Learning Java by Building Android Games

Android is the fastest growing operating system (OS) with one of the largest installed bases of any mobile OS. Android uses one of the most popular programming languages, Java, as the primary language for building apps of all types. So, you should first obtain a solid grasp of the Java language and its foundation APIs to improve the chances of succeeding as an Android app developer.

This book will show you how to get your Android development environment set up and you will soon have your first working game. The difficulty level grows steadily with the introduction of key Java topics such as loops, methods, and OOP. You’ll then use them in the development of games. You will learn how to build a math test game, a Simon-like memory game, a retro pong-style game, and for the grand finale, a Snake-style, retro arcade game with real Google Play leaderboards and achievements. The book has a hands-on approach and is packed with screenshots.

Who this book is written for
If you are completely new to either Java, Android, or game programming and are aiming to publish Android games, then this book is for you. This book also acts as a refresher for those who already have experience in Java on another platforms or other object-oriented languages.

What you will learn from this book
- Set up an efficient, professional game development environment in Android Studio
- Build your very own Android UI using easy-to-use tools in Android Studio
- Add real-time interaction with Java threads and implement locking/handling screen rotation, pixel graphics, clicks, animation, sound FX, and many other features in your games
- Explore object-oriented programming (OOP) and design scalable, reliable, and well-written Java games or apps on almost any Android device
- Build and deploy a graphical pong-style game using advanced OOP concepts
- Explore APIs and implement advanced features such as online leaderboards and achievements using Google game services
- Make your game compelling to be the next big hit on Google Play market with a content update strategy and in-game marketing
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 "Getting Started with Android"
- A synopsis of the book’s content
- More information on Learning Java by Building Android Games

About the Author

John Horton is a technology enthusiast based in UK. When he is not writing apps, books, or blog articles for www.gamecodeschool.com, he can usually be found playing computer games or indulging in a Nerf war.

A very big thanks to all the reviewers. I would also like to express my gratitude to the people at Packt Publishing who made this book possible, including Priyanka Budkuley for getting the project started, Pooja, Harsha, Shruti, Vikrant, Leena, Ting, Simran, Maria, Ameesha, Paul, Monica, and Arvindkumar.

To Jo, Jack, James, Ray, Rita, and Skipper, I dedicate this book to you.
Learning Java by Building Android Games

If you are completely new to any one of Java, Android, or game programming, and are aiming at publishing Android games for fun or for a business purpose, but you have no clue as to where to start, then this book is for you. This book also acts as a refresher for those who already have experience in Java on other platforms.

Android is the fastest growing operating system, and Android devices can empower, entertain, and educate the planet. Android uses one of the most popular programming languages—Java, which is a high-performance, secure, and object-oriented language.

In each chapter, we will build on what you learned from the previous chapter, steadily gaining an understanding of more advanced Java concepts and putting them to work by building Android games.

What This Book Covers

Chapter 1, Why Java, Android, and Games?, tells us why we might choose Java, Android, and games over other languages, platforms, and types of apps. Then we quickly prepare our Android and Java development environment by installing the Java Development Kit and Android Studio.

Chapter 2, Getting Started with Android, shows us how to design, build, and run a game menu UI on a real Android device. This is the first part of our math game project.

Chapter 3, Speaking Java – Your First Game, covers the Java fundamentals and their use in an Android environment to make a simple math quiz game.

Chapter 4, Discovering Loops and Methods, contains more Java fundamentals such as decision-making and looping. Then we use them to add features to our math game. This chapter also covers some basic Android game essentials such as locking and handling screen rotation, and introduces the device sensors.

Chapter 5, Gaming and Java Essentials, takes us on a quick ride through some vital essentials for any game. We also make the UI more exciting with animation and by storing player scores. Demonstrated with a Simon-style memory game, storing the sequence with Java arrays and a for loop.

Chapter 6, OOP – Using Other People’s Hard Work, is the chapter where everything you have learned so far is tied up with a pretty ribbon. You should find yourself muttering phrases like, "Oh, I see" and "So that's how it works." We explore object-oriented programming (OOP) in this chapter.
Chapter 7, *Retro Squash Game*, teaches us how to draw graphics pixel by pixel with the Android Canvas class. We can then combine these new skills with what we already know and make a retro, pong-style squash game.

Chapter 8, *The Snake Game*, covers the beginning of our most advanced game project—an animated *Snake-style* arcade game. We will also be able to practice more of the important Java fundamentals that you learned in the previous chapters.

Chapter 9, *Making Your Game the Next Big Thing*, puts the icing on the cake by teaching you how to publish your game and add online leaderboards and achievements. This will also let you examine the concept of Java libraries.

Appendix, *Self-test Questions and Answers*, contains all the answers to the questions to test your understanding of the topic.
In this chapter, we will take a roller coaster ride through all the Android topics that you need to learn in order to get started with Java. It won't just be theoretical though. We will be designing a user interface (UI) of a game menu and we will see and edit our first Java code as well.

In addition, we will see how we can run our apps in either an Android emulator on our PC/Mac or on a real Android device if we have one.

Some of what we will cover in this chapter will be the tip of the iceberg. That is, there is a lot more below the surface to some of the topics we discuss than would be appropriate for the second chapter of a learning Java book. Sometimes, we might need to take a bit of information on faith.

This will then enable us to actually design and run our very own Android app by the end of this chapter. Then we can start learning Java for real at the beginning of the next chapter.

If this chapter seems a little tough, then don't worry; keep going because each subsequent chapter lifts the lid a bit more from some of the less clear topics.

For this chapter and the next two, we will be building a math game. We will start simply and by the end of Chapter 4, Discovering Loops and Methods, we will scale to game features that use significant Java skills.
In this chapter, we will:

- Start our first game project
- Explore Android Studio
- Use the Android Studio visual designer to make our game UI
- Learn about structuring our code for Android
- Take our first look at some Java code
- Build and install our game on an emulator and a real device

Our first game project

Now we will get straight down to actually doing something with Android Studio. Run Android Studio by double-clicking on the Android Studio icon either on your desktop's start menu or in the folder where you installed it.

If you get any errors in a dialog box mentioning Privilege elevation, then try running Android Studio with administrator privileges. To do this, find the Android Studio icon by clicking on the Windows Start button and searching for Android Studio. Now right-click on the icon and click on Run as administrator. Do this every time you run Android Studio.

Preparing Android Studio

So with Android Studio and Java installed, we just need to add the latest versions of the Android API that we will use to make our first game. Here is what to do to install the API:

1. From the menu bar at the top of the Android Studio UI, navigate to Tools | Android | SDK Manager. Scroll down in the Android SDK Manager window and select the checkbox for Android 4.4.2 (API 19).

Note that because Android is evolving so quickly, by the time you read this chapter, there might be APIs newer than 19—20, 21, and so on. If this is the case for you, select the newer (higher numbered) API instead.
2. Click on Install Packages.
3. On the next screen, click on the Accept license checkbox and then click on the Install button. Android Studio will download and install the appropriate packages.

What we just did was setting up Android Studio to make available the latest, prewritten code called an API that we will interact with throughout the book.

**Building the project**

1. Click on New Project... as indicated in the following screenshot:
2. The **Create New Project** configuration window will appear. Fill in the **Application name** field with **Math Game Chapter 2** and **Company Domain** with **packtpub.com** (or you could use your own company website name here), as shown in the following screenshot:
3. Now click on the **Next** button. On the next screen, check that the **Phone and Tablet** checkbox has a tick in it. Now we have to choose the earliest version of Android we want to build our app for. Go ahead and play with a few options in the drop-down selector. You will see that the earlier the version we select, the greater is the percentage of devices our app can support. However, the trade-off here is that the earlier the version we select, the fewer are the cutting-edge Android features available in our apps. A good balance is to select **API 8: Android 2.2 (Froyo)**. Go ahead and do that now as shown in the next screenshot:
4. Click on **Next**. Now select **Blank Activity** as shown in the next screenshot and click on **Next** again:

5. On the next screen, simply change **Activity Name** to **MainActivity** and click on **Finish**.

   By default, Android Studio shows a **Tip of the day** dialog every time it starts. Some of the tips might not make sense while you are still learning Java but many of them are really useful and reveal great shortcuts and other time savers. It is well worth taking a few seconds to read them when they show up. As already discussed, Android Studio is built from IntelliJ IDEA, and you can find a complete list of keyboard shortcuts at [http://www.jetbrains.com/idea/webhelp/keyboard-shortcuts-you-cannot-miss.html](http://www.jetbrains.com/idea/webhelp/keyboard-shortcuts-you-cannot-miss.html).

6. Clear **Tip of the day** by clicking on **Close**.
If you are completely new to programming, then the code, options, and files might seem a bit daunting. Don't worry; stick to them as we don't need to concern ourselves with most of them in order to learn Java. When the time does come to interact with the finer details, we will do things one step at a time.

It might be hard to believe that at this stage, but we have just created our first working app. We could build and run it on an Android device, and soon we will.

Let's take a deeper look at Android Studio before we progress with our game.

**Exploring Android Studio**

Android Studio is a very deep tool, but it is only necessary to learn one part at a time in order to get started. What might be useful for us is naming a few parts of the UI so that we can refer to them easily as we progress through the book.

Take a look at this numbered diagram and a quick explanation of some of the key parts of Android Studio. If you can, try and memorize the parts to make future discussions of them easier for you.
Here is a handy table that you can quickly refer to and remember which part of Android Studio we are referring to. What follows is a more detailed explanation of each area.

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Project Explorer</td>
</tr>
<tr>
<td>2</td>
<td>The editor</td>
</tr>
<tr>
<td>3</td>
<td>Menu bar</td>
</tr>
<tr>
<td>4</td>
<td>Toolbar</td>
</tr>
<tr>
<td>5</td>
<td>Navigation bar</td>
</tr>
<tr>
<td>6</td>
<td>Important tool windows</td>
</tr>
</tbody>
</table>

- **Project Explorer (1):** This is shown as 1 in the screenshot and is a bit like Windows Explorer. It shows us all the files and folders that have been generated for our project. We will do a number of things from here as the book continues. Actually, if you delve into the files and folders that Android Studio creates, the Project Explorer is not an exact mapping. It is slightly simplified and highlighted to make managing and exploring our project easier.

- **The Editor (2):** As the name suggests, we will edit our Java code files here in the editor. However, as we will soon see, the Editor window changes depending on the type of file we are editing. We will also be viewing and editing UI designs here.

- **Menu bar (3):** Like most programs, the Menu bar gives us access to the full functionality of Android Studio.

- **Tool bar (4):** This contains lots of really useful one-click options to do things such as deploying and debugging our games. Hover the mouse cursor over an icon to get a pop-up tip and gain greater insight into each toolbar icon.

- **Navigation bar (5):** Like a file path, this shows exactly where the file that is currently in the editor is located within the project.

- **Important tool windows (6):** These are a number of tabs that can be popped up and popped down again by clicking on them. If you like, try some of them now to see how they work.

Let’s talk a bit more about the parts of the Android Studio UI and how the editor window can transform itself into a visual UI designer. After that, when we are familiar enough, we will look at building a simple menu screen for our math game.
Using the Android Studio visual designer

The Android Studio editor window is a very dynamic area. It presents different file types in the most useful way possible. A little earlier, when we created our project, it also made a basic UI for us. UIs in Android can be built in Java code or, as we will see, in a visual designer without the need for a single line of Java. However, as we will investigate after we have built the UI of our game menu, to get the UI to do anything useful, we need to interact with it. This interaction is always done with Java code. The visual designer also generates the UI code for us. We will take a very quick look at that too.

As the book progresses, we will mainly shy away from Android UI development, as that is a staple of more non-game apps. We will instead spend more time looking at directly drawing pixels and images to make our games. Nonetheless, the regular Android UI has its uses, and the Android Studio visual designer is the quickest way to get started.

Let’s have a look at that now:

1. In the Android Studio Project Explorer, double-click on the layout folder to reveal the activity_main.xml file within it. This should be easy to see unless you have collapsed the directories. If you can’t see the layout folder, navigate to it using the Project Explorer. It can be found at Math Game Chapter2/src/main/res/layout via the Android Studio Project Explorer, as shown in the following screenshot:
2. Now double-click on `activity_main.xml` to open it in the editor window. After a brief loading time, you will see something very similar to the next screenshot. The following screenshot shows the entire contents of what previously contained just our code. As you can see, what was just a text window now has multiple parts. Let's take a closer look at this screenshot:

In the preceding screenshot labeled (1), called **Palette**, you can choose from the available Android UI elements and simply click and drag them onto your UI design. Area (2) is the visual view of the UI you are building, where you will click and drag elements from the palette. To the right of the visual UI view, you will see the **Component Tree** area (3). The component tree allows you to examine the structure of the complex UI and select specific elements more easily. Under this tree is the **Properties** panel (4). Here you can adjust the properties of the currently selected UI element. These can be simple things such as color and size or much more advanced properties.
Note the tabs labelled (5). These tabs allow you to switch between the two main views that Android Studio provides for this type of layout file. These views, as you can see, are Design and Text. The design view is the default view and is shown in the previous screenshot. The text view also shows your under-construction UI, but it shows the code that has been autogenerated for us instead of the Palette element and the component tree.

We don’t need to worry about this code as it is all handled for us. It can be good to look on this tab from time to time so that we can begin to understand what the design tool generates for us. But it is not necessary to do this to learn Java. This code is called eXtensible Markup Language (XML).

3. Take a quick look at the Text tab, click on the Design tab when you’re done, and we will move on.

Now we have seen an overview of the visual designer and an even briefer glimpse of the automatically generated code that it generates for us. We can take a closer look at some of the actual UI elements that we will be using in our project.

**Android UI types**

We will now take a whirlwind tour of some really useful Android UI elements, a few key properties, and how to add them together to make a UI. These will introduce us to some of the possibilities as well as how to use them. We will then quickly use what we know to make our menu.

**TextView**

On the visual UI area, click on the words Hello world!. What we have just selected is a widget known as a TextView. TextViews can be small text like this one or large heading type text, which might be useful in our game menu.
Let's try dragging and dropping another TextView onto our visual UI:

1. Directly under the **Widgets** heading in our palette, you can see there are multiple types of TextView. They are presented in the palette as **Plain TextView**, **Large Text**, **Medium Text** and **Small Text**. Drag and drop a **Large Text** widget onto our visual design. Don't let go straightaway. As you drag it around the image of the phone, notice how Android Studio graphically shows you different positioning options. In the following screenshot, you can see what the designer looks like when the widget being dragged is positioned at the center:

![Screenshot of designing with Android Studio](image-url)
2. Let go of the left mouse button where you want the widget to go. If you let go when it is positioned as shown in the previous screenshot, then the text will appear in the center as expected.

3. Now we can play with the properties. In the Properties window, click just to the right of \texttt{textSize}. You might need to scroll to find it. Type \texttt{100sp} as the value and press the \texttt{Enter} key. Notice that the text gets much larger. We can refine the size of our text by increasing and decreasing the value entered here. The unit \texttt{sp} stands for scaled pixels, and is simply a measuring system that attempts to scale the text to an appropriate equivalent actual size across different screen densities.

![Text Size Property](image)

4. Play with some more properties if you like and when you're done, click on the TextView we just created in the visual designer to highlight it. Then tap on the \texttt{Delete} key to get rid of it. Now delete the TextView that was present when we started—the one that says \textbf{Hello world}!

### Layout elements

Now you have an apparently empty screen. However, if you click anywhere on the design preview, you will see that we still have some options in the Properties window. This element is called a RelativeLayout. It is one of several layout element types provided as a base to control and align the layout widgets such as buttons, text, and so on. If you look at the top of the Palette window, you will see the main layout options. We will use this layout element when we actually build our game menu in a moment.

### ImageView widgets

ImageViews unsurprisingly are for displaying images. In the standard Android UI, this is a really quick way to add our designers' artwork to our game:

1. Drag and drop an \texttt{ImageView} element onto the design in the same way as you positioned the TextView a moment ago. The \texttt{ImageView} element can be found below the \texttt{Widgets} heading. Now position it centrally as before or play with the options by dragging it around the design. We will delete it in a minute; we are just having a bit of an exploration before we do this for real.

2. In the Properties window, select the \texttt{src} property in the same way as you selected the \texttt{textSize} property previously.
3. Notice that after you select it, you have the option to click on ... to give you more options. Click on ... and scroll to the bottom of the list of options. These are all the image files that we can display in this ImageView. Just for fun, scroll to the bottom of the list, choose ic_launcher, and click on OK. We can make any image we like available and this is a simple, powerful way to build an attractive game menu screen.

4. Change the layout:width property to 150dp and the layout:height property to 150dp. The unit dp is a way of sizing elements and widgets that remains relatively constant across devices with screens that have very different numbers of pixels.

5. Delete the ImageView in exactly the same way as you deleted the other views previously.

**ButtonView**

The use of ButtonView is probably given away by its name. Try to click and drag a few buttons onto our layout. Notice that there are a few types of ButtonView, such as Small Button, Button, and, if you look further down the Widget list, ImageButton. We will be using the regular ButtonView, labelled simply as Button.

Now we will do something with each of these Android UI elements combined to make our game menu.

---

You can download the entire sample from the code download section of the book’s companion website.

---

**Using the sample code**

All of the code in this book is organized in projects. If a project spans more than one chapter, a project is provided for each chapter in its finished state. This helps you see the progression and not just the end result. All you need to do to open the project in Android Studio is explained as follows:

1. Download the code for this book.
2. In Android Studio from the menu bar, navigate to File | Close project.
3. Now create a new blank project as we did previously. Browse to where you downloaded the code for this book.
4. Navigate to the Chapter2/MathGameChapter2 folder. Here you will find the code for all the files we create in this chapter.
5. Open the code files using a plain text editor such as the free Notepad++.
6. Copy and paste in your Android Studio project or just compare the code as you see it.

Although every line of code required in this book is supplied for your convenience, you still need to create each project for yourself through Android Studio. You can then simply copy and paste either the code in its entirety in the file with the matching name, or just the part of the code that you might be struggling with. Keep in mind that if you create a project with a different package name, then you must omit the line of code that is the package name from the supplied code files. The reasons for this will be clearer when we talk more about packages later in the chapter.

Let's actually see how to do it all for ourselves.

**Making our game menu**

For now we will just make our game menu functional. Later in *Chapter 5, Gaming and Java Essentials*, we will see how we can make it look good by adding some cool animation to make the menu more visually interesting and fun.

Here is what we are aiming for in this tutorial:
Before you start coding, you should design your layouts on paper first. However, the Android Studio designer is so friendly there is a strong argument, especially for simple layouts, to refine your design actually in the layout designer. Perform the following steps to create the game menu:

1. Delete all widgets from your designer by clicking on them one at a time and then tapping the Delete key on each in turn. Be careful not to delete the RelativeLayout layout element as we are going to use it as a base for all the other elements.

2. Click and drag a Large Text element from the palette to the top center of the design area and give it the following properties. Remember that you can change properties in the Properties panel by clicking to the right of the property to be changed. Change the text property to My Math Game and size to 30sp.

3. Click and drag an ImageView element from the palette to the center of the design, slightly below the previous TextView. Change the layout:width property to 150dp and the layout:height property to 150dp.

4. Now click and drag three buttons for Play, High Scores and Quit. Center them vertically, below the previous ImageView and one below the other, as per our design shown previously.

5. Click on the top button, configure the text property, and enter the value Play.

6. Click on the middle button, configure the text property, and enter the value High Scores.

7. Click on the lowest button, configure the text property, and enter the value Quit.

8. As the buttons now contain different amounts of text relative to each other, they will be of slightly different sizes. You can even them up to match the intended layout by clicking and dragging the edges of the smaller buttons to match the larger ones. This is done in mostly the same way as you might resize an application window in Windows.

9. Save the project with Ctrl + S or by navigating to File | Save All.
If you are going to be testing your games on a much larger or much smaller screen than the Nexus 4 shown in the designer, then you might like to adjust the values of the sp and dp units used in this tutorial.

A full discussion of Android UI on multiple devices is beyond the scope of this book and is not necessary to make any of the games in this book. If you want to start designing for different screens right away, take a look at http://developer.android.com/training/multiscreen/index.html.

You can view what your menu looks like on other devices simply by selecting the device from the drop-down menu shown in the following screenshot:

Before we make our menu come to life on an actual device, let's take a look at the structure of an Android app and how we can use that structure when writing our Java code.
Structuring our code for Android

If you have ever used an Android device, you have probably noticed that it works quite differently from many other operating systems. For example, you are using an application—say you’re checking what people are doing on Facebook. Then you get an e-mail notification and you tap the e-mail icon to read it. Midway through reading the e-mail, you might get a Twitter notification and because you’re waiting on important news from someone you follow, you interrupt your e-mail reading and change the app to Twitter with a touch.

After reading the tweet, you fancy a game of Angry Birds, but midway through the first daring fling, you suddenly remember that Facebook post. So you quit Angry Birds and tap the Facebook icon.

Then you resume Facebook, probably at the same point you left it. You could have resumed reading the e-mail, decided to reply to the tweet, or started an entirely new app. All this backwards and forwards takes quite a lot of management on the part of the operating system, apparently independent from the individual apps themselves.

The difference between a Windows PC and Android in the context we have just discussed is that with Android, although the user decides which app they are using, the Android OS decides if and when to actually close down (destroy) an application. We just need to consider this when coding our games.

Life cycle phases – what we need to know

The Android system has different phases that any given app can be in. Depending on the phase, the Android system determines how the app is viewed by the user or whether it is viewed at all. Android has these phases so that it can decide which app is in current use and then allocate the right amount of resources such as memory and processing power. But also allow us as game developers to interact with these phases. What if someone quits our game to answer a phone call? Will they lose their progress?

Android has a fairly complex system that, when simplified a little for the purpose of explanation, ensures that every app on an Android device is in one of the following phases:

• Being created
• Starting
• Resuming
• Running
• Pausing
• Stopping
• Being destroyed

The list of phases will hopefully appear fairly logical. As an example, the user presses the Facebook app icon and the app is created. Then it is started. All are fairly straightforward so far but next in the list is resuming! It is not as illogical as it might first appear if, for a moment, we can just accept that the app resumes after it starts, and then all will become clear as we proceed.

After resuming, the app is running. This is when the Facebook app has control over the screen and probably the greater share of system memory and processing power. Now what about our example where we switched from the Facebook app to the e-mail app?

As we tap to go to read our e-mail, the Facebook app will probably have entered the paused phase, and the e-mail app will enter the being created phase followed by resuming and then running. If we decide to revisit Facebook, as in the scenario earlier, the Facebook app will probably then go straight to the resume phase and then running again, most likely exactly on the post where we left it.

Note that at any time, Android can decide to stop or destroy an app, in which case, when we run the app again, it will need to be created all over again. So had the Facebook app been inactive long enough or had Angry Birds required so many system resources that Android would have destroyed the Facebook app, then our experience of finding the exact post we were previously reading might have been different.

Now, if all this phase stuff is starting to get confusing, then you will be pleased to know that the only reasons to mention are as follows:

• You know it exists
• We occasionally need to interact with it
• We will take things step by step when we do

Life cycle phases – what we need to do

When we are making games, how do we possibly interact with this complexity? The good news is that the Android code that was autogenerated when we created our first project does most of the interaction for us.

All we have to do as game developers is make sure that Android knows what to do with our app in each phase when it happens. Even more good news is that all of these phases are handled by default, unless we override the default handling.
This means we can go ahead with learning Java and making games until we come to one of the few instances where we need to do something in our game, specifically in one of the phases.

**Dividing our game into activities**

The Java code that we write will be divided into sections or parts called activities. We can think of activities as different screens for our game. For example, during the book, we will often create an activity for a home screen, an activity for the game screen and an activity for the high score screen.

Each activity will have its own life cycle and will be further divided into parts that will correspond to (go into) one of the Android phases we just discussed. The parts in Java are known as methods. Methods are a significant concept in Java programming.

At this stage, however, all we need to know is that methods are used to compartmentalize the Java code we write and that some methods are provided by the Android system so that we can easily handle the otherwise complex Android life cycle.

The forthcoming list is a quick explanation of the methods provided by Android for our convenience, to manage the phases of the life cycle. To clarify our discussion of life cycle phases methods are listed next to their corresponding phases that we have been discussing. However, as you will see, the method names make it fairly clear on their own where they fit in.

In the list, there is also a brief explanation or suggestion about when we should use a given method and thereby interact during a specific phase. We will meet most of these methods as we progress through the book. We will see the `onCreate` method later in this chapter. Here is the list:

- **onCreate**: This method is executed when the activity is being created. Here we get everything ready for the game, including graphics, sound, and perhaps the high scores.
- **onStart**: This method is executed when the app is in the starting phase.
- **onResume**: This method runs after `onStart` but can also be entered, perhaps most logically, after our activity is resumed after being previously paused. We might reload a previously saved game situation when the app had been interrupted, perhaps by a phone call or the user running another app.
- **onPause**: This occurs when our app is pausing. Here we might want to save the current game. You are probably getting the hang of these methods.
- **onStop**: This relates to the stopping phase. This is where we might undo everything we did in `onCreate`. If we reach here, our activity will probably get destroyed sometime soon.
• **onDestroy**: This is when our activity is finally being destroyed—our last chance to dismantle our game. If we reach here, we will definitely be going through the phases of the life cycle from the beginning again.

All the method descriptions and their related phases should appear straightforward. Perhaps, the only real question is about the running phase. As we will see, when we write our code in other methods/ phases, the **onCreate**, **onStart**, and **onResume** methods will prepare the game, which persists, forming the running phase. The **onPause**, **onStop**, and **onDestroy** methods will occur afterwards. Now we can actually take a look at one of these methods and some other methods as well.

### Our first look at Java

So what about all that code that Android Studio generated when we created our new project earlier? This is the code that will bring our game menu to life. Let's take a closer look. The very first line of code in the editor window is this:

```java
package com.packtpub.mathgamechapter2;
```

This line of code defines the package that we named when we first created the project. As the book progresses, we will write more complex code that spans more than one file. All the code files we create will need the package they belong to, clearly defined like the previous line of code, at the top. The code doesn't actually do anything in our game. Notice also that the line ends with a semicolon (`;`). This is a part of the Java syntax and it denotes the end of a line of code. Remove a semicolon and you will get an error because Android Studio tries to make sense of two lines together. Try it if you like.

Remember that if you are going to be copying and pasting the code from the download bundle, this is the one line of code that might vary depending on how you set up your project. If the package name in the code file is different from the package name you created, always use the package name from when you created the project.
To see the next four lines of code, you might need to click on the small + icon to reveal them. Android Studio tries to be helpful by simplifying our view of the code. Notice that there are several little - icons as well down the side of the editor window. You can expand and collapse them to suit yourself without affecting the functionality of the program. This is shown in the following screenshot:

```
package com.packtpub.mathg

import ...

public class MainActivity
```

Once you have expanded the code, you will see these four lines:

```java
import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
```

Notice that all the preceding lines start with the word `import`. This is an instruction to include other packages in our game, not just our own. This is very significant because it makes available to us all of the hard work of other programmers, the Android development team in this case. It is precisely these imports that give us the ability to use the methods we discussed earlier, and allow us to interact with the Android life cycle phases. Notice again that all the lines end with a semicolon (`;`).

The next line introduces a fundamental building block of Java known as a `class`. Classes are something that we will continually expand our knowledge and understanding of throughout the book. For now, take a look at this line of code, then we will discuss it in detail:

```java
public class MainActivity extends ActionBarActivity {
```

Word by word, here is what is going on. The preceding line is saying: make me a new `public class` called `MainActivity` and base it upon (`extends`) `ActionBarActivity`.

You might remember that `MainActivity` is the name we chose while creating this project. `ActionBarActivity` is the code (known as a class) written by the Android development team that enables us to put our Java into Android.
If you have a keen eye, you might notice there is no semicolon at the end of this line. There is, however, an opening curly brace ({$}). This is because MainActivity encompasses the rest of the code. In effect, everything is part of our MainActivity class, which is built based on the ActionBarActivity class/code. If you scroll down to the bottom of the editor window, you will see a closing curly brace ($}). This denotes the end of our class called MainActivity.

- We do not need to know how a class works yet
- We will use classes to access some methods contained within its code and without doing any more, we are already, by default, taking advantage of the Android life cycle methods we discussed earlier
- We can now pick and choose if, when, and which methods defined in these classes we wish to override or leave as default

So, it is the ActionBarActivity class that contains the methods that enable us to interact with the Android life cycle. Actually, there are a number of different classes that enable us to do this and in a moment, we will change from using ActionBarActivity to a more appropriate class that also does all the things just mentioned.

It is not important at this point to properly understand Java classes; just understand that you can import a package and a package can contain one or more classes that you can then use the functionality of or base your own Java programs on. We will bump into classes regularly in the next few chapters. Think of them as programming black boxes that do stuff. In Chapter 6, OOP – Using Other People's Hard Work, we will open the black box and really get to grips with them and we will even start making our own classes.

Moving on with the code, let's look at what the code that is contained within our class actually does.

Here is the code chunk directly after the crucial line we have just been discussing:

```java
@override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
}
```

Hopefully, some of this code will start to make sense now and tie in with what we have already discussed. Although the precise syntax will still feel a little alien, we can continue learning Java as long as we are aware of what is happening.
The first thing we notice in the preceding code is the word `@Override`. Remember when we said that all the methods that interact with the Android life cycle were implemented by default and we can pick and choose if and when to override them? This is what we are doing here with the `onCreate` method.

The `@Override` word says that the method that follows next is being overridden. The `protected void onCreate(Bundle savedInstanceState) {` line contains the method we are overriding. You might be able to guess that the action starts with the opening `(` at the end of the line in question and ends with the closing `) three lines later.

The somewhat odd-looking `protected void` before the method name `onCreate` and `(Bundle savedInstanceState)` after the method name are unimportant at this time because they are handled for us. It is to do with the data that travels between various parts of our program. We just need to know that what happens here will take place in the creating phase of the Android lifecycle. The rest will become clear in Chapter 4, Discovering Loops and Methods. Let’s move on to the line:

```
super.onCreate(savedInstanceState);
```

Here, the `super` keyword is referencing the code in the original `onCreate` method, which is still there even though we can’t see it. The code is saying: even though I am overriding you, I want you to set things up, just like you normally do first. Then, after `onCreate` has done loads of work that we don’t see and don’t need to see, the method continues and we actually get to do something ourselves with this line of code:

```
setContentView(R.layout.activity_main);
```

Here we are telling Android to set the main content view (our users screen), which is the cool game menu we created earlier. To be specific, we are stating it is an `R` or resource in the `layout` folder and the file is called `activity_main`.

### Cleaning up our code

The next two blocks of code were created by Android Studio on the assumption that we would want to override another two methods. We don’t, because they are methods more often used in non-gaming apps:

1. Delete the entire content shown in the following code. Be careful not to delete the closing curly brace of our `MainActivity` class:

```
@Override
    public boolean onCreateOptionsMenu(Menu menu) {

        // Inflate the menu; this adds items to the action bar if
        // it is present.
        getMenuInflater().inflate(R.menu.main, menu);

    }
```

```
return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Handle action bar item clicks here. The action bar will
    // automatically handle clicks on the Home/Up button, so
    long
    // as you specify a parent activity in AndroidManifest.
    int id = item.getItemId();
    if (id == R.id.action_settings) {
        return true;
    }
    return super.onOptionsItemSelected(item);
}

2. Now we can delete a couple of the @import statements. The reason for this is that we just deleted the overridden methods of classes (imported earlier) we no longer need. Notice that the following lines in the editor window are grey. Note that the program would still work if you leave them in. Delete them both now to make your code as clear as possible:
import android.view.Menu;
import android.view.MenuItem;

3. Some final amendments before our code is done: at this point, you might be thinking that we have deleted and changed so much of our code that we might as well have started from an empty page and typed it in. This is almost true. But the process of having Android Studio create a new project for us and then making these amendments is more thorough and also avoids quite a few steps. Here are the last code changes. Change the import android.
support.v7.app.ActionBarActivity; line to import android.support.
app.Activity;

4. Now you will get several red lines underlining our code and indicating errors. This is because we are attempting to use a class we have not yet imported. Simply amend the public class MainActivity extends
ActionBarActivity { line to public class MainActivity extends
Activity {

What we did with those last two changes was using a slightly more appropriate version of the Activity class. To do this, we also had to change what we imported.
When you're done, your editor window should look exactly like this:

```java
package com.packtpub.mathgamechapter2.mathgamechapter2;

import android.support.v7.app.ActionBarActivity;
import android.os.Bundle;

public class MainActivity extends ActionBarActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
  }
}
```

Now that we know what's going on and our code is clean and lean, we can actually take a look at the beginnings of our game in action!

If any of what we have just discussed seemed complicated, there is no need for concern. Android forces us to work within the Activity lifecycle, so the previous steps were unavoidable. Even if you didn't follow all the explanations about classes and methods and so on, you are still perfectly placed to learn Java from here. All the classes and methods will seem much more straightforward as the book progresses.

### Building and installing our game

Soon, we will actually see our menu in action. But before we do that, we need to find out how to use the Android emulators and how to build our game. Then we will put these together and put our game into an emulator or real device to see it as our players would.
Emulators and devices
Now we have the first part of our game ready to run. We need to test it to check for any errors, crashes, or anything else unintended. It is also important to ensure that it looks good and runs correctly on the device types/sizes and that you want to target.

We will not go into any details about handling different device types. All our games are fullscreen and we will later lock the orientation and dynamically calculate aspects such as screen resolution. So we can get away with writing for a single device type and focus on learning Java.

It might be useful to know for now that you can create a different layout file for any screen size categorization or pixel density. All you need to do is place the layout file using exactly the same filename in the appropriate folder. The Android device will then know the most appropriate layout for it to use. For a detailed discussion, see the Google developers website at http://developer.android.com/guide/practices/screens_support.html.

Note that you do not need to understand any of the information at the preceding link to learn Java and publish your first games.

There are a few options to do this and we will look at two. First, we will use Android Studio and the Android Development Tools to make a device emulator so that we can use, test, and debug our games on a wide range of device emulators on the same PC/Mac we are developing on. So we don't need to own a device. This will allow us to get crash reports from our games.

Then we will install the game directly to a real device so that we can see exactly what the owner of that device will see when they download our app.

There are more options. For example, you can connect a real device via USB and debug directly on the device with the errors and syntactical feedback in Android Studio. The process for this might vary for different devices and since we won't be focusing on anything but basic debugging, we will not cover that in this book.
Creating an emulator
Let's get our emulator up and emulating:

1. On the right-hand side of the Android Studio quick launch bar, find the AVD manager icon:

2. Click on the icon to start the Android Virtual Device Manager. Then click on the Create Virtual Device... button on the bottom-left side to bring up the Virtual Device Configuration window.

3. Now click on the Nexus 4 option and then click on Next.

4. Now we need to choose the version of Android we will use to build and test our games on. The latest version (at time of writing) is Lollipop - 21 - x86. It is the only option where we don't need to complete a download to continue. So select it (or whatever the default is at the time you are reading this) and then click on Next to continue.

5. On the next screen we can leave all the default settings. So click on Finish.

We now have a ready-to-run Android emulator.

Running the emulator
Now we will launch (switch on) our virtual device then actually run our game that we made earlier by performing the following steps:

1. Click on Nexus 4 API 21 under the Name column. Now click on the triangular play icon to the right of the description of our emulator.

   Android emulators take a long time to start. Even on a high-specification PC. Expect to wait at least a few minutes or even 10.
2. Once it has started, unlock the device by clicking and dragging anywhere on the screen of the emulated device. This is analogous to swiping to unlock a real Nexus 4. Here is what our Nexus 4 virtual device looks like when it is running and unlocked:

![Emulator Screen](image)

You can play with this emulator in almost the same way as you can a real Android device. However, you cannot download apps from Google Play. You might notice that the emulator is a bit slow compared to a real device, even compared to an old one. Shortly, we will look at running our apps on a real device.
Running our game on the emulator

Once the emulator is running, it's usually best to leave it running so that each time we want to use it, we don't have to wait for it to start. Let's use the emulator:

1. Launch the emulator if it is not already running and make sure the device is unlocked as described previously.
2. Click on the run icon in the toolbar (shown next) to run your app. You can achieve the same thing by navigating to Run | Math Game Chapter 2 from the menu bar:

   ![Run icon]

3. After a pause while Android Studio builds our application, a pop-up dialog will ask you which device you want to run the app on. Choose the device with Nexus 4 API 21 in the description. This is the already running device that we created earlier. Now press OK.
4. Notice at this point that the useful Android window appears at the bottom section of Android Studio. In the unlikely event of you having any problems, just check for typos in the code. If things really don't work out, just go back to the Using the sample code section to compare with or copy and paste the supplied code.

   After another pause, our game menu screen will appear on the emulator. Of course, it doesn't do anything yet, but it is running and the buttons can be pressed.

   When you're done, you can press the back or home icons to quit the application, just as you would on a real Android device.

   Now we have seen one of the ways we can test our app by running it in the Android emulator. Let's find out how to make our code into an app we can distribute and use on a real device.

Building our game

To run our game on a real Android device, we need to create a .apk file, that is, a file that ends with the extension .apk. A .apk file is a compressed archive of files and folders that the Android system uses to run and install our app. These are the steps to use Android Studio to make a .apk of our game:

1. From the menu bar, navigate to Build | Generate Signed APK.
2. A slightly verbose window will pop up and say: For Gradle-based projects, the signing configuration should be specified in the Gradle build scripts. You can safely dismiss this window by clicking on OK.

3. Next up is the Generate Signed APK Wizard dialog. Here, we are creating a key that identifies the key holder as authorized to distribute the APK. At the end of this process, you will have a .keys file that you can use each time you build a .apk file. So this step can be missed out in future. Click on the Create new button.

4. In the Key Store Path field, type or go to a location on your hard drive where you would like to store your key. You will then be prompted to choose a filename for the keystore. This is arbitrary. Type MyKeystore and click on OK.

5. Type a password in the Password field and then retype it in the Confirm field. This is the password to a store that will help protect your key.

6. Next, in the Alias field, type a memorable alias. You can think of this as a kind of username for your key. Again type a password in the Password field and then retype it in the Confirm field. This is the password to your key.

7. Leave the Validity Years dropdown at the default of 25.

8. You can then fill out your Name and organization details (if any) and click on OK.

9. Now our key and keystore are complete, and we can click on OK on the Generate Signed APK wizard dialog.

10. We are then prompted to select Run Proguard. Encrypting and optimizing our .apk is unnecessary at this time. So just click on Finish to generate our app's .apk file.

11. The generated .apk file will be put in the same directory that you chose to put the project files. For example, MathGameChapter2/app.

We have now built a .apk file that can be run on any Android device that was specified when we first created the project.

**Installing the setup to a device**

So we have our .apk file and we know were to find it. Here is how we will run it on our Android device.
We can use one of a number of methods to get the .apk file into the device. The method I find one of the easiest is the use of a cloud storage service such as Dropbox. You can then simply click and drag the .apk file to your Dropbox folder and you're done. Alternatively, your Android device probably came with PC synchronization software that allows you to drag and drop files to and from your device. After you have placed the .apk file on your Android device, continue with the tutorial.

Most Android phones are set not to install apps from anywhere except the Google Play Store. So we need to change this. The exact menus you will navigate to might vary very slightly on your device but the following options tend to be almost the same on most devices, old and new:

1. Find and tap the Settings app. Most Android phones also have a Settings menu option. Either will do. Now select Security and scroll down to the Unknown sources option. Tap the Unknown sources checkbox to allow apps to be installed from unknown sources.

2. Locate the file on your Android device using the Dropbox app or your devices file browser depending on the method you chose to put the APK on your device. Tap the MathGameChapter2.apk file.

3. You can now install the app just like any other. When prompted, press Install and then Open. The game will now be running on your device.

Hold your device in a portrait orientation as this is how the UI was designed. Congratulations on running your very own Android app on your own device. In a later version of the math game, we will lock the orientation to make this more user friendly.

**Future projects**

Throughout the book, we will test and run our game projects. It is entirely up to you which of the methods we discussed you prefer. If you are getting crashes or unexplained bugs, then you will need to use an emulator. If all is working well, then the quickest and probably most pleasing way will be to run it on a device you own.

**Self-test questions**

Q1) What should you do if all this talk of life cycles, classes, and methods is a bit bemusing?

Q2) What exactly is a Java class?

Q3) What is the difference between a method and a class?
Q4) Take a look at the Android developer site and its more technical explanation of the phases of the life cycle at http://developer.android.com/reference/android/app/Activity.html. Can you see the phase and its related method that we haven't discussed? When would it be triggered in an app? What is the precise pathway an activity takes from creation to destruction?

Summary

We discussed that so far, it has not been not important to completely understand exactly how the code works. This is because it will act just as a container for the code we write in the rest of the book. However, as we cover in detail topics such as methods in Chapter 4, Discovering Loops and Methods, and classes in Chapter 6, OOP – Using Other People’s Hard Work, we will begin to make sense of all of the code in our games.

We discussed the somewhat complex Android life cycle in detail. We learned that all we need to understand at this stage is that we must write our code within the correct methods that relate to different phases of the life cycle. Then we will have no trouble making good progress with learning Java. As with classes and methods, all will be explained along the way and become clearer with practice.

We also learned the key areas of the Android Studio UI. We built our start menu for our math game using the Android Studio designer. Furthermore, we created the Java code necessary to make the game appear on the player’s device. This was achieved mainly by modifying the code that was automatically generated for us.

This was probably the most difficult chapter of the book because it was necessary to introduce a few things such as Java classes, Java methods, and the Android life cycle. We did this because we need to know what is going on around us as we learn Java.

From now on, however, we can take things a step at a time in a very logical manner. If you have reached this point, you will have no problem completing the toughest of the projects in this book.

If this chapter made your brain ache a little, rest assured that the fact that you have made it this far is a very good indication that you are going to be a Java ace someday soon. Starting from the basics, let's learn some Java now.
Where to buy this book

You can buy Learning Java by Building Android Games from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.