Community Experience Distilled

Learning Android Canvas

Develop and deploy graphics-rich Android applications using Android Canvas

Mir Nauman Tahir

[PACKT] PUBLISHING open source* community experience distilled

Learning Android Canvas

Develop and deploy graphics-rich Android applications using Android Canvas

Mir Nauman Tahir

BIRMINGHAM - MUMBAI

Learning Android Canvas

First published: November 2013

Production Reference: 1181113

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

ISBN 978-1-78328-539-6

www.packtpub.com

Cover Image by Aniket Sawant (aniket_sawant_photography@hotmail.com)

Credits

Author
Mir Nauman Tahir

Reviewers
Luca Mezzalira
Daniel Nadeau

Acquisition Editors
Usha Iyer
James Jones

Commissioning Editor
Mohammed Fahad

Technical Editors
Anita Nayak
Amit Shetty

Copy Editors
Alisha Aranha
Aditya Nair
Deepa Nambiar

Project Coordinator
Michelle Quadros

Proofreader
Stephen Copestake

Indexer
Priya Subramani

Graphics
Abhinash Sahu

Production Coordinator
Nitesh Thakur

Cover Work
Nitesh Thakur

About the Author

Mir Nauman Tahir completed his MS in Information Technology from Khyber Pakhtoonkhwa, Pakistan. He specialized in web technologies. He is also a Microsoft Certified Technology Specialist. He started his professional programming career in 2004 and has worked on developing different systems, such as Library Management System, Procurement System, a lot of web applications, and dynamic websites with content management systems. Mir worked on the Android source code for almost 1.5 years in a research group as a developer and published a research paper on the basis of his research with that team. Mir has also worked on other open source technologies such as OpenERP, PostGre-SQL, and different flavors of Linux. Mir currently has more than eight years of industry experience consisting majorly of in-house development for some of Pakistan's most well reputed organizations and universities.

He started his career as a Software Engineer for COMSATS Institute of Information Technology, Abbottabad, Pakistan and has since worked with Security Engineering Research Group, Peshawar, Pakistan; UN-HABITAT, Pakistan; Save the Children, Pakistan; and currently working as an Information Management Officer in a well-reputed humanitarian organization in Pakistan.

This is Mir's first book, but he writes most of his technical articles on his personal blog, `http://mirnauman.wordpress.com`. The articles are mostly about Android and Microsoft .Net technologies. Mir loves sketching and playing computer games.

Acknowledgments

I would like to thank my mother, Farkhanda Roohi, for providing all the motivation to take up the challenging task of writing this book and my father, Shahzaman Khan, for teaching me to adjust when I had a million things to do in a day other than writing this book. I especially thank my wife, Manahil Mir, for her support in completing the book and keeping to her promise that she wouldn't ask me to take her shopping when I come back from the office and had to work on the book. I would like to mention my elder brother, Dr. Zeeshan Mir, and younger brother, Mir Mohammad Suleman Sarwar, here; they asked me to include their names in this section because my first book is just as important to them as it is to me. I love my family a lot and I live for them. Last but not the least, I would like to thank Packt Publishing for giving me the chance to make this dream come true.

About the Reviewers

Luca Mezzalira is an Italian software developer oriented towards frontend technologies and has a great passion for the techie world. He has been an Adobe Certified Expert and Instructor since 2002. He became an Adobe Italy consultant and, after few years, an Adobe Community Professional as well. Luca worked on many projects for different environments, from web to desktop applications and from embedded systems to mobile applications.

His passion for and his focus on new technologies allowed him to work on a lot of projects for important Italian and international companies with great results. To complete his list of skills, Luca is also a teacher, a speaker in national and international events, and a writer for technical magazines and books. In his spare time, he likes to watch football, play with his dogs Paco and Maya, and study new programming languages.

My first giant thank you is for my family, who have always helped me, and in particular for my parents, who support and inspire me everyday with their strength and love.

Big thanks to my brother who is also one of my best friends. He is the most intelligent person that I've ever met in my life, and his suggestions and his ideas are the most important thing for me.

Then, I really have a lot of other friends to say thanks to, for what we have created together until now. I hope I haven't forgotten anybody: Thomas Baggio, Marzia Baggio, Piergiorgio Niero, Chiara Niero, Matteo Oriani, Manuele Mimo, Sofia Faggian, Matteo Lanzi, Zohar Babin, Peter Elst, Francesca Beordo, Federico Pitone, Tiziano Fruet, Gianluigi Fragomeni, Giorgio Pedergnani, Andrea Sgaravato, Marco Casario, Fabio Bernardi, Andrea Brauzzi, Sumi Lim, Jonathan Air, Mihai Corlan, Serge Jespers, Renaun Erickson, Christie Fidura, Piotr Walczyszyn, Michael Chaize, and many many others.

A special thanks also to the Inside a Bit and Sparkol teams, places where you can learn something every day.

Last but not least, I'd like to say thanks to my girlfriend and my life partner, Maela, for the amazing time we spend together. Her passion and commitment in our relationship give me the strength to go ahead and do my best every day. Thanks, my love!

Daniel Nadeau is currently an independent Android developer and C programmer with five years of programming experience under his belt. He is also currently a student in his second year of electrical engineering studies at Queen's University in Canada.

He has worked for a wide variety of companies, sometimes on a volunteer basis. Some of those companies include elementary OS, OHSO Media (with OMG! Ubuntu!), Naquatic LLC, and NowPhit. More recently, he has also begun applying his programming skills to microcontrollers by joining the Queen's Space Engineering Team.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

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.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across 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 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 nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Learning Android Canvas provides basic knowledge and understanding of graphics and programming for Android Canvas. The target audience is assumed to have zero prior knowledge of Canvas and working with graphics in Android application development. The audience are taken has basic graphic and canvas programming to mid-level Android Canvas programming knowledge. The book concentrates only on 2D graphics, and not 3D or animation, but provides a very strong base to move to animation and 3D graphics learning. It supplies practical step-by-step guidance from the basics of graphics through different graphics objects and techniques to more complex interactive graphics-rich Android applications.

What this book covers

Chapter 1, *Getting Started with Android Canvas*, provides some background knowledge of Android Canvas, 2D graphics, display screens and their basic understanding, and also about the important files in a graphics-rich application.

Chapter 2, *Drawing Threads*, helps in understanding the need, role, and use of threads, and the issues and solutions related to threads.

Chapter 3, *Drawing and Drawables in Android Canvas*, introduces the readers to some Drawables and drawing on a canvas, view, and surface view.

Chapter 4, *NinePatch Images*, explains the basic concept of slicing, `NinePatch` images, repeating and non-repeating regions, and creating backgrounds with them.

Chapter 5, *Touch Events and Drawing on Canvas*, explains capturing touch events and responding to them accordingly. Creation of a custom `View` class and its implementation, including touch events implementation, is also covered.

Chapter 6, *Putting it all Together*, deals with planning an application, creating it from scratch with a complex user interface, and putting all the previously learned knowledge in to practice.

What you need for this book

You need the following software to run the examples in this book:

- A computer with reasonable processing power and RAM, an Intel Core i3 will do the job
- Java Runtime Environment
- Eclipse Classic
- Android SDK, the latest would be the best

Who this book is for

Developers familiar with Java coding and some basic Android development knowledge. The book is for developers who have basic Android development knowledge but zero knowledge of Android Canvas development, and also developers interested in graphics-rich applications and developing games.

Conventions

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

Code words in text are shown as follows: " We will name our application `MyFirstCanvasApp`."

A block of code is set as follows:

```
class OurGameView extends SurfaceView implements
SurfaceHolder.Callback {
// Our class functionality comes here
}
```

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

```
android:background="@drawable/myfirstninepatch"
android:text="@string/buttonwith9patchimage"
```

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

```
C:\learningandroidcanvasmini\chapter1\firstapp
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "To do this, in Eclipse we will navigate to **File** | **New** | **Android Application Project**."

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 may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via 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 on `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 for all Packt books you have purchased from your account at `http://www.packtpub.com`. 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.

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 would 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 on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright 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

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Android Canvas

In this chapter we will learn a little about Android Canvas, why we need it, and what it provides. We will create a simple Android application displaying an image on the screen. We will test the application on an emulator. Then we will discuss each section of the application and things that the reader needs to know about while working with Android Canvas. By the end of this chapter, the reader will have learned to create a basic Android application that will display an image using one of the simple techniques provided by Canvas and some additional information and good conventions to deal with graphics in applications.

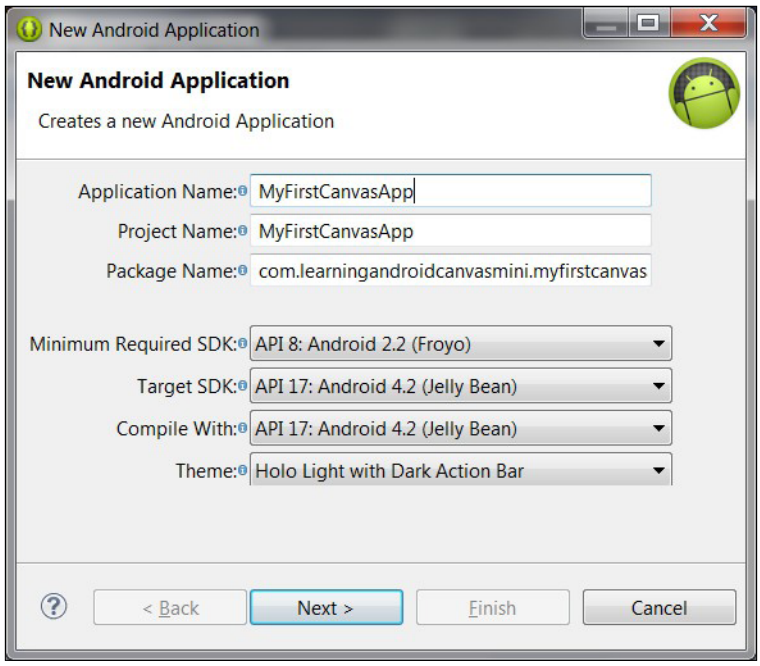
Android Canvas

Android Canvas provides the developer with the ability to create and modify 2D images and shapes. Moreover, the Canvas can be used to create and render our own 2D objects as this class provides various drawing methods to do so. Canvas can also be used to create some basic animations such as frame-by-frame animations or to create certain `Drawable` objects such as buttons with textures and shapes such as circles, ovals, squares, polygons, and lines. Android also provides hardware acceleration to increase the performance of the drawing done by Canvas. Now we know what we want to develop, why we need to know about graphics, what our graphic needs are, and what we will use. We also know what Android Canvas is and what it provides us with. In short, Android Canvas is the answer to all our questions as it provides all the right ingredients for our graphics and some basic animation to get the job done. For 3D graphics, Android provides support for OpenGL; but in the context of this book, we will not cover 3D graphics and so will not discuss OpenGL. However, interested readers can check the details on `http://developer.android.com/guide/topics/graphics/opengl.html`.

Our first Android Canvas application

Our goal in this section will be to create a very simple application that will display an image on the screen. Do not get frightened. It might seem too early to start developing Canvas—and it is—but our first application will not involve any coding or any complicated stuff. We will go through the different steps of creating an application. We will look into SDK versions and available APIs and what we are going to use. You may not understand certain sections of the applications or the steps that we are performing, but after performing each step we will explain what's going on in that step. We have only talked up to this point; let's start some real work. We will start by opening Eclipse and starting a new Android application project. To do this, in Eclipse we will navigate to **File** | **New** | **Android Application Project**.

The following screenshot shows the new application window in Eclipse:



This is the first screen that appears. We will name our application `MyFirstCanvasApp`.

Application Name

Application Name is shown in the **Manage Application** list in **Settings**; if we publish our application on **Play Store**, the same string will be displayed there. So the name should be attractive and meaningful.

Project Name

Project Name is only used by Eclipse, but it should be unique within the workspace. It can be the same as the application name.

Package Name

Package Name must be a unique identifier of our application. It is not shown to the users, but it should stay the same throughout the life of the application. The package name is something with the help of which different versions of the same applications are identified. Our package name will be `com.learningandroidcanvasmini.myfirstcanvasapp`. This specific syntax is not a hard and fast rule to define package names, but it's a good way to avoid package name collisions. For example, if we have two applications with the same exact name, as follows:

- `com.learningandroidcanvasmini.myfirstcanvasapp`
- `com.learningandroidcanvasmini.myfirstcancasapp`

Let's say that the first application is just a simple application displaying some images and the second application is a simple paint application with freehand drawing. If we want to publish them on Google Play Store, a package name collision will occur and the second application will not be allowed to get published because there will already be an application with the same exact name. There are two solutions to avoid this situation. First, changing the package name so that there is no collision in package names and the second application is considered as an all new application. For example:

- `com.learningcandroidcanvasmini.picdisplayapp`
- `com.learningandroidcanvasmini.paintapp`

Second, we can keep the package name the same but change the `android:version` code and `android:version` name. Publishing will be successful, giving the impression that the first application is the core application (like `myfirstcanvasapp` Version 1.0) and the second application is the updated version of the same app (like `myfirstcanvasapp` Version 2.0). For more details and better understanding of publishing applications on Google Play Store, follow the link `http://developer.android.com/distribute/googleplay/publish/preparing.html`.

Least possible version of SDK

Selecting a lower possible required SDK means that our application will be able to run on the maximum available devices but with the limitation that if we select a very low version of Android, we won't be able to use hardware acceleration as it is not available on the lower versions of Android. If we don't require hardware acceleration, we can go for lower versions to target more devices, but if we are using some animations and graphics that may require hardware accelerations, we need to use a little higher version of Android.

Target SDK

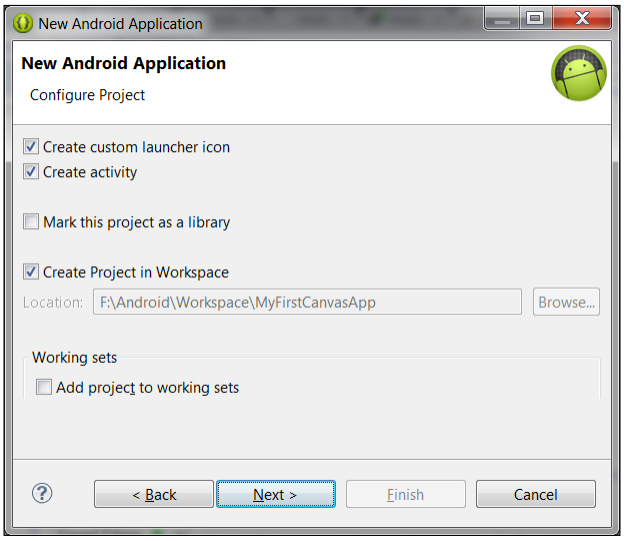
This is the highest API level our application will work with, but applications are generally forward compatible and will even work with SDK levels higher than our target SDK if the libraries used are not changed. This is a rare scenario but sometimes some libraries in the new versions are changed and that affects calls to different methods and properties of different classes, and this results in abnormal functions of the application. This attribute informs the system that we have tested the application against the target API and the system should not create any compatibility issues. Moreover, the application will have backward compatibility till the Minimum Required SDK. We will select the highest available SDK as we won't like our application to look outdated as lower versions will lack some new features (such as lack of hardware acceleration).

Theme

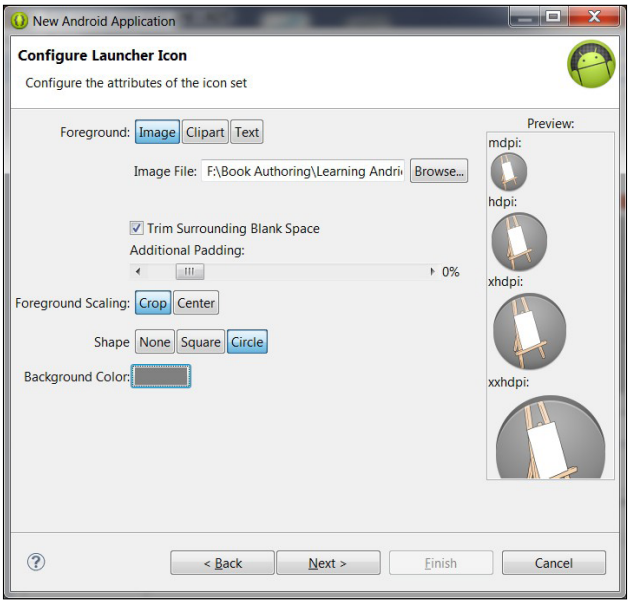
Theme is the last option. We will keep the default selection for this, as at this point it is not important.

After this, we will click on **Next** and the following screen will appear. This is the **Configure Project** screen.

The following screenshot shows the new application's configuration screen:



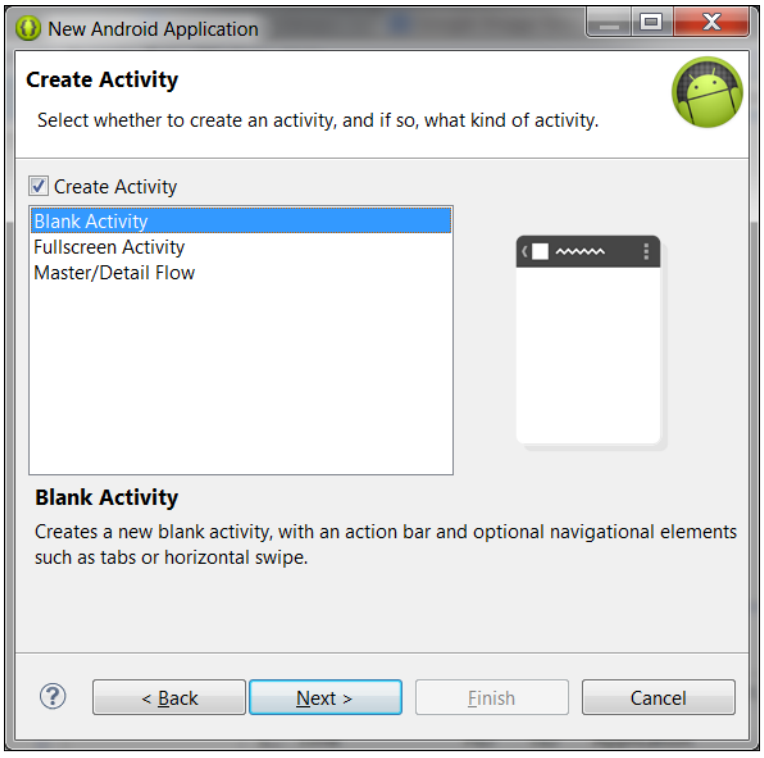
Here we will check whether we want a custom launcher icon or not. On selecting **Create Activity**, the system will create a default activity for us automatically. Also configure where to create the project; mostly, it's in our workspace. If we have selected the **Create custom launcher icon** checkbox, clicking on **Next** will take us to the following screen, the icon configuration screen:



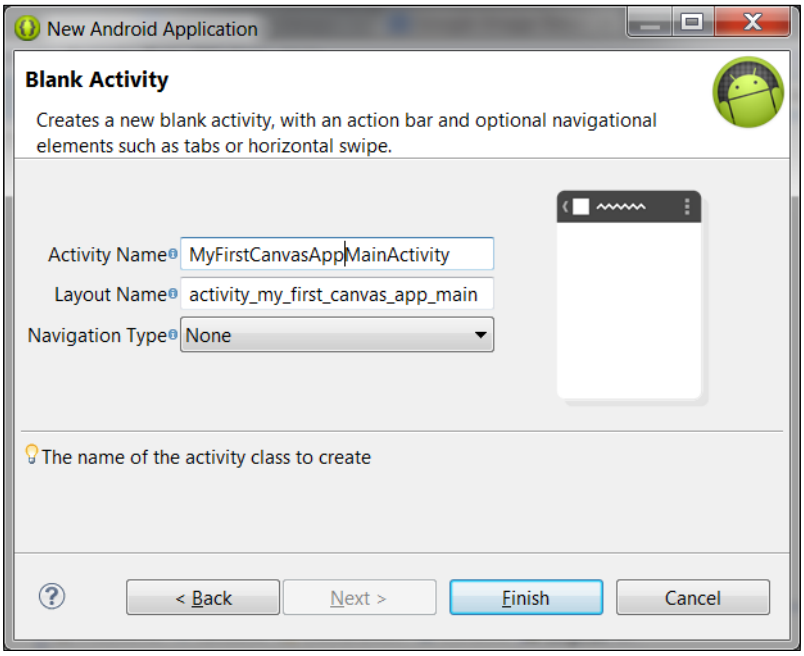
On this screen, we will configure the custom launcher icon attributes, such as the path from where it should read the source image. We will configure the icon's foreground, shape, and background color. Click on **Next** to move to the next screen. We have selected a randomly downloaded Canvas image from the web as our source. The image is in PNG format. PNG images support transparency; for example, a completely transparent image or with some transparent background. Select the **Shape** as **Round** and the **Background Color** as dark gray. Other options are to make the shape **Square** or **None**. For more details about icons, follow the link `http://developer.android.com/design/style/iconography.html`.

The form shows different sizes of the image to the right with **mdpi**, **hdpi**, **xhdpi**, and **xxhdpi** written on top of each; `dpi` is dots per inch, `m` is medium, and `h` is high. These are the different sizes of our image for different screen sizes. Android devices come in different screen sizes and resolutions. If we want our application to have support for multiple screen sizes, starting from old to new devices, we should gather some information about their dpis, screen sizes, screen resolutions, screen densities, and so on; as we are dealing with graphics here, we should know about them. However, we will come to this at the end of this chapter.

The following screenshot shows selecting the default activity screen:

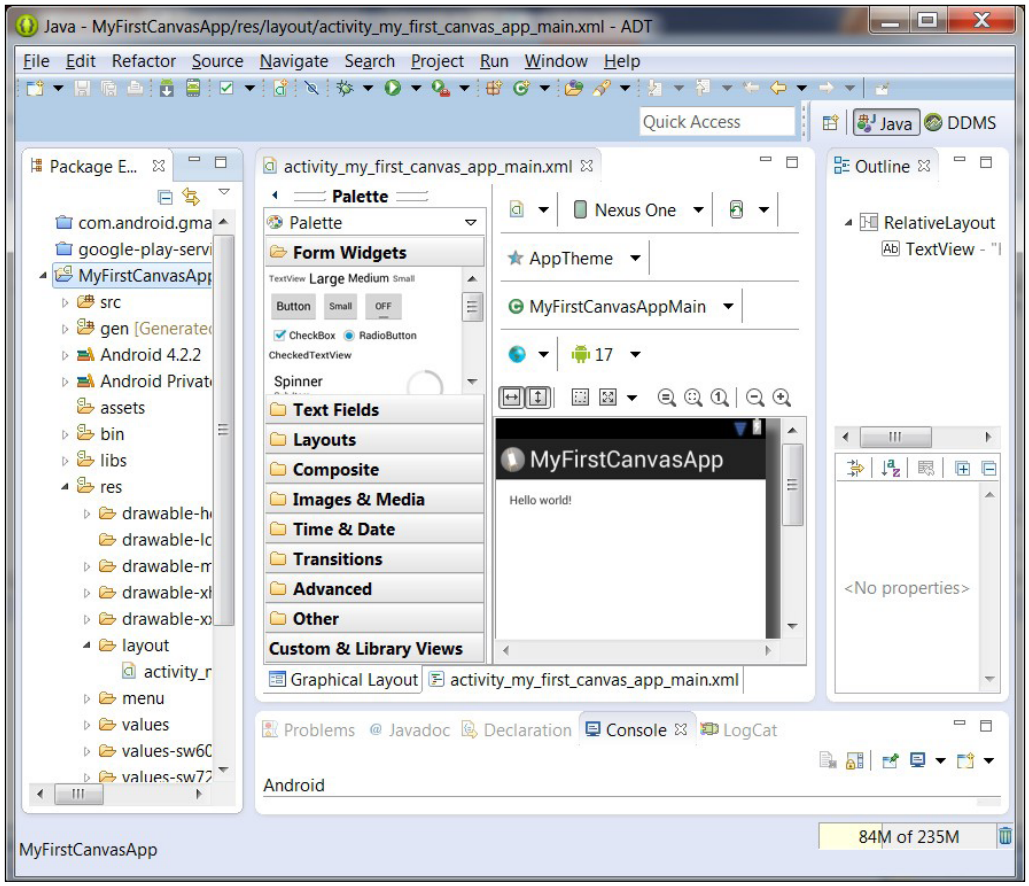


On this form, the wizard gives us options to create an application with a **Blank Activity**, **Fullscreen Activity**, or **Master/Detail Flow** activity form. We will move forward with a blank activity by selecting **Blank Activity** and clicking on **Next**. Now the wizard will take us to the following form, the default activity form:



On this form, we will name our activity as `MyFirstCanvasAppMainActivity`. The **Layout Name** will be populated automatically for us and **Navigation Type** should be selected as **None** because at this point we are not interested in any navigation complications in our application.

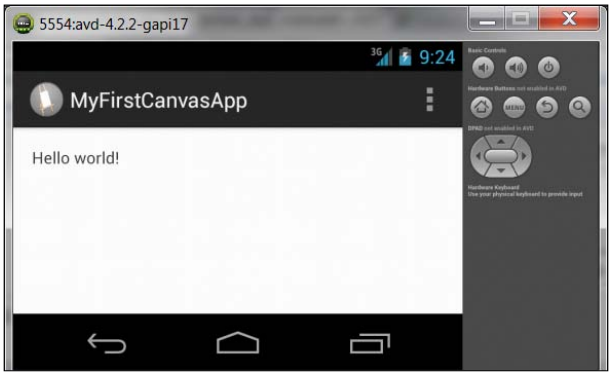
Clicking on **Finish** will close the wizard and we will be taken back to Eclipse with the screen as shown in the following screenshot, which shows our application in the **Design** mode.The following screenshot shows the first Eclipse view of our project after the wizard has completed execution successfully:



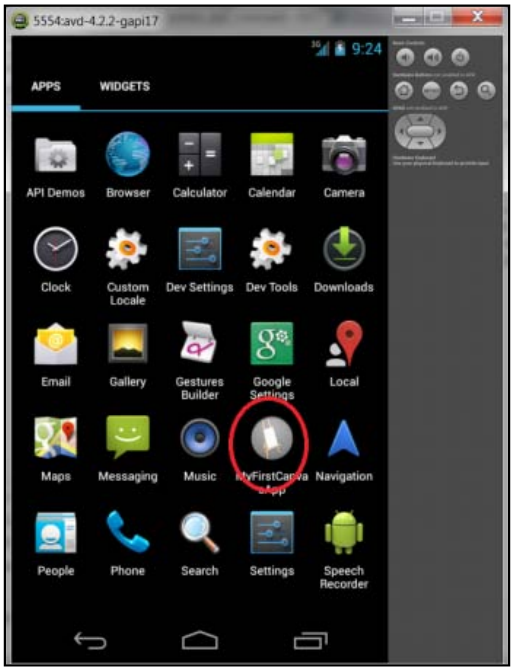
Here we will create an **Android Virtual Device** (**AVD**) and our emulator from the AVD Manager with the configuration of Target SDK Ver. 4.2.2 to test our application. To do this, we will right-click on our project in the **Package Explorer** panel that appears on the left side by default. From the menu that appears, we will navigate to **Run As | Run Configurations**. In this window, we will select our project in the **Android** tab. Then we will navigate to the **Target** tab and select the AVD that we created earlier to test our application and click on **Run**. This will trigger the emulator to run and our application will be shown in the emulator as shown in following screenshot.

If we click on the Home key on the emulator and then click on the menu to see all the applications installed on the emulator, we will see that our custom launcher icon is also present in the menu as shown in the following screenshot. To give a good impression of our application, we must design an attractive, relevant icon for us. For this, Photoshop or any other graphic a design tool can be used. This step can be skipped if the developers own an actual device and it's configured for testing in Eclipse.

The following screenshot shows the first default activity with **Hello world!** displayed:



The following screenshot shows our application icon in the fourth row from top:



Now that we have our first application up and running, we will try to understand the most important sections of our project in Eclipse. After understanding the important sections, we will meet our goal; that is, displaying an image on the screen.

Mining our first application

For starters, in every Android application there are three files that need our special attention; without understanding these three files, we won't be able to develop applications for Android. The following sections talk about these files.

The configuration file

The main configuration file in every Android Application is `AndroidManifest.xml`. This is an XML file and can be seen at the root level of the project in **Package Explorer**. This is the main configuration file of our application and one of the most important files of our project. This file contains the information of the package or our application, the minimum and maximum SDKs used by our application, the activities used in our application, and permissions that our application needs to run or perform certain specific tasks. Whenever an application is about to get installed on an Android device, this is the file that provides the system with all the details about the permissions to resources that the application will require and the activities used in it. After reading this file, the system knows about the package name of this application, what the compatible SDKs are, what activities the application contains, and what permissions the application needs to run or perform certain tasks.

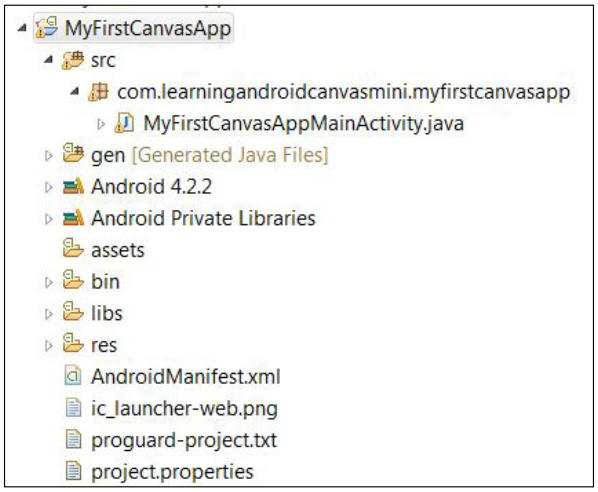
The layout file

The layout file in our application is `activity_my_first_canvas_app_main.xml` and is located in the `layout` folder that's inside the `res` folder. So the complete path is `res/layout/activity_my_first_canvas_app_main.xml` in our project in **Package Explorer**. This is an XML file that is responsible for the layout of our activity and the views that appear on the activity in our application. Other activities can be laid out using the same XML format using the same XML layout files.

The code file

The main activity code file in our application is `MyFirstCanvasAppMainActivity.java`. This is the coding file of our activity where we code all our functionality. This file is located in the package folder of our project; that is, inside the `src` folder, so the path inside the project in the **Package Explorer** becomes `src/com.learningandroidcanvasmini.myfirstcanvasapp/MyFirstCanvasAppMainActivity.java`.

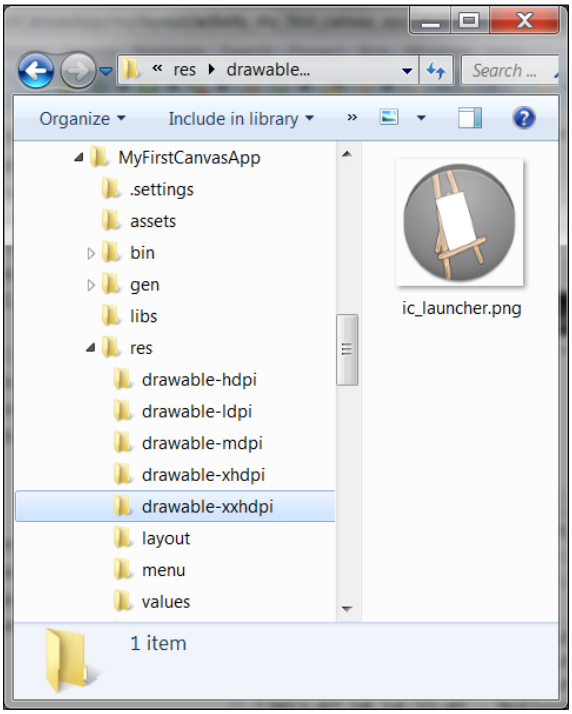
The following screenshot shows it clearly:



Besides the preceding files, we will discuss the `res` folder. The `res` folder contains the following `Drawable` folders:

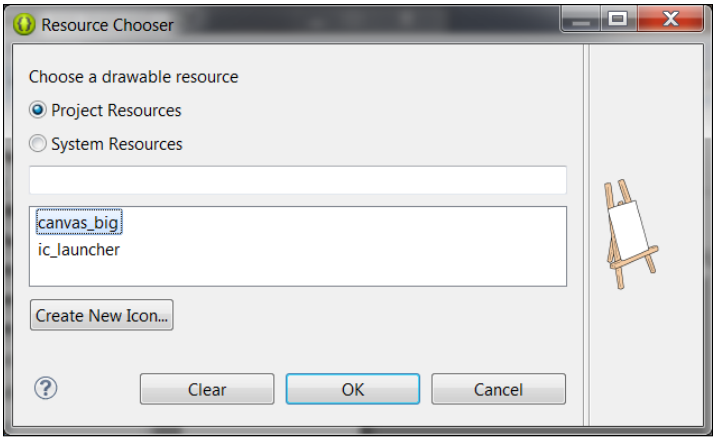
- `drawable-hdpi`
- `drawable-ldpi`
- `drawable-mdpi`
- `drawable-xhdpi`
- `drawable-xxhdpi`

The following screenshot displays the `drawable-xxhdpi` folder inside our `res` folder. This is where our icon is placed.



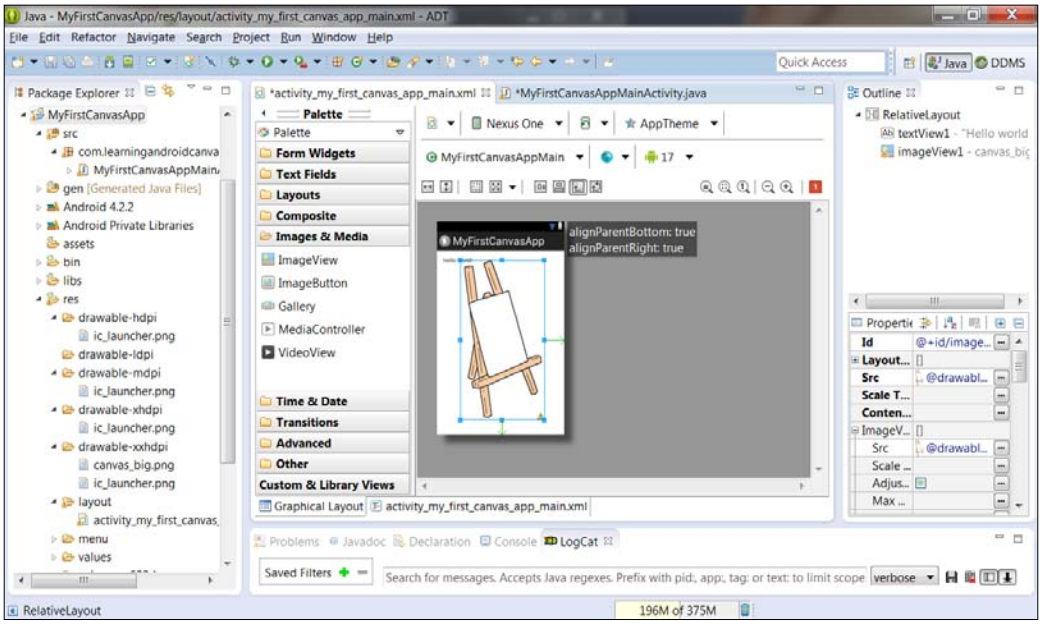
If we check all these folders, we will find that each one contains an image by the name `ic_launcher.png` that is in fact the Canvas image we used during the creation of our application. The image is the same in each folder, but the size is different. Now let's say we want the original Canvas image to be displayed on the screen. We will copy our original Canvas image to one of these folders; let's say we copy the image to the `drawable-xhdpi` folder. Refresh the folders in the Package Explorer and then go to the **Design** view of our activity that is displaying the **Hello world** string. Select the string and delete it. In the Palette, click on **Images & Media** to expand it. Drag-and-drop an ImageView on the activity in the **Design** view. We will be prompted by a dialog box to select the source image for the ImageView.

The following screenshot shows the dialog box that prompts us to select the source image for the ImageView that we have dragged-and-dropped on the activity in the **Design** mode:

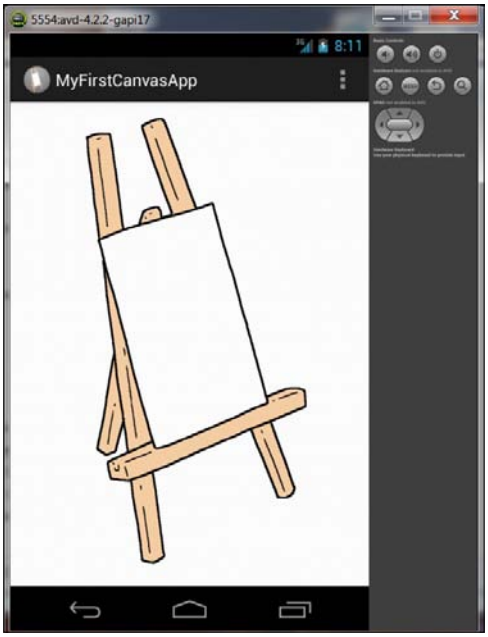


We will select our Canvas image and click on **OK**. Our Canvas image will appear on the screen.

The following screen shows the image that appears on the screen in the **Design** mode:



We will run the application. The following is what we will get on the emulator – our application with the image drawn on an ImageView that uses Canvas:



This is something we need to be very careful about: when we are saving image files in the `res` folder, we need to rename our image files carefully. Although the image filename may not make any difference outside this project, in Eclipse the image filename will give you errors if one of the following mistakes is made:

- Filenames containing spaces; for example, `our canvas.png`:

  This will return an error and will not be shown in the Package Explorer pane in our project files. Filenames containing special characters other than _ and `.`; for example, `our-canvas(1).png` will also return an error.

The following screenshot displays the error that will be shown:

```
error opening trace file: No such file or director ⏎
y (2)
```

- Filenames that don't start with alphabet characters; for example, `886_1_Canvas.png`. This will return an error.

The best convention to name image files is to start with alphabet characters; numbers can be included after that. Use only _ and . among the special characters; for example, `our_canvas_1.png`. This filename will be accepted and we will be able to use this image in our application.

We have now completed the goal of this chapter, but we need to understand a few more things if we don't want to have any issues with the graphics of our application on different screen sizes and resolutions. If we want our application to support multiple screens, we need to understand the following:

- **Screen size**: This is the physical screen size, normally measured in inches diagonally. Android has four groups for all displays: small, normal, large, and extra large.
- **Screen density**: This is the dpi (dots per inch). These are the number of pixels on a physical area. This means that a 3-inch high-density screen will have more pixels than a 6-inch low-density screen. Lower dpis means low-density screens and higher dpis means high-density screens. Android has four groups for densities: low, medium, high, and extra high. This is where `ldpi`, `hdpi`, `mdpi`, `xhdpi` come in. For more details on screen size and density, follow this link: `http://developer.android.com/guide/practices/screens_support.html`.
- **Orientation**: This is the orientation of the screen. This is either portrait or landscape. We need to take care of this because different devices operate in different modes and orientation can be changed at run time by the user. So, if we are only designing for one of them, we have to lock the orientation of the screen. Then, even if the user rotates the screen, the graphics of our UI stay the same. It is best to design layouts and graphics for both orientations.

Summary

In this chapter we have learned the following:

- The need to learn about Android Canvas
- What Android Canvas is and what it provides us
- Creating a simple and basic application displaying an image on the screen
- Understanding important files and sections of our first application
- Some additional information about screen sizes, densities, and orientations

By the end of this chapter the reader will know the basics of working with graphics in Android. The reader will be able to create a simple application displaying an image on the screen and will have a basic understanding of the different sections of the project. He/she will also have gained knowledge about which file to use for what purpose and where to find it. The reader will also have gained some basic knowledge of screens and densities; thus, while designing and using graphics in an application, the reader will use this information to make better decisions about designing the user interface of the application.

In the next chapter we will discuss threads, their importance, how can they be used in drawing in Canvas, and the performance tradeoff our application gets while using threads.

2
Drawing Threads

A thread is the smallest sequence of instructions that can be managed by the operating system independently. Normally, a single task is achieved in a single thread in a single process but, if we want to change the normal behavior and want multiple tasks to run at the same time, we will use multi-threading. The threads share the same process resources but are executed independently. If the system has a single processor, it may appear that the tasks are being processed at the same time, but they are not. In reality, with a single processor, the allocation of processing threads is switched from thread to thread but the switching is so fast that it appears to be processed at the same time. If the system has multiple processors, two threads can be executed at the same time—in parallel to each other. Multi-threading is an execution model in which more than one thread can be executed in a single process. Threads can be used in a number of ways and each has its own importance; for example, either to execute multiple tasks at the same time or just to move away the load from the main thread if something needs to be processed in the background and the frontend stays responsive and active. This is the ideal situation for Android applications as we have to keep the maximum possible load away from the main thread and keep the frontend responsive and active. If we don't do this and retain heavy operations that consume a lot of processor power and memory, the application may end up nonresponsive or even ask us for a forced close.

The objective of this chapter is to clear the understanding of threads in Android. We will not dig deep into the details of threads and their coding, but will have a basic understanding of threads in Android. What are the known issues with this? Why are Android threads important in terms of drawing and Canvas? We will see a simple structure of code through which we want our tasks to run on another thread.

All applications run on a single thread in Android. All instructions run in a sequence, meaning that the second instruction will not start unless the first one is finished. This main thread is also called the **UI** (**User Interface**) thread as it is responsible for drawing all the objects or views on the screen and processing all events, such as screen touches and button clicks. Now the problem is that, if we have two operations scheduled to run in the same default thread or UI thread and the first operation takes too long to finish, the system will ask the user to forcibly close the application or wait for the process to complete. This scenario is called **ANR** (**Application Not Responding**).

The need for drawing threads

We know we will work with images, drawings, and other graphics processing and we also know that they are pretty heavy on system resources. So we would like to design our application very carefully, keeping performance in mind. What if we overlook this point and put all our images, bitmaps, graphics, and other graphics processing items on the default UI thread? This is the way new Android developers work—keep everything in the code of the default activity, meaning keep the entire load on the UI thread. The default activity is the activity that we want to load first when our application runs. The UI thread is the main thread of execution of our application. This is the thread where most of the code of the application runs. Application components such as `Activities`, `Services`, `ContentPorviders`, and `BroadcastReceivers` are created in this thread. What will happen in this case? Even if our application is the most useful and most attractive application on the planet, it won't last a day. Every time users run our application, it will end up nonresponsive. A few angry comments from users on Play Store, and our application is done. We will lose the idea, as by that time it will be public, and we will also lose our audience. To cope up with this problem, we will take away all the hard work and heavy load from the main UI thread to another thread. Ideally, when this runs, it will look like all the threads are running in parallel, but that's only in the case of multiple CPUs. If there is only a single CPU but multithreading is supported, the system will decide which thread to start and which to stop, but no thread will be permanently stopped. So the control will switch between the running threads and it will look as if all threads are running in parallel.

The issue with multithreading in Android

We will put our heavy, resource-consuming operations on a separate thread, but in Android this generates a problem and that is the reason why no other thread is allowed to update the main UI thread that is responsible for all UI elements and processes. To cope up with this issue, we need to sync it with the current state of the UI thread. Android provides a class that specifically deals with this issue. It's called the `AsyncTask` class. We will discuss this later in this chapter.

The Thread class

The `Thread` and `Runnable` classes are the basic classes that enable us to use multithreading, but they have a very limited functionality but still do provide the base for `AsyncTask`, `HandlerThread`, `IntentService.Thread`, and `ThreadPoolExecuter`. These classes automatically manage threads and can run multiple threads in parallel.

The following is a piece of sample code of the `Runnable` class:

```
public class ImageReSize implements Runnable {
  public void run(){
    //the main functionality of the thread comes here
  }
}
```

If we want our thread to run in the background, we will add the following line to the `run()` method mentioned previously:

```
Android.os.Process.setThreadPriority(Android.os.Process.THREAD_
PRIORITY_BACKGROUND);
```

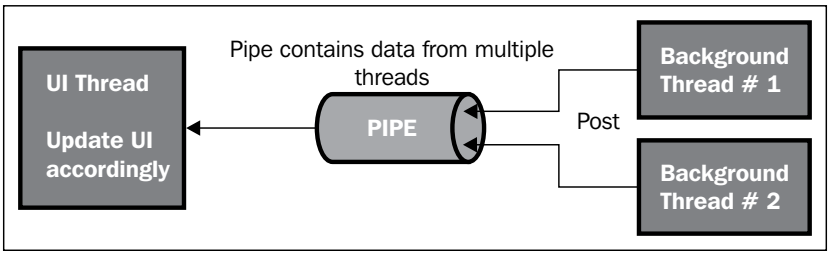
Let's say we have our `Runnable` class. We still won't be able to display anything on the user interface because only the UI thread executes the UI objects such as views. Objects running on the UI thread have access to other objects. Now the tasks that are running on our thread are not on the UI thread and therefore they don't have access to UI objects. To give our tasks access to the UI objects on the UI thread, we must use something with which we can move data from the background thread to the UI thread. If a thread is running in the background and it needs to change something on the UI, it can't do that by itself but will use the facility provided by `runOnUiThread`, which will enable us to run the code inside a background thread on the main UI thread. Optionally, we can use `Handler` objects.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. 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.

Handler objects that run on the UI thread

To code `Handler` objects that run on the UI thread, firstly a `Handler` object should be defined on the UI thread and then the data should be stored in a `Task` object. The status of the object should be sent up the hierarchy of objects. When we are done with this, the data should be moved to the UI thread. We do all this to achieve our goal of running tasks on another thread in the background and when needed communicate with the main UI thread to get our desired output. However, this requires a lot of efforts.



AsyncTask

To significantly reduce workload and complications, Android provides the `AsyncTask` class. This class will allocate our tasks to run on another thread in the background and when needed will communicate with the UI thread automatically, saving us the time and effort we will spend on `Handler` objects. To get the job done, we will create a class that will extend `AsyncTask`, will put in our functionality, and will execute our application. `AsyncTask` will do a lot for us.

Summary

In this chapter we have learned about the following:

- The thread structure of a simple Android application
- The role of the UI thread and its power
- The need to keep heavy resource-consuming operations away from the main UI thread
- The limitations of threads that are not the UI thread
- How to handle problems using `Handler` objects and communicate with the UI thread
- The code structure of a class that implements the `Runnable` interface, which enables us to use threading
- The facility that Android provides us in the form of the `AsyncTask` class; we also learned about its importance

In the next chapter we will learn about the `Drawable` class and drawing on Canvas using images from resources and XML. Drawing on `View` and `SurfaceView` and drawing of basic shapes such as circles will also be covered.

3
Drawing and Drawables in Android Canvas

In this chapter our goal is to learn about the following:

- Drawing on a Canvas
- Drawing on a View
- Drawing on a SurfaceView
- Drawables
- Drawables from resource images
- Drawables from resource XML
- Shape Drawables

Android provides us with 2D drawing APIs that enable us to draw our custom drawing on the Canvas. When working with 2D drawings, we will either draw on view or directly on the surface or Canvas. Using View for our graphics, the drawing is handled by the system's normal View hierarchy drawing process. We only define our graphics to be inserted in the View; the rest is done automatically by the system. While using the method to draw directly on the Canvas, we have to manually call the suitable drawing Canvas methods such as `onDraw()` or `createBitmap()`. This method requires more efforts and coding and is a bit more complicated, but we have everything in control such as the animation and everything else like being in control of the size and location of the drawing and the colors and the ability to move the drawing from its current location to another location through code. The implementation of the `onDraw()` method can be seen in the drawing on the view section and the code for `createBitmap()` is shown in the *Drawing on a Canvas* section.

We will use the drawing on the View method if we are dealing with static graphics—static graphics do not change dynamically during the execution of the application—or if we are dealing with graphics that are not resource hungry as we don't wish to put our application performance at stake. Drawing on a View can be used for designing eye-catching simple applications with static graphics and simple functionality—simple attractive backgrounds and buttons. It's perfectly okay to draw on View using the main UI thread as these graphics are not a threat to the overall performance of our application.

The drawing on a Canvas method should be used when working with heavy graphics that change dynamically like those in games. In this scenario, the Canvas will continuously redraw itself to keep the graphics updated. We can draw on a Canvas using the main UI thread, but as we discussed in maximum possible detail in *Chapter 2*, *Drawing Threads*, when working with heavy, resource-hungry, dynamically changing graphics, the application will continuously redraw itself. It is better to use a separate thread to draw these graphics. Keeping such graphics on the main UI thread will not make them go into the non-responding mode, and after working so hard we certainly won't like this. So this choice should be made very carefully.

Drawing on a Canvas

A Canvas is an interface, a medium that enables us to actually access the surface, which we will use to draw our graphics. The Canvas contains all the necessary drawing methods needed to draw our graphics. The actual internal mechanism of drawing on a Canvas is that, whenever anything needs to be drawn on the Canvas, it's actually drawn on an underlying blank bitmap image. By default, this bitmap is automatically provided for us. But if we want to use a new Canvas, then we need to create a new bitmap image and then a new Canvas object while providing the already created bitmap to the constructor of the Canvas class. A sample code is explained as follows. Initially, the bitmap is drawn but not on the screen; it's actually drawn in the background on an internal Canvas. But to bring it to the front, we need to create a new Canvas object and provide the already created bitmap to it to be painted on the screen.

```
Bitmap ourNewBitmap =
Bitmap.CreateBitmap(100,100,Bitmap.Config.ARGB_8888);
Canvas ourNewCanvas = new Canvas(ourNewBitmap);
```

Drawing on a View

If our application does not require heavy system resources or fast frame rates, we should use `View.onDraw()`. The benefit in this case is that the system will automatically give the Canvas its underlying bitmap as well. All we need is to make our drawing calls and be done with our drawings.

We will create our class by extending it from the `View` class and will define the `onDraw()` method in it. The `onDraw()` method is where we will define whatever we want to draw on our Canvas. The Android framework will call the `onDraw()` method to ask our View to draw itself.

The `onDraw()` method will be called by the Android framework on a need basis; for example, whenever our application wants to draw itself, this method will be called. We have to call the `invalidate()` method whenever we want our view to redraw itself. This means that, whenever we want our application's view to be redrawn, we will call the `invalidate()` method and the Android framework will call the `onDraw()` method for us. Let's say we want to draw a line, then the code would be something like this:

```
class DrawView extends View {
  Paint paint = new Paint();
  public DrawView(Context context) {
    super(context);
    paint.setColor(Color.BLUE);
  }
  @Override
  public void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawLine(10, 10, 90, 10, paint);
  }
}
```

Inside the `onDraw()` method, we will use all kinds of facilities that are provided by the Canvas class such as the different drawing methods made available by the Canvas class. We can also use drawing methods from other classes as well. The Android framework will draw a bitmap on the Canvas for us once our `onDraw()` method is complete with all our desired functionality. If we are using the main UI thread, we will call the `invalidate()` method, but if we are using another thread, then we will call the `postInvalidate()` method.

Drawing on a SurfaceView

The `View` class provides a subclass `SurfaceView` that provides a dedicated drawing surface within the hierarchy of the View. The goal is to draw using a secondary thread so that the application won't wait for the resources to be free and ready to redraw. The secondary thread has access to the `SurfaceView` object that has the ability to draw on its own Canvas with its own redraw frequency.

We will start by creating a class that will extend the `SurfaceView` class. We should implement an interface `SurfaceHolder.Callback`. This interface is important in the sense that it will provide us with the information when a surface is created, modified, or destroyed. When we have timely information about the creation, change, or destruction of a surface, we can make a better decision on when to start drawing and when to stop. The secondary thread class that will perform all the drawing on our Canvas can also be defined in the `SurfaceView` class.

To get information, the `Surface` object should be handled through `SurfaceHolder` and not directly. To do this, we will get the Holder by calling the `getHolder()` method when the `SurfaceView` is initialized. We will then tell the `SurfaceHolder` object that we want to receive all the callbacks; to do this, we will call `addCallBacks()`. After this, we will override all the methods inside the `SurfaceView` class to get our job done according to our functionality.

The next step is to draw the surface's Canvas from inside the second thread; to do this, we will pass our `SurfaceHandler` object to the thread object and will get the Canvas using the `lockCanvas()` method. This will get the Canvas for us and will lock it for the drawing from the current thread only. We need to do this because we don't want an open Canvas that can be drawn by another thread; if this is the situation, it will disturb all our graphics and drawings on the Canvas. When we are done with drawing our graphics on the Canvas, we will unlock the Canvas by calling the `unlockCanvasAndPost()` method and will pass our Canvas object. To have a successful drawing, we will need repeated redraws; so we will repeat this locking and unlocking as needed and the surface will draw the Canvas.

To have a uniform and smooth graphic animation, we need to have the previous state of the Canvas; so we will retrieve the Canvas from the `SurfaceHolder` object every time and the whole surface should be redrawn each time. If we don't do so, for instance, not painting the whole surface, the drawing from the previous Canvas will persist and that will destroy the whole look of our graphic-intense application.

A sample code would be the following:

```
class OurGameView extends SurfaceView implements
SurfaceHolder.Callback {
  Thread thread = null;
  SurfaceHolder surfaceHolder;
  volatile boolean running = false;
  public void OurGameView (Context context) {
    super(context);
    surfaceHolder = getHolder();
  }

  public void onResumeOurGameView (){
    running = true;
    thread = new Thread(this);
    thread.start();
  }
    public void onPauseOurGameView(){
  boolean retry = true;
  running = false;
  while(retry){
    thread.join();
    retry = false;
  }

  public void run() {
  while(running){
    if(surfaceHolder.getSurface().isValid()){
      Canvas canvas = surfaceHolder.lockCanvas();
    //... actual drawing on canvas
      surfaceHolder.unlockCanvasAndPost(canvas);
      }
    }
  }
}
```

Drawables

The two-dimensional graphics and drawing library that Android provides is called **Drawable**. The exact package name is `android.graphics.drawable`. This package provides all the necessary classes for drawing our 2D graphics.

In general, a Drawable is an abstraction for something that can be drawn. Android provides a number of classes that extends the `Drawable` class to define special types of Drawable graphics. The complete list can be found at `http://developer.android.com/reference/android/graphics/drawable/package-summary.html`.

Drawables can be defined and instantiated in three ways:

- From an image saved in the `resource` folder of our project
- From an XML file
- From the normal class constructor

In the context of this book, we will explain only the first two methods.

Drawables from a resource image

This is the quickest and simplest method to add graphics to our application. We have already explained the different important folders of our project and discussed in detail which folder contains what kind of files in *Chapter 1*, *Getting Started with Android Canvas*. By the end of this chapter, we will know how to copy an image to the `resource` folder and where to find the `resource` folder.

We will use the image that we have already copied in the `res/drawable` folder in our applications project in *Chapter 1*, *Getting Started with Android Canvas*. The image name is `lacm_5396_01_14.png` and the exact location is `res/drawable-xhdpi`. One important point here is that the supported formats are PNG, JPEG, and GIF. The most preferable format to use is PNG and the least preferable is GIF. Whenever we put an image in the `res/drawable` folder, during the build process, the image will be compressed with lossless compression to save system memory; this process is automatic. The compressed images normally retain the same quality but of a much lesser size. If we don't want the system to compress our images, we should copy our images to the `res/raw` folder.

We will use the same application source code from *Chapter 1*, *Getting Started with Android Canvas*, to explain this section of the chapter. We will open our project `MyFirstCanvasApp`. This is the code before we make any changes:

```
package com.learningandroidcanvasmini.myfirstcanvasapp;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
public class MyFirstCanvasAppMainActivity extends Activity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_my_first_canvas_app_main);
  }
```

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
      // Inflate the menu; this adds items to the action bar if it
      //is present.
      getMenuInflater().inflate(R.menu.my_first_canvas_app_main,
        menu);
      return true;
    }
}
```

We will open our layout file `activity_my_first_canvas_app_main.xml` in **Design** view. We will delete the already placed **ImageView** object from the Activity that we added in *Chapter 1, Getting Started with Android Canvas*. Now we will open our code file again and will add the following lines of code to the preceding code step-by-step. In our main activity class, we will define a `LinearLayout` object:

```
LinearLayout myLinearLayout;
```

This will be our custom layout on which we want to display our image using this code. Then, inside our main activity class, we will instantiate the `LinearLayout` object:

```
myLinearLayout = new LinearLayout(this);
```

Next we will add the following lines of code to our file:

```
ImageView MySecondImageView = new ImageView(this);
MySecondImageView.setImageResource(R.drawable.lacm_5396_01_14);
MySecondImageView.setAdjustViewBounds(true);
MySecondImageView.setLayoutParams(new
  ImageView.LayoutParams(LayoutParams.WRAP_CONTENT,
  LayoutParams.WRAP_CONTENT));
myLinearLayout.addView(MySecondImageView);
setContentView(myLinearLayout);
```

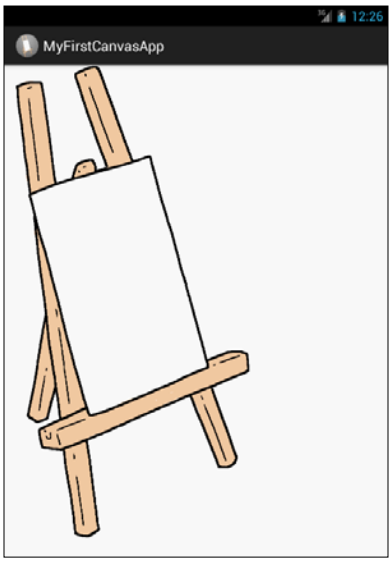
In the preceding block of code, first, we have defined an `ImageView` object. Then we set the source to an image that we want our `ImageView` object to display. In the next line, we have adjusted the View bounds so that the `ImageView` bounds match the width and height of the source image. The `setLayoutParams` method will help us wrap the view borders around the image content even if there is a difference in the dimensions. After this, we will supply our `ImageView` control to our custom layout using the code line:

```
myLinearLayout.addView(MySecondImageView);
```

In the last line, we will set the activity layout to our custom layout. To do this, we will set the content view to our custom layout:

```
setContentView(myLinearLayout);
```

Now, we will test our application in the emulator and then we will see the following on the emulator screen:



If we compare this output image on the activity screen to what we had in *Chapter 1*, *Getting Started with Android Canvas*, there is not much difference in what we see. We achieved the same output in *Chapter 1*, *Getting Started with Android Canvas*, very easily. Then, why have we gone through all this complicated coding to achieve the same output in this chapter?

We went through all the hard work and complicated code because in *Chapter 1*, *Getting Started with Android Canvas*, we hardcoded the `ImageView` object to display only one image that we have defined in the **Properties** tab in the **Design** view. Now, when we deleted the `ImageView` object from the screen in the **Design** view and started coding, there was nothing on the screen in the **Design** view at that time. What we did in the preceding example was create our own custom layout that will host our graphics and drawing. We created an `ImageView` object and supplied it with a source image and, set its other properties. Later on, we added the `ImageView` object to our custom layout and at the end, we asked the activity to appear on the screen without a custom-created layout and the automatic layout. The code gives us the flexibility to keep our graphics application dynamic. We can supply our application with run-time images controlled from our code logic.

The complete code looks like this now:

```
package com.learningandroidcanvasmini.myfirstcanvasapp;
import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.widget.ImageView;
import android.widget.LinearLayout;

public class MyFirstCanvasAppMainActivity extends Activity {
  LinearLayout myLinearLayout;
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_my_first_canvas_app_main);
  myLinearLayout = new LinearLayout(this);
  ImageView MySecondImageView = new ImageView(this);
  MySecondImageView.setImageResource(R.drawable.lacm_5396_01_14);
  MySecondImageView.setAdjustViewBounds(true);
  MySecondImageView.setLayoutParams(new
    ImageView.LayoutParams(LayoutParams.WRAP_CONTENT,
      LayoutParams.WRAP_CONTENT));
  myLinearLayout.addView(MySecondImageView);
  setContentView(myLinearLayout);
  }
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it
    //is present.
    getMenuInflater().inflate(R.menu.my_first_canvas_app_main,
      menu);
    return true;
  }

}
```

If we want our resource image to be handled as a Drawable, we will create a `Drawable` object from our resource image:

```
Resources myRes = mContext.getResources();
Drawable myImage = myRes.getDrawable(R.drawable.5396_01_14);
```

Here, we need to understand that each resource in our `Resources` object can maintain only one state at a time. If we are using the same resource image in two different instances and we update the property of one instance, the image in the second instance will also reflect that change. So, whenever we are dealing with multiple instances of our `Drawable` object, instead of changing the Drawable itself, we can create a tween animation.

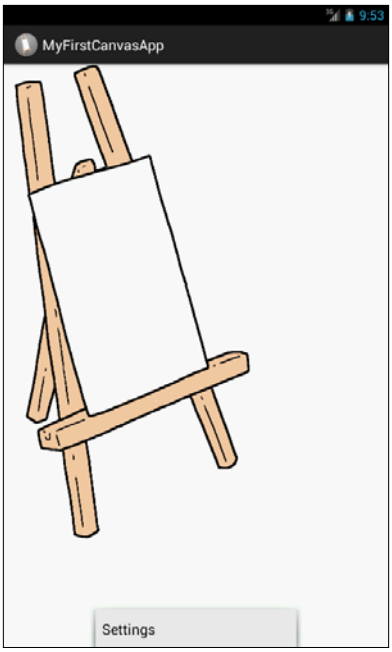
Drawables from resource XML

For those developers who have a little background of Android development, know that every activity in our application has an XML layout file. In this file, every View or control that we drag-and-drop on the Activity has an XML markup defined. So, we assume that the developers that are reading this book know how the user interface works while developing for Android. Objects can be defined and initialized in XML. If we are dealing with a graphic whose properties do not depend on what we plan to do in our code, or perhaps the graphic is static, it's a good way to define the graphic object in XML. Once the graphic is instantiated, its properties can always be tweaked according to need.

We will save the file in `res/drawable`, define the Drawable in XML, and get the Drawable by calling `Resouces.getDrawable()`. This method will take the resource ID as an argument from our XML file.

To exemplify, and to understand which Drawable can use this method and how we can have a look at the menu that's automatically created in our application, note the method in the preceding code `onCreateOptionMenu()`. When we click on the **Menu** button on the screen or from the hardware keys, we see a small menu at the bottom of the screen, named **Settings**. The menu has no functionality at this point. Now if we check the code of `onCreateOptionMenu()`, we see a call to the `inflate()` method. We can define any Drawable in XML that supports the `inflate()` method. The previously mentioned menu is a simple example of this.

The **Settings** menu can be seen in the following screenshot:



Let's say we want to go for an expand-collapse transition Drawable; the following code will get the job done for us in XML. This XML code will be saved in the `res/drawable expand_collapse.xml` file.

```
<transition
xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:drawable="@drawable/image_expand">
    <item android:drawable="@drawable/image_collapse">
      </transition>
```

The `expand` and `collapse` files are two different images saved in the `drawable` folder in our project. Now to get this transition working, we will need the following code:

```
Resources myRes = mContext.getResources();
TransitionDrawable myTransition = (TransitionDrawable)
  res.getDrawable(R.drawable.expand_collapse);
ImageView myImage = (ImageView) findViewById(R.id.toggle_image);
  myImage.setImageDrawable(myTransition);
```

First, we created a `resources` object from resources and asked the object to get everything from these resources (the resources are all the images and XML files that we have saved in the subfolders of the `res` folder in our project). Then, we created a `TransitionDrawable` object and asked the object to get the `expand_collapse` file from the `res/drawable` folder. After this, we will create an `ImageView` object that will get another view named `toggle_image`. In the last line of the preceding code, we set the Drawable type to the already-created transition.

Now including the following line of code will run the transition with a speed of once per second:

```
myTransition.startTransition(1000);
```

We won't go into too much detail about these transitions and animations as animation by itself is a very lengthy topic. But I can explain some key types of animation that can be done in Android while working with graphics to give the reader an idea of the domain and what animation covers. The types of animation in Android are as follows:

- Property animation
- View animation
- Drawable animation

Shape Drawables

Whenever we want to draw certain shapes dynamically or programmatically on our Canvas, Shape Drawables come in handy. With Shape Drawables, we can draw circles and all forms of circles such as ovals, squares, rectangles, and many other shapes. To explain Shape Drawables, we will start a new project just the way we did in *Chapter 1*, *Getting Started with Android Canvas*. We will name our project `MyShapeDrawablesApp` and go through the same steps as in the first chapter using a blank starting activity. Our objective of this exercise is to draw an oval on the screen with some color filled within it.

1. To do this, we will add another class inside our main activity class just before the ending bracket. We will name our class `MyCustomDrawableView` that will extend the `View` class.

   ```
   public class MyCustomDrawableView extends View {.....
   ```

2. Inside the constructor of this class, we will define our drawing. We will define a `ShapeDrawable` object and provide the `OvalShape()` method to its constructor as an argument to define the type of shape:

   ```
   myDrawable = new ShapeDrawable(new OvalShape());
   ```

3. Next, we will get the paint object and set the color for our `ShapeDrawable` object:

```
myDrawable.getPaint().setColor(0xff74fA23);
```

4. After that, we will define the dimensions of the object that is to be drawn. Let's say we want to draw an oval shape. The first `x, y` are the points from where it will start and the next are the width and height of the oval, as shown:

```
myDrawable.setBounds(x, y, x + width, y + height);
```

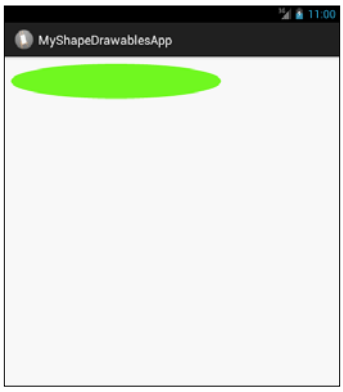
5. We will close the constructor at this point and will define the `onDraw()` method for our object. Inside this method, we will call the `draw()` method for our object.

```
protected void onDraw(Canvas canvas) {
  myDrawable.draw(canvas);
}
```

6. The next step would be to create an object of our custom class in the main activity class and set the content view to our new custom class:

```
MyCustomDrawableView myCustomDrawableView;
.
.
.
myCustomDrawableView = new MyCustomDrawableView(this);

  setContentView(myCustomDrawableView);
```

7. We will run the application in the emulator.
8. The following screenshot shows a green oval drawn on the Canvas:



The complete code for the `MyShapeDrawablesMainActivity.java` file is as follows:

```
package com.learningandroidcanvasmini.myshapedrawablesapp;
import android.os.Bundle;
import android.app.Activity;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.OvalShape;
import android.view.Menu;
import android.view.View;

public class MyShapeDrawablesMainActivity extends Activity {
  MyCustomDrawableView myCustomDrawableView;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_my_shape_drawables_main);

    myCustomDrawableView = new MyCustomDrawableView(this);
      setContentView(myCustomDrawableView);
  }
  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    // Inflate the menu; this adds items to the action bar if it is
    //present.
    getMenuInflater().inflate(R.menu.my_shape_drawables_main, menu);
    return true;
  }

  public class MyCustomDrawableView extends View {
    private ShapeDrawable myDrawable;

    public MyCustomDrawableView(Context context) {
      super(context);

      int x = 10;
      int y = 10;
      int width = 300;
      int height = 50;
```

```
        myDrawable = new ShapeDrawable(new OvalShape());
        myDrawable.getPaint().setColor(0xff74fA23);
        myDrawable.setBounds(x, y, x + width, y + height);
      }

      protected void onDraw(Canvas canvas) {
        myDrawable.draw(canvas);
      }
    }

}
```

Summary

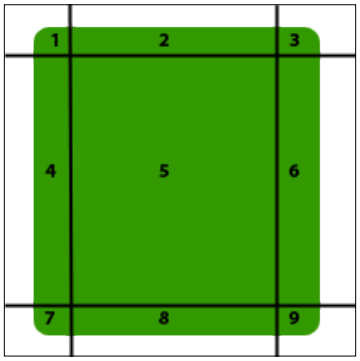
In this chapter we have learned about the `Canvas` class, its methods, properties, and how we can use them to draw. We have also learned about the `View` and `SurfaceView` classes and learned how to draw using both of these classes and which one to use in which scenario. We have also learned about Drawables and some of the different ways of using them to draw such as from images in the resources or from the XML code in resources and drawing shapes using code. We also saw a functional example while working with Shape Drawables and Drawing on a Canvas. The source code for both the example applications is downloadable from the *Packt Publishing* website. In the next chapter, we will discuss 9-patch (also known as NinePatch) images in detail. We will develop a working application using a 9-patch image and will discuss its importance in designing layouts for our mobile applications.

4
NinePatch Images

In this chapter we will learn about NinePatch images: what they are, their importance, how they are created, how they can be used, and what difference they can make in our Android applications.

NinePatch images are important because we want to develop our layout in such a way that it can adjust if either the orientation changes or our application is hosted on devices with different resolution. Our application layout needs to be flexible enough to adjust according to the resolution, screen size, and orientation of different devices. NinePatch images are images that have stretchable areas. The stretchable areas can stretch horizontally and vertically to encapsulate the content inside. The best part is that, if the width and height of the content is not defined, NinePatch images can stretch horizontally and vertically to fit all kinds of content with any type of width and height. Normally NinePatch images are used in the backgrounds of different types of containers or views that will host some kind of content. It can be used as a background for our activity in our application or it can be used only to form the background of a button with some text in it. Ninepatch images are normal PNG images with an extra 1-pixel wide border. One thing must be made sure that NinePatch images must be saved with `.9.png` file extension.

NinePatch image s are so called because the Drawable object is allowed to draw an image in nine sections. To make the idea clear, just have a look at the following diagram:



In the previous diagram there are stretchable and non-stretchable areas. In a NinePatch image, four sections will be stretchable; in the preceding example they are sections **2**, **4**, **6**, and **8.** Four sections will be non-stretchable, in the preceding example they are sections **1**, **3**, **7**, and **9**. The last section is the content section that will stretch in both the directions; in the previous example it is section **5**.

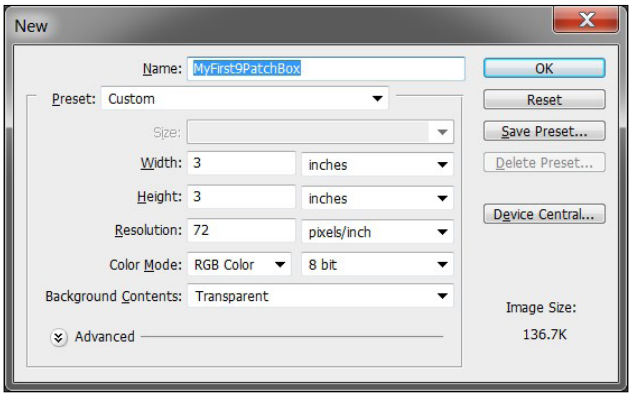
In the stretchable sections, sections **2** and **8** will stretch only horizontally. Sections **4** and **6** will stretch only vertically. Section **5** will stretch horizontally and vertically and this is the main section of the graphic that will hold the contents.

One important thing here is that it's not necessary to have the patches in the exact style as shown in the example. The NinePatch image can be created to have patches that may extend only horizontally or vertically; moreover it can have more patches than in the mentioned example.

Creating a NinePatch image

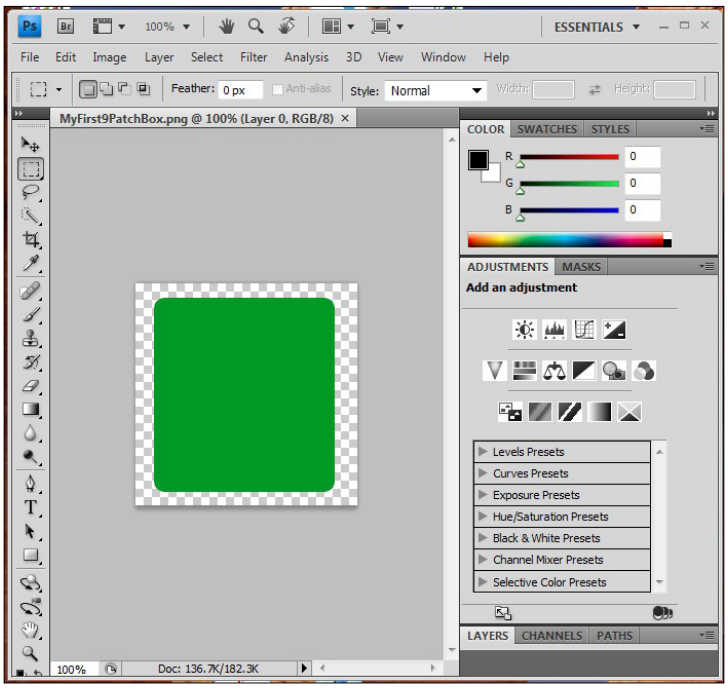
To create a NinePatch image, Android provides a very simple tool; however, before we move to that tool we need to know the requirements to use the tool. The first thing is that we need to have our base PNG graphic file that we will convert to our first NinePatch image. We can create our base PNG image in any graphics-editing tool such as Photoshop. So we will start creating our base PNG image by opening Photoshop and creating a new PNG image.

The following screenshot shows the **New** file-creation dialog in Photoshop:



We are planning to create a box with green color and we will convert the content inside that to a NinePatch image later. To start, we will name our file as `MyFirst9PatchBox` and will set **Background Contents** to **Transparent**. By clicking on **OK** we will get an empty canvas. Next we will draw a box with green color inside and with a transparent background.

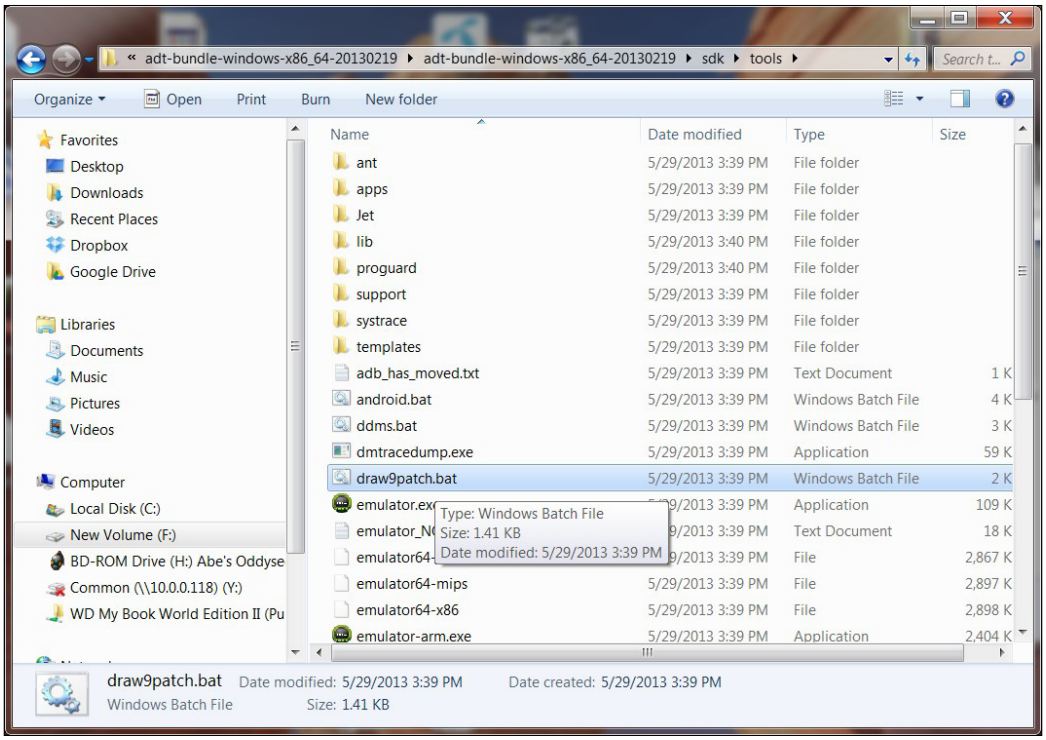
The following screenshot shows the green box we drew in Photoshop:



By default Photoshop saves the file in the PSD format but we will save the file in the PNG format because we will need a PNG file to convert it to NinePatch image. Alternatively, we can save the file for web; that way the PNG created will be lighter. This will increase the overall application performance. Using many heavy images in our application can reduce the performance of our application.

To create a NinePatch image from our green box PNG file, we will open the Draw 9-patch tool provided in the Android SDK in the `Tools` folder. To do this we will browse to `Tools` folder in Android SDK and will locate the `draw9patch.bat` file.

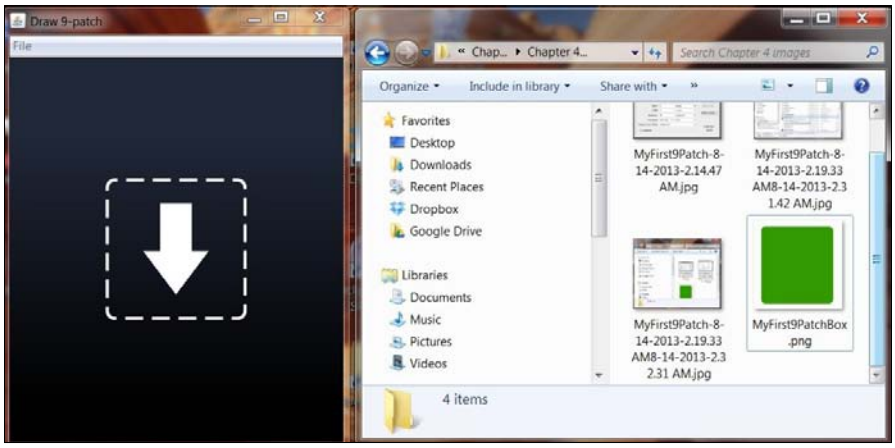
The path for the file would be `F:\Android\adt-bundle-windows-x86_64-20130219\adt-bundle-windows-x86_64-20130219\sdk\tools`, where `F` is my drive, Android is a folder at the root of `F` drive and the rest comes inside the Android folder. The following screenshot shows the location of the `draw9patch` tool.



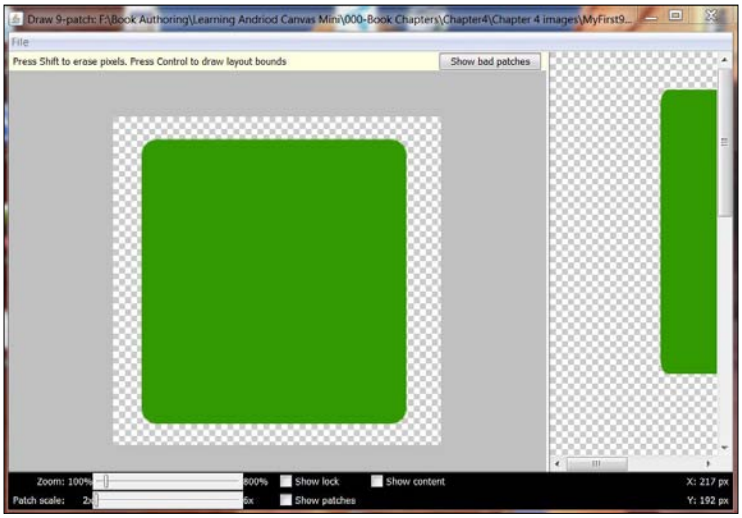
By double-clicking on that we can open the `draw9patch` tool. The following screenshot shows the `draw9patch` tool with an empty screen:

In the next step we will either drag-and-drop our PNG base image to the `draw9patch` tool or will open our PNG file in the `draw9patch` tool.

The following screenshot shows our base PNG file and the `draw9patch` tool side by side:



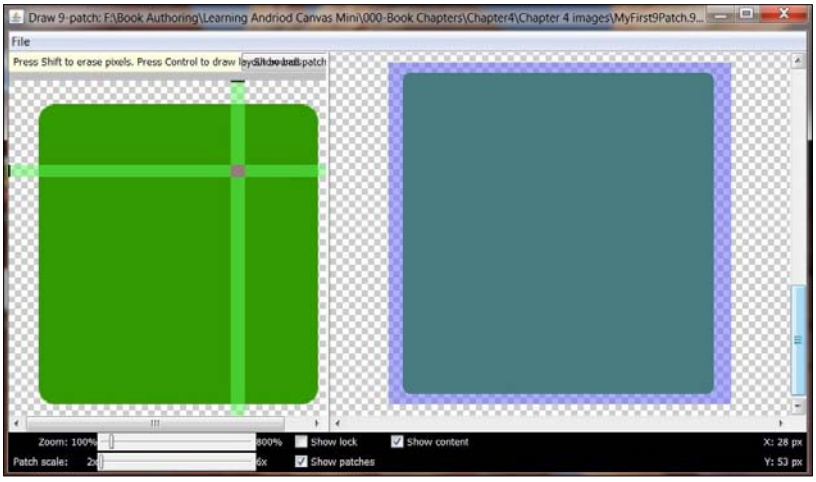
When we drag-and-drop or open our base PNG file in the `draw9patch` tool, the following screenshot is what we will see:



The right-hand side area shows the drawing area where we will define our patches—the areas that we want to stretch and the areas that we don't want to stretch. The left-hand side pane shows the preview area.

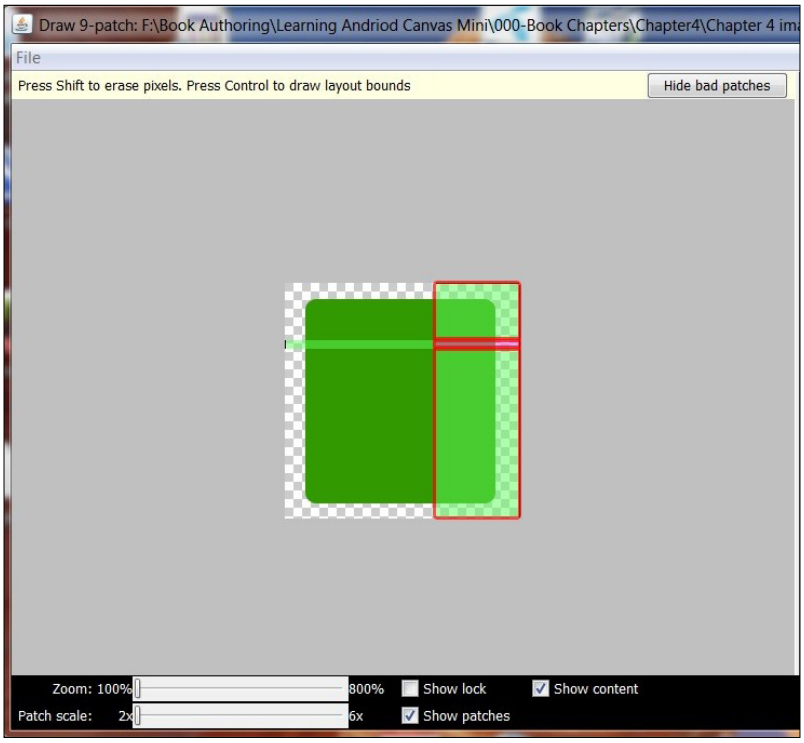
In the next step we will define our patches. When we move the cursor over the image, we will see very light horizontal and vertical lines. We will drag these horizontal and vertical lines to define our patches.

The following screenshot shows our defined patches:



The light green vertical and horizontal regions show us our defined patches and these are the stretchable sections. If we define our patches incorrectly, then when we put contents inside them, they won't stretch properly to capture all the content. This tool shows us whether we have bad patches. There is a button on top right corner in the left-hand side pane named **Show bad patches**. By clicking on it, it will show us the bad patches if we have bad patches in our 9-patch image.

The following screenshot shows the bad patches:

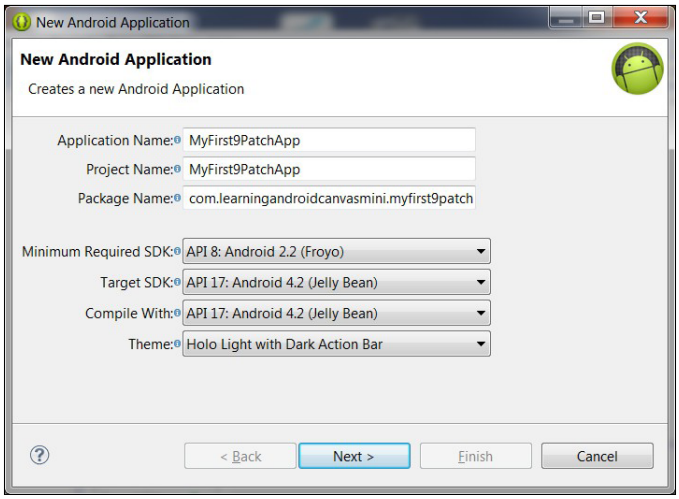


These patches marked with a red border are the bad patches that won't stretch properly to capture the entire contents inside it. I have done this just to make the idea clear what a bad patch will look like. We will adjust our horizontal and vertical lines to get proper patches, will check if there are no bad patches, and then we will save our file as `MyFirst9Patch.9.png`; then we are done with creating our first NinePatch image.

Using a NinePatch image

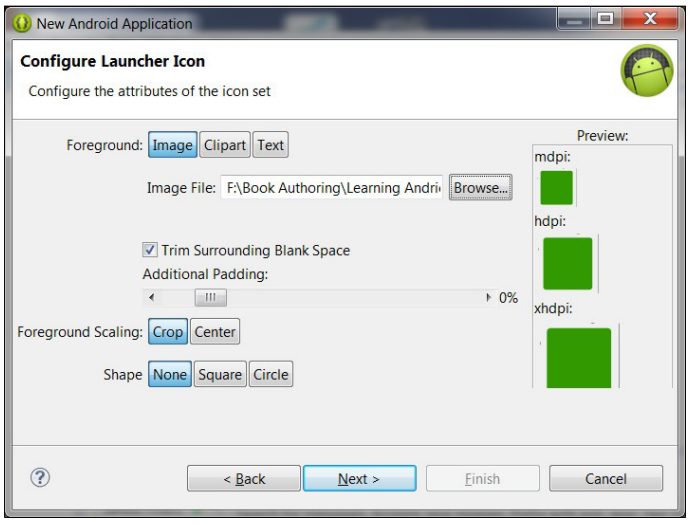
We will start by opening Eclipse and starting a new Android Project. We will name our project as `MyFirst9PatchApp`.

The following screenshot shows the new app's configuration settings:

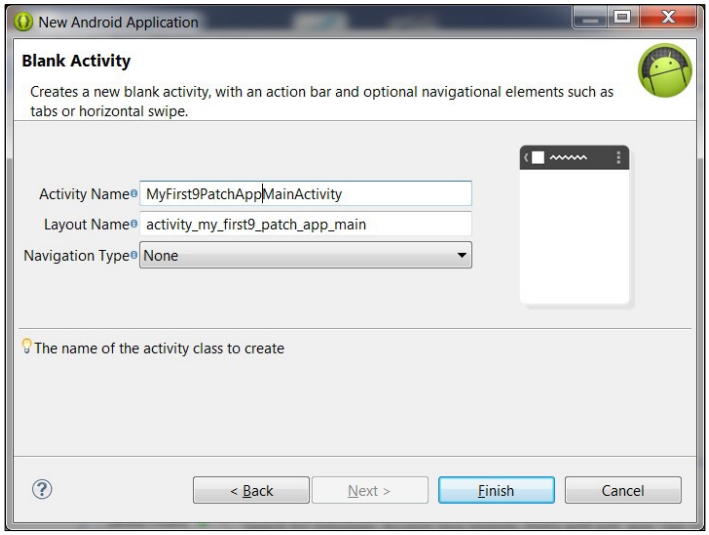


In the next step of the wizard we will provide our base PNG file as the icon for our application.

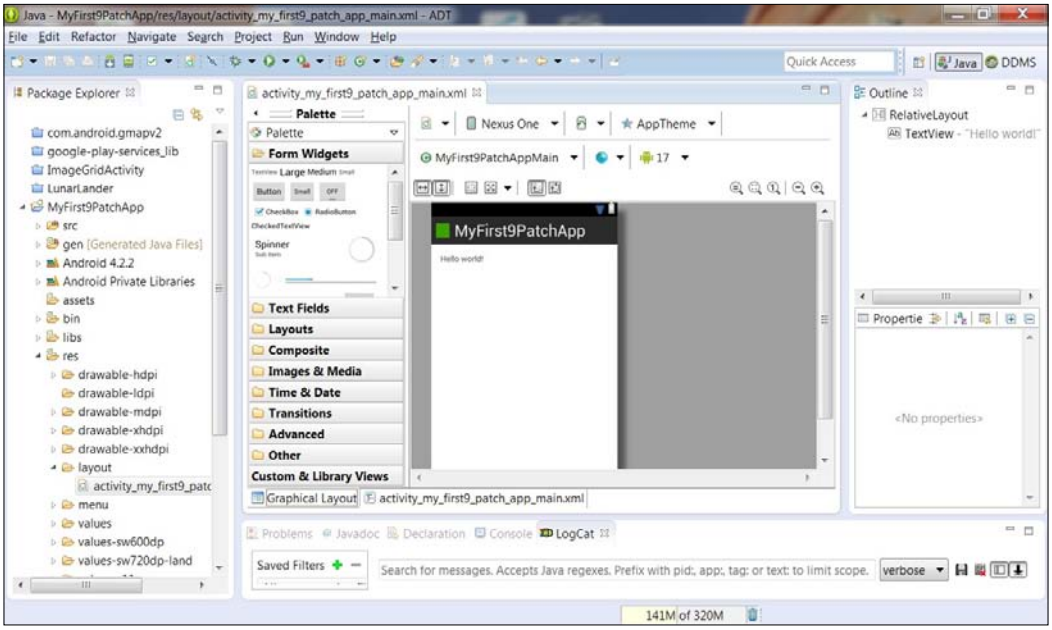
The following screenshot shows the icon configuration screen:



In the next step we will name our main activity as `MyFirst9PatchAppMainActivity`. The following screenshot shows the main activity configuration screen:



We will click on **Finish**, which completes the execution of the wizard, and will come to the **Design** view of our application. The following screenshot shows the **Design** view of our application:



The next thing we will do is to copy our NinePatch image in the `res/drawable` folder of our project, so that we can use that NinePatch image in our code.

Next we will open the `activity_my_first9_patch_app_main.xml` file of our main activity and will create a button in the code. The entire `Button` code looks like the following code snippet:

```
<Button
id="@+id/btnninepatch"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_alignParentTop="true"
  android:layout_centerHorizontal="true"
  android:layout_marginTop="26dp"
  android:background="@drawable/myfirstninepatch"
  android:text="@string/buttonwith9patchimage"
  android:textSize="38sp" />
```

The following two lines are the most important in the previous code:

```
android:background="@drawable/myfirstninepatch"
android:text="@string/buttonwith9patchimage"
```

The first line shows which Drawable tool the button will use as its background and the second line shows which text to display on the button as its content. Now in the preceding example, the string name is `buttonwith9patchimage` and its value is **Button With 9 Patch Image in the Background and we need to add some more text to make the button background extended more than 3 lines**. The need to add so much text to the button is to make it multiline; this will enable us to see the stretching effect of the NinePatch image.

The following screenshot shows the button with the stretched NinePatch image in the background:

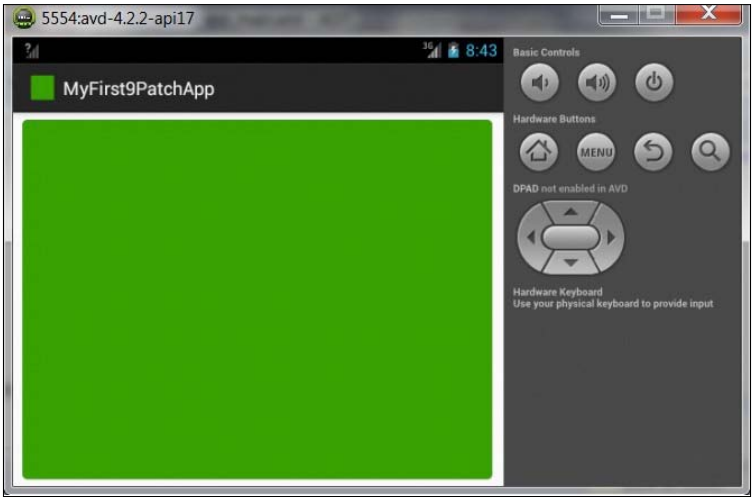
Up until now, it is clear that we will use the NinePatch images mostly for dealing with our background graphics; especially when we don't know the width and height of the content that we want it to contain. So the next thing that we will do is change the whole background of our application's main activity. Let's say we don't like the default white background color for our activity and we need a custom background. In the preceding application, we will delete the button code from our XML file and will add the following code to the `Layout` tag.

```
android:background="@drawable/myfirstninepatch"
```

The previous code will enable us to use a stretchable background for our whole activity. Complete code for this tag will look like the following code snippet:

```
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:background="@drawable/myfirstninepatch"
    tools:context=".MyFirst9PatchAppMainActivity" >

</RelativeLayout>
```

And when we test the application in the emulator, this is what we will see:



In the previous image we can see that our NinePatch box image has stretched vertically and horizontally to fill the entire activity screen. This will provide a custom stretchable background for all our graphics-rich applications.

Summary

In this chapter we have learned about images that can stretch to fill the container they are hosted in. These are known as NinePatch images. We learned about the importance of these stretchable images, their architecture, and the basic understanding of these images. We have also learned how we will convert simple images to these stretchable images and what are the common mistakes that we need to avoid. We have also learned how we can use these images in our Android applications in the background of views, controls, or in the background of the entire activity. In the next chapter we will learn about user interactions and graphics such as touch events.

5
Touch Events and Drawing on Canvas

In this chapter, we will learn about how we can interact with our application, capturing the touch events and rendering the graphics on the canvas in response to those touch events. To achieve this goal, we will develop a very simple Android application with an image displayed on the screen. The core functionality of the application will be that, when we click or touch the image and drag it along the screen, the image is dragged from its current location with our finger. The moment we release the touch and lift our finger from the screen, the image stops moving. In other words, we will touch the image and, by keeping it pressed, will drag-and-drop the image on the screen from point A to point B.

Some background knowledge

We first need to understand that the screen is filled with points called pixels. Horizontal points are `x` and vertical points are `y`. Whenever we put an object on the screen, it's at a certain `x, y` location. Let's say if the object is at the top-left corner of the screen, then its position will be `x = 0` and `y = 0`.

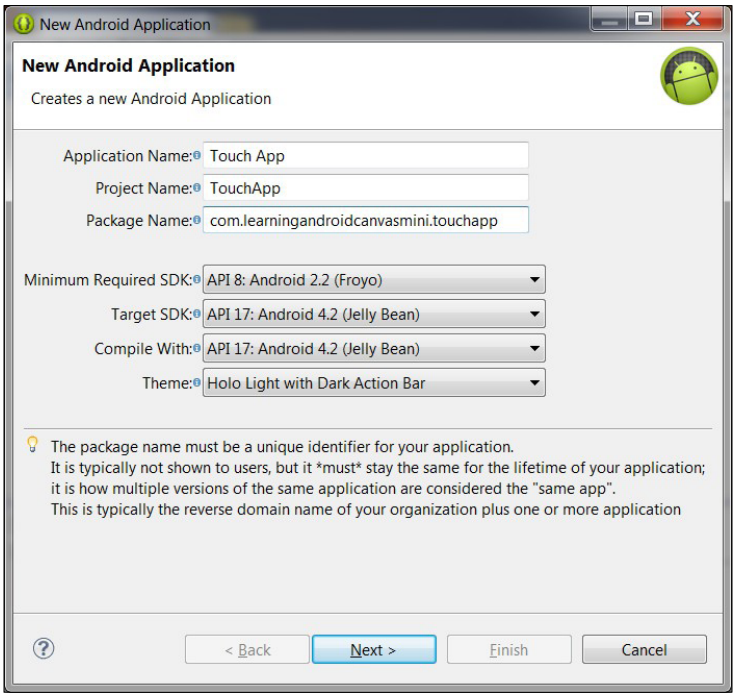
During our code, we will set a default location for our image when the application first runs. The default location will be at the top-left corner of the screen. So we will set `x = 0` and `y = 0` for our image location. The most important point here will be when we start dragging the image; at every change of `x, y` of our touch, we will update the image location to the current location of the touch. Thus, it will appear as if we are dragging the image from location A to B. We will do this by first listening to the touch events, then we will capture those touch events.

After that, we will capture actions of the touch. Is the action down or up? And if the action is down, is there any movement? Because if there is movement and the action is down, we have to do the dragging.

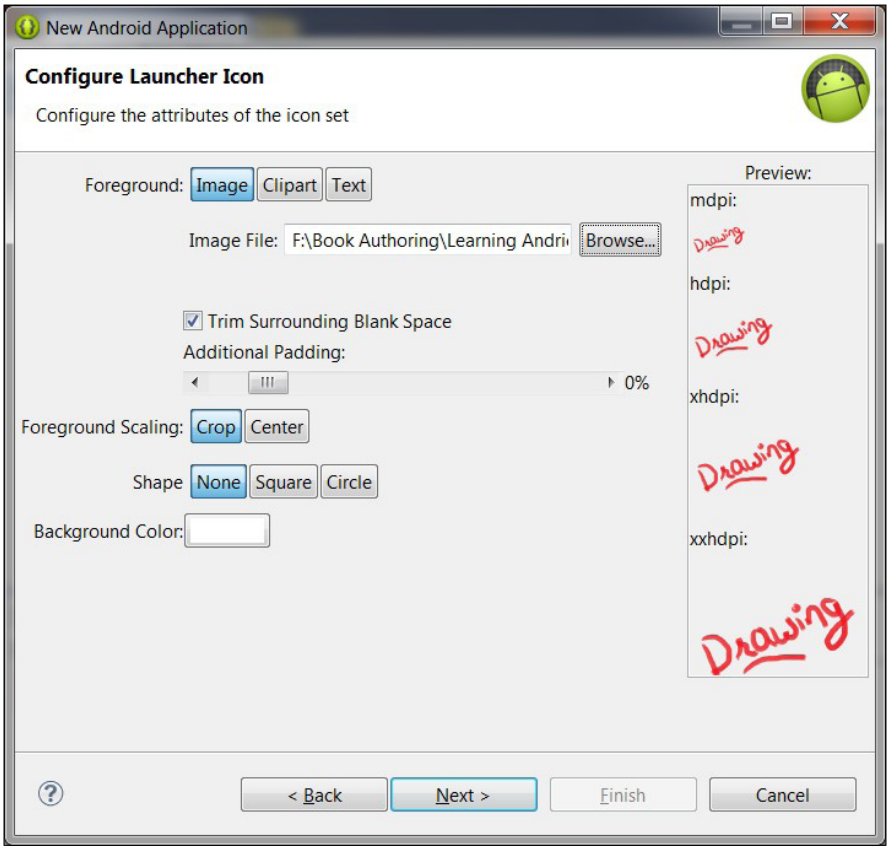
Some more details will come along as we develop our simple application and do the coding.

Application development

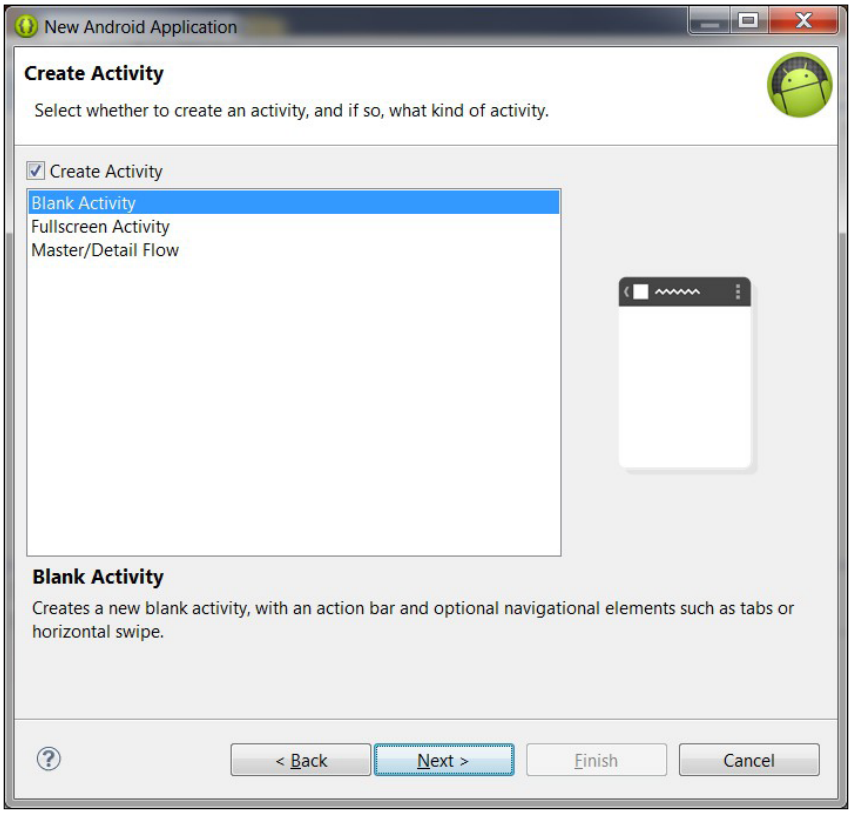
We will start by creating a new project in Eclipse by the name of `Touch App`. The following screenshot shows the first screen of the new Android application wizard:



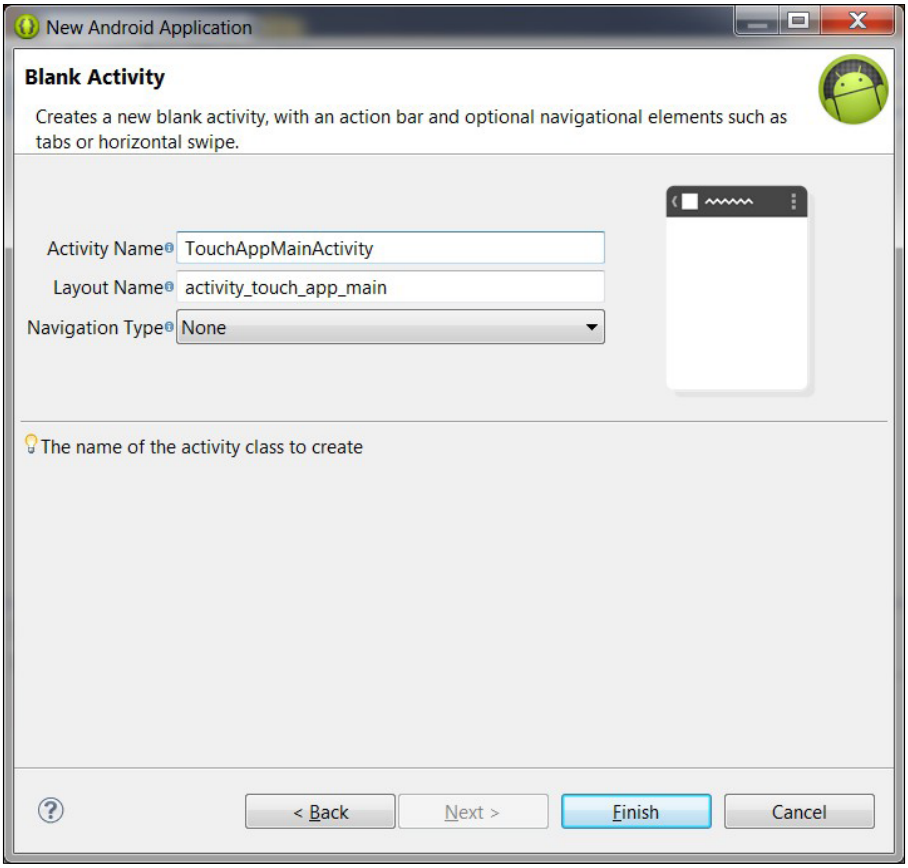
The following screenshot shows that we have selected our custom-created drawing PNG file as the icon for our application:



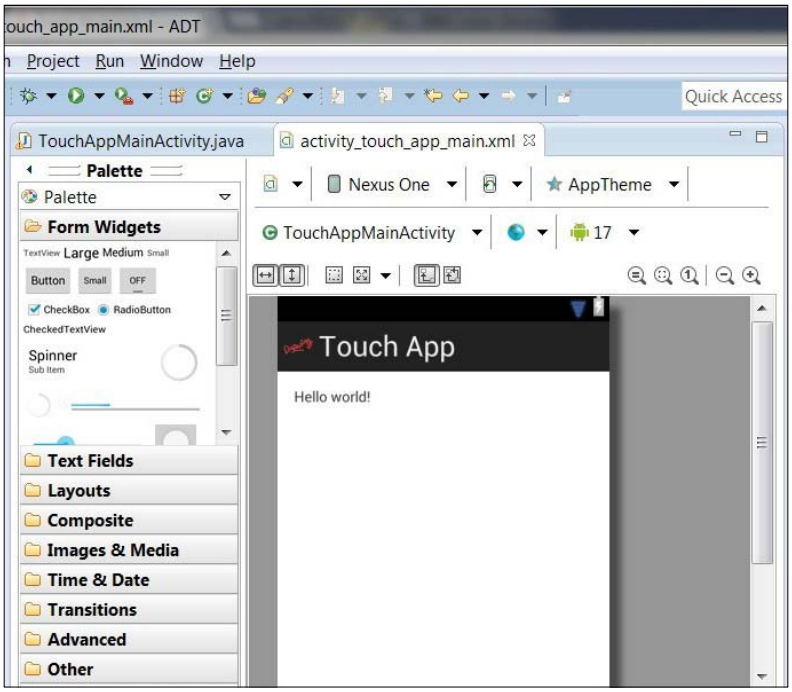
The following screenshot shows that we have to start our project with a blank screen as we want it to be our playground:



The following screenshot is the last screen in the wizard that shows we have created a default main activity by the name of `TouchAppMainActivity` and its layout filename is automatically populated:



The following screenshot shows that our wizard is finished and now we have a working skeleton application:



From this point onward, first what we will do is delete the **Hello world!** text from the screen as we want our screen to be completely blank and just display the image that we will drag.

Next, we will browse to the `res` folder of this project on our hard drive and will create a new folder named `drawable`, if it's not already created or any of the available folders in the `res` folder can be used, for example, `drawable-hdpi`.

Then, we will copy our `drawing.png` file to that folder and will come to Eclipse again. We will refresh the project files in the package explorer. The following screenshot shows the final state of the `res` folder:

Our custom-view class

We will open the `TouchAppMainActivity` Java file that extends the `Activity` class. Inside this class, we will create another class that will contain all our core functionality and will extend the `View` class:

```
public class TouchAppView extends View {
```

Defining class properties and objects

Inside this class, we will define some global objects and variables so that they are visible to all the methods in the class. First we will define a `Paint` object:

```
private Paint paint = new Paint(Paint.ANTI_ALIAS_FLAG);
```

Then variables for `x, y` points on the screen:

```
private float x, y;
```

Following is a Boolean variable to check if a touch event is occurring or not:

```
boolean touching = false;
```

After this, we will define a `Bitmap` object and provide our custom `drawing.png` file as its source that we have copied in the `drawable` folder in our project resources:

```
Bitmap drawingPic = BitmapFactory.decodeResource(
                    getResources(),R.drawable.drawing);
```

Next comes the variables that we will use to set the default position of our image on the screen. The initialized values will be `0, 0` for the `x, y` position and the image will be displayed at the top-left corner of the screen. Later, when we change the position of the image, the values of these variables will be updated accordingly.

```
int drawingPic_x = 0;
int drawingPic_y = 0;
```

Next we capture the width and height of the image that is displayed:

```
int drawingPic_w = drawingPic.getWidth();
int drawingPic_h = drawingPic.getHeight();
```

Then we will define the offset variables. The offset variables define the relative position of an object in reference to some other object or position.

```
int drawingPic_offsetx;
int drawingPic_offsety;
```

Following is just another Boolean variable to check the touch action. By default it's set to `false`.

```
boolean dm_touched = false;
```

The first Boolean variable checks if the screen is touched and the second Boolean variable checks whether, while the screen is touched, it is actually the image that is touched or whether it is some other point away from the image.

Methods in our custom-view class

Now we will add the following four methods to our class:

- The constructor
- The drawing method
- The measurement method
- The touch event

The constructor

We will define the constructor but the body will not put any functionality in there for the time being. For now, we will only call the context of the parent.

```
public TouchAppView(Context context) {

  super(context);

}
```

The drawing method

This is the method that will draw the bitmap image for us every time we change its location. As we want that, the old image is removed and the same image is drawn in the new location.

```
protected void onDraw(Canvas canvas) {

  canvas.drawBitmap(drawingPic,
                    drawingPic_x,
                    drawingPic_y,
                    paint);
}
```

This method actually draws the bitmap on the canvas by using `drawingPic` as a source image. `drawingPic` is the object that has our `drawing.png` image. It gets the `x` and `y` points from the initialized variables, where `x` is the position on the left-hand side of the bitmap to be drawn and `Y` is the position of the top of the bitmap to be drawn. At the end, the `paint` object will draw the bitmap but, if we are using a source bitmap, the value of this object can be null.

The measurement method

This is a method that will tell the system how much space is required vertically and horizontally by the view and its contents. We can complete this application without this method; however, mentioning it here is necessary as this method can provide us valuable information when working with more complex graphic application. The code is as follows:

```
protected void onMeasure(int widthMeasureSpec,
                         int heightMeasureSpec) {
  setMeasuredDimension(MeasureSpec.getSize(widthMeasureSpec),
                       MeasureSpec.getSize(heightMeasureSpec));
}
```

The touch event

We will define an `onTouchEvent()` method that will receive an action as an argument. Now actions can be of three types: a movement, a down motion or pressing a button type of action, and an up motion or releasing of a button type of action. We will define cases and functionality for the three cases and will integrate our logic of what we want to do in which type of action.

We define the `onTouchEvent()` method:

```
public boolean onTouchEvent(MotionEvent event) {
```

We define a variable that will store the action value:

```
  int action = event.getAction();
```

For all the available properties and methods, the following link can be visited:

`http://developer.Android.com/reference/android/view/MotionEvent.html`

A switch case start that will take the action and will check it with different scenarios and will perform actions according to each action is as follows:

```
  switch(action){
```

If the action is down, set the `x, y` values to the current events `x, y` positions and turn the touching variable to `true`.

```
case MotionEvent.ACTION_DOWN:
x = event.getX();
y = event.getY();
touching = true;
```

Now check if the action is down and if the image touched, as we don't want the image to be dragged by touching somewhere else on the screen. In the following code, we will check the vertical and horizontal positions of our image with the initial values and the width and height of our image with the initial values. If it's exactly the same and everything returns `true`, we will turn the `dm_touched` variable to `true`. This will mean that the image is touched.

```
if((x > drawingPic_x)
&& (x < drawingPic_x+drawingPic_w)
&& (y > drawingPic_y)
&& (y < drawingPic_y+drawingPic_h)){
```

Now that we know that the image is touched, we will update the `x, y` position of the image according to the new `x, y` position, that is, the position that we get from the event occurring in the current location.

```
  drawingPic_offsetx = (int)x - drawingPic_x;
  drawingPic_offsety = (int)y - drawingPic_y;
  dm_touched = true;
}

break;
```

After the preceding code is executed once, execute the case code mentioned in the second case. The `if` statement will execute because now we have confirmed that a touch event is performed and the touched area is actually our image. Now check whether the `dm_touched` variable is `true` and then update the `x, y` position of the image. At the start, the code in the first case will not be executed because the `dm_touched` variable is `false`.

```
if(dm_touched){
  drawingPic_x = (int)x - drawingPic_offsetx;
  drawingPic_y = (int)y - drawingPic_offsety;
}

break;
```

If the action is `MOVE`, then set the values for the earlier defined `x, y` variables to the current event `x, y` values and the touching Boolean variable to `true`.

```
        case MotionEvent.ACTION_MOVE:
        x = event.getX();
        y = event.getY();
        touching = true;
```

If the case is `ACTION_UP`, it means we are lifting our finger. Releasing the touch will simply make both `touching` and `dm_touched` Boolean variables `false`.

```
        case MotionEvent.ACTION_UP:
        default:
        dm_touched = false;
        touching = false;
      }
```

Lastly, we will call the `invalidate()` method so that the previous drawing is removed and a fresh new bitmap is drawn according to the new parameters.

```
      invalidate();
      return true;
    }
  }
```

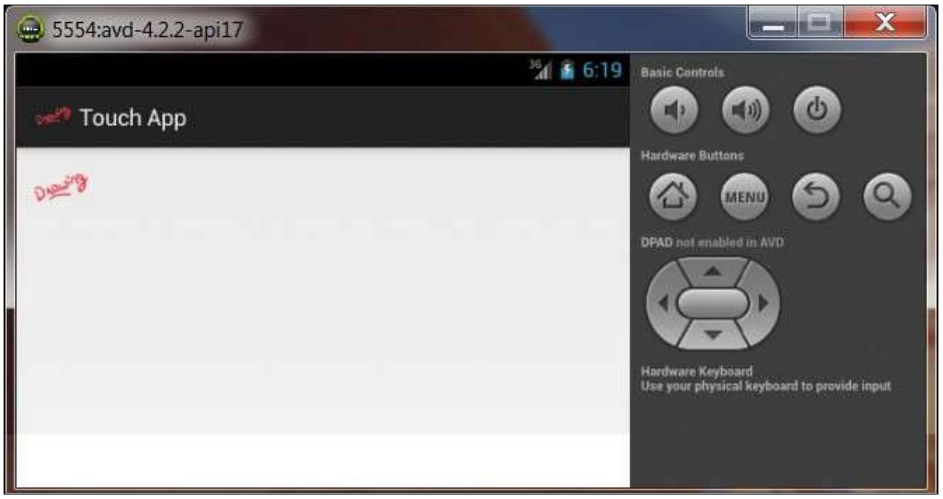
Our main activity class and final output

We will come to the `onCreate()` method of our main class, that is `TouchAppMainActivity`. In the `OnCreate` method, we will add the following code:

```
setContentView(new TouchAppView(this));
```

This method provides us with the facility to use our user interface for the activity on which we are working. If we define a custom view but don't set it to our custom-view class, our custom view will not appear. So everything will be alright and no errors will be generated, but our application will not function as planned. The importance of `setContentView()` is that it is the method responsible for displaying our XML-based layouts or even our dynamic layouts. By calling this method and supplying our custom-view class, that is, `TouchAppView` as an argument, we are enabling our application to execute whatever we have coded in the `TouchAppView` class.

The following screenshot shows the final output of the application when tested in an emulator:



Now we will click and keep the screen pressed as we try to drag the image with the mouse. This will imitate the touching of a screen; keep the finger pressed and drag the image on the screen from point A to B.

The following screenshot shows that we have dragged the image to another location from its default location:

Summary

In this chapter we have learned the following:

- Creating a custom-view class for our graphic application
- Capturing the on-touch event
- Identifying the action that is performed and the types of actions that can be performed while touching the screen
- Our functionality in response to the actions done by the touch event
- Lastly, how we can implement this custom-view class in our main activity class

In the next chapter, we will gather all the learned experience from the previous chapters and will create a complete application in which we will have different colors. We will select from them and will draw on the canvas our own drawing. It will be a very simple paint-like application.

6
Putting it All Together

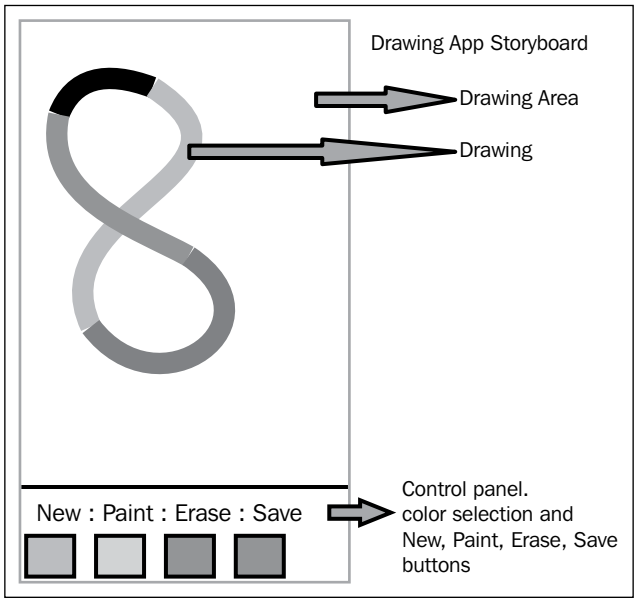
At last we have made it to the end of our process of gaining basic knowledge and skills to work with Android Canvas. In this chapter, we will develop a complete application step-by-step from scratch with all the functionality, using all the knowledge and techniques that we have used in the previous chapters. In each chapter, throughout this book, we have learned and practiced one core essential component of drawing with Canvas. Now in this chapter, we will bring all those components together with some additional features to create a fully functional application.

To create any application or game, we need to have a plan, plot, or story board for it. The story board will tell us how things will work and appear in our application, what will be the core functionality or output of our application, and what are the extra or supporting features that we need to make our application useful and complete. The plan could be some rough bulleted notes on a paper or it can be some hand-drawn sketches, but it will encapsulate all our needs and outputs that we want from it. Some notes attached to the drawing will complete our plan and the overall vision of our application. One good thing about planning is that it freezes the requirements and our application. In normal software engineering, before starting the development of an application, a **Software Requirements Specification** document is prepared in that all the requirements are written and signed by both parties. It defines the boundaries of the project and this is very important. Without defining the boundaries or limits of requirements, the development of an application will never end because, with time, requirements increase during development. So we must have a defined set of requirements and functions. Later on, we can add more functionality in the new versions of the application, but at least the core requirements and functionality will be complete in the base application.

The story board

We will start with the wireframe of our application with some notes on it and some pointers and sections; then we will explain the sketch.

The following diagram shows the story board of our application that we will develop:



The previous diagram is quite self-explanatory and gives us a complete sketch of what we are planning to do. The top-most area shows the application title and will also display a small icon if we have used one while creating the application. Below the title bar is the major empty **Drawing Area,** where we will perform all our free-hand drawing and painting. At the bottom of the screen, we will have a **Control panel**, from where we can select the function as well as the drawing and painting style. The **Control panel** will help us in selecting the colors for our brush. We can change the size of the brush. We can perform certain functions such as create a new drawing, save the existing drawing, select a brush size, click on the eraser to select it, and erase the currently drawn painting. We will use our fingers and touch events to draw on the Canvas. Later, we can either **Save**, **Erase**, or change the color and brush size, and proceed with our painting. So our goal is a simple Paint-like application for Android.

The project and application development

We will achieve our goal by dividing the project into four stages. During the development process, we will go through each stage step-by-step and at the end, we will have our working paint application. Following are the stages:

- The user interface
- Enabling the touch and painting
- Enabling the color selection
- Spicing up our application with more functions

The user interface

We will start our project by creating a new Android application in Eclipse through the wizard, as we have practiced earlier in this book. Only the first screen of the new Android application wizard is attached.

The following screenshot shows the first screen of the new Android application wizard:

We will name our application as `OurFirstPaintApp`, the **Project Name** will be populated automatically, and we will change the **Package Name** to `com.learningandroidcanvasmini`. During the development of this application, we will be working with XML files in the `res` folder and Java files in the `src` folder of our project.

Screen orientation

Before we start anything, we need to decide what orientation our project will support. Let's say we want our project application to always be in **Portrait** form even if the user is holding the device in landscape mode. To do this, we will open the `AndroidManifest.xml` file and will change `android:screenOrientation="landscape"` in the `Activity` tag to `android:screenOrientation="portrait"`.

The paint brush

From our story board, we know that we have to use different sizes of brushes; for this, we will define some numbers that will refer to certain brush sizes. We will go to the `res/values` folder and will open the `dimens.xml` file. If the file is not there, we will create a new XML file with the name `dimens.xml` and will put our values in it. In this file, we will look for the `<resources></resources>` tag. Inside this tag, we will put our values as follows:

```
<!-- Available brushes -->
<dimen name="small_brush">10dp</dimen>
<integer name="small_size">10</integer>
<dimen name="medium_brush">20dp</dimen>
<integer name="medium_size">20</integer>
<dimen name="large_brush">75dp</dimen>
<integer name="large_size">75</integer>
```

We will keep the dimension and the integer value the same so that the user interface can show the exact brush size when we use it for drawing.

Designing the Control panel

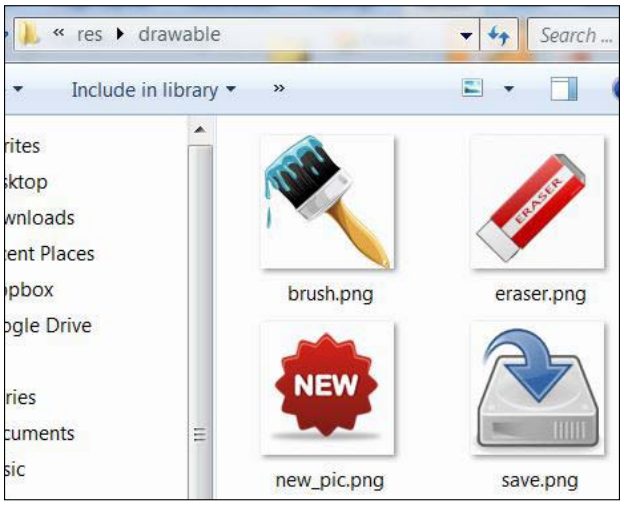
In the Control panel that we have in our story board sketch, we have a row of colors and another row with some button that performs certain functions. Now we will start designing the **Control panel**. To begin with, we will add all the strings that we will use in the **Control panel**. To do this, we will open the `string.xml` file in the `res/values` folder and will add the following code to the file:

```
<string name="start_new_painting">New Painting</string>
<string name="brush_size">Brush Size</string>
```

```
<string name="erase">Erase</string>
<string name="save_painting">Save Painting</string>
<string name="paint">Paint</string>
```

Setting up the layout

First, we will either create or download the following images from the Web and copy them in the `res/drawable` folder. The following screenshot shows the images in the `res/drawable` folder:



Now, we will set up the **Layout** for our Canvas and **Control panel**, the drawing space, and the space that will host our buttons. We will open the `activity_main.xml` file. Inside the main `Layout` tag, we will enter three sub layout tags as mentioned in the following code snippet. The first one will hold the image for the new, brush, erase, and save buttons and the next two will each hold a row of colors.

```
<LinearLayout
  android:layout_width="wrap_content"
  android:layout_height="100dp"
  android:layout_gravity="center"
  android:orientation="horizontal">

  <!-- content code starts here -->
  <ImageButton
    android:id="@+id/new_btn"
    android:layout_width="wrap_content"
```

```
        android:layout_height="fill_parent"
        android:contentDescription="@string/start_new"
        android:src="@drawable/new_pic" />

    <ImageButton
        android:id="@+id/draw_btn"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:contentDescription="@string/brush"
        android:src="@drawable/brush" />

    <ImageButton
        android:id="@+id/erase_btn"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:contentDescription="@string/erase"
        android:src="@drawable/eraser" />

    <ImageButton
        android:id="@+id/save_btn"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:contentDescription="@string/save"
        android:src="@drawable/save" />
    <!-- content code ends here -->

</LinearLayout>
```

Next we will add two more layouts in which we will have two rows of colored buttons as shown:

```
<LinearLayout
    android:id="@+id/paint_colors"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >

    <ImageButton
        android:layout_width="@dimen/large_brush"
```

```
        android:layout_height="@dimen/large_brush"
        android:layout_margin="2dp"
        android:background="#FF660000"
        android:contentDescription="@string/paint"
        android:onClick="paintClicked"
        android:src="@drawable/paint"
        android:tag="#FF660000" />
    .
    .
    .
```

The dots in the preceding code represent the code for the rest of the color buttons that we would like to add. The only thing that needs to be changed in the preceding block of code is the `android:tag="#FF660000"` code, which is the value of the color. Now we have the layout ready to for our Control panel. We need to fix some space in the layout for our drawing. Here, we will create a new Java class in the `src` folder by the name of `CustomDrawingView` that will extend View. For the time being, we will only create the skeleton code of the class so that we can refer to it in our layout XML file. Later, we will customize every section of our `CustomDrawingView` class and will put in all the functionality that we want our application to have. The skeleton code will be as follows:

```
import android.content.Context;
import android.util.AttributeSet;

public class CustomDrawingView extends View
{
  public CustomDrawingView(Context context, AttributeSet attrs){
    super(context, attrs);
    setupDrawing();
}

private void setupDrawing(){
  //get drawing area setup for interaction
}

}
```

So we have added a `CustomDrawingView` class with a constructor and a `setupDrawing()` method. These are the most essential components right now. Without the constructor, the program will generate an error. Now we will come back to our `activity_main.xml` file and will add the following code just below the opening tag of the parent `layout` tag:

```
<com.learningandroidcanvasmini.ourfirstpaintapp.CustomDrawingView
  android:id="@+id/drawing"
  android:layout_width="fill_parent"
  android:layout_height="0dp"
  android:layout_marginBottom="3dp"
  android:layout_marginLeft="5dp"
  android:layout_marginRight="5dp"
  android:layout_marginTop="3dp"
  android:layout_weight="1"
  android:background="#FFFFFFFF" />
```

The line of code, `<com.learningandroidcanvasmini.ourfirstpaintapp.CustomDrawingView`, in our XML file shows the package directory of our custom view. Second, this also shows us how we can add customized code-based layouts in our XML layout files.

Although we now have our layout almost ready, it is still not complete. We need color selection buttons that are proper rectangles in our **Control panel**. So far, we have defined the image buttons, but some configuration is still needed such as the shape of those buttons. We want the same square-shaped buttons with slightly rounded corners. For this purpose, we will create a new `paint.xml` file in the `res/drawables` folder. In this file, we will use the technique explained in Drawables from Resource XML. We will create a two-layered Drawable shape: one layer for the rectangle object and the other for rounding its corners.

```
<layer-list
xmlns:android="http://schemas.android.com/apk/res/android" >
<item>
  <shape android:shape="rectangle" >
    <stroke
      android:width="5dp"
      android:color="#FF999999" />
    <solid android:color="#ff000000" />
    <padding
      android:bottom="0dp"
      android:left="0dp"
      android:right="0dp"
```

```
        android:top="0dp" />
    </shape>
</item>
<item>
    <shape
xmlns:android="http://schemas.android.com/apk/res/android" >
      <stroke  android:width="4dp"  android:color="#FF999999" />
      <solid android:color="#ff000000" />
      <corners android:radius="15dp" />
    </shape>
</item>
</layer-list>
```

In the preceding code, we have used a layer-list tag in which we have used two items. The first item is for the rectangle shape and the second is for rounding its corner. We will save all files and will run our code in the emulator.

The following screenshot shows our application without any functionality:

So far, we have completed the user interface and are done with the graphics work. Now we have to make our application touch enabled.

Enabling the touch and painting using touch

The target is to enable the touch: in other words, our application will paint as we drag our finger on the screen. We will start with opening our `CustomDrawingView` class and will add the following objects and variables:

```
private Path drawPath;
private Paint drawPaint, canvasPaint;
private int paintColor = 0xFF660000;
private Canvas drawCanvas;
private Bitmap canvasBitmap;
```

After this in the `setupDrawing()` method, we will instantiate the `Path` and `Paint` objects and will set its different attributes to certain values that we want it to get as default values:

```
drawPath = new Path();
drawPaint = new Paint();
drawPaint.setColor(paintColor);
drawPaint.setAntiAlias(true);
drawPaint.setStrokeWidth(20);
drawPaint.setStyle(Paint.Style.STROKE);
drawPaint.setStrokeJoin(Paint.Join.ROUND);
drawPaint.setStrokeCap(Paint.Cap.ROUND);
```

Lastly, in the method we will instantiate the canvasPaint object:

```
canvasPaint = new Paint(Paint.DITHER_FLAG);
```

Now we will jump to the `onDraw()` method and will add the following lines of code:

```
canvas.drawBitmap(canvasBitmap, 0, 0, canvasPaint);
canvas.drawPath(drawPath, drawPaint);
```

The `onDraw()` method receives an object of type `Canvas` as an argument and is used to draw the bitmap and the drawing or painting that we want. Each time we touch the screen and drag our finger, the `invalidate()` method will be called on the previous painting on the Canvas and the `onDraw()` method will be called automatically, displaying our current painting.

We have almost everything in place but we have still not made our application touch enabled. For this, we will add an `onTouchEvent()` method to our `CustomDrawingView` class, as shown:

```
public boolean onTouchEvent(MotionEvent event) {
}
```

Inside this method, we have to do two jobs: detect the user's touch and check the motion direction. The touch can be either up, meaning the finger is not touching the screen, or down, meaning that the finger is pressed on the screen. Motion can be a pressed finger moving in any direction. First, we will get the x, y location of the touch using the following code:

```
float touchX = event.getX();
float touchY = event.getY();
```

Then we will respond to that touch event according to one of the possible cases, which we will include in the following switch code:

```
switch (event.getAction()) {
```

With the down action event, the finger is pressed. Move the drawing point to the location touched:

```
case MotionEvent.ACTION_DOWN:
  drawPath.moveTo(touchX, touchY);
  break;
```

With the motion event, the finger is pressed and dragged in some direction. First, the point of drawing will be the point that is touched and then a line will be drawn along the motion of the finger during the dragging action:

```
case MotionEvent.ACTION_MOVE:
  drawPath.lineTo(touchX, touchY);
  break;
```

Lastly, when the finger is lifted, a path on the Canvas will be drawn and the `Path` object will be refreshed so that it's ready to draw a new line from the next location touched:

```
case MotionEvent.ACTION_UP:
  drawPath.lineTo(touchX, touchY);
  drawCanvas.drawPath(drawPath, drawPaint);
  drawPath.reset();
  break;
  default:
  return false;
}
```

At the end, we will call the `invalidate()` method so that we can activate the `onDraw()` method:

```
invalidate();
return true;
```

The whole code of the class is as follows:

```
public class CustomDrawingView extends View {
  private Path drawPath;
  private Paint drawPaint, canvasPaint;
  private int paintColor = 0xFF660000;
  private Canvas drawCanvas;
  private Bitmap canvasBitmap;
  public CustomDrawingView(Context context, AttributeSet attrs){
    super(context, attrs);
    setupDrawing();
  }
  private void setupDrawing(){
    drawPath = new Path();
    drawPaint = new Paint();
    drawPaint.setColor(paintColor);
    drawPaint.setAntiAlias(true);
    drawPaint.setStrokeWidth(20);
    drawPaint.setStyle(Paint.Style.STROKE);
    drawPaint.setStrokeJoin(Paint.Join.ROUND);
    drawPaint.setStrokeCap(Paint.Cap.ROUND);
    canvasPaint = new Paint(Paint.DITHER_FLAG);
  }
  @Override
  protected void onSizeChanged(int w, int h, int oldw, int oldh) {
    super.onSizeChanged(w, h, oldw, oldh);
    canvasBitmap = Bitmap.createBitmap(w, h,
      Bitmap.Config.ARGB_8888);
    drawCanvas = new Canvas(canvasBitmap);
  }
  @Override
  protected void onDraw(Canvas canvas) {
    canvas.drawBitmap(canvasBitmap, 0, 0, canvasPaint);
    canvas.drawPath(drawPath, drawPaint);
  }
  @Override
  public boolean onTouchEvent(MotionEvent event) {
    float touchX = event.getX();
    float touchY = event.getY();
    switch (event.getAction()) {
    case MotionEvent.ACTION_DOWN:
      drawPath.moveTo(touchX, touchY);
      break;
```

```
        case MotionEvent.ACTION_MOVE:
          drawPath.lineTo(touchX, touchY);
          break;
        case MotionEvent.ACTION_UP:
          drawPath.lineTo(touchX, touchY);
          drawCanvas.drawPath(drawPath, drawPaint);
          drawPath.reset();
          break;
        default:
          return false;
        }
        invalidate();
        return true;
      }
    }
```

Enabling the color selection

We will enable the color selection from the user interface that we created earlier. We know that we have two rows of different colors. Now we need our application to be able to set the color of our paint brush to the color that we select from the available color palette. To do this, we will open our main activity class and will add `import` statements for `View`, `ImageButton`, and `LinearLayout` to our import section of the main activity class:

```
import android.view.View;
import android.widget.ImageButton;
import android.widget.LinearLayout;
```

Next, inside our main activity class, we will create an object of our `CustomDrawingView` class:

```
private CustomDrawingView drawView;
```

In the `onCreate()` method of our main activity class, we will instantiate the `drawView` object by getting a reference to its layout as defined in the `activity_main.xml` file:

```
drawView = (CustomDrawingView)findViewById(R.id.drawing);
```

We will add another object that will let us know which paint button is clicked:

```
private ImageButton imgBtnSelectedPaint;
```

Next in the `onCreate()` method, we will add the following code. First we will get the layout that hosts our paint buttons. The `paint_colors` property is defined in the `activity_main.xml` file:

```
LinearLayout paintLayout
  =(LinearLayout)findViewById(R.id.paint_colors);
```

Then, we will select the first color in the top-color row as the default selected color:

```
imgBtnSelectedPaint = (ImageButton)paintLayout.getChildAt(0);
```

To differentiate between the select color button and others, we will use a different Drawable for the selected one. To have a separate `Drawable` object for the selected button, we will first define a separate XML file in our `drawable` folder. The code of the selected button in the XML file is listed as follows:

```
<layer-list
  xmlns:android="http://schemas.android.com/apk/res/android" >
    <item>
      <shape android:shape="rectangle" >
        <stroke
          android:width="4dp"
          android:color="#FF333333" />
          <solid android:color="#00000000" />
          <padding
            android:bottom="0dp"
            android:left="0dp"
            android:right="0dp"
            android:top="0dp" />
        </shape>
    </item>
    <item>
      <shape
xmlns:android="http://schemas.android.com/apk/res/android" >
        <stroke
          android:width="4dp"
          android:color="#FF333333" />
          <solid android:color="#00000000" />
          <corners android:radius="10dp" />
        </shape>
    </item>

</layer-list>
```

The code is exactly the same as the previous XML file that we created in the `drawable` folder for our normal paint buttons. The only difference here would be a different value for `android:color` and solid `android:color`. After this, we will open our main activity class and, in the `onCreate()` method, which is below the image button code that we have recently added, we will add the following line to enable a different style for the select button:

```
imgBtnSelectedPaint.setImageDrawable(getResources().getDrawable(R.
drawable.paint_pressed));
```

Next we will create a method in the `CustomDrawingView` class to update the paint color used when we draw our painting:

```
public void setColor(String newColor){
  invalidate();
  paintColor = Color.parseColor(newColor);
  drawPaint.setColor(paintColor);
}
```

We will use the preceding method in our main activity class shortly. Note that, in the code of our image buttons in the XML file, we have mentioned an `onClick()` method with the method name `paintClicked`. Now is the time to create the `paintClicked` method in our main activity class. We will start by defining the method as follows:

```
public void paintClicked(View view){
```

Now we will check whether the clicked color button is not already selected:

```
if(view!=imgBtnSelectedPaint){
```

If not selected, we will get the tag from the button clicked and will set the current color for our painting to the selected tag color:

```
ImageButton imgView = (ImageButton)view;
String color = view.getTag().toString();
```

The following line of code will call the `setColor()` method created earlier in the `CustomDrawingView` class to change the paint color:

```
drawView.setColor(color);
```

After this, we will simply update the **Control panel** to display that the selected button is modified by changing the Drawable of the selected button; we will revert the previously selected button to its normal state:

```
imgView.setImageDrawable(getResources().getDrawable(R.drawable.pa int_
pressed));
imgBtnSelectedPaint.setImageDrawable(getResources().getDrawable(R.
drawable.paint));
imgBtnSelectedPaint=(ImageButton)view;
}}
```

Our progress status shows that we have covered three quarters of our project:

- We have a relatively attractive user interface for our application in place
- Our application is touch enabled
- We are able to draw using finger touch and drag actions; moreover, we can change the color or the paint brush with which we are drawing

The following screenshot shows our current state with a rough drawing:

I have tried to draw a brush that appears on the brush button, as shown in the preceding screenshot. I am not a good artist and it's not even a good drawing, but it's good enough to explain where we stand now.

Spicing up our application with more functions

The core structure of our application is already complete, but we should still make it more useful and attractive; we need to make it more spicy with more functions, rather than just providing a selection of colors and painting. We will start with the functionality of saving our paintings to the device.

Saving the painting

We will create an instance of `savePaintingButton` in our main activity class:

```
private ImageButton savePaintingButton;
```

Next, we will create its instances and will listen to its clicks in the `onCreate()` method in our main activity class:

```
savePaintingButton = (ImageButton)findViewById(R.id.save_btn);
savePaintingButton.setOnClickListener(this);
```

When we write the preceding line of code for the click listener, a red line will appear under it, warning us that something is wrong. Right-clicking on this line will give us all the possible options to fix the problem. Select the option that best solves the problem. Let's say that, in the example, we select the option of the main activity implementing the inherited `OnClickListner`; two things will happen. First our main activity class will inherit `OnClickListner` as shown in the line of code:

```
public class MainActivity extends Activity implements
OnClickListener {
```

And second, the following method will be created inside our main activity class with an empty body.

```
public void onClick(View v) {
}
```

Now we will write our saving logic here inside the preceding method. To save our painting we will start with a reference to the layout that contains our painting and will enable the drawing cache for it. Then we will create a `Bitmap` object and will provide a drawing cache with our view objects.

```
View ourView = findViewById(R.id.drawing);
ourView.setDrawingCacheEnabled(true);
Bitmap ourBitmapImage = ourView.getDrawingCache();
```

Next, we will define a string that will contain the path where we will save our file and will create a `File` object that will be provided with that path:

```
String extr = Environment.getExternalStorageDirectory().toString()
  + "/SaveCapture";
File myPath = new File(extr);
```

If the path doesn't exist, we will create a new directory at that location and, if the path exists, we will create our file at that location, as follows:

```
if (!myPath.exists()) {
  boolean result = myPath.mkdir();
}
  myPath = new File(extr, getString(R.string.app_name) + ".jpg");
```

After this we will create a `FileOutputStream` object. An output stream writes bytes to a file if it exists; if not, a new file is created and the stream is written to it. The `Bitmap` object that we have created earlier that has our drawing cache, we will use it to call a method `compress()` providing it with the information of compression type and the `FileOutputStream` object to create an image file for us with a `JPEG` extension. To track it back, that is, to see how the painting information got here in this JPEG file, the `Bitmap` object is provided with the `FileOutputStream` object that contains the `myPath` object. The `myPath` object contains the complete path where the file has to be saved and the name with which the file will be saved. In addition, the `Bitmap` object has already got the necessary information of our painting from the `View` object that we created and what's available in the drawing cache at the specific moment. The block of code explains the creation of the output steam object and supplies it to our already created `Bitmap` object. The last line in the `try` block uses `MediaStore`. `MediaStore` contains metadata for all available media on external and internal storage devices. The `insertImage` from `MediaStore` will insert an image and will create a thumbnail for it in our gallery:

```
FileOutputStream fos = null;
  try {
    fos = new FileOutputStream(myPath);
    ourBitmapImage.compress(Bitmap.CompressFormat.JPEG, 100, fos);
    fos.flush();
    fos.close();
    MediaStore.Images.Media.insertImage(getContentResolver(),
    ourBitmapImage,
      "Screen", "screen");
  }

  catch (FileNotFoundException e) {
```

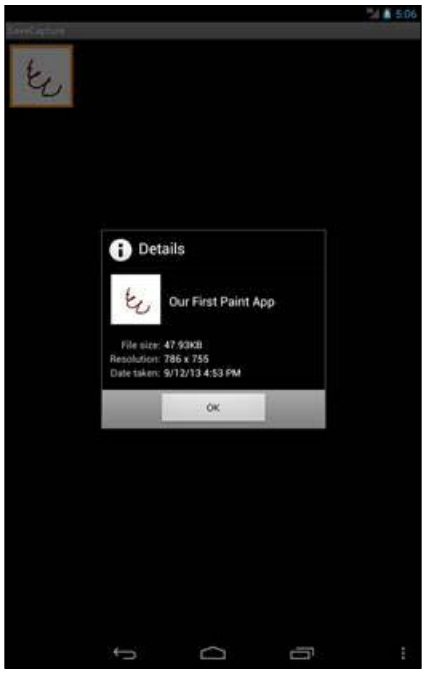
```
      Log.e("Error", e + "");
    }

    catch (Exception e) {
      Log.e("Error", e + "");
    }
```

Lastly, we will add the following permission to our `AndroidManifest.xml` file:

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

Without these permissions, our application will not be allowed to read or write from the storage drives. For more details on permissions and its understanding, visit the link `http://developer.android.com/guide/topics/security/permissions.html`.

With the end of this block of code, our saving functionality completes. Now we can draw and save our paintings in our gallery. The following screenshot shows us the saved paintings:



If we check the gallery, we will find our image saved there if we check the details of the newly created image, it will show us a thumbnail of the image, its name, and the date created. Our image is saved with our application name because that's what we provided as the filename in our code. We must also make sure that our emulator is created with SD card options. If our emulator is created without SD card options, this code will not work.

Creating a new drawing

To create a new painting or enabling the functionality of the new button, we need to clear whatever is currently drawn on the Canvas and clear the previous drawing cache. To do this, we will create an instance of the new button, a listener, and an `onCreate()` method just as we did for the `savePaintingButton` button. We will add the following code in the listener of the new button:

```
drawCanvas.drawColor(0, PorterDuff.Mode.CLEAR);
invalidate();
```

The preceding code will clear the screen and cache. I won't go into the details of `PorterDuff` but will only write here that it's not Android-specific. `PorterDuff` is the `AlphaComposite` class implementing the 12 rules defined by Porter and Duff. More details in reference to Android can be found in the following link:

`http://developer.android.com/reference/android/graphics/PorterDuff.html`

Enabling eraser in our application

We will add one more functionality to our application in this chapter: using the eraser. I know I am not a Picasso and I'll definitely need an eraser in my painting application. So we will write some more code to make our application eraser enabled. First, we will create an instance of the eraser button and a listener for it, as we did for the `savePaintingButton` button. The simplest way of making the eraser functionality is to set the paint color to that of the background color. It will appear that we are erasing the drawing but actually, we will be painting it with the paint color set as the background color.

This is the code of our paint selection:

```
if(view!=imgBtnSelectedPaint){
  ImageButton imgView = (ImageButton)view;
  String color = view.getTag().toString();
  drawView.setColor(color);
  imgView.setImageDrawable(getResources().getDrawable(R.drawable.
paint_pressed));
```

```
    imgBtnSelectedPaint.setImageDrawable(getResources().getDrawable(R.
drawable.paint));
    imgBtnSelectedPaint=(ImageButton)view;
```

We know that our background color is white and, from our XML layout file, we know that the tag we have defined for the color white is `#FFFFFFFF` and the background color for our white button is `#FFFFFFFF`. So, to enable erasing in our application, we will simply change the values of `String color = view.getTag().toString();` to `String color = "#FFFFFFFF";`.

Save the file and run the application. Now our application will have the erasing functionality. The following screenshot shows the eraser used in our application on our previous paint brush drawing:

With the eraser functionality in place, our basic paint application is complete but, in terms of extra spice, the extra functionalities are not limited to new, paint, erase, and save. We can put in a lot of other functionalities, as far as our imagination can take us. But one thing is clear that: whatever we plan to put in there, we will be using one of the things that are explained in the course of this book. The approach and the order in which things will be coded might be different but, as far as the basic 2D graphics are concerned, the lessons and examples in this book will cover it.

Summary

In this chapter, we have learned more than we actually promised at the beginning. What we have learned is listed as follows:

- Planning our application and the use of story boards
- Creating a rich, complex user interface with nested layouts
- Creation of graphic objects using just XML
- Creation of our own separate `View` class and referencing to that class in XML files and our main activity file code
- Use of the `Path`, `Paint`, `Bitmap`, and `Canvas` objects
- Defining listeners and capturing touch events
- Responding to those touch events
- Switching the color for the paint brush
- Working with a filesystem and output streams
- Saving files to the storage devices
- Adding permissions to the `AndroidManifest.xml` file
- Clearing the Canvas, drawing cache, and creating a new drawing
- Erasing the already drawn painting using very simple logic

We made it to the end of this book. This book only focuses on the very basics, starting from level 0 to mid-level knowledge of working with Canvas for Android. Still, the knowledge that is enclosed in this book will work for all kinds of basic 2D Android graphic applications. For a more complex application with animation, we might need advance-level knowledge of Android Canvas and some core knowledge of working with 3D in OpenGL ES. OpenGL ES is a flavor of OpenGL for embedded devices or handheld devices.

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

Thank you for buying

Learning Android Canvas

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, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

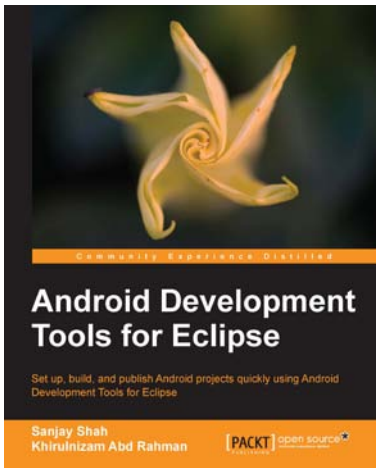
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

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, 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] open source
community experience distilled
PUBLISHING

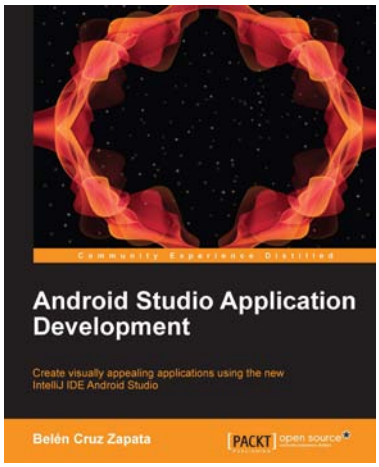


Android Development Tools for Eclipse

ISBN: 978-1-78216-110-3 Paperback: 144 pages

Set up, build, and publish Android projects quickly using Android Development Tools for Eclipse

1. Build Android applications using ADT for Eclipse
2. Generate Android application skeleton code using wizards
3. Advertise and monetize your applications



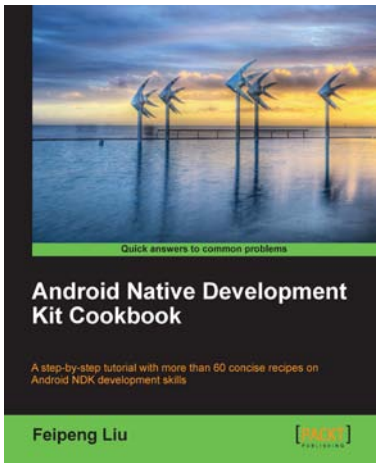
Android Studio Application Development

ISBN: 978-1-78328-527-3 Paperback: 110 pages

Create visually appealing applications using the new IntelliJ IDE Android Studio

1. Familiarize yourself with Android Studio IDE
2. Enhance the user interface for your app using the graphical editor feature
3. Explore the various features involved in developing an android app and implement them

[PACKT] open source
community experience distilled
PUBLISHING



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

Android 4: New Features for Application Development

ISBN: 978-1-84951-952-6 Paperback: 166 pages

Develop Android applications using the new features of Android Ice Cream Sandwich

1. Learn new APIs in Android 4
2. Get familiar with the best practices in developing Android applications
3. Step-by-step approach with clearly explained sample codes