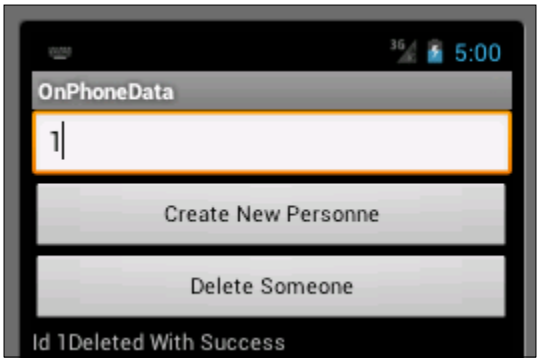
8. Delete a record:

   The delete operation is as simple as the other operations. Indeed, you have to specify which ID you want to delete and the targeted table. Here is an example of deleting a record inside the person identified by an ID typed by the user:

```
Button deleteButton = FindViewById<Button>
(Resource.Id.button2);
deleteButton.Click += delegate {
```

```
    var DeletedId = db.Delete<Person>
    (Convert.ToInt32(FindViewById<EditText>
    (Resource.Id.editText1).Text));
    FindViewById<TextView> (Resource.Id.textView2).Text = "Id
    " + DeletedId + "Deleted With Success";
};
```

In the previous code, we considered a `delete` button. This deletes a button's `Click` event, which will parse into an integer of the ID typed by the user and then delete the row in the database. The following screenshot shows what could be the corresponding application:



9. Commit your changes and then rollback:

   SQLite is a transactional database and, as such, your operations have to commute in order to truly be persisted. A transaction is a set of operations that will be persisted in the database only when the commit operation is performed. Also, we can restore the database in the state it was at the last commit by using the rollback operation. In other words, when we have completed a logically related sequence of operations, I perform a commit and if something went wrong during the sequence of the operation I perform a rollback:

```
db.Commit();
db.Rollback();
```

How it works...

It is now time to understand what is really happening to the user's phone when we use SQLite databases. Behind the scenes (below is the C# abstraction offered by Xamarin) we are working with SQLiteDatabase objects defined by the Android SDK. This class, that is SQLiteDatabase, represents the database and provides methods for the SQLite operations we saw earlier. When we perform any operation, we receive a cursor in response. A cursor in a database is a control structure, which allows us to visit records corresponding to a query. Also, cursor facilitates further processing with the returned records.

Here we expose the different arguments you can use on the fields you intend to store according to the Xamarin documentation:

- **[PrimaryKey]**: This attribute can be applied to an `integer` property to force it to be the underlying table's primary key. Composite primary keys are not supported by SQLite.
- **[AutoIncrement]**: This attribute will cause an integer property's value to be auto-incremented for each new object inserted into the database.
- **[Column(name)]**: Supplying the optional name parameter will override the default value of the underlying database column's name (which is the same as the property).
- **[Table(name)]**: This marks the class as being able to be stored in an underlying SQLite table. Specifying the optional name parameter will override the default value of the underlying database table's name (which is the same as the class name).
- **[MaxLength(value)]**: This restricts the length of a text property when a database insert is attempted. Consuming code should validate this prior to inserting the object, as this attribute is only 'checked' when a database insert or update operation is attempted.
- **[Ignore]**: This causes SQLite.NET to ignore this property. This is particularly useful for properties that have a type that cannot be stored in the database, or properties that model collections that cannot be resolved automatically by SQLite.
- **[Unique]**: This ensures that the values in the underlying database column are unique.

There's more...

LinQ

LinQ stands for **Language-Integrated Query** and was first introduced in Visual Studio six years ago in order to extend the C# and VB.Net syntax capabilities. LinQ has become the de-facto standard for querying and updating data in the .Net framework because of its easily learned keywords and patterns. As we use C# and an open-source implementation of the .Net, which also supports LinQ, we can use it for interacting with our data. The following code shows how to use LinQ in our little `Person` example:

```
var personsNamedMathieu = from p in db.Table<Person> ()
    where p.Name = "Mathieu"
    select p;
Console.WriteLine (
">", personsNamedMathieu.FirstOrDefault().Name);
```

As you can see, we build a request in a similar way that we would in pure SQL except that strings are replaced by objects. Indeed, we assign a variable with the result of a selection in the person table. Then, we display the first statement inside the console. Obviously, LinQ isn't within the scope of this book, however, many books and online references are available. For example, see the official Microsoft documentation: `http://msdn.microsoft.com/en-us/library/vstudio/bb397926.aspx`.

Direct SQL

Xamarin proposes a powerful abstraction to the SQLite support provided by the Android SDK. Despite the fact that we do recommend you to use these tools, you may feel more comfortable in directly writing SQL. If this is the case, here is the solution:

```
var personsNamedMathieu = db.Query<Person>("SELECT * FROM
Person WHERE Name = ?", "Mathieu");
  foreach (var p in personsNamedMathieu) {
    Console.WriteLine ("p: " + p.Name);
  }
```

Using the `Query` method of the `Database` object, we can directly write some classical SQL using the"`?`" for variables. If you have more than one dynamic value to include in your SQL statement, you will have to add more "`?`" variables, followed by more values separated by a comma. At the end of the code sample, we simply use a `foreach` block in order to visit all the records returned by the query.

See also

Also see the next chapter, which will map data coming from the phone to graphical components.

6
Populating Your GUI with Data

In the previous chapter, we learned how to work with data stored in the internal memory of our customer's phone using file storage and SQLite. In this chapter, we will see how to populate our GUI (Graphical User Interface) with this data.

In this chapter, we will go through the following recipes:

- Populating simple objects
- Populating datepicker
- Populating the spinner
- Populating ListViews
- Creating a custom adapter

Introduction

After offering the possibility to insert and delete data from a database or file, the logical follow-up is to display this data on our GUI. We already did this for simple elements like buttons and labels in the previous chapter without paying too much attention to what was happening.
Also, we never tried to display types other than string or int. In this chapter, we will look at the mechanisms involved in the modification of the data displayed by the GUIs and display complex objects into complex views such as ListView or a drop-down list (known as the spinner).

To populate these GUI objects, we will need to master the concepts of adapter play a little with the `String.xml` file of the application.

Populating simple objects

For this short recipe, we will review some code samples we saw before without paying attention to them. Indeed, we will simply populate the `TextView` and `TextLabel` elements.

Getting ready

Create a new solution named populating `GUIs` object and then open the project of the same name that Xamarin Studio has created for you.

How to do it...

1. Open the `Main.xaml` file under the `Ressources/layout` option:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
<Button
    android:id="@+id/myButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  android:text="@string/hello" />
  <EditText
    android:inputType="date"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  android:id="@+id/editText1" />
</LinearLayout>
```

   If you remember from *Chapter 3*, *Building a GUI*, where we learned the different available layouts and how to build a GUI, we will see that we simply create a linear layout in which each element comes after the others. We also add a button and an editable text box.

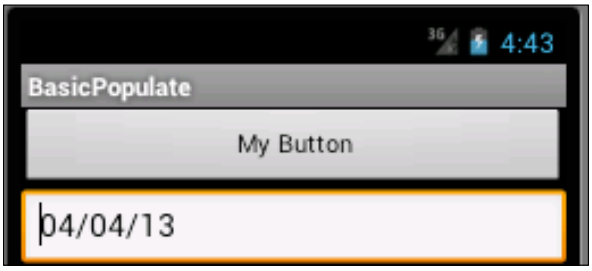
2. Add the following code as a replacement of the `onCreate()` method in the `Main.cs` file:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);
  // Set our view from the "main" layout resource
  SetContentView (Resource.Layout.Main);
```

```
        // Find the button and change the text
        Button button = FindViewById<Button>
        (Resource.Id.myButton);
        button.Text = "My Button";
        // A textView with a Date Format
        TextView textView = FindViewById<TextView>
        (Resource.Id.editText1);
        textView.Text = "04/04/13";
    }
```

3. Run the program.

   Heres a screenshot of what you should have on your emulator:



4. As simple as that, we populate a button and a `TextView` element with data.

How it works...

We learned in *Chapter 2*, *Mastering the Life and Death of Android Apps*, that if the `OnCreate()` method is one of the methods called toward the display of an activity and the `increase()` method happened before that, the activity is actually rendered to the clients. Therefore, in the `onCreate()` method, we have the possibility to manipulate the default look and feel of our GUIs before their first display to the user.

In the preceding code, we first assign the `Main.xaml` file as the layout of our activity just after calling the default constructor of the `Activity` class. Then, we use the generic method `FindViewById<Button>` in order to obtain references to our `Button` and `TextView` elements. Finally, we use the `Text` property to update the text they are displaying.

There's more...

In this *There's More...* section, we will showcase each component.

Labels

You can reuse the exact same code to update the values of classical labels such as `EditableTextView`, a specialization of `TextView` that is the class used for the label.

Retrieve the text

Obviously, text is a `get set` enabled property. Therefore, you can use the following to retrieve the text:

```
TextView textView = FindViewById<TextView>
(Resource.Id.editText1);
var text = textView.Text;
```

Know if the text has been modified

We can supercharge the `TextChanged()` method of any `TextView` element in order to add some behavior when the text is changed.

```
textView.TextChanged += (object sender,
Android.Text.TextChangedEventArgs e) => {
  // Some actions
};
```

See also

Check out the next recipe to see how to populate the datepicker.

Populating the datepicker

In many languages that offer graphical interaction with users, the datepickers are a nightmare, especially when you want to transform this data retrieve from the datapicker to process it in a database or business object. For example, in Java + Swing, you first have to pick the right `Date` object between `java.util.date` and `java.sql.date` and then try to manipulate them with the `Calendar` class, which doesn't offer the same level of abstraction as the `Date` object. Xamarin comes with a very simple way to handle the datepicker, and we will see how in this recipe.

Getting ready

To follow this recipe, create a new project in the same solution as the **Populating Simple Object** project and name it `Populating Datepicker`.

How to do it...

1. Open the `Main.xaml` file under `Ressource/Layout` folder and modify it so it looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <DatePicker
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  android:id="@+id/datePicker1"/>
</LinearLayout>
```

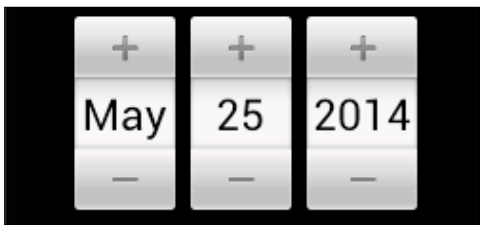
   Here, we have created a simple `LinearLayout` containing only a `datePicker` object.

2. Modify the `onCreate()` method in order to obtain the following:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);
  // Set our view from the "main" layout resource
  SetContentView (Resource.Layout.Main);
  // A datePicker
  DatePicker datePicker = FindViewById<DatePicker>
  (Resource.Id.datePicker1);
  //Set the date pcker to may 25 2014
  datePicker.DateTime = new DateTime (2014, 05, 25, 23, 00,
  00);
}
```

3. Run the program.

   Here a screenshot of what is expected to be displayed, which showcases a simple `DateTime` picker:



How it works...

As for the previous recipe, we first call the base constructor and then affect the `Main.xaml` file as the layout of our activity and obtain a reference to the `datePicker` object using the `FindViewById()` method. However, unlike in the previous recipe, we can't set a simple string as the value of a datepicker. Therefore, we create a `DateTime` object using one of the twelve constructors of the `DateTime` class. In the example, we choose to display May 25, 2014 at 11 p.m., 0 minutes and 0 seconds. Then, we simply equate this newly created `DateTime` object on the `DateTime` property of the `DatePicker` object.

Here are the 12 constructors with their arguments:

```
public DateTime()

public DateTime (long ticks, DateTimeKind kind)

public DateTime (int year, int month, int day, int hour, int
minute, int second, Calendar calendar)

public DateTime (int year, int month, int day, int hour, int
minute, int second, int millisecond, Calendar calendar,
DateTimeKind kind)

public DateTime (int year, int month, int day, int hour, int
minute, int second, int millisecond, DateTimeKind kind)

public DateTime (int year, int month, int day, int hour, int
minute, int second, DateTimeKind kind)

public DateTime (int year, int month, int day)

public DateTime (long ticks)

public DateTime (int year, int month, int day, int hour, int
minute, int second)

public DateTime (int year, int month, int day, int hour, int
minute, int second, int millisecond)

public DateTime (int year, int month, int day, Calendar calendar)
```

Obviously, we can retrieve the selected date and add the `EventHandler` handler in the same way as we did in the previous recipe.

There's more...

Date manipulation

It is pretty easy to manipulate `Date` in C# and therefore, in Xamarin. Here a little example to check if a birthdate picked on a `datePicker` object is over 18:

```
DateTime today = DateTime.Today;
int age = today.Year - datePicker.DateTime.Year;
if (datePicker.DateTime.Year > today.AddYears (-age)) {
  age--;
}

if (age < 18) {
  Console.WriteLine ("Under 18");
}
```

We use the static `Today` property of the `DateTime` class to create a `DateTime` object representing the current date. After that, we create an `age` variable with the subtraction between today and the picked date. Finally, we use the `AddYears()` method of the `DateTime` object to subtract the age.

See also

Have a look at the msdn website to learn more about the `DateTime` object: `http://msdn.microsoft.com/en-us/library/system.datetime.datetime(v=vs.110).aspx`.

Populating the spinner

The spinner is the name given by Android developers to the classic drop-down list. Drop-down lists, or drop-down menus, are very useful for choosing from a vast list of choices. When deactivated, they only show the current choice, and we show all the possible choices.

Getting ready

In order to follow this recipe, you will have to create yet another project in the current solution. Name this project **PopulateSpinner**.

How to do it...

1. Open the `main.xaml` file of your newly created project and modify it so it looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:minWidth="25px"
  android:minHeight="25px">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginTop="10dip"
    android:text="@string/canada"
  />
  <Spinner
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:id="@+id/spinner1"
  android:prompt="@string/canada" />
</LinearLayout>
```

   In the previous code, we create a `LinearLayout` element containing both a `TextView` and a `Spinner` parameter. The `TextView` parameter and the `Spinner` parameter use the `@string/Canada` command. The value of this string is held in the `string.xml` file.

2. Update the `String.xml` file to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="canada">Choose a State</string>
  <string-array name="canada_array">
    <item>Ontario</item>
    <item>Québec</item>
    <item>Nova Scotia</item>
    <item>New Brunswick</item>
    <item>British Columbia</item>
    <item>Prince Edouard Island</item>
    <item>Saskatchewan</item>
    <item>Alberta</item>
```

```
    <item>Newfoundland and Labrador</item>
  </string-array>
</resources>
```

The `String.xml` file contains a classical string identified as `Canada` and containing `Choose a State` as a value. The second item in this file is a string array composed of several string items.

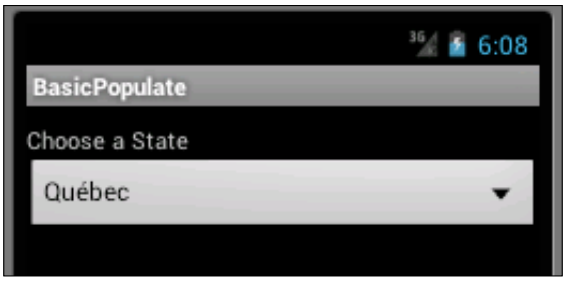
3. Attach the string-array to the `spinner` object in your `onCreate()` method in the `main.cs` file:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);

  // Set our view from the "main" layout resource
  SetContentView (Resource.Layout.Main);

  // Create the spinner reference
  Spinner spinner = FindViewById<Spinner>
  (Resource.Id.spinner1);

  // Create the adaptor for the Spinner with the state of
  Canada
  var adapter = ArrayAdapter.
  CreateFromResourceCreateFromResourceCreateFromResource (
    this, Resource.Array.canada_array,
  Android.Resource.Layout.SimpleSpinnerItem);

  // Add the adapter type
  adapter.SetDropDownViewResource
  (Android.Resource.Layout.SimpleSpinnerDropDownItem);

  //Add the adapter to the spiner
  spinner.Adapter = adapter;
}
```

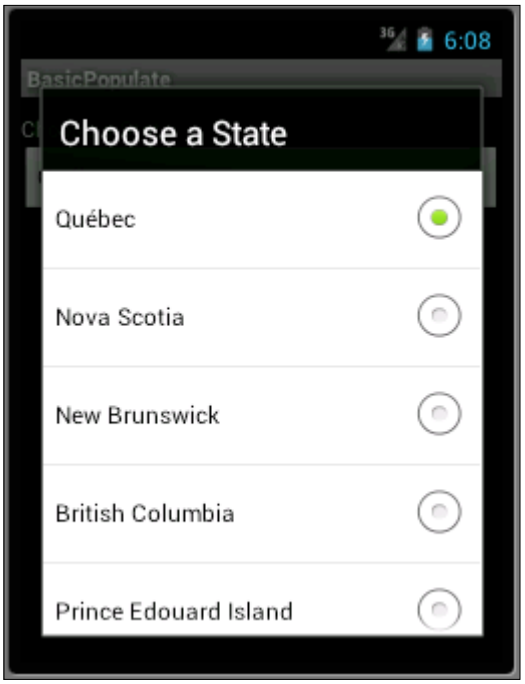
Here, we first call the base constructor and obtain a reference to the `Spinner` object on our GUI. Then, we create an `ArrayAdapter` object and affect this adapter to our spinner. We will explain what an `ArrayAdapter` is in the *How it works...* section .

1. Run the programs.

   Here a screenshot of the expected result in the default state:



   The next screenshot represents the visual state of the `spinner` object when you click on it:



2. Now that we have a functional `spinner` object, we can add some behavior when the selected value changes. Add the method in your `Main` class:

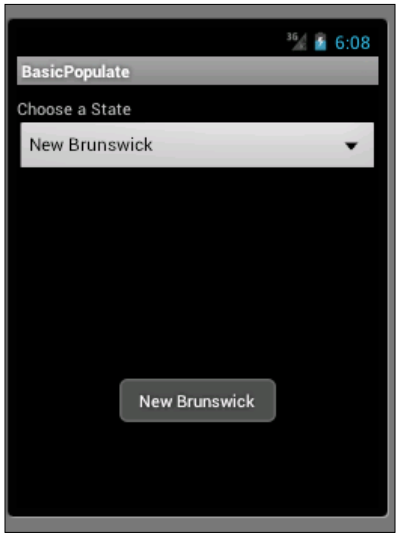
```
private void spinnerSelected (object sender,
AdapterView.ItemSelectedEventArgs e) {
  // Cast the event sender as spinner
  Spinner spinner = (Spinner)sender;

  // Create a toast with the selected spinner
  Toast.MakeText (this, (String) spinner.GetItemAtPosition
  (e.Position), ToastLength.Long).Show ();
}
```

   This code is a simple method with both a `sender` and an `AdapterView.ItemSelectedEvent` as arguments. When a spinner fires the `ItemSelectedEvent` instance, we will enter in this method and create a `Toast` containing the selected item. Toast provides simple feedback about an operation in a small popup.

3. Add the `spinnerSelected()` method to the `itemSelected` property of our `Spinner` object:

```
// Add an event handler when an item is selected
spinner.ItemSelected += new
EventHandler<AdapterView.ItemSelectedEventArgs>
(spinnerSelected);
```

This code must be added before the end of the `OnCreate()` method and be specified to call the `spinnerSelected()` method when a selection is made. The following screenshot exposes the expected behavior:



How it works...

This `Spinner` object owns a property called `Adapter`. We assign a value to this adapter at the very last line of the code sample presented in the third step of the *How to do it...* section. Adapters are the needed bridge between the `AdapterView` instance and the underlying data of the said view. The `Adapter` object will enable access to the data and create a `View` instance for each item inside our data. In other words, the `Adapter` object has built a view for each one of Canada's states written in the `String.xml` file.

Adapter is a widget; therefore, we cannot use it directly. Indeed, we have to use one of its subclasses. In our previous examples, we use an `ArrayAdapter` class. The `ArrayAdapter` classes are arrays of string. We have created it with the following code:

```
ArrayAdapter.CreateFromResource (
  this, Resource.Array.canada_array,
  Android.Resource.Layout.SimpleSpinnerItem);
```

The first argument of the `CreateFromResource()` method is the context of this adapter. Consequently, we pass the current activity (`this`) as the value. Then comes the data followed by the `Layout` type. The data is the `string_array` of our `String.xml` file, while `SimpleSpinnerItem` is the default spinner view. This view represents the view generated by the adapter for every single item, but not the view of the spinner when we click on it. This view has been defined by the following statement:

```
// Add the adapter type
adapter.SetDropDownViewResource
(Android.Resource.Layout.SimpleSpinnerDropDownItem);
```

The `SimpleSpinnerDropDownItem` instance is the default view for the spinner look and feel when active.

See also

Check out the next section to see how to create an `ArrayAdapter` instance programmatically and the last section of this chapter to learn how to create a custom adapter. Also, go to `http://developer.android.com/reference/android/R.layout.html`.

Populating ListView

A popular alternative to a spinner is the `ListView` instance. The `ListView` instances are views that will show a list of items in full screen with a vertical scrolling. In other words, it is equivalent to a full screen spinner, which is always active.

Getting ready

In order to follow this recipe, create a new project in the very same solution and name it `Populating ListView`.

How to do it...

1. Edit the `Main.axml` file under `Resource/Layout` file with the following:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:id="@+id/textItem"
  android:textSize="44sp"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content" />
```

2. Create another `.axml` file under `Resource/Layout` and name it `TextViewItem.axml`.

3. Add the following code in the `TextViewItem.axml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:id="@+id/textItem"
  android:textSize="44sp"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content" />
```

   This new layout will represent every item of the `ListView` instance.

4. Replace the code of your `MainActivity` class with the following:

```
[Activity (Label = "States", MainLauncher = true)]
public class MainActivity : ListActivity {

  string[] items; // items of the list

  protected override void OnCreate (Bundle bundle) {
    // Class the base onCreate
    base.OnCreate (bundle);

    // Populating the items array
    items = new string[] { "Ontario","Québec","Nova
    Scotia","New Brunswick","Manitoba","British Columbia",
```

```
        "Prince Edouard Island", "Saskatchewan", "Alberta",
        "Newfoundland and Labrador"};

        // Create an adaptor containing the data
        ArrayAdapter adapter = new ArrayAdapter<String>(this,
        Resource.Layout.TextViewItem, items);

        ListAdapter = adapter;

    }

}
```

In the previous code, we created an array of string containing the Canadian states as well as an `ArrayAdapter` instance with the context, the layout for a single item, and the items as arguments. Also note that we are not extending a classical `Activity` but a `ListActivity`. `ListActivity` is a special activity that can be bound to a data source.

5. Run the program.

   The following screenshot exposes the expected behavior of the previous codes:

If you get the following error:

[ArrayAdapter] You must supply a resource ID for a TextView

[AndroidRuntime] Shutting down VM

It means that you made a mistake when creating the `ArrayAdapter` instance and gave the `MainLayout` class as an argument in place of the `TextViewItem` class. As a reminder, adapters are responsible for generating a view for each item; therefore, you need to give them a one item enabled view.

6. As we did for the spinner, we can add an event listener and display a toast when an item is selected.

```
protected override void OnListItemClick(ListView l, View v,
int position, long id) {
  // The name of the clicked state
  var items = items [position];

  // Create a Toast with this line
  Android.Widget.Toast.MakeText(this, items,
  Android.Widget.ToastLength.Short).Show();
}
```

The previous code has to be added in the `Main` class and override the default `OnListItemCLick()` method of the `ListActivity` class. In this override, we receive the `ListView`, the `view`, the `position` of the selected item, and a unique `id`. We use the position of the clicked item to retrieve the state name in the item array and then create a simple `Toast`. The following screenshot shows the interface after a click:

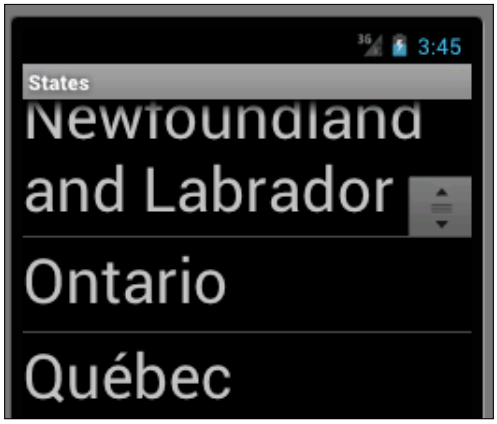
How it works...

The `ListView` elements are specialization/implementation of the `AdapterView` instance and the `AbsView` classes. In the same way as the spinner, they own an adapter, which operates as the bridge between the data and the view. The adapter has to be created with a reference of the layout built for a single item.

There's more...

Fast scroll

The `ListView` instance can be populated with a high number of items. Therefore, Android developers have implemented an option called fast scroll. The fast scroll will be represented by a little icon on the right side of the phone, as exposed in the following screenshot:



To activate this fast scroll option, you can either do it in the `.axml` file or in the `onCreate()` method file, as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:p1="http://schemas.android.com/apk/res/android"
  p1:minWidth="25px"
  p1:minHeight="25px"
  p1:layout_width="fill_parent"
  p1:layout_height="fill_parent"
  p1:id="@+id/listView1"
  p1:fastScrollEnabled="true" />
```

Or:

```
  ListView.FastScrollEnabled = true;
```

See also

See the next section to learn how to create a custom adapter.

Creating a custom adapter

The `ArrayAdapter<Type>` instances are really powerful and easy to use. However, they do lack flexibility. Indeed, you cannot populate a `ListView` instance or spinner with a business object of your own, for example. In this recipe, we will learn how to create a custom adapter.

Getting ready

Create two C# classes in the **Populating ListView** project named `States` and `StateAdaptor`, respectively.

How to do it...

1. Add the following code in the `State` class:

```
public class State {
  private string _Name;
  public string Name {
    //set the state name
    set { this._Name = value; }
    //get the state name
    get { return this._Name; }
  }

  private int _Pop;
  public int Pop {
    //set the state pop
    set { this._Pop = value; }
    //get the state pop
    get { return this._Pop; }
  }

  // Default constructor
  public State (String name, int pop) {
    this.Name = name;
    this.Pop = pop;
  }
}
```

This class represents states in terms of names and population. Just like the constructor, it owns both the getter and setter for the name and population.

2. Add the following code in the `SateAdaptor` class:

```
public class StateAdaptor : BaseAdapter<State> {
  // Data
  State[] items;
  Activity context;

  //Default constructor
  public StateAdapter(Activity context, State[] items) :
  base() {
    this.context = context;
    this.items = items;
  }

  // Return a row identifier
  public override long GetItemId(int position) {
    return position;
  }

  //return the data associated with a particular row
  number.
  public override State this[int position] {
    get { return items[position]; }
  }

  //tells how many rows are in the data.
  public override int Count {
    get { return items.Length; }
  }

  //Return a View for each row, populated with data.
  public override View GetView(int position, View
  convertView, ViewGroup parent) {
    var view = convertView; // re-use an existing view, if
    one is available
    if (view == null) // otherwise create a new one
    view = context.LayoutInflater.Inflate
    (Resource.Layout.TextViewItem, null);
    view.FindViewById<TextView>(Resource.Id.textItem).Text
= items[position].Name + " Pop:" + items[position].Pop;
    // Concatenate Name & pop of state
    return view;
  }
}
```

The `StateAdaptor` class extends the `BaseAdaptor` generic class and overrides the `GetItemId`, `Count`, `this` and `GetView()` methods. The `GetView()` method is the one responsible for creating the view associated with every item in the data set and, as you can see, we concatenate the name of the state and population in this method. Therefore, both sets of information will be printed on the screen. We will further explain other methods in the upcoming *How it works...* section.

3. Modify the `Main.cs` class of the **Populate ListView** project in order to use this newly created `Adaptor` object:

```
string[] statesName; // items of the list
int[] statesPop;
State[] states;

protected override void OnCreate (Bundle bundle) {
  // Class the base onCreate
  base.OnCreate (bundle);

  // Populating the statesName array
  statesName = new string[] { "Ontario","Québec","Nova
  Scotia","New Brunswick","Manitoba","British Columbia",
  "Prince Edouard Island", "Saskatchewan", "Alberta",
  "Newfoundland and Labrador"};

  // Populating the statePopArray
  statesPop = new int[] { 12851821, 7903001, 921727,
  751171, 1208268, 4400057, 140204, 1033381, 3645257,
  514536};

  // Initiate the states array
  states = new State[statesName.Length];

  // Creating the states
  for(int i = 0; i < states.Length; i++) {
    states[i] = new State (statesName [i], statesPop
    [i]);
  }

  // Create an adaptor containing the data
  ListAdapter = new StateAdapter(this, states);

}
```

In the previous code, we create three arrays of `String`, `Ints`, and `States`, respectively. The first two arrays are fed with the state names and their population. Then, we populate the `State` array using both sets of information. Finally, we affect a `StateAdapter` instance to the `ListAdapter`. The following screenshot shows the result:

As expected, the list's items are now a combination of the state names and their population.

How it works...

The only difference between this recipe and the previous ones is the use of `CustomAdapter`. To implement a custom adapter, you first need to extend the generic `BaseAdapter` class and then override the following methods:

- `Count`: Tells the control how many rows are in the data.
- `GetView`: Returns a `View` for each row, populated with data. This method has a parameter for the `ListView` instance to pass in an existing, unused row for re-use.
- `GetItemId`: Returns a row identifier (typically the row number, although it can be any long value that you like).
- `this[int] indexer`: Returns the data associated with a particular row number.

Three out of these four methods are trivial, and we can easily understand them from the commented code in the second step of the previous section. However, the `GetView()` method is a little bit trickier. As a reminder, here's the code:

```
//Return a View for each row, populated with data.
public override View GetView(int position, View convertView,
ViewGroup parent) {
  View view = convertView; // re-use an existing view, if one is
  available
  if (view == null) // otherwise create a new one
  view = context.LayoutInflater.Inflate
  (Resource.Layout.TextViewItem, null);
  view.FindViewById<TextView>(Resource.Id.textItem).Text =
  items[position].Name + " Pop:" + items[position].Pop;
  // Concatenate Name & pop of state
     return view;
  }
}
```

First of all, this method has three arguments: the position, the `View`, and the `ViewGroup`. The `View` is a graphical representation of a single item while the `ViewGroup` represents the group in which its items are contained, in other words, the `ListView` as a whole. This method will be called for every item in the data set as long as the row is displayed. Therefore, for every item, we first check if the view is null, if so, we create a new view from the `TextViewItem` layout we created in the previous recipe. However, views are not created per item; they are created by view type, and if an item goes out of scope on the list, the view it was attached to previously gets reused by another item. Once the view is successfully created, we obtain a reference to the `TextView` inside the `TextViewITem` layout and modify its text with the combination of the name and the population of a state using the items array on the current position.

You can use this custom adapter for the spinner, too.

See also

See the previous chapter for information on populating your adapter directly from stored data instead of creating the data every time.

7
Using Android Services

Android services are important components of the Android ecosystem. They enable work to be done in the background regardless of the user activity. In this chapter, you will learn when Android services are best suited to situations, and how to use them in your application. More specifically, we will pass through the following recipes:

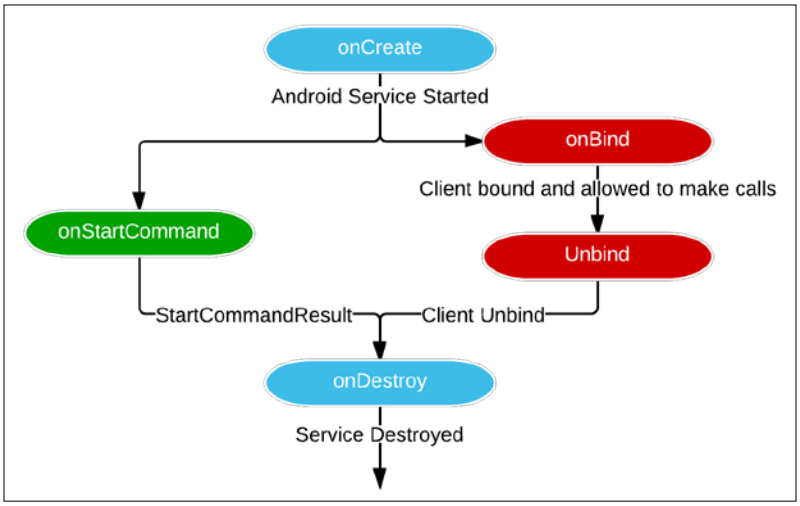
- Implementing a started service
- Implementing a bound service
- Sending notifications from your service
- Creating a news feed service

Introduction

Android services are some of the building blocks available in Android, and they allow tasks to be performed in the background. In addition, they follow their own lifecycle, completely disconnected from the application lifecycle seen in *Chapter 2*, *Mastering the Life and Death of Android Apps*. There are three kinds of Android services that fulfill two specific uses:

- **Started Android services**: On the one hand, started Android services are made to execute background tasks that don't interact with users, for example, downloading content from the Internet. Also, services neither create a new thread, nor do they run in another process. It's up to the application to handle threading.
- **Bound Android services**: On the other hand, bound Android services provide an interface that can be accessed programmatically. The Android operating system provides a wide range of bound Android services, such as location or sensor services. The applications can bind against these services to get information about the current phone location and the status of the phone sensors, respectively.
- **Hybrid Android services**: Hybrid services are both bound and started services.

The lifecycle of Android services is a lot simpler than that of standard applications, as depicted in the following figure. In this figure, the blue methods are shared by two kinds of services, whereas the green one is related to the started services. The red ones are related to bind services. These methods are callback methods that are invoked as the service is starting up:



Similar to applications, the services have the `OnCreate()` and `OnDestroy()` method states, which will make sure that the service is properly created and destroyed, respectively. For the started services, the `OnStartCommand()` method is called right after the `OnCreate()` method. The `OnStartCommand()` method is the place to start background tasks. Also, this method returns a `StartCommandResult` object that describes how the service should be restarted in case the Android operating system shuts it down in the extreme need for memory. For the bound services, clients can bind themselves to an already-running service that will trigger the `OnBind()` method of the service. In the same way, when the clients unbind, they trigger the `Unbind()` method. Both the services can be destroyed by calling a `StopService()` or `StopSelf()` method.

Implementing a started service

In this recipe, we will create and explain all the necessary code to implement a started service. A started service is a service that performs heavy tasks in the background and doesn't offer the possibility to communicate with it through an interface.

Getting ready

Create a new solution named **Services** and open it.

How to do it...

We will now see how to implement a service:

1. Create a new `C#` class named `MyService.cs`.
2. Add `using Android.App`, `using Android.Util`, and `using System.Threading` at the beginning of the class.
3. Specify that your class extends the `Service` class, and use the `Service` attribute, shown as follows:

```
using System;
using Android.App;
using Android.Util;
using System.Threading;

namespace Services {
  [Service]
  public class MyService : Service {

  }
}
```

4. Create a method named `HeavyBackgroundWork()` that starts a new thread. This thread will sleep for 1 second and then log into the **Heavy work completed** option in the console to stop the service using the `StopSelf()` method, shown as follows:

```
private void HeavyBackgroundWork() {
  var heavyWorkThread = newThread(()=> {

    Thread.Sleep (1000);
    //Sleeps for 1 s
    Log.Debug ("MyService","Heavy work completed");
    StopSelf ();
    //Stop (and destroy) the service
  });

  heavyWorkThread.Start();//Start the thread
}
```

5. Override the `OnStartCommand()` method and make a call to the `HeavyBackgroundWork()` method. Also, the `StartCommandResult` instance must return a `StartCommandResult` object, shown as follows:

```
public override StartCommandResult OnStartCommand
(Android.Content.Intentintent,StartCommandFlagsflags,
intstartId) {
  HeavyBackgroundWork();
```

```
    return StartCommandResult.Sticky;
    //Determines how the service should be restarted
  }
```

6. In the `MainActivity` class of the application, created by Xamarin Studio at the same time as the project, add a call to the `StartService()` method in the `OnCreate()` method:

```
namespace Services {
  [Activity (Label = "Services", MainLauncher = true)]
  public class MainActivity : Activity {
    protected override void OnCreate (Bundlebundle) {
      base.OnCreate (bundle);

      SetContentView (Resource.Layout.Main);

      StartService (new Intent (this, typeof(MyService)));
    }
  }
}
```

7. Run the application, and the **[MyService] Heavy work completed** line will appear in your console. This means that the service has been successfully started, and the thread created in the `HeavyBackgroundWork` has waited for 1 second to destroy itself.

How it works...

In the previous section, we saw a lot of new instructions and concepts. We will now explain them in depth one by one. First of all, we have the `[Service]` annotation that we have placed before the declaration of our `MyService` class. Second, `MyService` extends `Service` class. This will create an entry in the `AndroidManifest.xml` file in the same way as `[Activity]` for activities. We can avoid this, and write the following directly in the XML file:

```
<service android:name="services.MyService"></service>
```

The next element that deserves an explanation is the following line, which we have added as the return statement of the overridden `OnStartCommand()` method:

```
returnStartCommandResult.Sticky;
//Determineshowtheservicesshouldberestarted
```

From the introduction of this chapter, we know that the `OnStartCommand()` method will return a `StartCommandResult` object, which specifies how the service should be restarted, in case it was destroyed by the OS in an attempt to free some memory. From the code, we can deduce that the `StartCommandResult` object is an enumerator, but we don't know what the `Sticky` option refers to. The `Sticky` option is one of the three different behaviors that we can give to the service in case of a restart. The possible values for the `StartCommandResult` object are:

- **Sticky**: A sticky service will be restarted, and a `null` intent will be delivered to the `OnStartCommand()` method at the restart. This is used when the service continuously performs a long-running operation, such as updating a stock feed.
- **RedeliverIntent**: This service is restarted, and the last intent that was delivered to the `OnStartCommand()` method before the service was stopped gets redelivered. This is used to continue a long-running command, such as the completion of a large file upload.
- **NotSticky**: The service is not automatically restarted. **StickyCompatibility**—restart will behave as a Sticky on an API of level 5 or greater, but will downgrade to prelevel 5 behavior on earlier versions.

There's more...

According to the Android documentation (`http://developer.android.com/reference/android/content/IntentFilter.html`), intent filters are structured descriptions of intent values to be matched. An intent filter can match against actions, categories, and data (either via its type, scheme, and/or path) in an intent. It also includes a "priority" value that is used to order multiple matching filters. In other words, we can define a string representing an intent and associate it with a class. The string is extra data for the intent that helps Android figure out what service to start. With the service that we created earlier, it will look like the following:

```
[Service]
[IntentFilter(newString[]{"ca.services.MyService"})]
publicclassMyService : Service
```

Then, we can refer to the `ca.services.MyService` argument to create `MyService`, shown as follows:

```
Intent MyIntent = newIntent ("ca.services.MyService");

StartService (MyIntent);
StopService (MyIntent);
```

Instead of:

```
StartService (new Intent (this, typeof(MyService)));
```

See also

See also the next recipe to see how to create services that offer a way to communicate to services.

Implementing a bound service

Bound services are services that allow users to bind themselves to them and ask for information. The examples of bound services can be seen in the location, and sensor services are natively run on modern Android phones. The principal addition of the bound service in comparison to the started service is its ability to accept clients' requests. The clients bind themselves to services using an `IBinder` instance. In this recipe, we will see how to implement a bound service and its `Binder`.

Getting ready

To follow this recipe, create a new project in the services solution that we created earlier and name it `BoundService`.

How to do it...

Let's have a look at steps required to implement a bound service:

1. Create a new class in the `BoundService` project and name it `MyBoundService`. As this class is the service, we have to add the `[Service]`(and subclass `Service`) attribute and the `OnStartCommand()` method, as seen in the previous recipe. Additionally, `MyBoundService` contains a reference to an `IBinder` object; override the `OnBind()` method, shown as follows:

   ```
   [Service]
   [IntentFilter(new
   String[]{"ca.services.MyBoundService"})]
   ```

```
public class MyBoundService : Service {
  IBinder _myBinder = null;

  public String SayHello() {
    return "HelloWorld";
  }

  public override StartCommandResult OnStartCommand
  (Android.Content.Intentintent, StartCommandFlags
  flags, intstartId) {
    Log.Debug ("MyBoundService", "StartCommandResult");
    this._myBinder = newMyBoundServiceBinder (this);
    return StartCommandResult.NotSticky;
  }

  public override IBinder OnBind (Intent intent) {
    Log.Debug ("MyBoundService", "NewClient");
    return _myBinder;
  }
}
```

2. Create a new class named `MyBoundServiceBinder` that extends the `Binder` class and owns a reference to `MyBoundService`. The constructor of `MyBoundServiceBinder` takes `MyBoundService` as an argument and populates the private reference with it. Finally, `MyBoundServiceBinder` has a getter that returns the `MyBoundService` reference:

```
public class MyBoundServiceBinder : Binder {
  private MyBoundService _myBoundService;

  public MyBoundService BoundService {
    get{return _myBoundService;}
  }

  public MyBoundServiceBinder (MyBoundService
  _myBoundService) {
    this._myBoundService = _myBoundService;
    Log.Debug ("MyBoundServiceBinder", "NewBinder");
  }
}
```

3. Add a reference to `MyBoundServiceBinder` in your main activity the same as an `IsBound` Boolean. Also, add a `consumerService()` method, shown as follows:

```
[Activity(Label="BoundService",MainLauncher=true)]
public class MainActivity : Activity {
  private MyBoundServiceBinder _myBinder;
  private Boolean _IsBound;

  public MyBoundServiceBinder Binder {
    get{ return_myBinder; }
    set{ _myBinder=value; }
  }

  public Boolean IsBound {
    get{ return_IsBound; }
    set{ _IsBound=value; }
  }

  public void ConsumeService() {
    Log.Debug ("MainActivity",
    _myBinder.BoundService.SayHello ());
  }
}
```

4. Create a new class named `MyBoundServiceConnection` that implements the `IServiceConnection` interface and owns a reference to `MyBoundService`. The `MyBoundServiceConnection` class implements the `OnServiceConnected()` and `OnServiceDisconnected()` methods belonging to the `IServiceConnection` interface and contains a reference to the main activity:

```
publicclassMyBoundServiceConnection:Java.Lang.Object,
IServiceConnection {
  MainActivity _myActivity;

  public MyBoundServiceConnection (MainActivity activity) {
    this._myActivity = activity;
  }

  public void OnServiceConnected (ComponentName name,
  IBinder service) {
    Log.Debug ("MyBoundServiceConnection", "Incoming
    Connection");
    var boundServiceBinder = service as
    MyBoundServiceBinder;
    if (boundServiceBinder != null) {
```

```
        _myActivity.Binder = boundServiceBinder;
        _myActivity.isBinded = true;
        _myActivity.consumeService ();
      }
    }

    public void OnServiceDisconnected (ComponentName name) {
      _myActivity.isBinded = false;
    }
  }
```

5. Bind your `MainActivity` to the `MyBoundService` service using the `BindService()` method that takes `Intent` as arguments, an implementation of `IServiceConnection`, and a `Bind` flag:

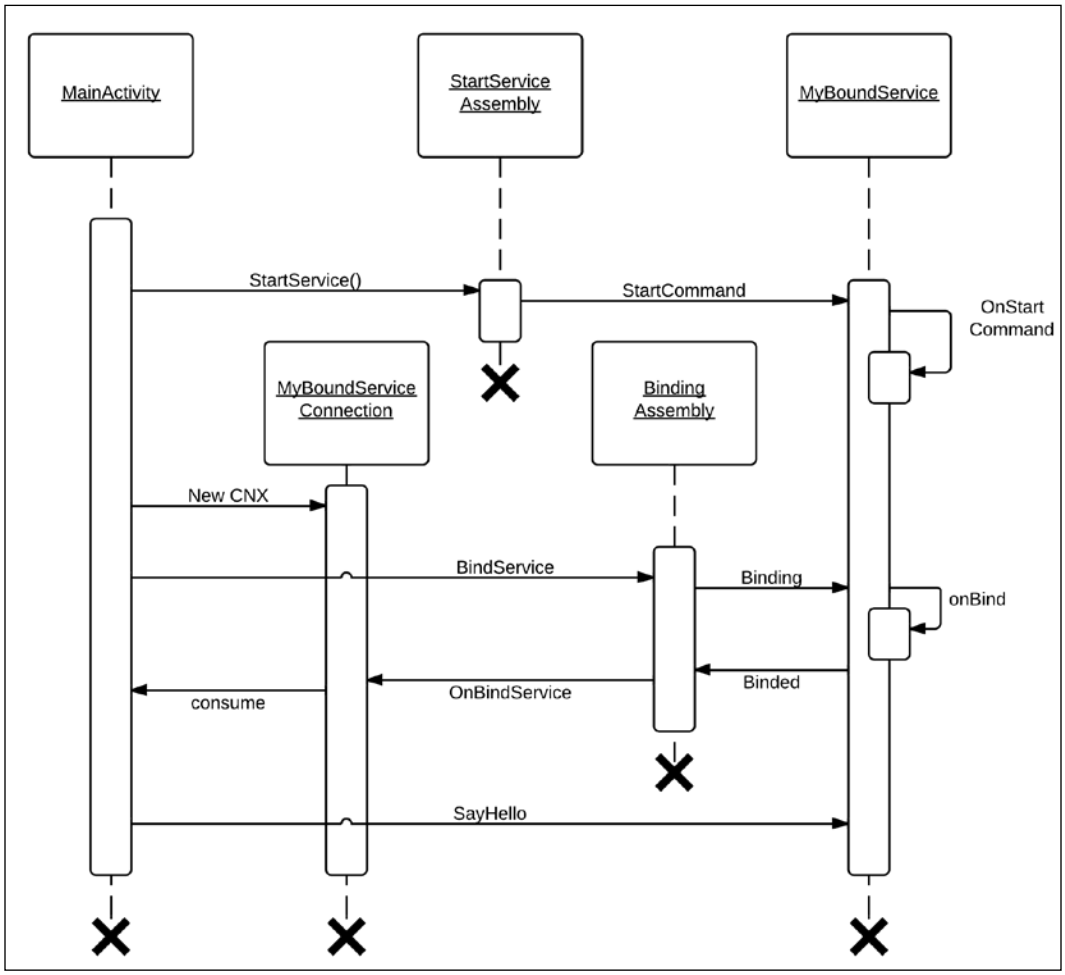
```
protected override void OnCreate(Bundle bundle) {
  base.OnCreate (bundle);

  Intent myBoundServiceIntent = newIntent
  ("ca.services.MyBoundService");
  StartService (myBoundServiceIntent);

  cnx = new MyBoundServiceConnection (this);
  BindService (myBoundServiceIntent, cnx, Bind.AutoCreate);

}
```

6. Run your application. The following lines will appear in the console:

```
[MyBoundService] StartCommandResult
[MyBoundServiceBinder] New Binder
[MyBoundService] New Client
[MyBoundServiceConnection] Incoming Connection
[MainActivity] Hello World
```

How it works...

In order to understand how it works, we first need to understand the sequence in which the objects interact with each other. The following figure depicts a simplified UML sequence diagram of the collaborations between the objects:



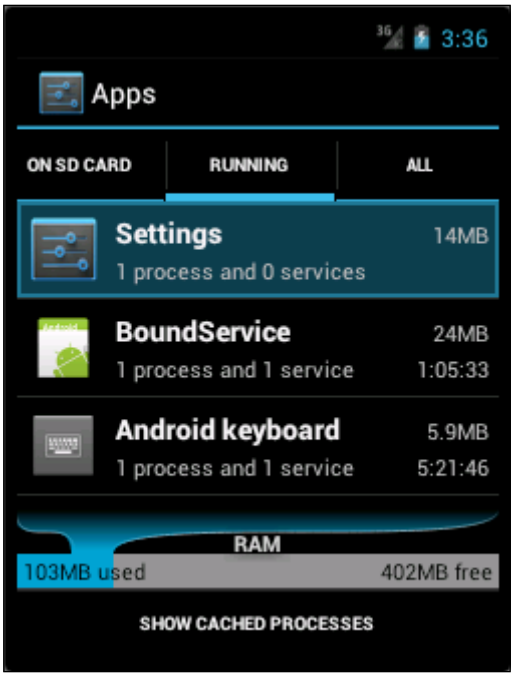
First, in the `OnCreate()` method, we make a call to the `StartService()` method in the `StartService` assembly. This assembly holds code and Xamarin.Android will take care of creating `MyBoundService` and triggering a call to the `OnStartCommand()` method. Note that there is no message from `MyBoundService` to `MainActivity` to inform `MainActivity` that `MyBoundService` has started correctly. Indeed, the `StartService()` method triggers the start and doesn't wait for an acknowledgment from `MyBoundService`. Then, `MainActivity` obtains a new `MyBoundServiceConnection` and asks the `BindingAssembly` to bind the `MyBoundService` service. The `BindingAssembly` will trigger the `OnBind()` method of `MyBoundService` and the `OnBindService()` method of `MyBoundServiceConnection` when the binding gets completed. Finally, `MyBoundServiceConnection` will invoke the `ConsumeService()` method, and the `MainActivity` class will consume `MyBoundService.SayHello()` method.

You may wonder why we don't call `MyBoundService.SayHello()` method directly after `BindService` and remove the `ConsumeService()` method? The response is simple, we don't know how long it will take for the service to bound effectively, and `MyBoundServiceBinder` may be `null` if we don't wait long enough. We can have an infinite `while(!IsBound){}` in order to wait for the Boolean to be `true`. However, the current solution avoids a potential infinite loop and stays simple by avoiding the use of Thread wait and join.

To summarize, clients use `MyBoundServiceBinder` to obtain a reference of `MyBoundService`, and call a public method on the service. `MyBoundServiceBinder` gets populated after the asynchronous call to the `BindingAssembly` that triggers the `OnBind()` and `OnBindService()` methods. Once the `OnBindService()` method has been executed, we can be sure that the reference to `MyBoundServiceBinder` has been populated, and we can start using the service.

There's more...

If you want to be sure that your application is running your services, you can simply access the running application on your phone (or emulator) under **Settings** | **Apps** | **Running**, and check how many services are running. In the following picture, we can see one service and one process for the `BoundService` application:



See also

See also the next section to learn how to send notifications from Android services.

Send notifications from your service

In this short recipe, you will learn how to send notifications from the services that we created earlier. Our objective will be to create an application where the call to an Android-bound service is done manually through a button. The notification will inform the user that the service is correctly bound, and the call can safely be made.

Getting ready

For this recipe, we will reuse the code from the recipe related to the bound service.

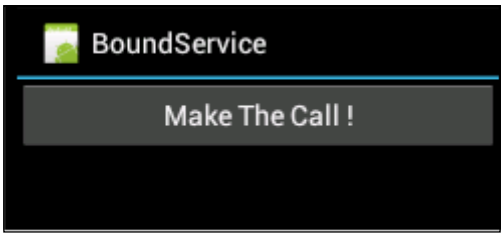
How to do it...

Let's take a look at these steps:

1. Add a button containing the **Make The Call** label and an empty `TextView` to the `MainActivity` class. The `Button.Click` event will trigger a call to the `MyBoundService` service from the previous recipe and populate the `TextView` with the answer. The `MainActivity` class's `OnCreate()` method now reads as follows:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);

  SetContentView (Resource.Layout.Main);

  Intent myBoundServiceIntent = new Intent
  ("ca.services.MyBoundService");
  StartService (myBoundServiceIntent);

  cnx = new MyBoundServiceConnection (this);
  BindService (myBoundServiceIntent, cnx, Bind.AutoCreate);

  Button button = FindViewById<Button>
  (Resource.Id.myButton);

  button.Click += delegate {
    TextView tx = FindViewById<TextView>
    (Resource.Id.textView1);
    tx.Text = _myBinder.BoundService.SayHello();
  };
}
```

The application is as depicted in the following screenshot:

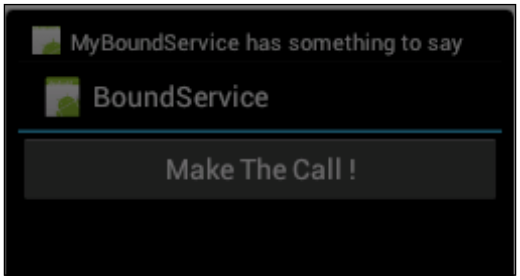


2. Remove the call to the consume method of the `MainActivity` class in the `MyBoundServiceConnection.OnServiceConnected()` method, and remove the consume from the `MainActivity` class. Consequently, the `MyBoundServiceConnection.OnServiceConnected()` method is as follows:

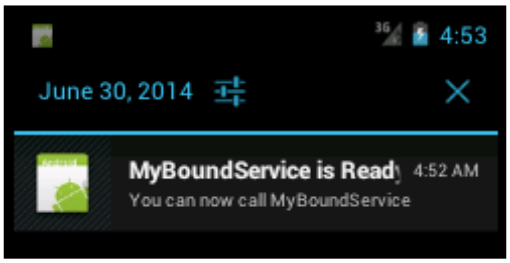
```
public override IBinder OnBind (Intent intent) {
  Log.Debug ("MyBoundService", "NewClient");

  varn Mgr = (NotificationManager) GetSystemService
  (NotificationService);

  Notification.Builder builder = new Notification.Builder
  (this)
  .SetContentTitle ("MyBoundService is Ready")
  .SetTicker ("MyBoundService has something to say")
  .SetContentText ("You can now call MyBoundService")
  .SetSmallIcon (Resource.Drawable.Icon);

  nMgr.Notify (0, builder.Notification);

  return _myBinder;
}
```

3. Run your application.

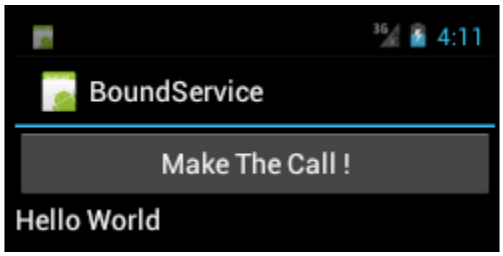
   A few moments after the application first appears, you will receive a notification in the notification area, as shown in the following screenshot:



If the user opens the notification he/she can see it entirely:



4. Click on the **Make The Call** button:



How it works...

The modifications performed in order to have the notification are all contained in the following code:

```
var nMgr = (NotificationManager) GetSystemService
(NotificationService);

Notification.Builder builder = new Notification.Builder (this)
.SetContentTitle ("MyBoundService is Ready")
.SetTicker ("MyBoundService has something to say")
.SetContentText ("You can now call MyBoundService")
.SetSmallIcon (Resource.Drawable.Icon);

nMgr.Notify (0,builder.Notification);
```

In this code, we use a lot of new objects. The first one is the `NotificationManager` instance. As its name suggests, `NotificationManager` is a class that notifies the user in case events happen in the background, in other words, in services. Different types of notifications can be sent to the user, for example, an icon and text in the launch bar, turning the notification led, or more intrusively, firing up the back light, playing a sound or vibrating. The kind of notification will depend on the method invoked on the builder. Here, we used a simple notification in the notification area:

```
var notification = new Notification (Resource.Drawable.Icon,
"MyBoundService is Ready")
```

The arguments for the notification constructor are the icon of the application `Resource.Drawable.Icon`, and the sentence `"MyBoundService is Ready"`.

Then, we use a `Notification.Builder` instance:

```
Notification.Builder builder = new Notification.Builder (this)
.SetContentTitle ("MyBoundService is Ready")
.SetTicker ("MyBoundService has something to say")
.SetContentText ("You can now call MyBoundService")
.SetSmallIcon (Resource.Drawable.Icon);
```

The `Notification.Builder` is a helper class used to build the notification. Using this helper class, you can set the title, content, icons, and all the other ways of notifying a user you can imagine. Finally, we send the notification using the notification manager:

```
nMgr.Notify (0, builder.Build());
```

In the previous line, `0` is the tag of the notification that may be used later on.

See also

Take a look at the next section to see how to create a service that can periodically execute an action and notify the user.

Creating a news feed service

In this practical example, we will put together all the knowledge acquired during the first three recipes in order to create a news feed service. This news feed service will first populate a `ListView` element of news, and then will periodically check if fresh news is available regardless of whether the activity is focused or not.

Getting ready

To follow this recipe, create a new project named `NewsFeedService`.

How to do it...

1. Create a layout named `Welcome.xml` and update it to the following:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <ListView
    android:id="@+id/feedItemsListView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
</LinearLayout>
```

2. Update the `Main` layout to the following (see *Chapter 6*, *Populating Your GUI with Data*, for the details):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="100dp"
  android:gravity="center_vertical"
  android:orientation="horizontal">
  <LinearLayout
    android:layout_width="100dp"
    android:layout_height="fill_parent"
    android:gravity="center">
  <ImageView
    android:id="@+id/image"
    android:layout_width="80dp"
    android:layout_height="80dp"
    android:gravity="center"
    android:src="@drawable/Icon"
    android:contentDescription=""
    android:layout_gravity="center"/>
  </LinearLayout>
  <LinearLayout
    android:layout_width="0dip"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:orientation="vertical"
    android:gravity="center_vertical">
  <TextView
```

```
    android:id="@+id/title"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Title"
    android:textColor="#ffffff"
    android:textSize="16dp"
    android:gravity="top"
    android:textStyle="bold"/>
  <TextView
    android:id="@+id/creator"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Author"
    android:textSize="12dp"
    android:textColor="#808080"
    android:layout_marginTop="5dp"
    android:layout_marginBottom="5dp"/>
  <TextView
    android:id="@+id/pubDate"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Date"
    android:textSize="12dp"
    android:padding="2dp"/>
  </LinearLayout>
</LinearLayout>
```

3. In the `MainActivity` class, update the method `OnCreate()` in order to bind to a service using an intent filter `"ca.services.NewsService"` and display a progress dialog. The method `OnStart()` populates the `ListView` instance of news using the `NewsParser.ParseMethod()` method, and it owns a static method to refresh the list view:

```
namespace NewsFeedService {
  [Activity (Label = "NewsFeedService", MainLauncher=
  true)]
  public class MainActivity : Activity {
    //private variables
    private NewsServiceBinder _myBinder;
    private Boolean _isBound;
    private ProgressDialog _progressDialog;
    private static List<NewsItem> news = new
    List<NewsItem>();
    private ListView newsItemsListView;
```

```
NewsFeedServiceConnection cnx;

//getter and setter
public NewsServiceBinder Binder {
  get { return_myBinder; }
  set { _myBinder=value; }
}

public Boolean IsBound {
  get { return_isBound;}
  set { _isBound=value; }
}

protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);

  //Set our view from the "Welcome" layout resource
  SetContentView(Resource.Layout.Welcome);
  newsItemsListView = this.FindViewById<ListView>
  (Resource.Id.feedItemsListView);

  //Start and bind to the service
  Intent myServiceIntent = new Intent
  ("ca.services.NewsService");
  StartService (myServiceIntent);
  cnx = new NewsFeedServiceConnection (this);
  BindService (myServiceIntent, cnx, Bind.AutoCreate);

  //Display a progress dialog
  this.progressDialog = newProgressDialog(this);
  this.progressDialog.SetMessage("PleaseWait...");
  this.progressDialog.Show();
}

protected override void OnStart() {
  base.OnStart ();
  this.BootStrap ();
  this.progressDialog.Dismiss ();
}

public void boostrap() {

  Log.Debug ("Main", "bootstrap");
```

```
            news = NewsParser.parse ();

            //Populate the adapter with the results from the
            parser
            var adapter = new NewsAdapter(this, news);
            newsItemsListView.Adapter = adapter;
            newsItemsListView.ItemClick += OnListViewItemClick;
            Log.Debug ("Main", "bootstrapend");
          }

          //accepts new items
          public static void Refresh(List<NewsItem> fresh_news) {
            news = fresh_news;
          }

          protected void OnListViewItemClick(object sender,
          AdapterView.ItemClickEventArgs e) {
            //Do Something
            var t = news[e.Position];
            Android.Widget.Toast.MakeText(this, t.Link,
            Android.Widget.ToastLength.Short).Show();
          }

        }
      }
```

4. Create a `NewsItem` class to represent the news:

```
public class NewsItem {

  public NewsItem() {
  }

  public string Title { get;set; }

  public string Link { get;set; }

  public DateTime PubDate { get;set; }

  public string Creator { get;set; }

  public string Category { get;set; }

  public string Description { get;set; }

  public string Content { get;set; }

}
```

5. Create `NewsAdapter` to map `NewsItem` into `ListView`:

```
namespace NewsFeedService {
  //Details in Chapter - Populating your GUI with Data
  public class NewsAdapter : BaseAdapter<NewsItem> {
    protected Activity context = null;
    protected List<NewsItem> feedsList = new
    List<NewsItem>();

    public NewsAdapter (Activity context, List<NewsItem>
    feedsList)
    : base() {
      this.context = context;
      this.feedsList = feedsList;
    }

    public override NewsItem this[int position] {
      get { return this.feedsList[position]; }
    }

    public override long GetItemId(int position) {
      return position;
    }

    public override int Count {
      get { return this.feedsList.Count; }
    }

    public override View GetView(int position, View
    convertView, ViewGroup parent) {
      var feedItem = this.feedsList[position];

      var view = (convertView ?? context.LayoutInflater.
      Inflate(Resource.Layout.Main, parent, false))as
      LinearLayout;

      //Cut the title if too long
      view.FindViewById<TextView>(Resource.Id.title).Text =
      feedItem.Title.Length<51?feedItem.Title :
      feedItem.Title.Substring(0, 53)+"...";
      view.FindViewById<TextView>(Resource.Id.creator).Text
      = feedItem.Creator;
      view.FindViewById<TextView>(Resource.Id.pubDate).Text
      = feedItem.PubDate.ToString("dd/MM/yyyyHH:mm");
      return view;
    }
  }
}
```

6. Create a `NewsParser` class to parse an `rss` feed and retrieve the news:

```
namespace NewsFeedService {
  public class NewsParser {
    //Private variables
    private static List<NewsItem> news = new
    List<NewsItem>();
    private static DateTime last = System.DateTime.Now;

    //Check if a fresh news is available
    public static Boolean CheckItem(DateTime date) {
      //XMl Document
      XmlDocument xmlDocument = new XmlDocument();
      Stream stream = getStream();
      xmlDocument.Load (stream);

      //Retrieve the nodes
      XmlNodeList itemNodes =
      xmlDocument.SelectNodes("rss/channel/item");
      XmlNamespaceManager nsmgr = new
      XmlNamespaceManager(xmlDocument.NameTable);
      nsmgr.AddNamespace("dc", xmlDocument.DocumentElement.
      GetNamespaceOfPrefix("dc"));
      nsmgr.AddNamespace("content", xmlDocument.
      DocumentElement.GetNamespaceOfPrefix("content"));

      //If the first news is newer than the last batch
      if (System.DateTime.Compare (Convert.ToDateTime
      (itemNodes [0].SelectSingleNode ("pubDate")),last)<
      0) {
        parse (stream);
        return true;
      }
      else {
        return false;
      }

    }

    //Construct a web request and provide a stream of data
    out of it
    private static Stream GetStream() {

      WebRequest webRequest = WebRequest.Create
      ("http://feeds.feedburner.com/canadiannews");
```

```
        WebResponse webResponse = webRequest.GetResponse();

        return webResponse.GetResponseStream();
      }

      public static List<NewsItem> Parse(Stream stream =
      null) {
        XmlDocument xmlDocument = new XmlDocument();

        if (stream == null) {
          stream = getStream();
        }

        //Assign the variable of NewsItem with the data of
        the stream.
        for (int i = 0; i < itemNodes.Count; i++) {
          NewsItem newsItem = new NewsItem();

          if (itemNodes[i].SelectSingleNode("title") != null)
          {
            newsItem.Title = itemNodes[i].SelectSingleNode
            ("title").InnerText;
          }

          if (itemNodes[i].SelectSingleNode("link") != null)
          {
            newsItem.Link = itemNodes[i].SelectSingleNode
            ("link").InnerText;
          }

          [……]

          newsItem.Content = itemNodes[i].SelectSingleNode
          ("content:encoded", nsmgr).InnerText;
        }
        else {
          newsItem.Content = newsItem.Description;
        }

        news.Add(newsItem);
      }

    return news;

    }
  }
```

7. Create a `NewsService` class:

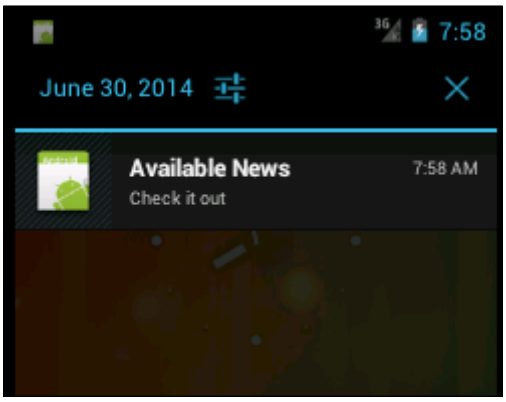
```
namespace NewsFeedService {
  [Service]
  [IntentFilter(newString[]{"ca.services.NewsService"})]
  public class NewsService : Service {

    IBinder _myBinder = null;

    //invoked on Start of the service
    public override StartCommandResult OnStartCommand
    (Android.Content.Intentintent, StartCommandFlags
    flags, int startId) {
      Log.Debug ("NewsFeedService", "StartCommandResult");
      this._myBinder = new NewsServiceBinder (this);
      check_update ();
      return StartCommandResult.Sticky;
    }

    public void CheckUpdate() {
      //Recursivly check (every5seconds) if a fresh news
      need to be displayed
      Thread t = newThread(()=> {
        Thread.Sleep (5000);
        if (NewsParser.check_item(System.DateTime.Now)) {

          var nMgr = (NotificationManager) GetSystemService
          (NotificationService);

          //Details of notification in previous recipe
          Notification.Builderbuilder = new
          Notification.Builder (this)
          .SetContentTitle ("AvailableNews")
          .SetTicker ("AvailableNews")
          .SetContentText ("Checkitout")
          .SetSmallIcon (Resource.Drawable.Icon);

          nMgr.Notify (0, builder.Notification);
        }
      });
      t.Start();
    }

    public override IBinder OnBind (Intent intent) {
      Log.Debug ("NewsFeedService", "NewClient");
```

```
      return _myBinder;
    }

  }

}
```

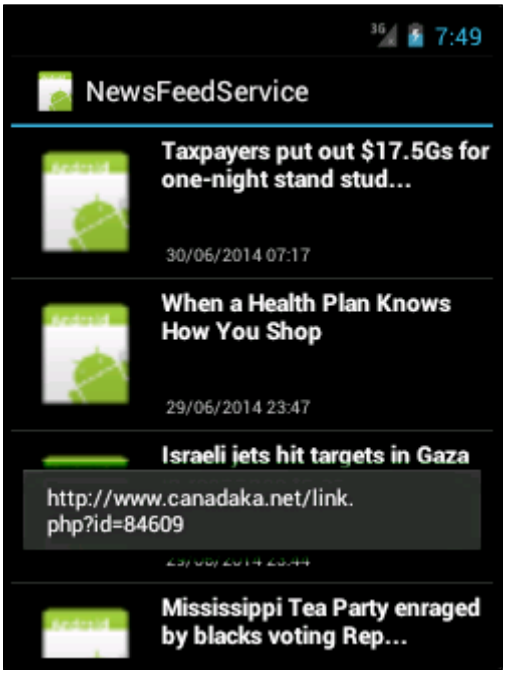
8. Create a `NewsServiceBinder` and a `NewsServiceConnection`:

```
public class NewsServiceBinder : Binder {
  private NewsService _myBoundService;

  public NewsService NewService {
    get{ return_myBoundService; }
  }

  public NewsServiceBinder (NewsService _myBoundService) {
    this._myBoundService = _myBoundService;
    Log.Debug ("MyBoundServiceBinder", "NewBinder");
  }
}

public class NewsFeedServiceConnection : Java.Lang.Object,
IServiceConnection {
  MainActivity _myActivity;

  public NewsFeedServiceConnection (MainActivity activity)
  {
    this._myActivity = activity;
  }

  public void OnServiceConnected (ComponentName name,
  IBinder service) {
    Log.Debug ("NewsFeedServiceConnection", "Incoming
    Connection");
    var boundServiceBinder = service as NewsServiceBinder;
    if (boundServiceBinder != null) {
      _myActivity.Binder = boundServiceBinder;
      _myActivity.isBinded = true;
      _myActivity.boostrap ();
    }
  }

  public void OnServiceDisconnected (ComponentName name) {
    _myActivity.isBinded = false;
  }
}
```

9. Start your application. You'll receive a notification, shown as follows:



10. Open the notification, and the news will be displayed:



How it works...

The `NewsService` instance is a combination of what you learned so far. Therefore, we don't have any technical explanation here. However, the sequence of calls and the orchestration between the numerous objects requires some details.

The `NewsService` project displays a custom `ListView` of `NewsItem` (see *Chapter 6*, *Populating Your GUI with Data*) for a reminder. This `Listview` element is first populated with the construction of a web request, which allows us to create an XML document. Every child of the document will be parsed by `NewsParser` and will be transformed into a `NewsItem`. After this starting phase, `NewsService`, which is bound to the `MainActivity` via `NewsServiceBinder` and `NewsServiceConnection`, really kicks in. Indeed, the `NewsService` project owns an infinite loop based on a thread that will check, every 50 seconds, whether fresh news is available:

```
public void check_update() {
  //Recursivly check (every 5 seconds) if a fresh news need
  to be displayed
  Thread t = newThread(()=> {
    Thread.Sleep (5000);
    if (NewsParser.check_item(System.DateTime.Now)) {

      var nMgr = (NotificationManager) GetSystemService
      (NotificationService);

      //Details of notification in previous recipe
      Notification.Builderbuilder = new
      Notification.Builder (this)
      .SetContentTitle ("AvailableNews")
      .SetTicker ("AvailableNews")
      .SetContentText ("Checkitout")
      .SetSmallIcon (Resource.Drawable.Icon);

      nMgr.Notify (0, builder.Notification);
    }
  });
  t.Start();
}
```

If so, the `NewsParser` instance will send a notification to the user. The `ParseNews` class will parse the news again and update the `ListView` of the `MainActivity` class.

See also

See the next chapter to learn how to use intents.

8
Mastering Intents – A Walkthrough

In this chapter, you will learn the details behind Intent implementation and use with the following recipes:

- Opening external applications
- Monitoring time
- Application monitoring
- Solving equations
- Sending an SMS

Introduction

In the previous chapter, that is, *Chapter 7*, *Using Android Services*, we used Intents to communicate with background services and didn't pay too much attention to them. Android Intent is a component of the Android OS responsible for triggering an operation. While Ruby developers tend to master this concept, it's a little bit trickier for the rest of us.

In the previous chapter, we used Intent to communicate with background services as follows:

```
StartService (new Intent (this, typeof(MyService)));
```

And even in more complicated ways like this:

```
[Service]
[IntentFilter(new String[]{"ca.services.MyService"})]
public class MyService : Service
Intent MyIntent = new Intent ("ca.services.MyService");
StartService (MyIntent);
```

However, we didn't put much thought into this and barely used them as a convenient way to communicate with our background services. Android Intents are, from my point of view, a useful UFO in the Android ecosystem. Indeed, the Intent object is a concept that allows us to abstractly describe operations that we intend to do in an asynchronous way. Concretely, Intents are often used to launch an external application such as maps, phone, or messages. Moreover, we can use an Intent for more powerful purposes such as browsing through the contact list or interacting with e-mails.

Opening external applications

In this recipe, we will create and explain all the code that is necessary to open external applications with Intents.

Getting ready

Create a new solution named `Intent_Project` and open the project of the same name that Xamarin Studio has created for you.

How to do it...

1. Add the following code in the `OnCreate()` method to the `MainActivity` class so that it looks like the following:

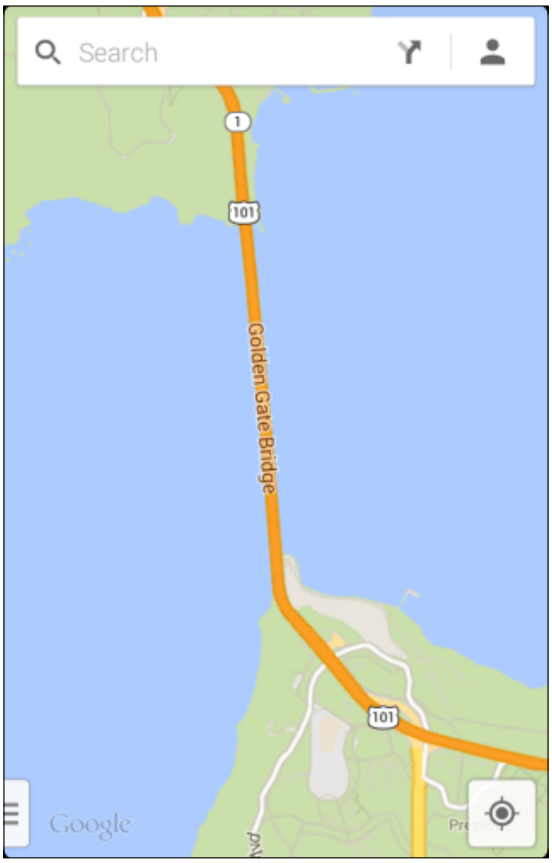
```
using System;

using Android.App;
using Android.Content;
using Android.Runtime;
using Android.Views;
using Android.Widget;
using Android.OS;

namespace intent_project {
```

```
[Activity (Label = "intent_project", MainLauncher =
true)]
public class MainActivity : Activity {
  protected override void OnCreate (Bundle bundle) {
    base.OnCreate (bundle);

    var geoUri = Android.Net.Uri.Parse ("geo:37.8143849,-
    122.4719223");
    var mapIntent = new Intent (Intent.ActionView,
    geoUri);
    StartActivity (mapIntent);

  }
 }
}
```

2. Run your application.
3. If you're running your code on the emulator, then the following exception should appear:

Note that this exception won't occur if you have an application that can handle the Intent on your emulator. However, it is unlikely unless you have installed additional applications.

4. If you're on a real device, the Maps application should open as follows:



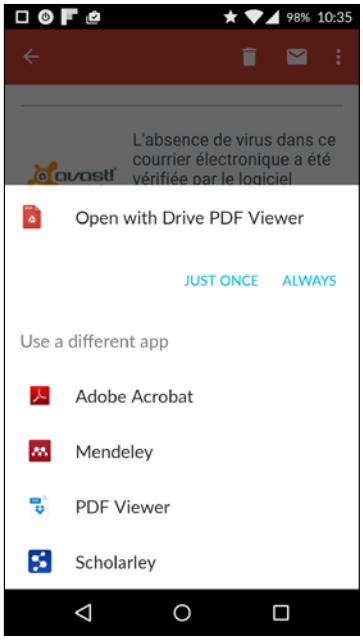
How it works...

First of all, let's take a look at the little code snippet that we've written in the previous section:

```
var geoUri = Android.Net.Uri.Parse ("geo:37.8143849,-
122.4719223");
  var mapIntent = new Intent (Intent.ActionView, geoUri);
  StartActivity (mapIntent);
```

With this code, we created an URI out of `"geo:37.8143849,-122.4719223"` and then an Intent with two arguments: `Intent.ActionView` and the `geoUri`. Finally, we started an Activity and sent the intent. Here, we asked the Android OS to perform a view operation of a URI containing the geographic coordinates. The OS is responsible for mapping this request to the appropriate application: Google Maps or any application capable of handling coordinates. However, if no application can handle what we ask, then the application we have created will yield an exception. On one hand, we abstractly describe what we expect from the OS and what action we intend to take, but on the other hand, this abstraction has some limitations as we are not sure that an app can perform this operation.

If several applications are able to handle the request, the user will be prompted with a dialog to choose which one they want to use, as displayed in the following example screenshot where several applications can open a PDF attachment:



In the same fashion as the `ActionView` element, the `Intent` class proposes (at the time of writing this), 114 other actions that you can use. The most common actions are as follows:

- `CALL`: This performs a call to the contact that is specified in the data.
- `CALL_BUTTON`: The user can press the Call button to go to the dialer or other appropriate UI for placing a call
- `EDIT`: This provides explicit editable access to the given data.
- `OPEN_DOCUMENT`: This allows the user to select and return one or more existing documents
- `DIAL`: This dials a number as specified by the data
- `GET_CONTENT`: This allows the user to select a particular kind of data and return it
- `SEND`: This delivers some data to someone
- `SENDTO`: This sends a message to the contact specified in the data
- `VIEW`: This displays the data to the user
- `WEB_SEARCH`: This performs a web search

The way in which the OS actually handles the intents requests, fold in the Android system programming and isn't within the scope of this book

There's more...

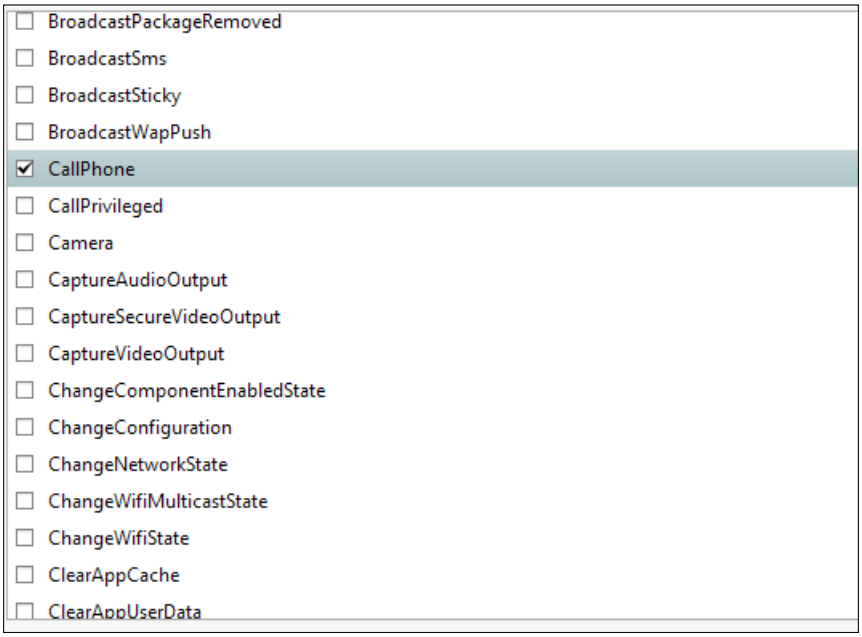
Let's look at how to dial a number and open a web page using Intents.

Dialing a number

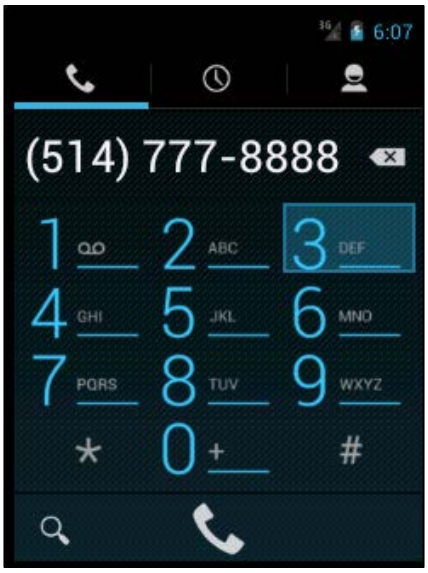
```
var phoneURI = Android.Net.Uri.Parse ("tel:5147778888");

var phoneIntent = new Intent (Intent.ActionView, phoneURI);

StartActivity (phoneIntent);
```

In the previous code sample, I renamed the variable name to keep the code coherent, but the only true change is the string passed to `Android.Net.Uri.Parse`, which we changed from `"geo:37.8143849,-122.4719223"` to `"tel:5147778888"`. Pretty straightforward, isn't it?

Nonetheless, dialing on behalf of the user requires the `CALL_PHONE` permission. In order to allow your application to do so, tick the `CallPhone` checkbox on the `Properties/AndroidManifest.xml` file as shown in the following screenshot:



Next, run your application. You will see the following:



It is noteworthy that the back button of the phone will redirect the user toward the origin of the intent action—your application.

Opening a web page

In order to open a web page using intents, we will have to do the following:

```
var uri = Android.Net.Uri.Parse ("https://www.packtpub.com/");
var intent = new Intent (Intent.ActionView, uri);
StartActivity (intent);
```

On successful completion, the following should appear on your emulator:

In conclusion, pass a string that can be parsed into a URI and the OS will do its best to find a matching application.

See also

Refer to the next recipe to see how to use an Intent to enhance our applications instead of redirecting toward another application. Also, refer to the following URLs for more information:

- `http://developer.android.com/guide/components/intents-filters.html`
- `http://androidapi.xamarin.com/?link=T%3aAndroid.Content.Intent`
- `http://developer.android.com/reference/android/content/Intent.html#ACTION_DIAL`

Monitoring time

For the next four recipes, we will use Intents to build the application imagined by *Vince Vaughn* in the movie *The Internship*. This application is fairly simple and can be summarized as follows. If you send a text message to someone between 12 a.m. and 6 a.m., well, it is probably a bad idea as you might be quite influenced by alcohol. The solution? You have to solve an equation to prove that you're OK!

On a serious note, here's what we will have to do:

1. Check the time and redirect the user from the default text message app to ours if the time is between 12 a.m. and 6 a.m
2. Propose a fairly simple equation for the user to solve
3. Pick a contact in the contact list held on the phone
4. Write a text message
5. Send it to the selected contact

In this recipe, we start with the building of our application by getting the time of the day using a broadcast receiver.

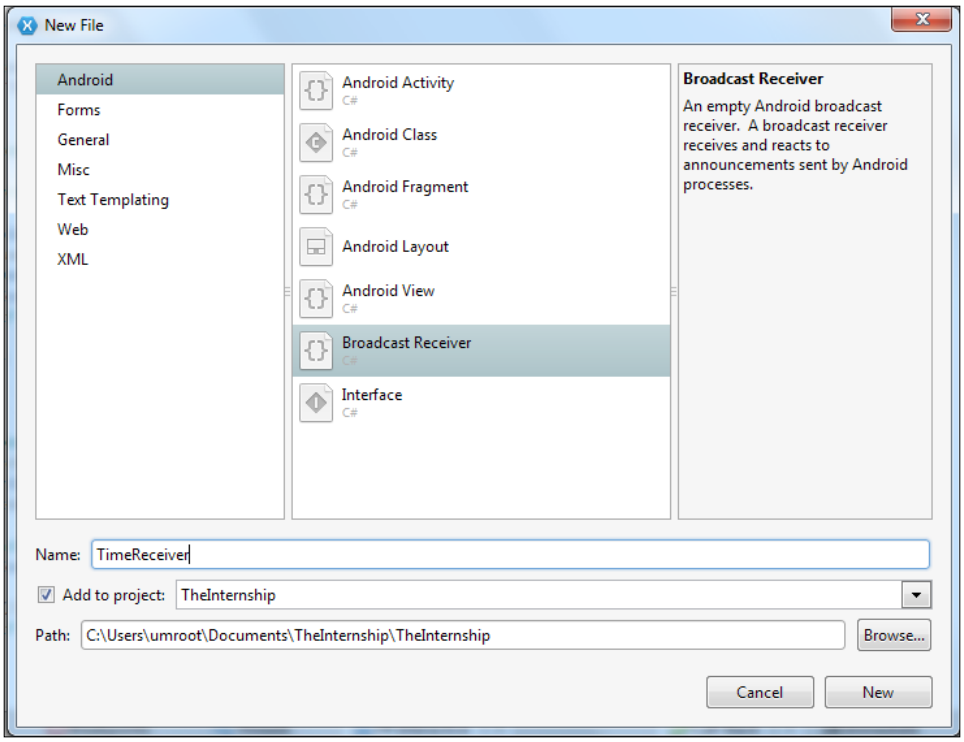
Getting ready

To follow this recipe, create a new project called `TheInternship`.

How to do it...

We will do the time monitoring using the following steps:

1. Press the *Ctrl* + *N* sequence and add a new **Broadcast Receiver** project named **TimeReceiver** to your project.



2. Adapt the created class so that it looks like the following:

```
[BroadcastReceiver]
public class TimeReceiver : BroadcastReceiver {

  public override void OnReceive (Context context, Intent intent)
  {
    Console.WriteLine ("Time Received");
    //If we are in the dangerous timeframe
    if (DateTime.Now.Hour >= 0 && DateTime.Now.Hour <= 6) {
      Console.WriteLine ("We are in the dangerous hours");
    }
  }
}
```

3. Register the `TimeReceiver` class as a `TimeTick` broadcast receiver in the `OnCreate()` method of your `MainActivity` class:

```
protected override void OnCreate (Bundle bundle) {

  base.OnCreate (bundle);

  //Register our broadcast receiver. It will be triggered
  when the time change, every minute.
  RegisterReceiver (new TimeReceiver (), new IntentFilter
  (Intent.ActionTimeTick));

}
```

4. Run your application. The following statements will be printed in the Application Output tab of Xamarin Studio:

```
Console.WriteLine ("Time Received");
Console.WriteLine ("We are in the dangerous hours");
```

How it works...

The `TimeReceiver` class extends the `BroadcastReceiver` class. The `BroadcastReceiver` class is a base class that can intercept intent broadcast calls. Android OS contains a huge number of intent broadcast calls, such as `onBootCompleted` and `onTimeChanged`, which we use in this recipe.

With the following statement, we check if the current time is between 12 a.m. and 6 a.m. and print a logging line to the console if this is the case.

```
if (DateTime.Now.Hour >= 0 && DateTime.Now.Hour <= 6) {
  Console.WriteLine ("We are in the dangerous hours");
}
```

This code takes place in the `OnReceive()` method and is called every minute. Indeed, the `ActionTimeTick` broadcast call also happens every minute. Nonetheless, for our `BroadcastReceiver` class to actually receive broadcast calls, we have to register it. This is what we did in the `MainActivity` class with the following statements:

```
RegisterReceiver (new TimeReceiver (), new IntentFilter
(Intent.ActionTimeTick));
```

The `RegisterReceiver()` method is responsible for registering a new receiver that can be set for broadcast calls or local calls. In our case, we register a new `TimeReceiver` object, and we add an `IntentFilter` argument specifying that our receiver is only interested in the `ActionTimeTick` broadcast call.

Broadcast receiver is a very elegant way to monitor events in the Android System without using a `while (true) {//Some Check}` instance, which will drain the phone's battery. However, they can only be used when you want to monitor something linked to a broadcast event.

See also

Refer to the *Application monitoring* recipe to see how we can monitor the running application on the user's phone.

Application monitoring

This is the second recipe on our way to building *The Internship* application. Here, we will monitor the applications that are open on our system.

Getting ready

For this recipe, we will continue to work with the `TimeReceiver` class created in the previous recipe.

How to do it...

We will do the application monitoring using the following steps:

1. Create the `SMS_APP_NAME` constant in the `TimeReceiver` class as follows:

```
private const String SMS_APP_NAME =
"com.android.mms.ui.ConversationList";
```

2. Add a method called `smsAppLaunched`, taking a `Context` object as parameter and returning a `Boolean` value:

```
private bool smsAppLaunched(Context context) {
  //Check the 20 last launched activity
  ActivityManager activityManager = (ActivityManager)
  context.GetSystemService (Context.ActivityService);
  IList<ActivityManager.RunningTaskInfo> list =
  activityManager.GetRunningTasks (20);

  foreach (ActivityManager.RunningTaskInfo task in list) {

    //If the sms app is open
    if (task.BaseActivity.ClassName.Equals (SMS_APP_NAME,
    StringComparison.Ordinal)) {
```

```
        Console.WriteLine ("The SMS app is open...");

        return true;
      }
    }
    return false;
  }
```

3. Add a call to `smsAppLaunched` in the `OnReceive()` method so that it now looks as follows:

```
public override void OnReceive (Context context, Intent
intent) {
  Console.WriteLine ("Time Received");
  //If we are in the dangerous timeframe
  if (DateTime.Now.Hour >= 0 && DateTime.Now.Hour <= 6) {
    Console.WriteLine ("We are in the dangerous hours");
    if (smsLaunched(context)) {
      Console.WriteLine ("We have to do something !");
    }
  }
}
```

4. Add the `READ_LOG` and `GET_TASKS` permissions in your `AndroidManifest` file in order to have the authorization to execute `activityManager.GetRunningTasks (20);`.

5. Run your application and the following statements will be printed if the time in the phone is between 12 a.m. and 6 a.m.:

```
Console.WriteLine ("Time Received");
Console.WriteLine ("We are in the dangerous hours");
Console.WriteLine ("The SMS app is open...");
Console.WriteLine ("We Have to do something!");
```

Note that the time changed broadcast message is sent only once every minute, therefore, you might have to wait for a whole minute to get those statements printed in your console.

How it works...

The modifications performed in order to have the running application are contained in the `smsAppLaunched()` method:

```
ActivityManager activityManager = (ActivityManager)
context.GetSystemService (Context.ActivityService);
IList<ActivityManager.RunningTaskInfo> list =
activityManager.GetRunningTasks (20);
```

In this snippet of code, we use the context object received in the `OnReceived()` method passed to the `smsAppLaunched` parameter to obtain the activity manager. The activity manager is cast from the `ActivityService` instance running continuously on Android. Then, we retrieve the last 20 running tasks using the `GetRunningTasks()` method of the `activityManager` class and store the response in a list of `RunningTaskInfo`.

You will need to add the following `using System.Collections.Generic;` in order to use the `IList` type as depicted here.

This `RunningTaskInfo` instance contains a reference to the base activity that it represents. We can get the name of the activity using the following:

```
RunningTaskInfo.BaseActivity.ClassName
```

Every application has a fully qualified Java name that looks like `com.android.mms.ui.ConversationList`.

By browsing through our list with a `foreach` statement and testing the class name, we can determine if the SMS activity is running or not:

```
foreach (ActivityManager.RunningTaskInfo task in list) {

  //If the sms app is open
  if (task.BaseActivity.ClassName.Equals (SMS_APP_NAME,
  StringComparison.Ordinal)) {
    Console.WriteLine ("The SMS app is open...");
    return true;
  }
}
```

Hold on! Is this a security hole? Should we be able to see what other applications are running on the phone? Well, yes and no. Yes, you can retrieve all the applications running on the system at the same time as your application and as often as you want, which can be seen as a privacy problem, but you won't be able to kill those applications. Now, you may be wondering how the task killer works. Well, they don't kill applications but restart them. As you know from *Chapter 2*, *Mastering the Life and Death of Android Apps*, restarting an application will save the application state and remove the process if the application was pushed to the background.

Solving equations

In this third recipe, we will create a new activity responsible for displaying an equation and checking the answer.

How to do it...

We will now see how to solve the equations:

1. Create a new activity named `EquationActivity` and make it look like the following:

```
namespace TheInternship {
  [Activity (Label = "EquationActivity")]
  public class EquationActivity : Activity {

    Dictionary<String, int> equations = new
    Dictionary<string, int> ();
    KeyValuePair<String, int> current_equation;

    protected override void OnCreate (Bundle bundle) {
      base.OnCreate (bundle);

      SetContentView (Resource.Layout.Equation);

      // Add some basic equations in our Dictionary of
      equation
      equations.Add ("2+2", 4);
      equations.Add ("4/2", 2);
      equations.Add ("3*3", 9);
      equations.Add ("3*25", 75);

      //Select one equation
```

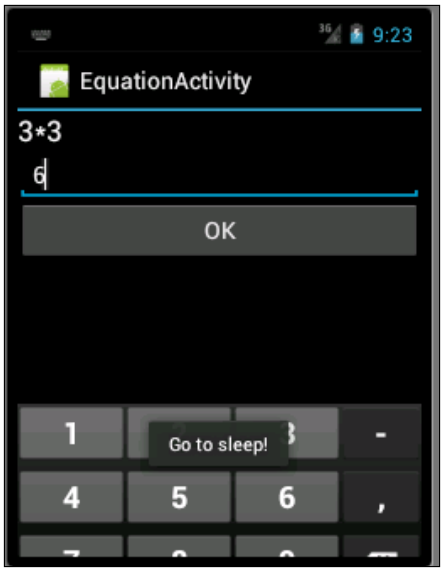
```
            changeEquation ();

            // Check the response of the equation on click via a
            delegate
            Button button = FindViewById<Button>
            (Resource.Id.send_eq);
            button.Click += delegate {

              // Convert the response to int32. We could use a
              try/catch here.
              if(Convert.ToInt32(FindViewById<TextView>
              (Resource.Id.response).Text) ==
              current_equation.Value) {
                Console.WriteLine ("Answer is OK...");
              }
              else {
                Toast.MakeText (this, "Go to sleep!",
                ToastLength.Short).Show ();
              }
            };

          }

          private void changeEquation() {

            //Load a random equation from the Dictionary
            Random rand = new Random ();
            current_equation = equations.ElementAt (rand.Next (0,
            equations.Count));
            FindViewById<TextView> (Resource.Id.eq_text).Text =
            current_equation.Key;
          }
        }
      }
```

2. Create a `.axml` file under the `Resources/Layout` folder named `Equation.axml`, and add the following code in order to construct a graphical interface with a `TextView`, `TextInput`, and `button` elements:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
```

```
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/eq_text" />
  <EditText
    android:inputType="number"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/response" />
  <Button
    android:text="OK"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/send_eq" />
</LinearLayout>
```

3. Launch your application. If you enter a wrong answer, a popup saying **Go to sleep!** will be prompted. The correct answer will print **Answer is ok** to the console:



How it works...

In the previous code, we use the `OnCreate()` method of our newly created activity to create a list of equations and their answers:

```
Dictionary<String, int> equations = new Dictionary<string, int>
();
equations.Add ("2+2", 4);
equations.Add ("4/2", 2);
equations.Add ("3*3", 9);
equations.Add ("3*25", 75);
```

The `Dictionary` argument belongs to the `System.Collections.Generic` package and represents a list of keys and values, which fit our needs.

Then, we call the `changeEquation()` method in which we randomly select one of the equations and populate the graphical interface with it:

```
Random rand = new Random ();
current_equation = equations.ElementAt (rand.Next (0,
equations.Count));
FindViewById<TextView> (Resource.Id.eq_text).Text =
current_equation.Key;
```

Finally, we add a `delegate` argument on the **OK** button and click and check whether the answer is correct:

```
Button button = FindViewById<Button> (Resource.Id.send_eq);
button.Click += delegate {

  // Convert the response to int32. We could use a try/catch here.
  if(Convert.ToInt32(FindViewById<TextView>
  (Resource.Id.response).Text) == current_equation.Value) {
    Console.WriteLine ("Answer is OK...");
  }
  else {
    Toast.MakeText (this, "Go to sleep!", ToastLength.Short).Show
    ();
  }
};
```

See also

Refer to the *Sending an SMS* recipe to see how to send an SMS using an intent, and finish *The Internship* application.

Sending an SMS

With this fourth recipe on `TheInternship` project, we will pick a contact from the contact list of the user and send an SMS to him/her.

How to do it...

Let's look at how to send an SMS.

1. Create a new activity named `SendActivity` and add the following code to the `OnCreate()` method:

```
// On click of the contact field, run a contact selection
Intnet.
FindViewById<TextView> (Resource.Id.ppl1).Click += delegate
{

  //Concrete description of our Intent. Here we want to
  pick an User
  Intent contact_intent = new Intent(Intent.ActionPick,
  ContactsContract.Contacts.ContentUri);

  //More specifically, we would like the phone number
  contact_intent.SetType
  (ContactsContract.CommonDataKinds.Phone.ContentType);

  // Start the Intent and provide a response code. When the
  user has picked a contact,
  // OnActivityResult will be called.
  StartActivityForResult(contact_intent, contact_code);
};

// When the send button is clicked, Send a sms to the
elected contact and the text
FindViewById<Button> (Resource.Id.send1).Click += delegate
{

  var uri = Android.Net.Uri.Parse("sms:"+this.phone);
  // Abstract URI for sms

  Intent send_intent = new Intent(Intent.ActionSendto,
  uri);
```

```
    // Send to Action

    //Append the text with -Checked by the Internship and add
    it to the sms_body variable
    send_intent.PutExtra("sms_body", FindViewById<TextView>
    (Resource.Id.sms2).Text  + "-Checked by the Internship");

    //We don't expect anything back here
    StartActivity(send_intent);

};
```

2. Override the `OnActivityResult()` method of the `SendActivity` class with the following code:

```
protected override void OnActivityResult(int requestCode,
Result resultCode, Intent data) {

  base.OnActivityResult (requestCode, resultCode, data);

  //Are we called after a contact pick ?
  if (requestCode == contact_code) {

    //Build a cursor w/ the response we get
    Android.Net.Uri contactData = data.Data;
    Android.Database.ICursor contactCursor = ManagedQuery
    (contactData, null, null, null, null);
    StartManagingCursor (contactCursor);

    //If a user is in the response, get the name and the
    phone
    if (contactCursor.MoveToFirst ()) {

      this.name = contactCursor.GetString
      (contactCursor.GetColumnIndexOrThrow
      ("display_name"));
      this.phone =  contactCursor.GetString
      (contactCursor.GetColumnIndexOrThrow ("data4"));

      FindViewById<TextView> (Resource.Id.ppl1).Text =
      this.name + " (" + this.phone +")";
    }
  }
}
```

3. Add the following three variables to the `SendActivity` class:

```
private const int contact_code = 1;
private String phone;
private String name;
```

4. Create a `.axml` file named `Send.axml` under the `Resources/Layout` folder, and add the following code in it in order to have two text fields and one button:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/ppl1" />
  <EditText
    android:inputType="textMultiLine"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/sms2" />
  <Button
    android:text="Send"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:id="@+id/send1" />
</LinearLayout>
```

5. Modify the `OnReceive()` method of the `TextReceiver` class so that it starts the `EquationActivity` instance if the SMS app is open:

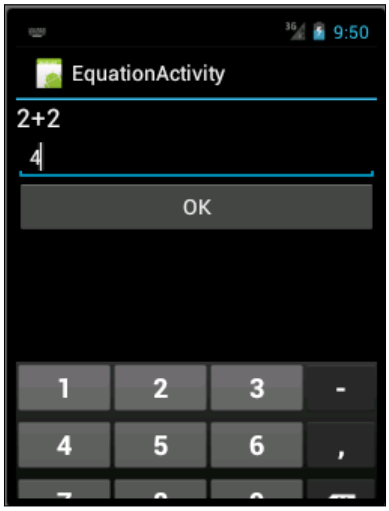
```
public override void OnReceive (Context context, Intent
intent) {
  Console.WriteLine ("Time Received");
  //If we are in the dangerous timeframe
  if (DateTime.Now.Hour >= 0 && DateTime.Now.Hour <= 6) {
    Console.WriteLine ("We are in the dangerous hours");
    if (smsLaunched(context)) {
      context.StartActivity (typeof(EquationActivity));
    }
  }
}
```

6. Modify the `OnCreate()` method of the `Equation` class so that the `sendActivity` class is called when the answer is correct:

```
// Check the response of the equation on click via a
delegate
Button button = FindViewById<Button> (Resource.Id.send_eq);
button.Click += delegate {

  // Convert the response to int32. We could use a
  try/catch here.
  if(Convert.ToInt32(FindViewById<TextView>
  (Resource.Id.response).Text) == current_equation.Value) {
    Console.WriteLine ("Answer is OK...");
    StartActivity(typeof(SendActivity));
    // Redirect the user tozards the Send Activity
  }
  else {
    Toast.MakeText (this, "Go to sleep!",
    ToastLength.Short).Show ();
  }
};
```

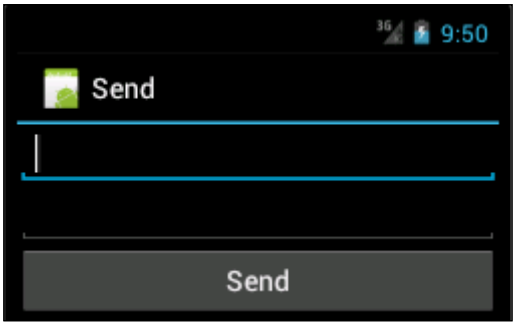
7. Add the `READ_CONTATCS` and `SEND_SMS` permissions to your `AndroidManifest.xml` file.
8. Launch your application!

   The following screens should appear if the SMS application is launched and the time is between 12 a.m. and 6 a.m. Answer the equation correctly:

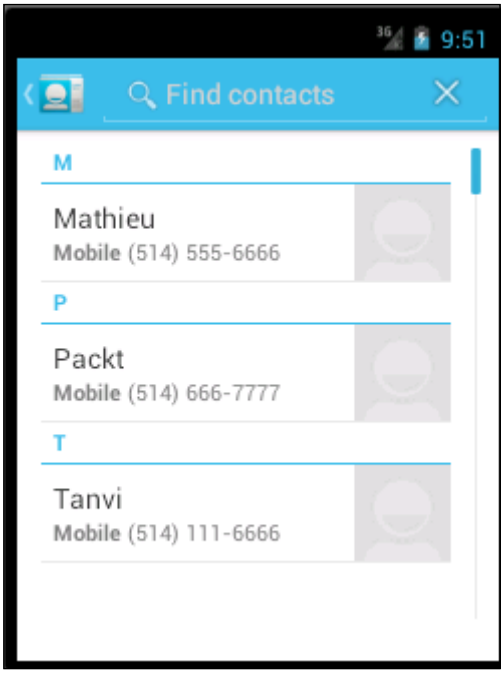


It is 9:50 p.m. Why is the `Equation` activity triggered? This is because I've modified the condition in `TimeReceiver` so that I can test my application without changing the emulator time or waiting for 12 a.m.

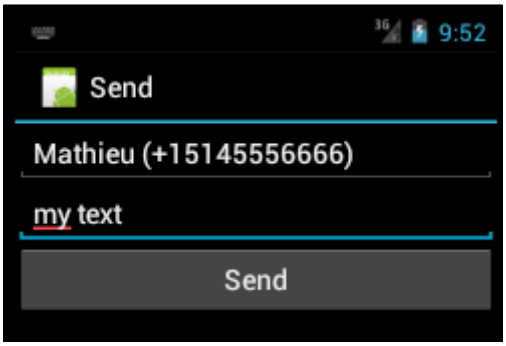
The `Send` activity is displayed.



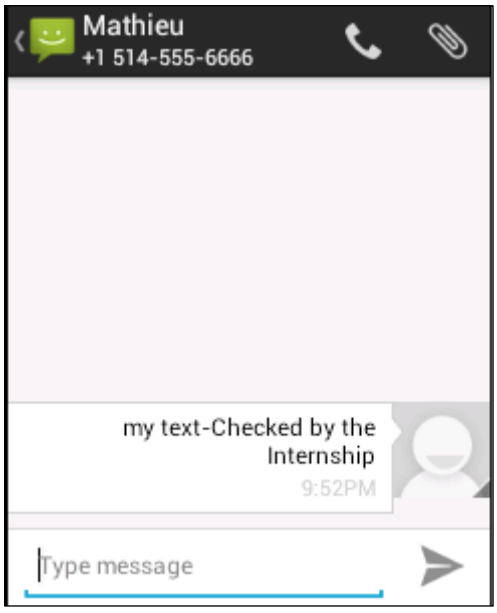
9. Click on the first `textView` element of the send label and your agenda will open:



10. Choose a contact. We will be redirected to our application in which we can write our SMS:

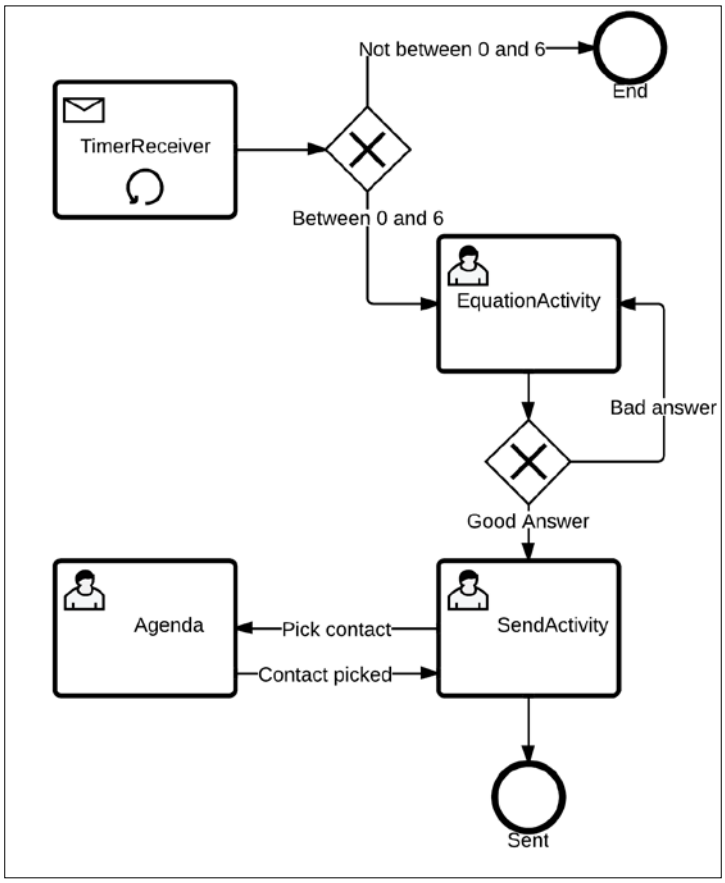


11. Finally, when you click on the **Send** button and you will see the following screen:



How it works...

In order to understand what happens in this application, let's have a look to the following transition diagram:



First of all, our broadcast receiver class named `TimeReceiver` will receive a time change message from the Android operating system every minute. If the time is between 12 a.m. and 6 a.m., the `Equation` activity will be launched. In this activity, we ask the user to solve a simple equation and launch `SendActivity` in case the correct answer is given. In `SendActivity`, we ask the user to pick a contact in their agenda and retrieve the picked contact. Finally, the user can write their SMS and send it.

In the `OnCreate()` method of the `send` activity, we have the following snippet, which will start the `Agenda` application:

```
//Concrete description of our Intent. Here we want to pick an User
Intent contact_intent = new Intent(Intent.ActionPick,
ContactsContract.Contacts.ContentUri);

//More specifically, we would like the phone number
contact_intent.SetType(ContactsContract.CommonDataKinds.Phone.
ContentType);

//Start the Intent and provide a response code. When the user has
picked a contact,
// OnActivityResult will be called.
StartActivityForResult(contact_intent, contact_code);
```

In this code, we first create a concrete action and build our intent with with `ContactsContract.Contacts.ContentUri`. Here, concrete intent is used in opposition to the abstract intent, which we saw earlier in this chapter where a String built out of an URI was used. Using the `ContactsContract.Contacts.ContentUri` instance, we can ensure that the returned `URI` is correct and the agenda application will effectively be launched. We also set a type for our intent: `ContactsContract.CommonDataKinds.Phone.ContentType`.

This content type specifies what we are after, that is, what we want to obtain with this intent. In our case, we would like to access the phone number of the picked contact. Finally, we call the `StartActivityForResult()` method with the newly created intent and `contact_code (1)` as arguments. The `StartActivityForResult()` method is the same method as `StartActivity()` except that we specify that we expect an answer from the launched activity. When the activity launched with our intent has an answer to give, the `OnActivityResult()` method will be called.

Let's take a look at our overridden `OnActivityResult()` method:

```
//Build a cursor w/ the response we get
Android.Net.Uri contactData = data.Data;
Android.Database.ICursor contactCursor = ManagedQuery
(contactData, null, null, null, null);
StartManagingCursor (contactCursor);

//If a user is in the response, get the name and the phone
if (contactCursor.MoveToFirst ()) {

  this.name = contactCursor.GetString
  (contactCursor.GetColumnIndexOrThrow ("display_name"));
```

```
    this.phone =  contactCursor.GetString
    (contactCursor.GetColumnIndexOrThrow ("data4"));

    FindViewById<TextView> (Resource.Id.ppl1).Text = this.name + "
    (" + this.phone +")";
}
```

In this code, we first retrieved a `URI` instance from the intent (named data) received as an argument with the `data.Data()` method. Then, we obtain a cursor as a response of a `ManagedQuery` instance. The signature of this method is as follows:

```
managedQuery (Uri uri, String[] projection, String selection,
String[] selectionArgs, String sortOrder)
```

The parameters of the intent correspond to the following:

```
uri - The URI of the content provider to query.
projection - List of columns to return.
selection - SQL WHERE clause.
selectionArgs - The arguments to selection, if any ?s are present
sortOrder - SQL ORDER BY clause.
```

The `ManagedQuery` instance is a wrapper around an existing cursor. The cursor can have any source—not necessarily an Android database. In this case, we query against the Android OS database. In our case, however, we already have a content provider that only contains what we are interested in, that is, a contact. Therefore, we have to set all other arguments to `null`. We then move to the first item of the cursor and select the name and phone number of the picked contact:

```
this.name = contactCursor.GetString
(contactCursor.GetColumnIndexOrThrow ("display_name"));
this.phone =  contactCursor.GetString
(contactCursor.GetColumnIndexOrThrow ("data4"));
```

The values inside the cursor are accessed by the `contactCursor.GetString()` method, which will return a `String` value and take the ID of the targeted column as a parameter. As we don't know this ID, we use the `contactCursor.GetColumnIndexOrThrow ("display_name")` and `contactCursor.GetString (contactCursor.GetColumnIndexOrThrow ("data4")` methods. The `data4` column contains the phone number without any added data, that is, 5145556666 and not (514) 555-6666.

To know what the column names and their data during the development phase are, you can use the following code:

```
foreach (String column in contactCursor.GetColumnNames)
{
  Console.WriteLine (contactCursor.GetString
  (contactCursor.GetColumnIndexOrThrow (column)));
}
```

This will output the name of the column and the value.

Now that we have the contact to which we want to send our SMS, we can grab the text typed by the user and send it. In the `OnCreate()` method of the `SendActivity` class, we have the following:

```
// When the send button is clicked, Send a SMS to the selected
contact and the text
FindViewById<Button> (Resource.Id.send1).Click += delegate {

  var uri = Android.Net.Uri.Parse("sms:"+this.phone); // Abstract
  URI for sms

  Intent send_intent = new Intent(Intent.ActionSendto, uri);
  // Send to Action

  //Append the text with -Checked by the Internship and add it to
  the sms_body variable
  send_intent.PutExtra("sms_body", FindViewById<TextView>
  (Resource.Id.sms2).Text  + "-Checked by the Internship");

  //We don't expect anything back here
  StartActivity(send_intent);

};
```

For sending the SMS, we use an intent built out of a `URI` object composed of the SMS string and the phone number and the phone number of the contact. We build an `ActionSent` intent and call the `PutExtra()` method on it. The `PutExtra()` method will enable us to add extra information to the intent in addition to the bare `URI` object. In our example, we concatenate the text of our `TextView` with `Checked by the Internship` into the `sms_body` column into the SMS. Finally, we use the `StartActivity` instance and not the `StartActivityForResult` instance as we don't expect anything back.

In the previous four recipes, we created an application, based on Intents, that check the time and ask you to solve an equation before sending an SMS if it is between 12 p.m. and 6 a.m.

9
Playing with Advanced Graphics

In this chapter, we will cover the following recipes:

- Using the camera
- Taking screenshots with the camera
- Creating animations
- Creating your own gestures

Introduction

In *Chapter 3*, *Building a GUI*, and *Chapter 5*, *Using On-Phone Data*, we learned how to take advantage of classical available layouts, display icons, and customize buttons. In *Chapter 6*, *Populating Your GUI with Data*, we designed a custom adapter to display our data. In this chapter, we will push forward the creation of advanced graphics such as 2D graphics in our application and animation. We will also take advantage of the Android API made available by Xamarin to use the camera.

Using the camera

In this recipe, we will learn how to display the camera feed in our application. This simple yet powerful feature can be integrated in many ways to your applications. Indeed, you can envision applications that allow a live video feed, such as Skype.

Getting ready

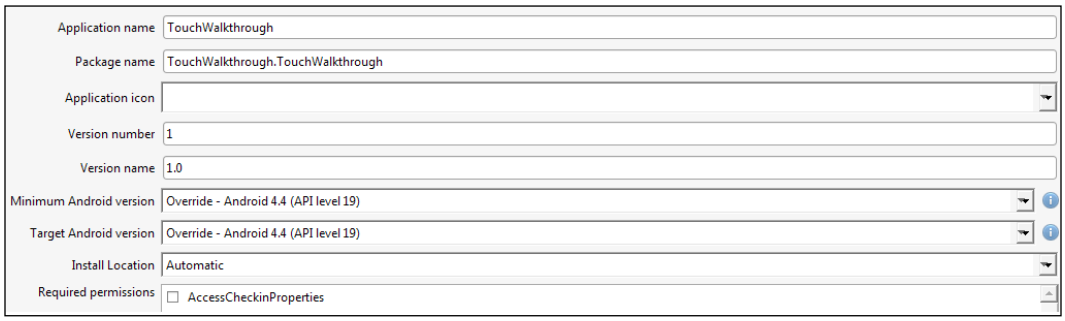
Create a new solution named `CameraTest` and open it in Xamarin Studio. This recipe will only work on a physical device.

How to do it...

1. Open the `AndroidManifest.xml` file under the `Properties` folder and request the **Camera**, **CaptureVideoOutput** and **Write external** permissions.
2. Now add the following statement to the `AndroidManifest.xml` file:

```
<application android:hardwareAccelerated="true"/>
```

3. In the minimum and target Android version, choose the following: **Override – Android 4.4 (API level 19),** as shown in the following screenshot:



`http://developer.xamarin.com/recipes/android/other_ux/textureview/display_a_stream_from_the_camera/`

4. Import the `Android.Hardware` package in your `MainActivity` class.
5. Add the following two variables to your activity:

```
Camera2 _camera;
TextureView _textureView;
```

6. Make your main activity implement `TextureView.ISurfaceTextureListener` and implement the `OnSurfaceTextureAvailable()`, `OnSurfaceTextureAvailable()`,`OnSurfaceTextureSizeChanged()`, and `OnSurfaceTextureUpdated()` methods with the following code:

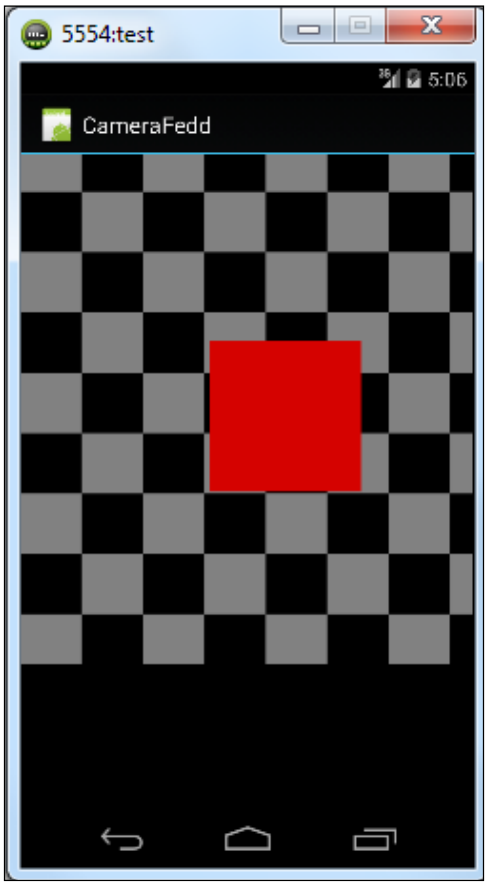
```
public void OnSurfaceTextureAvailable
(Android.Graphics.SurfaceTexture surface, int w, int h) {
```

```
    _camera = Camera2.Open ();

    _textureView.LayoutParameters = new
    FrameLayout.LayoutParams (w, h);

    try {
      _camera.SetPreviewTexture (surface);
      _camera.StartPreview ();

    }
    catch (Java.IO.IOException ex) {
      Console.WriteLine (ex.Message);
    }
  }

  public bool OnSurfaceTextureDestroyed
  (Android.Graphics.SurfaceTexture surface) {
    _camera.StopPreview ();
    _camera.Release ();

    return true;
  }

  public void OnSurfaceTextureSizeChanged
  (Android.Graphics.SurfaceTexture surface, int width, int
  height) {
    // Camera takes care of that
  }

  public void OnSurfaceTextureUpdated
  (Android.Graphics.SurfaceTexture surface) {
    // Camera takes cares of that too
  }
```

7. Add the following code in the `OnCreate()` method:

```
base.OnCreate (bundle);

_textureView = new TextureView (this);
_textureView.SurfaceTextureListener = this;

SetContentView (_textureView);
```

8. Run your application and see the following results:



How it works...

In this section, we will see how we manage to display a stream from the camera in our application. First of all, in addition to the classical `Activity` class, our activity implements the `TextureView.ISurfaceTextureListener` interface. This interface is available since API 14, which explained why we can't use it with devices running API 13 or less, and it's a listener that can be used to be notified when the associated surface texture view is available. We associate the surface in the `OnCreate()` method of the `MainActivity` class with the following statements:

```
_textureView = new TextureView (this);
_textureView.SurfaceTextureListener = this;

SetContentView (_textureView);
```

The four methods defined in this interface are:

- `onSurfaceTextureAvailable(SurfaceTexture surface, int width, int height)`: This is invoked when a `SurfaceTexture` instance is ready to be used.
- `onSurfaceTextureDestroyed(SurfaceTexture surface)`: This is invoked when the specified `SurfaceTexture` is about to be destroyed. If the method returns `true`, no rendering should happen inside the surface texture after this method is invoked. Otherwise, if it returns `false`, the client needs to call `release()`. Most applications should return `true`.
- `onSurfaceTextureSizeChanged (SurfaceTexture surface, int width, int height)`: This is invoked when the `SurfaceTexture` instance's buffer's size changed.
- `onSurfaceTextureUpdated (SurfaceTexture surface)`: This is invoked when the specified `SurfaceTexture` is updated through the `updateTexImage()` method.

The most interesting method is `OnSurfaceTextureAvailable`, which takes care of effectively displaying the stream from the camera to our texture.

Now consider the following snippet:

```
_camera = Camera.Open();

_textureView.LayoutParameters = new FrameLayout.LayoutParams
(w, h);

try {
  _camera.SetPreviewTexture (surface);
  _camera.StartPreview ();

}
catch (Java.IO.IOException ex) {
  Console.WriteLine (ex.Message);
}
```

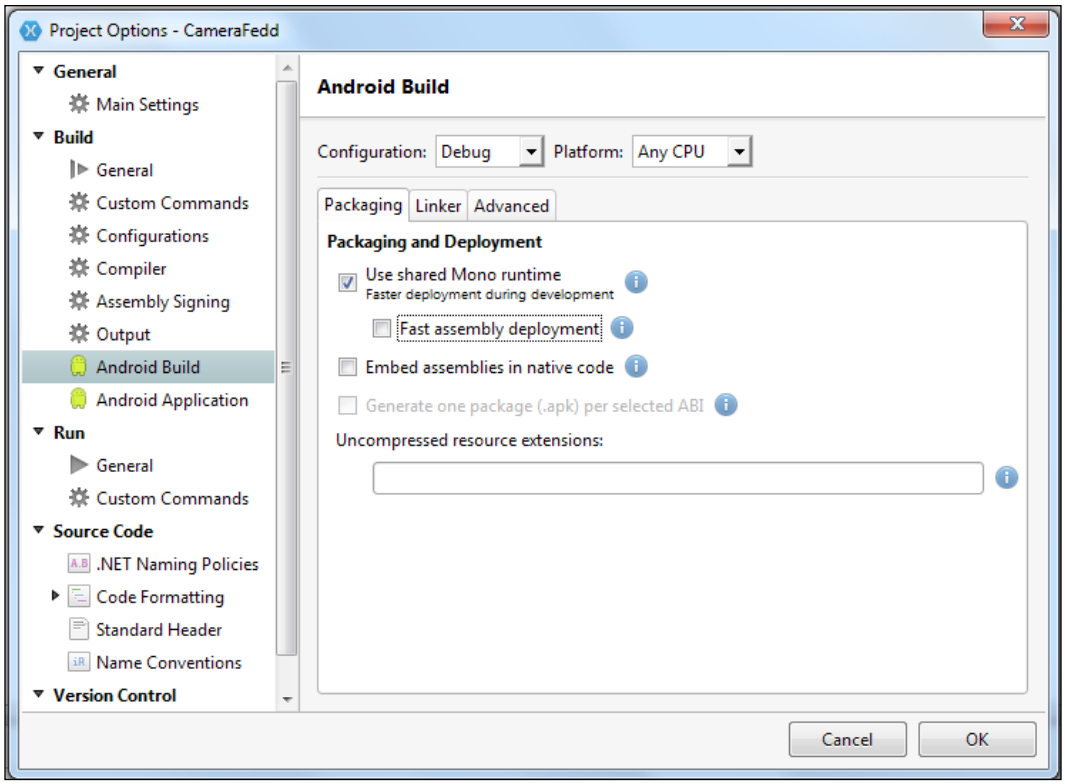
In this snippet, we first give the `_camera` instance the value of `Camera.Open()`, which will return a new `Camera` object to access the first, rear camera of the device. Then, we block out an area of size `w`, `h` on the screen to display our feed using the new `FrameLayout.LayoutParams(w, h)`. Finally, we set the surface to be used for the camera preview with `SetPreviewTexture` and start the preview itself.

There's more...

As we are starting to play with hardware and, more specifically, hardware emulation of the camera, several things could go wrong in this recipe. In what follows, I will try to cover the most common errors related to both Android and Xamarin.

Deployment failed. FastDev directory creation failed

When deploying applications containing hardware acceleration, I found that the error **Deployment failed. FastDev directory creation failed** was appearing from time to time. This is related to the fast assembly deployment feature of Xamarin. The workaround here is to simply deactivate the feature by unchecking it in the **Android Build** section of the **Project Options,** as shown in the following screenshot:

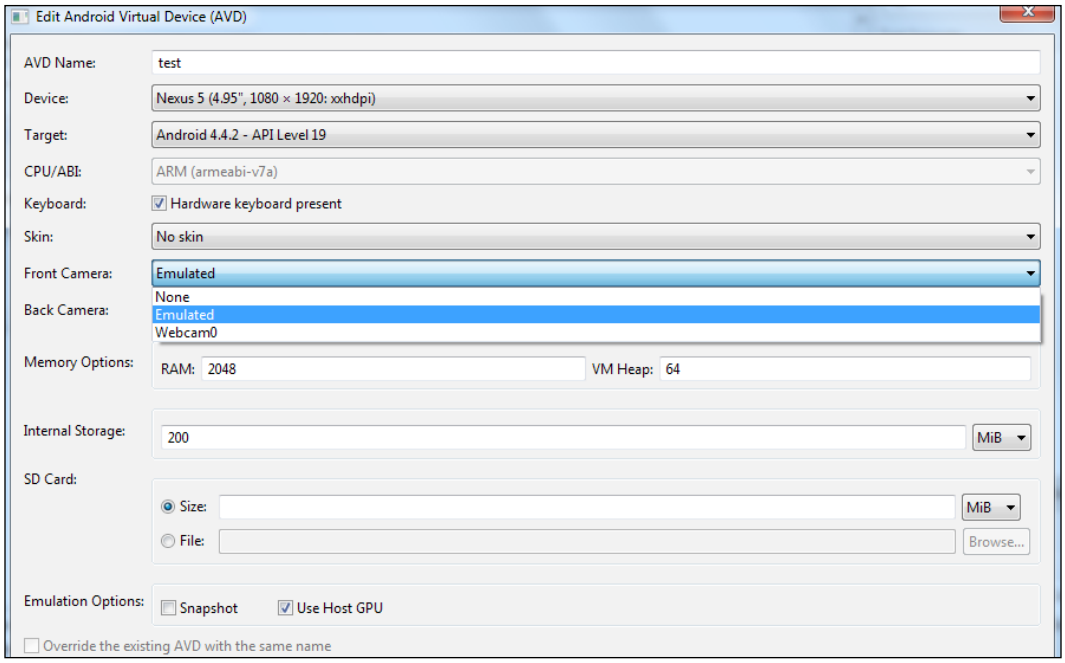


A TextureView or a subclass can only be used with hardware acceleration enabled

If you add the following statement to your `AndroidManifest.xml` file, as indicated in the second step of this recipe, and **A TextureView or a subclass can only be used with hardware acceleration enabled** error message still appears, then it means that the device you are testing simply doesn't support the hardware acceleration. The minimum versions are API 15 (Android 4.0.3) for a physical device and API 17 (Android 4.2) for an emulated device. If you don't see an emulator superior to API 17 while running your project, you can create a new emulator in the **Manage devices** menu.

Black screen

If your application starts normally but you only get a black screen in the emulator, then it might indicate that your emulated device doesn't have a rear camera. Indeed, `Camera.Open` automatically opens the rear camera, even if your device doesn't have one. You can set the camera in the **Edit Android Virtual Device (AVD)** option, as shown in the following screenshot:



See also

See also, the next recipe to see how to take pictures with the camera. You can find more information at `http://developer.xamarin.com/recipes/android/` and `http://developer.android.com/reference/android/hardware/Camera.html`.

Taking screenshots with the camera

In this recipe, we will use the default camera application in our own application and take advantage of it to save pictures. This is the opposite to the first recipe of this chapter, where we were displaying a feed coming directly from the hardware camera and bypassing the Android default application for the camera.

Getting ready

Create a new solution named `CameraPicture` and open it in Xamarin Studio.

How to do it...

1. Create another project in the solution created in the first recipe and name it `CameraPicture`.
2. Add the following code into your `Main.axml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <Button
    android:id="@+id/myButton"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/openCamera" />
  <ImageView
    android:src="@android:drawable/ic_menu_gallery"
    android:layout_width="fill_parent"
    android:layout_height="300.0dp"
    android:id="@+id/imageView1"
    android:adjustViewBounds="true" />
</LinearLayout>
```

3. In the `MainActivity` class, declare a static `App` class:

```
public static class App{
  public static File _file;
  public static File _dir;
  public static Bitmap bitmap;
}
```

4. Modify the `OnCreate()` method of your `MainActivity` class so it matches the following:

```
protected override void OnCreate(Bundle bundle) {
  base.OnCreate(bundle);
  SetContentView(Resource.Layout.Main);

  if (IsThereAnAppToTakePictures()) {
    CreateDirectoryForPictures();

    Button button =
    FindViewById<Button>(Resource.Id.myButton);
    _imageView = FindViewById<ImageView>
    (Resource.Id.imageView1);
    if (App.bitmap != null) {
      _imageView.SetImageBitmap (App.bitmap);
      App.bitmap = null;
    }
    button.Click += TakeAPicture;
  }
}
```

5. Add the following code (reference `http://developer.xamarin.com/recipes/android/other_ux/camera_intent/take_a_picture_and_save_using_camera_app/`) to the `MainActivity` class we previously created:

```
private bool IsThereAnAppToTakePictures() {
  Intent intent = new Intent
  (MediaStore.ActionImageCapture);
  IList<ResolveInfo> availableActivities =
  PackageManager.QueryIntentActivities(intent,
  PackageInfoFlags.MatchDefaultOnly);
  return availableActivities != null &&
  availableActivities.Count > 0;
}

private void CreateDirectoryForPictures() {
  App._dir = new File
  (Android.OS.Environment.GetExternalStoragePublicDirectory
  ( Android.OS.Environment.DirectoryPictures),
  "CameraAppDemo");
  if (!App._dir.Exists()) {
    App._dir.Mkdirs();
  }
}
```

6. Add a resource named `openCamera` in the `Strings.xml` file:

```
<string name="openCamera">Open Camera</string>
```

7. Implement the `TakeAPicture()` method that will be called when the user clicks on our button:

```
private void TakeAPicture(object sender, EventArgs
eventArgs) {
  Intent intent = new Intent
  (MediaStore.ActionImageCapture);

  App._file = new File(App._dir, String.Format
  ("myPhoto_{0}.jpg", Guid.NewGuid()));

  intent.PutExtra(MediaStore.ExtraOutput,
  Android.Net.Uri.FromFile(App._file));

  StartActivityForResult(intent, 0);
}
```

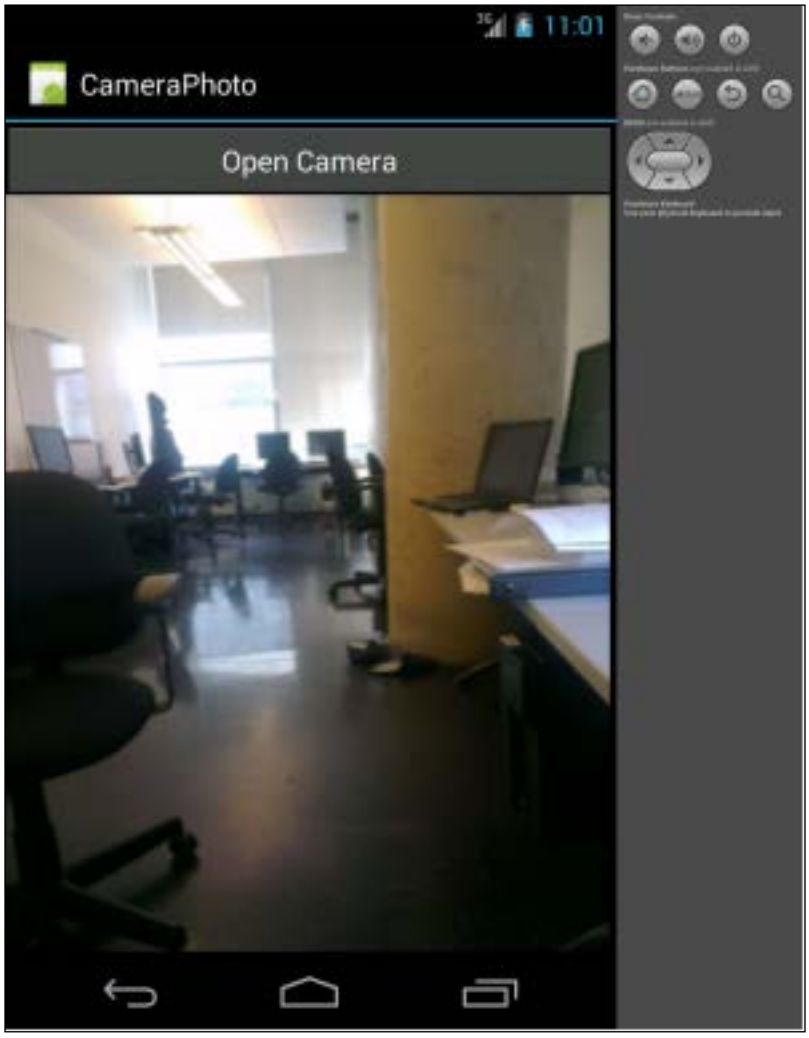
8. Add an `OnActivityResult()` method to the `MainActivity.cs` file:

```
protected override void OnActivityResult(int requestCode,
Result resultCode, Intent data) {
  base.OnActivityResult(requestCode, resultCode, data);

  // make it available in the gallery
  Intent mediaScanIntent = new
  Intent(Intent.ActionMediaScannerScanFile);
  Android.Net.Uri contentUri = Uri.FromFile(App._file);
  mediaScanIntent.SetData(contentUri);
  SendBroadcast(mediaScanIntent);

  // display in ImageView. We will resize the bitmap to fit
  the display
  // Loading the full sized image will consume to much
  memory
  // and cause the application to crash.
  int height = Resources.DisplayMetrics.HeightPixels;
  int width = _imageView.Width ;
  App.bitmap = App._file.Path.LoadAndResizeBitmap (width,
  height);
}
```

9. Create a static class named `BitMapHelpers`:

```
public static class BitmapHelpers {
  public static Bitmap LoadAndResizeBitmap(this string
  fileName, int width, int height) {
    // First we get the the dimensions of the file on disk
    BitmapFactory.Options options = new
    BitmapFactory.Options { InJustDecodeBounds = true };
    BitmapFactory.DecodeFile(fileName, options);

    // Next we calculate the ratio that we need to resize
    the image by
    // in order to fit the requested dimensions.
    int outHeight = options.OutHeight;
    int outWidth = options.OutWidth;
    int inSampleSize = 1;

    if (outHeight > height || outWidth > width) {
      inSampleSize = outWidth > outHeight
      ? outHeight / height
      : outWidth / width;
    }

    // Now we will load the image and have BitmapFactory
    resize it for us.
    options.InSampleSize = inSampleSize;
    options.InJustDecodeBounds = false;
    Bitmap resizedBitmap =
    BitmapFactory.DecodeFile(fileName, options);

    return resizedBitmap;
  }
}
```

10. Run your application. You should see something similar to the following screenshot where you can see a picture of my lab at Concordia University taken with the webcam that replaces the camera in the Android emulator:



How it works...

First of all, the code we use here is composed of intents—see *Chapter 8*, *Mastering Intents – A Walk-through*—and, more specifically, the `ActionImageCapture` intent to launch the camera application. As said in the preamble of this recipe, this time we use the default camera application instead of building our own like in the first recipe. The next action of this application is to create a directory to save the pictures taken with the application. The process of saving the picture in a file can seem a little bit tricky as we first create an empty file and add the URI of the said file to the camera intent. Finally, we intercept the `OnActivityResult` action of the default camera application to make the picture available on the device's gallery and in the `ImageView` instance.

See also

See also the next recipe to create beautiful animations in your applications, and the links mentioned in the previous recipe for more information.

Creating animations

Users tend to be attracted by animation in applications. Animations can be the right push towards the summit of user experience. Of course, the best animations are the ones that users don't notice because they feel natural.

View animations are tied to a specific view and can perform simple transformations on the contents of the view. Because of its simplicity, this API is still useful for things such as alpha animations, rotations, and so forth.

Getting ready

Create a new solution named `Animations` in Xamarin Studio.

How to do it...

1. Create an `.xml` file named `anim.xml` under the `Resources` folder and add the following content to it:

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:shareInterpolator="false">

  <scale android:interpolator=
  "@android:anim/accelerate_decelerate_interpolator"
```

```
    android:fromXScale="1.0"
    android:toXScale="1.4"
    android:fromYScale="1.0"
    android:toYScale="0.6"
    android:pivotX="50%"
    android:pivotY="50%"
    android:fillEnabled="true"
    android:fillAfter="false"
    android:duration="700" />

  <set android:interpolator=
  "@android:anim/accelerate_interpolator">
    <scale android:fromXScale="1.4"
    android:toXScale="0.0"
    android:fromYScale="0.6"
    android:toYScale="0.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:fillEnabled="true"
    android:fillBefore="false"
    android:fillAfter="true"
    android:startOffset="700"
  android:duration="400"/>

  <rotate android:fromDegrees="0"
    android:toDegrees="-45"
    android:toYScale="0.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:fillEnabled="true"
    android:fillBefore="false"
    android:fillAfter="true"
    android:startOffset="700"
    android:duration="400" />
  </set>
</set>
```

2. Modify the `OnCreate()` method of the `MainActivity.cs` method, adding the following:

```
Animation myAnimation = AnimationUtils.LoadAnimation(this,
Resource.Animation.MyAnimation);
ImageView myImage = FindViewById<ImageView>
(Resource.Id.imageView1);
myImage.StartAnimation(myAnimation);
```

3. Run your application.

How it works...

Animations are bound to the view they are in, meaning that an animation cannot spread over several activities. They are defined in terms of start and end points, size rotation, and transparency. To create them, we can use an XML file as shown in the recipe, which is the recommended way because it eases the readability, maintenance, and evolution of animations or programming.

As shown in the recipe, we store the animations' XML files under the `Resources/anim` directory of our project. This file must have one of the following elements as the root element:

- `alpha`: This is a fade-in or fade-out animation
- `rotate`: This is a rotation animation
- `scale`: This is a resizing animation
- `translate`: This is a horizontal and/or vertical motion
- `set`: This is a container that may hold one or more of the other animation elements

While not mandatory, the `android:startOffset` attribute will be used with most of your animations. Indeed, it's the attribute responsible for distributing the events composing your animations over time. Without this argument, expressed in milliseconds, all the events will be applied at the same time.

Besides these standard attributes for animation, the Android framework also offers the possibility to control the rate at which effects are played within your animations using `interpolator`. While creating your own `interpolator` object is beyond the scope of this book, the Android framework does embed some classical `interpolator` that you can use such as `Acceleration`, `Deceleration`, `Bounce`, and `Linear` movement:

- `AcclerateInterpolator/DecelerateInterpolator`: These interpolators increase or decrease the rate of change in an animation
- `BounceInterpolator`: These change bounces at the end
- `LinearInterpolator`: The rate of change is constant

In order to use such an `interpolator` you have to place an `android:interpolator` attribute in your `.xml` file as `android:interpolator="@android:anim/accelerate_interpolator` for an `accelerate_interpolator` for example.

To run our application, we have to map it in the view using the following code:

```
Animation myAnimation = AnimationUtils.LoadAnimation
(Resource.Animation.Anim);
ImageView myImage = FindViewById<ImageView>(Resource.Id.myImage);
myImage.StartAnimation(myAnimation);
```

In this code, we first create a new animation out of the `Anim.xml` file using the `AnimationUtils.LoadAnimation` public static method. Then, we load an image of your choice that has to be placed in the `Resource` folder and, finally, run the animation on that image. Our animation will begin with the scale animation that stretches our image horizontally while shrinking it vertically. Then, the image is rotated by 45 degrees. Note that the rotation will be counter-clockwise as we specified `-45` in the `.xml` file.

There's more...

In addition to the classical animation shown in this recipe, we can also animate properties of objects and create drawable animations and, thus, enhance once again our user interface.

Property animations

Property animations first appeared in Android 3.0 and became mainstream in the last two major versions of Android 4.0 and 5.0. Property animations, as the name suggests, animates any property of any object by means of a comprehensible API.

Animated properties are created by using the `Animator` subclasses such as:

- `ValueAnimator`: This is the main subclass of Animator as it computes the values of properties that need to be updated.
- `ObjectAnimator`: A specialization of `ValueAnimator` to ease the animation of objects by accepting a target object and defining the properties to update.
- `AnimationSet`: Similar to classical animation that we saw earlier in the recipe, property animations start altogether by default. If you want to span them over time you have to use an `AnimationSet` element. The `AnimationSet` element will define the time between animations.
- `Evaluators`: These are special classes that are used by animators to calculate the new values during an animation. You can use the following specialized `evaluators`:
  - `IntEvaluator`: This calculates values for integer properties
  - `FloatEvaluator`: This calculates values for float properties
  - `ArgbEvaluator`: This calculates values for color properties

If you are not animating a `float`, `int` or `color`, you may create your own `evaluator` by implementing the `ITypeEvaluator` interface.

Drawable animations

The latest animation type that remains to be covered is named `Drawable` animations. `Drawable` animations will simply use a set of `Drawable` resources as data, and animate them one after the other. Similar to classical animations, drawable animations are defined in XML files in the `/Resource/drawable` folder. However, `Drawable` animations' XML files must contain the `<animation-list>` as a root element. Here's an example of such a `Drawable` animation:

```
<animation-list xmlns:android=
"http://schemas.android.com/apk/res/android">
<item android:drawable="@drawable/image1" android:duration=
"100" />
<item android:drawable="@drawable/image2" android:duration=
"100" />
<item android:drawable="@drawable/image3" android:duration=
"100" />
<item android:drawable="@drawable/image4" android:duration=
"100" />
<item android:drawable="@drawable/image6" android:duration=
"100" />
<item android:drawable="@drawable/image7" android:duration=
"100" />
</animation-list>
```

This simple `Drawable` animation contains six different items and each item will appear one after the other at a 100 ms rate as defined in the `android:duration` attribute. This kind of animation is often used to display a sprite-based animation to our users.

Then, you can simply start the animation on a button click, for example:

```
AnimationDrawable _imageDrawable;

protected override void OnCreate(Bundle bundle) {
  base.OnCreate(bundle);
  SetContentView(Resource.Layout.Main);

  _imageDrawable= (Android.Graphics.Drawables.AnimationDrawable)
  Resources.GetDrawable(Resource.Drawable.image1);

  ImageView MyImageView = FindViewById<ImageView>
  (Resource.Id.MyImageView );
  MyImageView .SetImageDrawable
  ((Android.Graphics.Drawables.Drawable) _imageDrawable );

  Button myButton = FindViewById<Button>(Resource.Id.myButton);
```

```
    myButton .Click += (sender, e) => {
      _imageDrawable.Start();
    };
  }
```

See also

See also, the next recipe to supercharge your application with custom gestures, and the following online resources to learn more about animations:

`http://developer.xamarin.com/recipes/android/other_ux/animation/`

`http://developer.android.com/reference/android/animation/package-summary.html`

`http://gamedevelopment.tutsplus.com/tutorials/an-introduction-to-spritesheet-animation--gamedev-13099`

Creating your own gestures

In the past three recipes of this chapter we learned how to use the camera and how to create animations to enhance the user experience. However, the animations are passive in a sense that your user can only watch them. In this recipe, we will try to engage our future users by proposing custom gestures for our application.

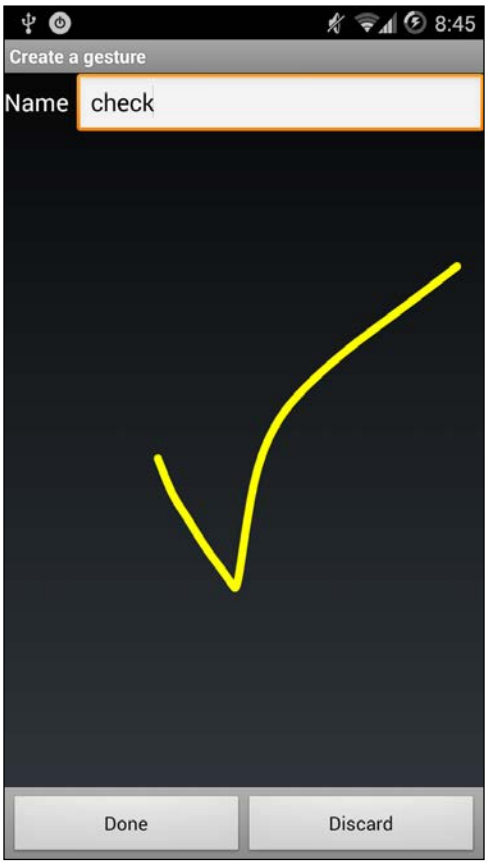
Getting ready

This recipe can only be achieved with the use of a physical Android phone. That said, create a new solution named `CustomGesture` in Xamarin Studio.

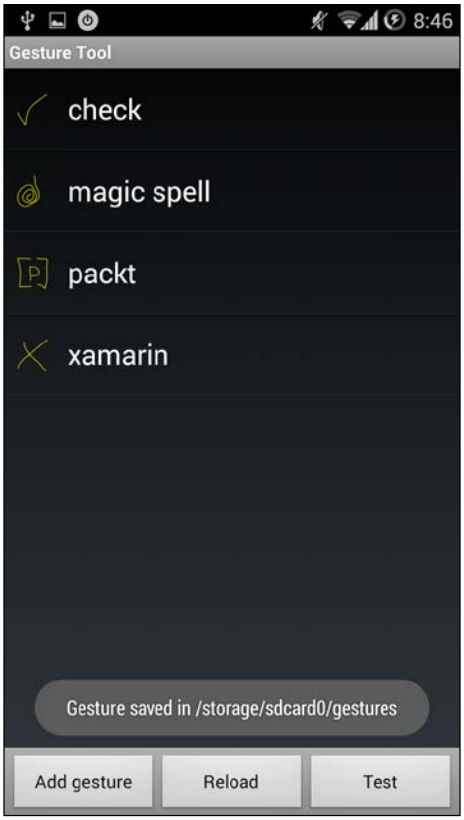
How to do it...

1. Go to the Google Play store and install the **Gesture** tool app created by **davemac**: `https://play.google.com/store/apps/details?id=com.davemac327.gesture.tool&hl=en`

2. Create some gestures with the tool, as shown in the following screenshot:



3. Notice the path where the gestures are saved:



4. Extract the gesture binaries created by the `Gesture` tool and import them in the `Ressource/raw` directory of your project using the following command:

```
adb pull /storage/sdcard0/gestures
PathToCustomeGesture/Resources/raw
```

5. In your `MainActivity` class' `OnCreate()` function, load the gesture files into a gesture library:

```
private Android.Gestures.GestureLibrary _myGestures;

  protected override void OnCreate (Bundle bundle) {
    base.OnCreate (bundle);
    _myGestures = Android.Gestures.GestureLibraries.
    FromRawResource(this, Resource.Raw.gestures);
    if (!_myGestures.Load()) {
```

```
        // The library didn't load, so close the activity.
        Finish();
      }
    }
  }
```

6. Just after the previous snippet, add a gesture overlay to your activity:

```
GestureOverlayView gestureOverlayView = new
GestureOverlayView(this);
gestureOverlayView.AddOnGesturePerformedListener(this);
SetContentView(gestureOverlayView);
```

7. Add a `GestureOverlayViewOnGesturePerformed()` function to your `MainActivity` class. You will also have to import `System.Linq`, as we did in earlier chapters:

```
private void GestureOverlayViewOnGesturePerformed(object
sender, GestureOverlayView.GesturePerformedEventArgs
gesturePerformedEventArgs) {

  System.Collections.Generic.IEnumerable<Prediction>
  predictions = from p in _myGestures.Recognize
  (gesturePerformedEventArgs.Gesture)
  orderby p.Score descending
  where p.Score > 1.0
  select p;
  Prediction prediction = predictions.FirstOrDefault();

  if (prediction == null) {
    Toast.MakeText(this, "No Match",
    ToastLength.Short).Show();
    return;
  }

  Toast.MakeText(this, prediction.Name,
  ToastLength.Short).Show();
}
```

8. Run your application and try some gestures. If the current gesture is recognized, then a toast with the name of the gesture will be displayed. Otherwise, a toast containing **No Match** will appear.

How it works...

First of all, for the first time in this book, we use raw resources and use one of them to bootstrap a gesture library:

```
_myGestures = Android.Gestures.GestureLibraries.
FromRawResource(this, Resource.Raw.gestures);
```

Raw resources are arbitrary files that are saved in their raw form. To open these resources with a raw `InputStream`, we can call the `Resources.openRawResource()` method with the resource `Resource.Raw.gestures` as a resource ID. This mechanism is provided by the `GestureLibraries` and the convenient `FromRawResource()` method. This statement returns a `GestureLibrary` object initialized with our raw file and it's responsible for maintaining gesture examples—created with the gesture tool we download as a first step—and makes predictions on a new gesture.

We used the prediction mechanism provided by the `GestureLibrary` in the `GestureOverlayViewOnGesturePerformed()` method:

```
System.Collections.Generic.IEnumerable<Prediction> predictions =
from p in _myGestures.Recognize(gesturePerformedEventArgs.Gesture)
orderby p.Score descending
where p.Score > 1.0
select p;
```

In this snippet, we ask our `GestureLibrary` to recognize the gesture that the user just made and order our custom gesture by score, using the `Linq` syntax described in *Chapter 5*, *Using On-Phone Data*. The closest the gesture is to one of our custom gestures, the higher the score. Finally we selected the first `Gesture` in the collection of predictions ranked by score using:

```
Prediction prediction = predictions.FirstOrDefault();
```

The documentation about how the score is actually computed is rather scarce, but my best guess is a Euclidian distance between each pixel of the custom gesture and the pixels of the draw gesture, with some vector transformation in order to take into account the different sizes of screens and orientations.

There's more...

In this recipe we learned how to create our own gesture to improve the user experience, however, the Android SDK and therefore Xamarin, allow us to use some pre-configured gestures such as `OnTouch`, `OnLongPress`, and so on.

Detecting those preconfigured events is fairly simple. First you have to create a class extending the `IOnGestureListener` interface. Then, in the `OnCreate()` method of your `MainActivity` class, you have to create a gesture detector using your implementation of `IOnGestureListener`, as follows:

```
GestureOverlayView.IOnGestureListener myListener = new
MyImplementationOfIGestureListener();
_gestureDetector = new GestureDetector (this, myListener);
```

Finally, in your `MainActivity` class, implement, for example, the following method:

```
public override bool OnTouchEvent(MotionEvent e) {
  //Some business here
}
```

This will be triggered when an `OnTouch` event happens.

See also

See also *Chapter 10*, *Taking Advantage of the Android Platform*—to build even more advanced applications, and the following online resources to deepen your understanding of custom gestures:

`http://developer.android.com/training/gestures/detector.html`

`http://developer.android.com/training/gestures/multi.html`

`https://play.google.com/store/apps/details?id=com.davemac327.gesture.tool`

10

Taking Advantage of the Android Platform

In this chapter we will cover the following recipes:

- Mastering fragments
- Exploring Jelly Bean
- Exploring KitKat
- Integrating maps

Introduction

In the chapters— *Chapter 7*, *Using Android Services*, *Chapter 8*, *Mastering Intents – A Walkthrough* and *Chapter 9*, *Playing with Advanced Graphics*—you learned how to use some of the most advance modules that Android has to offer through Xamarin. In this chapter, you will learn how to use the new features, such as fragments and Speech recognition, that come with different versions of Android. Also, we will present some of the features that you can use with Lollipop, KitKat, Ice Cream Sandwich, and Jelly Bean versions of Android.

Mastering fragments

Fragments are yet another component that are offered by the Android platform, which can be used inside the Activity that you learned about in *Chapter 3*, *Mastering the Life and Death of Android Apps*. Fragments encapsulate behavior, functionality, or even user interface in an easy-to-reuse format. Also, fragments have their own life cycle so many of them can be combined in the same activity and they can communicate with each other very much like Intents do (which is described in *Chapter 8*, *Mastering Intents – A Walkthrough*).

In this recipe, we will see how to use two different fragments in the same activity.

Getting ready

Create a new solution named `Fragment_Project` and open the project of the same name that Xamarin Studio has created for you.

How to do it...

Let's take a look at the following steps:

1. Open the `Main.axml` file under the `Resources/Layout` folder and edit it to make it look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical" >
  <Button
    android:id="@+id/button1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Show first fragment"/>

  <Button
    android:id="@+id/button2"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Show second fragment" />

  <fragment
    android:name="Fragment_Project.FragmentOne"
    android:id="@+id/fragment_place"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

</LinearLayout>
```

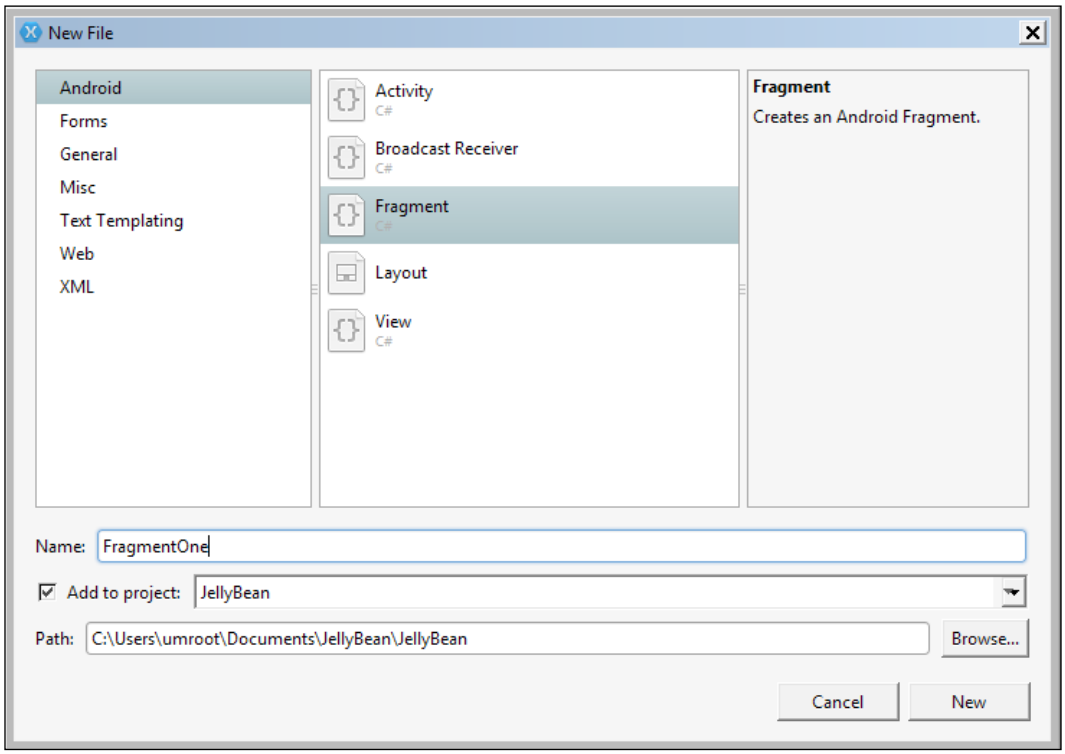
2. Create a new layout file named `fragment_one.axml` under the `Resources/Layout` folder and add the following content to it:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:background="#00ffff">

  <TextView
    android:id="@+id/textView1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    android:text="This is fragment No.1"
    android:textStyle="bold" />

</LinearLayout>
```

3. Create a new layout file named `fragment_two.axml` under the `Resources/Layout` folder and add the following content to it:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:background="#ffff00">

  <TextView
    android:id="@+id/textView2"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:text="This is fragment No.2"
    android:textStyle="bold" />

</LinearLayout>
```

4. Create a new Fragment named the `FragmentOne` class using the **New file** wizard, as shown in the following screenshot:



5. In the `FragmentOne` class, replace the `OnCreateView()` method by the following:

```
public override View OnCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
  return inflater.Inflate(Resource.Layout.fragment_one ,
  container, false);
}
```

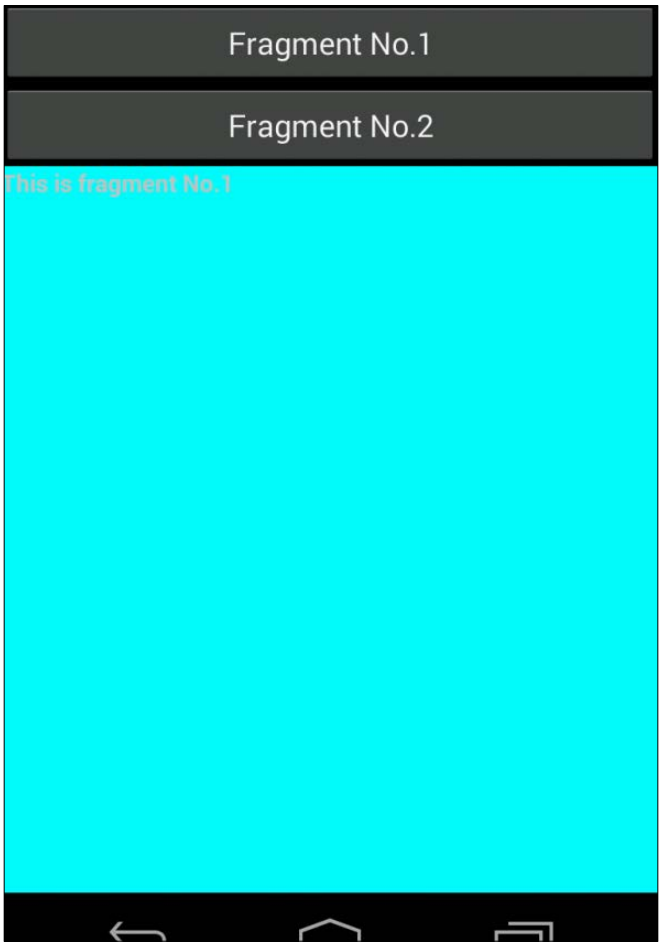
6. Create a new Fragment named the `FragmentTwo` class using the **New file** wizard.
7. Replace the `OnCreateView()` method of the `FragmentTwo` class:

```
public override View OnCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
  return inflater.Inflate(Resource.Layout.fragment_two ,
  container, false);
}
```

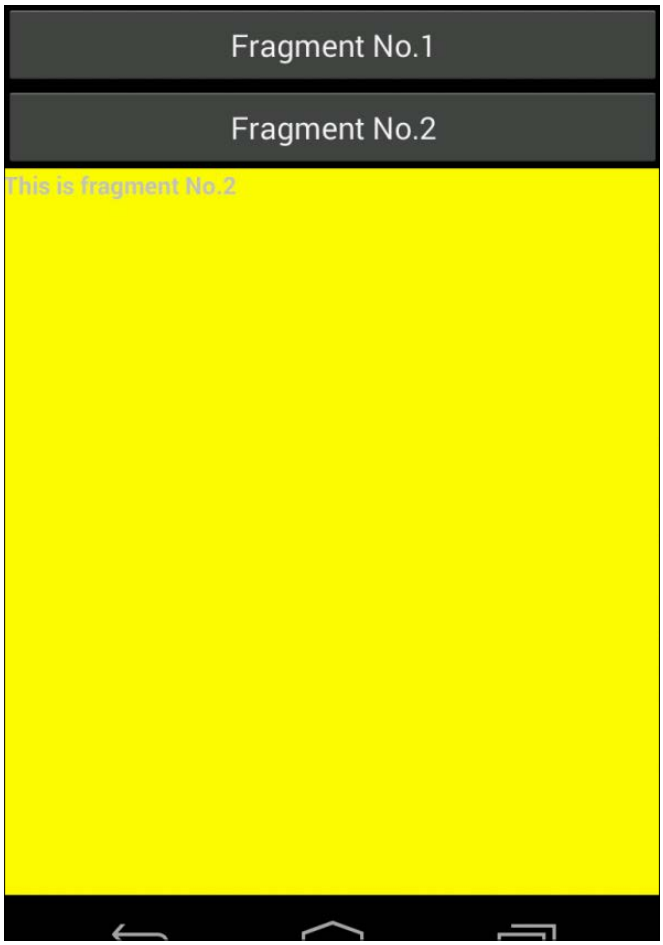
8. Modify the `MainActivity.cs` file to match the following:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);

  // Set our view from the "main" layout resource
  SetContentView (Resource.Layout.Main);

  Button button = FindViewById<Button>
  (Resource.Id.button1);

  button.Click += delegate {
    Fragment frag = new FragmentOne ();
    selectFrag(frag);
  };

  Button button2 = FindViewById<Button>
  (Resource.Id.button2);

  button2.Click += delegate {
    Fragment frag = new FragmentTwo ();
    selectFrag(frag);
  };

}

public void selectFrag(Fragment frag) {

  FragmentTransaction fragmentTx =
  this.FragmentManager.BeginTransaction();
  fragmentTx.Replace (Resource.Id.fragment_place, frag);
  fragmentTx.Commit ();

}
```

9. Run the application. The following screenshot shows the application at the initialization state:

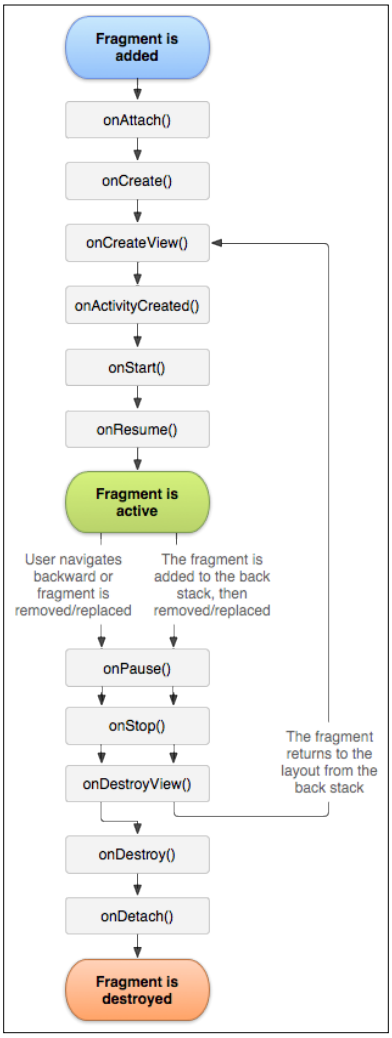


10. Click on **Fragment No.2**. We will see the following output:



How it works...

As we stated in the introduction of this recipe, fragments encapsulate behavior, functionality, or even the user interface in an easy-to-reuse format. Fragments are not self-sufficient though. Indeed, fragments absolutely need to be embedded in an activity and fragments completely depend on the activity's life cycle. For example, if the activity is stopped, so are all the fragments embedded in that activity. As long as the activity is running, the fragments contained in it follow their own life cycle. The life cycle of fragments, which is really similar to the life cycle of activity as seen in *Chapter 3*, *Mastering the Life and Death of Android Apps*, is described by the following figure:



source: http://developer.android.com/guide/components/fragments.html

The `OnCreate()`, `OnPause()`, and `OnCreateViews()` methods states of fragments reflect the exact same behavior as activities and might be overwritten by your application during the development process.

In order for fragments to be created and integrated to an activity, we must declare them in the `.axml` file of the said activity as shown earlier.

```
<fragment
  android:name="Fragment_Project.FragmentOne"
  android:id="@+id/fragment_place"
  android:layout_width="match_parent"
  android:layout_height="match_parent" />
```

Fragments can be replaced using a `FragmentTransaction` object:

```
FragmentTransaction fragmentTx =
this.FragmentManager.BeginTransaction();
fragmentTx.Replace (Resource.Id.fragment_place, frag);
fragmentTx.Commit ();
```

There's more...

Fragment transactions can be used for more than a simple replacement of a fragment. In this section, we will see how to add a new fragment programmatically and how to add fragment to the backstack.

Programmatically adding Fragments

In the previous example, we simply use a transaction to replace a fragment with another, when the user presses the button, however, we can do other great things with fragment transactions. Indeed, we can add a completely new fragment to an activity using the `add` method:

```
FragmentTransaction fragmentTx =
this.FragmentManager.BeginTransaction();
fragmentTx.Add (Resource.Id.fragment_place, new Fragment());
fragmentTx.Commit ();
```

Adding fragments to the backstack

We can also use the transaction mechanism to register the fragments in the back stack, allowing users to navigate back through fragments in fragments by pressing the **back** button of the phone. This is done exactly as you would navigate back through web pages when pressing previous in a web page:

```
FragmentTransaction fragmentTx =
this.FragmentManager.BeginTransaction();
fragmentTx.AddToBackStack ("MyFragmentName");
fragmentTx.Commit ();
```

Exploring Jelly Bean

For the next four recipes, we will introduce the features of Jelly Beans that each major version of Android brought. Depending on how far you want your application to be retro-compatible, you might want to know which feature came with each version. In this particular recipe, we will tackle the features proposed by Android 4.1—Jelly Bean—which was released in July 2012. The different versions of JellyBean spanning from Android 4.1 to Android 4.3 is the operating system of 37% of the Android phones as of July 2015.

Getting ready

To follow this recipe, create a new project called `JellyBean` and create a new AVD running Android 4.1 to deploy your code. In order to create an AVD running Android 4.1, you'll need to download and install the corresponding SDK using the Android SDK Manager.

How to do it...

1. Create a new C# class named `MyListener` that extends the `TimeAnimator.ITimeListener` class.
2. Create a `OnTimeUpdate()` method that takes `TimeAnimator`, `long totalTime`, and `long deltaTime` as arguments.
3. Add the following code in the `OnTimeUpdate()` method:

```
class MyListener : Java.Lang.Object,
TimeAnimator.ITimeListener {
  public  void OnTimeUpdate(TimeAnimator animation, long
  totalTime, long deltaTime) {
    Console.WriteLine("Something here " +  totalTime + " "
    + deltaTime);
  }
}
```

4. Add the following code in the `OnCreate()` method of your `MainActivity` class:

```
var myAnimator = new TimeAnimator();
myAnimator.SetTimeListener(new MyListener ());
myAnimator.Start();
```

5. Make your `MainActivity` class inherit the `Camera.IAutoFocusCallback` class in addition to the `Activity` class, as follows:

```
public class MainActivity : Activity,
Camera.IAutoFocusCallback
```

6. Add the `OnAutoFocus()` method to your `MainActivity` class that takes a `Boolean` value for success and a `Camera` object:

```
public void OnAutoFocus(bool success, Camera camera)
```

7. Add the following code in the newly created `OnAutoFocus()` method:

```
var newMediaActionPlayerFromJB = new MediaActionSound();
newMediaActionPlayerFromJB.Load(MediaActionSoundType.
ShutterClick);
newMediaActionPlayerFromJB.Play(MediaActionSoundType.
ShutterClick);
newMediaActionPlayerFromJB.Release();
```

8. Run you application, you should see **Something here** written on the console with the time of day.

How it works...

JellyBeans brought several new components, which we used in this recipe. The first one is the new `TimeAnimator` instance that came with the `ITimeListener` interface. Implementations of this interface are meant to detect every time an application changes the frame and triggers the `OnTimeUpdate()` method. Consequently, if an `Activity` class, and more particularly, an animation (you learned this in *Chapter 9*, *Playing with Advanced Graphics*), changes the frame, then you will be notified and can take actions. In our little example, this behavior has only been used to write a message in the console:

```
public  void OnTimeUpdate(TimeAnimator animation, long totalTime,
long deltaTime) {
  Console.log("Something here " +  totalTime + " " + deltaTime);
}
```

Another useful API introduced by JellyBeans is the `Camera.IAutoFocusCallback` instance that will inform you if the auto focus of the camera worked, or if, for some reasons, it didn't auto focus via the `success` Boolean value.

```
public void OnAutoFocus(bool success, Camera camera)
```

Finally, we used the new `MediaAction` sound with the following code in order to play the shutter sound when the `OnAutoFocus()` method is triggered:

```
var newMediaActionPlayerFromJB = new MediaActionSound();
newMediaActionPlayerFromJB.Load(MediaActionSoundType.
ShutterClick);
newMediaActionPlayerFromJB.Play(MediaActionSoundType.
ShutterClick);
newMediaActionPlayerFromJB.Release()
```

The first line creates a new `MediaActionSound` object and the second line takes care of loading the sound we want to play: `MediaActionSoundType.ShutterClick` instance. The third line will effectively trigger the playing of the sound while the last line will release the resources involved in playing the shutter sound.

There's more...

Before JellyBean, three majors version of Android have been released and two of them are still significantly used. However, they are powering less than 8% of all the Android devices altogether as of November 2015 and this percentage goes down every month.

Ice Cream Sandwich

Ice Cream Sandwich is the operating system used by 3.3% of the Android phones currently out there, and it corresponds to all Android versions except 4.0 to 4.1. In addition to a new layout system presented in *Chapter 3*, *Building a GUI*, ICS also introduced the Calendar API, event Intent, and a new media management layer. Finally, in ICS, we got **AndroidBeam** and some refined sensor APIs that we will discover in *Chapter 11*, *Using Hardware Interactions*.

GingerBread

GingerBread contains Android releases from 2.3.3 to 2.3.7 and owns 3.8% of the market share. It was the first version to handle **Near Field Communications** (**NFC**) and Bluetooth. GingerBread refined the media framework and speech recognition.

See also

Check out `http://developer.android.com/about/versions/jelly-bean.html` for a complete summary of JellyBean.

Exploring KitKat

KitKat is, at the time of writing this, the latest Android version on the market but should be very soon replaced by **Lollipop**. Kitkat runs 37.8 % of the Android phone as of November 2014, and therefore, you should be careful while using what it proposes as you are targeting a subset of Android users equipped with last generation phones.

Getting ready

To follow this recipe, create a new project called `KitKat` and create a new AVD running Android 4.4 to deploy your code. In order to create an AVD running Android 4.4, you'll need to download and install the corresponding SDK using the Android SDK Manager.

How to do it...

1. In your `MainActivity` class, add the following code in the `OnCreate()` method:

```
if(Build.VERSION.SdkInt >= BuildVersionCodes.Kitkat) {
  Console.WriteLine("I'am KitKat powered");
}
```

2. In your `MainActivity` class, add the following code in the `OnCreate()` method:

```
Uri uri = Android.Net.Uri.Parse("http://www.packtpub.com");
Intent myIntentToBeStarted = new Intent(Intent.ActionView,
uri);
AlarmManager alarmManager = (AlarmManager)
GetSystemService(AlarmService);
PendingIntent pendingIntent = PendingIntent.GetActivity
(this, 0, myIntentToBeStarted, PendingIntentFlags.
UpdateCurrent);
alarmManager.Set(AlarmType.ElapsedRealtimeWakeup,
SystemClock.ElapsedRealtime() + 10000,  pendingIntent);
```

3. Drag and drop button to your UI and then create a `Ressources/Layout.axml` file that contains the following:

```
<resources>
  <style name="KitKatTheme" parent=
  "android:Theme.Holo.Light">
    <item name="android:windowBackground">
    @color/xamgray</item>
    <item name="android:windowTranslucentStatus">
    true</item>
    <item name="android:windowTranslucentNavigation">
    true</item>
    <item name="android:fitsSystemWindows">true</item>
    <item name="android:actionBarStyle">
    @style/ActionBar.Solid.KitKat</item>
  </style>

  <style name="ActionBar.Solid.KitKat" parent=
  "@android:style/Widget.Holo.Light.ActionBar.Solid">
    <item name="android:background">@color/xampurple</item>
  </style>
</resources>
```

4. Run your application. You will see a line printed on the console confirming that the device is powered by KitKat, and ten seconds later, a webpage will open redirecting to the `https://www.packpub.com` website. You can also notice some modification to the theme of the application as compared to what we saw earlier. The theme is now translucent.

How it works...

With three simple tags to place in your `ressources/layout.axml` file, Android KitKat allows us to use the translucent theme that comes with KitKat in your own application:

- `windowTranslucentStatus`: If this is set to `true`, the status bar at the top will be translucent.
- `windowTranslucentNavigation`: This is the same as `windowTranslucentStatus` but for the navigation bar.
- `fitsSystemWindows`: If your applications are translucent and so is the OS, then there is a possibility of overlapping. In order to avoid overlapping between translucent applications or applications and the OS, you have to set this argument to `true`.

In this recipe, we also use an alarm service to browse to the `packtpub.com` website every half an hour. In KitKat, the management of the alarm service has been improved in order to group several applications that have to be woken up in the same time interval. For example, you can run a bunch of health-check applications every half an hour, and therefore, enhance the battery life, compared to always keeping them active or activating them one by one.

In order to specify the window of time in which your applications have to be awoken at least once, you can use the `SetWindow()` method of the `AlarmManager` instance as we did in this recipe:

```
alarmManager.SetWindow (AlarmType.Rtc,
AlarmManager.IntervalHalfHour, AlarmManager.IntervalHour,
myIntentToBeStarted);
```

See also

Go to `http://developer.android.com/about/versions/kitkat.html` for a complete summary of KitKat features. You can also refer to *Chapter 9*, *Playing with Advanced Graphics*, for the animations functionalities and *Chapter 11*, *Using Hardware Interactions*, to learn about the sensor interactions of KitKat.

Integrating maps

In *Chapter 8*, *Mastering Intents – A Walkthrough*, we use an intent aiming to redirect us to the google map application at a specific address. In this recipe, we will see all the options available with the Google Map and Google Street Intents.

Getting ready

Create a new project named `GoogleMapsIntents`.

How to do it...

1. Drag and drop three different buttons in your application.
2. Map the newly created three buttons on press events with the following three code examples:

   Button 1:

```
var geoUriToMontreal = Android.Net.Uri.Parse
("geo: 45.5088400, -73.5878100?z=4x");
var mapIntentToMontreal = new Intent (Intent.ActionView,
geoUriToMontreal);
StartActivity (mapIntentToMontreal);
```

   Button 2:

```
var geoUriTotheEiffelTower = Android.Net.Uri.Parse
("geo:0,0?q=Eiffel tower&z=2x");
var mapIntentToTheEiffelTower = new Intent
(Intent.ActionView, geoUriTotheEiffelTower);
StartActivity (mapIntentToTheEiffelTower);
```

   Button 3:

```
var streetViewUriToCambridge = Android.Net.Uri.Parse
("google.streetview:cbll=42.374260,-71.120824,90,0,1.0");
var streetViewIntentToCambridge = new Intent
(Intent.ActionView, streetViewUriToCambridge);
StartActivity (streetViewIntentToCambridge);
```

3. Start your application. Every time you press a button, the Google Map or the Google Street View application will open to a different location.

How it works...

As the working principles of intents have been explained in *Chapter 8*, *Mastering Intents – A Walkthrough*, I will only go through the arguments and options you can use with the Google Map and Google Street intents:

- `geo:latitude,longitude`: This is the classic and oldest way to use the Google Map Intent. It will open Google Maps on a latitude, longitude location.
- `geo:latitude,longitude?z=zoomlevel`: You can combine the previous latitude, longitude with a zoom level. Usable zoom levels are 1.0 for normal zoom, 2.0 for 2.x zoom, 3.0 for 4.x zoom, and so on.

- `geo:0,0?q=any+search`: This particular combination, with longitude and latitude set to zero and followed by the `?q=` value will allow you to use Google Map Intents as you use the Google Maps website. For example, you search for "`McDonads near New york city`" or complete addresses such as "`1 Rue de la République, Paris`".

  The Google Street View intent slightly differs as it has to start with "`google.streetview:cbll=`" and then arguments have to be placed as follows:

  `Latitude`, `longitude`, `yaw`, `pitch`, and `zoom`.

The two new arguments, `yaw` and `pitch`, are the horizontal orientations in degrees from north and the vertical orientation (90% to the sky, -90% to the floor).

See also

If you wish to modify the map's look and feel for a deeper integration with your application, you can take a look at `http://developer.xamarin.com/guides/android/platform_features/maps_and_location/maps/part_2_-_maps_api/`.

11
Using Hardware Interactions

In this chapter, we will see the following recipes:

- Beaming messages with NFC
- Using the accelerometer and other sensors
- Using Bluetooth
- Using GPS

Introduction

Until now, we have only investigated the software capacities of Android with the exception of the camera. In this chapter, we will put the emphasis on the hardware interactions. Android developers build APIs, for each hardware type, so that we can use any hardware in the same way and Xamarin allows us to use those APIs in C#.

Beaming messages with NFC

NFC stands for **Near Field Communication** and allows NFC-equipped owners to exchange data by contact (or near contact) of their phones. In Android, we can use NFC through **Beam**. Using Beam, users can exchange a small amount of data, such as web bookmarks, contact info, directions, YouTube videos links, and so on. NFC is more convenient than other wireless technologies, such as Bluetooth or Wi-Fi for quickly sending data as no pairing or authentication is required.

The minimum SDK version to use NFC is level 10.

Getting ready

Create a new Solution named `Beam_Project`. To test this recipe, you'll need to deploy your application on two different NFC-enabled phones, as the emulator doesn't support NFC. Also, we will need to add the NFC permission.

How to do it...

To complete this recipe, you need to perform the following steps:

1. Import the NFC APIs by adding `using Android.Nfc` at the top of the `MainActivity.cs` file.
2. Import text capacities by adding the `System.Text` element at the top of the `MainActivity.cs` file.
3. In the `MainActivity.cs` file, implement the `NfcAdapter.ICreateNdefMessageCallback` and `NfcAdapter.IOnNdefPushCompleteCallback` interfaces, as shown in the following code:

```
public class MainActivity : Activity,
NfcAdapter.ICreateNdefMessageCallback,
NfcAdapter.IOnNdefPushCompleteCallback
```

4. Add a `TextView` element in the `Main.axml` file and create a reference to it in your `OnCreate()` method:

```
TextView textView =  FindViewById<TextView>
(Resource.Id.mytextView);
```

5. Test if the device is NFC enabled by adding the following snippet in the `OnCreate()` method of your `MainActivity` class:

```
NfcAdapter nfcAdapter = NfcAdapter.GetDefaultAdapter(this);
if (nfcAdapter == null) {
  Console.WriteLine ("Not NFC enabled");
  textView.Text = "Not NFC";
}
else {
  textView.Text = "NFC OK";
  nfcAdapter.SetNdefPushMessageCallback (this, this);
  nfcAdapter.SetOnNdefPushCompleteCallback (this, this);
}
```

6. Create a private constant named `message_id` and assign its value as `1`:

```
private const int message_id = 1;
```

7. Implement the `OnNdefPushComplete()` method defined by the `NfcAdapter.IOnNdefPushCompleteCallback` interface as follows:

```
public void OnNdefPushComplete (NfcEvent myNfcEvent) {
  Handler myHandler = new Handler ();
  myHandler.ObtainMessage (message_id).SendToTarget ();
}
```

8. Implement the `CreateNdefMessage()` method defined by the `NfcAdapter.ICreateNdefMessageCallback` interface as follows:

```
public NdefMessage CreateNdefMessage (NfcEvent myNfcEvent)
{
  String myBeamText = ("MyAwesomeBeamMessage");

  byte [] mimeBytes = Encoding.UTF8.GetBytes
  ("application/com.example.android.beam");

  NdefRecord ndefRecord = new NdefRecord (
    NdefRecord.TnfMimeMedia, mimeBytes, new byte [0],
  Encoding.UTF8.GetBytes (myBeamText));

  NdefMessage myBeamMessage = new NdefMessage (new
  NdefRecord[]{ ndefRecord  });

  return myBeamMessage;
}
```

9. Create a `displayMessage()` method that takes an `Intent` object as an argument:

```
protected void displayMessage (Intent intent) {
  IParcelable [] parcelMessage =
  intent.GetParcelableArrayExtra
  (NfcAdapter.ExtraNdefMessages);

  NdefMessage msg = (NdefMessage) parcelMessage [0];

  TextView textView =  FindViewById<TextView>
  (Resource.Id.mytextView);

  textView.Text = Encoding.UTF8.GetString (msg.GetRecords
  () [0].GetPayload ());
}
```

10. Override the `OnResume()` and `OnNewIntent()` methods as follows:

```
protected override void OnResume () {
  base.OnResume ();
  if (NfcAdapter.ActionNdefDiscovered == Intent.Action) {
    displayMessage (Intent);
  }
}

protected override void OnNewIntent (Intent intent) {
  Intent = intent;
}
```

11. Deploy your application and two different phones, launch the application on those phones, and then put their NFC chips close to each other. You'll receive a **New Tag Collected** message that contains the name of your APK.

Ensure that both phones have their NFC sharing settings set to activate and that you have to request the NFC permission in your `manifest.xml` file. Finally, both screens must be unlocked.

How it works...

Android Beam allows us to beam messages from one phone to another as long as they are close enough to each other (almost in contact) and own an NFC chip. The messages that can be sent via beam are **NFC Data Exchange Format Message** (**NdefMessage**), which are composed of one or more NdefRecords. This is a convenient way to encapsulate data in a binary format for NFC transfer. NdefRecords have the following four attributes:

- **Tnf**: This is a type of data
- **Type**: This is the type of application that should receive the message
- **Id**: This is the ID of the message
- **Playload**: This is the actual content

The name of the attributes can be confusing. The `Tnf` attribute defines what the type of data is. For example, we can use `TnfMimeMedia` or `TnfAbsoluteUri`, which are public constants of the `NdefRecord` class. The `Type` attribute refers to the application that should handle your message, such as `application/com.example.android.beam`.

To build our `NdefRecords()` method we used the following constructor:

```
public NdefRecord (short tnf, byte[] type, byte[] id, byte[]
payload)
```

Also, we used the following code:

```
byte [] type = Encoding.UTF8.GetBytes
("application/com.example.android.beam");

NdefRecord mimeRecord = new NdefRecord (
  ndefRecord.TnfMimeMedia, type , new byte [0],
  Encoding.UTF8.GetBytes (myBeamText));
```

In this snippet, we first create an array of the byte representing `"application/com.example.android.beam"`. This array will be used as the `Type` argument of the `NdefRecords()` method. Then, we call the constructor with `NdefRecord.TnfMimeMedia` as the `Tnf` argument, our newly created array of bytes named `type`, and `MyAwesomeBeamMessage` is also transformed into bytes by means of the `Encoding` helper.

We now have every piece required to construct our `NdefMessage()` method with the following:

```
NdefMessage myBeamMessage = new NdefMessage (new NdefRecord[]{
ndefRecord });
```

In order to receive beam messages, we created the `displayMessage()` method that takes an `Intent` as an argument:

```
protected void displayMessage (Intent intent) {
  IParcelable [] parcelMessage = intent.GetParcelableArrayExtra
  (NfcAdapter.ExtraNdefMessages);

  NdefMessage msg = (NdefMessage) parcelMessage [0];

  TextView textView =  FindViewById<TextView>
  (Resource.Id.mytextView);

  textView.Text = Encoding.UTF8.GetString (msg.GetRecords ()
  [0].GetPayload ());
}
```

Then, the `displayMessage()` method is called by the `OnResume()` method if our application is resumed by an NFC call:

```
protected override void OnResume () {
  base.OnResume ();
  if (NfcAdapter.ActionNdefDiscovered == Intent.Action) {
    displayMessage (Intent);
  }
}
```

In the `displayMessage()` method, we first get an array of the `parcelable` object and cast the first one as `NdefMessage`. Indeed, this method will be called only in case of beam contact, and we can only send one beam message at a time. Therefore, we can safely execute `(NdefMessage) parcelMessage [0]`. Finally, we assign our `textview` text with the `Payload` argument of the first record of our message.

The activity we created is able to send and receive beam messages because it implements the related interface and will register itself as a beam application.

See also

You can also visit the following links:

- `http://developer.android.com/guide/topics/connectivity/nfc/nfc.html` for the in depth specification of NFC for Android
- `http://open-nfc.org/wp/` if you would like to use NFC on your emulators or `https://github.com/xamarin/monodroid-samples/blob/master/AndroidBeamDemo/AndroidBeamDemo/Activity.cs`

Using the accelerometer and other sensors

The Accelerometer records the device acceleration in three dimensions, that is, along the *x*, *y*, and *z* axis. Almost all phones are now shipped with an accelerometer as it's a pretty old technology. Indeed, the very first phone to have one was the Apollo 11 Lunar Module. A phone with wheels. Actually, the first mainstream and terrestrial phone equipped with this technology was the Samsung SCH-S310 in 2005 on which you could dial numbers by writing them in the air. Today's application of the accelerometer are almost all related to games. In this recipe, we will detect if the phone is falling; this can be useful for the elderly in the sense that we can be warned of emergencies automatically.

Getting ready

To follow this recipe, create a new project called `Accelerometer`. Similar to the previous recipe, you'll have to deploy your code on a physical phone as the emulator doesn't support the accelerometer.

How to do it...

Now, we will see how to use the accelerometer and other sensors:

1. Add the `using Android.Hardware` and `using System.Text` instances at the top of your `MainActivity` class.

2. Add a `textview` element in your `Resources/layout/Main.axml` file and name it `myTextView`.
3. Make your `MainActiviy` class implement the `ISensorEventListener` interface:

```
public class MainActivity : Activity, ISensorEventListener
```

4. Add the following six attributes to your `MainActivity` class:

```
private static readonly object myLock = new object();
private SensorManager mySensorManager;
private float x=0, y=0, z=0;
private TextView myTextView;
```

5. Modify your `OnCreate()` method so that it looks like the following:

```
protected override void OnCreate(Bundle bundle) {
  base.OnCreate(bundle);
  SetContentView(Resource.Layout.Main);
  mySensorManager = (SensorManager)
  GetSystemService(SensorService);
  myTextView = FindViewById<TextView>
  (Resource.Id.myTextView);
  mySensorManager.RegisterListener(this, mySensorManager.
  GetDefaultSensor(SensorType.Accelerometer),
  SensorDelay.Ui);


}
```

6. Implement the `OnSensorChanged` function from the `ISensorEventListener` interface:

```
public void OnSensorChanged(SensorEvent e) {
  StringBuilder text = new StringBuilder ();

  lock (myLock) {
    float currentX = e.Values [0];
    float currentY = e.Values [1];
    float currentZ = e.Values [2];

    // First time
    if (z == 0 && y == 0 && z == 0) {

      z = currentZ;
      y = currentY;
      x = currentX;

    }
```

```
        else if (Math.Abs (currentX / x) > 2 || Math.Abs
        (currentY / y) > 2 || Math.Abs (currentZ / z) > 2) {
          z = currentZ;
          y = currentY;
          x = currentX;

          text.Append ("Is Probably Falling");
        }

        text.Append("x = ")
        .Append(currentX)
        .Append(", y=")
        .Append(currentX)
        .Append(", z=")
        .Append(currentZ);

        myTextView.Text = text.ToString();
      }
    }
```

7. Implement the `OnAccuracyChanged()` function from `ISensorEventListener`, which receives `[Android.Runtime.GeneratedEnum] SensorType sensor` and `[Android.Runtime.GeneratedEnum] SensorStatus` as parameters and leaves its body blank.
8. Deploy your application to a physical phone, run the application and move it around for the `textview` element to be refreshed with new values. If you move the phone fast enough, **The Label Is Probably Falling** will appear before the *x*, *y*, and *z* accelerations.

How it works...

Each sensor's data can be accessed through background services. In order for our application to be fed by the data of a sensor, we need to create a reference to the `SensorManager` instance. Finally, we can register our application via a `Listener` instance, so the `OnSensorChanged()` method will be triggered every time the sensor changes its value.

To acquire a reference to the `SensorManager` instance, we simply use the `GetSystemService()` helper as follows:

```
mySensorManager = (SensorManager) GetSystemService(SensorService);
```

Then, we register our application to receive the data from the accelerometer as follows:

```
mySensorManager.RegisterListener(this, mySensorManager.
GetDefaultSensor(SensorType.Accelerometer),
SensorDelay.Ui);
```

In the `OnSensorChanged()` method, we implemented a C# thread-safe lock as this method can be retriggered before it finishes its execution:

```
lock (myLock)
```

Then, we retrieve the values of the acceleration for the three axes and compare them with the previous values. If the division results is more than two (value of accelerometer) on one of the three axes, it likely means that the phone is falling:

```
float currentX = e.Values [0];
float currentY = e.Values [1];
float currentZ = e.Values [2];

// First time
if (z == 0 && y == 0 && z == 0) {

  z = currentZ;
  y = currentY;
  x = currentX;

}
else if (Math.Abs (currentX / x) > 2 || Math.Abs (currentY / y) >
2 || Math.Abs (currentZ / z) > 2) {
  z = currentZ;
  y = currentY;
  x = currentX;

  text.Append ("Is Probably Falling");
}
```

There's more...

Android phones can be equipped with a very large range of sensors that can be used in the same way as the accelerator. They are categorized in three categories: motion, position, and environment. For each of them, you'll have to register your activity as follows:

```
mySensorManager.RegisterListener(this, mySensorManager.
GetDefaultSensor(SensorType.MY_TYPE),
SensorDelay.MY_FREQ);
```

Here `SensorType.MY_TYPE` and `SensorType.MY_FREQ` have to be adapted to your needs.

While we can easily discover the different sensors using the Intellisense of Xamarin, the data they return is only float. In what follows, I report what those floats represent.

Motion sensors

Here, I present the different values that can be retrieved for motion sensors:

- Accelerometer:
  - `SensorEvent.value[0]`: This is the acceleration force on *x* axis
  - `SensorEvent.value[1]`: This is the acceleration force on *y* axis
  - `SensorEvent.value[2]`: This is the acceleration force on *z* axis
- Gravity:
  - `SensorEvent.value[0]`: This is the gravity force on *x* axis
  - `SensorEvent.value[1]`: This is the gravity force on *y* axis
  - `SensorEvent.value[2]`: This is the gravity force on *z* axis
- Gyroscope:
  - `SensorEvent.value[0]`: This is the rotation on *x* axis
  - `SensorEvent.value[1]`: This is the rotation on *y* axis
  - `SensorEvent.value[2]`: This is the rotation on *z* axis
- LinerAcceleration:
  - `SensorEvent.value[0]`: This is the acceleration force on *x* axis without gravity
  - `SensorEvent.value[1]`: This is the acceleration force on *y* axis without gravity
  - `SensorEvent.value[2]`: This is the acceleration force on *z* axis without gravity
- RotationVector:
  - `SensorEvent.value[0]`: This is the rotation on *x* axis
  - `SensorEvent.value[1]`: This is the rotation on *y* axis
  - `SensorEvent.value[2]`: This is the rotation on *z* axis
- SignificantMotion: This doesn't have any value but triggers the `onSensorChanged()` method when a significant movement is detected.
- StepCounter:
  - `SensorEvent.value[0]`: This is the number of steps since the last activation of the sensor
- StepDetector: This doesn't have any value but triggers the `onSensorChanged()` method when a step is detected

Position sensors

In what follows, I present the different values that can be retrieved for position sensors:

- GameRotationVector:
  - `SensorEvent.value[0]`: This is the rotation on *x* axis
  - `SensorEvent.value[1]`: This is the rotation on *y* axis
  - `SensorEvent.value[2]`: This is the rotation on *z* axis
- GeoMagneticRotationVector:
  - `SensorEvent.value[0]`: This is the rotation on *x* axis
  - `SensorEvent.value[1]`: This is the rotation on *y* axis
  - `SensorEvent.value[2]`: This is the rotation on *z* axis
- MagneticField:
  - `SensorEvent.value[0]`: This is the geomagnetic field on *x* axis
  - `SensorEvent.value[1]`: This is the geomagnetic field on *y* axis
  - `SensorEvent.value[2]`: This is the geomagnetic field on *z* axis
- Proximity:
  - `SensorEvent.value[0]`: This is the distance with the object

Environment sensors

Here, I present the different values that can be retrieved for environment sensors:

- AmbientTemperature:
  - `SensorEvent.value[0]`: This is the air temperature in degree Celsius
- Light:
  - `SensorEvent.value[0]`: This is the illuminance in Lx
- Pressure:
  - `SensorEvent.value[0]`: This is the air pressure in hPa
- RelativeHumidity:
  - `SensorEvent.value[0]`: This is the air humidity in percentage

Refresh rate of sensors

Android provides four different rates at which you can refresh your sensors' data. The faster rate is the most accurate and will consume much more battery. You have to evaluate your needs and choose the right trade-off between accuracy and battery consumption for your applications. The four different rates are as follows:

- `SensorDelay.Fastest`: This rate is the fastest possible rate
- `SensorDelay.Game`: This is a suitable rate for games
- `SensorDelay.Normal`: This is a suitable rate for screen orientation change
- `SensorDelay.UI`: This is a suitable rate for graphical interfaces

Listen only when needed

In order to enhance the battery life of the phones your applications are running on, it is a good practice, if possible, to listen to the sensors only when your application is displayed. To accomplish this, we can register ourselves using the `OnResume()` method and unregister ourselves using the `OnPause()` method. Consequently, our application will only consume the sensors' data when displayed:

```
protected override void OnResume() {
  base.OnResume();
  mySensorManager.RegisterListener(this, mySensorManager.
  GetDefaultSensor(SensorType.Accelerometer),
  SensorDelay.Ui);
}

protected override void OnPause() {
  base.OnPause();
  mySensorManager.UnregisterListener(this);
}
```

See also

You can also visit the following links to learn how the values of these sensors are computed:

- `http://developer.android.com/guide/topics/sensors/sensors_motion.html`
- `http://developer.android.com/guide/topics/sensors/sensors_position.html`
- `https://github.com/xamarin/monodroid-samples/tree/master/AccelerometerPlay`
- `http://developer.android.com/guide/topics/sensors/sensors_environment.html`

Using Bluetooth

Bluetooth is a well-known wireless communication standard and almost all phones embed a Bluetooth controller. In this recipe, we'll connect ourselves to a Bluetooth object and send/receive data from it.

Getting ready

To follow this recipe, create a new project called `Bluetooth`, and you'll need to deploy your application on a physical phone as the emulator doesn't support Bluetooth. Also, you'll need to bond your phone to the target Bluetooth device of your choice—using the Bluetooth Manager of Android—prior to launching the app.

How to do it...

Here are the steps to complete this recipe:

1. Add `using Android.Bluetooth` and `Java.Util` at the top of your `MainActivity` class. In addition, you will need the Bluetooth permissions.
2. Add the following attribute to your `MainActivity` class:

```
BluetoothAdapter myBTAdapter;
BluetoothDevice myBTDevice;
BluetoothSocket myBTSocket;
Byte[] buffer = new Byte[124];
```

3. Construct a reference to the `BluetoothAdapter` instance in your `OnCreate()` method:

```
myBTAdapter = BluetoothAdapter.DefaultAdapter;
if (myBTAdapter == null || myBTAdapter.IsEnabled) {
  Console.WriteLine ("Can't do anything");
}
```

4. Create a `connectToDevice()` method, which takes the name of the device to which you want to send data to in an argument:

```
protected void connectToDevice(String name) {
  myBTDevice = (from bd in myBTAdapter.BondedDevices
  where bd.Name == name select bd).FirstOrDefault();
}
```

5. Create an `openSocket()` method:

```
async protected void openSocket() {
  myBTSocket = myBTDevice.CreateRfcommSocketToServiceRecord
  (UUID.FromString("00001101-0000-1000-8000-
  00805f9b34fb"));
  await myBTSocket.ConnectAsync();
}
```

6. Create a `readData()` method:

```
async protected void readData() {
  await myBTSocket.InputStream.ReadAsync (buffer, 0, 124);
}
```

7. Create a `WriteData()` method:

```
async protected void writeData() {
  //Fill buffer w/ something
  await myBTSocket.OutputStream.WriteAsync (buffer, 0,
  124);
}
```

8. Use these methods to read, write and to communicate with your paired device.

How it works...

Bluetooth communication can occur by socket reading and writing. To create the socket, or open such a socket, we have to retrieve the default Bluetooth adapter with the following code:

```
BluetoothAdapter.DefaultAdapter;
if (myBTAdapter == null || myBTAdapter.IsEnabled) {
    Console.WriteLine ("Can't do anything");
}
```

Test that the Bluetooth adapter is not `null`, meaning that the phone has a Bluetooth controller and that the Bluetooth is enabled.

Note that you have to ask permission to use Bluetooth like we did with the NFC in the Android `Manifest.xml` file.

Then, we use the query form we saw in *Chapter 5*, *Using On-Phone Data*, to retrieve a reference to the Bluetooth device we've paired using its name:

```
myBTDevice = (from bd in myBTAdapter.BondedDevices
  where bd.Name == name select bd).FirstOrDefault();
```

Now that we have our Bluetooth device, we can open a socket using the `CreateRfcommSocketToServiceRecord()` method:

```
myBTSocket = myBTDevice.CreateRfcommSocketToServiceRecord
(UUID.FromString("00001101-0000-1000-8000-00805f9b34fb"));
myBTSocket.ConnectAsync();
```

The `UUID` instance used in this recipe is the default UUID for RFCOMM. Note that you could either search the specification of your devices for non-default UUID if any or generate your own.

The `ConnectAsync()` method is asynchronous, as its name suggests, and we use the C# keywords `async` and `await` to secure our statements.

Finally, we can read and write on the socket using the `myBTSocket.InputStream.ReadAsync()` and `myBTSocket.OutputStream.WriteAsync()` methods.

See also

Go to `https://github.com/xamarin/monodroid-samples/tree/master/BluetoothChat` for a very interesting, yet lengthy to write and understand, chat based on Bluetooth.

Using GPS

GPS is used by many applications in order to localize the phone and propose recommendations or orientation depending on where the user is. In this recipe, we'll see how to retrieve the longitude and the latitude and then compute a street address.

Getting ready

To follow this recipe, create a new project called `Gps`, and you'll need to deploy your application on a physical phone as the emulator doesn't support GPS.

How to do it...

Here are the steps required to complete this recipe:

1. Add four `TextView` elements to your `Main.axml` file, so it looks like the following:

2. Add `using Android.Locations`, `using System.Collections.Generic`, and `using System.Text` instances at the top of your `MainActivity` class. In addition, you have to request for GPS permission.
3. Create these four attributes in the `MainActivity` class:

```
Location myCurrentLocation;
        LocationManager myLocationManager;
        TextView myLocationTextView;
        TextView myAddressTextView;
        String myLocationProvider = "";
```

4. Create a `BuildMyLocationManager()` method:

```
private void BuildMyLocationManager() {
  myLocationManager = (LocationManager)
  GetSystemService(LocationService);
  Criteria criteriaForLocationService = new Criteria {
    Accuracy = Accuracy.Fine
  };

  IList<String> acceptableLocationProviders =
  myLocationManager.GetProviders
  (criteriaForLocationService, true);

  if (acceptableLocationProviders != null &&
  acceptableLocationProviders.Count > 0) {
    myLocationProvider = acceptableLocationProviders[0];
  }
}
```

5. Modify your `OnCreate()` method to assign the `myLocationTextView` and `myAddressTextView` instances and call the `BuildMyLocationManager()` method:

```
protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);

  SetContentView (Resource.Layout.Main);

  myLocationTextView = FindViewById<TextView>
  (Resource.Id.myLocationTextView);
           myAddressTextView = FindViewById<TextView>(Resource.
Id.myAddressTextView);

  BuildMyLocationManager ();

}
```

6. Make your activity implement the `ILocationListener` instance:

```
public class MainActivity : Activity, ILocationListener
```

7. Add the following three methods and leave their body blank:

```
public void OnProviderDisabled(string provider) {}

public void OnProviderEnabled(string provider) {}

public void OnStatusChanged(string provider, Availability
status, Bundle extras) {}
```

8. Add the `OnLocationChanged()` method, which takes a `location` instance in the argument to your `MainActivity` class:

```
async public void OnLocationChanged(Location location) {

  myCurrentLocation = location;
  myLocationTextView.Text = "N " +
  myCurrentLocation.Latitude + ", W" +
  myCurrentLocation.Longitude;

  if (myCurrentLocation == null) {
    myAddressTextView.Text = "Can't determine the current
    address.";
    return;
  }

  Geocoder geocoder = new Geocoder(this);
```

```
    IList<Address> addressList = await
    geocoder.GetFromLocationAsync(
      myCurrentLocation.Latitude,
    myCurrentLocation.Longitude, 1);

    if (addressList != null && addressList.Count == 1) {

      StringBuilder myCurrentAddressBuilder = new
      StringBuilder();
      for (int i = 0; i < addressList[0].MaxAddressLineIndex
      ; i++) {
        myCurrentAddressBuilder.Append
        (addressList[0].GetAddressLine(i))
        .AppendLine(",");
      }

      myAddressTextView.Text =
      myCurrentAddressBuilder.ToString();
    }
  }
```

9. Override the `OnPause()` method to stop listening to the location manager:

```
protected override void OnPause() {
  base.OnPause();
  myLocationManager.RemoveUpdates(this);
}
```

10. Override the `OnResume()` method to start listening to the location manager:

```
protected override void OnResume () {
  base.OnResume ();
  myLocationManager.RequestLocationUpdates
  (myLocationProvider, 0, 0, this);
}
```

11. Deploy your application and move around, you will see that the coordinates will change. If you move enough, the address will be updated too.

Ensure that you have GPS activated and that you have requested `AccessFineLocation` and `AccessCoarseLocation` in your `AndroidManifest.xml` file.

How it works...

The very first thing to do in order to retrieve the location of the phone is to get a reference to `LocationManger`, very much like we did for the sensors and Bluetooth. As in the previous recipe, the location manager is a system service that we can access with the `GetSystemService()` helper method:

```
myLocationManager = (LocationManager)
GetSystemService(LocationService);
```

Then, we try to retrieve a location provider using the `GetProviders()` method of the `myLocationManager` class.

```
IList<String> acceptableLocationProviders =
myLocationManager.GetProviders(criteriaForLocationService, true);
```

This method has two arguments, the first one is a `Criteria` object defining the accuracy at which we want to locate the phone. The `Criteria` objects can be built as follows:

```
Criteria criteriaForLocationService = new Criteria {
  Accuracy = Accuracy.Fine
};
```

Also, the accuracy criteria can have six different possible values, which are as follows:

- Fine
- Coarse
- High
- Low
- Medium
- No Requirement (no filters)

Once again, the more precision you want, the more battery you'll consume.

The second argument of the `myLocationManager.GetProviders` method stands for `enabledOnly` and ensures that the returned `locationProviders` instance are enabled. To conclude with `BuildMyLocationManager`, we assign the `myLocatonProvider` instance with the value returned by the `myLocationManager.GetProviders()` method, if any.

In the `OnLocationChanged()` method, we try to first set the `myLocationTextView` instance with the current coordinates:

```
myCurrentLocation = location;
myLocationTextView.Text = "N " + myCurrentLocation.Latitude +", W"
+ myCurrentLocation.Longitude;
```

Then, we use a `Geocoder` object to reverse the coordinate into a street address:

```
Geocoder geocoder = new Geocoder(this);
IList<Address> addressList = await geocoder.GetFromLocationAsync(
myCurrentLocation.Latitude, myCurrentLocation.Longitude, 1);
```

The `GetFromLocationAsync()` method has three arguments, which are `latitude`, `longitude`, and the number of nearby addresses we want to retrieve. In this recipe, we retrieve only one nearby address.

Finally, if the `GetFromLocationAsync()` method returns an address, we construct a `String` value with a `StringBuilder` instance that will aggregate all the different lines of the address:

```
if (addressList != null && addressList.Count == 1) {

  StringBuilder myCurrentAddressBuilder = new StringBuilder();
  for (int i = 0; i < addressList[0].MaxAddressLineIndex ; i++) {
    myCurrentAddressBuilder.Append
    (addressList[0].GetAddressLine(i))
    .AppendLine(",");
  }

  myAddressTextView.Text = myCurrentAddressBuilder.ToString();
}
```

There's more...

The `Geocode` objects are really handful and easy to use. In addition to the coordinate to street address translation, they can do street address to coordinate translation as well.

Reverse geocode address

In order to reverse a geocode address, we have to use the `GetFromLocationAsync()` method of a `Gecode` object with the street address and the number of the nearby address you want in the parameter:

```
addressList = await geocoder.GetFromLocationAsync (
  "895 Rue de la Gauchetière Ouest, Montréal, QC H3B 2M4",
  1);

if (addressList != null && addressList.Count == 1) {
  if (addressList [0].HasLatitude && addressList
  [0].HasLongitude) {

    myLocationTextView.Text = "N " + addressList
    [0].Latitude +
    ", W" + addressList [0].Longitude;
  }
}
```

You can test that the address has coordinates with the `HasLatitude()` and `HasLongitude()` methods before you use them.

See also

You can find the complete documentation about what geocoder can achieve at `http://developer.android.com/training/location/index.html`.

12
Debugging and Testing

In this chapter, we will cover the following recipes:

- Debugging in an emulator
- Debugging on a phone
- Unit testing

Introduction

So far, we have been interested in understanding and applying key concepts behind the construction of Android applications based on Xamarin, and if you've followed the code sample provided in this book, you wouldn't feel the need to master the Xamarin debugger. However, when building you own applications you'll need to know how to efficiently debug and test.

Debugging in an emulator

In this recipe, you'll learn how to use the breakpoint and the step-by-step debugging capacities of Xamarin Studio. This will be particularly helpful when facing a bug or to inspect variables values on the fly.

Getting ready

Create a new solution named `Debugging`.

How to do it...

To begin with debugging, you'll need to perform the following steps:

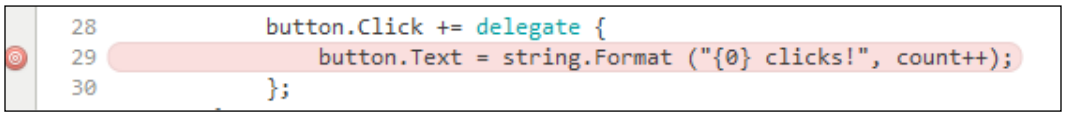
1. Your `MainActivity` class should look as follows if you have created your new solution with Xamarin Studio:

```
int count = 1;

protected override void OnCreate (Bundle bundle) {
  base.OnCreate (bundle);

  // Set our view from the "main" layout resource
  SetContentView (Resource.Layout.Main);

  // Get our button from the layout resource,
  // and attach an event to it
  Button button = FindViewById<Button>
  (Resource.Id.myButton);

  button.Click += delegate {
    button.Text = string.Format ("{0} clicks!", count++);
  };
}
```

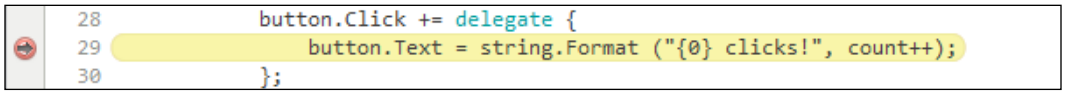
2. Place a breakpoint on the following line:

```
button.Text = string.Format ("{0} clicks!", count++);
```

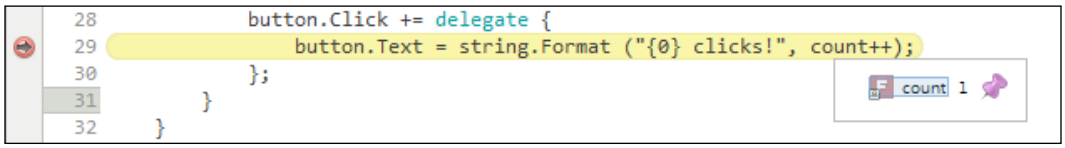
3. To set a new breakpoint in your program—a point where your program will pause and wait—double-click on the left-hand side margin of your code file:
4. The following will appear on your screen:



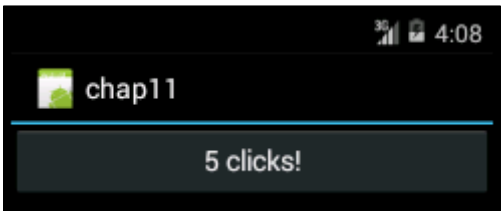
5. Next, run your program in the debug mode.
6. When the application starts on your emulator, click on the only button that the application displays. A line of code on Xamarin Studio will be highlighted as shown in the following screenshot:



7. Hover over the `count++` expression:



8. Click on the **1** inside the popup and replace it with **5**. Then, press *Enter*.
9. Click on **Step Over** on the top bar of Xamarin.
10. The application on the emulator should look as shown in the following screenshot:



How it works...

Breakpoints are placed on executable lines of a program. When the program execution reaches a line marked as a breakpoint, the debugger kicks in and suspends the execution of the program before that line of code.

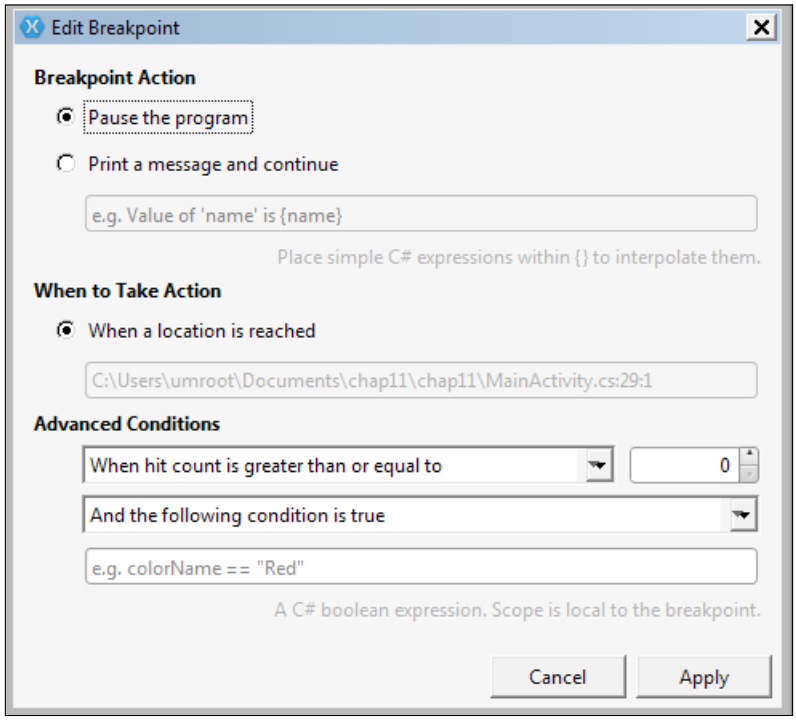
When the program is suspended, all the variables are accessible for introspection and modification. In this recipe, we introspected the value of the count and saw that it was **1** before we modify it to **5**. The changes are instantaneous and directly apply to the running application.

There's more...

The breakpoints can also be parameterized in terms of actions or advanced conditions.

Parameterizing breakpoints

By right-clicking on a breakpoint, we can access its properties from the popup context menu, which appears (the properties is the last item in the menu) and parameterizes it in terms of actions and advanced conditions. For example, we can replace the default action of a breakpoint that pauses the program using a simple log (by selecting **Print a message and Continue**) in the console or pausing the program only at the *n*th time that the execution passes over the breakpoint:



Debugging on a phone

The emulators, which are provided by the Android development team and integrated in Xamarin, are great pieces of technology but do have some limitations. For example, in the previous chapter, we saw that they don't support most of the hardware interactions, such as NFC, GPS, or Accelerometer. Consequently, some of your applications will have to be developed and tested on real devices.

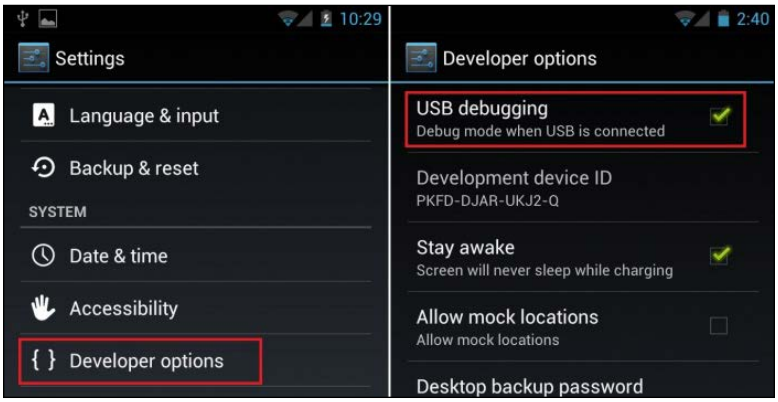
Getting ready

To follow this recipe you will, obviously, need an Android phone and a USB cable to connect it to your PC/MAC.

How to do it...

To get started with debugging on a phone, you'll need to perform the following steps:

1. Enable the **Debugging on Device** feature of your Phone. On Android 4.0 and 4.1, it's under **Setting** | **Developer Options** | **USB Debugging**. Starting with Android 4.2, the developer option is hidden and you have to activate it by pressing seven time on the build number by navigating to **Settings** | **About phone**.



2. On Windows, you need to install the USB Driver by executing the `android.bat` file located in the `tools` folder of your Android SDK installation.
3. Download and install the USB driver made by your phone manufacturer as explained at: `http://developer.android.com/tools/extras/oem-usb.html#Drivers`.
4. Your phone should now be recognized by Xamarin and you can run, test, and debug your application directly on it.

How it works...

The processes involved in debugging on an emulator or a physical phone are exactly the same.

If you stop Xamarin and disconnect your USB cable, the application will stay on your phone and you will be able to use it afterward.

See also

Go to `http://www.learn2crack.com/2014/07/adb-android-debug-bridge-over-wifi.html` to learn how to debug over Wi-Fi instead of USB.

Unit testing

According to `searchsoftwarequality.techtarget.com`, unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing is often automated but it can also be done manually.

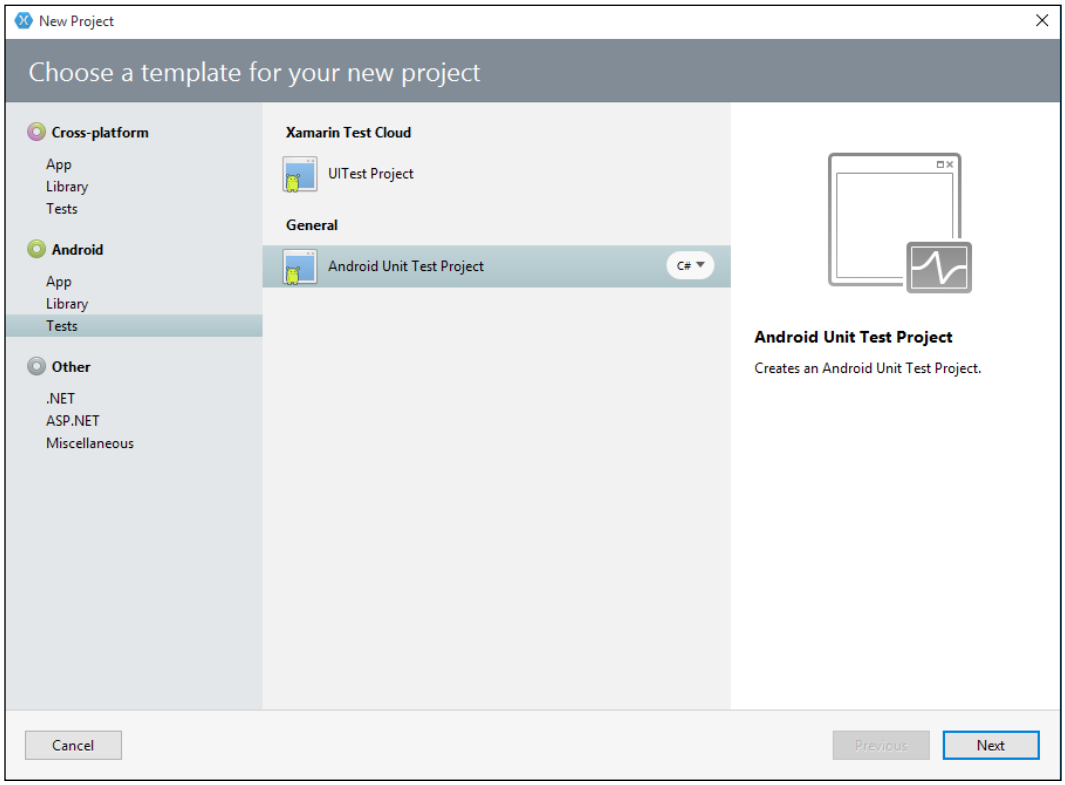
Getting ready

In this recipe, we will reuse the solution we created in the first recipe.

How to do it...

The following steps should be performed in order to do unit testing:

1. Add an Android unit test project to your solution.



2. The `MainActivity` class should contain the following:

```
public class MainActivity : TestSuiteActivity {
  protected override void OnCreate (Bundle bundle) {
    // tests can be inside the main assembly
    AddTest (Assembly.GetExecutingAssembly ());
    // or in any reference assemblies
    // AddTest (typeof (Your.Library.TestClass).Assembly);

    // Once you called base.OnCreate(), you cannot add more
    assemblies.
    base.OnCreate (bundle);
  }
}
```

3. Another file named `TestsSample` is also part of this `AndroidTestProject` project. This file contains the following code:

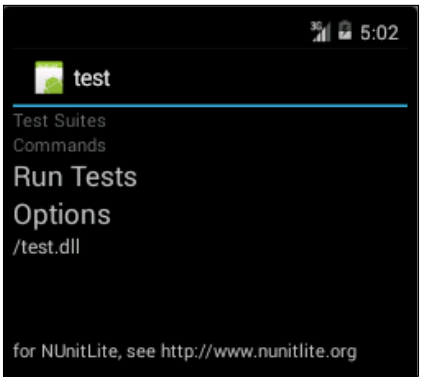
```
[TestFixture]
public class TestsSample {

  [SetUp]
  public void Setup () {
  }


  [TearDown]
  public void Tear () {
  }

  [Test]
  public void Pass () {
    Console.WriteLine ("test1");
    Assert.True (true);
  }

  [Test]
  public void Fail () {
    Assert.False (true);
  }

  [Test]
  [Ignore ("another time")]
  public void Ignore () {
```

```
    Assert.True (false);
  }

  [Test]
  public void Inconclusive () {
    Assert.Inconclusive ("Inconclusive");
  }
}
```

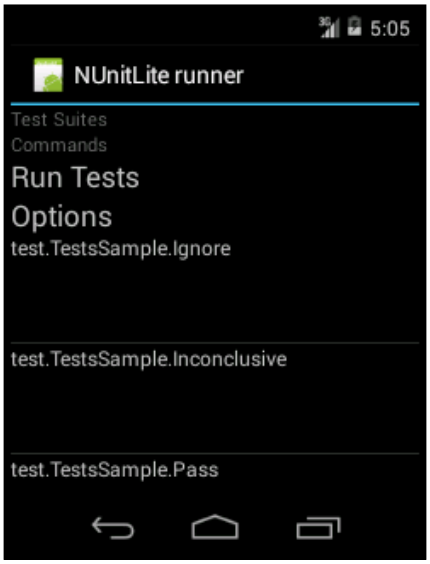
4. Modify your `MainActivity` class so it looks like the following:

```
[Activity (Label = "test", MainLauncher = true)]
public class MainActivity : TestSuiteActivity {
  protected override void OnCreate (Bundle bundle) {

    AddTest (typeof (TestsSample).Assembly);

    base.OnCreate (bundle);
  }
}
```

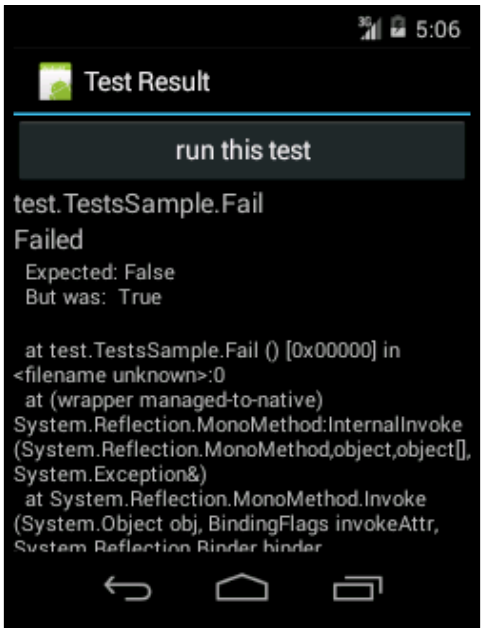
5. Set your test project by right-clicking on it, then select the **Set as Startup project** option. Finally, run your project in your emulator and the following screen will appear:



6. Click on `/tests.dll` and then on `test.TestsSample`. You will now have a list of all the available tests:



7. Click on the first one and then, click on **run this test** and watch it fail.



How it works...

The unit tests of Xamarin are based on NUnit. Let's take a look at the following explanation:

- `[Test]`: These are annotations for marking the method to use for a test.
- `[SetUp]`: This is for annotations for methods that should be executed before the tests begin.
- `[TearDown]`: This is for methods that should be executed when all the tests are completed. We use them to clean/close all the resources we have used, if any, during the tests.

Each test contains an assertion which will be evaluated, and if the assertion is true, the test is validated. There are plenty of static methods that you can use on the `Assert` class, such as:

- `Assert.AreEqual(obj1, obj2)`: This asserts that `obj1` and `obj2` are equal
- `Assert.AreNotEqual(obj1, obj2)`: This asserts that `obj1` and `obj2` are not equal
- `Assert.AreSame(obj1, obj2)`: This asserts that `obj1` and `obj2` are the same
- `Assert.DoNotThrow(TestDelegate)`: This asserts that the test does not throw the specified delegate
- `Assert.Catch(TestDelegate)`: This asserts that the specified delegate is caught
- `Assert.IsNotNull(Object)`: This asserts that the object is not `null`
- `Assert.IsNull(Object)`: This asserts that object is `null`
- `Assert.IsTrue(Object)`: This asserts that the Boolean is `true`
- `Assert.IsFalse(Object)`: This asserts that the Boolean is `false`

See also

Learning how to use NUnit in depth is out of the scope of this book, but if you are interested in learning more about it and to use it efficiently, I would recommend that you refer to *Pragmatic Unit Testing in C# with NUnit*, *Second Edition*, by *Andy Hunt* or the official documentation of nunit at `http://www.nunit.org/index.php?p=documentation`.

13
Monetizing and Publishing Your Applications

In this chapter, we will cover the following recipes:

- Creating an Ad unit
- Installing the required SDKs
- Integrating advertisements in our applications
- Preparing your application for publication
- Publishing your application

Introduction

From the beginning of this book and until now, we have focused on how to build an Android application using Xamarin Studio. In the last two chapters, you will learn how to benefit from an advertisement in our applications. In this chapter, we will see how to monetize our application by integrating advertising in it.

Creating an Ad unit

AdMob (**advertising on mobile**) is a mobile advertising company created in 2006 by Omar Hamoui and bought by Google in 2009. In this book, we choose to explain how to integrate an advertisement with AdMob, as they are the easiest one to go with. Indeed, since Google bought them, the integration of advertisements provided by AdMob for Android applications is effortless. Moreover, AdMob also allows us to integrate advertisements in iOS, webOS, Flash Lite, Windows Phone, and in all standard mobile browsers. Consequently, you can advertise on all your mobile channels with AdMob. The best of all the features are the analytic features that are merged into Google analytics and offer a tremendous amount of detail.

In this recipe, you'll learn how to create an AdMob Ad unit.

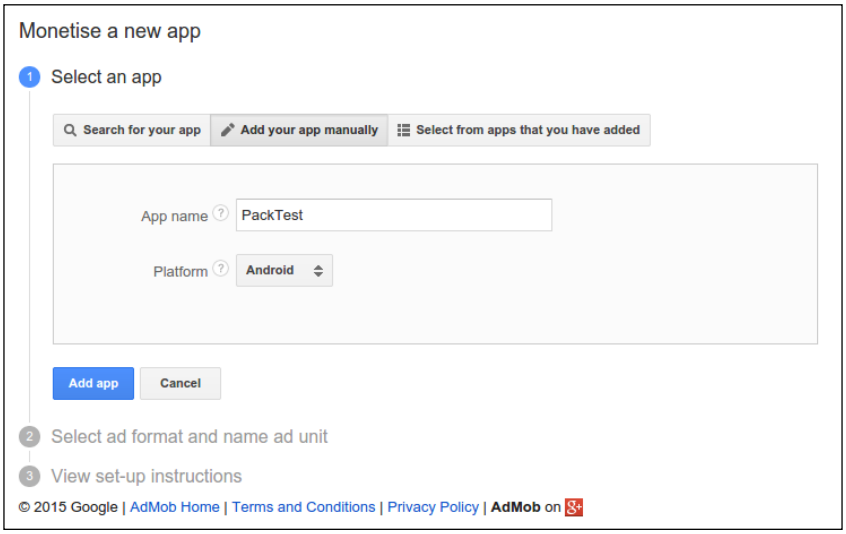
Getting ready

For this recipe, you'll need to have an AdMob account. Simply visit `https://www.google.com/admob/` and create an account for yourself. If you already own an **Adsense** account linked to a Google account, it is recommend that you sign in to AdMob using this one.

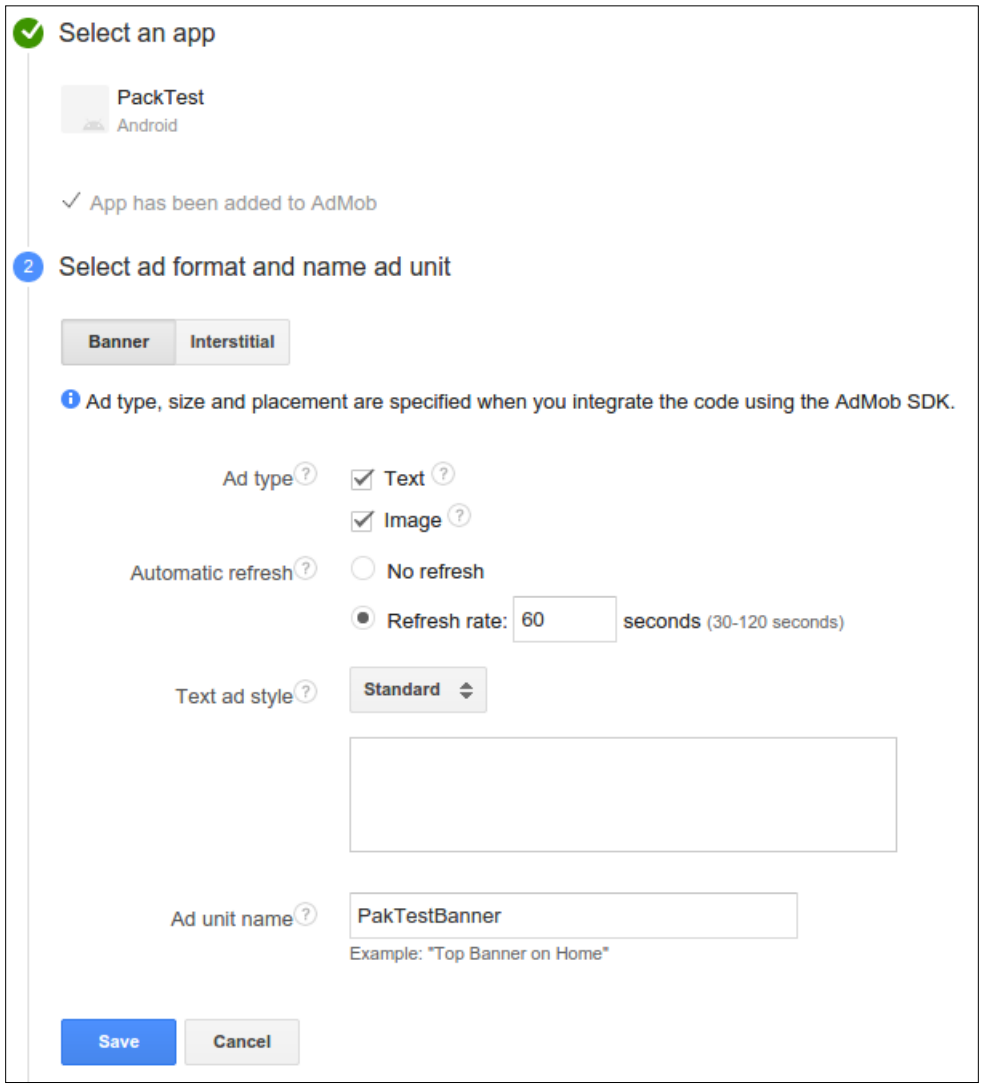
How to do it...

In order to complete this recipe, you need to perform the following steps:

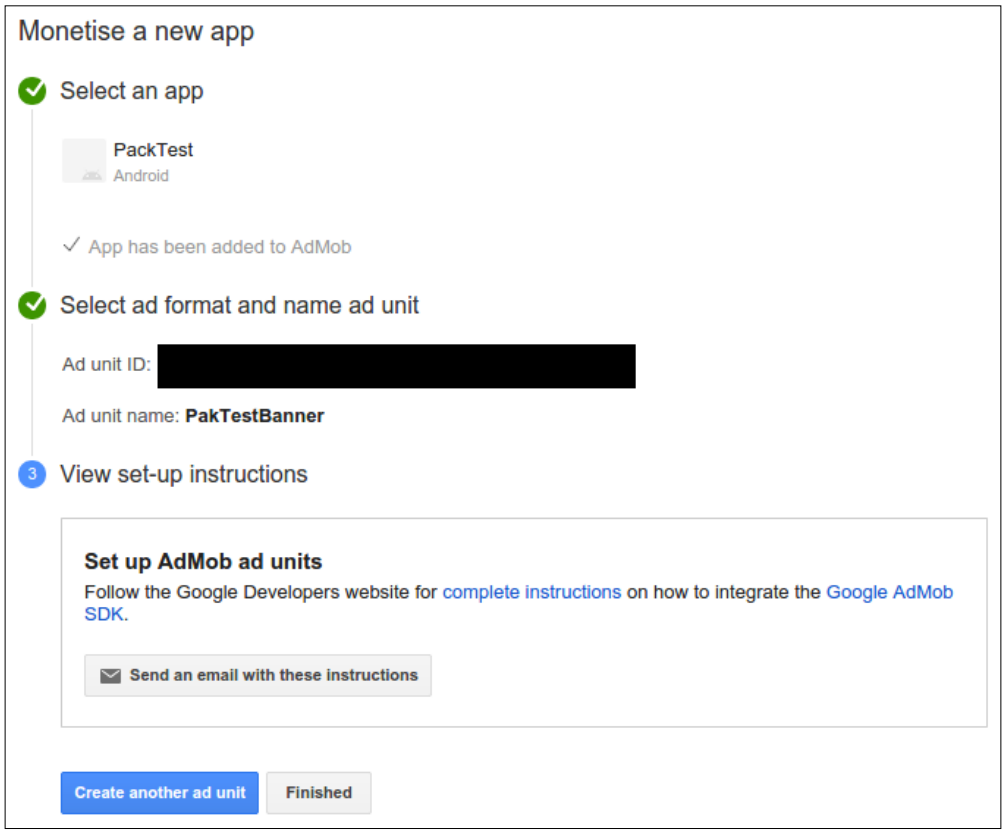
1. While logged in on AdMob, go to the **Monetize** tab and select **Monetize new App** option.
2. Add your app manually and pick a test name.



3. Select an Ad format and name the Ad unit as shown in the following screenshot:



4. Click on **Finished** and save your Ad unit ID. You'll need it for the next recipes.



How it works...

As seen in the introduction of this recipe, AdMob allows us to monetize our application by displaying advertisements on them. When you integrate one Ad unit from AdMob to your application, you don't have to care about finding the company who will advertise on your applications, AdMob takes care of the hassle for you. With AdMob, we can have two types of ads. The first one is **Banners**: these are small rectangle ads to be placed on the top or at the bottom of your application and will be displayed constantly. They can be text- or image-based and can be automatically refreshed every 30 to 120 seconds. The recommended rate is between 45 to 60 seconds in order to not be too annoying for the users and still display a fair amount of ads.

The second type of ads you can display with AdMob are called **Interstitial**. Interstitials are, according to Google, immediately present rich HTML5 experiences or "web apps" at natural app transition, such as in between game levels. Web apps are in-app browsing experiences with a simple close button rather than any navigation bar—the content provides its own internal navigation scheme. You can even offer in-app purchase promotion ads to your users. Interstitial ads are typically more expensive and subject to impression constraints. They don't have a refresh rate but a timeout that can range from 3 to 10 seconds.

The ad unit ID will simply tell AdMob what to display in the dedicated content of your application.

See also

Refer to *Installing the required SDKs* recipe to install the required SDKs and work with AdMob.

Installing the required SDKs

As you might have guessed, integrating advertisements in our applications is not part of the standard Android SDK, which came when you installed Xamarin, and we have to install some additional pieces. In this recipe, we'll see which ones.

Getting ready

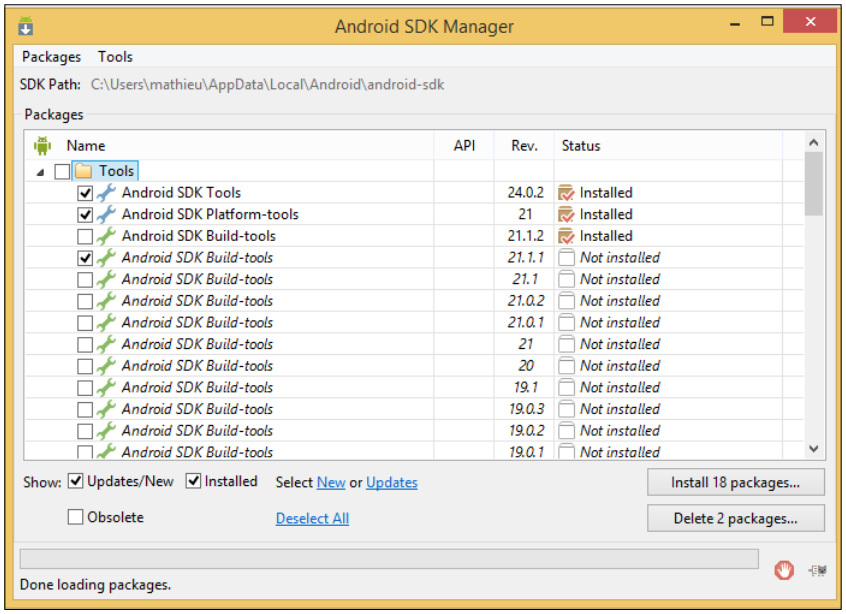
To follow this recipe, you will need to have access to the Android SDK manager and the permissions to make new installations in your system.

How to do it...

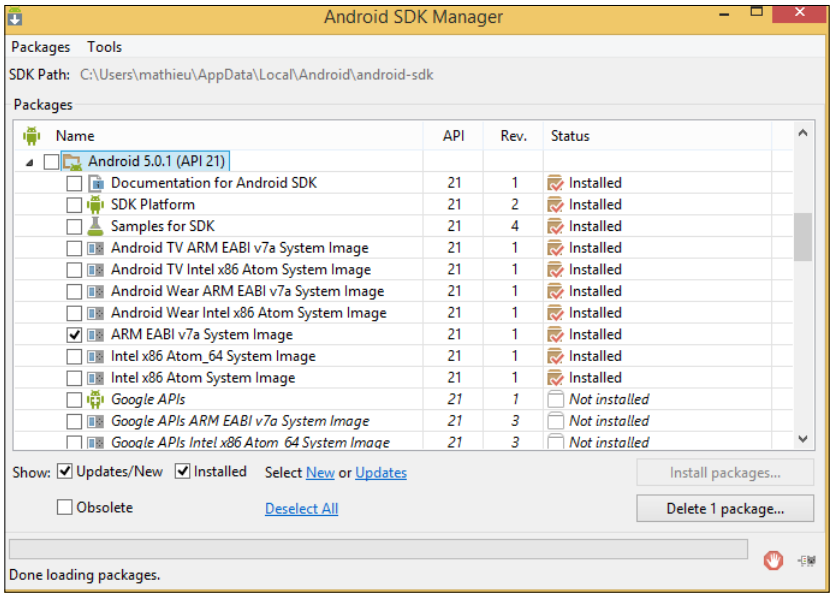
In order to complete this recipe, you will have to follow these steps:

1. Open the Android SDK Manager.

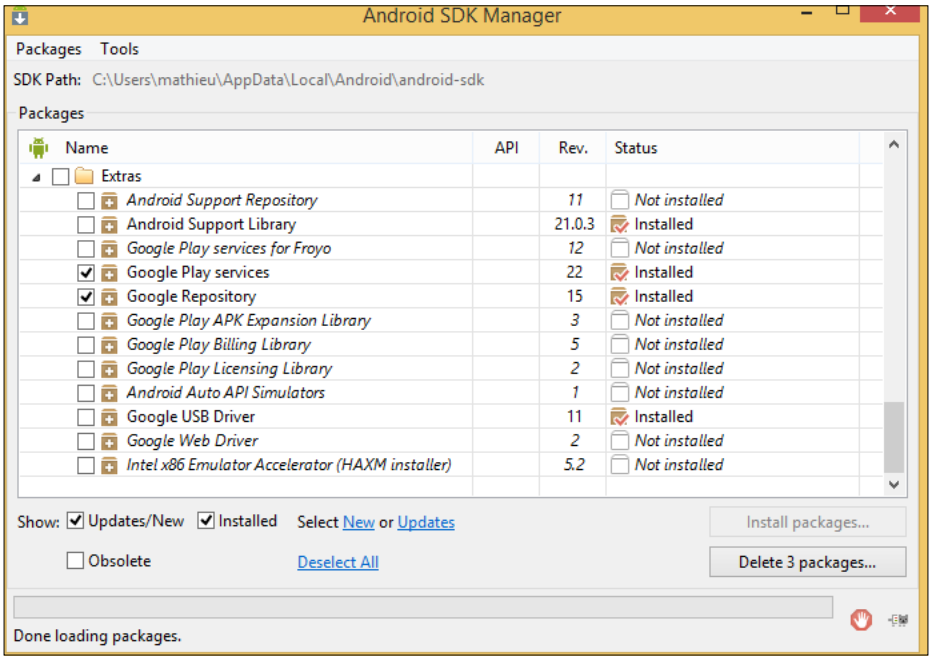
2. Open the tool directory and install the **Android SDK tools**, **Android SDK Platform-tools**, and **Android SDK Build-tools**.



3. Open the latest version of Android and ensure that you select **ARM EABI v7a System Image** for the installation.



4. Open the **Extra** directory and select the **Google repository** and **Google Play services** for installation.



5. Click on **Install packages...** and you're done.

How it works...

AdMob used to have its own SDK that we, as developers, had to integrate inside our Android SDK and ship our application with some of the AdMob libraries for the ads to be displayed. Thanks to Google buying AdMob, all required materials are now part of the Android SDK, while not yet by default, and their integration is even easier. With the new packages we've installed, we are now ready to integrate our first ads. See how this is done on the next recipe.

See also

Refer to *Installing ads in your applications* recipe to integrate AdMob in your applications.

Integrating advertisements in your applications

In this recipe we will leverage our AdMob account and the newly installed package to display some ads in our applications.

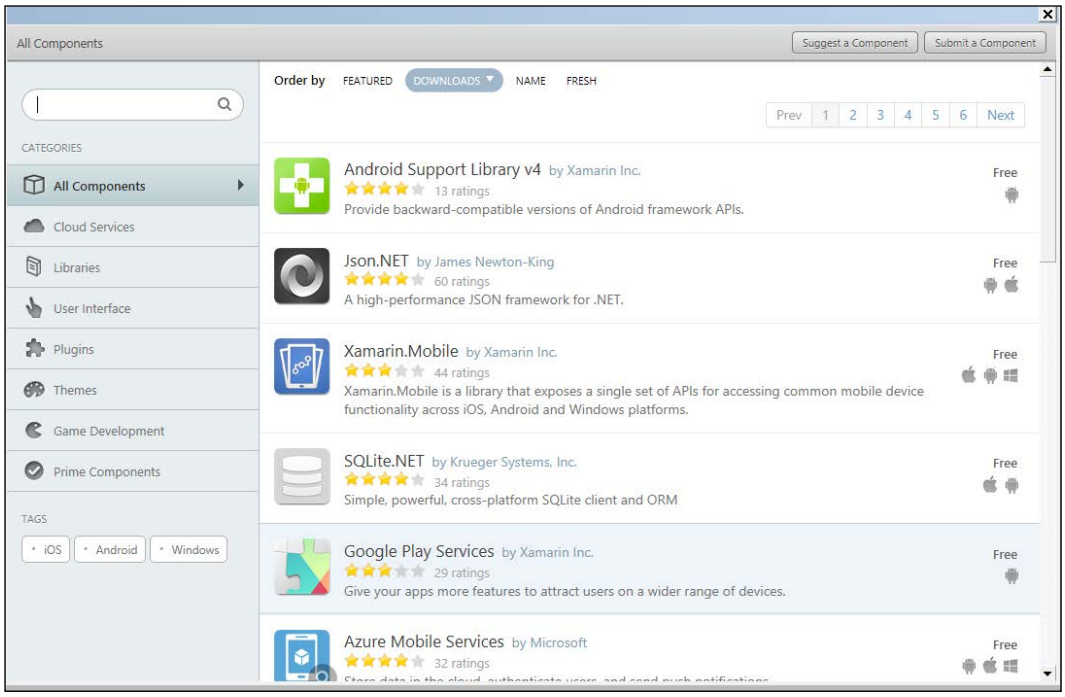
Getting ready

To follow this recipe, create a new solution named `Ads`.

How to do it...

In order to complete this recipe, you will have to follow these steps:

1. Right-click on the `components` folder of your solution and click on **Get more components**.
2. Then, in the new window, search for `Google Play Services` and add it to your solution.



3. Ensure that the components have been added to your solution. It should look as follows:

4. Add the following two lines to your `AndroidManifest.xml` file inside the `<application>` tag:

```
<meta-data android:name="com.google.android.gms.version"
android:value="@integer/google_play_services_version" />
<activity android:name=
"com.google.android.gms.ads.AdActivity"
android:configChanges="keyboard|keyboardHidden|orientation|
screenLayout|uiMode|screenSize|smallestScreenSize" />
```

5. Add the **Internet and Access Network Status** permissions to your application.
6. Add using `Android.Gms.Ads` at the top of your `MainActivity` class.
7. Add the following code in the `OnCreate()` method of your `MainActivity` class:

```
var adMob = new AdView(this);
  adMob.AdSize = AdSize.SmartBanner;
  adMob.AdUnitId = 'Replace this w/ your adUnitId';
  var requestbuilder = new AdRequest.Builder();
  adMob.LoadAd(requestbuilder.Build());
  var layout =FindViewById<LinearLayout>
  (Resource.Id.mainlayout);
  layout.AddView(adMob);
```

8. Run your application.

How it works...

First of all, we created a reference to an `AdView()` method using the context of the `OnCreate()` method:

```
var adMob = new AdView(this);
```

Then, we specified that the ad is of the banner type:

```
adMob.AdSize = AdSize.SmartBanner;
```

We also had to specify our `AdUnitId` using this line of code :

```
adMob.AdUnitId = 'Replace this w/ your adUnitId';
```

Finally, we construct, load, and display the banner as follows:

```
var requestbuilder = new AdRequest.Builder();
adMob.LoadAd(requestbuilder.Build());
var layout = FindViewById<LinearLayout>(Resource.Id.mainlayout);
layout.AddView(adMob);
```

If you run this on the emulator, it might or might not work. Indeed, I found that it depends on your Xamarin version and the Android emulator you are using.

There's more

In this recipe, we implement banner ads only. However, AdMob offers the possibility to use the `InterstitialAd()` method that we described in the first recipe of this chapter.

In order to integrate one of those on your application, you need to have the following code:

```
var myInterstitialAd = new InterstitialAd(this);
myInterstitialAd.AdUnitId = "Replace this by your id";
var requestbuilder = new AdRequest.Builder();
myInterstitialAd.LoadAd(requestbuilder.Build());
myInterstitialAd.Show()
```

This works exactly the same as banner except that we use an `InterstitialAd` object and not an `AdView` object. As the `InterstitialAd` object are full screen only, we don't need to specify the size of the ad.

See also

Refer to the following documentation to learn much more about the monetization of your mobile applications and the business model you should choose:

- *Business Models for the Social Mobile Cloud: Transform Your Business Using Social Media, Mobile Internet, and Cloud Computing* by Ted Shelton
- *Mobile Service Innovation and Business Models* by Bouwman Harry

Preparing your application for publishing

In this recipe, we will pass through all the required steps to prepare an Android application created with Xamarin before publishing it to the Google Play Store.

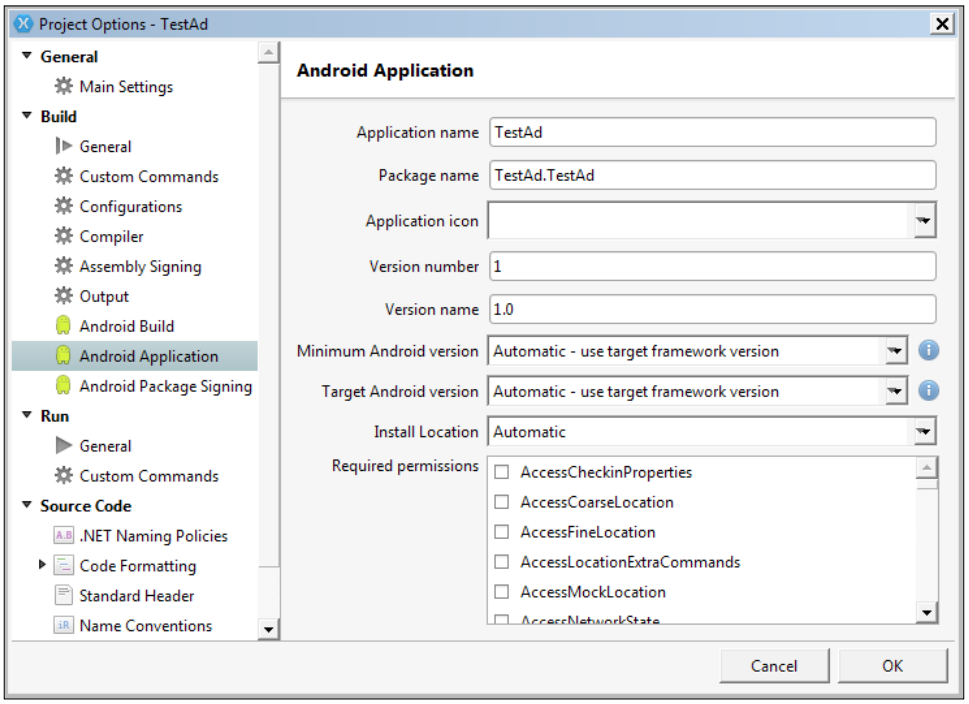
Getting ready

To follow this recipe, reuse the solution we created in the previous recipe.

How to do it...

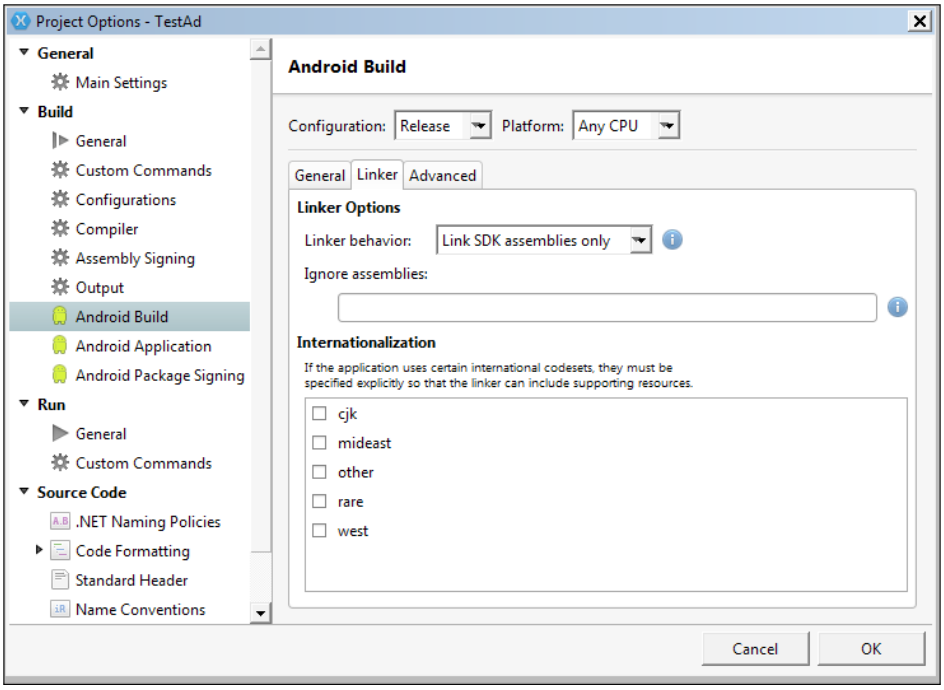
In order to complete this recipe, you will have to follow these steps:

1. Specify an icon for your application in the **Project option** windows, navigate to **Build** | **Android Application**.



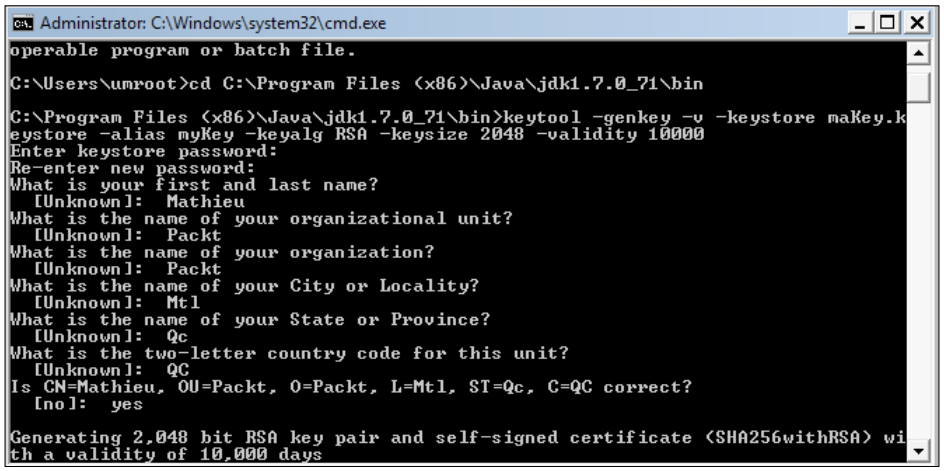
2. In the same window, specify the version number and the version code.
3. Still, on the same dialog, go to **Android Build** and change the configuration to **Release**.

4. Then, change the **Linker behavior** option from **SDK and User Assemblies** to **Link SDK assemblies only**.



5. Create a private keystore by entering the following line of code in a console:

```
$ keytool -genkey -v -keystore maKey.keystore -alias myKey -
keyalg RSA -keysize 2048 -validity 10000
```



Keytool is an utility program shipped with the Java JDK. On a classical installation, it will be located under `C:\Program Files (x86)\Java\jdk1.7.0_71\bin`.

6. Open the **Publish Android Application** dialog of the **Project** menu.
7. Select **use an existing keystore** and then the keystore you created in the fifth step.
8. Then, click on **Forward** and choose the location of your choice for the signed APK.
9. Click on **Create** and the final APK will be created and signed.

How it works...

We have already dealt with the application icon earlier in this book, in *Chapter 4*, *Using Android Resources*, but as we are a few recipes away from publishing to the Google Play Store, it is better to check whether everything is set for production. The application icon is what users who downloaded your application will see on their home screen and you can find some tips on how to create a good icon back in *Chapter 4*, *Using Android Resources*.

After the icon, we took care of the version of our application. Versioning is very important for software at large but it especially is for mobile application. Indeed, the version number will be used to automatically update your applications on your customers' phones and tablets. The first field we filled is the version number and is uniquely composed of numbers. Most applications start at version 1.0 but you can start at 0.1 or whichever nomenclature suits you. The second field is the version name and is the string that has no relation whatsoever with the version number. I personally don't give a name to my applications' versions but you can be creative and, like Apple or Android, have a nice name for your versions, such as **Mountain Lion** or **KitKat**.

See also

Refer to the next and last recipe of this book, *Publishing your application*, to publish your application to the Android Play Store.

Publishing your application

In this recipe, we will finally leverage all the hard work we've done throughout this book and publish our very first application to the Google Play Store.

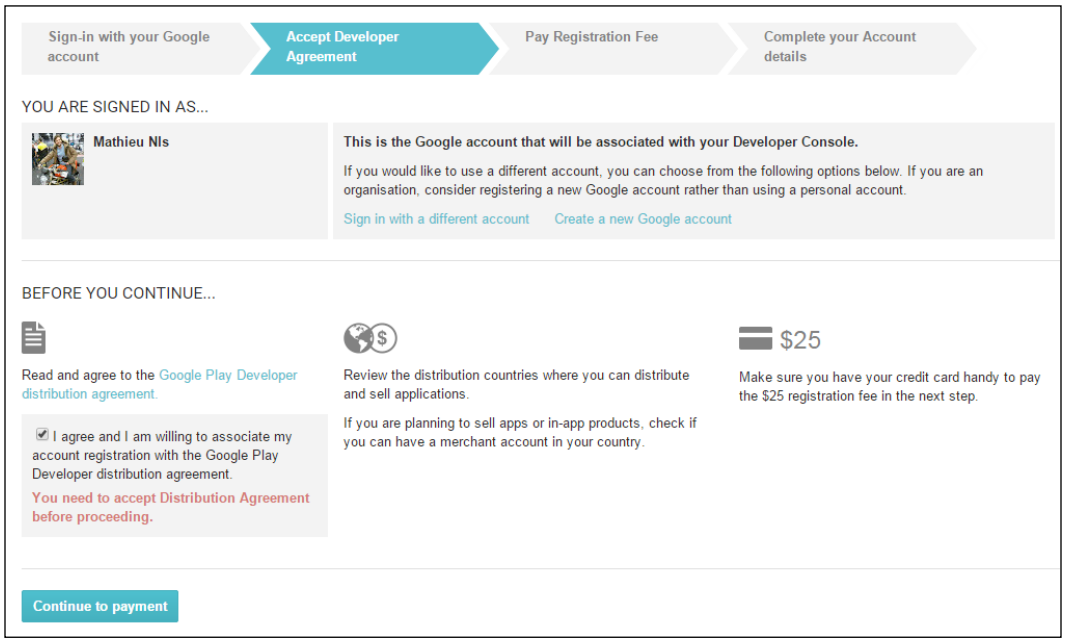
Getting ready

You must have a full-fledged application ready to be made available to your customers if you want to follow this recipe to the end.

How to do it...

Perform the following steps for this recipe:

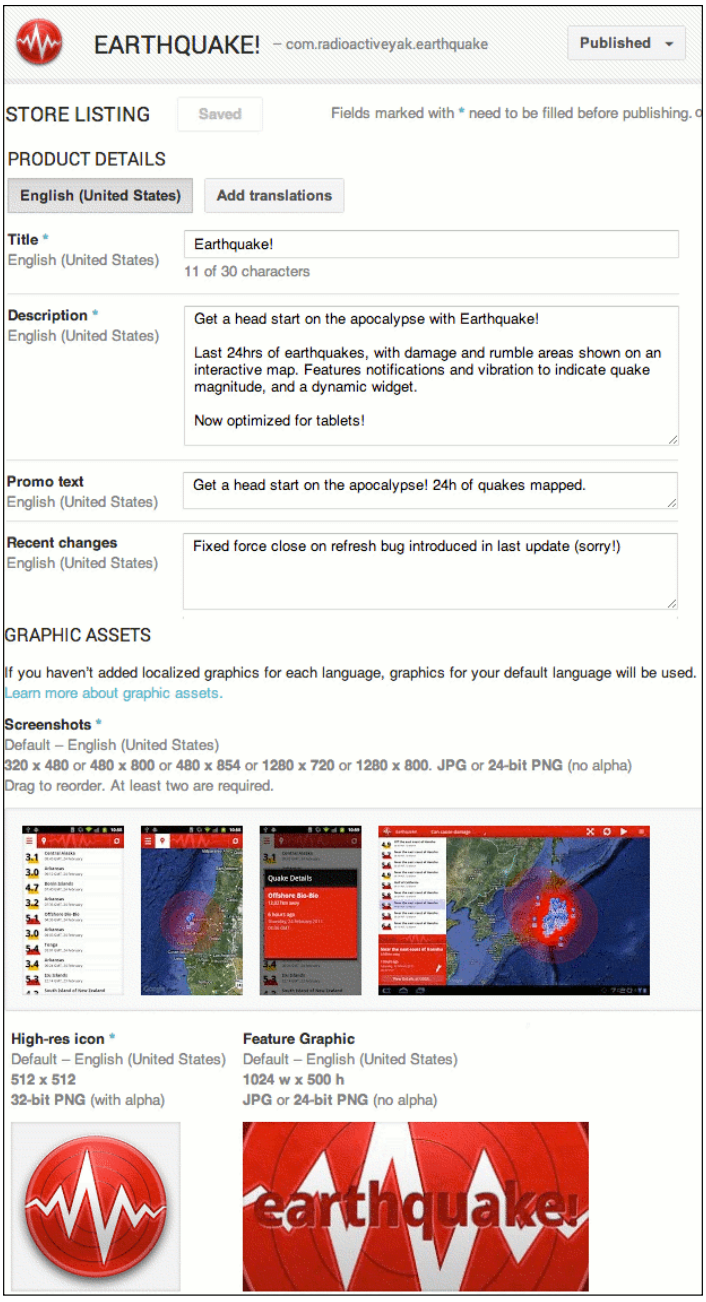
1. Go to `https://play.google.com/apps/publish`.
2. Login with your Google account or create a new Google account.



3. Pay the fees, that is 25$ USD.
4. Complete your account details with the required information.
5. Don't forget to confirm your e-mail address.
6. Create a 512x512 px icon that will be used for your application on Play Store.

7. Take a screenshot of your application and resize the screenshot to 480x800 px.

   Let's take a look at the following screenshot:



Source: http://developer.android.com/images/gp-dc-details.png

8. Optionally, make a video, 30 seconds to 2 minutes long, of yourself using the application.
9. Upload your APK through the developer console.
10. Upload the assets you have created in steps 6 and 7.
11. Fill out the listing details.
12. In the publishing option, set the content rating to **Everyone**, **Low maturity**, **Medium maturity**, or **High maturity**.
13. Select the countries in which you want to distribute your application.
14. Select a price for your application, if any.
15. Enter the contact information for your application.
16. Click on **Finalize** and wait for the first download and feedback!

How it works...

The Android Play Store used to be free of charge for publishers. However, Google now charges a $25 fee that you have to pay in order to be able to publish applications. Nevertheless, this is a one-time fee and you won't be asked to pay for publishing other applications.

The launcher icon is the first thing, and might be the only thing, Play Store visitors will see of your application. So, better make it very appealing and concordant with what your application is doing. Here are some tips by Google for the Launcher icons.

Simple and uncluttered: Launcher icons should be kept simple and uncluttered. This means excluding the name of the application from the icon. Simpler icons will be more memorable and will make it easier to distinguish at the smaller sizes.

- **Icons should not be thin**: Overly thin icons will not stand out well on all backgrounds
- **Use the alpha channel**: Icons should make use of the alpha channel, and should not been full-framed images

The screenshots and the video you made about your application will be displayed in the details page that kicks in when a user wants to know more about your application. Google only accepts portrait screenshots and any attempts on displaying landscape ones will be cropped.

In the publishing option, the content rating must comply with the Google rating system, which is as follows:

- **Everyone**: This option may not access, publish, or share location data. This option may not host any user-generated content, and may not enable communication between users.
- **Low maturity**: Applications that access, but do not share, location data. Depictions of mild or cartoon violence.

- **Medium maturity**: References to drugs, alcohol or tobacco. Gambling themes or simulated gambling, inflammatory content, profanity or crude humor. Suggestive or sexual references. Intense fantasy violence, realistic violence Allowing users to find each other. Allowing users to communicate with each other. Sharing of a user's location data.
- **High maturity**: A focus on the consumption or sale of alcohol, tobacco, or drugs. A focus on suggestive or sexual references, and graphic violence. The items in medium maturity list are subjective. As such, it is possible that a guideline that may seem to dictate a Medium maturity rating may be intense enough to warrant a high maturity rating.

Then, for the price, you must know that if you choose to make your application available free, then it will have to be available for free forever. Paid applications, however, can become free at any time, and they can change prices.

There's more...

In this recipe, we saw how to publish our app for the first time. However, you'll certainly need to publish updates of your applications in order to fix bugs or provide new functionalities to your customers. The truth is that publishing updates is quite simple:

1. Go back to your developer console: `https://play.google.com/apps/publish/`
2. Select your application and upload a new APK.
3. While your app is being processed, that is, verified by Google, you'll see an **Update Pending** dialog in the top-right corner.
4. When your app has been processed, the **Update Pending** dialog disappears and your updates are distributed to your existing users within the next 24 hours.

If your customers have activated the automatic updates, then the update will be automatically installed. If not, they can install it from Play Store.

See also

Refer to the complete developer content policy at:

- `https://play.google.com/about/developer-content-policy.html`
- `https://developer.android.com/distribute/googleplay/guide.html`

Conclusion

In *Xamarin Studio for Android Programming – A C# Cookbook*, we saw almost one hundred step-by-step recipes to master Android programming. In *Chapter 1*, *Getting Started*—you learned about Xamarin studio and the Android emulator in order to run our first Hello World application. Then, we mastered the life and death of Android Applications through their activities' life cycles in *Chapter 2*, *Mastering the Life and Death of Android Apps*. You also learned how to manage the different states of an Android activity.

In *Chapter 3*, *Building a GUI*, we constructed our very first Android GUI and saw no less than 19 different components to be prepared to every situation. Then, we made things look better by using Android resources. In *Chapter 4*, *Using Android Resources*, we discussed how to use a splash screen, manage multiscreen, play songs or videos, and display our application icon.

At this stage, we were ready to take a leap forward using on-phone data in *Chapter 5*, *Using On-Phone Data*, in order to persist and access to user data between different launch of our applications. To do so, we talked about user preferences, file writing/reading, SQLite, and LinQ. To complete the loop between the on-phone data and the GUI, we covered how to populate GUI components with data in *Chapter 6*, *Populating Your GUI with Data*.

On our journey, we saw how to create, bound to, and identify running Android services in order to pursue our application processes even when the users are not looking at them. You also learned, in *Chapter 8*, *Mastering Intent – A Walkthrough*, how to switch back and forth between applications with Android Intents for accessing a contact number or an address on map, for example.

In *Chapter 9*, *Playing with Advanced Graphics*, we brought the last piece to our GUI building skills by mastering Android fragments that represent a behavior or a portion of the user interface in an Activity and learnt how to use NFC, Bluetooth, Accelerometer, and GPS in *Chapter 10*, *Taking Advantage of the Android Platform*.

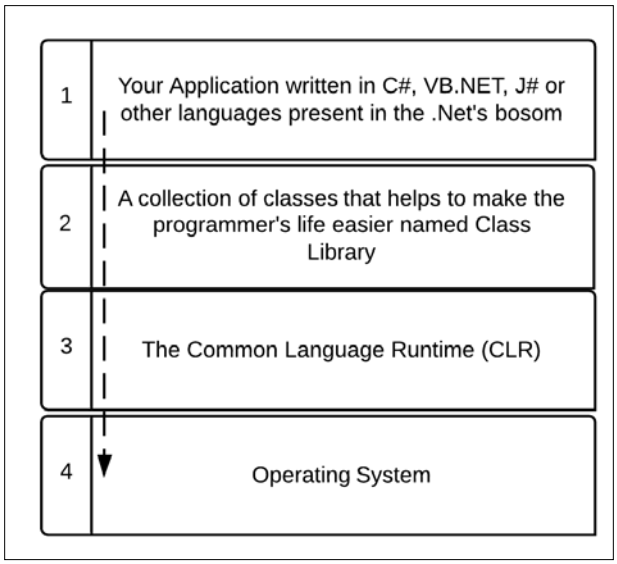
Finally, in *Chapter 11*, *Using Hardware Interactions*, and *Chapter 12*, *Debugging and Testing*, we covered how to leverage the debugging and testing capacities of Xamarin Studio and the Android emulator before learning how to prepare and publish our application to the Android Play Store.

Mono – The Underlying Technology

Mono is a software project that successfully brings the Java motto, "Write once, run everywhere" to the .Net community. Indeed, Mono is an open source implementation of the Microsoft .Net Framework and the **Common Language Runtime** (**CLR**).

The Microsoft .Net Framework was first announced in 2000 and released in 2002. This release includes a large collection of libraries and enables a noteworthy language compatibility. Programs using the .Net Framework do not execute themselves directly on the hardware but in a software environment known as the CLR. The CLR is a virtual machine that handles the execution of programs on a large range of hardware while providing security, memory management, and other services. The important point here is that the CLR is itself based on the Common Language Infrastructure, which is also developed by Microsoft but under an open standard ECMA-335; thus opening the doors to open source projects aiming an open source implementation of the .Net Framework like Mono.

Even if the Mono ways of working are far out the scope of this book, the following screenshot presents the big picture composed of four steps. As expected, Mono handles the execution of applications written in any of the .Net supported languages, such as C#, VB.Net, and so on. Applications built using these .Net languages lie on a collection of classes that help make the programmers' lives easier. The programs are translated into Common Intermediate Language, and finally, they are executed inside the CLR, which manages interactions with the operating system. Let's take a look at the following diagram:



Index

M

N

O

P

R

S

T

U

V

W

X

Thank you for buying

Xamarin Studio for Android Programming: A C# Cookbook

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

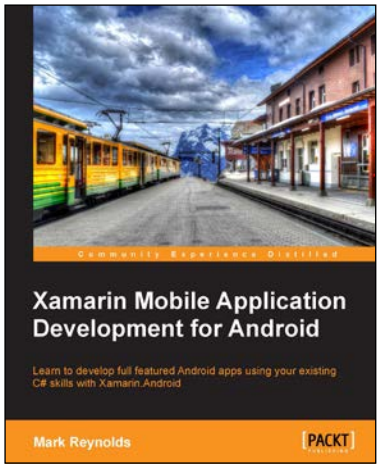
[PACKT] PUBLISHING

Xamarin Cross-platform Application Development

ISBN: 978-1-84969-846-7 Paperback: 262 pages

Develop production-ready applications for iOS and Android using Xamarin

1. Write native iOS and Android applications with Xamarin.
2. Add native functionality to your apps such as push notifications, camera, and GPS location.
3. Learn various strategies for cross-platform development.



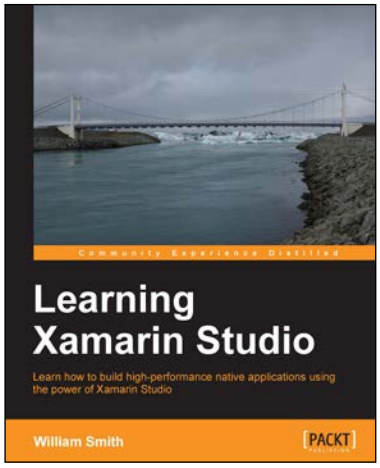
Xamarin Mobile Application Development for Android

ISBN: 978-1-78355-916-9 Paperback: 168 pages

Learn to develop full featured Android apps using your existing C# skills with Xamarin.Android

1. Gain an understanding of both the Android and Xamarin platforms.
2. Build a working multi-view Android app incrementally throughout the book.
3. Work with device capabilities such as location sensors and the camera.

[PACKT]
PUBLISHING

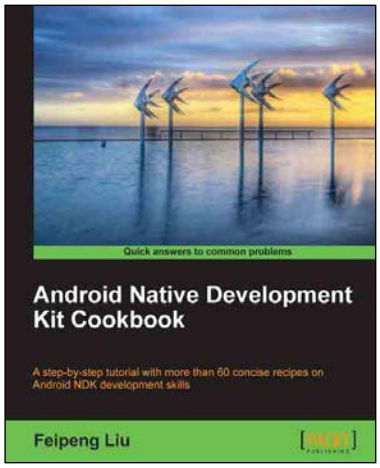


Learning Xamarin Studio

ISBN: 978-1-78355-081-4 Paperback: 248 pages

Learn how to build high-performance native applications using the power of Xamarin Studio

1. Get a full introduction to the key features and components of the Xamarin 3 IDE and framework, including Xamarin.Forms and iOS visual designer.
2. Install, integrate and utilise Xamarin Studio with the tools required for building amazing cross-platform applications for iOS and Android.
3. Create, test, and deploy apps for your business and for the app store.



Android Native Development Kit Cookbook

ISBN: 978-1-84969-150-5 Paperback: 346 pages

A step-by-step tutorial with more than 60 concise recipes on Android NDK development skills

1. Build, debug, and profile Android NDK apps.
2. Implement part of Android apps in native C/C++ code.
3. Optimize code performance in assembly with Android NDK.