

Learning Java by Building Android Games

Second Edition

Learn Java and Android from scratch by building six exciting games



By John Horton

Packt

www.packt.com

Learning Java by Building Android Games

Second Edition

Learn Java and Android from scratch by building six
exciting games

John Horton



BIRMINGHAM - MUMBAI

Learning Java by Building Android Games

Second Edition

Copyright © 2018 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(s), 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.

Commissioning Editor: Amarabha Banerjee

Acquisition Editor: Reshma Raman

Content Development Editors: Onkar Wani

Technical Editor: Ralph Rosario

Copy Editor: SAFIS

Project Coordinator: Devanshi Doshi

Proofreader: Safis Editing

Indexers: Pratik Shirodkar

Graphics: Jason Monterio

Production Coordinator: Aparna Bhagat

First published: January 2015

Second edition: August 2018

Production reference: 1270818

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78883-915-0

www.packtpub.com



80% off

All eBooks & Videos

(Redeemable once)

Go to www.packtpub.com
and use this code in the checkout:

HBPROG80

Valid from 17th June to 4th August 2019

Packt>



`mapt.io`

Mapt is an online digital library that gives you full access to over 5,000 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
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

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

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

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

Boon Hian Tek is a 18 years Java veteran. He has honed his software development skills in a varying industries. He helped bringing medical records online, aided the advertising industry target ads, assisted in getting disposing of chemical weapons, and more recently part of making fin tech work some wonders.

He is always passionate with automation. Boon spends his days automating so that he can spend more time with his family. No one should spend be forced to spend time doing mundane stuff. Except for wax on, wax off!

He works for one of the largest fin tech in Singapore also reviewed First Edition of this book.

My wife and daughter putting up with the time I spent away from them to help with the book.

Packt is Searching for Authors Like You

If you're interested in becoming an author for Packt, please visit 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.

Table of Contents

Preface	xvii
Chapter 1: Java, Android and Game Development	1
What's new in the second edition?	2
Why Java, Android and Games?	2
Java: The first stumbling block	3
The games you will build	4
Sub' Hunter	4
Pong	5
Bullet Hell	6
Snake Clone	6
Scrolling Shooter	7
Open-World Platformer	8
How Java and Android work	9
Setting up Android Studio	10
Final step	17
Starting the first project: Sub' Hunter	18
Extra step 1	24
Extra step 2	24
Android Studio and our project – A very brief guided tour	24
The Project panel	25
The Editor window	26
Locking the game to full-screen and landscape orientation	28
Deploying the game so far	29
Running the game on an Android emulator	30
Running the game on a real device	32
Summary	33

Chapter 2: Java: First Contact	35
Planning the Sub' Hunter game	35
Actions flowchart/diagram	38
Code comments	39
Mapping out our code using comments	40
Introduction to Java methods	43
Overriding methods	45
Structuring Sub' Hunter with methods	45
Introduction to Object Oriented Programming	47
Classes and Objects	48
The important thing about OOP, Classes, and Objects	48
Classes, objects, and instances	48
Final word on OOP, Classes, and Objects – for now	49
Using Java packages	51
Adding classes by importing packages	52
Linking up our methods	53
Summary	59
Chapter 3: Variables, Operators and Expressions	61
Handling syntax and jargon	61
Java Variables	62
Different types of variables	64
Primitive types	64
Reference variables	66
How to use variables	68
Declaring variables	68
Initializing variables	68
Making variables useful with operators	69
Most used operators in this book	69
Casting	71
Concatenation	71
Declaring and Initializing the Sub' Hunter Variables	72
Planning the variables	72
Declaring the variables	73
Handling different screen sizes and resolutions	74
Handling different screen resolutions part 1: Initialising the variables	76
Errors, warnings, and bugs	79
Printing Debugging Information	80
Testing the game	81
Summary	82

Chapter 4: Structuring Code with Java Methods	83
Methods	83
Methods revisited and explained further	84
The method signature	85
Modifier	87
Return type	88
A closer look at method names	89
Parameters	90
Doing things in the method body	91
Method Overloading by Example	92
Creating a new project	92
Coding the method overloading mini-app	93
Running the method overloading mini-app	95
Scope: Methods and Variables	96
Revisiting the code and methods we have used already	97
Generating random numbers to deploy a sub	98
The Random class and the nextInt method	98
Adding Random based code to newGame	99
Testing the game	100
Summary	101
Chapter 5: The Android Canvas Class – Drawing to the Screen	103
Understanding the Canvas class	103
Getting started drawing with Bitmap, Canvas, and ImageView	104
Canvas and Bitmap	104
Paint	104
ImageView and Activity	105
Canvas, Bitmap, Paint and ImageView quick summary	105
Using the Canvas class	106
Preparing the objects of classes	106
Initializing the objects	107
Setting the Activity content	107
Canvas Demo app	108
Creating a new project	108
Coding the Canvas demo app	109
Drawing on the screen	111
Android Coordinate system	113
Plotting and drawing	114
Drawing the Sub' Hunter graphics and text	115
Preparing to draw	115
Initializing a Canvas, Paint, ImageView, and Bitmap	116
Drawing some grid lines	118

Drawing the HUD	120
Upgrading the printDebuggingText method	122
Summary	124
Chapter 6: Repeating Blocks of Code with Loops	125
Decisions, decisions	125
Keeping things tidy	126
More operators	126
Loops	128
While loops	129
Breaking out of a loop	130
Do while loops	131
For loops	132
Nested loops	132
Using for loops to draw the Sub' Hunter grid	133
Summary	136
Chapter 7: Making Decisions with Java If, Else and Switch	137
If they come over the bridge shoot them	137
Else do this instead	138
Switching to make decisions	141
Switch example	142
Combining different control flow blocks	143
Continue...	144
Making sense of the screen touches	144
Coding the onTouchEvent method	147
Final tasks	149
Coding the takeShot method	149
Explaining the takeShot method	150
Coding the boom method	152
Drawing the shot on the grid	153
Running the game	155
Summary	156
Chapter 8: Object-Oriented Programming	157
Basic Object-Oriented Programming	157
Humans learn by doing	158
Introducing OOP	158
Encapsulation	159
Inheritance	159
Polymorphism	160
Why we do it like this?	160
Class recap	161

Looking at the code for a class	161
Class implementation	162
Declaring, initializing and using an object of the class	162
Basic classes mini-app	166
Creating your first class	166
More things we can do with our first class	170
Remember that encapsulation thing?	171
Controlling class use with access modifiers	172
Class access in a nutshell	173
Controlling variable use with access modifiers	173
Variable access summary	174
Methods have access modifiers too	174
Method access summary	175
Accessing private variables with getters and setters	176
Setting up our objects with constructors	179
Using "this"	181
Static methods	181
Encapsulation and static methods mini-app	183
OOP and inheritance	188
Inheritance mini-app	190
Polymorphism	194
Abstract classes	195
Interfaces	197
Starting the Pong game	198
Planning the Pong game	199
Setting up the Pong project	199
Summary	200
Chapter 9: The Game Engine, Threads, and The Game Loop	201
Coding the PongActivity class	202
Coding the PongGame class	205
Coding the PongGame class	207
Adding the member variables	208
Coding the PongGame constructor	211
Coding the startNewGame method	212
Coding the draw method	213
Adding the printDebuggingText method	214
Understanding the draw method and the SurfaceView class	215
The game loop	217
Threads	219
Problems with threads	219
Java try, catch exception handling	223

Implementing the game loop with a thread	224
Implementing Runnable and providing the run method	224
Coding the thread	225
Starting and stopping the thread	225
Activity lifecycle	226
A simplified explanation of the Android lifecycle	227
Lifecycle phases: What we need to know	227
Lifecycle phases: What we need to do	229
Using the Activity lifecycle to start and stop the thread	230
Coding the run method	230
Running the game	234
Summary	235
Chapter 10: Coding the Bat and Ball	237
The Ball Class	237
Communicating with the game engine	239
Representing rectangles and squares with RectF	239
Coding the variables	240
Coding the Ball constructor	240
Coding the RectF getter method	241
Coding the Ball update method	242
Coding the Ball helper methods	243
Coding a realistic-ish bounce	245
Using the Ball class	247
The Bat class	250
Coding the Bat variables	251
Coding the Bat constructor	252
Coding the Bat helper methods	254
Coding the Bat's update method	254
Using the Bat Class	256
Coding the Bat input handling	257
Running the game	259
Summary	259
Chapter 11: Collisions, Sound Effects and Supporting Different Versions of Android	261
Handling collisions	262
Collision detection options	262
Rectangle intersection	262
Radius overlapping	264
Crossing number algorithm	265

Optimizations	266
Multiple hitboxes	266
Neighbour checking	266
Best options for Pong	267
The RectF intersects method	268
Handling different versions of Android	268
Detecting the current Android version	269
The Soundpool class	269
Initializing SoundPool the new way	270
Java Method chaining explained	270
Back to initializing SoundPool (the new way)	270
Initializing SoundPool the old way	271
Loading sound files into memory	272
Playing a sound	273
Stopping a sound	274
Generating sound effects	274
Adding sound to the Pong game	277
Adding the sound variables	277
Initializing the SoundPool	278
Coding the collision detection and playing sounds	280
The bat and the ball	280
The four walls	281
Playing the game	282
Summary	283
Chapter 12: Handling Lots of Data with Arrays	285
Planning the project	286
Starting the project	287
Change the AndroidManifest.xml to lock in landscape and full-screen	288
Creating the classes	288
Reusing the Pong engine	289
Coding the BulletHellActivity	289
Coding the BulletHellGame class	290
Coding the member variables	291
Coding the BulletHellGame constructor	292
Coding the BulletHellGame methods	293
Coding draw and onTouchEvent	294
Coding pause, resume, and printDebuggingText	295
Testing the Bullet Hell engine	296
Coding the Bullet class	297
Spawning a bullet	300
Java Arrays	302
Arrays are objects	304

Simple array example mini-app	305
Getting dynamic with arrays	307
Dynamic array example	307
Entering the nth dimension with Arrays	309
Multidimensional Array mini app	309
Array out of bounds exceptions	313
Spawning an array of bullets	313
Running the game	318
Summary	319
Chapter 13: Bitmap Graphics and Measuring Time	321
The Bob (player's) class	321
Adding the Bob graphic to the project	322
Coding the Bob class	322
Using the Bob class	326
Adding Bob to the collision detection	327
Drawing Bob to the screen	329
Adding the sound effects	330
Activating Bob's teleport	331
Coding the printDebuggingText method	332
Coding the spawnBullet method (again)	333
Coding the startGame method	335
Running the game	336
Summary	336
Chapter 14: The Stack, the Heap, and the Garbage Collector	339
Managing and understanding memory	340
Variables revisited	340
The stack and the heap	341
A quick break to throw out the trash	342
Stack and Heap quick summary	343
Introduction to the Snake game	343
Looking ahead to the Snake game	345
Getting started with the Snake game	345
Make full screen and landscape	345
Adding some empty classes	346
Coding SnakeActivity	346
Adding the sound effects	348
Coding the game engine	348
Coding the members	348
Coding the constructor	350
Coding the newGame method	352
Coding the run method	352
Coding the updateRequired method	353

Coding the update method	354
Coding the draw method	355
Coding onTouchEvent	356
Coding pause and resume	357
Running the game	358
Summary	358
Chapter 15: Android Localization -Hola!	359
Making the snake game Spanish, English and German	359
Adding Spanish support	360
Adding German support	360
Add the string resources	361
Amending the code	362
Run the game in German or Spanish	363
Summary	364
Chapter 16: Collections, Generics and Enumerations	365
Adding the graphics	365
Coding the apple	366
The Apple constructor	367
Using the apple	369
Running the game	370
Using Arrays in the snake game	371
ArrayLists	371
The enhanced for loop	373
Arrays and ArrayLists are polymorphic	373
Enumerations	375
Summary	377
Chapter 17: Manipulating Bitmaps and Coding the Snake class	379
Rotating Bitmaps	379
What is a Bitmap exactly	380
The Matrix class	381
Inverting the head to face left	381
Rotating the head to face up and down	383
Add the sound to the project	384
Coding the Snake class	384
Coding the constructor	386
Coding the reset method	388
Coding the move method	389
Coding the detectDeath method	391
Coding the checkDinner method	392
Coding the draw method	393

Coding the switchHeading method	395
Using the snake class and finishing the game	396
Running the completed game	399
Summary	400
Chapter 18: Introduction to Design Patterns and much more!	403
Introducing the Scrolling Shooter project	404
Game programming patterns and the structure of the Scrolling Shooter project	407
Starting the project	408
Editing the manifest	409
Code the GameActivity class	409
Getting started on the GameEngine class	410
Controlling the game with a GameState class	413
Passing GameState from GameEngine to other classes	414
Communicating from GameState to GameEngine	414
Giving partial access to a class using an interface	415
Interface refresher	415
What we will do to implement the interface solution	416
1. Coding the new interface	416
2. Implementing the interface	416
3. Passing a reference to the interface into the class that needs it	417
Coding the GameState class	418
Saving and loading the high score- forever	419
4. Pressing the "special button" - Calling the method of the interface	422
Finishing off the GameState class	422
Using the GameState class	425
Building a sound engine	426
Adding the sound files to the project	426
Coding the SoundEngine class	427
Using the SoundEngine class	428
Testing the game so far	429
Building a HUD class to display control buttons and text	429
Coding the prepareControls method	431
Coding the draw method of the HUD class	433
Coding drawControls and getControls	435
Building a Renderer class to handle the drawing	436
Using the HUD and Renderer classes	438
Running the game	439
Summary	440

Chapter 19: Listening with the Observer Pattern, Multitouch and Building a Particle System	441
The Observer pattern	442
The Observer pattern in the Scrolling Shooter project	442
Coding the Observer pattern in Scrolling Shooter	444
Coding the Broadcaster interface	444
Coding the InputObserver interface	445
Making GameEngine a Broadcaster	445
Coding a Multitouch UI controller and making it a listener	446
Coding the required handleInput method	447
Using the UIController	450
Running the game	451
Implementing a particle system explosion	452
Coding the Particle class	453
Coding the ParticleSystem class	455
Adding a particle system to the game engine and drawing it with the Renderer	459
Building a physics engine to get things moving	462
Running the game	464
Summary	465
Chapter 20: More Patterns, a Scrolling Background and Building the Player's ship	467
Meet the game objects	468
Reminder of how all these objects will behave	468
The Entity-Component pattern	469
Why lots of diverse object types are hard to manage	469
The first coding nightmare	469
Using a generic GameObject for better code structure	471
Composition over inheritance	472
The Simple Factory pattern	473
At last some good news	475
Summary so far	476
The object specifications	477
Coding the ObjectSpec parent class	477
Coding all the specific object specifications	479
AlienChaseSpec	479
AlienDiverSpec	481
AlienLaserSpec	481
AlienPatrolSpec	482
BackgroundSpec	483
PlayerLaserSpec	484

PlayerSpec	485
Coding the component Interfaces	485
GraphicsComponent	486
InputComponent	486
MovementComponent	487
SpawnComponent	487
Coding the player's and the background's empty component classes	488
StdGraphicsComponent	488
PlayerMovementComponent	489
PlayerSpawnComponent	490
PlayerInputComponent and the PlayerLaserSpawner interface	490
LaserMovementComponent	492
LaserSpawnComponent	492
BackgroundGraphicsComponent	493
BackgroundMovementComponent	494
BackgroundSpawnComponent	494
Every GameObject has a transform	495
Every object is a GameObject	502
Completing the player's and the background's components	506
The player's components	506
Completing the StdGraphicsComponent	506
Completing the PlayerMovementComponent	508
Completing the PlayerSpawnComponent	510
Completing the PlayerInputComponent	511
Completing the LaserMovementComponent	515
Completing the LaserSpawnComponent	517
Coding a scrolling background	518
Completing the BackgroundGraphicsComponent	519
Completing the BackgroundMovementComponent	523
Completing the BackgroundSpawnComponent	524
GameObject/Component reality check	524
Building the GameObjectFactory	525
Coding the Level class	531
Putting everything together	534
Updating GameEngine	534
Updating PhysicsEngine	537
Updating Renderer	538
Running the game	539
Summary	540
Chapter 21: Completing the Scrolling Shooter Game	541
Adding the alien's components	541
AlienChaseMovementComponent	542

Code AlienLaserSpawner	548
Implement the Interface in GameEngine	548
AlienDiverMovementComponent	549
AlienHorizontalSpawnComponent	551
AlienPatrolMovementComponent	552
AlienVerticalSpawnComponent	556
Spawning the Aliens	557
Updating GameEngine	557
Updating Level	558
Updating GameObjectFactory	559
Running the game	560
Detecting collisions	560
Running the completed game	564
Summary	564
Chapter 22: Exploring More Patterns and Planning the Platformer Project	565
Platform Game: Bob Was in A Hurry	566
How we will build the platformer	569
Level design files example	569
The graphics	571
Cameras and the real world	572
The slightly modified ObjectSpec	573
New improved transform hierarchy	574
Project patterns	574
BitmapStore	575
Levels and LevelManager	575
Getting started with Bob was in a hurry	576
Creating the project and adding the assets	576
Specifying all the game objects with GameObjectSpec classes	577
GameObjectSpec	578
BackgroundCitySpec	580
BackgroundMountainSpec	581
BackgroundUndergroundSpec	582
BrickTileSpec	582
CartTileSpec	583
CoalTileSpec	583
CollectibleObjectSpec	584
ConcreteTileSpec	585
DeadTreeTileSpec	585
FireTileSpec	586
GrassTileSpec	586

LamppostTileSpec	588
MoveablePlatformSpec	588
ObjectiveTileSpec	589
PlayerSpec	589
ScorchedTileSpec	590
SnowTileSpec	591
SnowyTreeTileSpec	591
StalactiteTileSpec	592
StalagmiteTileSpec	592
StonePileTileSpec	593
Summary of component classes	593
Coding the component interfaces	593
GraphicsComponent	594
UpdateComponent	595
Coding the other interfaces	595
EngineController	596
GameEngineBroadcaster	596
InputObserver	596
Summary	597
Chapter 23: The Singleton Pattern, Java HashMap, Storing Bitmaps Efficiently and Designing Levels	599
The Singleton pattern	601
The Singleton code	601
More Java Collections – Meet Java Hashmap	604
The memory problem and the BitmapStore	606
Coding the BitmapStore class	606
Coding the basic transform	611
Coding the inanimate and decorative components	615
DecorativeBlockUpdateComponent	615
InanimateBlockGraphicsComponent	615
InanimateBlockUpdateComponent	618
Create the levels	619
Level	620
LevelCity	620
LevelMountains	622
LevelUnderground	623
Coding a stripped down GameObjectFactory	624
Coding a slightly commented-out game object	627
Coding the GameState	629
Code the SoundEngine	635

Coding the physics engine (without collision)	638
Coding a Renderer	639
Coding the camera	640
Testing the formulas with hypothetical coordinates	644
Coding the Hud	647
Coding the UIController class	652
Code the Activity	654
Code the GameEngine	655
Coding the LevelManager class	660
Running the game	666
Summary	668
Chapter 24: Sprite-sheet animations, Controllable Player and Parallax Scrolling Backgrounds	669
Animating sprite-sheets	670
Coding the Animator	670
Coding the player's components and transform	675
PlayerTransform	675
AnimatedGraphicsComponent	681
PlayerInputComponent	684
PlayerUpdateComponent	688
Coding a parallax background	693
Coding the BackgroundTransform	694
Coding the BackgroundGraphicsComponent	695
Coding the BackgroundUpdateComponent	698
Updating the levelManager, GameObjectFactory and GameObject	699
Running the game	706
Summary	706
Chapter 25: Intelligent Platforms and Advanced Collision Detection	707
Coding the moving platform component class	707
MoveableBlockUpdateComponent	708
Update LevelManager and GameObjectFactory	710
Running the game	711
Coding the detectCollisions method: Part 1	711
Explaining part 1	713
Coding the detectCollisions method: Part 2	716
Explaining part 2	718
Running the game	721
Summary	721

Table of Contents

Chapter 26: What next?	723
Publishing	723
Using the assets from the book	724
Future learning	724
My other channels	725
Thanks	725
Other Books You May Enjoy	727
Index	731

Preface

About the book

Android is one of the most popular mobile operating systems presently. It uses the most popular programming language, Java, as the primary language for building apps of all types. However, this book is unlike other Android books in that it doesn't assume that you already have Java proficiency.

This new and expanded second edition of *Learning Java by Building Android Games* shows you how to start building Android games from scratch. The difficulty level will grow steadily as you explore key Java topics, such as variables, loops, methods, object-oriented programming, and design patterns, including code and examples that are written for Java 9 and Android P.

At each stage, you will put what you've learned into practice by developing a game. You will build games such as Minesweeper, Retro Pong, Bullet Hell, and Classic Snake and Scrolling Shooter games. In the later chapters, you will create a time-trial, open-world platform game.

By the end of the book, you will not only have grasped Java and Android but will also have developed six cool games for the Android platform.

Who this book is for

Learning Java by Building Android Games is for you if you are completely new to Java, Android, or game programming, and you want to make Android games. This 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, In this first chapter we will get straight into Java, Android and game development. Also, we will look at some images and an outline of each of the six games we will develop throughout the book. Further, we will explore and discover what is so great about Android, what exactly Android and Java are, 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, In this chapter, we will make significant progress with the Sub' Hunter game even though this is the first lesson on Java. We will look 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 an even briefer first glimpse at **object-oriented programming** that will begin to reveal the power of Java and the Android API.

Chapter 3, Variables, Operators, and Expressions, We are going to make good progress in this chapter. We will learn about Java variables that allows us to give our game the data it needs. Things like 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, As we are starting to get comfortable with Java programming, in this chapter, we will take a closer look at **methods** because although we know that you can call them to make them execute their code, there is more to them that haven't been discussed so far.

Chapter 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 about 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 some related classes like `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, In this brief chapter, we will learn about Java loops that 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 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, Welcome to the final part of the first game. By the end of this chapter, you can say you have learned most of the Java basics. In this chapter, we will 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, 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 the 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. Topics including **encapsulation**, **inheritance**, and **polymorphism**. We will start to write our own reusable classes and look at some more advanced OOP like **abstract classes** and **interfaces**. At 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, During 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**, 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. Furthermore, this game engine code will be used as an approximate template (we will improve it each project) for future projects making the realization of future games faster and easier.

Chapter 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 in 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.

Chapter 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 in to practice near the end of the chapter. We will also learn about how we can detect and handle different versions of Android. We will then be able 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 on the game.

Chapter 12, Handling Lots of Data with Arrays, In this chapter, we will first get the Bullet Hell project setup 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.

Chapter 13, Bitmap Graphics and Measuring Time, So far in this book, we have drawn exclusively with primitive shapes and text. In this section, 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 a **HUD**, measure the time, and code a solution to remember the longest (best) time.

Chapter 14, 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 the 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!, 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 hard-coding Strings in our code directly as we have done so far throughout the book.

Chapter 16, Collections, Generics, 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 to new Java concepts `ArrayList` and enumerations (`enum` 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, In this chapter, we will finish the Snake game and make it fully playable. We will put what we learned about `ArrayList` and `enum` 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 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!, Since the second project, we have been using objects. You might have noticed that many of the objects have things in common. Things like variables for speed and direction, a `RectF` for handling collisions and more besides.

As our objects have more in common we should start taking advantage of OOP, inheritance, polymorphism and a 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 disorganised 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.

Chapter 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 of 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- Ship, This 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 of it 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, 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 alien and their 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, Exploring More Patterns and Planning the Platformer Project, Welcome to the start of the final project. Over the next four chapters we will build a platform game packed full of features like parallax effect scrolling, animated controllable character, multiple different and challenging levels and much more (keep reading for more details.) During this project we will discover more Java patterns like the **Singleton**, another Java data class, the **HashMap** and explore the gaming concepts of a **controllable camera** and an **animator**. At the same time, we will reinforce all the Java concepts we have already learned including aiding our understanding of and improving upon the entity-component/factory patterns we used in the previous project.

Chapter 23, The Singleton Pattern, Java HashMap, Storing Bitmaps Efficiently, and Designing Levels, This is going to be a very busy and varied chapter. We will learn the theory of the **Singleton** design pattern. We will be introduced to another of the classes of the Java Collections, `HashMap` in which we will see how we can more efficiently store and make available the wide variety of bitmaps that are required for this project. We will also get started on our new and improved `Transform` class, code the first of the component-based classes, code the all-new `Camera` class and make a significant start on some of the more familiar classes, `GameState`, `PhysicsEngine`, `Renderer`, `GameEngine` and more besides.

Chapter 24, Sprite Sheet Animations, the Controllable Player, and Parallax Scrolling Backgrounds, In this chapter we will look at how basic animation occurs by "flipping" through different images to create a moving effect. This is the same technique used to make a simple cartoon. Of course, we will then need to code a solution to achieve this simple animation in our game objects and will do so by coding an `Animator` class and adding one to any graphics-related component class that needs it. We will also implement another scrolling background but slightly differently to the scrolling background in the previous project because this time it will need to be positioned both horizontally and vertically, each frame, relative to the position of the camera.

Chapter 25, Intelligent Platforms and Advanced Collision Detection, This chapter is the final chapter of the final project and shorter than most. We will however, add quite a bit to the game. First, we will code an **intelligent platform**- one that moves up and down of its own accord. This greatly increases the potential for interesting level designs and looks good too. The final thing we will add is the collision detection. This will be more advanced than in any of the other projects as we will need to respond differently depending on which collider of the player collide with various different game objects. In addition, the collision code will need to communicate with the movement code to make the whole system behave as needed.

Chapter 26, 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.

To get the most out of this book

- To succeed with this book, you don't need any experience whatsoever. If you are confident with your operating system of choice (Windows, Mac or Linux) you can learn to make Android games using the Java programming language. Learning to develop professional quality games is a journey that anybody can embark upon and stay on for as long as they want.
- If you do have previous, programming (Java or any other language), Android or game development experience then you will make faster progress with the earlier chapters.

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at <http://www.packtpub.com>.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen 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
- Zipeg / 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-Second-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!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://www.packtpub.com/sites/default/files/downloads/LearningJavabyBuildingAndroidGamesSecondEdition_ColorImages.pdf.

Conventions used

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

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

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

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

[ Warnings or important notes appear like this.]

[ Tips and tricks appear like this.]

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@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, <http://www.packtpub.com/submit-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@packtpub.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 <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 packtpub.com.

1

Java, Android and Game Development

Welcome to Learning Java by Building Android Games (second edition). In this first chapter we will get straight into Java, Android and game development and by the end, we will have a great insight into what we will achieve in the book as well as have built and deployed the first part of the first game.

Also, we will look at some images and an outline of each of the six games we will develop throughout the book.

Further, 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 we can build and deploy the outline for the first game.

In summary this chapter will cover:

- Why choose the combination of Java, Android, and games?
- Look at the six 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.

What's new in the second edition?

Everything! The Java theory has been reworked and reorganized so if you read the first edition you might recognize a few of the words to be similar but all the games are new. Even the Snake and Pong games which made an appearance in the first edition have totally reworked code now.

This second edition goes much deeper than the previous, into crucial topics like object-oriented programming and goes even further to cover more advanced topics like Java design patterns, loading game levels from files and Java collections.

More game specific topics are covered as well. Collision detection, physics and huge scrolling game worlds using a movable camera are just some of the additions.

Also, more Android topics are covered, including handling different screen sizes/resolutions and input with multiple fingers. And as you would expect if you had read the first edition, everything is put into practice using a game.

It is hard to know which topics should be highlighted in this summary of contents as so many are covered and I urge potential readers to read ahead in this chapter to see the wide range of games we will build together and then look at the table of contents to see the full scope of the Java topics you will learn at the same time.

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. But quite quickly, through diverse handset offers that struck a chord with both the practical price-conscious as well as the fashion-conscious and tech-savvy, Android user numbers exploded.

For many, myself included, developing Android games is the most rewarding pastime and business bar none.

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. Any programming can be fun, and I have been programming all my life, but creating games, especially for Android is somehow extraordinarily rewarding.

Defining exactly why this is the case is quite difficult. Maybe it is the fact that the platform is free and open. You can distribute your games without needing the permission of a big controlling corporation - nobody can stop you. And at the same time, you have the well-established, corporate controlled mass markets like Amazon App Store and Google Play.

More likely, the reason developing Android games gives such a buzz is the nature of the devices themselves. They are deeply personal. You can develop games which interact with people's lives. Educate, entertain, tell a story, etc. But it is there in their pocket ready to play in the home, the workplace or on holiday.

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

No longer is developing games considered geeky, nerdy or reclusive. In fact, developing Android games is considered highly skillful and the most successful are hugely admired, even revered.

If all this fluffy and spiritual stuff doesn't mean anything to you then that's fine too; developing for Android can make 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 are 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 ever. 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? Obviously, 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?

Java: The first stumbling block

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

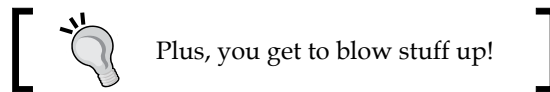
Android uses Java to make games. Every Android book, even those aimed at so-called beginners assumes readers to have at least an intermediate level of Java, and most require an advanced level. So good-to-excellent Java knowledge was a prerequisite for learning Android.

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 into the world of Android anyway.

To add to this that 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.

But it doesn't need to be like this. In this book I have carefully placed all the Java topics you would learn in a thick and weighty beginner's tomb and reworked them into six games, starting from the incredibly simple through to an open-world 2D platformer.

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 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-games Android environments. And that's what this book is about.

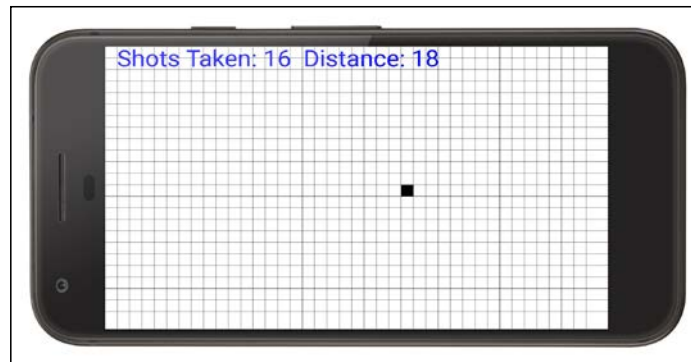


The games you will build

Let's look at some of the screen-shots and get a little bit more detail about each of the games from the book. We will go in to further detail and explanation as we start each project.

Sub' Hunter

The first game we build will allow us to introduce some key Java beginner's topics. Code comments, variables, operators, methods, loops, generating random numbers, `if`, `else`, `switch` and a brief introduction to object-oriented programming. We will also see how to communicate with the Android OS, detect screen touches, draw simple graphics, detect the screen resolution and handle different screen sizes. All this will be used to build a simpler variation of the classic Minesweeper type game. Here is one of the screens from that 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 60 frames per second smoothly animated Pong clone. The Java topics covered include classes, interfaces, threads, try-catch blocks, method overloading vs. overriding and a much deeper look at object-oriented programming including writing and using our own classes. This project will also leave the reader competent with understanding the game loop, simple bouncing physics, playing sound effects and detecting game object collisions. Here is a picture of the simple but still a step-up Pong game.



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

Bullet Hell

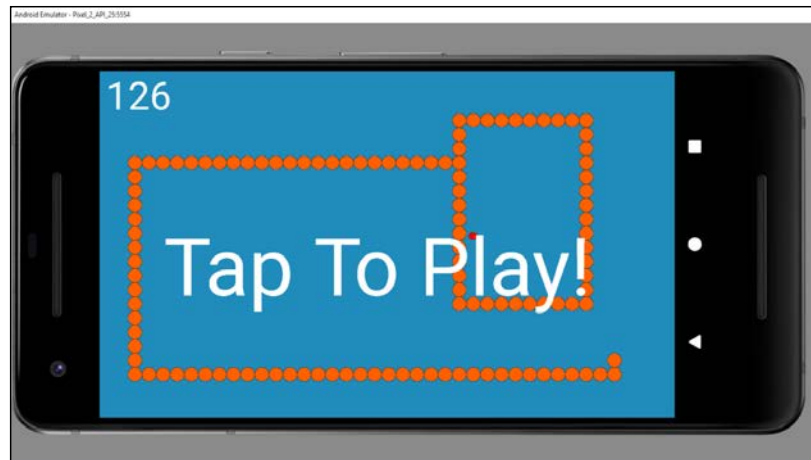
In this project, we will meet Bob. In this Bullet Hell game, Bob will be a static image that can teleport at will anywhere on the screen to avoid the ever-growing number of bullets. In the final game we will also animate Bob, so he can run and jump around an explorable scrolling world. This short implementation will enable us to learn about Java arrays, Bitmap graphics and measuring time. We will also see 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.



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, but 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 gaining points by eating apples. Each time you eat an apple your snake gets longer and harder to keep alive. In this project we learn about Java `ArrayList`, enhanced `for` loop, the Stack, the Heap and the Garbage collector (seriously- it's a thing) and we will also see how to make our games multilingual. Hola!

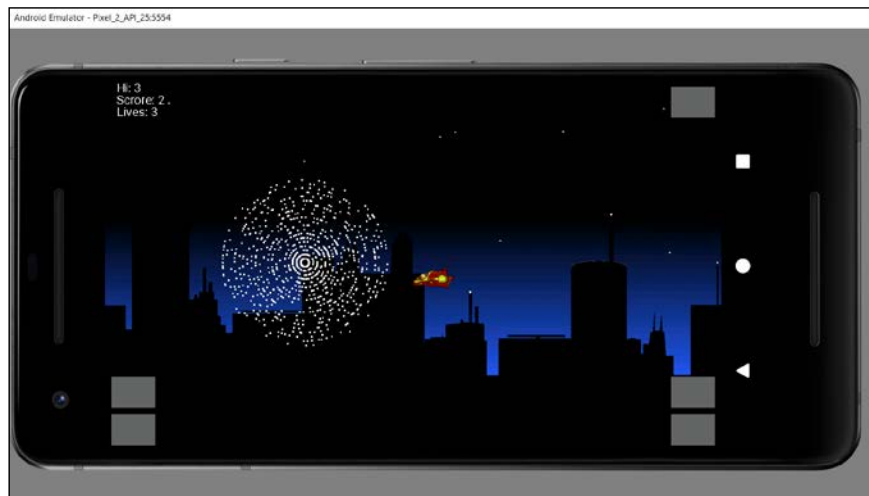


The game ends when the snake bumps into the edge of the screen or ends up eating part of his own body.

Scrolling Shooter

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

In this project we are introduced 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, Strategy pattern, Factory pattern and Observer pattern.



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 and yet keep the code manageable.

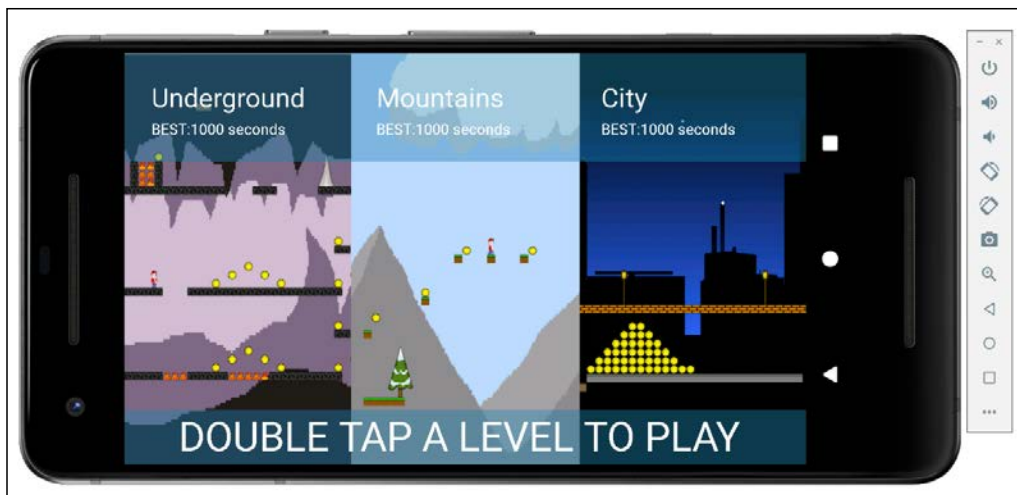
Open-World Platformer

In this project Bob makes a second appearance. The game will be a time trial where the player must get from the start point to the exit in the fastest time possible. There will be deadly jumps, moving platforms to navigate, fire pits and collectible coins. Every coin the player fails to collect will add a time penalty.

In this project we learn about the concept of a moveable camera that tracks the part of the game-world that needs to be shown to the player at any given moment. We will also learn some new Java patterns and reinforce the vital knowledge from the previous chapter. Furthermore, we will see how we can design levels as text layouts and then load them in code as playable levels.



There are three levels and the player will choose the level they want to play from a home-screen which will also show the fastest times for each level.



Let's learn a little about how Java and Android work.

How Java and Android work

After we write a game in Java for Android, we click on a button in Android Studio to change our code into another form, a form that is understood by Android devices. We call this "other form" Dalvik EXecutable (**DEX**) **code**, and the transformation process is called **compiling**.

Compiling takes place on the development machine after we click on that button. We will see this work right after we set up our development environment in a minute.

Android is a complex system, but you do not need to understand it in depth, in fact, it is designed to hide the complexity, all operating systems are, and Android does this better than most.

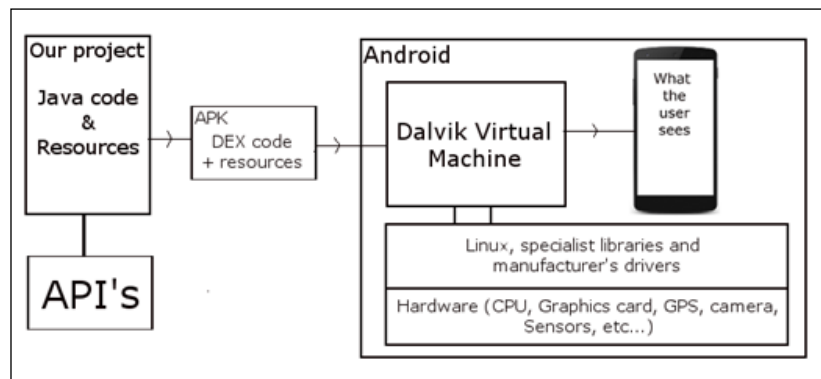
The part of the Android system that **executes** (runs) our compiled DEX code is called the **Dalvik Virtual Machine** (DVM). The DVM itself is a piece of software that runs on a specially adapted version of the Linux operating system. So, what the user sees of Android, is, Android just as an app running on yet another operating system. Therefore the apps and games that we write are apps running on top of the app which is Android itself.

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

The DVM itself must have access to the hardware, which it does, but this access must be programmer friendly and easy to use. The way the DVM allows us access is indeed easy to use because of the Android Application Programming Interface (**API**).

This API is primarily designed to use with Java. In fact, most of the Android API, is itself, Java code. As I may have mentioned already, this makes Android games the most fun and thorough way to learn Android, Java and game development.

If you want to see the relationship between The Android API, DEX code, DVM and an Android device, look at this picture.



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

Let's set up Android Studio.

Setting up Android Studio

Setting up Android Studio is quite straightforward if a little lengthy. Grab some refreshment and get started with the following steps.

1. Visit developer.android.com/studio/index.html. Click the big green button to proceed.

Android Studio

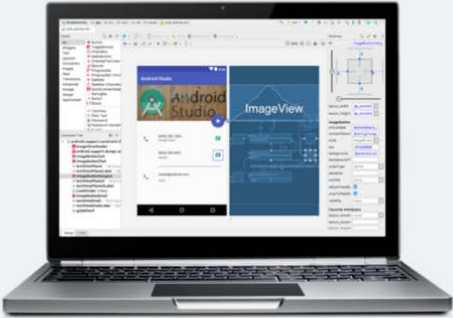
The Official IDE for Android

Android Studio provides the fastest tools for building apps on every type of Android device.

World-class code editing, debugging, performance tooling, a flexible build system, and an instant build/deploy system all allow you to focus on building unique and high quality apps.

DOWNLOAD ANDROID STUDIO
3.0.1 FOR WINDOWS (583 MB)

[Read the docs](#) [See the release notes](#)



2. Accept the terms and conditions by checking the box and then click the big blue button **DOWNLOAD ANDROID STUDIO FOR WINDOWS**.

Download Android Studio

Before downloading, you must agree to the following terms and conditions.

Terms and Conditions

This is the Android Software Development Kit License Agreement

1. Introduction

1.1 The Android Software Development Kit (referred to in the License Agreement as the "SDK" and specifically including the Android system files, packaged APIs, and Google APIs add-ons) is licensed to you subject to the terms of the License Agreement. The License Agreement forms a legally binding contract between you and Google in relation to your use of the SDK.

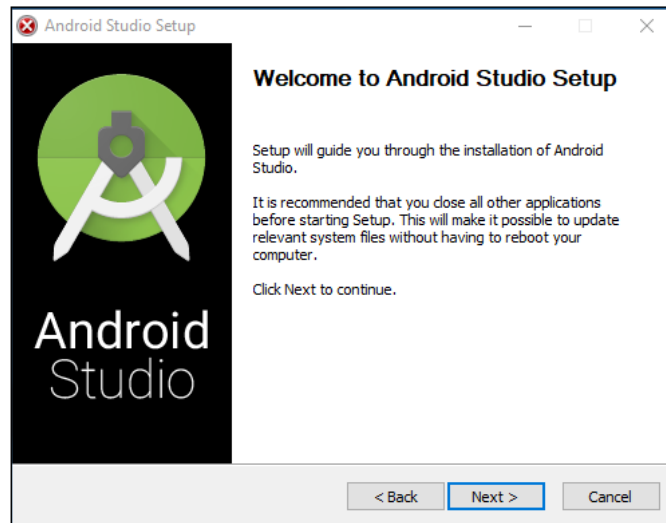
1.2 "Android" means the Android software stack for devices, as made available under the Android Open Source Project, which is located at the following URL: <http://source.android.com/>, as updated from time to time.

1.3 A "compatible implementation" means any Android device that (i) complies with the Android Compatibility Definition document, which

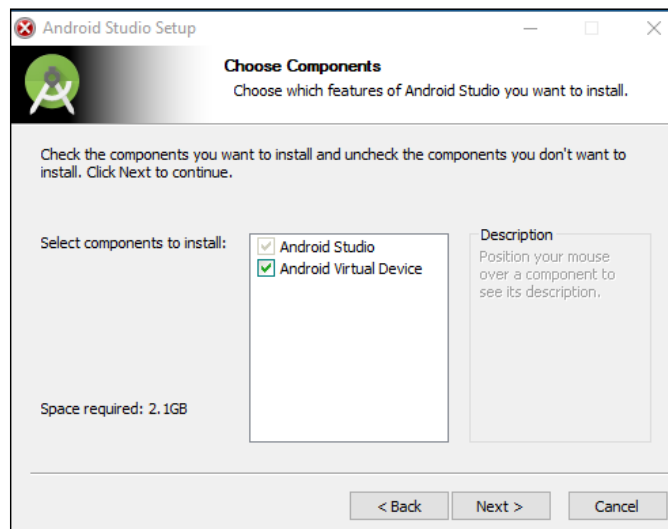
☒ I have read and agree with the above terms and conditions

DOWNLOAD ANDROID STUDIO FOR WINDOWS

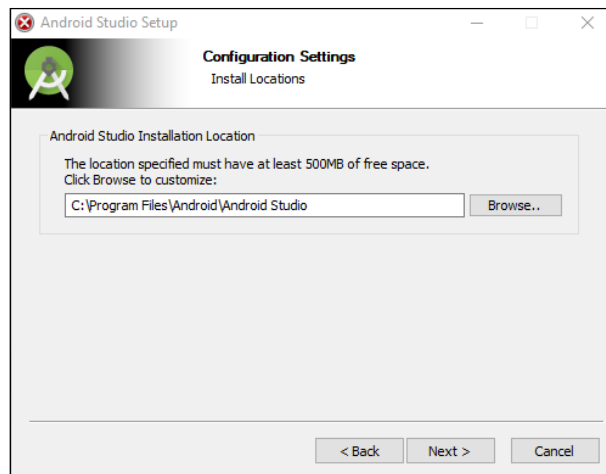
3. When the download is completed run the file you just downloaded. It has a name that starts android-studio-ide... the end of the name of the file will vary based on the current version at time of reading.
4. Click the **Next** button to proceed.



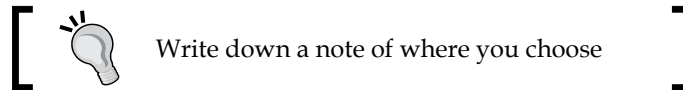
5. Leave the default options selected as shown next and click the **Next** button.



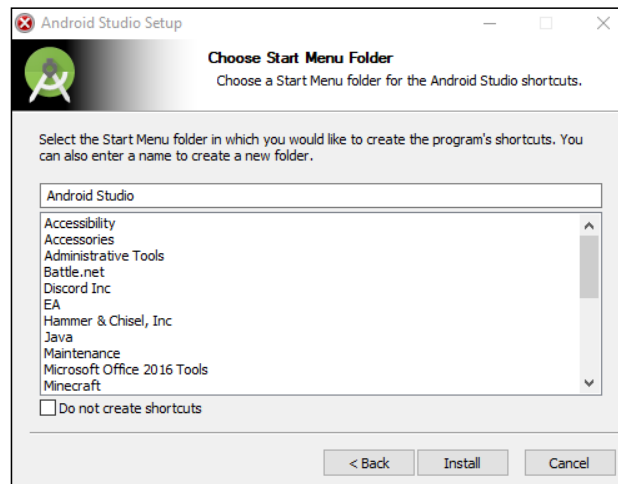
6. Next, we need to choose where to install Android Studio as shown in the next image.



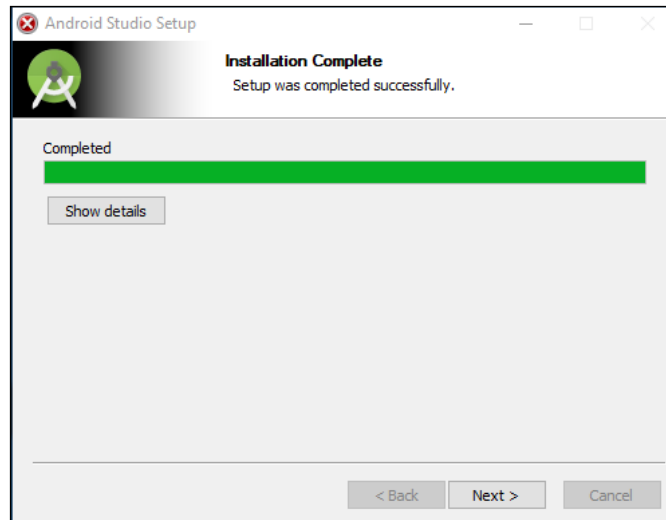
The install wizard recommends 500mb, however there are more requirements later in the install process. It is much easier if you have all your Android Studio parts as well as your project files on the same hard drive. I recommend having at least 2gb of free space. If you need to switch drives to accommodate this, then use the **Browse..** button to browse to a suitable place on your hard drive.



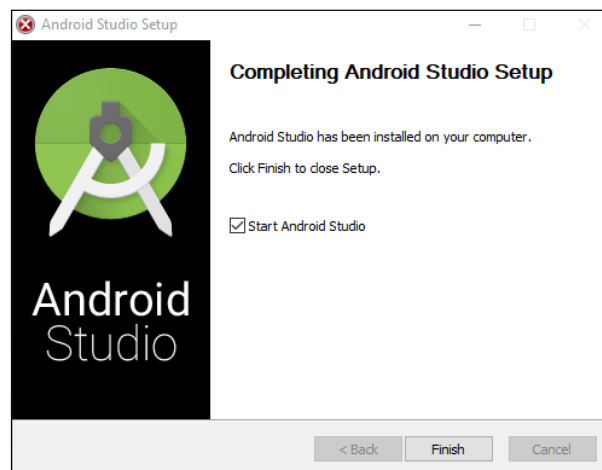
7. When you are ready click the **Next** button.
8. In this next window you are choosing the folder in your start menu where Android Studio will appear. Leave it at the default as shown next.



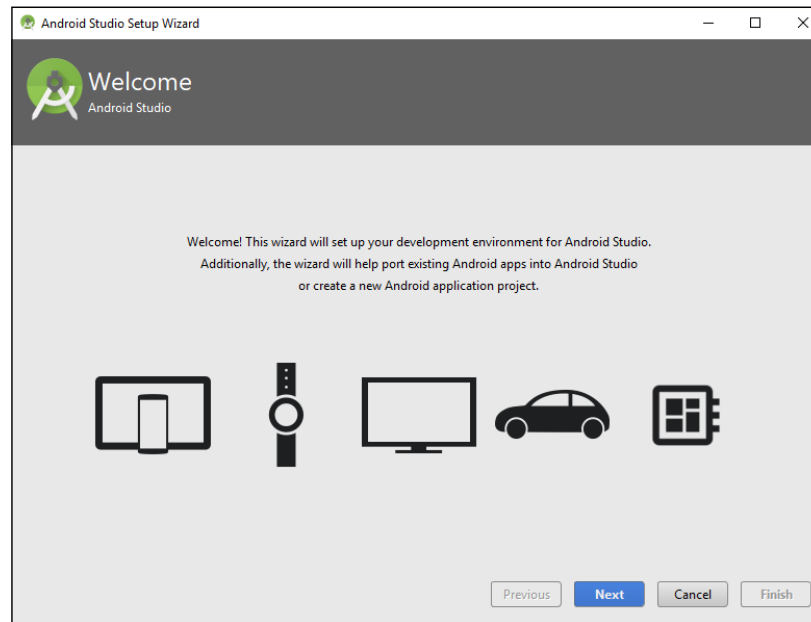
9. Click **Install**.
10. This step might take some time, especially on older machines or if you have a slow Internet connection. When this stage is done you will see this screen.



11. Click **Next**.
12. Android Studio is now installed, check the **Start Android Studio** check-box and click the **Finish** button.

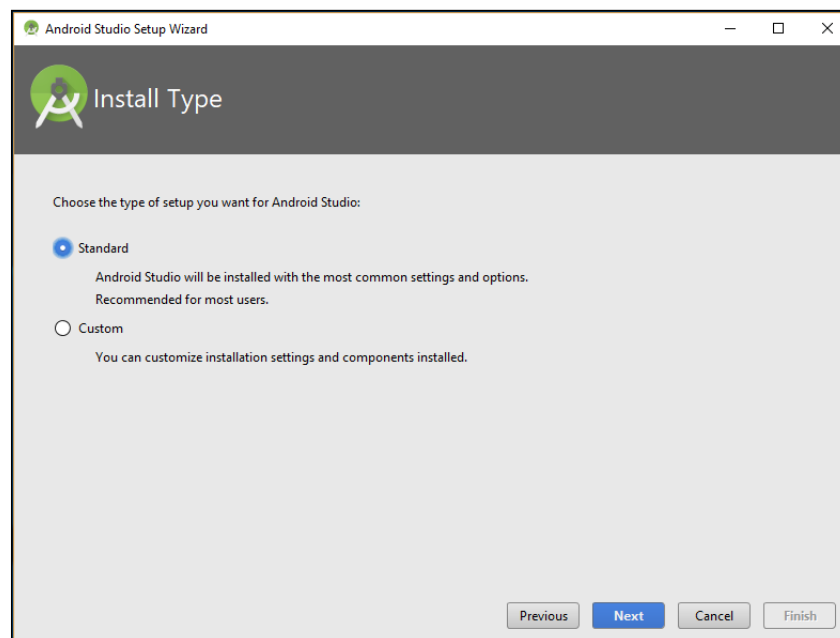


13. You will be greeted with the **Welcome** screen as shown next.

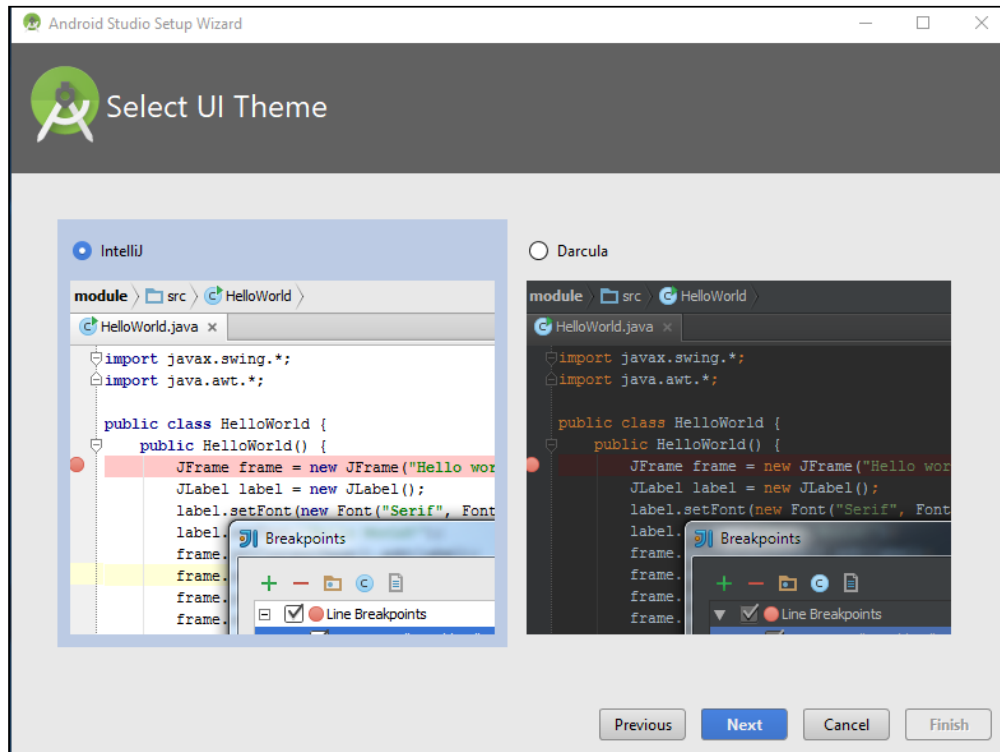


14. Click the **Next** button.

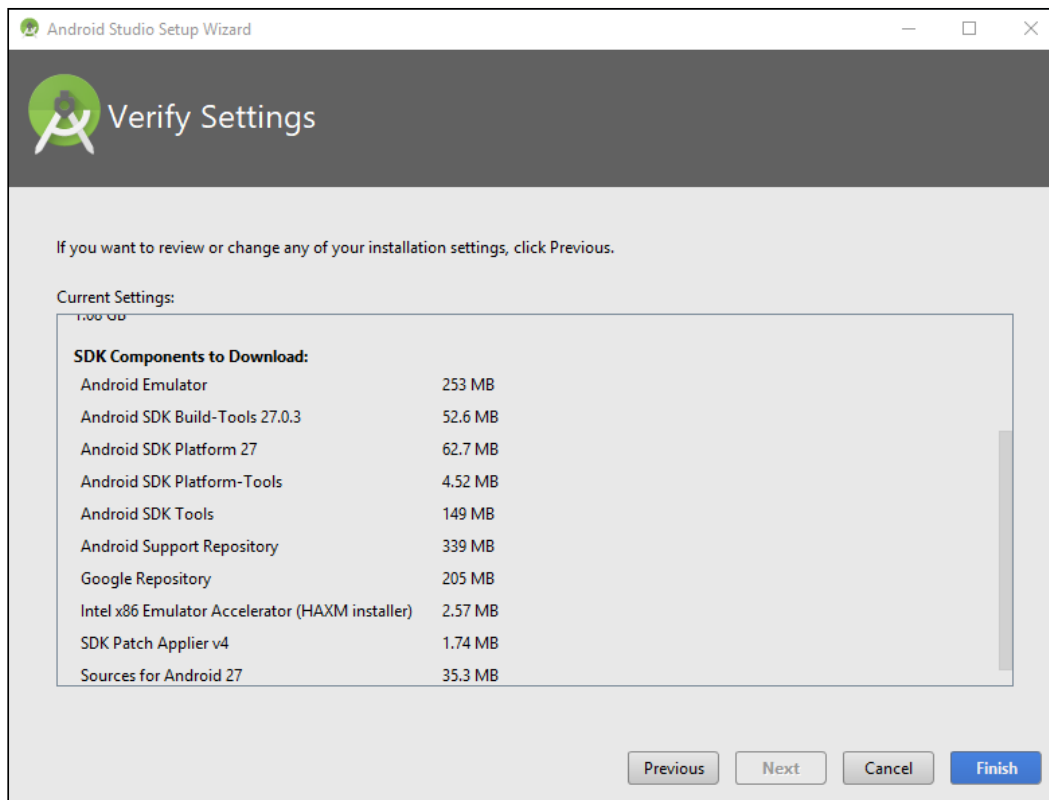
15. Choose **Standard** install type as shown next.



16. Click the **Next** button.
17. Choose whichever color scheme looks nice to you, I chose **IntelliJ** as shown next.



18. Click **Next**.
19. Now you will see the **Verify Settings** screen.



20. Click the **Finish** button. Android Studio will now commence some more downloads, they could take some time.
21. When Android Studio is ready you will have the option to run Android Studio. At this point click the **Finish** button. Android Studio is ready- probably. You can leave it open if you are carrying straight on with the next section or you can close it and then reopen it when instructed in the next section.

Final step

Using your preferred file manager software, perhaps Windows Explorer, Create a folder called `AndroidProjects`. Make it at the root of the same drive where you installed Android Studio. So, if you installed Android Studio at `C:/Program Files/Android` then create your new folder as `C:/AndroidProjects`.

Or if you installed Android Studio at `D:/Program Files/Android` then create your new folder as `D:/AndroidProjects`.



Note that the screen shots in the next section show the `AndroidProjects` folder on the `D:` drive. This is because my `C:` drive is a bit full-up. Either is fine. Keeping it on the same drive as the Android installation is neater and could avoid future problems so do so if you can.



Notice that there is no space between the words `Android` and `Projects` and that the first letter of both words is capitalized.

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 but for now, let's just build something and see our first Android game start to take shape. Then we can run it on an emulator and a real device.



The complete code as it stands at the end of this chapter is in the download bundle in the `Chapter 1` folder. Note however that you still need to go through the project creation phase explained in this chapter (and at the beginning of all projects) as Android Studio does lots of work that we can't see.

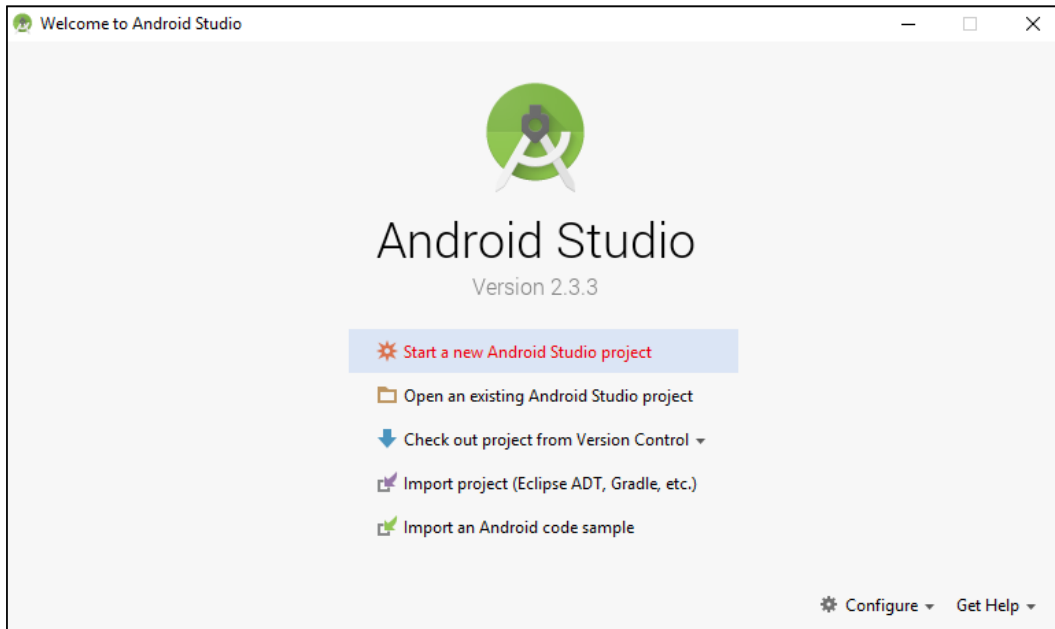
Follow these steps to start the project:

1. Run Android Studio in the same way you run any other app. On Windows 10, for example, the launch icon appears in the start menu.



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 as shown in the next image. Locate the **Start a new Android Studio project** option and left-click it.



3. After this, Android Studio will bring up the **New Project** window. This is where we will:
 - Name the new project
 - Choose where on our computer the project files should go
 - Provide a **Company domain** to distinguish our project from any others in case we should ever decide to publish the game on the Play store

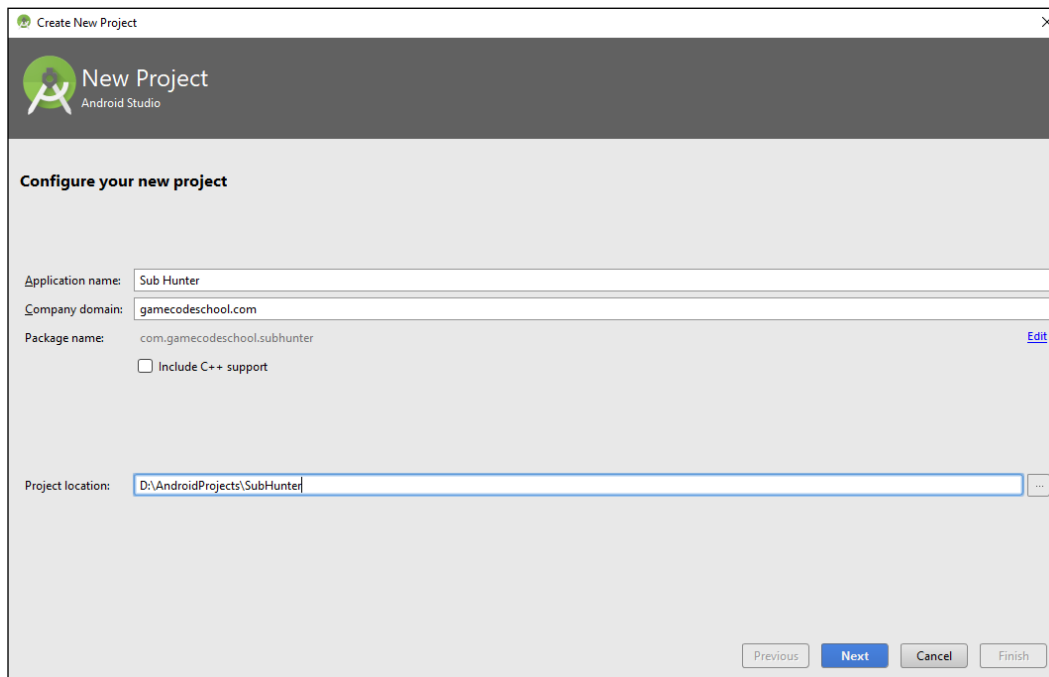
The name of our project is going to be `Sub Hunter` and the location for the files will be your `AndroidProjects` folder that we created in the *Setting up Android Studio* section.

The company domain can be almost anything you like. If you have a website, you could use the format `Yourdomain.com`. If not, feel free to use `gamecodeschool.com` or something that you just make up yourself. It is only important when you come to publish.

4. To be clear in case you can't see the details in the next image clearly, here are the values I used. Remember that yours might vary depending upon your choices for company domain and project location.

Option	Value entered
Application name:	Sub Hunter
Company domain:	gamecodeschool . com
Include C++ support	Leave this option unchecked (see the next information box if you want to know more)
Project location:	D:\AndroidProjects\SubHunter

5. The next picture shows the **New Project** screen once you have entered all the information.



6. In the previous image, you can see that Android Studio has auto-generated a **Package name**: based on the information entered. Mine is **com.gamecodeschool.subhunter**. Yours might be the same or not, it doesn't matter.



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 will be a great introduction to other languages and Java is also the official language of Android. Most top apps and games on the Play store are written in Java.

7. Click the **Next** button and we will continue to configure the Sub Hunter project. The following set of options is the **Target Android Devices** window. We can leave the default options selected as we are only making games for **Phone and Tablet**. The **Minimum SDK** option can be left as it is because it means our game will run on most (nearly all) Android devices from Android 4 to the latest version.



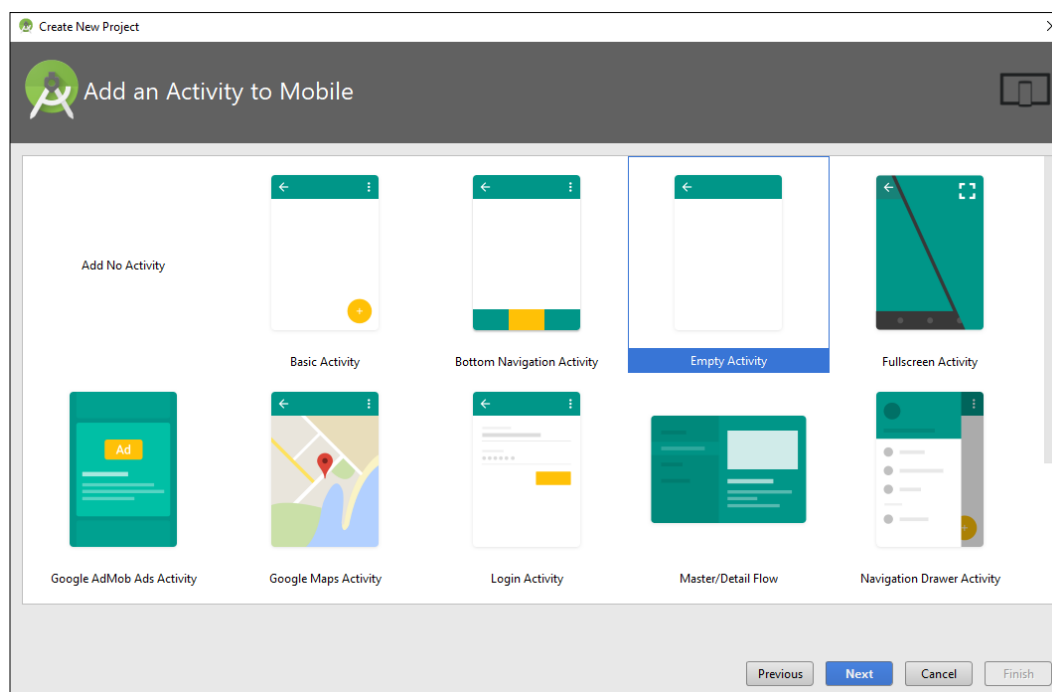
If you are reading this some years in the future, then the **Minimum SDK** option will probably default to something different but the code in this book will still work.

8. This next picture shows the **Target Android Devices** window we have just discussed, mainly just for your reference.

9. Click the **Next** button and we will move on.

10. The window that follows has the slightly obscure-sounding title, **Add an Activity to Mobile**. These are some useful project templates that Android Studio can generate for you depending on the type of app you are going to develop.

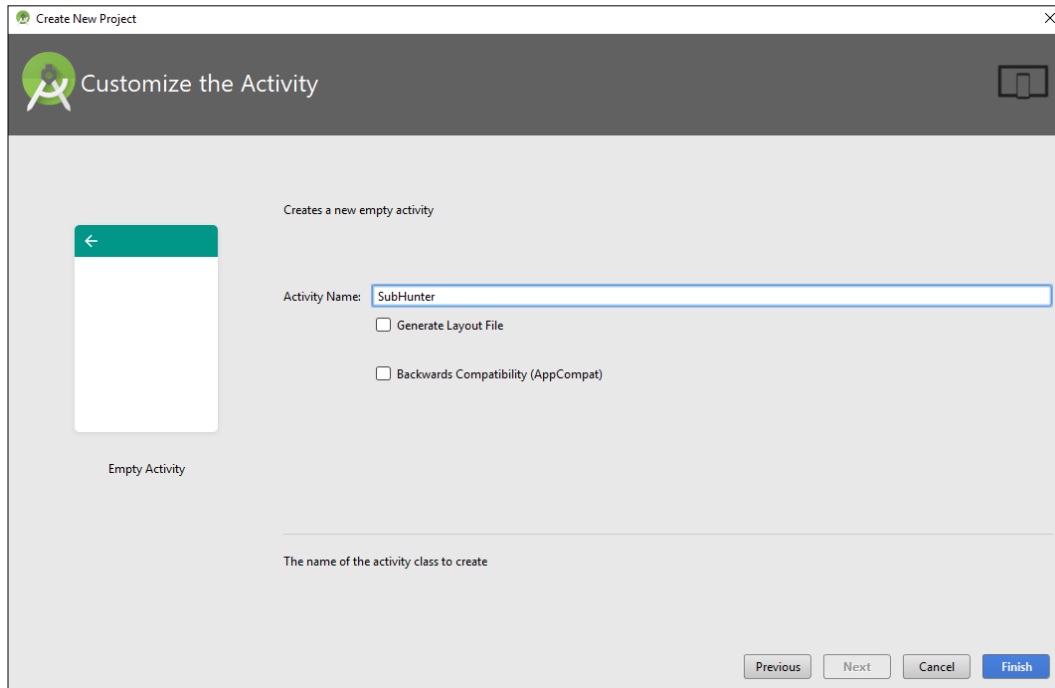
We will learn about Android **Activity** as the book progresses, but we are making games and want to control every pixel, beep and player input. For this reason, we will use the **Empty Activity** option. Android Studio will auto-generate a small amount of code to get our project started. Here is a picture of the **Add an Activity to Mobile** screen with the **Empty Activity** option selected.



11. Make sure **Empty Activity** is selected and click **Next**.
12. On the **Customize Activity** screen which you should now be looking at, we have a few changes to make. We could leave the defaults but then Android would generate more files than we need. In addition, we want to change the **Activity Name** to something more appropriate than **MainActivity**. Follow this short list of changes to configure the **Customize Activity** screen:
 - Rename the **Activity Name:** to SubHunter
 - Uncheck the **Generate Layout File** option – we do not want a layout generated as we will be laying out every pixel ourselves

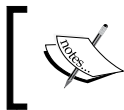
- Uncheck **Backwards Compatibility (AppCompat)** we won't need it for our games and they will still run on almost every phone and tablet

13. This is what the **Customize the Activity** screen should look like when you're done:



14. Finally, for this section, you can click the **Finish** button and we will explore a little of what we (and Android Studio) have just done.

At this stage you might be ready to proceed but depending on the install process you might need to click a couple of extra buttons. This is why I mentioned that we are "probably" finished installing and setting up. Look in the bottom window of Android Studio and see if you have the following message.



Note that if you do not see a horizontal window at the bottom of Android Studio like the one shown below you can skip these two extra steps.

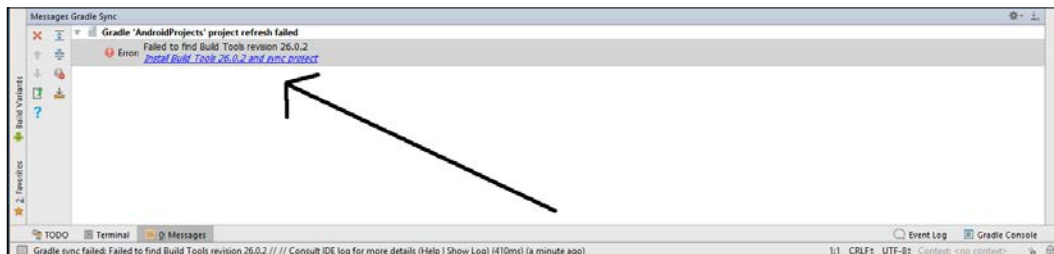
Extra step 1



If you do, click **Install missing platform(s) and sync project**, accept the license agreement and click **Next** followed by **Finish**.

Extra step 2

If you get another message like this:



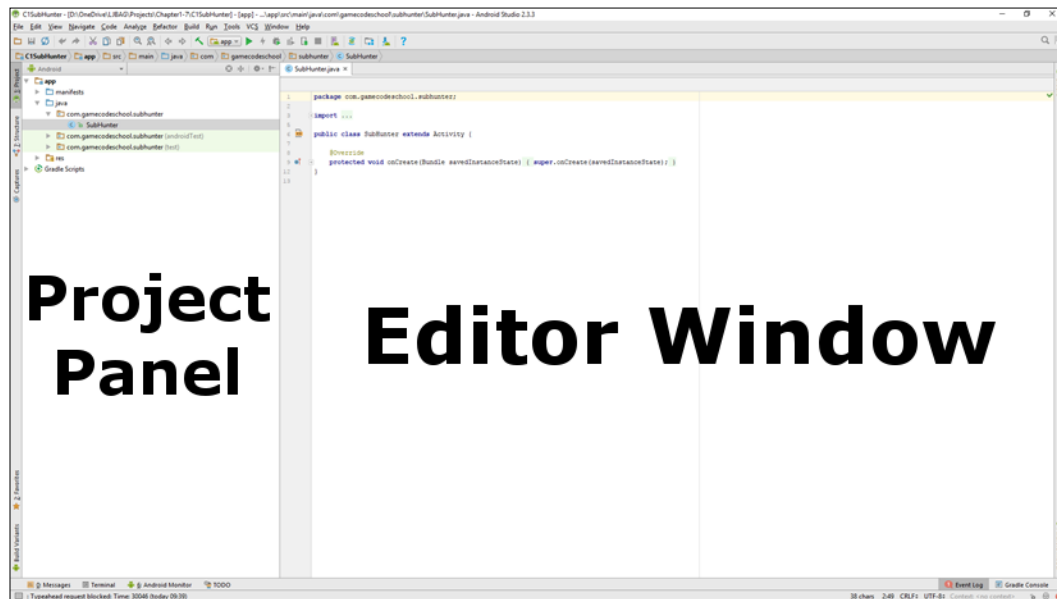
Click **Install Build tools....** Then click **Finish**.

[ You can tidy up a bit and close this bottom horizontal window by clicking the **Messages** tab on the very bottom of Android Studio, but this isn't required.]

Android Studio and our project – A very brief guided tour

I won't go into all the dozens of different windows and menu options because we will cover them as we need them but here are a few details to help begin getting familiar with Android Studio.

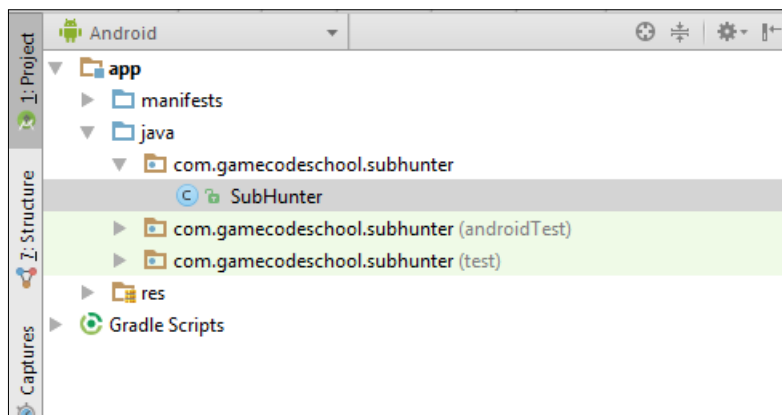
Have a look at the screen below and you will notice two major sections. One on the left and a larger window on the right:



Let's have a look at the panel on the left

The Project panel

The panel on the left can be changed to 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.



As you can see there are a few folders and sub-folders. For around 90% 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 **SubHunter** file in it. The little blue **C** icon to the left of **SubHunter** indicates that this file is a **class** and 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. All we need to know for now is that a class file is a file with code in it.

Notice that below the **com.gamecodeschool.subhunter** folder (with the **SubHunter** 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 prefix, that contains the **SubHunter** class/file.

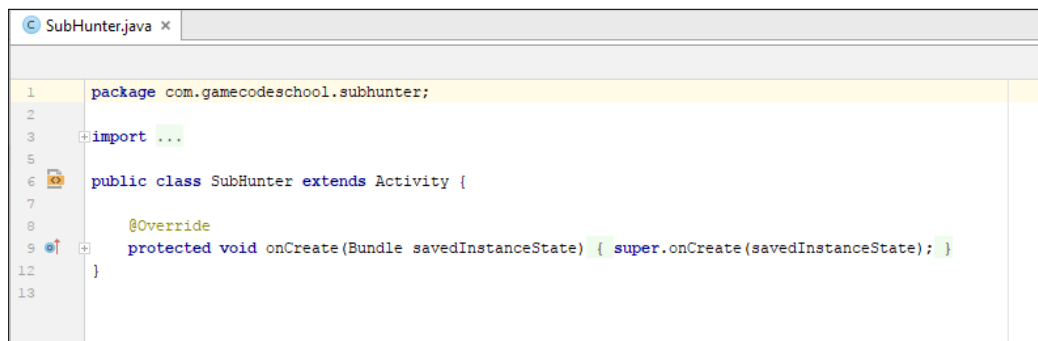
Feel free to explore the other folders. We will also be using the **res** folder in later projects to add sound files and graphics. We will be making brief adjustments to the file in the **manifests** folder in a moment as well.

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

Let's explore the place where the real action happens – the **Editor window**.

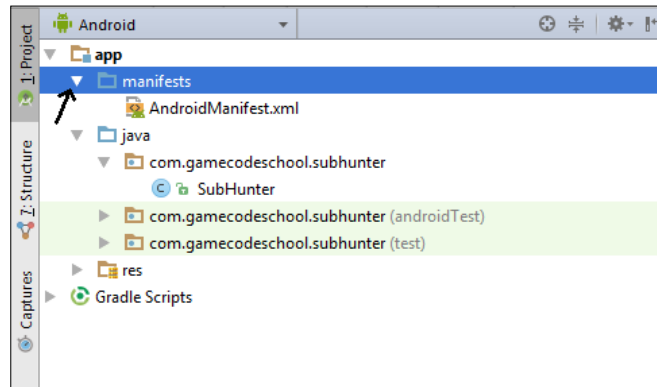
The Editor window

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 ready for editing at once. Have a look at the next image of the code editor as it stands now.

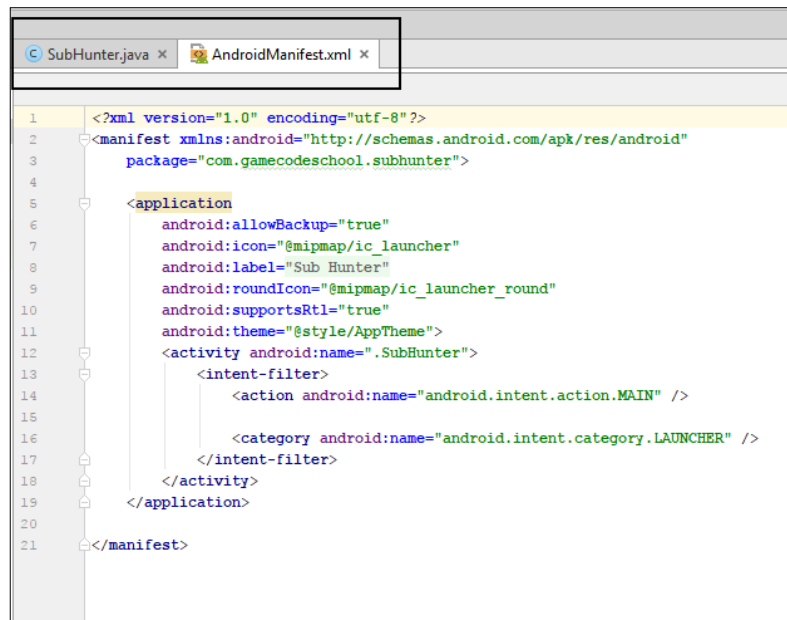


```
1 package com.gamecodeschool.subhunter;
2
3 import ...
4
5
6 public class SubHunter extends Activity {
7
8     @Override
9     protected void onCreate(Bundle savedInstanceState) { super.onCreate(savedInstanceState); }
10
11 }
12
13
```

Now, in preparation for the next section, open the **manifests** folder at the top of the **Project** panel. You do this by left-clicking on the little triangle to the left of the folder. I have highlighted this in the next image.



Inside the **manifests** folder, there is a single file, **AndroidManifest.xml**. Double-click the file and notice that it has been opened in the editor window and that we now have two tabs so that we can quickly switch between **AndroidManifest.xml** and **SubHunter.java**. This next image makes this clear.



Also, note that you can now see the **.java** extension of **SubHunter**. 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 Sub Hunter project.

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 which allows us to make configuration changes.

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

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

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

Place the cursor before the closing `>` shown above. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line shown above.

Immediately below `".SubHunter"` but before the newly positioned `>` type or copy and paste these two lines to make the game run full screen and lock it in the landscape orientation.

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
android:screenOrientation="landscape"
```

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 range of this file with the code we just added highlighted amongst it.

```
...
<activity android:name=".SubHunter"
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
    android:screenOrientation="landscape"
    >
    <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` we call **XML** (extensible **markup** language). Android uses XML for several different configurations. It is not necessary to learn this language as 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 and the screen has been locked in landscape mode even if the player holds their device in portrait orientation.

Deploying the game so far

Before we explore 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 see how to do that now. You have three options:

- Run the game on the Emulator on your PC (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 the emulator and a real device. However, screen touches are emulated by mouse clicks and proper testing of the player's experience is not possible.

The second option using a real device has a couple more steps but once set up, 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.

Probably 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 as expected. Only occasionally will you want to export an actual deployable package.



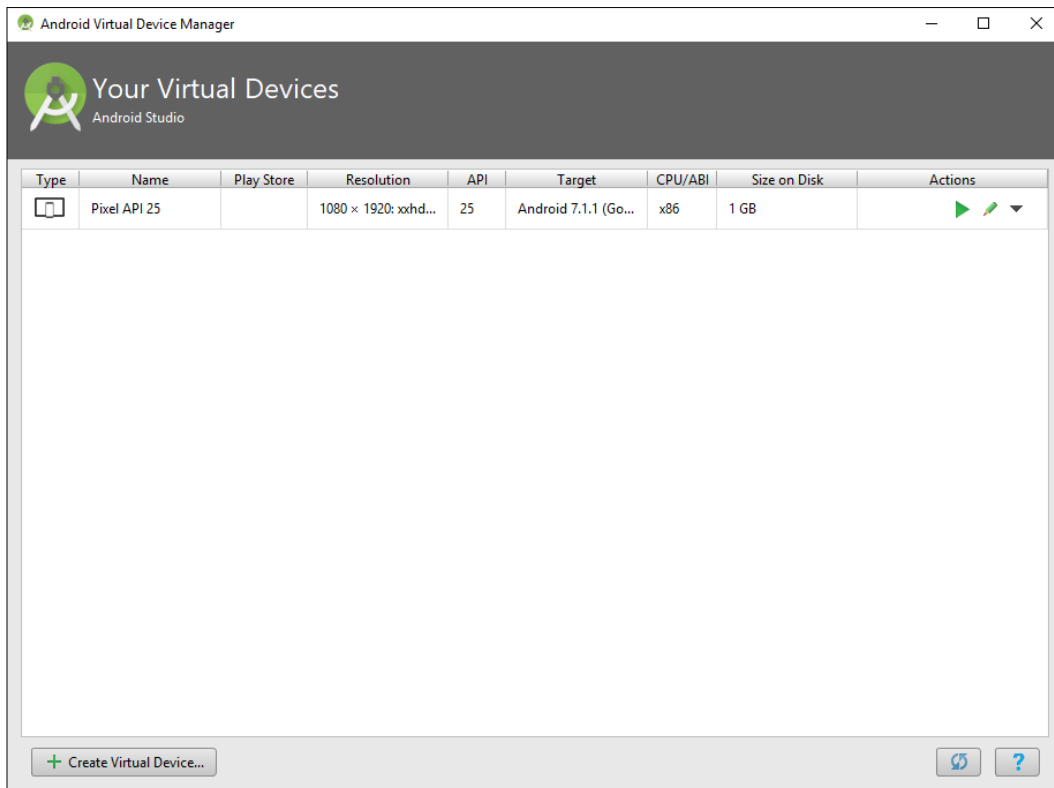
If you have an especially slow PC or a particularly aging Android device, you will be fine just running the projects in this book using just one option or the other. Note that a slow Android phone will probably be OK and cope, but a very slow PC will probably not handle the emulator running games and 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.

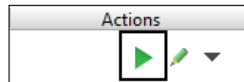
Running the game on an Android emulator

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

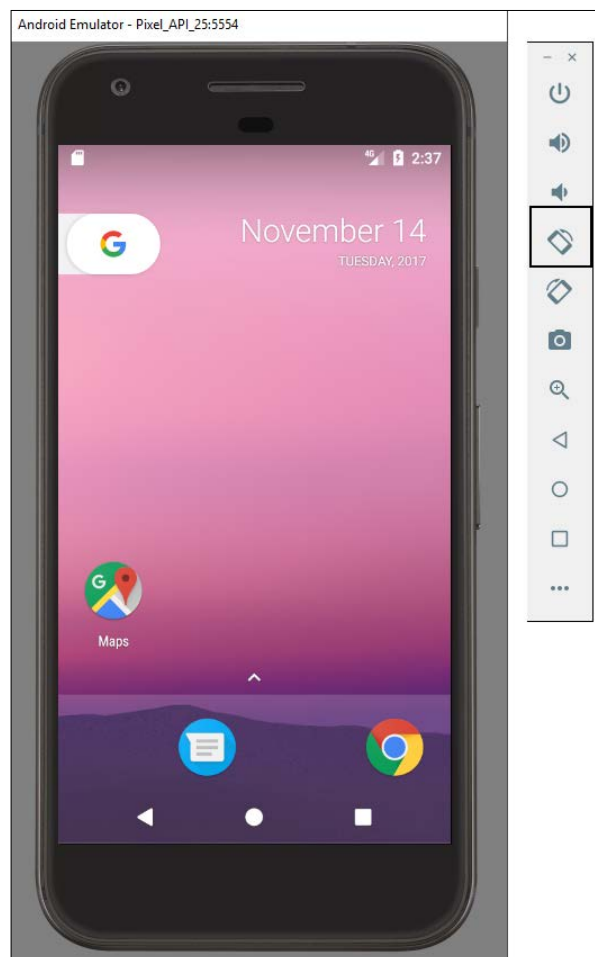
1. On the Android Studio menu bar select **Tools | Android AVD Manager**. AVD stands for Android Virtual Device (an emulator). You will see the following window.



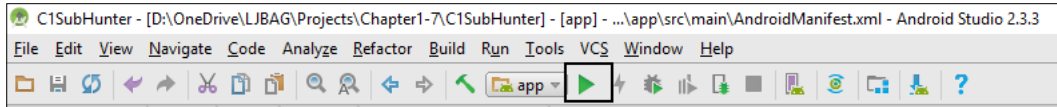
2. Notice there is an emulator in the list. In my case, it is **Pixel API 25** if you are following this sometime in the future it will probably be a different emulator that was installed by default. It won't matter. Click the green play icon shown in the next image and wait while the emulator boots up.



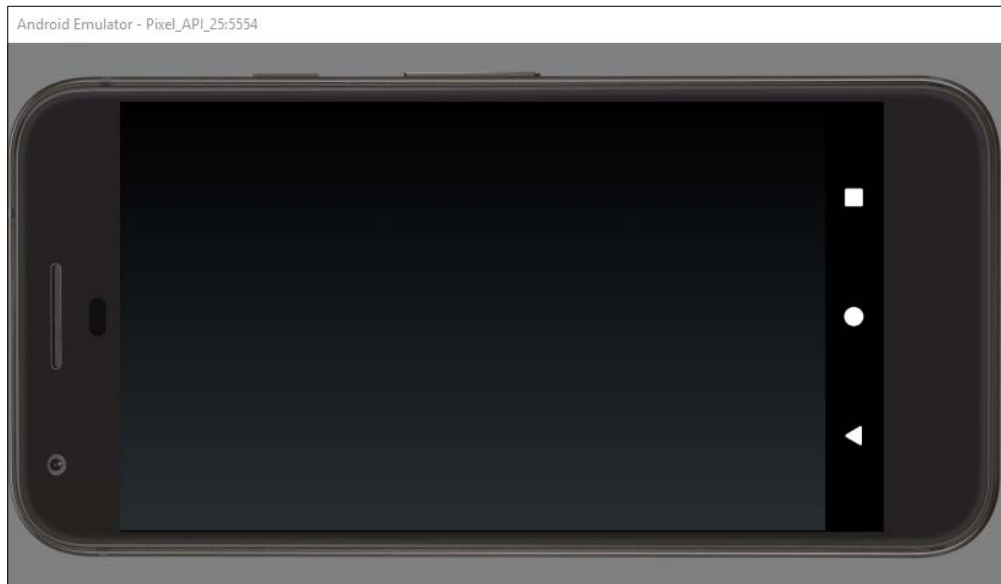
3. Now you can see the emulator as shown in the next image. Note also in the next image I have highlighted one of the icons in the control panel to the right of the emulator. Click that icon to rotate the phone into landscape mode ready to run the game.



4. Now you can click the play icon on the Android Studio quick-launch bar as shown in the next image and when prompted choose **Pixel API 25** (or whatever your emulator is called) and the game will launch on the emulator.



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



Clearly, we have more work to do but it is a good start. Let's see how to do the same on a real device and then we can go about building Sub' Hunter further.

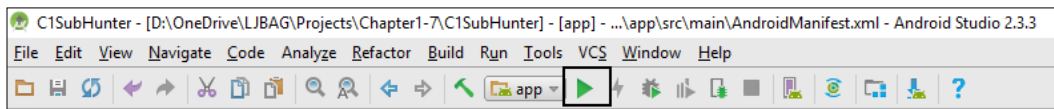
Running the game on a real device

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

[ Most newer devices won't need 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 to others. But the following sequence is probably very close, if not exact for enabling debugging on most devices:

1. Tap 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 the **About device** or **About Phone** option. Find the **Build Number** option and repeatedly tap it until you get a message informing you that **You are now a developer!**
3. Go back to the **Settings** menu
4. Tap **Developer options**
5. Tap the checkbox for **USB Debugging**
6. Connect your Android device to the USB port of your computer
7. Click the play icon from the Android Studio toolbar as shown in the next image



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

We are now ready to learn some Java and add real code to the Sub' Hunter project.

Summary

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

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

We can then make sense of the code that Android Studio generated for us (in `SubHunter.java`) and start to add our own code.

2

Java: First Contact

In this chapter, we will make significant progress with the Sub' Hunter game even though this is the first lesson on Java. We will look 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 an even briefer first glimpse at **object-oriented programming** that will begin to reveal the power of Java and the Android API

The auto-generated code we saw in the previous chapter will also be explained as we proceed and add more code too. Here is what you can expect to learn in this chapter:

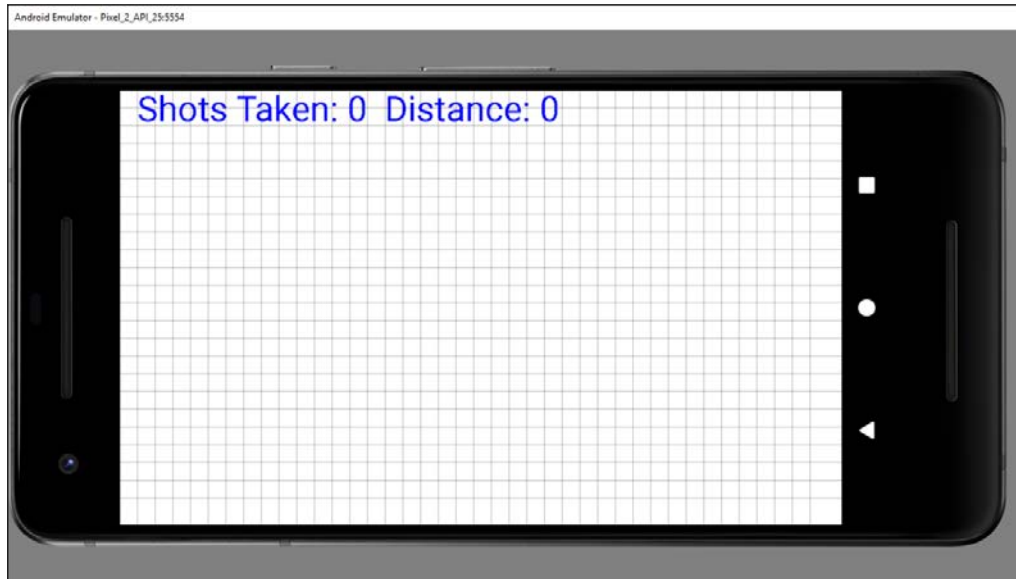
- Planning the Sub' Hunter game
- Introduction to Java methods
- Structuring Sub' Hunter with methods
- Introduction to Object-Oriented Programming
- Using Java Packages
- Linking up the Sub' Hunter methods

First, let's do some planning.

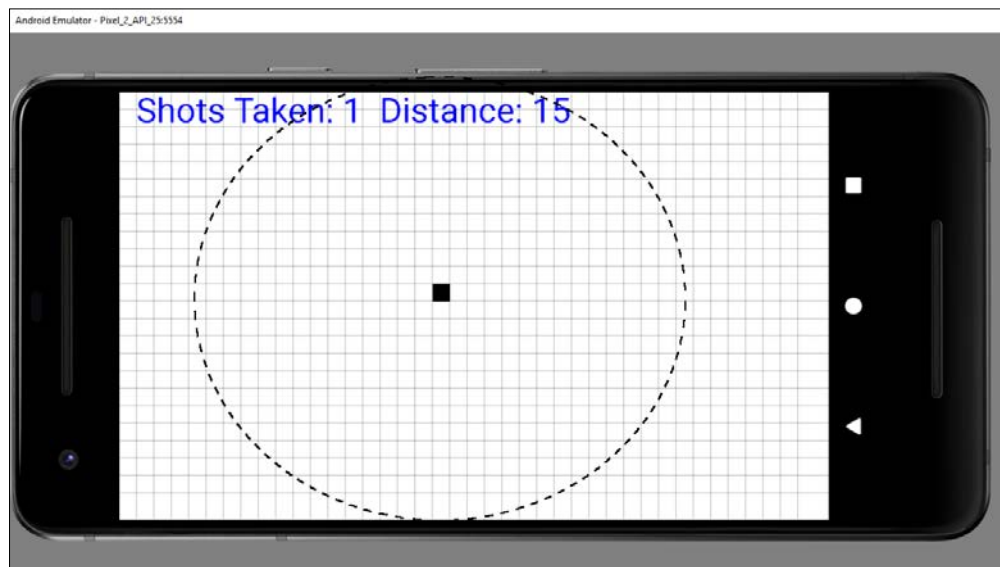
Planning the Sub' Hunter game

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

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



The grid represents the sea and each place on the grid is a possible hiding place for the submarine 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. The tapped square is shown highlighted and the distance to the sub' from the tapped square is shown.

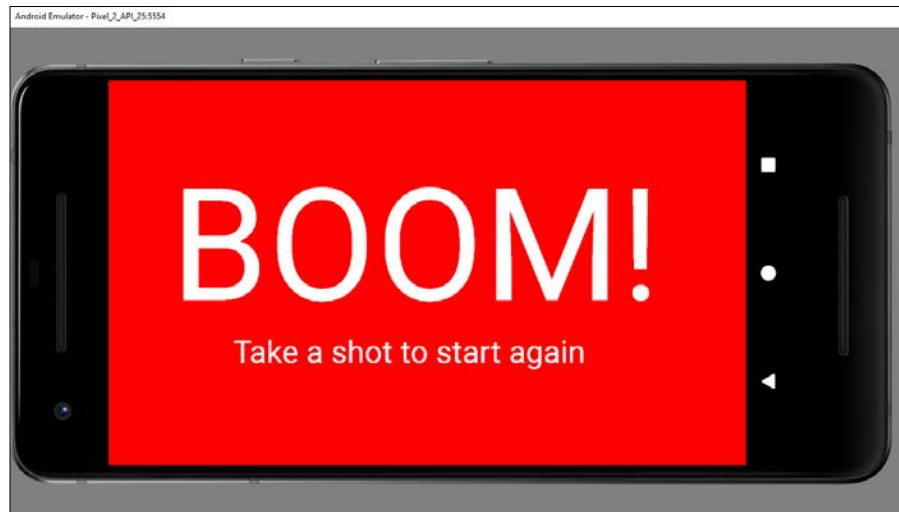


This feedback means the sub' is hiding somewhere *on* (not within) the radius of 15 squares as demonstrated in the previous image.



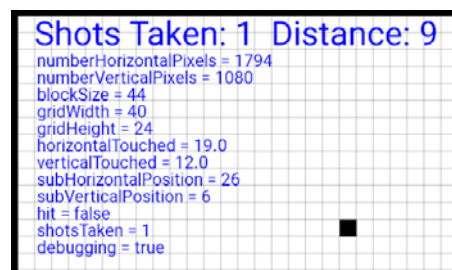
Note that the dashed-circle in the previous image is not part of the game. It is my attempt to explain the possible hiding places of the sub' based on the distance.

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



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.

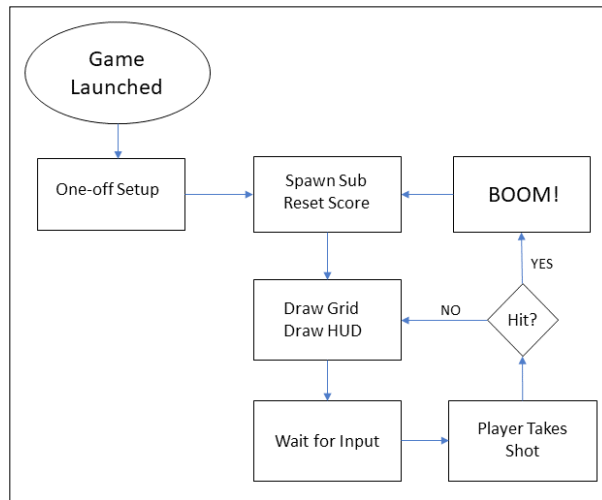
In addition to the game itself we will be writing code to display debugging information, so we can test the game and see if everything is working as it should be. This next image shows the game running with debugging information enabled.



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

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 this flowchart and then we will discuss it and 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 a decision is made, and execution of the code could go either way.



The flowchart shows the steps the game will take:

- The game is launched by tapping its icon in the app drawer(or running it in Android Studio).
- The sub' is placed at a random location by generating random horizontal and vertical numbers. The score is set to zero- in case this is not the first play of the game.
- Next everything is drawn to the screen; the grid-lines and the text (**HUD** heads-up-display) including debugging text (if enabled).
- At this point the game does nothing, it is waiting for the player to tap the screen.
- When the player taps the screen, the pixel 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. The program then branches either back to the drawing phase where the grid, the HUD, and added an extra square to show the shot location is drawn.
- If there was a hit then the "BOOM!" screen is shown.

- Actually, 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 as well. When the screen is tapped again it is considered the first shot of the next game and the flow of the code moves back to the "Spawn Sub' Reset Score" code and the whole process starts again.

The next two sections of this chapter will show how to flesh out this design with real Java code, and in the next chapter, we will see 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 we will see starting in this chapter and throughout the book, Java was designed to manage complexity by having us divide our code into separate chunks, 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 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. It 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 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 throughout the book.

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

In the preceding situation, the code will not run, as the compiler considers it 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. Also, like the single-line comment, it can be used to temporarily disable code, in this case 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 clever idea to read all the code:

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

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

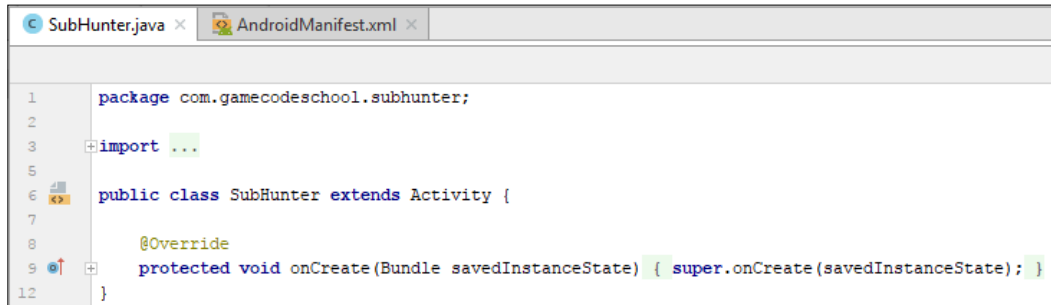
Let's add some useful comments to the Sub' Hunter project.

Mapping out our code using comments

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

In the previous chapter, we left the code having just added a couple of lines to the `AndroidManifest.xml` file 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 as shown in this image.



Android Studio is **folding** the code to help keep it organized. Look at the two little + icons in the previous image. Click on each of them to reveal the full code and then we will add some more to it.

Referring to our flowchart we have the "One-time Setup" element. In Android, the operating system dictates where some parts of our program must take place. For this reason, add the highlighted multi-line comment as shown next amongst the existing code. We will explore why this part of the code is where we do the one-time setup later in the chapter when we talk about *Linking up our methods*.



The complete code as it stands at the end of this chapter is in the download bundle in the Chapter 2 folder.

Add the highlighted code shown next.

```

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);
    }
}
```

Now, immediately before the final curly brace } of the code add the following highlighted comments. I have shown some of the existing code highlighted before the new comments to make it clear exactly where 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 comments above 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 to type the new code because it will be in context with these comments.



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**.

Introduction to Java methods

Java methods are a way of organizing and compartmentalizing our code. They are quite a deep topic and a full understanding requires knowledge of other Java topics. By the end of the book you will be a method Ninja but 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 like `draw`, `takeShot`, `newGame`, and `printDebuggingText` as well as a few more.

Code with a specific purpose can be wrapped inside a method, perhaps like this:

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

The above method called `draw` could hold all the lines of code that does the drawing for 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* but for now, you just need to know that all 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 this code:

```
draw();
```

Note the following, especially the last point, it 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 execution the program execution returns to the line after the method call.

So, in our example program **flow** would look like this.

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



In *Chapter 8, Object-Oriented Programming* we will also see how we can call methods in 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

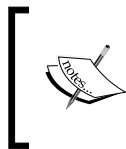
There is one more thing to know about methods before we do some more coding. All the methods I mentioned previously (`draw`, `takeShot`, `newGame` and `printDebuggingText`) are methods that *we* will be coding. They are our very own methods for our use only.

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

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

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

In the previous code, we are adapting and overriding the `onCreate` method. You will probably also notice that the prefix and postfix to the name look quite complicated. What exactly is going on here will be explained when we more thoroughly deal with methods in *Chapter 4, Structuring Code with Java Methods*.



The code `super.onCreate...` will be talked about in depth also but if you just can't wait for a brief explanation here is one. The `super.onCreate...` part of the code is calling another version of `onCreate` that also exists, even though we can't 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 shouldn't come as much 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 next.

```
/*
    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 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.

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

}
}
```

You have probably noticed 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 is work out how to handle a touch when the onTouchEvent method gets called. There is also an error in this code we will resolve this when we get introduced to object-oriented programming in a minute.

Add the takeShot definition after the comment shown next.

```
/*
    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 `// This code says "BOOM!"`.

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

Add the `printDebuggingText` definition after the comment about debugging text.

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

As the project progresses we will add code into each of the method definitions because at the moment they don't do anything. Furthermore, as we learn more about methods the postfix and prefix to the method names will also evolve as well as become more understandable.

Very closely related to methods and part of what we need to know to understand them better is **Object-Oriented Programming**.

Introduction to Object Oriented Programming

Object-Oriented Programming (OOP) makes it easy to do exceptional things. A simple analogy could be drawn with a machine, perhaps a car. When you step on the accelerator, a whole bunch of things happens under the hood. We don't need to understand about combustion or fuel pumps because a smart engineer has provided an interface for us. In this case, a mechanical interface—the accelerator pedal.

Take the following line of Java code as an example; it will look a little intimidating:

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

However, once you learn that this single line of code searches Space for the 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 object-oriented programming. 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 one of the important things- 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 are honed, refined and improved over years. Then, that same part is reused repeatedly, as well as occasionally improved further. Now, if you are going to be picky about my analogy, then you can argue that each of the car's components still must be built from raw materials using real-life engineers, or robots, and so on. This is true. Just stick with my analogy a bit longer.

The important thing about OOP, Classes, and Objects

What the software engineers do when they write their code is to 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 (manufactures) our custom-built creations into working code that can be run by the Android device.



We learned in chapter 1 that this working code is called DEX code.

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.



In programming, the words instance and object are virtually interchangeable but 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- until later.

Final word on OOP, Classes, and Objects – for now

Analogies are useful only to a certain point. It would be more useful if we can summarize what we need to know at this point:

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

One last analogy. We just need to know how to use that code, just as we only need to learn to drive the car- not manufacture it.

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 writing location aware apps and games. One of the things he does is he makes features such as getting the device's location in the world in a simple one-line task. So, the one line of code we saw previously sets many more lines of code in 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)
```

The locationManager is an object built from a class, getLastKnownLocation is a method defined in that class. Both the class that locationManager was built from and the code within the getLastKnownLocation method is exceptionally complex- but we just 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.



All methods are part of a class. You need an object built from the class to use the methods. All will be explained 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" even more than using classes. They use pre-made game libraries like LibGDX or complete game engines like Unity or Unreal. We will be learning Java without these cheats, leaving you well prepared to move on to libraries and engines should you wish to.

But where are all these classes? Do you remember this code from when we were typing the method definitions?

```
/*  
    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 that the previous code had an error. The first reason there was an error in the code `MotionEvent` is that Android Studio doesn't know anything about the `MotionEvent` class- yet. Note also in the previous code I have added a line of code.

```
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 previous code. Don't miss the semi-colon `;` on 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.os.Bundle;
```

These lines of code make available the `Activity` and `Bundle` classes as well as their methods. Comment out the above two lines like this:

```
// import android.app.Activity;
// 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 which Android Studio no longer is 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 which since we commented out the previous two lines Android 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 we will use in this project including one to fix the `MotionEvent` 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 one that I have highlighted in this next code.

```
package com.gamecodeschool.subhunter;

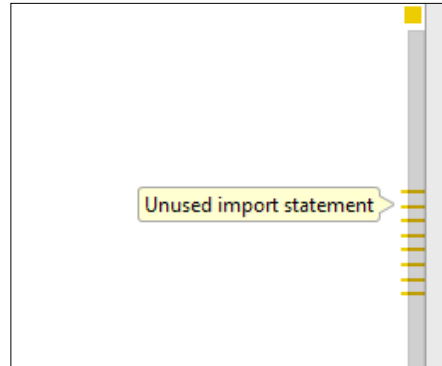
// These are all the classes of other people's
// (Android) code that we use for Sub Hunter
import android.app.Activity;
import android.os.Bundle;
import android.view.MotionEvent;
```

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

Add the highlighted code. The syntax needs to be exact so consider copy 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.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 the new lines of code are greyed-out in Android Studio. This is because we are not using them yet and at this stage, they are technically unnecessary. Also, 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.



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 see how to add `import` statements as and when we need them without any fuss.

We have briefly mentioned the `Activity` class. We need to learn a little bit more about it to proceed. We will do so while linking up our methods with some 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 alluded to, as well as mentioned in our comments that the `onCreate` method (provided automatically by Android) will handle the One-time 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 `onCreate`.

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



Actually, 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 to put the One-time Setup code in `onCreate` and we can be sure it will be executed before any of the other methods we write.

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` too. Add this highlighted code shown next.

```
/*
    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);

    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 `newGame` followed by `draw` but it also contains this 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 they should.

Now let's print some text in `newGame` so we can see it being called as well. Add the 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");
}
```

After this, to implement the course of our flowchart we need to call `takeShot` from `onTouchEvent`. Note as well that we are printing some text for tracking purposes here also. Remember that `onTouchEvent` is called by Android when the player touches the screen. Add the highlighted code to `onTouchEvent`.

```
/*
    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 `draw` and some debugging text into `takeShot` as per the flowchart and shown next.

```
/*
    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 it is being called. Remember that on the flowchart after we do 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 draw to `onTouchEvent`.



The connection between Android and `onTouchEvent` is permanent and never broken. We will explore how this works when we talk about threads in *Chapter 9, The Game Engine, Threads and The Game Loop*.

Add the highlighted code shown next 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 `printDebuggingText` or `boom`. Neither have we called these methods from any of the other methods. This is because we need to learn some more Java then do more coding before we can add any code to these methods.

In our tests when the screen is clicked, `onTouchEvent` which is analogous to the "Wait For Input" phase will call `takeShot`, which in turn will call the `draw` method. Later in this project `takeShot` will make a decision to *either* call `draw` or `boom` depending upon whether the player taps on the grid square with the sub' in it or not.

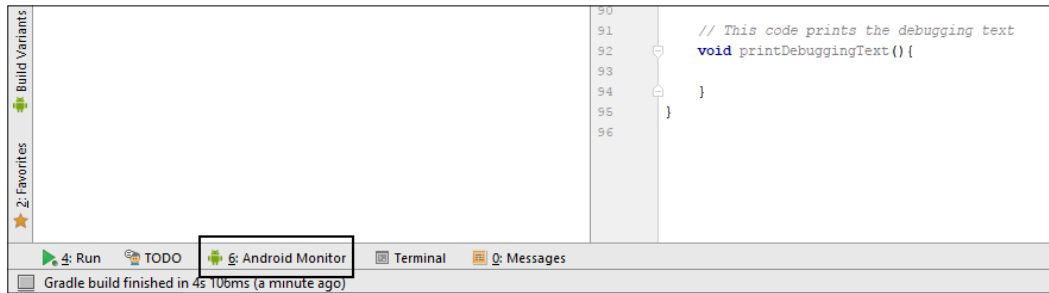
We will also add a call to `printDebuggingText` once we have some data to debug.

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

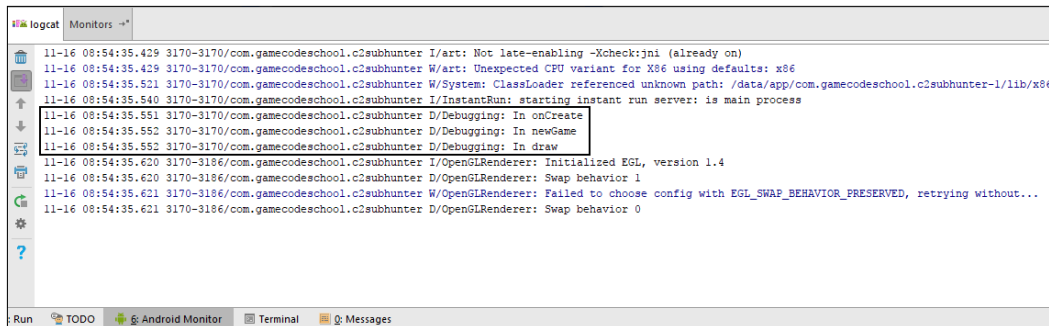
1. In the Android Studio menu bar select **Tools | Android AVD Manager**.
2. Notice there is an emulator in the list. In my case, it is Pixel API 25. If you are following this later it will likely be a different emulator that was installed by default. It won't matter. Click the green play icon.

- Now you can click the play icon in the Android Studio quick-launch and when prompted choose **Pixel API 25** (or whatever your emulator is called) and the game will launch on the emulator.

Now open the **logcat** window by clicking the **Android Monitor** tab at the bottom of the screen as shown in the next image.



In the **logcat** window, we can see that when we start the game that lots of text has been output to logcat. This next image is a snapshot of the entire logcat window to make sure you know exactly where to look.



This next image is zoomed in on the three relevant lines, so you can clearly see the output even in a black and white printed book.

```
11-16 08:54:35.551 3170-3170/com.gamecodeschool.c2subhunter D/Debugging: In onCreate
11-16 08:54:35.552 3170-3170/com.gamecodeschool.c2subhunter D/Debugging: In newGame
11-16 08:54:35.552 3170-3170/com.gamecodeschool.c2subhunter D/Debugging: In draw
```

In future, I will show only the most relevant part of the logcat output as text in a different font like this.

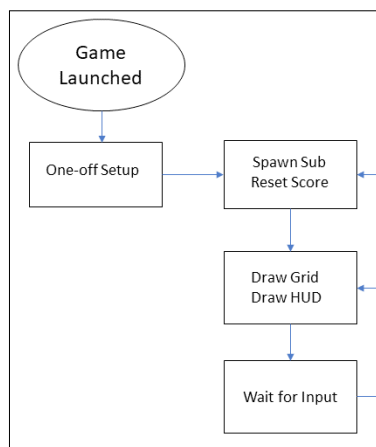
```
Debugging: In onCreate
Debugging: In newGame
Debugging: In draw
```

Hopefully, the font and the context of the discussion will make it clear when we are discussing logcat output and when we are discussing actual code.

What we can gather from all this is that:

- When the game was started the `onCreate` method was called (by Android)
- Followed by the `newGame` method which executed then returned to `onCreate`
- Which then called `draw`

The game is now currently at the "Wait For Input" phase, just as it should be according to the flowchart.



Now, go ahead and click the screen. Then we should see that the `onTouchEvent`, `takeShot` and then `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 the screen of the emulator.

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

As we 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 the mouse button (or finger) down and it is also called when the mouse button (or finger) is released. To simulate a tap, we only want to respond to releases (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) to Sub' Hunter.

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.

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 the methods yet but 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 object-oriented programming. It doesn't matter if 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. How the game "remembers" values like the position of the submarine or the size of the grid. We will see 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 allows us to give our game the data it needs. Things like 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.

This is what we will do in this chapter:

- Have a full discussion about Java variables including **types, references, primitives, declaration, initialization, casting, concatenation** and more too
- Practice writing code to use and manipulate variables with operators and expressions
- Learn about the Android coordinate system for drawing text and graphics to the screen
- Use what we have learned to add variables to the Sub' Hunter game
- See 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.

Java **syntax** is the way we put together the language elements of Java to produce code that can be translated to DEX to work in the Dalvik virtual machine. 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 already when using method definitions and method calls to structure Sub' Hunter.

These Java "words" are many in number, but taken in small chunks 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 the keywords skillfully. Many consider programming as an art as much as a science and there is some truth to this.



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 ready to be used or altered when necessary by using the name they were given.

Computer memory has a highly complex system of addressing, which fortunately we do not need to interact with 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 **DVM (Dalvik Virtual Machine)** 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. 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. We can then tell it to do things such as: get `livesRemaining` and multiply it by `bonusAmount`, set it to zero, and so on.

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:

- Player completes a level so add `levelBonus` to `playerScore`
- Player gets a new high score so assign the value of `score` to `highScore`
- 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.



We call this **camel casing**. For example, `camelCasing`.

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

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

The earlier examples also imply something.

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

We will see shortly that indeed there are distinct **types** of variable.



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

Before we look at some more Java code with some variables, we need to first 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 a width, height, color, ammo, shieldStrength and position variables?

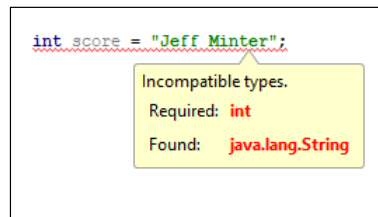
Another common requirement in a computer program, including games, is the right or wrong calculation. Computer programs represent right or wrong calculations using `true` or `false`.

To cover these and other types of data you might want to store or manipulate, Java has types. These types are many but 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 as well. 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. These examples are the best way to explain types.

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 `String`. And if we try to store a player's name, perhaps "Jeff Minter" in an `int` like `score`, meant for numbers, we will certainly run into trouble, as we can see from the next screenshot:



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, and then we will see how to start using them:

- `int`: The `int` type is for storing integers, 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 hints at, `long` data types can be used when even larger numbers are needed. A `long` type uses 64 bits of memory and 2 to the power of 63 is what we can store in this. If you want to see what that looks like, here it is 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.



You might be wondering when you might 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. That is 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 possible in a `float` is therefore decreased compared to non-floating-point numbers.
- `boolean`: We will be using plenty of Booleans throughout the 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?



I have kept this discussion of data types to 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 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 to use it.

Remember, these variables are known as the primitive types. They use predefined amounts of memory and so- using our warehouse storage analogy- fit into predefined sizes of the storage box.

As the "primitive" label suggests, they are not as sophisticated as reference types.

Reference variables

You might have noticed that we didn't cover the `String` variable type that we previously used 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 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 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?

As we are thinking about the device's memory as a huge warehouse full of racks of labeled storage boxes, then you can think of the DVM 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 DVM will quickly move stuff around in a fraction of a second to avoid collisions. Also, when appropriate, Dalvik, 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.

Dalvik keeps reference variables in a different part of the warehouse to the primitive variables. And we will learn more details about this in *Chapter 14, 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 with all the variables of a certain type lined up in a precise order. Arrays are reference types, so Dalvik 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 we 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 be getting 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 an amount of memory. Hence, we must let the Java compiler know the type of the variable before we begin to use it.

How to use variables

That's enough theory. Let's see how we would 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 type `int` with the name `score`, we would type:

```
int score;
```

That's it. Simply state the type, in this case, `int`, then leave a space and type the name you want to use for the variable. Note also the semicolon `;` on the end of the line will tell the compiler that we are done with this line and 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 used next are arbitrary.

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

So now we have reserved a space in the warehouse that is 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 to the variable. Think about placing a value inside the storage box:

```
score = 1000;  
millisecondsElapsed = 1438165116841L;// 29th July 2016 11:19 am  
subHorizontalPosition = 129.52f;  
debugging = true;  
playerName = "David Braben";
```

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

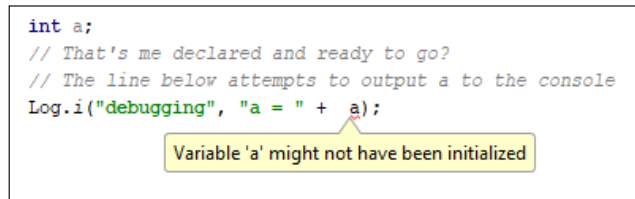
We can also combine the declaration and initialization steps. In the following we declare and initialize the same variables as we have previously, but in one step:

```
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 is dependent upon the specific situation. The important thing is that we must do both at some point:

```
int a;
// That's me declared and ready to go?
// The line below tries to output a to the console
Log.d("debugging", "a = " + a);
// Oh no I forgot to initialize a!!
```

This would cause the following warning:



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.

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.

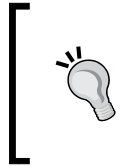
Once you have seen one operator you can guess what the others do but some examples of each will help to get familiar with them:

- The assignment operator (=): This makes the variable to the left of the operator the same as the value or variable to the right. For example:
 - `score = 1000;`
 - `highScore = score;`
- The addition operator (+): This adds together values on either side of the operator. 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. Perhaps like this:
 - `horizontalPosition = horizontalPosition + movementSpeed;`
 - `score = score + 100;`
- The subtraction operator (-): This subtracts the value on the right side of the operator from the value on the left. Usually used in conjunction with the assignment operator. Perhaps like this:
 - `numberAliens = numberAliens - 1;`
 - `timeDifference = currentTime - fastestTime;`
- The division operator (/): This divides the number on the left by the number on the right. Again, 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.
 - `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 --;`



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 meet a whole bunch later in *Chapter 7, Making decisions with Java If, Else and Switch*.



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 is provided for the extra curious readers.

A note about declarations, assignments, and operators; 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;
```

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

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

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 type `int`.
- 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`.
- 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 variables named `gridWidth` and `gridHeight` which will also be of type `int`.
- When the player touches the screen, we will need to remember and use the coordinates that were touched. These values will be precise floating-point coordinates and we will name them `horizontalTouched` and `verticalTouched`.

- It will also be necessary to choose and remember which grid position (horizontally and vertically) the sub' is randomly spawned in. Let's call them `subHorizontalPosition` and `subVerticalPosition`. These will also be of type `int`.
- Each time that the player takes a shot we will need to know whether the sub was hit- or not. This implies a `boolean` variable and we will call it `hit`.
- The `shotsTaken` variable will be of type `int` and as the name implies will be used to count the number of shots the player has had so far.
- The `distanceFromSub` 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 actual game. A `boolean` named `debugging` will do nicely.

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

Declaring the variables

In Android Studio add the following highlighted variable declarations.



The complete code as it stands at the end of this chapter can be found in the download bundle in the Chapter 3 folder.

Notice 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 the 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 the value 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 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 plain.

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, `numberHorizontalPixels`, and `numberVerticalPixels`. Also, once we have the resolution information we will also be able to do calculations for initializing `gridHeight` and `blockSize`.

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

```
    /*
    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);

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

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

```



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 Android Versions* when we discuss **chaining**. Simply explained, we create an object of type `Display` called `display` and initialized with the result of calling both `getWindowManager` then `getDefaultDisplay` methods in turn which are part of 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 an `x` and `y` variable 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 `numberHorizontalPixels` and `numberVerticalPixels`.

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 greyed 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;
```

The explanation just given is necessarily incomplete. Your understanding will improve as we proceed.

Now 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: Initialising the variables

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

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

    // 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 because it gives us a glimpse into the later chapters of the 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 has previously been 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 do some simple math to initialize `blockSize` and `gridHeight`. The `blockSize` variable is assigned the value of `numberHorizontalPixels` divided by `gridWidth`.

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

Remember that `gridWidth` was previously initialized with the value 40. So, assuming the screen once it is made full screen and landscape (as ours has) are 1776 x 1080 pixels (as the Google Pixel emulator is) then `gridWidth` will be as follows.

$1776 / 40 = 44.4$



Actually, the Google Pixel has a horizontal resolution of 1920 but the Back, Home, and Running Apps controls take up some of the screen space.

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

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.

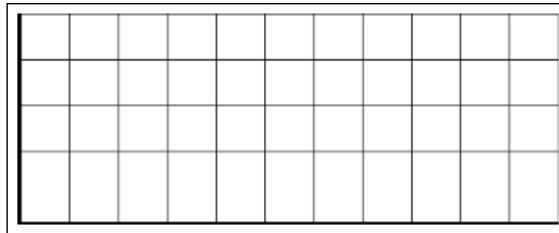
The final line in the previous code uses `blockSize` to match up how many blocks can be fitted into the height.

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

Using the resolution of the Google Pixel emulator as an example reveals the first imperfection of our code. Look at the math the previous line performs. Note that we had a similar imperfection when calculating `gridWidth`.

$1080 / 44 = 24.54545\dots$

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



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



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 the book.

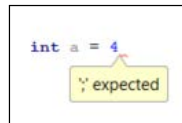
Another important point to consider for the future although it is not worth getting hung-up on now is that different devices have different ratios of width to height. It is perfectly possible, even likely that some devices will end up with a different number of grid positions than others. In the context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

As the code is becoming 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, so you know how to solve them when they arise.

Errors, warnings, and bugs

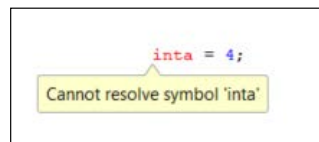
The most common getting started errors are the syntax errors. They most frequently occur when we mistype a Java keyword or forget to leave a space after a keyword.

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

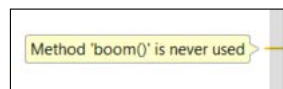


There are occasions when we don't need semicolons. We will see these as we proceed.

Look at this image that shows the error when we type a word the compiler doesn't recognize.



Also, notice that Android Studio is giving us a warning that we have some unused variables. Warnings for example like **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 looking at the unused import statements. There are other warnings like this one that indicates that method boom is never called.



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

Printing Debugging Information

We can output all our debugging information and a good time to do this is in the `draw` method because every time something happens the `draw` method is called. But 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 ask 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 this next code to output the text in real-time to the device screen instead of the logcat window.

Now add this 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 is all quickly explained. We are using `Log.d` to write to the logcat as we have done before. What is new is that for the second part of the output we write things like `"" + distanceFromSub`. This has the effect of concatenating the value held in an empty `String` (nothing) with the value held in `distanceFromSub`.

Examine the previous code once more and you can see that every line of code states the variable name inside the speech marks having the effect of printing the variable *name*. On each line of code, this is followed by the empty speech marks and variable concatenation which outputs the *value* of the variable.

Seeing the output will make things clear.

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 to 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 it opens a whole new world of possibilities 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 like `Canvas` 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 get comfortable with Java programming, in this chapter, we will take a closer look at **methods** because although we know that you can call them to make them execute their code, there is more to them that haven't been discussed so far.

In this chapter, we will look at the following topics:

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

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

Methods



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.

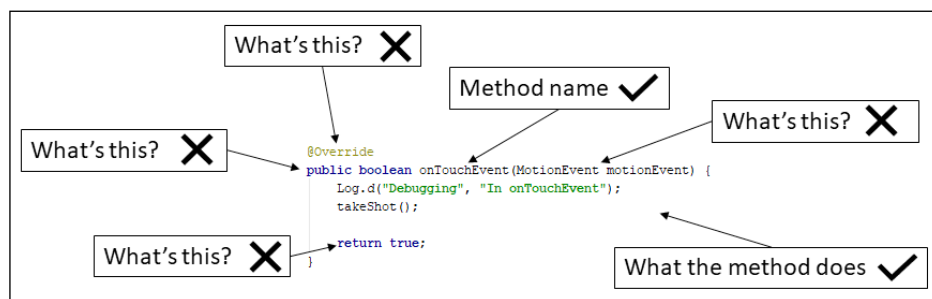
Having said this, it is not necessary to master or memorize everything about method theory before moving on with the project. If something doesn't quite make sense, the most likely reason is that something else later in the book will make things come more in to focus.



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, this image roughly sums up where our understanding of methods is now. The ticks indicate where we have discussed an aspect relating to methods and the crosses X indicate where we have not explored yet.



As we can see in the previous image, 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 later in the 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 signature of methods.

The method signature

The first part of a method that we write is called the **signature**. And as we will see soon 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 performs 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 could then use our new method from another part of our code like this:

```
...  
// I like it here  
// but now I am going off to 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 above method could look like this:

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


We don't need to write methods to add two `int` variables together, but the example helps us see a little more into the workings of methods. This is what is happening step by step:

- 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`.
- Within the method body, the variables `a` and `b` are added together and used to initialize the new variable `int answer`.

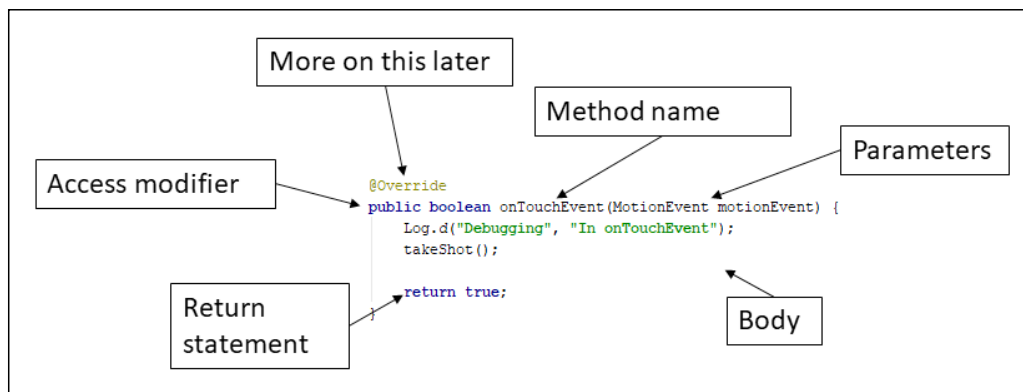
The line `return answer` returns the value stored in `answer` to the calling code, causing `myAnswer` to be initialized with the value 6.

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 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 below 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 next image which is the same method as the one in the previous image but this time I have labeled all the parts.



Note in the image I have not labeled the `@override` part. We already know that this method is provided by the class we are working within 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 section *Method overloading and overriding confusion* later in the chapter. In summary of the image here are the parts of the method signature and their names.

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

And here in the below table are a few examples some that we have used so far as well as some more hypothetical examples laid out in a table to begin to explain them further. We will then look at each one in turn.

Part of signature	Examples
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> , <code>long</code> and so on) or any predefined reference types (such as <code>String</code>) and user/Android defined types/classes (such as <code>Spaceship</code> , <code>Bullet</code> , <code>Bitmap</code> and so on.)
Name of method	The name which distinguishes this method from others like <code>setCoordinates</code> , <code>addAToB</code> and so on. 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> , <code>(int x, int y, float velocity)</code> . Parameters are values passed into the method from the calling code.

Modifier

In our earlier examples, we only used a 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 like **public** and **private**. 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 are discussing the other vital Java topic 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 so I just mentioned it.

Return type

Next up is the return type. Like a modifier a return type is optional. So, let's look a bit closer. 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 was:

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

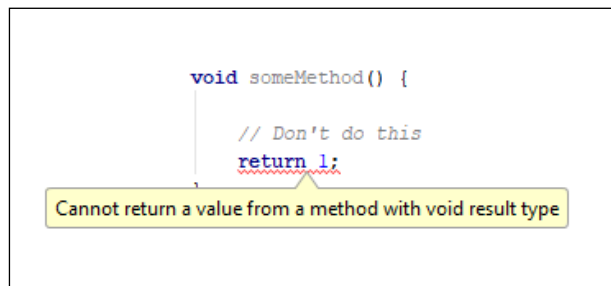
Here the return type in the signature is highlighted above. So the return type is an `int`. This means the method `addAToB` sends back (returns) to the code that called it a value that will fit in an `int` variable. And as you can see in the second highlighted part of the code this is exactly what happens with:

```
return answer;
```

The variable `answer` is of type `int`. 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.



It can, however, use the `return` keyword without a value. Here are some combinations of 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 the methods above, 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 code above would execute all the code in each method in turn.

A closer look at method names

The method name when we design our own methods is arbitrary. But it is a convention to use verbs that make clear what the method will do. Also using the convention of the first letter of the first word of the name being lower case and the first letter of later words being upper case. This is called camel case as we learned while learning about variable names. For example:

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

The above method is perfectly legal and will work. However, what does it do? Let us look at some examples (slightly contrived) that use the 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 avoid confusion. Let's have a look at the parameters in 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;  
}
```

Above, 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 next 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 is 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 with comments like:

```
// code here
```

or

```
// some code
```

We know exactly what to do in the body already.

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

The best thing we can do next is writing 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 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.



The complete code for this mini-app can be found in the download bundle 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
Application name:	Exploring Method Overloading
Company domain:	Gamecodeschool.com (or your own)
Include C++ support	Leave this option unchecked
Project location:	D:\AndroidProjects\ExploringMethodOverloading

As we did before, be sure the **Empty Activity** option is selected on the **Add an Activity to Mobile** screen before clicking **Next**. Also, be sure to uncheck **Generate Layout File** and **Backwards Compatibility (AppCompat)**. Don't worry about naming the Activity this is just a mini app to play around with we will not be returning to it.

We will get on with writing three methods but with a slight twist.

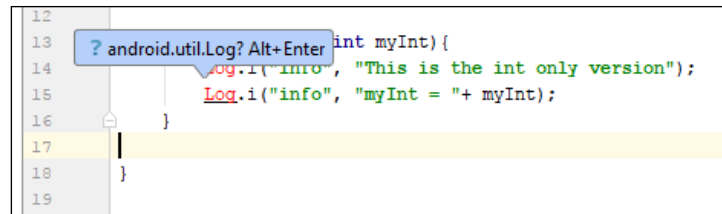
Coding the method overloading mini-app

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.

In 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 `onCreate` 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...` has 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) to see what to type but Android Studio can make this easier for us. You might have noticed Android Studio flashes up a quick message as shown in this image.



Click on one of the red-highlighted `Log` codes and the message will reappear. Hold the `Alt` key and then tap the `Enter` key. Scroll to the top of the code file and notice that the following code has been added:

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



As the 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.

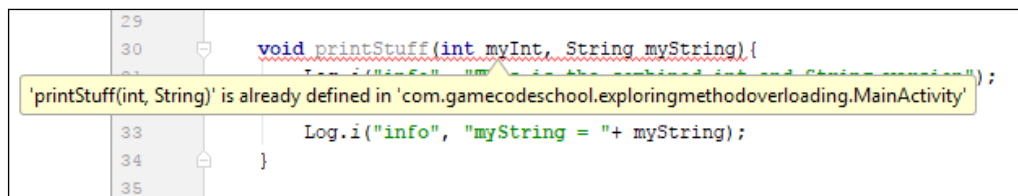
In this second method, we will also call it `printStuff` but pass in a `String` variable to be printed. Insert this method after the closing `}` of `onCreate` but before the closing `}` of `MainActivity`. Note that it doesn't matter which order we define the methods.

```
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 `onCreate` 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 exactly the same again and notice the **already defined** error.



Remove the offending method and we will write some code to see the methods in action.

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

```
// 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, it 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 the code `super.onCreate()`.

We know enough about overloading and overriding to complete this book; but if you are brave and your mind is wandering; 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 mean to call by using the appropriate arguments that match the parameters in the signature. We call each, in turn, using the appropriate parameters so 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 like `onCreate` or one of our own methods it can only be used within that method.

It is no use doing this in `onCreate`:

```
int a = 0;
```

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

```
a++;
```

We will get an error because `a` is only visible within the method it was declared. 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 it as we did in the previous code the variables can be used throughout the code file. As this and other projects progress we will declare some variables outside of all methods when they are needed in multiple methods and 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 the book progresses we will see ways to handle different situations properly.

The term used to describe this topic of whether a variable is usable is the **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 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**.



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 {
```

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

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 the book so far where we settled for a cursory explanation. We are now able to understand everything we have written. However, don't concern yourself 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 if you continue with this book and game coding that over the next year or two 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 is happening.

In the `onCreate` method, we saw this signature

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

The method receives an object of type `Bundle` named `savedInstanceState`. The `Bundle` class is a big deal in GUI based Android apps that often use more than one `Activity`. However, we will only need one `Activity` for 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 type `MotionEvent`. 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*.

This next line is the method's line of code which 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 next `Int`.

The `Random` class and the `nextInt` method

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



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 two.

```
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 type `Random`. We can do so like this:

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

Then we 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 line above 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 types of random number using `nextLong`, `nextFloat` and `nextDouble` methods.



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>

We are ready to spawn the sub'.

Adding Random based code to newGame

Add the following highlighted code in 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 first 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 line of code shown next, specifically look at the argument passed into `nextInt`.

```
subHorizontalPosition = random.nextInt(gridWidth);
```

The variable `gridWidth` 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 this is just what we need.



If we had needed non-zero numbers generated, we could have added the `+ 1` to the end of the previous line of code.

The next line of code is this:

```
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 passed to `nextInt` is `gridHeight`.

Our sub position 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 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 this abbreviated logcat output:

```
subHorizontalPosition: 30
subVerticalPosition: 13
```

Try running the game again and notice you will get different positions demonstrating that our `Random` based code is working, and we can confidently call `newGame` whenever we want to start a new game.

```
subHorizontalPosition: 14
subVerticalPosition: 7
```



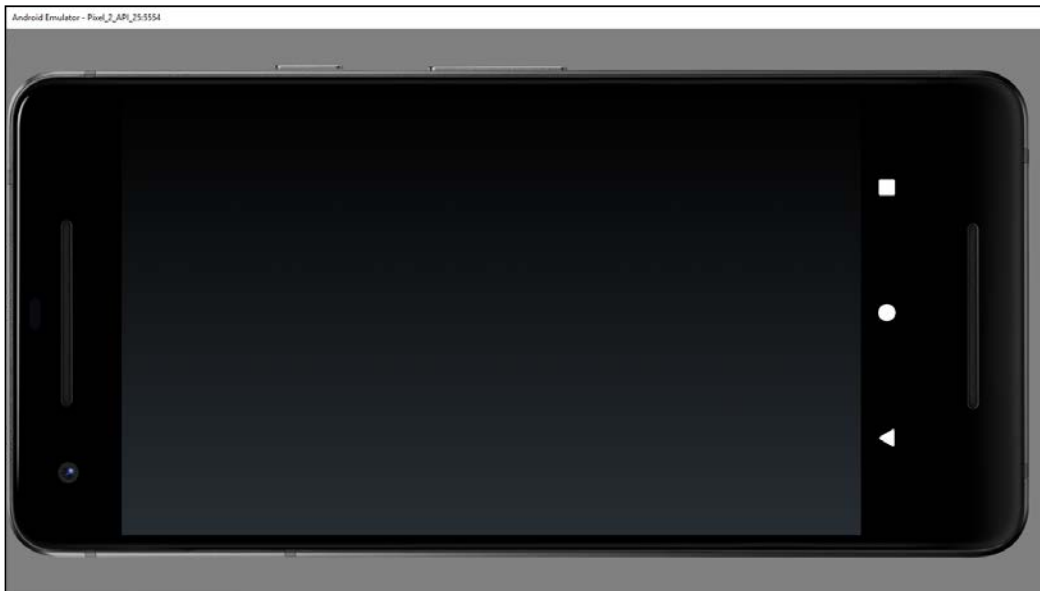
It is possible but very unlikely that you can call `newGame` and get a sub in the same position.

That's it, we now can spawn our sub' each new game.

Summary

We now know almost everything there is to know about methods. Parameters vs arguments, signatures, bodies, access specifiers and return types. The one missing link is how methods relate to classes. The class-method relationship gets 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.



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 about 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 some related classes like `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.

In this chapter, we will:

- Talk about understanding the `Canvas` and related classes
- Write a `Canvas` based demo app as practice before moving on to Sub' Hunter
- Look at the Android coordinate system so we know where to do our drawing
- Draw some graphics and text for the Sub' Hunter game

Let's draw!

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 these 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 the part of the `Canvas` to draw upon.

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.



The `Canvas` class supplies the *ability* to draw. It has all the methods for doing things like 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 like the color that we will draw on the `Bitmap` within the `Canvas`.

There is still another piece of the puzzle before we can get things drawn.

ImageView and Activity

The `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` which gets drawn on (through its association with `Canvas`) with the `ImageView`, once the drawing is done. The last step will be telling the `Activity` that our `ImageView` represents the content for the user to see.

Canvas, Bitmap, Paint and ImageView quick summary

If the theory of the interrelationship we need to set up seems like it is not simple you should breathe a sigh of relief when you see the relatively simple code very shortly.

A quick summary so far:

- Every app needs an `Activity` 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 the `View` class to suit the project at hand.
- The `Canvas` class supplies the *ability* to draw. It has all the methods for doing things like 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 upon.
- The `Canvas` class uses the `Paint` class to configure details like the color.
- Finally, once the `Bitmap` has been drawn upon we must associate it with the `ImageView` which in turn is set as the view for the `Activity`

The result is that what we drew on the `Bitmap` with `Canvas` that is displayed to the player through the `ImageView`. Phew!



It doesn't matter if that isn't 100% clear. It is not you that isn't seeing things clearly- it simply isn't a clear 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 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 then we can quickly move on to drawing something, for real, with the Canvas demo app.

Preparing the objects of classes

Remember back in chapter two I said this:

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.



We have already done this with the Random class in the previous chapter and will explore more deeply 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 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 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 must do is assign the `ImageView` to the `Activity`.

Setting the Activity content

Finally, before we can see our drawing we tell Android to use our `ImageView` 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 must 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, a 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
Application name:	Canvas Demo
Company domain:	gamecodeschool.com (or your own)
Include C++ support	Leave this option unchecked
Project location:	D:\AndroidProjects\CanvasDemo

As we did before, be sure the **Empty Activity** option is selected on the **Add an Activity to Mobile** screen before clicking **Next**. Also, be sure to uncheck **Generate Layout File** and **Backwards Compatibility (AppCompat)**. Don't worry about naming the Activity this is just a mini app to play around with we will not be returning to it.



The complete code for this mini-app can be found in the download bundle in the Chapter 5/Canvas Demo folder.

Coding the Canvas demo app

To get started, add the highlighted code, after the class declaration but before the `onCreate` method. This is what the code will look like after this step.

```
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);
    }
}
```

Notice that each of the four new classes is underlined in red. This is because we need to add the appropriate `import` statements. You could copy them from the first page of this chapter but much quicker would be to place the mouse pointer on each error in turn then hold the *ALT* key and tap the *Enter* key. If prompted, from the pop-up options, select **Import class**.

Once you have done this for each of `ImageView`, `Bitmap`, `Canvas`, and `Paint`, all the errors will be gone, and the relevant `import` statements will have been added to the top of the code.

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 this next 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 any deeper about classes now it might be worth exploring the Bitmap class initialization.

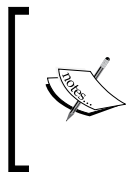
Exploring the Bitmap initialization

Bitmaps, more typically in games are used to represent game objects like the player, backgrounds, bombs, etc. Here we are simply using it to draw upon. In later projects, we will use all sorts of bitmaps to represent everything from spaceships to enemies.

The method that needs explaining is the `createBitmap` method. The parameters from left to right are:

- The width (in pixels),
- The Height (in pixels)
- The 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.



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 this next 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.rgb(255, 0, 0, 255));

// Re-size the text
myPaint.setTextSize(100);
// Change the paint to white
myPaint.setColor(Color.rgb(255, 255, 255, 255));
// Draw some text
myCanvas.drawText("Hello World!",100, 100, myPaint);

// Change the paint to yellow
myPaint.setColor(Color.rgb(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.

The `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. The `myCanvas.drawText` actually draws the text to the screen.

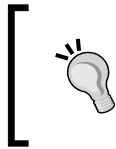
Analyze the arguments passed into `drawText` and we can see that the text will say "Hello World!" and it will be drawn 100 pixels from the left and 100 pixels from the top of our Bitmap (`myBitmap`).

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 reserved explaining the `Color.rgb` method until now.

Explaining Color.argb

The Color class, unsurprisingly helps us 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 zero(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.



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. The site helps you pick the RGB values you can then experiment 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 makes 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` means full opacity, full red, full green, full blue. When you combine light with these values you get white.

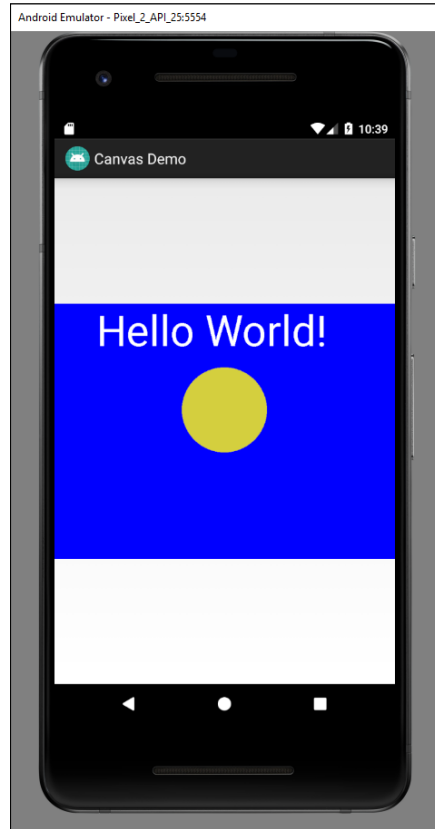
The final call to `argb` is in the final call to `setColor` when we are setting the color to draw the circle. `255, 21, 207, 62` makes a sun-yellow color.

The last step before we can run the code is to add the call to the `setContentView` method which places passes our `ImageView` (`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 by 800-pixel drawing. In all our games, we will see how we can utilize the entire screen.



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` is trivial. But the coordinate system that we use to draw our graphics onto needs 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.

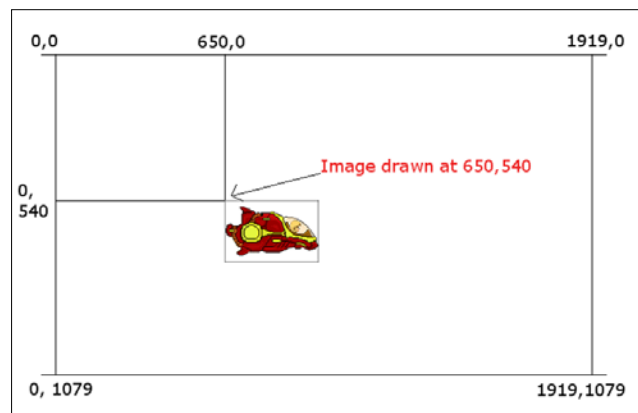
For example, the Google Pixel phone has a screen resolution of 1920 pixels (across) by 1080 pixels (down) when held in landscape view.

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` or any other drawable to the screen (like `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?

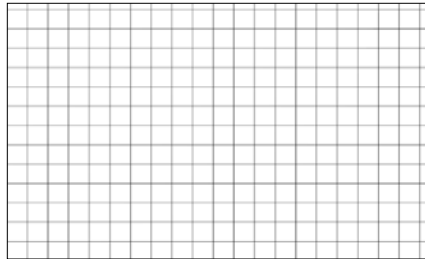
The answer is the top left pixel of the `Bitmap` object. Look at the next image which should clarify the screen coordinates using the Google Pixel phone as an example.



So, let's just bare 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 meet another method of the Canvas class called `drawLine`. Unsurprisingly this will be used to draw the grid lines.



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 reference variables we will need. The new code is highlighted amongst 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 `onCreate` to initialize our Canvas and other drawing objects.

Initializing a Canvas, Paint, ImageView, and Bitmap

Now we can initialize our drawing based objects all in `onCreate`. Add the highlighted code.

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

    // 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;
```

```

// 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. With one small but crucial exception.

You might be wondering where the call to `setImageBitmap` is? As we will need to draw the image over and over as the image will be slightly different each time we have moved the call to `setImageBitmap` to 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 end 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 `drawColor` using the `argb` method to pass in the required values (255, 255, 255, 255) for a blank white screen.

Run the game and check that you have the same result as this next image.



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. Now 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 grid lines

Let's draw the first horizontal and vertical lines using the `canvas.drawLine` method. However, soon we will see there is a gap in our 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();
}
```

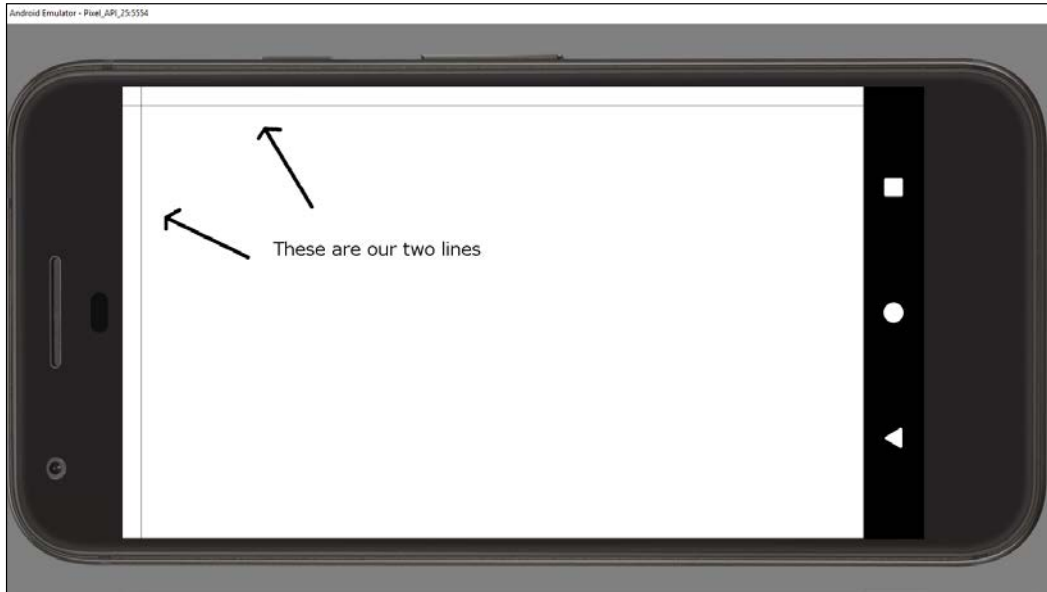
The first new line of code calls `setColor` and changes the drawing color to black. The next line of code calls `drawLine`. The parameters for `drawLine` can be described as follows.

```
(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 top left to 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.



We need to be able to run the same code over and over while each time adding one to the amount we multiply `blockSize` by. Like this, `blockSize * 1, blockSize * 2, blockSize * 3` and so on.

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 this 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 `setColor` 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 variable `shotsTaken` then concatenate onto that the word " Distance: " followed by the variable `distanceFromSub`.

The next two parameters determine the horizontal and vertical coordinates of the text. As with the grid lines, 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 type `Paint`.

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.

```
// This code prints the debugging text
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);

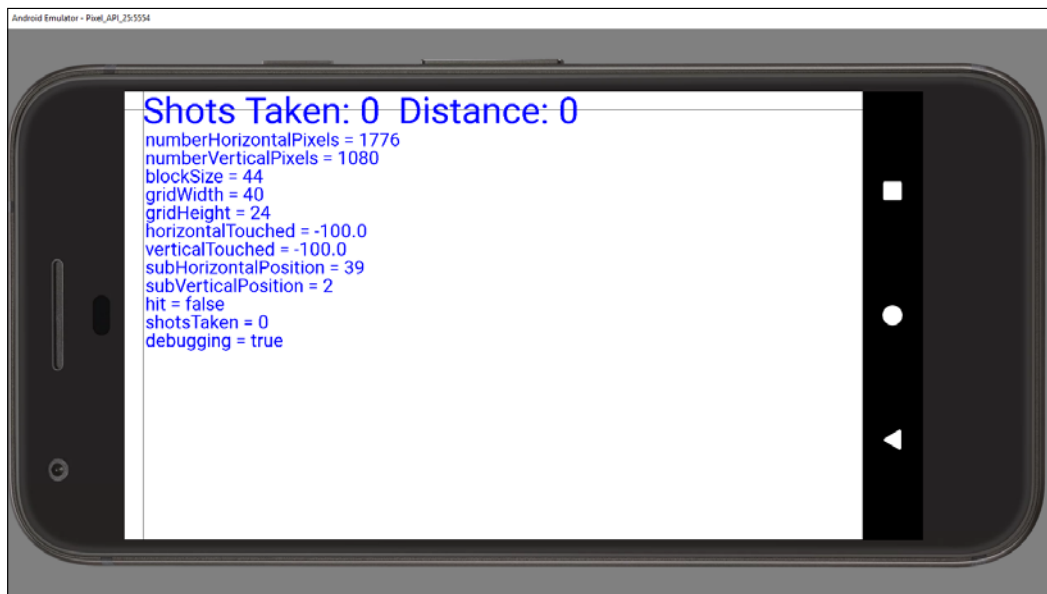
        canvas.drawText("hit = " + hit,
            50, blockSize * 12, paint);

        canvas.drawText("shotsTaken = " +
            shotsTaken,
            50, blockSize * 13, paint);

        canvas.drawText("debugging = " + debugging,
            50, blockSize * 12, 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 this next image.



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 graphically speaking quicker to achieve. We will use, `Canvas`, `Paint` (and some more related classes) in every project throughout this book.

Let's 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 grid lines of Sub' Hunter.

6

Repeating Blocks of Code with Loops

In this brief chapter, we will learn about Java loops that 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 throughout the rest of the book we will put each of them to good use.

In this chapter, we will look at:

- 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 have completed the first game and move on to a more advanced project.

Decisions, decisions

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 upon whether it did or not.

This chapter and the next will look at controlling the flow of execution of our game's code. This chapter, as the name suggests will discuss fine control over repeating sections of code based on predetermined conditions and the next chapter will look at branching to different sections of code, also based on predetermined conditions.

In the next two chapters, our code gets more in-depth and 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 the first line of code is indented inside each method. Here is an annotated image to make this clear and as another quick example:

```
void draw() {  
    →Log.d( tag: "Debugging",  msg: "In draw");  
    →printDebuggingText();  
}
```

Notice also that when the indented block has ended, often with a closing curly brace `}` that `}` is indented to the same extent as the line of code which began the block.



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 gets more complicated, indenting along with comments, help to keep the meaning and structure of our code clear. I mention 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 the same way.

Much of this indenting is done for us by Android Studio but not all of it.

Now we know how to present our code more clearly let's learn some more operators and then we can really get to work taking 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 go straight on to understand how to use them in Java.



Don't worry about memorizing every operator below. Take a glance at them and their explanations and then move quickly on to the next section. We will put some operators to use soon and they will become much clearer as we see 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 operators plain from the start. The list will also be more convenient to refer back to when not intermingled with the discussion about implementation that follows it.

We use operators to create an expression which is either true or false. We wrap that expression in parentheses like this: `(expression goes here)`.

- The **comparison** operator (`==`). This tests for equality and is either true or false. An expression like `(10 == 9)`, for example, is false. 10 is obviously not equal to 9.
- The logical **NOT** operator (`!`). The expression `(! (2+2 == 5))` tests if something is true because `2 + 2` is NOT 5.
- Another comparison operator (`!=`). This tests if something is **NOT equal**. For example, the expression `(10 != 9)` is true. 10 is not equal to 9.
- Another comparison operator (`>`). This tests if something is **greater than** something else. The expression `(10 > 9)` is true.
- You can probably guess what this does (`<`). This tests for values **less than**. The expression `(10 < 9)` 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 expression: `(10 >= 9)` is true. The expression `(10 >= 10)` is also true.
- Like the previous operator (`<=`), this one tests for two conditions but this time **less than or equal** to. The expression `(10 <= 9)` is false. The expression `(10 <= 10)` 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 in order for the result to be true. Logical AND is usually used in conjunction with the other operators to build more complex tests. The expression `((10 > 9) && (10 < 11))` is true because both parts are true, so the expression is true. The expression `((10 > 9) && (10 < 9))` is false because only one part of the expression is true, and 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, need to be **true** for the expression to be true. Let's look at the last example we used above but switch the `&&` for `||`. The expression `((10 > 9) || (10 < 9))` is now true because one part of the expression is true.

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 with loops.

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 the Java **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 the most appropriate situations to use the different types of loops.

We will look at all the major types of loops that Java offers us to control our code and we will use some of them to implement a working mini-app to make sure we understand them completely, then we can add some code to take advantage of loops in the Sub' Hunter game. Let us 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 loop executes. If it is false it doesn't. Take a look at this code:

```
int x = 10;

while(x > 0){
    x--;
    // x decreases by one each pass through the loop
}
```

What happens here is this: Outside of 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 code above 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 of course no longer greater than 0, the program will exit the loop and continue with the first line of code after the `while` loop.

It is important to realize that it is possible that the `while` loop will 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 body.

```
int lives = 3;
int aliens = 12;

while(lives > 0 || aliens > 0){
    // keep playing game
    // etc.
}

// continue here when lives and aliens equal 0
```

The above `while` loop would continue to execute until both `lives` and `aliens` were equal to or less than zero. As the condition uses logical OR operator `||` either one 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, part way through, as it is not tested again until the code tries to start another pass. For 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 above loop body will execute exactly once. We can also set a `while` loop that will run forever; called an **infinite loop**. Here is one.

```
int x = 0;

while(true){
    x++; // I am going to get mighty big!
}
```

The keyword `true` unsurprisingly evaluates to true. So, the loop condition is met. We need an exit plan.

Breaking out of a loop

We might use an infinite loop like this next code so that we can decide when to exit the loop from within its body. We would do this by using the `break` keyword when we are ready to leave the loop body. Like this:

```
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 like `if`, `else`, and `switch` within our `while` loops and all the rest of the loops we will look at in a minute. Here is a sneak look ahead at the `if` keyword. For 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 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 `while` loops with the exception that a `do while` loop evaluates its expression *after* the body. This means that a `do while` loop will always execute at least once before checking the loop condition

```
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.



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 while or do while loop as they take three parts to initialize. Have a look at the 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 put like this:

```
for(declaration and initialization; condition;  
change after each pass through the loop).
```

To clarify further we have:

- **Declaration and initialization:** We create a new int variable i and initialize it to zero.
- **Condition:** Just like the other loops, it refers to the condition that must evaluate to true for the loop to continue.
- **Change after each pass through the loop:** In the example i++ means that 1 is added/incremented to i on each pass. We could also use i-- to reduce/decrement i each pass.

```
for(int i = 10; i > 0; i--){  
    // countdown  
}  
// blast off i = 0
```



Note that break can also be used in for loops.

The for loop essentially takes control of initialization, condition evaluation and the control variable on itself.

Nested loops

We can also nest loops within loops. For example, take a look at this next 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 each and 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 code like the above to loop through a game board that is twenty spaces wide and ten spaces high.

Also note that loops of any type can be nested with any other loop. We will do this in most of the projects in this book.

Using for loops to draw the Sub' Hunter grid

By the end of the book we will have used every type of loop but the first one we get to utilize is the `for` loop. Can you imagine having to write a line of code to draw each and 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 show you the entire `draw` method just to be sure you can clearly recognize what to delete and what to add. Add the highlighted code shown next.

```

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 the code let's focus on the second `for` loop. This one:

```
// 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 code declares a new variable named `i` and initializes it to 0.

Then the loop condition is set to `i < gridHeight`. This means that all the time that `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 that `gridHeight` is 24 as it is on the Google Pixel emulator, 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. It is the limitations of printing in a book which 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`. This has the effect in the first part of the loop of drawing 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 `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.

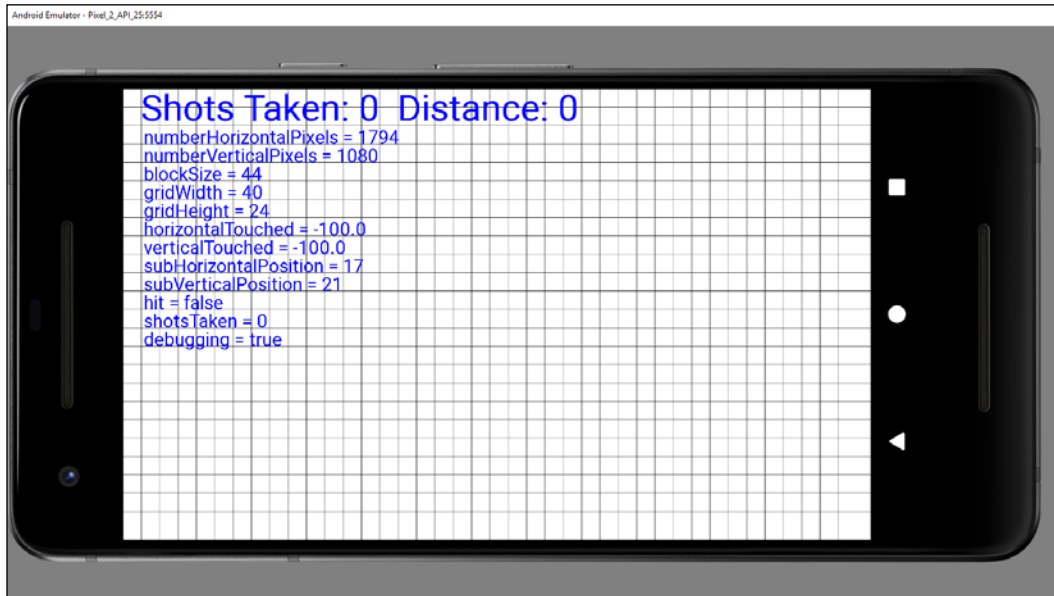
The coordinates to draw the line to are determined by `numberHorizontalPixels, blockSize * i`. This is exactly opposite the starting coordinate on the far right of the screen.

As `i` is incremented each pass through the loop the line moves the exactly correct number of pixels down each time. The process ends when `i` is no longer lower than `gridHeight`. If it didn't 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) works in exactly 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.



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 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`) we just had a quick look at. This time we will see how to branch 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 Java basics. In this chapter, we will 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.

In this chapter we will learn:

- A slightly contrived example to help remember how to use `if` and `else`
- Switching 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. They are just what we need for giving 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. Next follows a made up military situation that is kind of game-like in its 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 `isComingOverBridge`. The next bit of 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 introduce another Java keyword, **else**. When we want to explicitly do something when the `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 this code:

```
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 this 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 above code has three possible paths of execution. First, if the enemy is coming over the bridge and the friendly troops are greater in number.

```
if(isComingOverBridge && friendlyTroops > enemyTroops)
```

Second, if the enemy troops are coming over the bridge and outnumber the friendly troops.

```
else if(isComingOverBridge && 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.



Reader challenge

Can you spot a flaw in the above 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` 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 was obvious. Using the `wavingWhiteFlag` Boolean variable he wrote this test.

```
if (wavingWhiteFlag) {  
  
    // Take prisoners  
  
}
```

But where to put this code was less clear. In the end, the captain opted for the following nested solution and changing 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 of 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 very 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 and 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 better made in other ways.

When we must decide based on a clear list of possibilities that don't involve complex combinations, then the **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 `case` and `break` elements.

```
case x:
    // code to for case x
    break;

case y:
    // code for case y
    break;
```


You can see in the previous example each `case` states a possible result and each `break` denotes the end of that `case` and the point at which no further `case` statements will be evaluated.

The first `break` 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 like "Go East", "Go West" and "Take Sword", etc. In this case, `switch` could handle that situation like this example code and we could use `default` to handle the player typing a command which 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 East" );
        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 upon 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. Perhaps, like this next piece of code, I have highlighted the new line.

```
default:
    goWest();
    break;
```

Of course, we would then need to write the new `goWest` method. Then, when the `command` variable was initialized to `"go west"`, the `goWest` method would be executed and execution would return to the `break` statement which would cause the code to continue after the `switch` block.

Combining different control flow blocks

And you might have been able to guess that we can combine any of the decision-making tools like `if`, `else`, and `switch` within our `while` loops and all the rest of the loops too. For 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 we are confident with `if` and `else`, let's have a look at one more concept to do with loops. So here is one last concept combined with loops.

Continue...

The **continue** keyword acts in a similar way to **break**- that we learned about in the previous chapter- up to a point. The `continue` keyword will break out of the loop body but will also check the condition expression afterward so the loop *could* 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
}
```

The `continue` keyword can be used in all the loops just as the `break` keyword can.

Making sense of the 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 two. 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 first determine the actual event type and secondly 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 sketchy, our knowledge of methods should help us work out what is going on here. An object of type `MotionEvent` named `motionEvent` is passed into the method.

If we want to know the details of the touch event then it is this object, `motionEvent` that we will need to unpack.

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);
```

That had two variables contained within it, `x` and `y` as shown again highlighted next.

```
// 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 of it and 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 I said some of it. The `MotionEvent` class is quite extensive. It contains within it dozens of methods and variables.



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://stuff.mit.edu/afs/sipb/project/android/docs/reference/android/view/MotionEvent.html>. Note that it is not necessary to do further research to complete this book.

For now, all we need to know is what were 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 that 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 which with the help of a little bit more Java trickery can be used to filter the data from `getAction`.
- The `ACTION_UP` variable that we can use to compare to see if the action performed is the one (removing a finger) we want to respond to.
- The `getX` method tells us a horizontal floating-point coordinate where the event happened
- The `getY` method tells us a 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. Not to be confused with the logical `&&` operator we have been using in conjunction with the `if` keyword and loops.

The `&` bitwise operator checks to see if each corresponding part in two values is true. This is the filter that is required when using `ACTION_MASK` with `getAction`.



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 write in the next section determines the event type the player has just triggered, that is all you need to know. I just guessed that a discerning reader such as yourself would like to know the ins and outs. In summary, if you understand bitwise operators, great you are good to go. If you don't, 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

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` because it was just temporary code and it needs an upgrade to handle passing some values into the method. We will upgrade the `takeShot` signature to be compatible with this next code soon.

Code your `onTouchEvent` method to look exactly like this next code then we will discuss it.

```
/*
    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 look closely at the line of code that determines whether the player removed their finger from the screen.

It starts with an `if (...` and at the end of the `if` condition we have the body with some more code that we will look at next. It is the condition which 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 using `==` to `motionEvent.ACTIONUP`.

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.

 You could summarize and explain the entire discussion on 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 using `motionEvent.getX` and `motionEvent.getY` as arguments. If you remember the section *Making sense of screen touches*, 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 finish.

Final tasks

We only have a few more methods to complete and we have already coded the trickiest bits. The `takeShot` method having 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 see how we translate the floating-point coordinates retrieved in `onTouchEvent` to 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. He can then analyze the last shot compared to the distance and guess at his next shot.

Coding the `takeShot` method

Let's redo the entire `takeShot` method because the signature is changing too. Adapt the `takeShot` method to look like this and then we will analyze the code.

 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` passing in the `float` screen coordinates of the player's shot.

```
void takeShot(float touchX, float touchY){
```

This next 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 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, this 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;
```

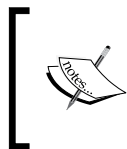
The next line of code in the `takeShot` method assigns either true or false to the `hit` variable. Using logical `&&` we see whether **both** of `horizontalTouched` and `verticalTouched` are the same as `subHorizontalPosition` and `subVerticalPosition`, respectively. We can now use `hit` later in the method to decide whether to call `boom` or `draw`.

```
// Did the shot hit the sub?
hit = horizontalTouched == subHorizontalPosition
    && verticalTouched == subVerticalPosition;
```

In case the player missed (and usually they will) we calculate the distance of the shot from the sub'. The first part of the code shown next 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)));
```



If the `Math.sqrt` looks a bit odd as did the `Random.nextInt` method, then 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 passed in as an argument.

Finally, for the `takeShot` method, we call `boom` if there was a hit or `draw` if it 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 new highlighted code inside the `boom` method and 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 as the `draw` method. In the previous code, we set `blankBitmap` to be viewed.

```
gameView.setImageBitmap(blankBitmap);
```

Fill 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 use 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 centre 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 of the next game.

```
canvas.drawText("Take a shot to start again",
               blockSize * 8,
               blockSize * 18, paint);
```

Restart the game ready to receive the first shot.

```
// Start a new game
newGame();
```

There is one final task. We need to draw the location of the player's shot on the grid.

Drawing the shot on the grid

Add the highlighted code to the draw method as shown next. Note 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 in the correct place.

```
// Draw the horizontal lines of the grid
for(int i = 0; i < gridHeight; i++){
    canvas.drawLine(0, blockSize * i,
```

```
        numberHorizontalPixels, blockSize * i,
        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.

1. The left `horizontalTouched * blockSize`
2. The top `verticalTouched * blockSize`
3. The right `(horizontalTouched * blockSize) + blockSize`
4. The bottom `verticalTouched * blockSize)+ blockSize`
5. Our Paint instance, `paint`

See how we use a simple calculation on the key variables `horizontalTouched`, `verticalTouched` and `blockSize` 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;
```

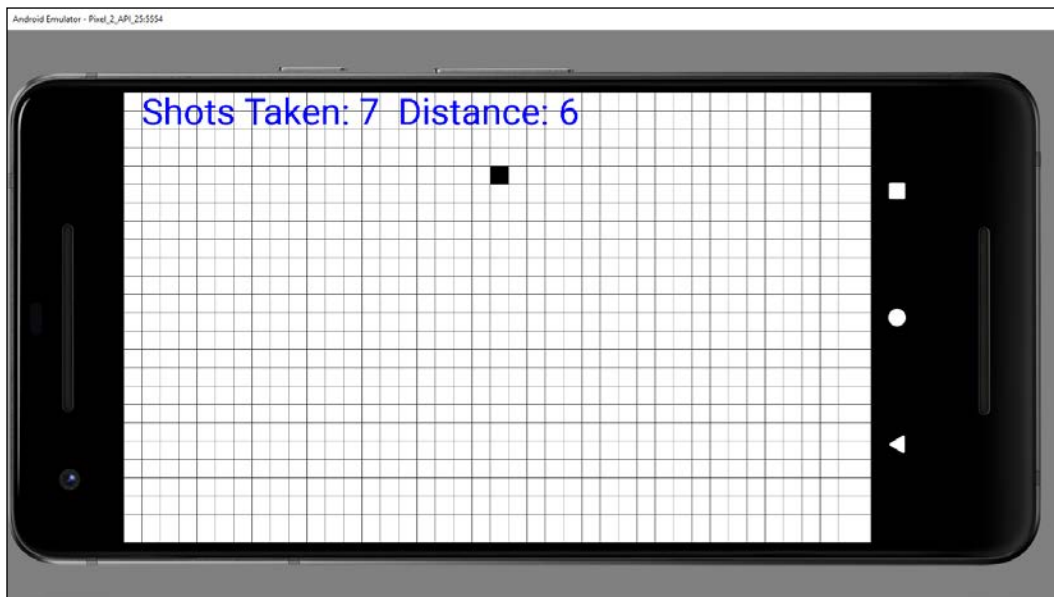
As the absolute last thing, still inside the `draw` method at the very end, change the call to `printDebuggingText()` by wrapping it in an `if` statement that will cause the debugging text to be drawn when `debugging` is set to `true` but otherwise not.

```
if (debugging) {  
    printDebuggingText();  
}
```

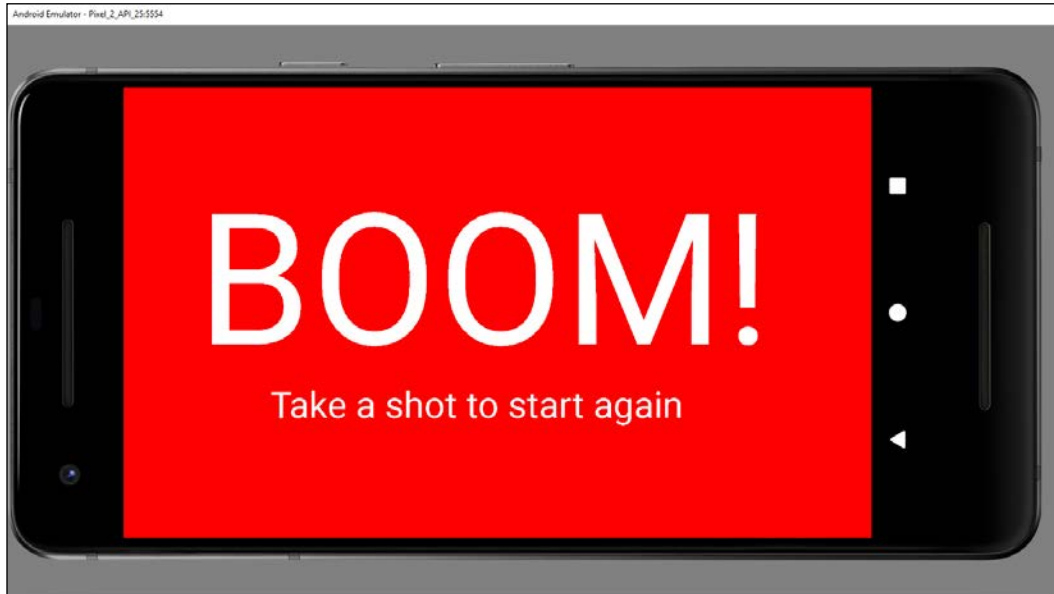
The Sub' Hunter game is complete!

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.



And when you shoot the sub'...BOOM!



As suggested in the image, taking a shot (tapping the screen will start the game all over again.)

Summary

We have learned a lot in this chapter as well as put much of it to practical use. We have studied and implemented `if` and `else` blocks. We have 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 the 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.

- Introduction to Object-Oriented Programming including encapsulation, inheritance, and polymorphism
- Write and use our first, very own class
- An explanation about 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
- Start the next project- Pong

Let's get started.

Basic Object-Oriented Programming

I am not going to throw the whole OOP book at you in one go. We will return to and expand upon our OOP knowledge as the book progresses and the games get more advanced.



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, like 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 cars mechanical systems will not make you a great driver. Understanding the options and possibilities of their interfaces (steering wheel, engine performance, breaks etc.), then practicing and testing will serve you much better. A good goal by the end of this chapter will be to try and *just-about get it*.



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 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.

Each chunk is self-contained yet potentially reusable by other parts of our code and even by other games and programs while 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. 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.

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 a step at a time is reasonably straightforward.

Encapsulation

Encapsulation: This means keeping the internal workings of your code safe from unwanted control by 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.

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

With good encapsulation, it doesn't matter if the Satellite company or the Android API team need 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 hover car; 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 peoples' classes, including the encapsulation and polymorphism, while further refining their code specifically to our situation. We have done this already, every time we used the `extends` keyword.

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

The `Activity` class provided the Sub Hunter game with features like:

- 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 and not worry too much about the code that implements them.

Polymorphism

Polymorphism allows us to write code that is less dependent on the *types* we are trying to manipulate, like `Invaders`, `Bullets`, and `Platforms` and allows us to deal with more generic things like 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 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 less 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 into manageable parts and types of "thing". By thing, you can probably guess I mean class.

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.

Makes sense really. And 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 a 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 so far. We created a class in the first seven chapters. It was called `SubHunter` and it inherited the features of `Activity` provided by Android. Next is the line of code which 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 features `getWindowManager`, `getDefaultDisplay`, and the `onTouchEvent` methods.



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 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, 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 war simulation game. It is a game where you get to micro-manage your troops in battle. Amongst others, we would probably need a class to represent a soldier.

Class implementation

Here is real code for our hypothetical class for our hypothetical game. We call it a class **implementation**. As the class is called `Soldier`, if we 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  
    }  
  
}
```

Above 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 a `void` return type, but class methods can be of any shape or size that 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 whichever fancy name they are referred to by. Although 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.

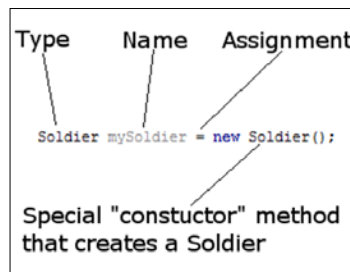
Declaring, initializing and using an object of the class

Remember that `Soldier` is just a class, not an actual 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 type `Soldier` from our `Soldier` class.

```
Soldier mySoldier = new Soldier();
```

The previous code can be broken into three main parts:

1. In the first part of the code, `Soldier mySoldier` declares a new variable of type `Soldier` 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. The next image summarizes all this information.
4. Consider this visually with the next image.



This is not far 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.

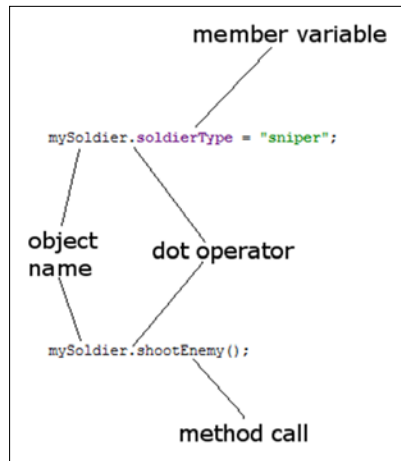
```
mySoldier.health = 100;
mySoldier.soldierType = "sniper";

// Notice that we use the object name, mySoldier.
// Not the class name, Soldier.
// We didn't do this:
// Soldier.health = 100;
// ERROR!
```

Above, the dot operator `.` is used to access the variables of the object. And this is how we would call the method. Again, by using the object name and not the class name and followed by the dot operator.

```
mySoldier.shootEnemy();
```

We can summarize the use of the dot operator with a diagram.

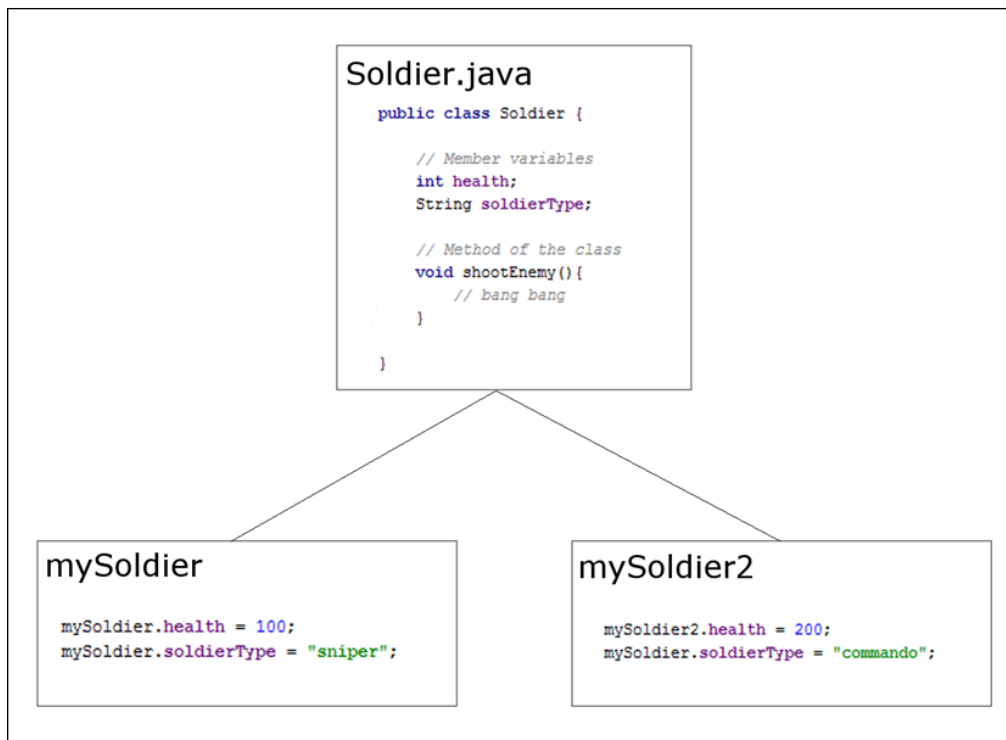


 We can think of a class' methods as what it can *do* and its instance/member variables as what it *knows* about itself. Methods act on data, variables are the data.

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 `mySoldier`.



What is also key here is that this previous code would not be written within the class itself. For example, we could 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 therefore contained within that memory too.


 As always there are exceptions to this rule. But they are in the minority and we will look at the 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

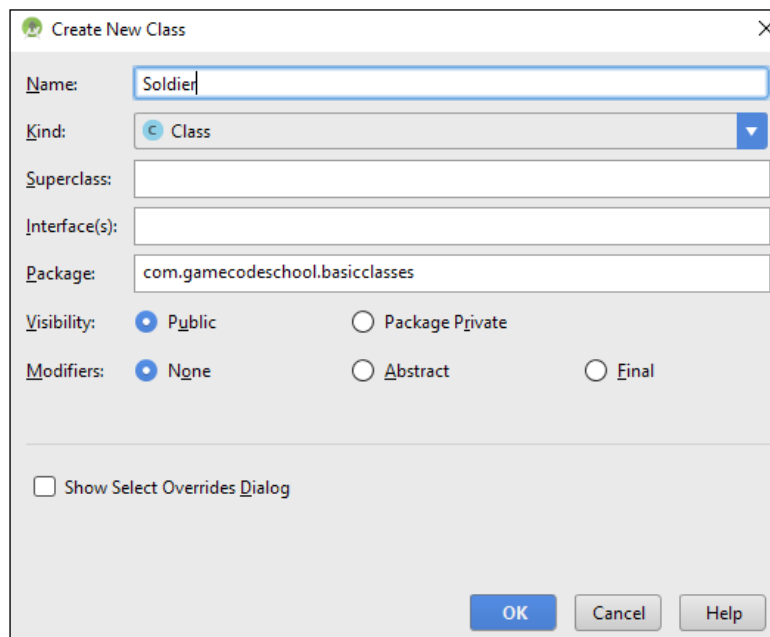
The hypothetical **real-time strategy** (RTS) game we are writing will need more than one `Soldier` object. In our game that we are about to build we will instantiate and use multiple objects. We will also demonstrate using the dot operator on variables and methods to show that different objects have their own instance variables contained in their own memory slot.

You can get the completed code for this example in the download bundle. 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 call the application `BasicClasses`. Choose the **Empty Activity** template. Call the Activity `RTSActivity`. As usual, we don't need the **Generate Layout File** or **Backwards Compatibility** options. It doesn't really matter too much as we won't be returning to this project after this short exercise.

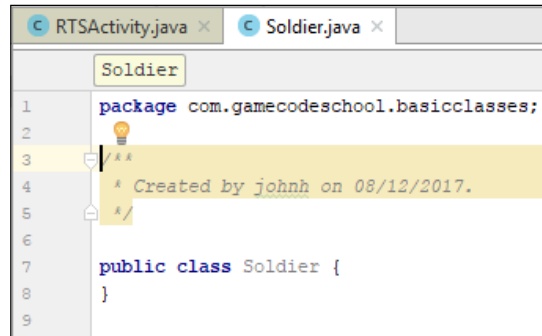
Creating your first class

After creating the new project we create a new class called `Soldier`. Select **File | New | Java Class**. You will see the following dialog box.



Name the class as **Soldier** as I have done in the previous image and click **OK**.

The new class is created for us with a code template ready to put our implementation within. Just like the next image shows.



Also, notice that Android Studio has put the class in the same package as the rest of our app. And now we can write its implementation.

Write the class implementation code below within the opening and closing curly braces of the `Soldier` class as shown. 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 hold the *ALT* key and tap *Enter* to import the class.

Now we have a class, a blueprint for our future objects of type `Soldier`, we can start to build our army. In the editor window left-click the tab of **RTSActivity.java**. We will write this code, as so often, within the `onCreate` method just after the call to **super.onCreate**. Type this code.

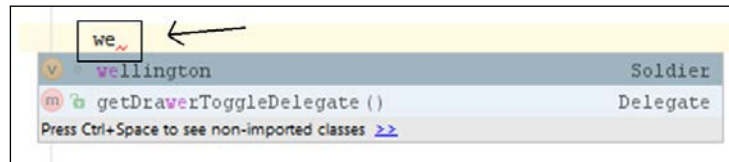
```
// first we make an object of type soldier
Soldier rambo = new Soldier();
rambo.soldierType = "Green Beret";
rambo.health = 150;
// It takes allot to kill Rambo
```

```
// 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
```



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, all you must do is begin typing the object's name and all the auto-complete options present themselves. The next image 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.



Now we have our extremely varied and somewhat unlikely RTS army, we can use it and also verify the identity of each object. Type this 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();
```



You will need to import the **Log** class into **RTSActivity** just as you did previously for **Soldier**.

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), an `int` and a `String` 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 the logcat. First, the `String` `soldierType` and then the arbitrary `" is shooting"`. What is neat here is that the `String` `soldierType` will be different depending upon which object we call the `shootEnemy` method on. This demonstrates that each of the objects of type `Soldier` is different and stored in their own memory slot.

Next, we declared and created three new objects of type `Soldier`. 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 console. 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 above example might raise questions like, should the `healSoldier` method be a method of a class?

```
fieldhospital.healSoldier(rambo);
```

It could be or not. It would depend upon what is the best solution for the situation. We will look at more OOP and then the best solution for lots of similar conundrums should present themselves more easily.

And, as you might guess we can also 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 type `Point` called `point` and passed it as an argument to the `getSize` method using 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.

Remember that encapsulation thing?

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

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 uses 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 you say we don't have to know what is going on inside? So you might question what we have seen so far like this.



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/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 at this encapsulation thing. But even at a glance, the access modifiers described are not very fine grained. We seem to be limited to:

- Completely locking down to anything outside of the package (default)
- or 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 default and public classes.

In addition to class level privacy controls, Java gives us programmers more fine-grained controls, but to use these controls we have to look into variables with 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();
```



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 class 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.



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 subclasses; that is, when we extend a class like 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. Including, like default access, they cannot be seen by subclasses (classes that inherit from the class in question).



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 default and public classes. Within these classes, variables will have carefully chosen and most likely varied access modifiers, chosen with a view to 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

It 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, well, use your class.

The access modifiers 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 useMeEverybody(){
    // do something everyone needs to do here
}
```

Whereas this method could only be used internally by the class that contains it.

```
private 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!
    */
}
```

And 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.

```
fairlySecretTask(){
    // allow just the classes in the package
    // Not for external use
}
```

As the 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 onCreate.

```
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. Thereby we achieve our encapsulation goals, like keeping the internal workings of your 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, 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 class `Hospital` wanted access to the health member variable from an object of type `Soldier`, 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. It is 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 their own file but 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 which 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 getter method `getHealth`, 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 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 which unsurprisingly returns the value held in the private `health` `int` variable. As this method is public, any code with access to an object of type `Soldier` can use it.

Next, the `setHealth` method is implemented. Again, it is public but this time it takes an `int` as a parameter and assigns whatever is passed in, to the private `health` variable.

[ In a more life-like example, we would write some more code here to make sure the value passed in to `setHealth` is within the bounds we expect.]

Now we declare, create, and assign 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 type `Soldier`. Not only that, we can 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.



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 is probably raising more questions than it answers. That's good.

By using encapsulation features (like access control) 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.



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, like 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 which uses OOP. In addition, we will be using classes from the Android API that use it extensively and it 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 type `Soldier` and assign it to an object called `mySoldier`.

```
Soldier mySoldier = new Soldier();
```

Nothing new here but look at the last part of that line of code.

```
...Soldier();
```

This looks suspiciously like a method call.

All along, we have been calling a special method called a constructor that has been created, behind the scenes, automatically, by the compiler.

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 like 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 which 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.



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 do 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 or the parameter.

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 the `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.

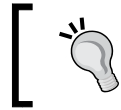
There are more twists and turns to be learned about variables and `this` and they make much more sense when applied to a practical project. In the next mini-app, we will explore all we have learned so far in this chapter and some more new ideas too.

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.



We can think of this as a static method belonging to the class and all other methods belonging to an object or an 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 type `String`.



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 which have uses that are so generic 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.



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.

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, more a demonstration to help understand access modifiers for classes, methods, and variables alongside the different types of a variable like 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 of the download bundle. It is called `Access Scope This And Static`.

Create a new **Empty Activity** project and call it `Access Scope This And Static`. Leave the Activity name at the default, `MainActivity`.

 Creating a new project will hopefully be straightforward by now. Refer back to *Chapter 1* 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`.

 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**, `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 one too) the private static `numShips` variable. We will see that this will happen each time we create a new object of type `AlienShip`. Also, the constructor sets a value for the private variable `shieldStrength` 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;
}
```

And this is our private `setShieldStrength` method. We could have just set `shieldStrength` directly from within the class but the code below 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 if that object's `shieldStrength` is zero. If it is, it calls the `destroyShip` method that we 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 decrement the `numShips` static variable so we can keep track of how many objects of type `AlienShip` 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();
```

Look 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 getter method getShieldStrength to print out the shieldStrength which 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 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 shieldStrngth: ", "" + girlShip.getShieldStrength());

Log.i("boyShip shieldStrngth: ", "" + boyShip.getShieldStrength());
```

```

boyShip.hitDetected(); // ahhh

Log.i("girlShip shieldStrngth: ", "" + girlShip.getShieldStrength());

Log.i("boyShip shieldStrngth: ", "" + boyShip.getShieldStrength());

```

When we think we have destroyed a ship, we again use our static `getNumShips` method to see if our static variable `numShips` was changed by the `destroyShip` method.

```
Log.i("numShips: ", "" + AlienShip.getNumShips());
```

Run the mini-app and look at the logcat output.

```

numShips: 2
girlShip shieldStrngth: 100
boyShip shieldStrngth: 100
Incoming: Bam!!
girlShip shieldStrngth:: 75
boyShip shieldStrngth:: 100
Incoming: Bam!!
Incoming: Bam!!
Incoming: Bam!!
girlShip shieldStrngth:: 75
boyShip shieldStrngth:: 25
Incoming: Bam!!
Explosion: Andre LaMothe destroyed
girlShip shieldStrngth: 75
boyShip shieldStrngth: 0
numShips: 1
boyShip shieldStrngth: 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 which 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 like 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. Here is the code from the previous mini-app:

```
public class MainActivity extends Activity ...
```

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 towards 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 and 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 = 57;
        this.weight = 1000;
        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`. An `Elephant` can now do anything an `Animal` can and it also has an instance of all its variables. We initialized the variables from `Animal` which `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` 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 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 which is extended from is the **superclass** and the class which inherits from the superclass is the **subclass**. We can also say parent and child class.



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, 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, 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 which 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`. 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 class demo `AlienShip`. The differences are that the constructor now takes an `int` parameter which 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` that is declared `abstract`. This guarantees that any class that subclasses `AlienShip` must implement their own version of `fireWeapon`. Notice the class has the `abstract` keyword as part of its declaration. We must do this because one of its methods also uses the keyword `abstract`. 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. 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 just print out the 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 just print out the 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");
    }
}
```

And 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 which uses our three new classes. The code looks quite ordinary, nothing new, it is the output which 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.

Supposing we want to use polymorphism to help write a zoo management game. We will probably want to have a method like `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 method above has an `Animal` as a parameter which means that ANY object which is built from a class which extends `Animal` can be passed into it. And as you can see in the code above the method also returns an `Animal` 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 which calls the method.

For example, we could handle an `Elephant` being passed into 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` from the returned `Animal`. This is called **casting**. We have already seen casting with primitive variables in *Chapter 3, Variables, Operators, and Expressions*.

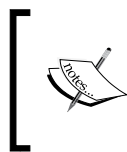
So you can even write code *today* and make another subclass in a week, month, or year, and the very same methods and data structures will still work.

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`?

Abstract classes

An **abstract class** is a class that cannot be instantiated, cannot be made into an object. We saw this in the inheritance mini-app.



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 wants to force us to inherit *before* we use their class, they 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 a method `abstract` 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 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 which extends it. This means that the abstract class can provide some of the common functionality that would be available in all its subclasses.

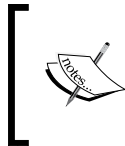
For example, the `Worker` class might have the `height`, `weight`, and `age` member variables. It might have the `getPayCheck` method which is not abstract and is the same in all the subclasses, but a `doWork` method which is abstract and must be overridden, because all the different types of worker `doWork` very differently.

Which leads us neatly 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 which 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.



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
*/

public void someAbstractMethod(){
    // code here if you like
    // but just an empty implementation will do
}

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;
}
}
```

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!* onwards we will also write our own. In the next project, one such interface we will use is the `Runnable` interface that 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

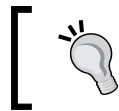
If you don't know what Pong is then you are much younger than me and you should take a look at its appearance and history before continuing. We will use everything we learned about OOP to create a class for each of the objects of the game (a bat and a ball) as well as methods within these classes to 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 3 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, 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, 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.



Once this project is complete it will serve as a template for all the remaining projects and more thorough up-front preparation *will* be practical.

Fire up Android Studio.

Setting up the Pong project

Create a new project called `Pong` with the same settings as `Sub Hunter`. This time, call the Activity `PongActivity` because that is all it will be an Activity to communicate between the OS and our game engine that will be separate.

Now we will create the three new classes that are required for this game.

- Create a new class called `PongGame` and check the **Package Private** checkbox then click the **OK** button.
- Create a new class called `Ball` and check the **Package Private** checkbox then click the **OK** button.
- Create a new class called `Bat` and check the **Package Private** checkbox then click the **OK** button.

Now lock the game to full screen and landscape orientation, just as we did for the `Sub Hunter` game but this time you need to look for the text `PongActivity`.



Reminder: locking screen to landscape and full screen.

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 which allows us to make configuration changes.

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 above. 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 these two lines to make the game run full screen and lock it in the landscape orientation.

```
    android:theme="@android:style/Theme.NoTitleBar.
Fullscreen"
    android:screenOrientation="landscape"
```

For further details refer to section Locking the game to full screen and landscape orientation from *Chapter 1, Java, Android and Game Development*.

We have declared our empty classes and we are ready to get coding.

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 peoples' 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 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

During 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, 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. Furthermore, this game engine code will be used as an approximate template (we will improve it each project) for future projects making the realization of future games faster and easier.

In this chapter, we will be:

- Coding the `PongGame` class
- Coding the `draw` method and a few more besides
- 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 OS 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 more simple and easy to understand.

Rather than have a class called `Pong` (analogous to `SubHunter`) that does everything, we will have a class which just handles start-up and shutdown of our game as well as help a bit with initialization by getting the screen resolution. It makes sense that this class will be of type `Activity`.

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. We called this class `PongActivity` and it was auto-generated for us when we created the project in the previous chapter.

Add the first part of the code for the `PongActivity` class.

```
import android.app.Activity;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;

public class PongActivity extends Activity {

    private PongGame mPongGame;

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

        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 very familiar. In fact, except for the `mPongGame` object of type `PongGame` all the code in the previous block, we saw the same 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 in full detail what we are doing with this new object. 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 to avoid getting confused about the scope of our variables. Also notice it is declared as a member should be, outside of any methods.



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 getting confused about which variables belong to the instance of the class and which only have scope within a method.

In `onCreate` after we have done our usual thing with `display` etc, 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, `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 like 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` as the content for the app. Remember that the `Activity` class' `setContentView` method must take a `View` object and `ImageView` is a `View`. 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`. Not yet anyway.

We will fix the constructor and the not-a-`View` problem after we add a few more lines of code to `PongActivity`.



Add these two overridden methods and then we will talk about them. Add them below the closing curly brace of `onCreate` but before the closing curly brace of `PongActivity`.

```
@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 to override 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 OS 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.

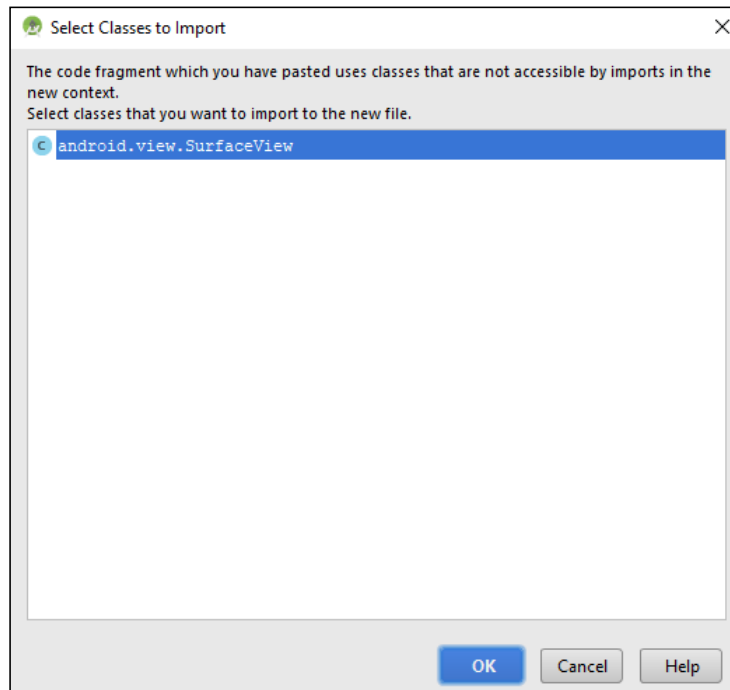
It makes sense at this point to move on to the `PongGame` class, the main class of this game. We will come back to `PongActivity` near the end of the chapter.

Coding the PongGame class

The first thing we will do is solve the problem of our `PongGame` class not being of type `View`. Update the class declaration as highlighted, like this:

```
class PongGame extends SurfaceView {
```

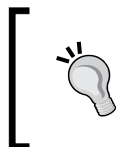
You will be prompted to import the `android.view.SurfaceView` class as shown in the next image:



Click **OK** to confirm.

`SurfaceView` is a descendant of `View` and now `PongGame` is, by inheritance, also a type of `View`. Look at the `import` statement that has been added. This relationship is made clear as highlighted next.

```
android.view.SurfaceView
```

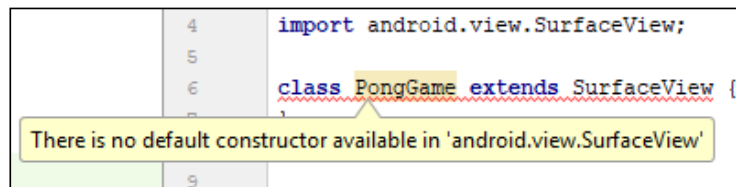


Remember that it is because of polymorphism that we can send descendants of `View` to `setContentView` method in the `PongActivity` class and it is because of inheritance that **`PongGame`** is a type of **`SurfaceView`**.

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 made this choice the right one for us.

We still have errors in both this class and `PongActivity`. Both are due to the lack of a suitable constructor method.

Here is an image showing the error in the `PongGame` class since we extended `SurfaceView`



The error in `PongActivity` is more obvious, we are calling a method that doesn't exist. However, the error shown in the previous image is less easily understood.

The `PongGame` class, now it is a `SurfaceView` must be supplied with a constructor because as mentioned (in a tip) in the previous chapter, once you have provided your own constructor the default (no 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.



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 developer's website. Here is a direct link to the `SurfaceView` page. <https://developer.android.com/reference/android/view/SurfaceView.html>. Look under the **Public constructors** heading and you will see that some constructors are optionally made available.

The `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 `PongActivity` that passes in `this` and the screen resolution and solve both problems at once.

Remember that `PongGame` 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 `PongGame`. 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
3. 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` 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 so that it does something, including adding code to the constructor, next.

Coding 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 (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, 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 like how we will time the animations of the bat and ball, how do we lock these timings without interfering with the smooth running of Android. These last 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.

Adding the member variables

Add the variables as shown below after the `PongGame` declaration but before the constructor. When prompted, click **OK** to import the necessary extra classes.

```
// 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 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 type `long` because they will be holding a huge number. Computers measure time in milliseconds since 1970. More on that when we talk about the game loop but for now we need to know that by 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 first `mFPS` will be reinitialized 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.

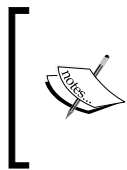
The next piece of the code we just added is shown again next for convenience.

```
// 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 variables `mScreenX` and `mScreenY` will hold the horizontal and vertical resolution of the screen. Remember that they are being passed in from `PongActivity` into the constructor.

The next two, `mFontSize` and `mMarginSize` will be initialized based on the screen size in pixels, to hold a value in pixels to make 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.



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 so 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 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` objects. `Paint` uses the default constructor as we have done previously but `mHolder` uses the `getHolder` method which is a method of the `SurfaceHolder` class. The `getHolder` method returns a reference which is initialized to `mHolder` so `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`.

```
// 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 as we have done before. We will see exactly what very soon.

Finally, we call `startNewGame`.

```
// 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 written it yet. Let's do that next. Also, notice the comment indicating where we will eventually get around to initialize the bat and the ball objects.

Coding the `startNewGame` method

Add the method's code immediately after the constructor's closing curly brace but before the `PongGame` class 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 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 `SurfaceView` 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() {

    // Draw the game objects and the HUD
    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 as described in the previous tip 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 `printDebuggingText`. The method doesn't exist yet. Let's add that now.

Add the 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 local variable `debugSize` to hold a value that is half that of the member variable `mFontSize`. This means that as `mFontSize` (which is used for the HUD) is initialized dynamically based on the screen resolution, `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 a guess at a good position vertically 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 `draw`, which in turn will be called from the main game loop, this line of text will be constantly refreshed up to sixty times per second.



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 focussed 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 `SurfaceView` 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 outwards for a change, we have a few familiar things like the calls to `drawColor`, `setTextSize`, and `drawText`. We can also see the comment which 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
- And `drawText` 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 which we want to manipulate to represent our frame of drawing is available, 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 (like moving the objects) will take place asynchronously with the code that detects the player input and listens for 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 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 does this code run asynchronously? That will be answered when we discuss threads shortly. For now, just know that the line of code checks if 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 `lockCanvas` 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 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);
```

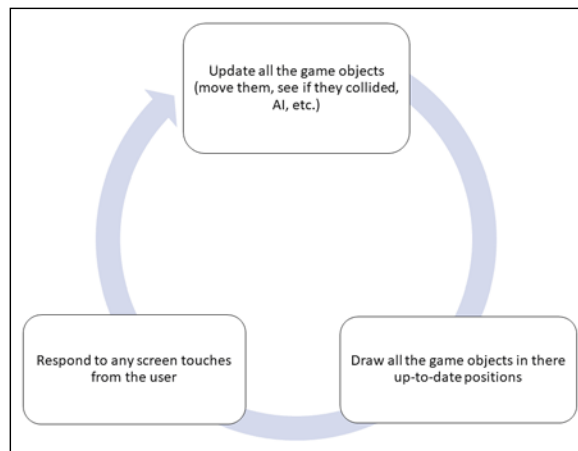
The `unlockCanvasAndPost` method sends our newly decorated `Canvas` object (`mCanvas`) for drawing to the screen and releases the lock so that other areas of code can use it again- albeit very briefly before the whole process starts again. This process happens every single frame of animation.

We now understand the code in the `draw` method, however, we still don't have the mechanism which 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, like turn-based games, still need to synchronize player input with drawing and AI while following the rules of the underlying operating system.

There is a constant need to update the objects in the game, perhaps by moving them, draw everything in its current position all the while responding to user input. A picture might help:



Our game loop comprises three main phases.

1. Update all game objects by moving them, detecting collisions and processing AI (artificial intelligence) if used
2. Based on the just-updated data, draw the frame of animation in its 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 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 to detect screen touches and listening to the operating system.

[ Just to be clear asynchronous means that it does not occur at the same time. Our game code will work by sharing execution time with Android and the user interface. The CPU will very quickly switch back and forth between our code and Android/user input.]

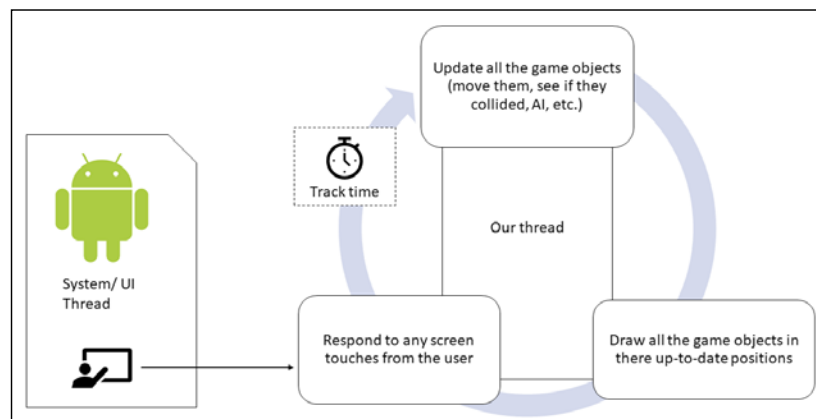
But how exactly will these three phases be looped through? How will we code this asynchronous system within which `update` and `draw` can be called and how will we make the loop run at the correct speed (frame rate)?

As we can probably guess, writing an efficient game loop is not as simple as a `while` loop.

[ Our game loop will, however, also contain a `while` loop.]

We need to consider timing, starting and stopping the loop as well as not causing the OS to become unresponsive because we are monopolizing the entire CPU within our single loop.

But 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 image. Notice the introduction to the concept of **threads**.



Now we know what we want to achieve; let's learn about threads.

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 are getting by, day to day. 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 which 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 (UI) 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 the three reasons simultaneously. It makes sense, it works, and we must because of 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 which is receiving the player's button taps for moving left and right at the same time as listening for messages from the OS like 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. Like 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 a variable `int x` 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 problem is the problem of **correctness** caused by multiple threads racing to completion oblivious—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 lowered. Avoiding an embarrassing splash.

The other problem with programs with multiple threads is the problem of **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 was 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 I tell you all this stuff about threads when you 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:

- 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.

Then we will have no sweat writing our Java code to create and work within our threads. 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.



What do you mean almost? What is happening is that the CPU switches between threads in turn/asynchronous. However, this happens so fast that we will not be able to perceive anything but simultaneity/synchronous. Of course in the story thread analogy, people do act entirely synchronously.

Let's take a glimpse of what our thread code will look like. Don't add any code to the project just yet. We can declare an object of type `Thread` 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 which 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`.

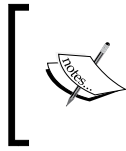


You can confirm this fact by looking at the `Thread` class on the Android developer's website here: [https://developer.android.com/reference/java/lang/Thread.html#Thread\(java.lang.Runnable\)](https://developer.android.com/reference/java/lang/Thread.html#Thread(java.lang.Runnable))

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. Like this code.

```
class someClass extends someotherClass implements Runnable{
```


We must then implement the abstract methods of the interface. Runnable has just one. It is the run method.



You can confirm this last fact by looking at the Runnable interface on the Android developer's 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 that we will write in our Pong game. First is `update` which is where all our calculations, artificial intelligence, and collision detection will go and then `draw` where perhaps unsurprisingly we will draw all our graphics. The code will look a bit like this. Don't add it 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 UI 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.

One more thing we need to know before adding the code.

Java try, catch exception handling

Before we move on to implement 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, where we need to use these `try` and `catch` blocks, include loading data from a file and stopping a thread.



Starting a thread does not need to use `try` and `catch` blocks.

It is the implementation of the classes that perform the fallible operations that force 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.
}
```



There is another block of code that can be used alongside `try` and `catch` called `finally` which can be used to put the code in 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, `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 which 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 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 discussed during the discussion on interfaces in the previous chapter and threads in the previous section. Add the empty `run` method, including `@Override` label as shown in a moment.

It doesn't matter where you add it if 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. Remember that the game needs to respond to the operating system starting and stopping the game.

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 and then we can consider further when and where from to call these methods. 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();
}
```

What is happening is slightly given away by the comments- you did read the comments? We now have a `pause` and `resume` method 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 `PongGame`.

Remember that `PongActivity` has the fully declared and initialized instance of `PongGame`?

Let's learn a little bit (more) about the Android Activity lifecycle.

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 override the `onCreate` method so that our code runs at this 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 to many other operating systems. For example, you can be using an app—say you're checking what people are doing on Facebook.

Then you get an email notification and you tap the email icon to read it. Mid-way 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 mid-way through the first daring fling you suddenly remember that Facebook post. So, you quit Angry Birds and tap the Facebook icon.

Then you resume Facebook and at the same place you left it. You could have resumed reading the email, decided to reply to the tweet or started an entirely new app.

All this backward and forwards 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 OS decides if and when to actually close down (destroy) an application and **our users data** (like 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 it can decide which app is in current use and so that it can then allocate the right amount of resources like memory and processing power. But also allow 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

The list of phases will hopefully appear fairly logical. As an example, the user presses the Facebook app icon and the app is **created**. Then it is **started**. All 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 greater 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 **resuming** 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 reason to mention it is so that:

1. You know it exists
2. We occasionally need to interact with it (and we do now)
3. 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 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 (like calling `setContentView`) and initializing major objects like `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 pausing. 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. 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 and override the `onResume` and `onPause` methods in `PongActivity` as shown 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 `onResume` is called after `onCreate` the first time an app is created not just after resuming from a pause. The code inside `onResume` and `onPause` 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 `collisionDetection` 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 condition `mPlaying` 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 local variable `frameStartTime` with whatever the current time is. The static method `currentTimeMillis` 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 `update` and `detectCollisions` is 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 zero. It is actually possible for the value to be zero 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 elapsed time by `MILLIS_IN_SECOND`. If you divide by zero, 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 (bat and ball) which we will code in the next chapter. They 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;
    }
}
```

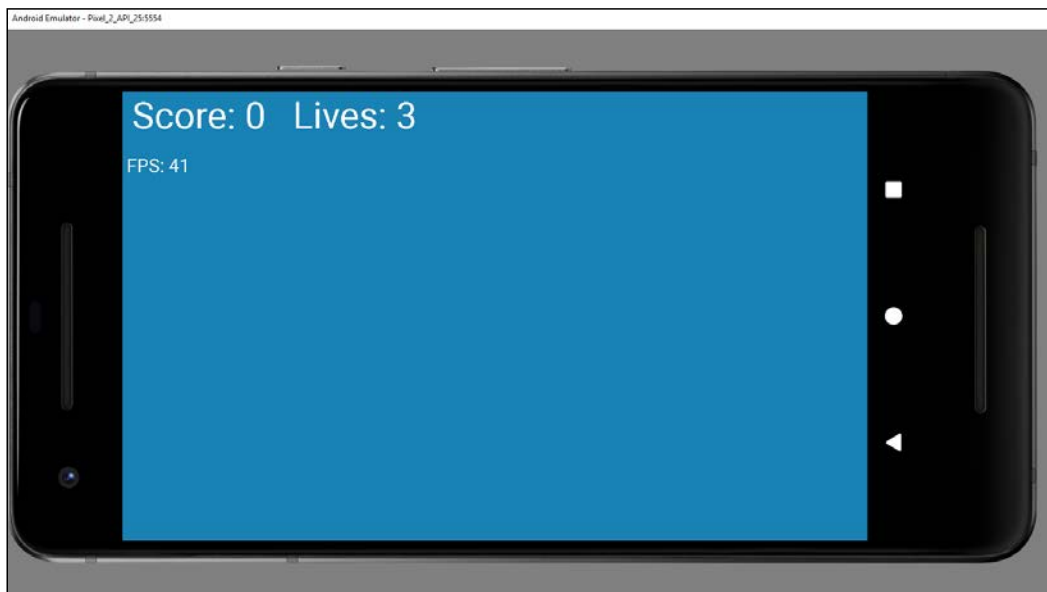
The result of the calculation that initializes `mFPS` each frame is that `mFPS` will hold a fraction of 1. Therefore, when we use this value inside each of the game objects we will be able to use the calculation:

`mSpeed / mFPS`

In order to determine the number of pixels to move on any given frame. As the frame rate fluctuates `mFPS` will hold a different value and supply the game objects with the appropriate number to calculate each move.

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.



Now we can make some much faster progress in the following chapters. 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, etc... It's an awfully long list of topics to cram into 35 pages. 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' `pause` and `resume` methods. It achieves this via the overridden `onPause` and `onResume` methods which are called by the OS.

Once the thread is running the code inside the `run` method executes alongside the UI thread that is listening for player input. As 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 `update` and draw in each call to `draw`.

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 very easy to achieve.

In this chapter we are going to:

- Code the `Ball` class
- Implement an instance of `Ball` in the game engine
- Code the `Bat` class
- Implement a `Bat` instance in the game engine
- Add movement controls to the `Bat`

Let's start by adding our first object to the game to bring a 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 which 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 take a sneak peek a few pages ahead (don't bother doing so) we will 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 nearly the same on devices with different resolutions.

As an example, a few pages ahead, we will initialize `mXVelocity` to the height of the screen divided by three. 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 3 seconds. This is regardless of whether the player has a super-high-resolution tablet or an old-fashioned 600 by 400-pixel phone.

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 be 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 one 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 (`PongGame` class) needs to know not only the size but also the horizontal and vertical size. 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.



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.

Exactly how we will make all this work is much better to show you than 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 what all the variables will be used for, add the highlighted member variables as well as the new `import` statement.

```
import android.graphics.RectF;

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
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.

```

// 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 is default access and therefore accessible via an instance of the `Ball` object used within the same package. In short, from `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`. The newly updated `mRect` object will then be available (via `getRect`) anytime `PongGame` 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 variable contains the value 60. If the screen was 1920 pixels wide, then `mXVelocity` will hold the value 640 (1920/3 see the reset method). We can then divide 640 by 60 giving the answer 10.6. In conclusion, 10.6 will be added on to the value stored in `mRect.left`. 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.top`, 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 amount of pixels per second. Even significant variations like a halving of the frame rate will be virtually undetectable but if the frame rate were to drop very low then this would result in choppy and unsatisfying gameplay.

Also note that we take no account in this method for the direction the ball is traveling (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 `PongGame` 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 on 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 in the horizontal when either the left or right sides are collided with.

Add the two new methods 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 `reverseYVelocity` 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 reverse `mXVelocity` 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 method does just that. Code the `reset` method 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 = (y / 3);
}
```

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 `(y / 3)` 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. Even a beginner's game tutorial should provide a little detail like this. 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 project 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:

- Detect the center of the bat and store it in the `batCenter` variable
- Detect the center of the ball and store it in `ballCenter` variable
- 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 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 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 `drawRect`. As `drawRect` has an overloaded version which takes a `RectF`, the job is done, and the ball is drawn to the screen. Note we also pass our `Paint` instance (`mPaint`) as we always do when drawing with the `Canvas` class.

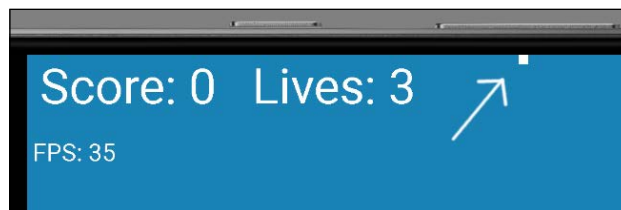
There is one more thing we need to do. We must call the ball's `reset` method to put it in an appropriate starting position relative to the screen resolution. Add this next highlighted line of code to the `startNewGame` method.

```
// 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);

    // Reset the score and the player's chances
    mScore = 0;
    mLives = 3;
}
```

Now the ball will be repositioned top and center every time we call `startNewGame` and this already happens once at the end of the `PongGame` constructor. Run the game to see the ball in action- kind of.



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 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 is 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.



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 `PongGame` 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` 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 the `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 `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 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`, `RIGHT`. They are `public`, so they can be directly accessed from `PongGame` (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/if 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` 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 ball is moving
        // Starting with STOPPED
        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.

```
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` (`mRect`) that represents the position of the bat. The four arguments passed in to initialize the left, top, right and bottom coordinates of the `RectF`, 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` with `PongGame`.

Coding the Bat helper methods

The `setMovementState` method is the setter that will receive a value from `PongGame` 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 which direction (if any) to move. Code the `update` method shown next and then we can 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 to the right.

The third `if` statement has the condition `mXCoord < 0`. This is detecting if 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 like this highlighted code.

```
// Initialize the bat and ball
mBall = new Ball(mScreenX);
mBat = new Bat(mScreenX, mScreenY);
```

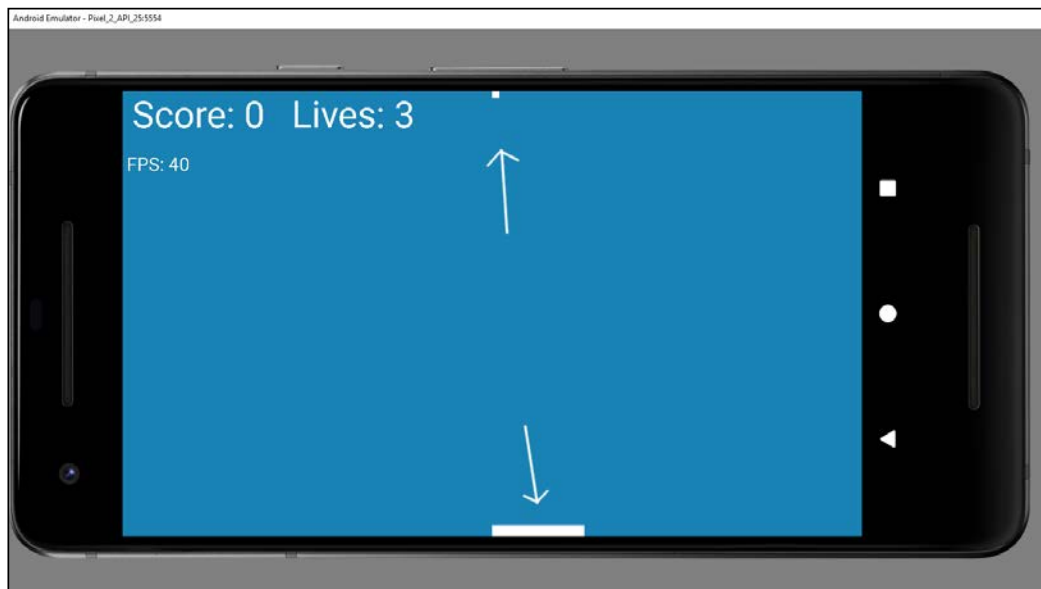
Update the bat each frame in the update method as shown highlighted next.

```
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.



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 `PongActivity` and `PongGame` and things might get into a bit of a muddle. Fortunately, the `onTouchEvent` method is also provided by `SurfaceView` which `PongGame` 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.

Add all the code at once and then we will dissect it and discuss how it works. Be sure to read the comments.

```
// 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;

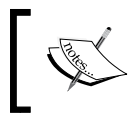
            // Where did the touch happen
            if(motionEvent.getX() > mScreenX / 2){
                // On the right hand side
                mBat.setMovementState(mBat.RIGHT);
            }
            else{
                // On the left hand side
                mBat.setMovementState(mBat.LEFT);
            }

            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;
}
```



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 the finger makes contact with the screen and `ACTION_UP` triggers when the finger leaves the screen.

In the first case, `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 are very simple, just one line. The `mBat` instance is used to call its `setMovementState` method to set its internal variable that will determine 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 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);
```

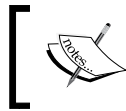
This line of code effectively cancels any movement.

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 it also detects 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.

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!



[No screen-shot provided because it doesn't look much different to the previous image when the bat was static.]

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 fourth project starting in chapter 18, we will have the game object classes draw themselves as well as handle their own part of responding to screen touches.

In the next chapter, we will finish this Pong game by detecting collisions, making some 1970's 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 in to practice near the end of the chapter. We will also learn about 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 on the game.

In summary, we will cover following topics:

- Study the different types of collision detection
- Learn how to handle different versions of Android
- Learn how to use the Android `SoundPool` class
- Finish the Pong game

Mind your head!

Handling collisions

As collision detection is something we will need to implement for all the remaining projects in this book I thought a broader discussion beyond that which will be required for Pong might be useful. Furthermore, I am going to use some images from the 5th 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.

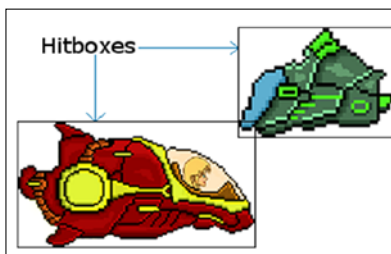
Collision detection options

First, here are a few of the different ways we can use mathematics to detect collisions, how we can utilize 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 if they intersect. If they do, we have a collision.



Where the hitboxes intersect we have a collision. As we can see from the previous image 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 a test for the intersection using the x and y coordinates of both objects.



Don't use the following code. It 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

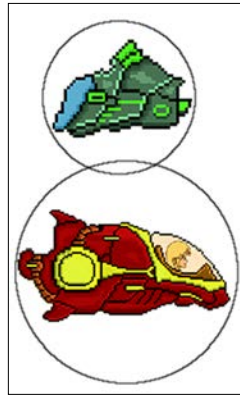
    if(ship.getHitbox().top < enemy.getHitbox().bottom
        && ship.getHitbox().bottom > enemy.getHitbox().top ){
        // Ship is intersecting enemy on y axis as well
        // Crash
    }
}
```

The above 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 code above, 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 axis. 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-axis in either order provided we check them both before declaring a collision.

Radius overlapping

This method is also checking to see if 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.



From the previous image, it is easy to see how the radius overlapping method is inaccurate for these objects and 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.

[ 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
}
```

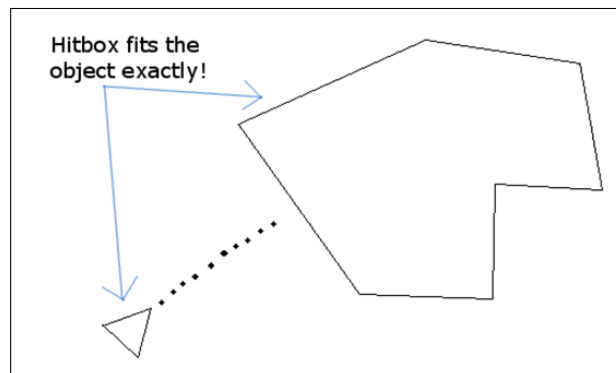
The code again makes some assumptions. Like we have a `getHitBox()` method that can return the radius as well as the central x and y coordinates.



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 which is 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` we can be certain that we must 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 if it crosses a line. The number of crosses is added and if the number is odd (for any single point) then a hit has occurred.





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.

Optimizations

As we have seen, the different collision detection methods can have at least two problems depending on which method you use in which situation. The problems are

1. Lack of accuracy
2. Drain on CPU power

The first two options (rectangle intersection and radius overlap) are more likely to suffer from inaccuracy and the second (crossing number algorithm) is more likely to cause a drain on the Android device's power and cause slowing down of the game.

Here is a solution to each of those problems.

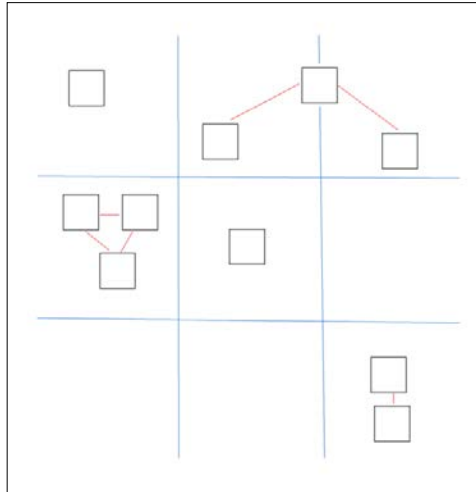
Multiple hitboxes

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.

Neighbour checking

This method allows us to only check objects that are in the approximate same area as each other. It can be achieved by checking which "neighborhood" of our game a given two 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 if objects share the same sector/ neighborhood.



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 when 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 the bat and the ball and we will also need to prevent the ball 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 doesn't only represent our objects (bat and ball) but also our hitboxes. It even has a neat `intersects()` method that basically does the same thing as rectangle intersection collision detection code we have hypothesized previously. So, let's think about how to add collision detection to our game.

We already use a `RectF` 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 really 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 being 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 95%) it has not seemed worthwhile. Unless you intend carving 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 the 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 versions `Build.VERSION_CODES` appropriate variable. If that explanation was a bit of a mouthful just look at how we determine if 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 a 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 FX; literally, a pool of sounds. The class handles everything from decompressing a sound file like a `.wav` or a `.ogg`, 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, declare an object of type `SoundPool` and an `int` identifier for each and every sound effect we intend to use. 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 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 which uses chaining and calls four separate methods on one object which initialize 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
        .setContentType
        (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, a user button clicks, collision, or similar.

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 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, and 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 get a warning about using deprecated methods. This is what the Android Developer's web site 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

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`.

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 `AssetFileDescriptor`.

The `AssetFileDescriptor` is initialized by using the `openFd` method of the `AssetManager` object which actually decompresses the sound file. We then initialize our 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 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 by 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 code below therefore simultaneously plays a sound and loads the value of the id which is being played into `nowPlaying`.

```
nowPlaying = sp.play(idFX1, volume, volume, 0, repeats, 1);
```



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 (1 is normal rate).

Just one more thing before we make the Pong game beep.

Stopping a sound

It is also very trivial 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

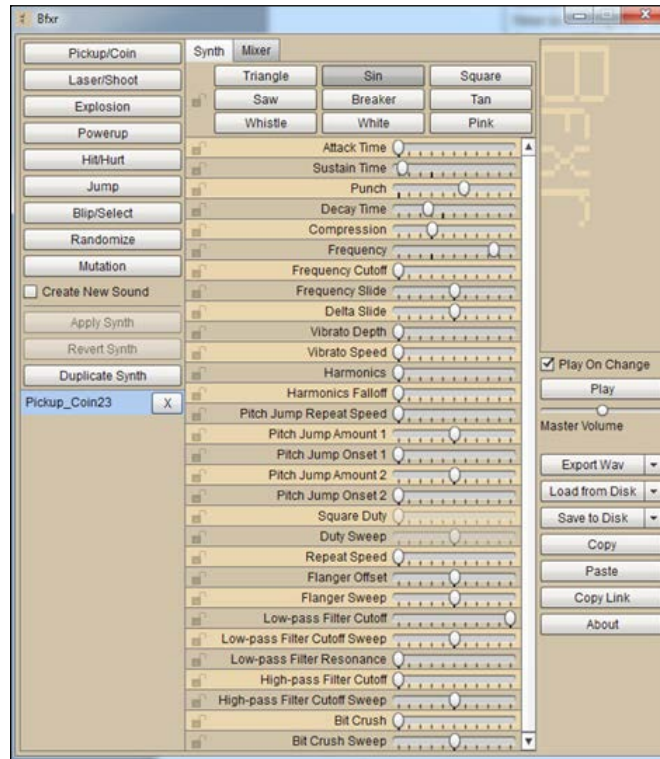
There is an open source app called **BFXR** that allows us to make our own sound effects. Here is a very fast guide to making your own sound effects using BFXR. Grab a free copy from www.bfxr.net.

 Note that the sound effects are supplied to you 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.:

 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.

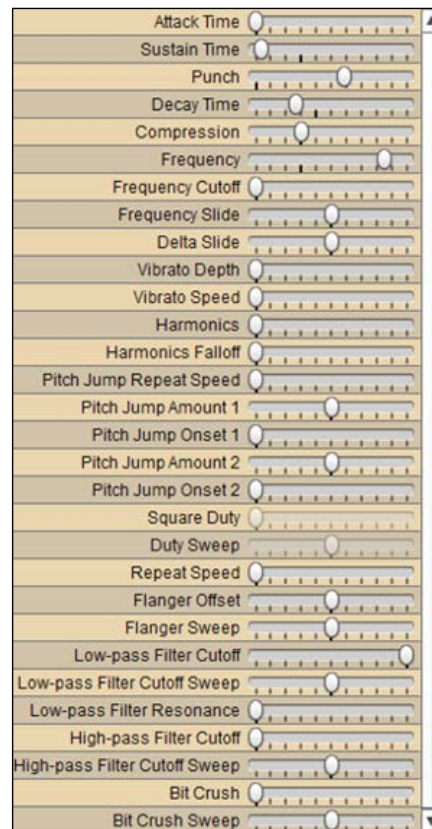
1. Try out a few of these things to make cool sound effects run `bfxr`.



2. Try out all the pre-set types which generate a random sound of that type. When you have a sound which is close to what you want, move to the next step.



3. Use the sliders to fine-tune the pitch, duration and other aspects of your new 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.



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.
6. Repeat steps 2 to 5 to create your cool sound FX. Name them beep.ogg, boop.ogg, bop.ogg, and miss.ogg. We use the .ogg file format as it is more compressed than formats like WAV.

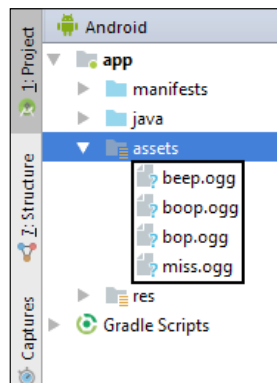
When you have your sound files ready we will continue with the Pong project.

Adding sound to the Pong game

Copy the `assets` folder and all its contents from the `Chapter 11` folder of the download bundle. 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 use the project explorer window in Android Studio to view the `assets` folder you can see that the sound effects have been added to the project.

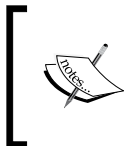


Let's write the code.

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;
```



You will need to import the `SoundPool` class using your preferred method or typing 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 as we did when we were exploring `SoundPool` in the previous section.

 If you are wondering about the names of the sound files/IDs, I could have called them `beep1`, `beep2` etc 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 code which should look quite familiar. Be sure to add the highlighted code before the call to `startNewGame`.

```
// 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`).



For this 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;
```

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 memory and associate them with the four `int` IDs we declared previously. They are now ready to be played.

Coding the collision detection and playing sounds

Now 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 and 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
}
```

 You will need to import the `RectF` class.
`import android.graphics.RectF;`

In the previous code this line returns true when the ball hits the bat and will cause the code inside to execute.

```
if(RectF.intersects(mBat.getRect(), mBall.getRect()))
```

Inside the `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 hit.

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 to the player's score. The final line plays the sound identified by the ID `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);
}
```

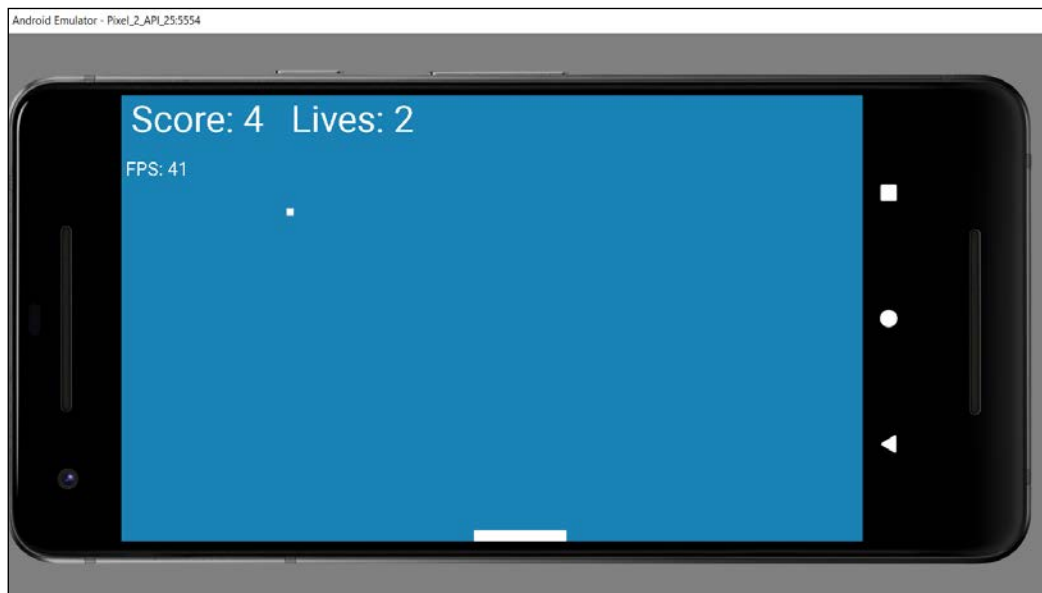
There are four separate `if` statements, one for each edge of the screen(wall). Inside of each the appropriate velocity (horizontal or vertical) is reversed and a sound effect is played.

Notice 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.



At this point, you have enough knowledge to revisit the Sub' Hunter game and add some sound effects.

Summary

We have done a lot of theory in this chapter. Everything from the mathematics of detecting collisions and learning how `RectF` has the `intersects` method which can handle rectangle intersections for us. We also looked closely at `SoundPool` including at how we can detect which version of Android the player has 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 that 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 setup 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 include:

- 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

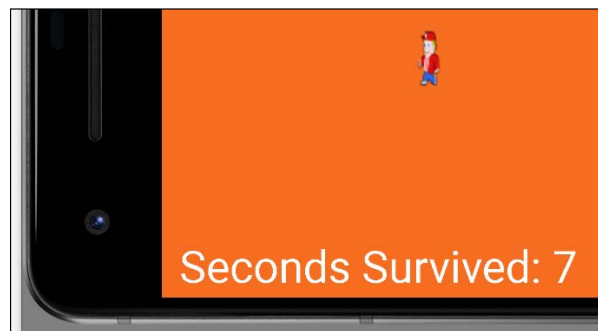
Let's plan what the final game will be like.

Planning the project

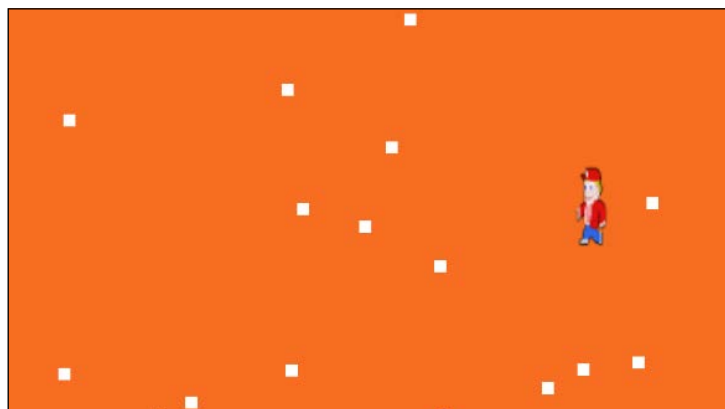
The Bullet Hell game is simple yet frantic to play. The game starts with Bob on the screen with a single bullet bouncing around the walls (edges of the screen).



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.



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.

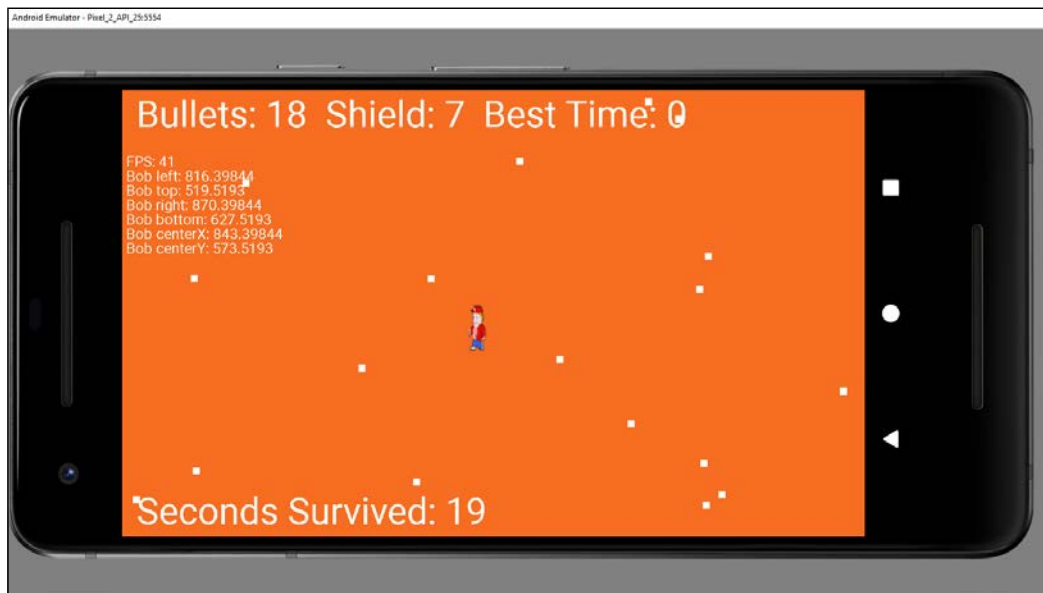


Clearly, this is a game which can only end badly for Bob. The game will also record the player's best performance (longest time) for the duration of the play session.



When we put the finishing touches on the fifth project in Chapter 21 we will see how to persist the high score even beyond the player switching his 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.



As well as learning the Java topics of 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 necessary for a game engine that is nearly identical to the Pong game and then we can move on to coding the Bullet class.

1. Create a new project called `Bullet Hell`.
2. Use the **Empty Activity** template and name the Activity `BulletHellActivity`. All other options the same as previous two game projects.

3. Uncheck the **Generate Layout File** option – we do not want a layout generated as we will be laying out every pixel ourselves.
4. Uncheck **Backwards Compatibility (AppCompat)** we won't need it for our games and they will still run on almost every phone and tablet.

As we have done before we will edit the Android manifest.

Change the AndroidManifest.xml to lock in landscape and full-screen

As a reminder, here is how to edit the `AndroidManifest.xml` file.

1. Make sure the `AndroidManifest.xml` file is open in the editor window.
2. In the `AndroidManifest.xml` file, locate the following line of code:
`android:name=".BulletHellActivity">`
3. Place the cursor before the closing `>` shown above. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line shown above.
4. Immediately below `BulletHellActivity` but before the newly positioned `>` type or copy and paste these two lines to make the game run full screen and lock it in the landscape orientation.

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
android:screenOrientation="landscape"
```



For more details refer to *Chapter 1, Java, Android and Game Development*, section *Locking the game to full-screen and landscape orientation*.

Creating the classes

Now we will generate 3 new classes ready for coding across this and the next chapter. You can create a new class by selecting **File | New | Java Class**. Create 3 empty, package private classes called `BulletHellGame`, `Bullet`, and `Bob`.

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

Paste and examine this code into the BulletHellActivity class.

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

        // 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 `setContentView` as well as in `onResume` and `onPause` 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 is exactly the same structure as the previous game. Familiarize yourself with it again by pasting and reviewing it a section at a time. Make note of 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{
}
```

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 exactly 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.

Coding the BulletHellGame methods

Add these four empty methods and the identically implemented `run` method 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 draw and onTouchEvent

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 clears the screen with a plain color and sets the paint color to white ready for drawing the bullets. Note the call to `printDebuggingText` when the `mDebugging` variable is set to `true`. We also added an empty (apart from return statement) method for `onTouchEvent` 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.

```

    public void pause() {
        mPlaying = false;
        try {
            mGameThread.join();
        } catch (InterruptedException e) {
            Log.e("Error:", "joining thread");
        }
    }

    public void resume() {
        mPlaying = true;
        mGameThread = new Thread(this);
        mGameThread.start();
    }

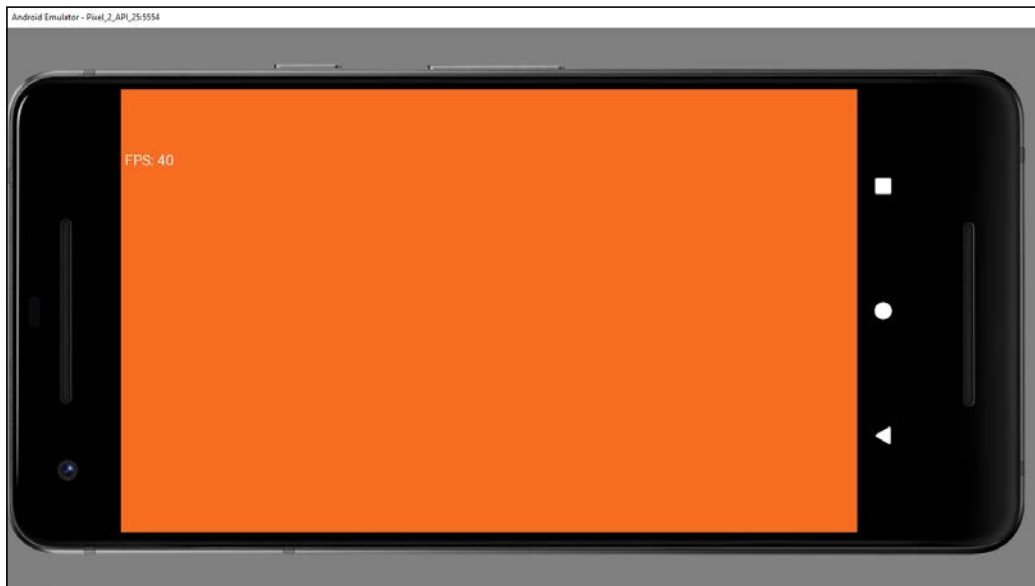
```

```
private void printDebuggingText() {  
    int debugSize = 35;  
    int debugStart = 150;  
    mPaint.setTextSize(debugSize);  
  
    mCanvas.drawText("FPS: " + mFPS ,  
        10, debugStart + debugSize, mPaint);  
  
}
```

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.



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.

```
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
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, each and every frame of the game loop. Add the `getRect` method shown next.

```
// Return a reference to the RectF
RectF getRect() {
    return mRect;
}
```

Next is the `update` method that 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 `mRect` 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 bullets 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' constructor. 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 *Chapter 13, Bitmap Graphics and Measuring Time*. In order to quickly see the bullets 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, *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`.

```
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.

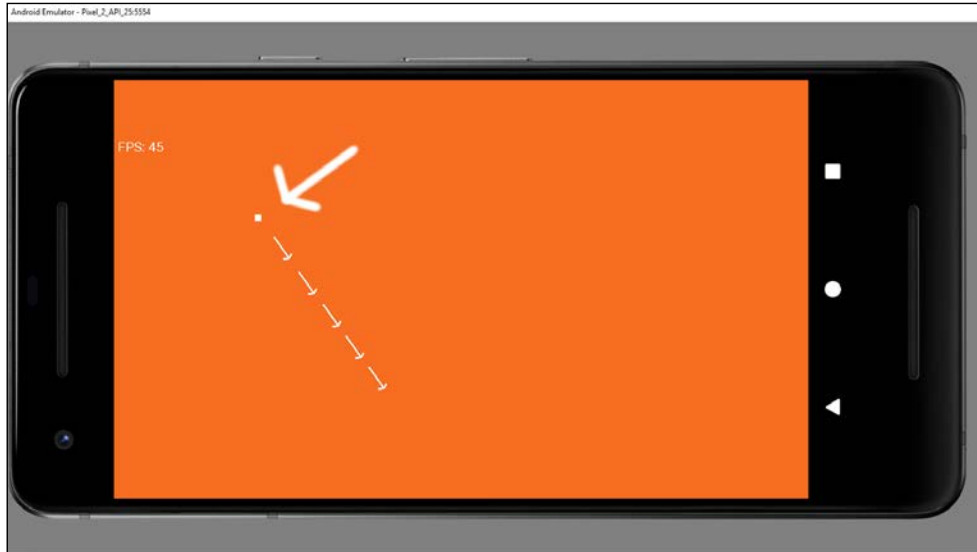
```
@Override
public boolean onTouchEvent(MotionEvent motionEvent) {

    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 vertical direction (1, 1).

You can now run the game and see the bullet whiz across the screen. Tap the screen and the same bullet will spawn again.



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 event is spawning and then respawning the bullet. For example, there is likely at least several `ACTION_DOWN` 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.

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 the game with a high score table with the top 100 scores?

We could 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:

```
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? Considering just the high scores scenario it is obvious we have a problem. 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` type variables- perhaps a high score table or a series of exam grades.

```
int [] intArray;
```

We could also declare arrays of other types, including classes like `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 labelled `intArray[0]`, `intArray[1]`, `intArray[2]` etc. With each space holding a single `int` value. Perhaps the slightly surprising thing here is that the storage spaces start at zero, not 1. Therefore, in a 100 *wide* array, the storage spaces would run from 0 to 99.

We could 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 replaces the name. And we can do anything with a variable in an array that we could do with a regular variable with a name.

```
intArray [3] = 123;
```

Here is another example:

```
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.



In *Chapter 14: 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 at 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 which we can use. For example:

```
int lengthOfSomeArray = someArray.length;
```

Above we assigned the length of `someArray` to the `int` variable called `lengthOfSomeArray`.

We can even declare an array of arrays. This is an array that, in each of its elements, lurks another array. This is shown as follows:

```
String[] [] countriesAndCities;
```

In the above array, we could 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 in the downloadable code bundle. 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 5 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 value 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.

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 in the download bundle. 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
// Because more than a few are lots of typing!

for(int i = 0; i < 1000; i++){
```

```
// Put the value 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 1000 `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 1000 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 of 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 in the download bundle. It is in the Chapter 12/Multidimensional Array Example/MainActivity.java.

Create a project with an empty Activity and call it Multidimensional Array Example. Leave the Activity name at the default it is not important.

After the call to `super.onCreate...` in `onCreate`, 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 5 arrays and each of these 5 inner arrays will be able to hold 2 Strings 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 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 value.

```
final int COUNTRY = 0;
final int CAPITAL = 1;
```

Now we set up a `for` loop to run 3 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 one time or a thousand times, the example would still work.

```
for(int i = 0; i < 3; i++){
```

Next, we actually determine which question to ask. Or more specifically, which element of our outer array. 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 also 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 multi-dimensional 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, 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. For example this:

```
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. The rule that 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 16 we will look at another Java topic closely related to arrays called Collections. This will take our data handling knowledge up yet another level.

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.



Make sure you deleted 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();
```

We have an array called `mBullets`, capable of holding 10000 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 starting at zero 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 zero 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 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 if 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 is true then the final 3 are not needed. This would make a significant difference with 10000 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 zero to `mNumbullets -1` and for each spawned bullet we use the `getRect` method to pass in to the `drawRect` method and draw 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 is the final version this and the next part we will amend in the next chapter which is the final chapter of this project. Add this 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.

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();

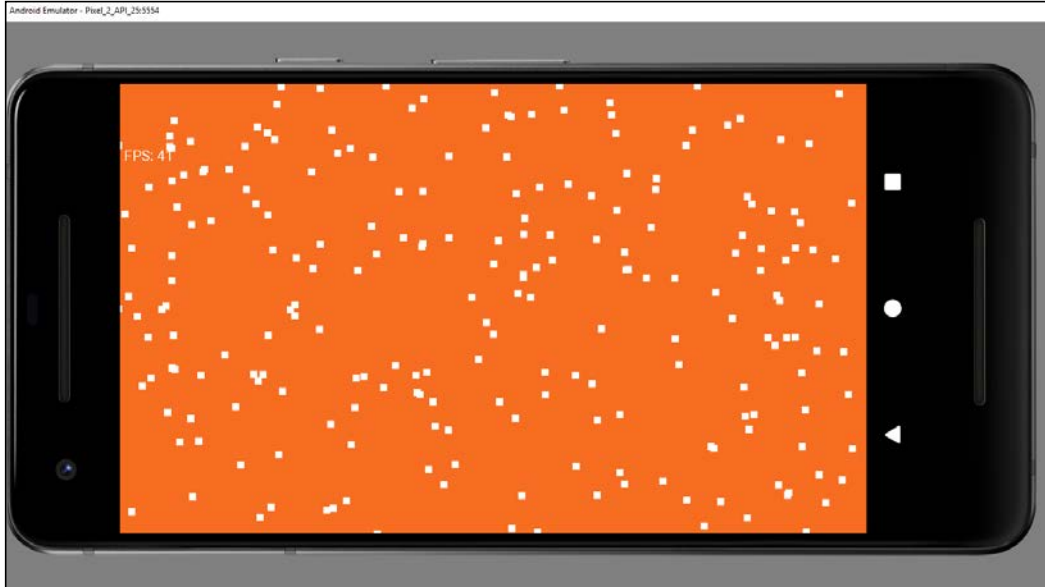
    return true;
}

```

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.



If you spawn the 10001st 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.

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 50000 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 as well as 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 section, 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 a HUD, measure the time, and code a solution to remember the longest (best) time.

In this chapter we will:

- Learn how to use bitmap graphics in Android
- Implement our teleporting superhero, Bob
- Finish all the other features of the game including the HUD and timing

Let's wrap this game up.

The Bob (player's) class

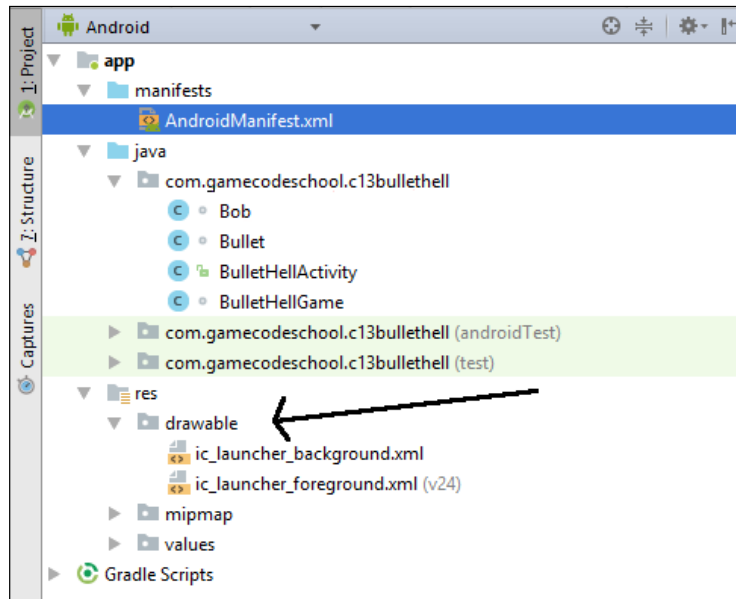
This is the final class for this project. As a reminder, we need to make Bob teleport every time the player touches the screen and he should teleport to the location of the touch. I predict that a `teleport` method is quite likely. As we code this class, we will also see how to use a `.png` file to represent Bob instead of just using boring rectangles like we have done so far.

The first thing we will do is add the graphics (`bob.png`) file to the project.

Adding the Bob graphic to the project

Right-click and select **Copy** to copy the `bob.png` graphics file from the download bundle 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



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 which takes a `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.

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 type `Bitmap`. You can probably guess what this is used for.

Add the member variables and import them into the `Bob` class that matches the discussion we have just had.

```
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 following code for the constructor 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 `BulletHellGame`. Also, you will see a third parameter, named `context`. This is used during the `Bitmap` initialization later in the method.

Next we initialize the `RectF` with this 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 and the second two parameters put the bottom right exactly the right distance away by adding `mBobWidth` and `mBobHeight` to the top and left calculation.

Finally, for the constructor, we initialize the bitmap with this line of code.

```
mBitmap = BitmapFactory.decodeResource
    (context.getResources(), R.drawable.bob);
```

The static `decodeResource` method of the `BitmapFactory` class 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. The `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 which are the location to move to. Later in this chapter we will code calling 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 variable `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, then 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 was 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 again will 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.

```

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 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 the `RectF` and the `Bitmap` in order that they can be used by `BulletHellGame` for things like 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 mStartGameTime;
private long mBestGameTime;
private long mTotalGameTime;
```

The first new member variable is our instance of `Bob` called `mBob`. Next, there is a `boolean` to keep track of whether `Bob` has been hit in the current frame.

The `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, for the new member variables, there are three more of type `long` that will be used to track and record time. We will see them 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 the collision detection

Add the following code into the `detectCollisions` method just before the final closing curly brace of the method. Be careful not to get this code mixed up among 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()
                - mStartGameTime;

            startGame();
        }
    }
}
```

The new code is basically one big `for` loop that loops from zero to `mNumBullets`.

```
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 the `RectF` from a bullet and from Bob and sees whether they have collided.

```
if (RectF.intersects(mBullets[i].getRect(), mBob.getRect())) {
    ...
}
```

If there was not a collision, then the `for` loop continues and the next bullet is tested for a collision. If there was a collision and the `if` block is executed, then 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 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 if the number of hits the player has taken is equal to the strength of the shield. If it is:

- The game is paused
- `mTotalGameTime` is initialized by subtracting `mStartGameTime` 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()
        - mStartGameTime;

    startGame();
}
```

Now we can draw Bob to the screen.

Drawing Bob to the screen

Add to the `draw` method as highlighted to draw Bob and the HUD

```
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 many times before and in the previous code we draw the number of bullets, the shield, best time, and current time to the screen.

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`. 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 of the download bundle. Use your operating system's file explorer to 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, as usual, the `onTouchEvent` method, as before we have two case statements in the switch block. The first case, `ACTION_DOWN`, checks if `mPaused` is currently true and if it is sets it to false as well as initializing `mStartTime` with the current time. `mStartTime` 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.

As 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 `teleport` 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` that were initialized back in the constructor 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, you will notice 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 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 too 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 too 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 again:

```
// Don't spawn too 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` 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 making it head 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 `spawnX` will then target the left and `velocityX` will head left.

The second `if-else` block uses the same techniques to make sure that the bullet is spawned at vertically opposite sides and heading vertically away from Bob.

We are nearly done.

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;

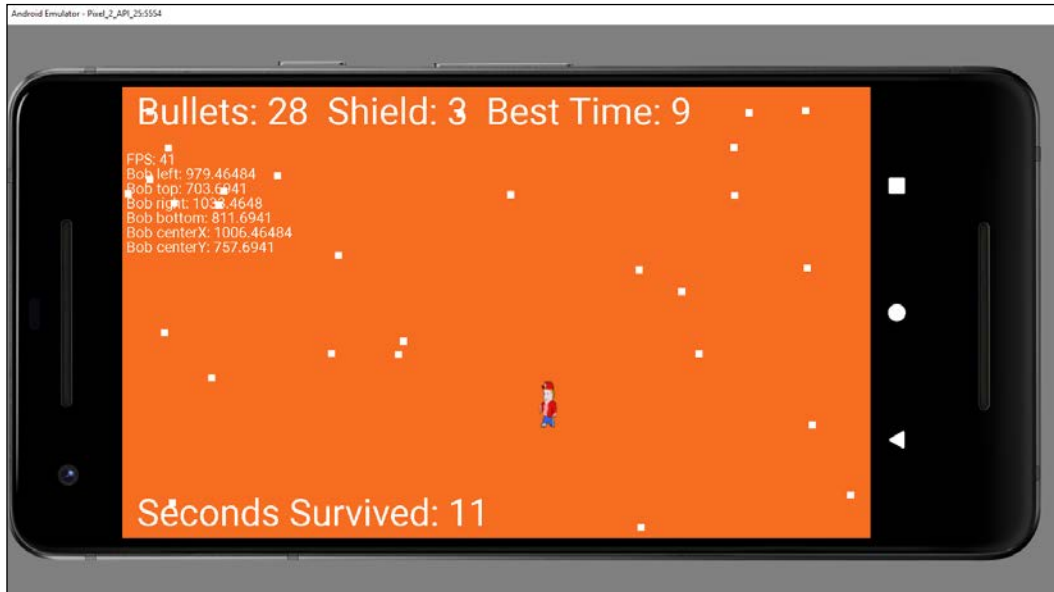
    // Did the player survive longer than previously
    if(mTotalGameTime > mBestGameTime){
        mBestGameTime = mTotalGameTime;
    }
}
```

The final line of code checks to see if the time survived this game is longer than the previous best and if it is 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 the game to see the completed project in action.



How long can you survive?

Summary

In the second and final chapter of the Bullet Hell project, we put the finishing touches on 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 the future projects bitmaps will play a much bigger role and we will do things like move them, scale them, rotate them and even animate a running Bob from them.

With our new-found skills and experience, we can now learn some more important Java and Android topics like:

- 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 will cover all of this over the next four chapters while we build a classic Snake game.

14

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 the 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?

- Managing memory with the Stack, the Heap, and the Garbage Collector
- 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



... known as a **reference type**. They quite simply refer to a place in memory where 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 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?

As we are thinking about the device's memory as a huge warehouse full of racks of labeled storage boxes, then you can think of the DVM 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 DVM will quickly move stuff around in a fraction of a second to avoid collisions. Also, when appropriate, Dalvik, 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...

In the same chapter, we 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.

Dalvik 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 a bit more.

You probably remember, right back to the first game project, we kept changing where we declared our variables? First, we declared some in `onCreate` and others just below the class declaration. When we moved them to just below the class declaration, we were making them member or instance variables.

And because we didn't specify the access, they were default access and visible to the whole class. And as everything took place in the one class we could access them everywhere. But why couldn't we do that when they were declared in `onCreate`? We learned that this phenomenon is known as scope and that variables, depending upon their access specifier (for members) or which method they are declared in (for local).

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 Virtual Machine 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 on the area of memory known as the **stack**. We can stick to our warehouse analogy when talking about the stack- almost. 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 includes objects (of classes) and arrays 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.

It 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 Dalvik to perform a task at a specific location as stored in the reference.



Reference variables are just that — a reference. A way to access and manipulate the object (variables, methods) but they are not the actual variables itself. An analogy 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 analogy, 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

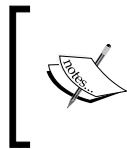
This whole stack and heap thing is forced upon us by computer architecture but Java acts as a middleman and basically helps us manage this memory.

As we have just learned, the Dalvik Virtual Machine keeps track of all our objects for us and stores them in a special area of our warehouse called the heap. Periodically the DVM 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, 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, your 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 the garbage collector has no knowledge of the time-critical parts of our game like the main game loop. It therefore also has the potential to make our game run poorly.

Now 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 DVM 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, it destroys (frees the memory for reuse). Or in Java terminology, it garbage collects.

If we are calling code like this

```
someVariable = new SomeClass()
```

And 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 allows it.

Stack and Heap quick summary

Let's take a quick look at what we learned about the Stack and the Heap.

- Our code doesn't explicitly delete objects. The DVM sends the garbage collector when it thinks it is appropriate. This is usually when there is no active reference to the object.
- Local variables and methods are on the Stack and the local variables are local to the specific method within which they were declared.
- Instance/class variables are on the Heap (with their objects) but the reference to the object, the address, is a local variable on the Stack.
- We control what goes onto the Stack. We can use the objects on the Heap but only by referencing them.
- The Heap is maintained by the garbage collector.
- An object is garbage collected when there is no longer a valid reference to it. So, when a reference variable either local or instance is removed from the Stack, then it's related object becomes viable for garbage collection. And when the DVM decides the time is right (usually very promptly), it will free up the RAM memory to avoid running out.

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 and gives you more understanding.

Now we can make the next game.

Introduction to the Snake game

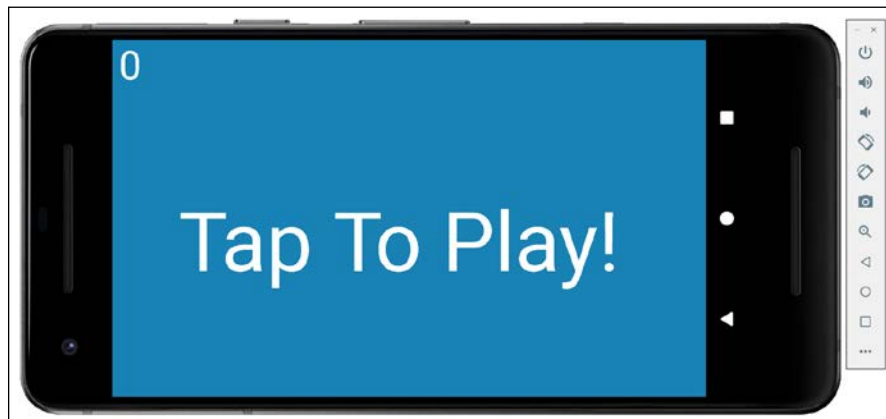
The history of the Snake game goes back to the 1970's. However, it was the 1980's when the game took on the look that we will be using. It was sold under numerous names and many platforms but probably gained widespread recognition when it was shipped as standard on Nokia mobile phones in the late 1990's.

The game involves controlling a single block or snakehead by turning only left or right by ninety degrees until you manage to eat an apple. When you get the apple, the Snake grows an extra block or body segment.

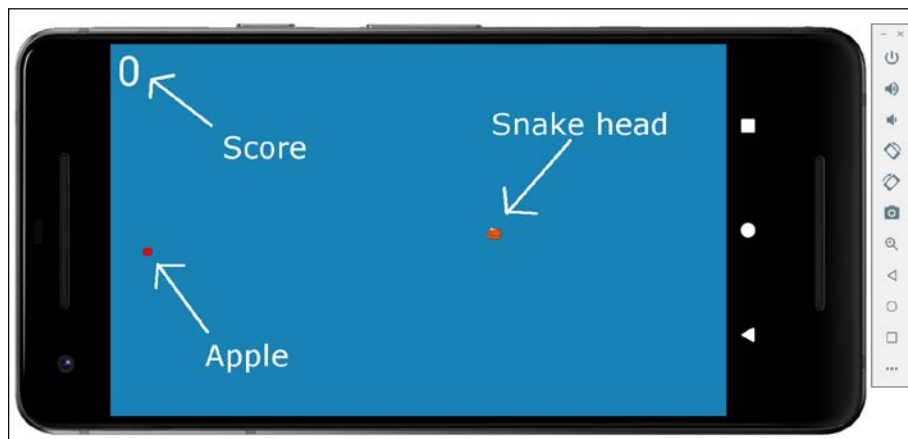
If, or rather when, the snake bumps into the edge of the screen or accidentally eats himself the game is over. The more apples the snake eats the higher the score.

[ 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.



When the player taps the screen, the game begins, and he must guide the tiny snake to get the first apple.



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 (`Snake` and the `Apple` classes) will draw themselves. We will achieve this by passing `Paint` and `Canvas` 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 confused 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 add to your own if you feel the need or if you get unexpected results.

The other new Java topic we will learn about is `ArrayLists`. These are part of the Java **Collections** and are part of a series of data storage and manipulation classes. `ArrayLists` have lots of similarities with arrays that we have already learned about but there are some advantages. We will also learn about the `HashMap` also from the Java Collections in the final project when we learn smarter ways to store our graphics.

Getting started with the Snake game

To get started, make a new project called `Snake` with the usual settings (empty Activity, no layout file, without backward compatibility) and call the Activity `SnakeActivity`.

Make full screen and landscape

As we have done in all the previous projects, let's make the game full screen and locked to landscape orientation.

As a reminder, here is how to edit the `AndroidManifest.xml` file:

1. Open the `AndroidManifest.xml` file in the editor window.
2. In the `AndroidManifest.xml` file, locate the following line of code:
`android:name=".SnakeActivity">`
3. Place the cursor before the closing `>` shown above. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line shown above.

4. Immediately below `SnakeActivity` but before the newly positioned `>` type or copy and paste these two lines to make the game run full screen and lock it in the landscape orientation.

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
android:screenOrientation="landscape"
```



For more details refer to *Chapter 1, Java, Android and Game Development*, section *Locking the game to full-screen and landscape orientation*.



Adding some empty classes

We will also make some empty classes ready for us to add code as we proceed through the project. This will mean there are fewer errors as we proceed.

As we have done in previous projects, you can create a new class by selecting **File | New | Java Class**. Create the three empty, package private 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 size straight into the `SnakeGame` constructor instead of dissecting it into separate horizontal and vertical `int` variables as we did in previous projects. Add all this code shown below.

Here is the entire `SnakeActivity` code:

```
import android.app.Activity;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;

public class SnakeActivity extends Activity {

    // Declare an instance of SnakeGame
    SnakeGame snakeGame;

    // Set the game up
    @Override
    protected void onCreate(Bundle savedInstanceState) {
```

```
        super.onCreate(savedInstanceState);

        // 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 SnakeEngine class
        mSnakeGame = new SnakeGame(this, size);

        // Make snakeEngine the view of the Activity
        setContentView(mSnakeGame);
    }

    // Start the thread in snakeEngine
    @Override
    protected void onResume() {
        super.onResume();
        mSnakeGame.resume();
    }

    // Stop the thread in snakeEngine
    @Override
    protected void onPause() {
        super.onPause();
        mSnakeGame.pause();
    }
}
```

The previous code should look very familiar.

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` and this `Point`.

The rest of the code for the `SnakeActivity` is identical in function to the previous project. Obviously, we are using a new variable name `mSnakeGame`.

Adding the sound effects

Grab the sound files for this project; they are in the Chapter 14 folder of the download bundle. 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.

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;
}
```

Let's run through those 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 to 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` and a couple of `int` variables for the related sound effects.

Following on we have a `final int` (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 member `mScore` is an `int` 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 they were 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.

Finally, we declare an instance of a `Snake` called `mSnake` and an `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, like 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 setup the `SoundPool` and load all the sound effects. Furthermore, we will initialize our `Paint` and `SurfaceHolder` methods 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 lines of new 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` called `blockSize` is declared and then initialized by dividing the width of the screen in pixels by `NUM_BLOCKS_WIDE`. The variable `blockSize` now represents the number of pixels that 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 zero 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.

Coding the run method

This method has some differences to 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 if `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 (that we code next) 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.

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 `updateRequired` method

The `updateRequired` method is what makes the actual `update` method execute only ten 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. `MILLIS_PER_SECOND` is initialized to 1000 because there are one thousand milliseconds in a second.

The `if` statement that follows is where the method gets its work done. It checks if `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 when `updateRequired` 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.

- Check if the Surface is valid
- Lock the Canvas
- Fill the screen with a color
- Do the drawing
- Unlock the Canvas and reveal our glorious drawings

In the "Do the drawing" phase mentioned in the list we scale the text size with `setTextSize` and then draw the score in the top left of the screen. Next, in this phase, we check whether the game is paused and if it is 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 onTouchEvent

Next on our to-do list is `onTouchEvent` 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 above 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 Activity will call this `resume` method and start the thread. When the player quits the game, the Activity will call `pause` that stops the thread.

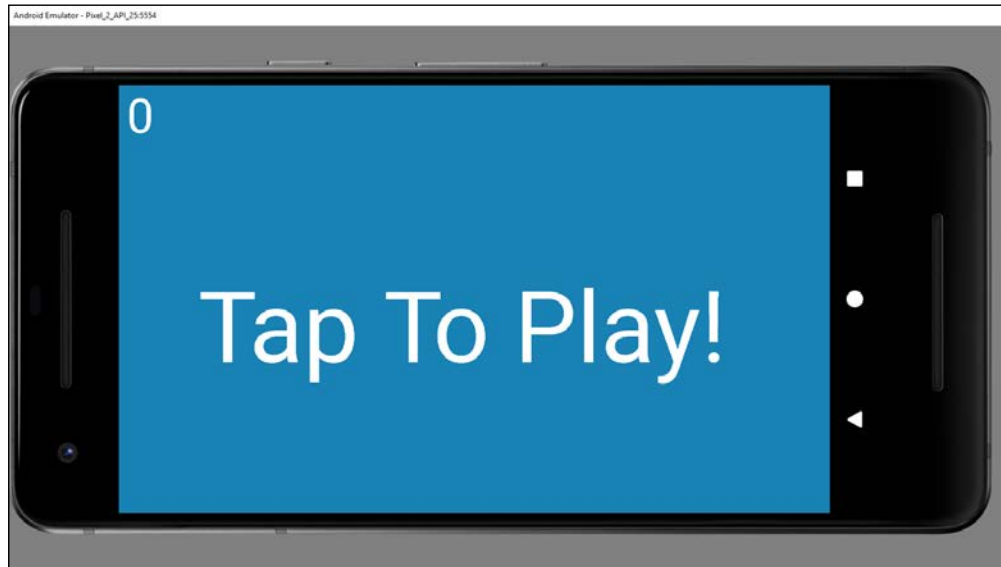
```
// Stop the thread
public void pause() {
    mPlaying = false;
    try {
        mThread.join();
    } catch (InterruptedException e) {
        // Error
    }
}

// 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 blank blue screen with the current score and the message **Tap To Play!**



Tap the screen, the text disappears, and the update method gets called ten 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 only accessible while in scope and that classes and their member variables are on the heap and 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 potentially problematic because it uses processor time that 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 call the `update` method only when one-tenth of a second has elapsed since the previous call to `update`.

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 of 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 hard-coding Strings in our code directly as we have done so far throughout the book.

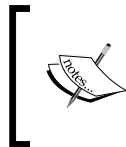
In this chapter, we will:

- Make Snake multilingual by adding the Spanish and German languages
- Learn how to use **String resources** instead of hardcoding text

Let's get started.

Making the snake game Spanish, English and 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.



While the source files for this project are provided in the Chapter 15 folder they are just for reference. You need to go through the processes described next to achieve multilingual functionality.

Adding Spanish support

Follow the next steps to add the Spanish language.

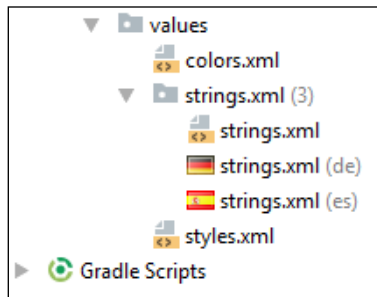
1. Right-click on the `res` folder then select **New | Android resource directory**. In the **Directory name** field type `values-es`.
2. Now we need to add a file in which we can place all our Spanish translations.
3. Right click on `res` then select **New | Android resource file** and type `strings.xml` in the **File name** field. Type `values-es` in **Directory name** field.

Adding German support

Follow these steps to add German language support.

1. Right-click on the `res` folder then select **New | Android resource directory**. In the **Directory name** field type `values-de`.
2. Now we need to add a file in which we can place all our German translations.
3. Right click on `res` then select **New | Android resource file** and type `strings.xml` in the **File name** field. Type `values-de` in **Directory name** field.

This is what `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 (apparently) to organize our files and folders as it is required by the Android operating system. You can, however, clearly see the Spanish and German files indicated by their flags.



Now we can add the translations to the files.

Add the string resources

To keep things simple our game will have just one short sentence to display in the appropriate language, "Tap To Play!" 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 Spanish flag. 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 German flag. 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 flag 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>
```

 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 both translations.



You can even localize to different versions of a language. For example, US or United Kingdom 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 like 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 some of 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 yourself. Once you start having any depth of translation needed, perhaps like 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.



My sincere apologies to any Spanish or German speakers who can likely see the limitations of the translations provided here.

Now we have the translations we can put them in to use.

Amending the code

Change the code in the `if(!mPaused)` block as highlighted next. I have commented out the line that we are replacing. The change will use the String resources instead of the hard-coded String "Tap To Play!".

```
// 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 chained methods `getResources.getString` to replace the previously hard-coded "Tap To Play!" text. Look closely and you will see that the argument sent to `getString` is the String identifier `R.string.tap_to_play`.

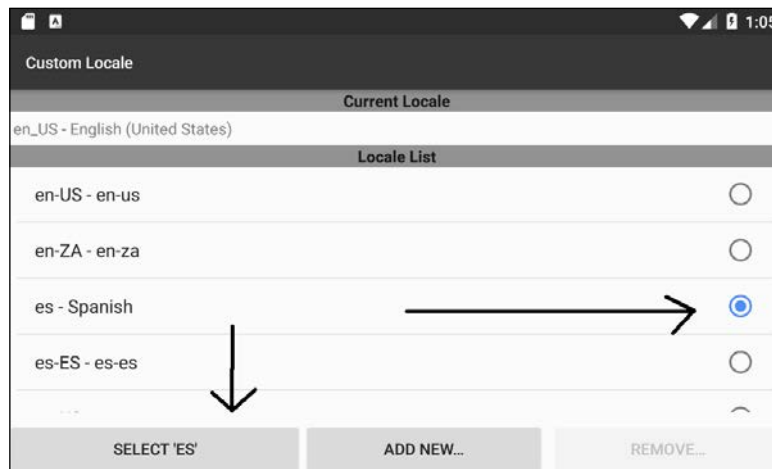
The code `R.string` refers to the String resources in the `res` folder and `tap_to_play` is our identifier. Android will then be able to decide which version (default, Spanish or German) is appropriate based upon the locale of the device on which the game is running.

Run the game in German or Spanish

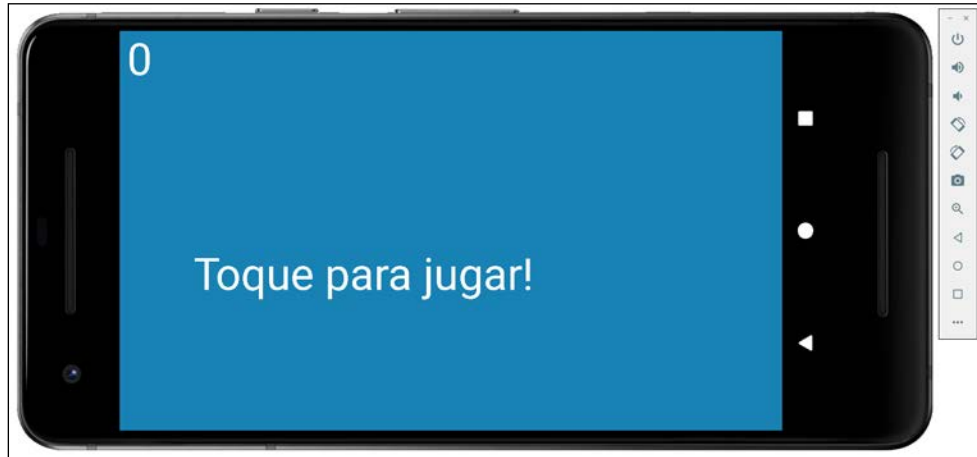
Run the app to see if it is working as normal. Now we can change the localization settings to see it in Spanish. Different devices vary slightly in how to do this but the Pixel 2 emulator can be changed by clicking on the **Custom Locale** app



Next select **es-ES** and then click the **SELECT 'ES'** button in the bottom left of the screen, as shown in the next image.



Now you can run the game again in the usual way



You can add as many string resources as you like. Note that using string resources is the recommended way to add all text to all types of app including games. The tutorials in the book (apart from this one) will tend to hard code them to make a more compact tutorial.

Summary

We can now go global with our games as well as add the more flexible String resources instead of hard coding all the text.

Let's carry on with the Snake game by coding the `Apple` class, then we can learn about `ArrayList` and enumerations in *Chapter 16, Collections, Generics, and Enumerations*. We will then be ready to code the Snake in *Chapter 17, Manipulating Bitmaps and Coding the Snake Class*.

16

Collections, Generics 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 `ArrayList` and enumerations (`enum` 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. In this chapter we will cover:

- Add the graphics to the project
- Code and use the `Apple` class
- Java Collections, Generics, and the `ArrayList` class
- The enhanced for loop

Let's go ahead with the project.

Adding the graphics

Grab the project's graphics from the download bundle; they are in the `Chapter 16/drawable` folder. Highlight the contents of this folder and copy them. Now right-click the `drawable` folder in the Android Studio solution explorer and select **Paste**. These files are the snake 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

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 the apple each time a new one is required.

There is also a simple `int` called `mSize` which we will initialize in a moment and it 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 (call `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`.



An interesting thing going on here that is 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 have been encapsulated. I just thought it would be interesting to do it this way and show 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 `sr`. The `s` parameter holds an actual `int` value- not a reference or any connection whatsoever to the data of the class which 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` ready for use. First, the `Bitmap` is loaded from the `apple.png` file and then it is neatly resized using `createScaledBitmap` to 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` is a simple getter method that returns a reference to `mLocation`. `SnakeEngine` will use this for collision detection. We will code the collision detection in the next (and concluding) chapter for this project.

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,
```

```

        location.x * mSize, location.y * mSize, paint);
    }

```

When this method is called from `SnakeEngine` 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) then you will better appreciate how encapsulating the responsibility to the classes to draw themselves makes `SnakeEngine` a much more manageable class.

Using the apple

The `Apple` class is done, and we can now put it to work in `SnakeEngine`.

Add the code to initialize the apple object in the constructor at the end as shown.

```

// 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 shown highlighted next.

```
// Draw the score
mCanvas.drawText("" + mScore, 20, 120, mPaint);

// Draw the apple and the snake
mApple.draw(mCanvas, mPaint);

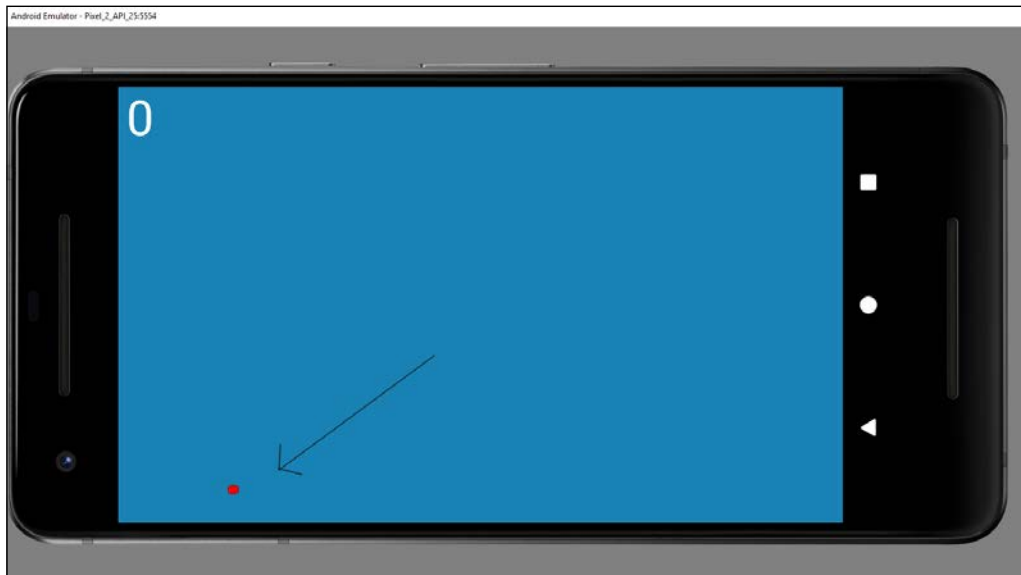
// Draw some text while paused
```

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 will be using 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 in SnakeGame. 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.



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 is 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.

ArrayLists

An `ArrayList` is like a regular Java array on steroids. It overcomes some of the shortfalls of arrays like having to predetermine its 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 clearer to use than a regular `for` loop. You can also use ordinary `for` loops with `ArrayList` 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.

`ArrayList` is part of the wider Java Collections which are a range of classes for handling data.

Let's look at some code that uses `ArrayList`.

```
// Declare a new ArrayList called
// myList to hold int variables
ArrayList<int> myList;

// Initialize the 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 like this code shows.

```
ArrayList<int> myList = new ArrayList<int>();
```

Nothing especially interesting so far, so 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>();

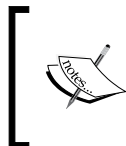
// Add a new String to myList in
// the next available location
myList.add("Donald Knuth");
// And another
myList.add("Rasmus Lerdorf");
// And another
myList.add("Richard Stallman");
// We can also choose 'where' to add an entry
myList.add(1, "James Gosling");

// Is there anything in our ArrayList?
if(myList.isEmpty()){
    // Nothing to see here
}else{
    // Do something with the data
}

// How many items 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 really 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 the `ArrayList` is empty (`myList.isEmpty()`), see how big it is (`myList.size()`) and get the current position of a given item (`myList.indexOf()`).



There are even more methods in the `ArrayList` class and you can read about them here: <http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html>. What we have seen so far is enough to complete this book, however.

With all this functionality, all we need now is a way to handle the data inside `ArrayLists` 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 of 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 of our 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.

```
int [] anArray = new String [];  
// We can initialize arrays quickly like this  
anArray {0, 1, 2, 3, 4, 5}  
  
for (String s : anArray){  
    Log.i("Contents = ", "" + s);  
}
```

There's another incoming news flash!

Arrays and ArrayLists are polymorphic

We already know that we can put objects into arrays and `ArrayList`. 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 that is 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, and easier to modify or 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
```

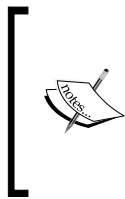
Also, we can write code for the superclass and rely on the fact that no matter how many times it is sub-classed, within certain parameters the code will still work. Let's continue our previous example.

```
// 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];
```

All we have just discussed is true for `ArrayLists` as well. Armed with this new toolkit of arrays, `ArrayLists` 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.



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 behaving objects as generic `GameObject` instances. This will make our code much cleaner (and shorter). For now, let's have a gentler introduction to `ArrayList` and finish the Snake game

Enumerations

An enumeration is a list of all the possible values in a logical collection. Java `enum` is a great way of, well, enumerating things. For example, if our game uses variables which can only be in a specific range of values and if those values could logically form a collection or a set, then enumerations are probably appropriate to use. They will make your code clearer and less error-prone.

To declare an `enum` 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.

```
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 do that now.

```
zombieTypes therresa = 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(therresa == 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 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 like this:

```
Heading heading = Heading.RIGHT;
```

We can change it when necessary with code like this:

```
heading = Heading.UP
```

We can even use the type as the condition of a `switch` statement (and we will) like this:

```
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 `ArrayList` 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 strongly hinted that 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 `ArrayList` and `enum` 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 `ArrayList` and `enum` 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 Bitmaps 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
- Add the sound effects to the project
- Code the `Snake` class
- Finish the game

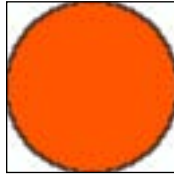
Let's start with the theory part.

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 next image of the snake's head.



And now look at one of the snake's body segments.



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 use 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 the 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 maths to the end of high school, you will probably understand the math without too much bother.

Unfortunately, understanding the math isn't enough. We would also need to devise efficient code as well as understand the Bitmap format. Fortunately, the Android API has done it all for us. Meet the `Matrix` class.

The Matrix class

The class is named as `Matrix` because it uses the mathematical concept and rules to perform calculations on a series of values known as matrices- the plural of matrix.



The Android `Matrix` class has little 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.



If you are intrigued by how the maths works and want an absolute beginner's guide to the mathematics 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();
```




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 one then the number doesn't change. However, when you multiply by minus one 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 like this line of code shown next.

```
mBitmapHeadLeft = Bitmap
    .createBitmap(mBitmapHeadRight,
        0, 0, ss, ss, matrix, true);
```

The previous code assumes that `mBitmapHeadLeft` is already initialized. The parameters to the `createBitmap` method are explained as follows:

- `mBitmapHeadRight` is a `Bitmap` object that has already been created, 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 in to.
- 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.



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 the single argument to `preRotate`. Look at this line of code.

```
// A matrix for rotating
matrix.preRotate(-90);
```

How simple was that? 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 rotate instead of the `preScale`.

```
mBitmapHeadUp = Bitmap
    .createBitmap(mBitmapHeadRight,
        0, 0, ss, ss, matrix, true);
```

This is what the `mBitmapHeadUp` will look like when drawn.



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.

Add 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 of the `Chapter 17` folder. 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.

Coding the Snake class

Add the single `import` statement and the member variables. Be sure to study the code it will give some insight and understanding to the rest of the `Snake` class.

```
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` object 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 the one variable is required.

The single `Point mMoveRange` will hold the furthest points horizontally and vertically that the snakehead can be at. Anything more than this will mean instant death. We don't need a similar variable for the lowest positions because that is simple, zero, zero.

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 games 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 way 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. For each direction that the snakehead 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 are heaps going on here so read all the comments and try to work it all out for yourself. 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 `SnakeEngine` 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 learnt about in the section *Rotating bitmaps*.

To do so, we first create a new instance of `Matrix` called `matrix`. We initialize `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 (head 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 the `mBitmapHeadUp` and `mBitmapHeadDown` ready to go.



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` with `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.

```
// 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 snakes heading 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 snakes head when the next game starts.

Coding the move method

The `move` method has two main sections. First, the body is moved and lastly the head. 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 which loops through all the body parts in the `ArrayList`.

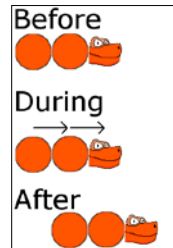
```
for (int i = segmentLocations.size() - 1;
     i > 0; i--) {
```

The reason we move the body parts first is that we need to move the entire snake starting at the back. This is why the second parameter of the `for` loop condition is `segmentLocations.size()` and the third `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 we 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.



The technique works regardless of which direction the snake is moving but it doesn't explain how the head itself is moved. The head is moved outside the `for` loop.

Outside the `for` loop, we create a new `Point` 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.

[ 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 to see if the snake has just died either by bumping into a wall or by attempting to eat himself.

```
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 of the screen to the left, right, top and bottom (in that order).

Next, we loop through the `segmentLocations` excluding the first position which has 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 himself and is now dead.

The last line of code returns the value of `dead` 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 to see if the snakehead 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 (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 snakehead. 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. `SnakeGame` 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 `SnakeGame` would become something of a nightmare.



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 for 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 `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 snakehead 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 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 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` so 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 checking 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 `checkDinner` 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` `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;
    }
}
```

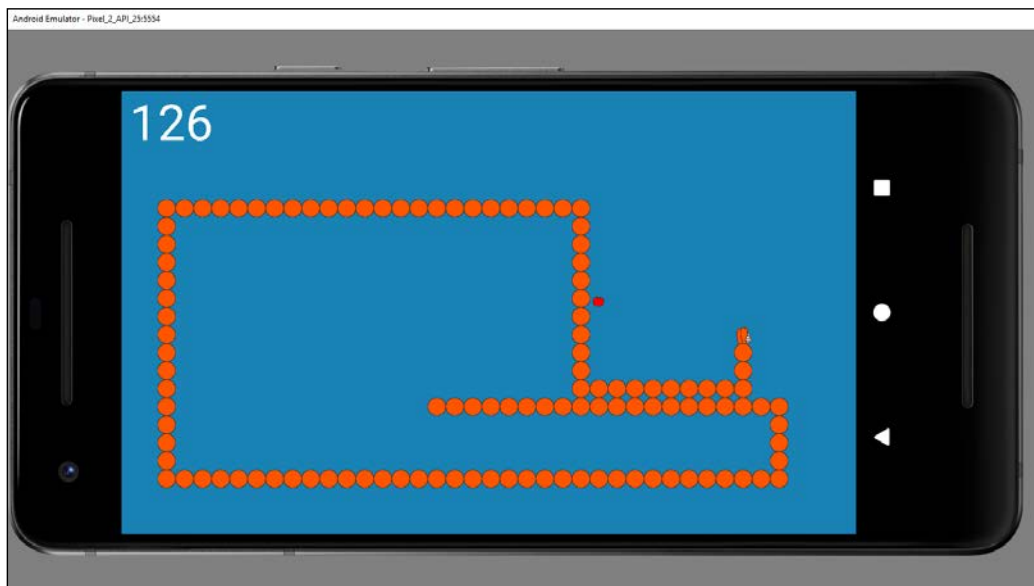
```
        default:
            break;

    }
    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.

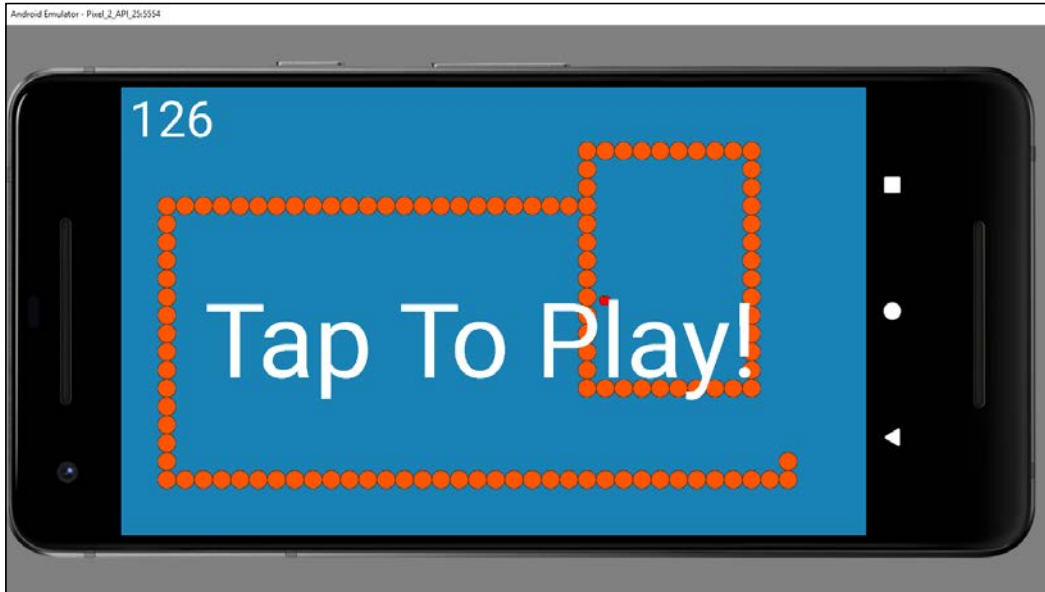
Running the completed game

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 next image.



Still hungry- must eat!

And when you die the game pauses to view the final catastrophe and a prompt (in your preferred language) to start again. This is shown in this next image.



That didn't taste so good!

Let's summarize where we are so far.

Summary

In this chapter, we got the chance to use `ArrayList` and `enum`. 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.

And don't worry, there will be plenty more game related lessons in the final two projects, including parallax scrolling backgrounds, particle explosions, multi-level explorable game worlds using a camera, persisting the high score, multiple, thinking enemy types and more.

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 like variables for speed and direction, a `RectF` for handling collisions and more besides.

As our objects have more in common we should start taking advantage of OOP, inheritance, polymorphism and a 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 disorganised 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 achieve 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



From this chapter on I will be helping the environment by not mentioning 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 the 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 go in to some more details about the features the game will have. Look at the starting screen for the Scrolling Shooter project in the next image:



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 just been launched by the player) you can see the message **Press Play**.

I have numbered some items of interest in the previous image, let's run through them:

- 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.
- The grey rectangles in the three remaining corners of the screen are the buttons which control the game. Number 2 in the image 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 next image).
- The two rectangles at the corner marked 3 are for shooting and flipping. The top button shoots a laser and the bottom button changes the horizontal direction that the ship is headed in.
- The buttons at the corner labeled 4 are for flying up and down to avoid enemy ships and lasers or to line up for taking a shot.



This next image 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 refocussing on the ship after a death is quite important and a particle effect distracts from this.



This next image (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, 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 image and then look at the brief explanation for each object:



- 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 his 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 is two in the image) labeled as number two 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 a Chaser, so he can outrun them but at some point, he will need to flip directions and shoot them 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 three is the player's ship. We already have 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 he 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 most notably 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 (like 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. Like reusing code, encapsulating code and designing classes that represent things. Patterns often amount to a best-practice way of encapsulating, allowing re-use and yet allowing a group of classes to interact.

[ As we will see throughout the rest of the book, design patterns are much more to do with the structure of your classes and 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 then go and investigate a particular pattern.

[ 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 which caused us to need the pattern in the first place.

Starting the project

Create a new project and call it `Scrolling Shooter`. Use the same settings as we always have: empty Activity, no layout file, no backward compatibility, for phones and tablets. When prompted, name the Activity `GameActivity`.

Editing the manifest

Make the game full-screen and landscape orientation as we have done before by editing the `AndroidManifest` file. Add the code just after `...:name=".GameActivity"` but before the trailing `>`. Here is the new code highlighted in context with the existing code:

```
<activity android:name=".GameActivity"
    android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
    android:screenOrientation="landscape"
>
```

The game is now locked to full screen and landscape orientation.

Code the GameActivity class

Here we will code the `Activity` 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 & paste. Add the code to the `GameActivity` class.

```
import android.app.Activity;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;

public class GameActivity extends Activity {

    GameEngine mGameEngine;

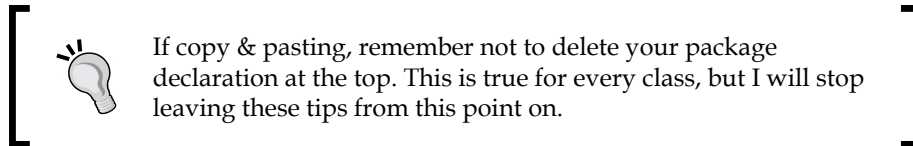
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        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();
}
}
```



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` by selecting **File | New Java Class** from the main menu. Type the name then select the **Package private** option and click **OK**. 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 `SurfaceView` and implementing `Runnable`.

```
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 `GameEngine` 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, you probably remember, is required because we implemented `Runnable`. 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).

When we have written some more code and classes, `mFPS` can then be passed to each of the objects in the game so 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 that 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 which controls when the `run` method executes. Notice as with all the methods so far 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.



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.

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!



The author's 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, etc.) of the game is to create a class (`GameState`) that holds all that state 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 class 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 respawning them again at the start of every game. The `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 `GameState` by simply declaring an instance of `GameState` but the reverse isn't true- `GameState` cannot trigger events in `GameEngine` except when `GameEngine` specifically asks for state 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` similarly 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 `GameState` 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 `GameEngine` 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.

Giving partial access to a class using an interface

The solution is an interface. While it is possible to pass a reference of the `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` had 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 `OnTouchListener`. 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 which 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 and 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 is to code a new interface.

1. Coding the new interface

Create a new interface by selecting **File | New Java Class**. In the **Name** section type **GameStarter**. In the **Type** section drop-down selector, choose **Interface**. Select **Package private** and then click the **OK** button to generate the empty interface.

Finally, for step 1, add the `deSpawnReSpawn` method without any body- as required by all interfaces. The new code is shown next highlighted amongst the code which 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.

2. 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.

3. 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* which 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.



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 as well.

Coding the `GameState` class

Create a new package private class called `GameState` in the same way we have done so often throughout this book.

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:

- If the thread is running
- If the game is paused

- If the game is finished (over)
- If 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 `GameState`.

Saving and loading the high score- forever

The first thing to point out about the `GameState` constructor which follows is the method 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 `GameStarter` as a parameter (and not `GameEngine`, `SurfaceView` or `Runnable` which would also have been syntactically allowable) we specifically get hold of the `GameStarter` functionality i.e. 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()` now has access to the exact same place in memory that contains the `deSpawnReSpawn` method in `GameEngine`.



You can think of `gameStarter.deSpawnReSpawn()` as a special button that when pressed offers remote access to the method we added to `GameEngine`. We will make quite a few of these special buttons with remote access over the final two projects.

Following on 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 type `SharedPreferences.Editor`? 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 which 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 zero (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 if `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 `mEditor` 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`.



Note that `SharedPreferences.Editor` also has a `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.

4. 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 section *What we will do to implement our interface solution*.

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 `GameEngine`.

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 changes 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 one from `mNumShips`. The `if` block checks if this latest catastrophe leaves `mNumShips` at 0 and if it does, pauses and then ends the game by calling `pause` and `endGame`.

There is another line of code in the `loseLife` 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 the `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.

```
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 `GameEngine` 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 whether `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());
    }
}
```

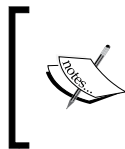
```
    }  
}  
  
public void startThread() {  
    // New code here soon  
    mGameState.startThread();  
  
    mThread = new Thread(this);  
    mThread.start();  
}
```

This new code calls the `stopEverything` method when `GameActivity` calls `stopThread`. And when `GameActivity` calls `startThread` 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`.

Building a sound engine

This is a bit different to how we have previously handled the sound but still should seem very familiar. We will be writing code that prepares a `SoundPool` 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.



We have already seen in the `GameState` class that the `loseLife` method receives an instance of `SoundEngine` and calls the `playPlayerExplode` method.

To prepare to code `SoundEngine` create a new class called `SoundEngine` in the usual way.

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 of the download bundle. 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

Add the now familiar members to get started.

```
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 now familiar code to prepare the 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();
    } 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.

```
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- the code itself is the same code we have used to play sound effects in previous projects. All it does is play the appropriate sound effect. 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 text.

```
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 if there are any problems before you proceed. Just for fun, you could test the `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 control buttons and text

The HUD, as usual, will display all the on-screen 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 like the score and player's lives it also has the control buttons.

The control buttons are basically slightly transparent rectangles drawn on the screen. It will be important that when we detect the player's touches we determine accurately if a touch happened within one of these rectangles as well as 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 the `Apple` classes in the previous project, the `HUD` class will also be responsible for drawing itself when requested. Add a new package private class called `HUD` and add 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 type `int`. They will be used to remember useful values like 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 previous code we declare five `static int` variables called `UP`, `DOWN`, `FLIP`, `SHOOT`, `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 which followed the member declarations initialize 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 declared 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 noted 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 are for 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 which initializes them is that they each take four parameters and that those parameters are for left, top, right and bottom positions, in that order.

The formulas used to calculate the values of each corner of each `Rect` all use the `mScreenWidth`, `mScreenHeight` and the three new variables we have just discussed as well. For example, the `up` `Rect` which needs to be in the bottom left corner above the `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 formula above 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 formula above in the third argument position means the right-hand side of the button will end 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 fourth and final parameter above 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 the `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` 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` and then adds each of the `Rect` objects using the `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 earlier project. In addition, the `draw` method receives a reference to `GameState` 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.rgb(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 fairly simply:

- The color to draw with is chosen with `setColor`
- The size of the text is set with `setTextSize` and `mFormatting` is used as the size
- Three lines of text are drawn using `drawText` to display the high score, score and the number of lives the player has. Notice how `mTextFormatting` 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 number of player 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 is 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 is 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 to the `draw` method but it would have made it more unwieldy. Notice the call to `drawControls` also passes a reference to `Canvas` and `Paint` as it will need it to draw the `Rect` objects that are the 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 `setColor`. Look at the first argument sent to the `Color.argb` method as it is different to every time 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.



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` 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, Generics and Enumerations*.

Outside the `for` loop of the `drawControls` method, there is one more line of code which sets the color back to full transparency. 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 the `controls ArrayList`. The `controls ArrayList` will also be useful when we are calculating the players 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 class that it will trigger to draw themselves at the appropriate time. As a result, we will regularly add code to this class including adding extra parameters to 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 package private class called `Renderer` and add 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` will hold the instances of `Canvas`, `SurfaceHolder` and `Paint` and therefore be responsible for passing the references of `Canvas` and `Paint` to the `HUD` class's `draw` method as well as the `draw` methods of all the other classes (when we have coded them).

The constructor that follows, initializes the `SurfaceHolder` and the `Paint` objects in the usual way using the `SurfaceView` reference that gets passed in from `GameEngine` (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` and a `HUD` 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 GameEngine 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:



As you can see, for the first time, we have actual button positions marked out on the screen. Up and down in the bottom left corner, 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 him how to start the game.

Summary

The was quite a chunky chapter but we have learned a huge amount. We have learned how we will 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 limited 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 of hundreds or even thousands of graphical objects that are used to create an effect. Our particle system will look like an explosion.

Here is a summary of 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 about the Observer pattern.

The Observer pattern

What we need is a way for the `GameEngine` class to send touch data to the `UIController` class that we will code later in this chapter and later (in the next chapter) the `PlayerController` class. We need to separate responsibility for different parts of touch handling because we want the `UIController` and the `PlayerController` to be responsible for handling the aspects of control related to them. It makes sense, the `UIController` knows all about the UI and how to respond and the `PlayerController` knows all about controlling the player. Making the `GameEngine` 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 don't want to do this anymore. 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.

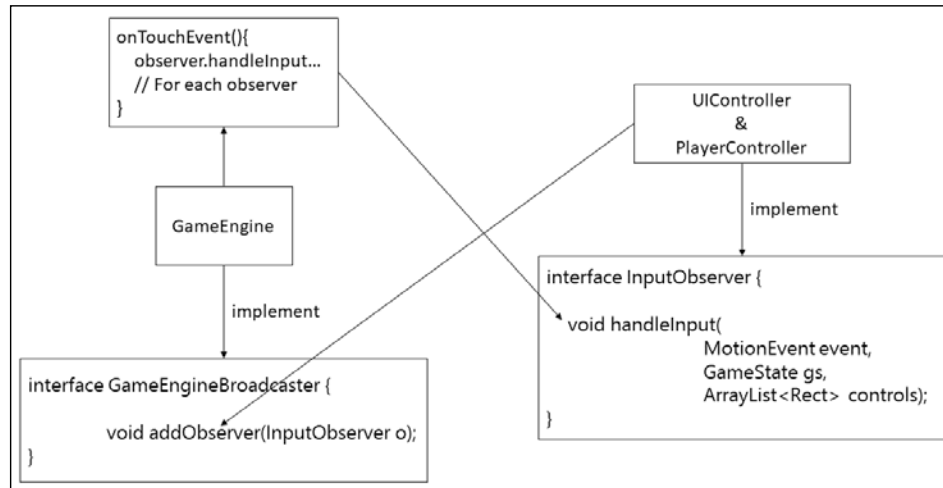
What we need is a mechanism for objects who decide for themselves while the game is running that they want to receive touch data and then let the `GameEngine` class know how to contact them each time some new data is received.

As there will be more than one recipient of data and just one sender of data this is a broadcaster-observer relationship. To be clear, the `GameEngine` will broadcast the touch data when it gets it and the `UIController` and `PlayerController` will receive it.

The Observer pattern in the Scrolling Shooter project

The `GameEngine` class will need a method that the observer classes can call to register/subscribe for updates. The observers will receive a reference to the `GameEngine` and the observers will call this special method. The `GameEngine` reference will be in the form of an appropriately coded interface (called `GameEngineBroadcaster`) to expose only the method we want. And to be sure the broadcaster can reach its observers/subscribers they will implement another interface called `InputObserver`.

Look at the below image that demonstrates this relationship:



As you can see from the image the GameEngine class implements the GameEngineBroadcaster interface and therefore has a addObserver method. The GameController and the PlayerController implement the InputObserver interface and therefore each has a handleInput method. GameController and PlayerController must each call the addObserver method once before the game starts and pass a reference to themselves in the form of an InputObserver. GameEngine will then call the handleInput method on GameController and PlayerController every time the onTouchEvent method is called. Note that it would be trivial to add more classes that needed to handle input. They just need to implement the InputObserver interface and call the addObserver method.



In a full implementation we would also add the ability to unsubscribe (another method) but that won't be necessary for this project.

We haven't discussed how GameEngine will store the InputObserver references or where the InputObservers call addObserver from. Now we have seen the theory we will implement the Observer pattern for real and that should help clarify things further.

Coding the Observer pattern in Scrolling Shooter

Now 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 action 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 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.



[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 selecting **File | New Java Class**. In the **Name** section type `GameEngineBroadcaster`. In the **Type** section drop-down selector choose **Interface**. Select **Package private** and then click the **OK** button to generate the empty interface.

Here is the entire code for the `GameEngineBroadcaster` interface with its single empty method called `addObserver` that takes an `InputObserver` as a parameter. The code to type is shown next highlighted amongst the code which was auto-generated.

```
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 by selecting **File | New Java Class**. In the **Name** section type **InputObserver**. In the **Type** section drop-down selector choose **Interface**. Select **Package private** and then click the **OK** button to generate the empty interface.

Here is the entire code for the `InputObserver` interface with its single empty method called `handleInput` that takes a `MotionEvent`, `GameState` and an `ArrayList` containing the position of the controls as parameters. The code to type is shown next highlighted amongst the code which was auto-generated.

```
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 the `GameEngineBroadcaster` to the list of interfaces that the `GameEngine` class implements.

```
class GameEngine extends SurfaceView implements Runnable,
    GameStarter, GameEngineBroadcaster {
```

On screen you will observe 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 `InputObservers`. An `ArrayList` will do the job.

Declare and initialize a new empty `ArrayList` that holds objects of type `InputObserver` as highlighted next as a member of the `GameEngine` class.

```
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 make our first `InputObserver` which can register for broadcasts, we will add the code that will call all the `InputObservers`' `handleInput` methods. Add this highlighted code in 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());
    }

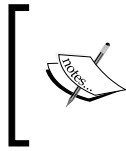
    return true;
}
```

The code loops through all the `InputObservers` in the `ArrayList` and calls their `handleInput` method (which they are guaranteed to have implemented). If there is zero, just one or 1000 `Observers` in the `ArrayList` 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 package private class and call it `UIController`.



As already stated, later in the project we will also have another `InputObserver` 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, the `implements InputObserver` code to the class declaration and 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`.

Coding the required `handleInput` method

Add the `handleInput` method that will be called from the `ArrayList` of `InputObservers` (in `GameEngine`) each time the player touches 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 notice 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 `onTouchEvent` receives a new `MotionEvent`.

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 something subtly but significantly different going on here. We are calling the `getActionIndex` method on our `MotionEvent` object and storing the result in an `int` 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, `MotionEvent` class holds data about multiple touches and more advanced data on things like movements on the screen and more besides. Locked away inside `MotionEvent` is 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 a whole array of events of the event that performed an action.

Therefore by calling `getActionIndex` 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 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 `handleInput`.

```
int eventType = event.getAction() & MotionEvent.ACTION_MASK;
```

This code gets an `int` 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 next.

```
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.

[



]

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 if 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.

```
// 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();
}
```

If the game is not currently paused, then it is set to paused with `gameState.pause()`. If the game was currently over, then a new game is started with `gameState.startNewGame()`. The final `else if` checks if the game is currently paused but the game is not over. In this case the game is un-paused.

Using the UIController

We are nearly there with the `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;
```

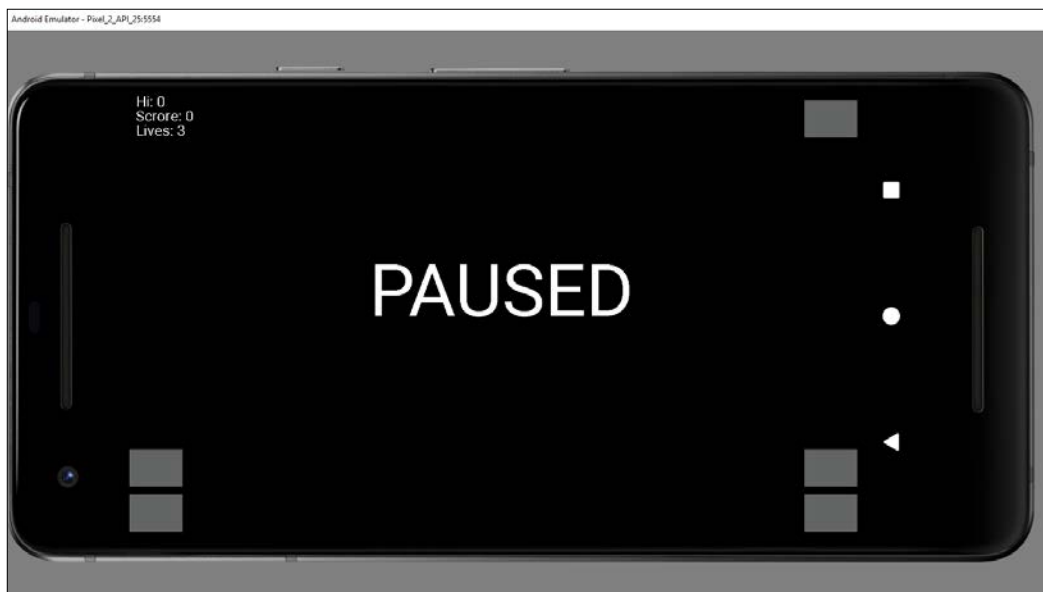
Now you can initialize it in the `GameEngine` constructor like this highlighted code.

```
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);  
}
```

The code simply calls a constructor like any other but also passes in `this` which is a `GameEngineBroadcaster` which the `UIController` class uses to call the `addObserver` method.

Running the game

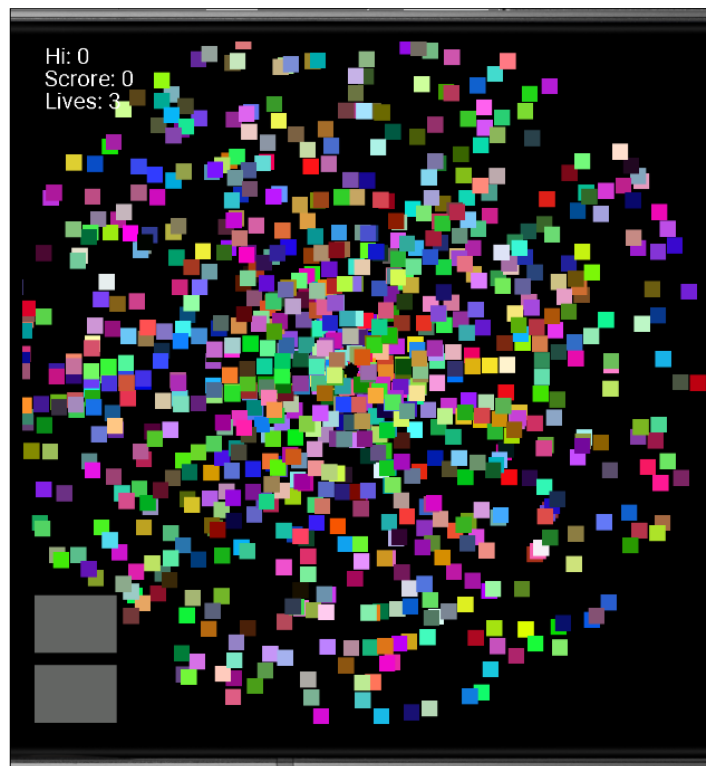
Run the game and you can tap the start/pause/resume button to start the game.



When the game is running you can also use the start/pause/resume button to flip between the paused and running states. Obviously, nothing happens yet when it's running but we will change that 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 look by the end of this chapter.



The particle system in the finished game is plain white with smaller particles. By the end of this chapter it will be plain how to achieve both types of particle and you can 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 the two member variables and the constructor 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 which will be held by the `PointF` called `mVelocity` and it will also have a current position on the screen held by `PointF mPosition`.



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`. The `PointF mVelocity` is not a reference to the `PointF` 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 then 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 each frame of the game by the `ParticleSystem`'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`.

 Notice 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 the exactly correct speed. But all the speeds are going to be random anyway. There is not much to gain from adding this extra calculation (for every particle). As we will soon see however, the `ParticleSystem` class will need to take account of the current frames per second to measure how long it should run for.

Next, we coded the `setPosition` method. Notice the method receives a `PointF` 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 has a few more details than 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 the following import statements and member variables to a new class called `ParticleSystem`:

```
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` that 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 because we need to generate so many random values that creating a new object each time would be sure to slow us down a bit.

Finally, the boolean `mIsRunning` will track whether or not 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 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 zero and 359 is generated and stored in the `float` `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 which we 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 which can be used inside the `update` method of the `Particle` class to update its position each frame.



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 zero 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/>

The single line of code which uses `Math.sin` and `Math.cos` to create a vector I have decided not to explain in full because the magic occurs partly in the formulas:

Cosine of an angle * speed

And 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` is created and then added to the `mParticles` `ArrayList`.

Next, we will code the `update` method. Notice that the `update` method does need the current frame rate as a parameter. Code the `update` method.

```
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 the 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 the `mParticles` `ArrayList`.

Finally the code checks to see if 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 while the `emitParticles` method will be called each time the effect needs to be started.

```
void emitParticles(PointF startPosition) {  
  
    mIsRunning = true;  
    mDuration = 1f;  
  
    for(Particle p : mParticles) {  
        p.setPosition(startPosition);  
    }  
}
```

First notice a `PointF` for 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` will be the coordinates of the alien ship which was destroyed.

The `mIsRunning` boolean is set to `true` and `mDuration` is set to `1f` so the effect will run for one second and the enhanced `for` loop calls the `setPosition` of every particle to move it to the starting coordinates.

The final method for our `ParticleSystem` 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.

```
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` in turn is drawn using `drawRect` and the `getPosition` method. Notice the call to `paint.setARGB`. 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

Declare a new instance of `ParticleSystem` as a member of `GameEngine`:

```

...
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 we can test the ParticleSystem (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.

Now we need to add our particle system into the Renderer each frame to be drawn. This involves three steps:

1. Change the code in the run method to pass in the particle system to the Renderer.draw method
2. Change the signature of the draw method to accept step one's change
3. Draw the particle system after checking it is active

Let's take look at above steps in detail:

- **Step 1:** Change the call to the draw method inside the run method to look 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 it is active by adding this code in the draw method right after the comment which is already there showing where the code should go.

```
// Draw a particle system explosion here
if (ps.mIsRunning) {
    ps.draw(mCanvas, mPaint);
}
```

That is it – almost. Run the game and tap the screen to trigger the `emitParticles` method.



Wow! One thousand flashing squares sat on top of each other. Looks 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.

Building a physics engine to get things moving

Create a new package private class called `PhysicsEngine` and add 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`. The code simply checks if the `ParticleSystem` is running and if it is calls its `update` method and passes in the required `fps`.

Now we can create an instance of the `PhysicsEngine` and call its `update` method each frame. 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` 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

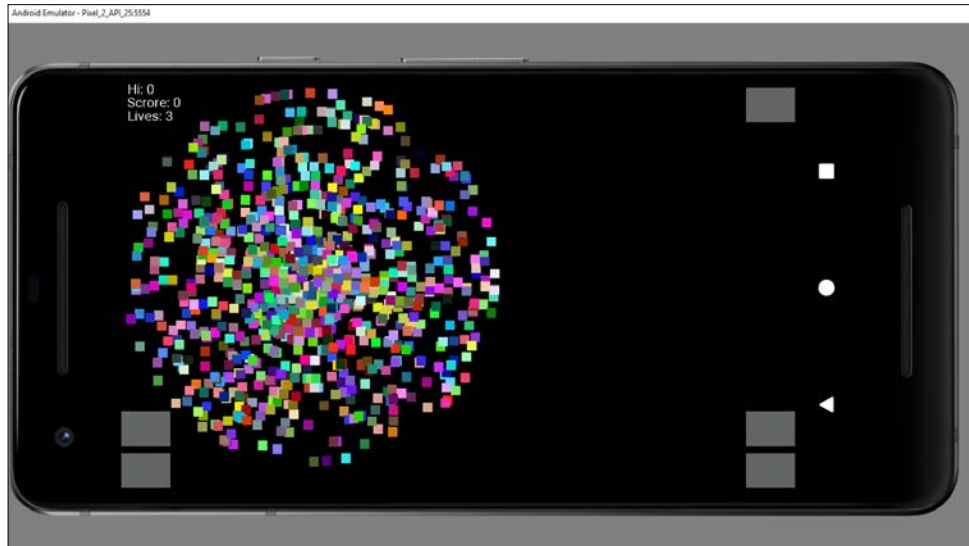
```
if (!mGameState.getPaused()) {
    // Update all the game objects here
    // in a new way

    // 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. Eventually this will be used to detect if the player has died. If it has 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 is that within the `if` condition being tested is the call to `mPhysicsEngine.update` that passes in the frame rate and the particle system.

Running the game

Now you can run the game, tap on the screen and then tap the play/pause button and the `ParticleSystem` class will burst into action.



Quite spectacular for half an hour's work. In fact, you will probably want to reduce the number of particles to less 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.

[ 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.

```
// 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 much ground in this chapter. We have learned a pattern called the Observer pattern and that one class is a broadcaster and 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` ready to use an explosion whenever the player destroys an enemy ship.

In the next chapter we will learn more 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 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 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 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 chapter onwards is for you.

Here is an overview of the topics covered in this chapter:

- 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 the chapter together

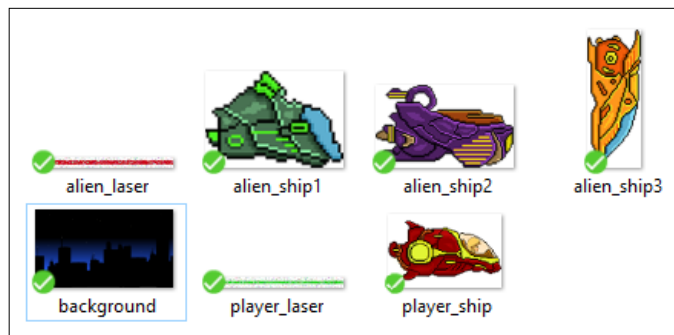
That's quite a big list so we better get started.

Meet 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. Copy and paste them all directly into the `app/res/drawable` folder in the project explorer window of Android Studio.

Reminder of how all these objects will behave

This is an important topic that will prepare us for when we discuss more about design patterns next. Have a quick look at all the graphics that will represent the game objects, so we have a full understanding of what we will be working with.



Now we can learn about the Entity-Component pattern.

The Entity-Component pattern

We will now spend five 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 previous projects we coded a class for each object. We had classes like `Bat`, `Ball`, `Snake` and `Apple`. Then in the `update` method we would update them and in the `draw` method we would draw them. In the most recent project, `Snake`, we took a step in the right direction and had the object handle what happened in both the updating and the drawing phases.

We could just get started and use this same structure for this project. It would work but a few major coding nightmares would become apparent by the end of the project.

The first coding nightmare

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);
```

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 update them:

```
mDiver.update(mFPS);
mChaser.update(mFPS);
mPatroller.update(mFPS);
mPlayerShip.update(mFPS);
```

Then 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 as well. Now consider the collision detection. We will need to separately get the collider of each and every one of the aliens and each and every laser too and 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 ten or even twenty 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 players, all 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` 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 `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 won't have the problem of all that wasted duplicate code. Great!

The problem with this approach is how varied- in some respects the game objects are. For one example, all the alien types move differently. The chasers chase after the player, 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 would we put this kind of diversity into the `update` method that would need to control this movement? Maybe 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 would handle the player and the lasers.

If you remember back to *Chapter 8, Object Oriented Programming*, you might remember that 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, however, there is also a problem with this approach as well.

The GameEngine engine would have to "know" which type of object it was updating or at the very least be able to query the GameObject 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 which did seem to work falls apart on closer inspection. I said that the code in the draw method was the same for six of the objects and therefore the draw method could be part of the parent class and used by all the sub-classes instead of coding six separate draw methods. Well what happens when perhaps just one of the objects needs to be drawn differently, like the scrolling background. The answer is that the solution doesn't work.

Now we have seen the problems that occur when objects are different from each other and yet cry out to be 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 with 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 the GameObject. Then when a GameObject does something differently we 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) the base class.

Notice 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 possible compose instead of inheriting.



The `GameObject` class is the entity and the classes it will be composed of that do things like update its position and draw it to the screen are the components. Hence the Entity-Component pattern.

When we use composition over inheritance to create a group of classes that represent behaviour/algorithms as we have here this is also known as the **Strategy** pattern. You could happily use everything you have learned here and refer to it as the Strategy pattern. Entity-Component is a lesser 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. In *Chapter 26, What next?* I will show you some good resources for this kind of detailed research.

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 glance but brings with it some problems of its own. Not least of which it exacerbates all the problems we discussed in *Chapter 18, Introduction to Design Patterns and much more!*. It would mean that our new `GameObject` class would need to know about all the different types of component 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 a Diver, Patroller, PlayerShip, PinkElephant or whatever 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 into the class itself would make it exceptionally unwieldy and defeat the entire reason for using the Entity-Component pattern.

We would need a constructor that did something like this hypothetical `GameObject` code:

```
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 we will need to do the same for each type of alien, background, player for each and every component. The `GameObject` would need to know not just which components go with which `GameObject` but also which `GameObjects` didn't need certain components like 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 passing an `ArrayList` of `Rect` objects to its `handleInput` method each time `GameEngine` received touch data in the `onTouchEvent` method.

The `GameObject` class would need to understand all this logic. Any benefit or efficiency 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 which teleports near to the player takes a shot 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 in the conventional manner 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 into the `GameObject` class we go to edit all those `if` statements. Then the game designer realises that although the cloaking-divers are cool, the original divers that were always visible were more menacing. We need divers and cloaking-divers. More changes required.

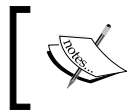
If you want a new `GameObject` that accepts input and then things get even worse because the class instantiating the `GameObject` must make sure to pass in a `GameEngineBroadcaster` reference and the `GameObject` must know what to do with it.

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 the `InputComponent` when it is not needed and wasting calls from the `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 type `Transform`. All `GameObject` instances will be composed with a `Transform` object that holds details like size, position and more besides. More details on the `Transform` class as we proceed with the 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 begin to learn the Factory pattern. Why not do a Web search for the Factory pattern once you have completed this project.

The game designer will provide a **specification** for each and every 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 and 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 the `GameObject`, `GameEngine`, `Renderer` and `PhysicsEngine` remain unchanged. Perhaps 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;
}
```

In the code 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 `GameObject` and we will see their real-life equivalents later in the chapter. This code is not the same as we will be using but it is not too far from it.

It is true that the code strongly resembles the code which we have just discussed and revealed to be so 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 `GameObjects` 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 roles and responsibilities of game design, specific level design and game engine/factory coding.

Summary so far

Take a look at these bullet points that describe everything we have discussed so far:

- We will have component classes like `MovementComponent`, `GraphicsComponent`, `SpawnComponent` and `InputComponent`. These will be interfaces with no specific functionality.
- There will be concrete classes that implement these interfaces like `DiverMovement`, `PlayerMovement`, `StandardGraphics`, `BackgroundGraphics`, `PlayerInput` etc.
- 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 like 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 `GameObjects` which is very easy to update, draw and pass around to the classes that need them.

The object specifications

Now we know that all our game objects will be built from a selection of components. Sometimes the 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 from which all the other specifications can derive so we can 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 the factory class can get at 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 abstract.

Create a new class called `ObjectSpec` and add the following code:

```
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 (`mTag`) that the factory will pass on to the finished `GameObject` instances so the `PhysicsEngine` can make decisions about collisions, and 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 `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 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 have learned how to implement a camera that moves around the game world our sizes will be more natural. You will be able to think of them as metres or perhaps game-units.

Probably the most important member variable to be aware of is the `mComponents` array of `Strings`. This will hold a list of all the components required to build this game object.

Look back at the constructor and 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 super class constructor with the relevant values and the new specification will be fully initialized.

Have a look at all the other methods of this class but 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 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 like 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 which chases after the player and fires lasers at him once it is in line or nearly in line. Add and examine the code and then 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 behaviour. 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 to a speed in pixels when we start coding some components later in this chapter.

The `PointF` for the size (`relativeScale`) is initialized to `15f, 15f`. This will mean the game object and its bitmap will be scaled to one fifteenth the size of the width of the screen. The code that does this will be seen when we code the component classes later in the chapter.

The components array has been initialized with three components:

- The `StdGraphicsComponent` will be a class that will handle the draw method and 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 the `StdGraphicsComponent` will handle the draw method.
- The `AlienChaseMovementComponent` is the class that 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 on each time we call `update` on a `GameObject`.
- The `AlienHorizontalSpawnComponent` 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 super class constructor with `super...` and all the values are passed into the `ObjectSpec` class constructor where they are initialized ready for use in the factory.

AlienDiverSpec

Add this class as shown next to specify the Diver alien that will continually swoop at the player in an attempt to destroy him.

```
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 of 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 most significantly you will notice that the graphics component remains the same, but the movement and spawn components are different.

The `AlienDiverMovementComponent` class will contain the diving logic and the `AlienVerticalSpawnComponent` will take care of spawning the Diver alien off the top of the screen.

AlienLaserSpec

Next add the specification for an alien laser that will be the projectile 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 the `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 `PlayerLaserSpec` but the `PlayerLaserSpec` 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 it uses a unique movement component (`AlienPatrolMovementComponent`) but makes use of the `StdGraphicsComponent` and makes use of the same `AlienHorizontalSpawnComponent` as the `Chaser` alien.

BackgroundSpec

Now 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:

- The `BackgroundGraphicsComponent` will take care of drawing the two copies of the background image side by side.
- The `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 is discussed when we code the component later in this chapter

- The `BackgroundSpawnComponent` spawns the game object in the unique way a background must be.

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);
    }
}
```

The `PlayerSpec` is more advanced than the other specifications. It has a commonplace graphics component but a player-specific movement and spawn components.

Note, however a completely new type of component. The `PlayerInputComponent` will handle the screen touches and will also register as an observer to the `GameEngineBroadcaster (GameEngine)` class.

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 graphics components will need to be initialized (bitmaps loaded and scaled) as well as get drawn each frame. The `initialize` and `draw` methods of the interface make sure this can happen. The different ways that it will happen 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`, the specific (derived) `ObjectSpec` and the size of the screen. All these things will be used to setup the object 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 mentioned earlier in the chapter, every game object will have a `Transform` that holds data about where it is, how big it is and which direction it is travelling.

InputComponent

Next code the InputComponent interface.

```
interface InputComponent {

    void setTransform(Transform t);
}
```

This has just a single method. The `InputComponent` related class, `PlayerInputComponent` will be quite in-depth the way it handles screen touches with all the various button options that the player has. However, the only thing the `InputComponent` interface needs to be sure to facilitate is that the component can update its related `Transform`. The `setTransform` method passes a reference in and then the component can manipulate 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, `move`. Look at the parameters they are quite significant. First of all, the frame rate so that all the objects can move themselves according to how long the frame has taken. Nothing new there but the `move` method also receives two `Transform` references. One is the `Transform` of the `GameObject` itself and the other is the `Transform` of the player. The need for the `Transform` of the `GameObject` itself is obviously so it can move itself according to whatever logic the specific component has.

The reason it also needs the player's `GameObject`'s `Transform` is because many of the alien's movement logic depends upon where they are in relation to the player. You can't chase the player or take a shot at him if you don't know where he is.

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, `spawn`. The `spawn` method also receives the specific `GameObject`'s `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

By coding an empty class for each player related component, it will allow us to more 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 of dipping into the same class (mainly `GameObject`) multiple times.

In this chapter we will deal with the player (and his 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 and we will be able to see how the components interact with the game engine via the `GameObject` class before coding 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 fulfil 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 the 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, the `StdGraphicsComponent`. Add the outline of the class as shown next.

```
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 the chapter. The class implements `GraphicsComponent` and therefore 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` that implements `MovementComponent`.

```

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 the chapter.

PlayerSpawnComponent

Next code the near-empty `PlayerSpawnComponent` that 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 the `PlayerInputComponent` and the `PlayerLaserSpawner` interface. We will cover them both together because they have a connection to one another.

Starting with the `PlayerInputComponent`, we will also add a few member variables to this class and we will then discuss them. Create a new class `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 the class implements two interfaces. `InputComponent` and our old friend from the chapter 18, `InputObserver`. The code implements both the required methods for both interfaces, `setTransform` and `handleInput` (so `GameEngine` can call it with the player's screen touches).

There is also a `Transform` member that will show as an error until we code it shortly and another member too. There is a member called `mPLS` of type `PlayerLaserSpawner`.

Cast your mind back to Chapter 18 when we coded the `GameStarter` interface. We coded the `GameStarter` interface, so 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 `GameEngine`.

We will do something very similar now to allow the `PlayerInputComponent` to call a method in `GameEngine` and spawn a laser.

The `PlayerLaserSpawner` interface will have one method, `spawnPlayerLaser`. By having an instance of `PlayerLaserSpawner` in `PlayerInputComponent` we will be able to call its method and have the `GameEngine` spawn lasers whenever we need it to.

Create a new class and code the `PlayerLaserSpawner` interface shown next.

```

public interface PlayerLaserSpawner {
    boolean spawnPlayerLaser(Transform transform);
}

```

Switch to the `GameEngine` class in Android Studio and make it implement `PlayerLaserSpawner` as highlighted next.

```
class GameEngine extends SurfaceView
    implements Runnable,
    GameStarter,
    GameEngineBroadcaster,
    PlayerLaserSpawner {
```

Now add the required method, `spawnPlayerLaser` to `GameEngine`, it will be empty- for now.

```
@Override
public boolean spawnPlayerLaser(Transform transform) {
    return false;
}
```

Now we can move on to the next component.

LaserMovementComponent

Code the `LaserMovementComponent` which implements `MovementComponent`.

```
import android.graphics.PointF;

class LaserMovementComponent implements MovementComponent {

    @Override
    public boolean move(long fps,
                       Transform t,
                       Transform playerTransform) {

        return true;
    }
}
```

The class implements the required move method.

LaserSpawnComponent

Code the `LaserSpawnComponent` which implements `SpawnComponent`.

```
import android.graphics.PointF;

class LaserSpawnComponent implements SpawnComponent {
```

```

        @Override
        public void spawn(Transform playerTransform,
                           Transform t) {

        }
    }

```

The code includes the empty but required `spawn` method.

BackgroundGraphicsComponent

Code the `BackgroundGraphicsComponent` that implements `GraphicsComponent`.

```

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 {

    private Bitmap mBitmap;
    private Bitmap mBitmapReversed;

    @Override
    public void initialize(Context c,
                           ObjectSpec s,
                           PointF objectSize) {

    }

    @Override
    public void draw(Canvas canvas,
                     Paint paint,
                     Transform t) {

    }
}

```


The two required methods of `GraphicsComponent`, `initialize` and `draw` have been implemented and there are also a couple of member variables ready for use when we code the class in full later this chapter.



The two variable names and the import of the `Matrix` class give a hint at how we will create the scrolling background effect.

BackgroundMovementComponent

Code the `BackgroundMovementComponent` which implements `MovementComponent`.

```
class BackgroundMovementComponent implements MovementComponent {
    @Override
    public boolean move(long fps,
        Transform t,
        Transform playerTransform) {

        return true;
    }
}
```

The code includes the required `move` method.

BackgroundSpawnComponent

Code the `BackgroundSpawnComponent` that implements `SpawnComponent`.

```
class BackgroundSpawnComponent implements SpawnComponent {
    @Override
    public void spawn(Transform playerTransform, Transform t) {

    }
}
```

Notice it contains the required `spawn` method.

We now have an outline for all the component classes that we will complete by the end of the chapter. Next, we will code the `Transform` class.

Every GameObject has a transform

As we learned earlier in the 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 quite a 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 becoming even more complicated. While planning the 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 `GameObjects` will need all the functionality/data. This isn't 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 like size, position, orientation and heading plus related methods. Ours will also contain things like colliders and a method to help 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 shown next.

```
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 re-list all those members and give 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 representing the horizontal position on the screen where two bitmaps that represent the background meet. If this sounds slightly odd all will be explained in the section *Coding a scrolling background* later 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 at which they meet the scrolling effect is achieved. See *Coding a scrolling background* for full details.
- `RectF mCollider`: As we have done in other projects a `RectF` will represent the area occupied by the object and can be used to perform collision detection.
- `PointF mLocation`: The pixel position of the top-left corner of a game object. 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 depends upon 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. Next follows 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 travelling (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 the `Transform` itself, however, the `Transform` refers so often to the screen size that 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 and means there is only one copy of `mScreenSize` shared across all instances of `Transform`.

The constructor receives lots of data, much of which originates from the specification classes and it also receives a `PointF` for the starting location and a `PointF` 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.

```
// 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` representing the scrolling background. They allow the component classes (via their `Transform` reference) to get the value and change/set the value of `mXClip` and `mReversedFirst`.

Add these next 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;
    }

```

This long list of the short methods we just added explains their purpose:

- `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 and 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` one after the other using the location, width and height of the game object. Objects which 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 the 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 of 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 which 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 the `RectF` that acts as the collider. Used by the `PhysicsEngine` class to detect collisions.

The last method of the `Transform` class is `getFiringLocation`. Add the method and 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 (quite arbitrary) 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 above 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 and none of the game objects except the background need horizontal clipping and only some ships need `getFiringPosition`. Technically this is bad practice, but we will refine `Transform` 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 etc) combination of our various components.

Create the `GameObject` class and add the `import` statements and constructor shown next.

```
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;
}
```

We can see in the previous code that we have an instance of the `Transform` class called `mTransform`. In addition, we have a boolean called `isActive` which will act as an indicator 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 section *Coding all the specific object specifications*.

The final three members that we declare are the most interesting because they are instances for our component classes. Notice that the types are declared as the interface types, `GraphicsComponent`, `MovementComponent` and `SpawnComponent`. Therefore, whatever 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 and 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 `SpawnComponent`.

The `setGraphics` method initializes the `GraphicsComponent` instance with the reference passed in (and some other values). As with the `SpawnComponent`, the `GraphicsComponent` could be of any type that implements `GraphicsComponent`.

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 `MovementComponent` 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...` we dream up in the future (perhaps a `PinkElephantStampedingMovementComponent`). As long as it correctly implements the interface `MovementComponent` interface it will work in our game.

The `setInput` method is a bit different because it uses the `InputComponent` passed to it and calls its `setTransform` method to pass in `mTransform`. The `InputComponent` now has a reference to the appropriate transform. 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 the `InputComponent`, it just passes in the `Transform` and can now forget it exists. The `InputComponent` must also register as an `Observer` to the `GameEngine` (and we will see that soon) and 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 that 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 labour 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` to call its draw method, the update method uses the `MovementComponent` to call its move method and the spawn method uses the `SpawnComponent` to call its spawn method. It doesn't matter what the game object is (alien, laser, player, background etc) 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 the `Transform`.

The move method gets sent the required current object's `Transform` as well as the player's `Transform` form and in addition checks the return value of the move method to see if the object needs to be deactivated.

The spawn method first checks that the object isn't already active before calling spawn on the component and setting the object as active.

There are four more simple getter and setter methods. Add them to the `GameObject` class and then we can move on.

```
boolean checkActive() {  
    return isActive;  
}  
  
String getTag() {  
    return mTag;  
}  
  
void setInactive() {  
    isActive = false;  
}  
  
Transform getTransform() {  
    return mTransform;  
}
```

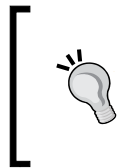
The four methods just added give access to whether the object is active, its tag and its `Transform`.

We can now start adding the code that make our various component classes do something- because 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 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.



The author hopes the reader understands that if we hadn't coded the empty component classes before `Transform` and subsequently `GameObject`, that 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, `initialize` and `draw`. Let's add the code into the 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 before. 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, `createScaledBitmap` is used to scale the `Bitmap` to the correct size for the game object.

Finally, a reversed version of the `Bitmap` is created with another `Bitmap`. Now we can show any `GameObject` with a `StdGraphicsComponent` facing left or right.

Now add the highlighted code into 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 the `Bitmap` that is facing right or left.

Completing the `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` to determine which way(s) the ship is heading and to move it accordingly. In a few pages time we will also code the `PlayerInputComponent` that will manipulate the `Transform` according to the player's screen touches. This is the class that responds to those manipulations.

Add the 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 height, the location of the object, speed of the object and the height of the object represented by the local variables, `screenHeight`, `location`, `speed` and `height`, respectively.

Next in the `move` method we use an `if` statement combined with an `else if` to determine if the ship is heading either up or down. Here it is again.

```

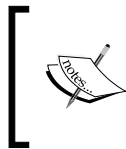
// Move the ship up or down if needed
if(t.headingDown()){
    location.y += speed / fps;
}
else if(t.headingUp()){
    location.y -= speed / fps;
}

```


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 `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) and 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 the PlayerSpawnComponent

This is a very simple component. This code executes whenever the `GameObject`'s `spawn` method is called. Add the 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 that 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 the PlayerInputComponent

This class is fairly long but not too complicated when taken a bit at a time. First add the following code to the `PlayerInputComponent` 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` 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 a made a reference to the `Transform` of the `GameObject`. Remember that the `GameObject` class was passed a `InputController` reference by the factory (soon to code `GameObjectFactory`) class and uses that reference to call this method.

Now that `mTransform` is a reference to the actual `Transform` that is part of the `GameObject` that is the player's ship the `handleInput` method can use it to manipulate it. Remember we manipulate the `Transform` in `handleInput` and the `PlayerMovementComponent` responds to those manipulations. As another reminder it is the `onTouchEvent` method in `GameEngine` which calls `handleInput` because `PlayerInputComponent` is registered as an `Observer`.

Add the highlighted code in the `handleInput` method and then we will discuss it. Be sure to look at the parameters that are passed in because they are key to the discussion on how the method does its job.

```
// 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 `handleInput` down into manageable chunks. First, we use the `getActionIndex`, `getX` and `getY` methods to determine the coordinates of the touch which 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 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 it is possible 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 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 the `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 handle what happens when `x` and `y` are calculated to be inside a specific button. Take a close look next.

```
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 and 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, presented again next together they will look familiar. In fact, apart from the first line of code for each case - the case statement itself the code is identical to the earlier two case statements.

You will notice that instead of `ACTION_UP` and `ACTION_DOWN` we are now responding to the `ACTION_POINTER_UP` and `ACTION_POINTER_DOWN`. The explanation 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 an `ACTION_UP` or `ACTION_DOWN`. If it is the 2nd, 3rd, fourth, etc. then it is held as an `ACTION_POINTER_UP` or `ACTION_POINTER_DOWN`. This hasn't mattered in any of the 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 like pinches, zooms or some custom touch then the order, even the timing and movement while 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 the LaserMovementComponent

We have already coded the `PlayerLaserSpawner` interface, implemented it through the `GameEngine` class, coded the `PlayerInputComponent` to receive an `PlayerLaserSpawner` instance as well as call 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`) which calculates an aesthetically pleasing location to spawn a laser based upon 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 see some code very similar to the PlayerMovementComponent. There is an if and an else-if block which detects which way the laser is heading (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 repeated next:

```
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` is 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 the LaserSpawnComponent

The last bit of code for the laser is the `spawn` method of the `LaserSpawnComponent`. If you are wondering how the laser will draw itself refer to the `PlayerLaserSpec` class and you will see it uses a `StdGraphicsComponent` that we have already coded.

Add the new highlighted code into the `spawn` method as shown next.

```
@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. Notice also that the size of the laser is passed to `getFiringLocation` as 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).

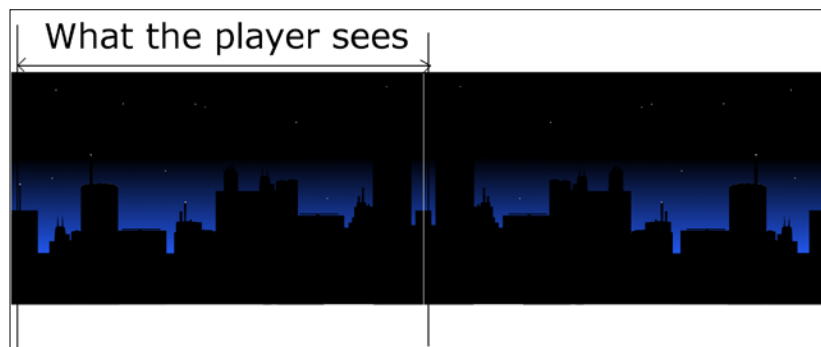
At this point the laser is ready to be drawn and moved.

Coding a scrolling background

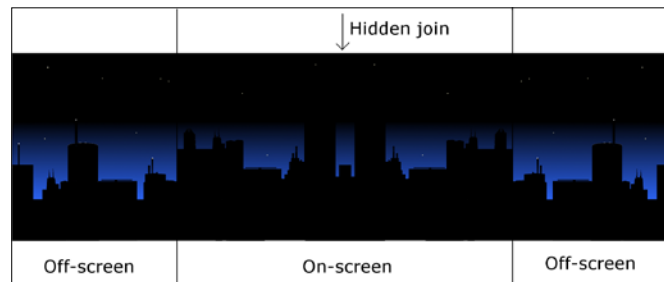
The first frame of the game shows the background image like this. This is unmanipulated in any way.



The way the next frame is shown is to move 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 next image shows a gap between the two images to make it plain 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 the join is invisible.



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.



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 we know what we need to achieve to scroll the background we can code the three background related component classes.

Completing the BackgroundGraphicsComponent

Let's start with the BackgroundGraphicsComponent. The first method we must code is the initialize method. Add the 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 code is nothing we haven't seen before. The resource is selected using the name of the bitmap, it is loaded using `decodeResource`, it is scaled using `createScaledBitmap` then the `Matrix` class is used in conjunction with `createBitmap` 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 that will be called each frame of the game to draw the background. Add the new highlighted code to the `draw` method and 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 need to do 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. The `Transform` holds this value and it is manipulated in the `BackgroundMovementComponent` that we will code next.
- The `width` variable is initialized from the width that the `Bitmap` was scaled to.
- The `height` variable is initialized from the height that the `Bitmap` was scaled to.
- The `startY` variable is the vertical point where we want to start drawing the images. This is simple- the top of the screen- zero. It is initialized accordingly.
- The `endY` variable is the bottom of the screen and is initialized to the height of the screen + twenty 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 one 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 the purpose of explanation 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 see 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` which takes the `Bitmap` to be drawn then a portion of the image to be drawn (`fromRect1` and `fromRect2`) and a screen destination coordinates (`toRect1` and `toRect2`)

Completing the 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 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;
        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 frame rate. If the player is facing left `currentXClip` is increased by the background's speed divided by the current frame rate. In both cases the `setXClip` method is used to update `mXClip` in the `Transform`.

Next in the code there is an `if-else-if` block. These test 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.

We just need to spawn the background and then we can start work on the factory that will compose and construct all these components into `GameObject` instances.

Completing the BackgroundSpawnComponent

Add the highlighted code to the `spawn` method of the `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 backgrounds 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 to complete our game object production line. The first is the factory itself and 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.



We will see how to create a game with multiple different levels for the final project of the book starting in *Chapter 22, Exploring More Patterns and Planning the Platformer Project*.

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

It is the job of the `GameObjectFactory` 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 shown next.

```
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;
    }
}
```

We have a `Context` object, a `PointF` to hold the screen resolution and a `GameEngine` object to hold a reference to the `GameEngine` class. In the constructor all these member variables are initialized. 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 that will do all the hard work 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 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. At last, after all this work we finally create new instance with this code:

```
GameObject object = new GameObject();
```

Next we work out how many components the specification has with this line of code:

```
int numComponents = spec.getComponents().length;
```

We create a soon-to-be-useful `int` called `HIDDEN` and initialize it to `-2000` with this line of code.

```
final float HIDDEN = -2000f;
```

And store the tag from the specification into the `GameObject` using the `setTag` method in this line of code.

```
object.setmTag(spec.getTag());
```



If it seems like my explanations are slightly laborious in this method, it is deliberate. This class, specifically the `create` method is where all the work from this enormous chapter comes together. It is where the `Transform`, `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 have thousands of lines of sprawling unmanageable code.

This next line of code declares and initializes 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();
```

This next line of code declares and initializes 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 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` now has a fully initialized `Transform`. Now for the components.

Add this next code inside the closing curly brace of the `create` method. Notice the highlighted comment at the top of the next code that 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 < mNumComponents; 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;
        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 at that `for` and `switch` conditions. Here they are again:

```
for (int i = 0; i < mNumComponents; i++) {  
    switch (spec.getComponents()[i]) {
```

This will have the effect of looping through every component in the array of components in the current specification we are building. Remember that not all 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 is 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. Into that method a new `PlayerInputComponent` reference is passed and into the `PlayerInputComponent` constructor the `GameEngine` reference is passed. This does two things. Firstly the `PlayerInputComponent` gets to call `addObserver` using the `GameEngine` reference and secondly, `GameObject` can use the `PlayerInputComponent` reference to call `setTransform` and pass the `Transform` reference that `PlayerInputComponent` requires.

The next case creates a new `StdGraphicsComponent` by calling the `setGraphics` method. The `GameObject` responds by storing the reference and calling the `initialize` method to prepare the object to be drawn.

Next, the `PlayerMovementComponent` is created by calling `setMovement` and passing in a new `PlayerMovementComponent`.

The next case statement executes when `LaserMovementComponent` is present. The case just calls `setMovement` in exactly the same way as for `PlayerMovementComponent` with the exception that the `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 the case statements need to do is call `setSpawner` and pass in the appropriate spawn-related class. Here are those two cases again for convenient reference.

```
case "PlayerSpawnComponent":
    object.setSpawner(new PlayerSpawnComponent());
    break;
case "LaserSpawnComponent":
    object.setSpawner(new LaserSpawnComponent());
    break;
```

The next three cases 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 `GameObject`.

```
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 `create` returns.

Coding the Level class

The `Level` class is a place to design the level. If you want more enemies of a certain type or less lasers to make the fire rate less rapid, then this is where to do it. In a game 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. As they are `public`, `static` and `final` they can be easily referred to but not altered, from any class.

There are two variables which aren't `final` these are `mNextPlayerLaser` and `mNextAlienLaser` which will be used 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` 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);
}
```

In the constructor the `ArrayList` is initialized. Next, a new instance of `GameObjectFactory` 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;
}
```

To start with the `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 next line of code again because 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 to add a new reference into the `ArrayList`. The first argument is `BACKGROUND_INDEX` and indicates the position in the `ArrayList` where we want the background to go.

The second argument 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` and add a bunch of lasers for the player to shoot.

The variable `mNextPlayerLaser` is initialized to the value of `FIRST_PLAYER_LASER` ready for the `GameEngine` class to spawn when it is needed.

The last method is the `getGameObjects` method. It returns a reference to `objects`. 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 and we will at last be able to 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. Remember that this method is called by the `GameState` class when a new game is needed to start. Add the 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 code creates a new `ArrayList` called `objects` and initializes it by calling `getGameObjects` on the instance of `Level`. A `for` loop goes through each of the `GameObject` instances and sets them to an inactive status.

Next, we call `spawn` on the player and then we call `spawn` 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 interface method contents.

```
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 the `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` which checks if the `LAST_PLAYER_LASER` was used and if it was sets `mNextPlayerLaser` back to `FIRST_PLAYER_LASER`.

Now, update the `run` method to get all the game objects from the `Level` class and pass them first into the `PhysicsEngine` and then the `Renderer`. The new and altered lines are highlighted.

```
@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 have errors until we update the `PhysicsEngine` and `Renderer` next.

Updating PhysicsEngine

Update the update method with the new and altered highlighted lines of code.

```

// 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 with the method signature to accept an `ArrayList` of `GameObject` instances. The next change is an enhanced for loop that goes through each of the items in the `objects ArrayList`, checks if it is active and if it is calls its update method.

At this point all the objects are being updated each frame. We just need to be able to see them.

Updating Renderer

Update the draw method in `GameEngine` with the new and modified lines which are highlighted next.

```
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);

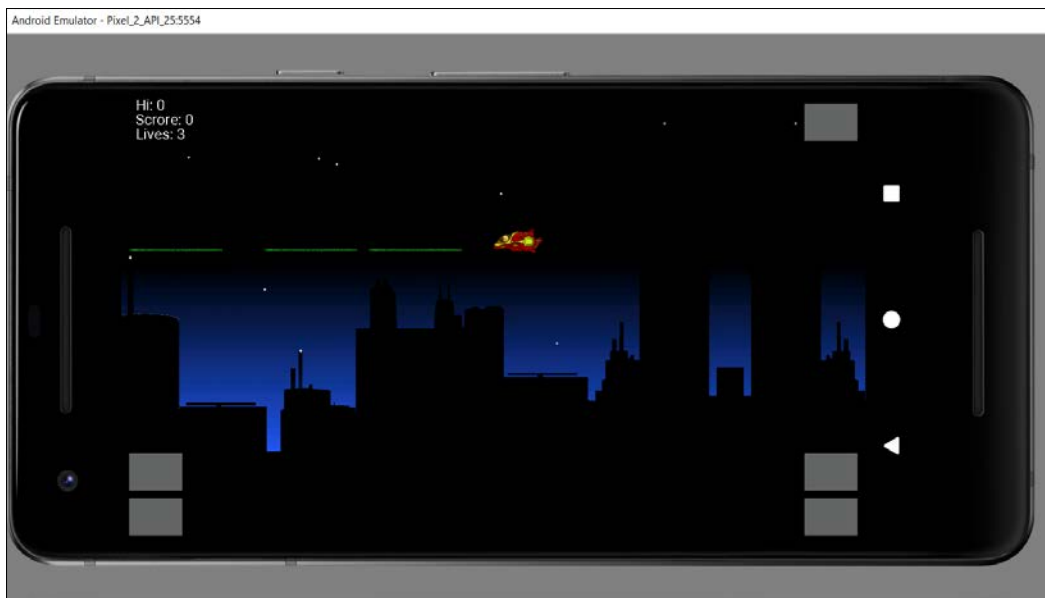
        mSurfaceHolder.unlockCanvasAndPost(mCanvas);
    }
}

```

First, we updated the method signature to receive the `ArrayList` with all the objects in. Next there is an enhanced `for` loop that takes each object in turn, checks if it is active and if it is 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.

Running the game

At last you can run the game and see the city zipping by in the distance.



Test out the pause button to see that you can pause the game, test the flip button to fly the other way and be sure to quickly tap the fire button to test your rapid-fire lasers.

Summary

That wasn't an easy chapter. It doesn't matter if the interconnection of all the various classes and interfaces doesn't make complete sense yet. Certainly, go back and re-read the Entity-Component pattern and the Simple Factory pattern sections along with studying the code so far. However, continuing with the project as we code more of the components to finish the game will also help just as much.

In addition, the next project starting in *Chapter 22, Exploring More Patterns and Planning the Platformer Project* will also help clarify the structure of the code. So, don't spend too long scratching your head if components and factories aren't completely clear- keep making progress because using the concepts will bring clarity much more quickly than thinking about them.

When planning the book, I thought about stopping after the Snake game but that wouldn't be fair because you would have been armed with just enough knowledge to start to implement a bigger game yet 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 of structure we are learning to cope with is a well worthwhile trade off against sprawling, unreadable, buggy classes and methods. Once you grasp patterns (including some more in the final project) and how they work you will be unstoppable with the depth of 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 alien and their 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.

- Code the alien's components
- Spawn the aliens
- Code 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 laser 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 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 method.

```
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` and we will get a refresher on how it works when we implement the rest of the parts of the `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 do 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 will be a one 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 take 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 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 like `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 travelling. 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`, `slowDownRelativeToPlayer` 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 centre (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 towards an alien heading straight at him, he will appear to whizz past them very quickly. As another example, if the player is headed away from an alien (perhaps chasing him), he 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 he will appear to pull away and if he is facing toward the alien he 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 if 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 one in a hundred 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 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 (towards 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 because 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`.

Code 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:



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 except for the name. Next, we will implement it in GameEngine.

Implement 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 which has a `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 than the `AlienChaseMovementComponent` because diving requires much less "thinking" on the part of the alien than chasing does.

After the variables, the code which 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 we will need a `AlienVerticalSpawnComponent` as well to spawn 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. Randomly 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 is stored in `distance` followed by a random float in `height`. We now know where to spawn the alien.

Using an `if-else` block which checks the value of `left` the alien is then spawned using `setLocation` 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 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 an `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 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 if the alien has reached either a vertical or horizontal position limit and if it has, change the direction it is heading.

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 the `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 off-screen. As the aliens always dive downwards we generate two random values. One for the horizontal position (`xPosition`) and one for how many pixels 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.

Spawning the Aliens

Now that all the alien components as well as the `ALienLaserSpawner` are coded we can put them all to work in the game. It will take three steps as follows:

1. Update the `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 `GameObjectFactory` 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 those steps now.

Updating GameEngine

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 the `objects 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 Level

Add this code into the end of the `buildGameObjects` method. 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 the `objects ArrayList` at the required positions.

Now that we are calling `create` with these specifications we will need to update `GameObjectFactory` so it knows how to handle them.

Updating GameObjectFactory

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;

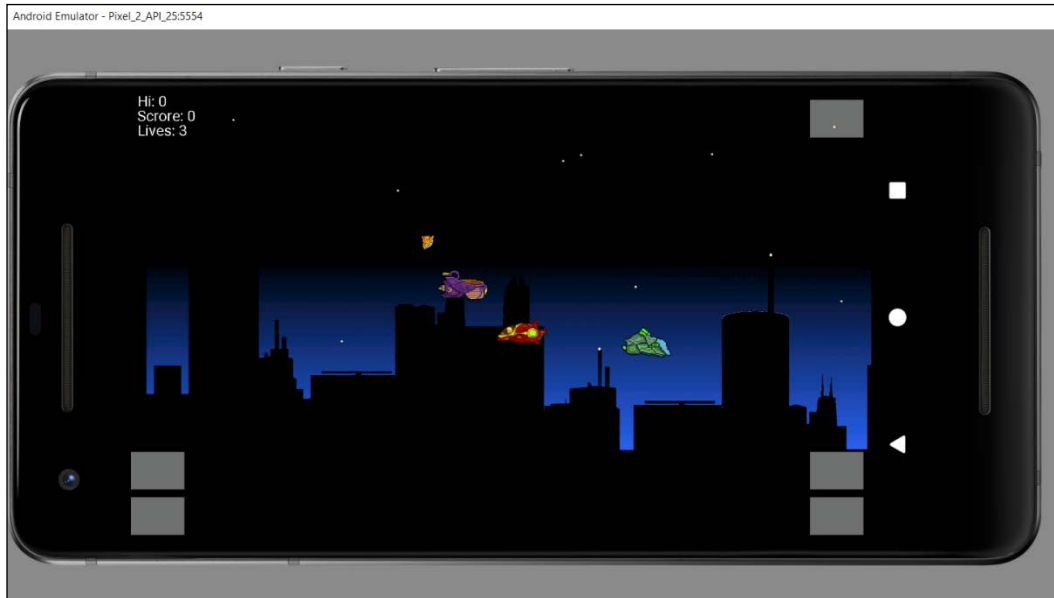
default:
    // Error unidentified component
    break;

```

The new code simply detects the various alien-related components and then initializes them and adds them to the `GameObject` under construction in the same way that we initialized the other movement and spawn-related components when we handled the player's components.

Running the game

Although the game is still not finished we can run it to see the progress so far.



The image shows 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 in to 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 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 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 gol : objects) {

        if(gol.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 mater in the game. For example, we don't test if different aliens collide with each other or if 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 reference to the `SoundEngine` is passed in so 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, cool particle effect is started, the alien is made inactive and then respawned, the laser is set inactive and finally the `playAlienExplode` method plays a sound effect.

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.



Note your high scores are saved. They will remain until you uninstall the application.

Summary

In this project- as I have said before- you have achieved so much. Not just because you have made a new game with neat effects like particles, multiple enemy types and a scrolling background. But because you have built a reusable system that can be put to work on a whole variety of different games.

And we will do just that with the grand finale – a multi-level platform game starting in the next chapter.

22

Exploring More Patterns and Planning the Platformer Project

Welcome to the start of the final project. Over the next four chapters we will build a platform game packed full of features like parallax effect scrolling, animated controllable character, multiple different and challenging levels and much more (keep reading for more details).

During this project we will discover more Java patterns like the **Singleton**, another Java data class, the **HashMap** and explore the gaming concepts of a controllable camera and an animator.

At the same time, we will reinforce all the Java concepts we have already learned including aiding our understanding of and improving upon the entity-component/factory patterns we used in the previous project.

Here is what we will do in this chapter:

- Discuss how we will build the Platformer project including design patterns, Camera class, level design files, improved graphics handling using the Singleton pattern and improved Transform class
- Get started with the Platformer project
- Specify all the game objects with ObjectSpec classes
- Code the interfaces for the components that we will begin coding in the next chapter
- Code the communications interfaces like EngineController, GameEngineBroadcaster and InputObserver

Be sure to read the next information box. It is probably the most useful in the book.



The first 2 chapters of this project are long, and the classes are all interdependent with each other- they are parts of a system. The project is not executable until the end of chapter 23. You can read and code chapter 22 and 23 a step at a time if you like but it will take quite a while- make sure you have at least the day to spare. Alternatively, you could just read chapter 22 and 23 and carefully review the code and the patterns to make sure you understand what is happening. Then you could simply add the graphics and sound (explained in this chapter), create 2 packages (explained in *Chapter 22, Specifying all the game objects with GameObjectSpec classes*, section and *Chapter 23, Create the levels*, section). Then you can just copy and paste all the class files from chapter 23 folder of the download bundle into the proper package of your project (via the Android Studio project explorer window). You will then need to make sure/correct a few lines of text in a few files to do with importing your new packages and the first part of the engine can be working within an hour's work- **after** having first read through chapter 22 and chapter 23 thoroughly.

Choose whichever method you find the most rewarding.



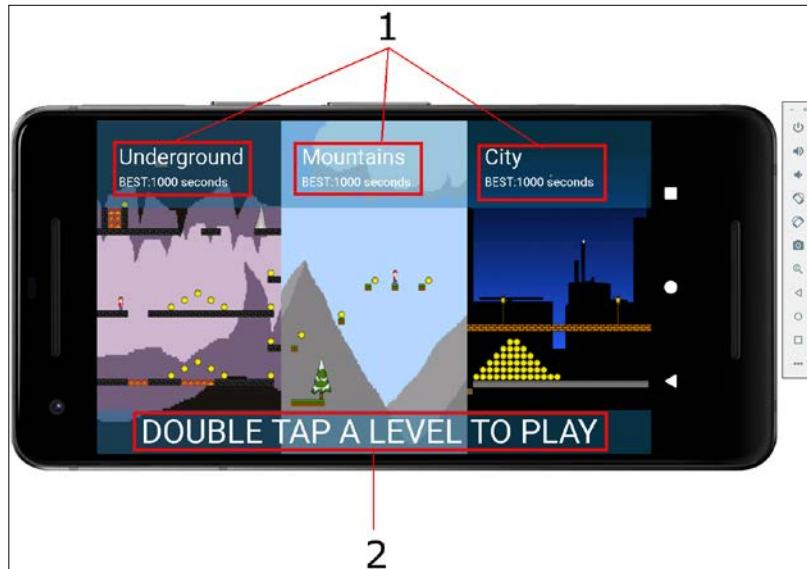
WARNING! If you don't read the two chapters first you won't understand how to make the package import corrections I referred to in the earlier information box, you won't understand the structure of the code and the code won't work.

Now we can make a start. Let's call the platformer, Bob was in a hurry.

Platform Game: Bob Was in A Hurry

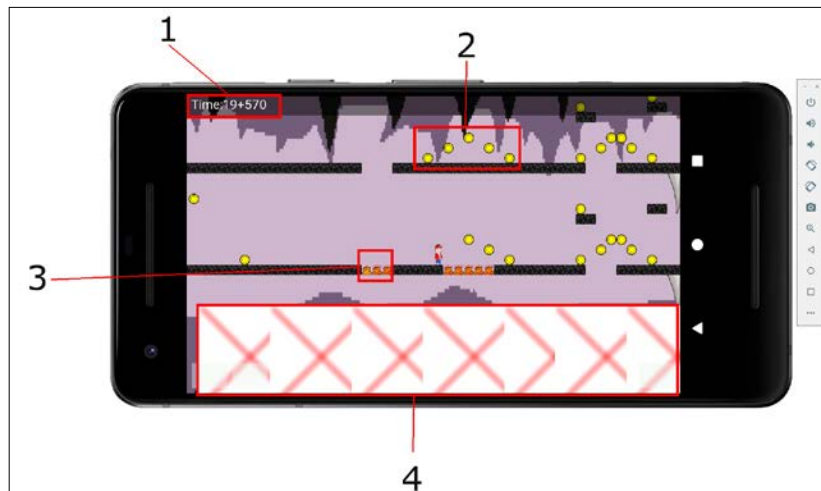
This is the most advanced game in the book. It has multiple levels and hundreds of game objects per level. The player must get from the start of the level, find and then reach the exit. Fastest times are kept as high scores for each of the three levels. However, there are also collectable coins scattered throughout the levels and a time penalty is added on to the finish time for each one left uncollected. Precise jumps are required, and the best route is not always obvious so a player wanting to do well will need to explore and experiment. The whole level is not visible on screen all at once. The level can be dozens of times the size of the screen, so exploration is required to see the whole thing.

Let's have a look at some screen-shots to understand the game better.



In the previous image you can see the home screen. The text labelled as number one is the fastest times for each of the three levels, Underground, Mountains and Cave. Double tapping on the appropriate third of the screen will start the level and the timer.

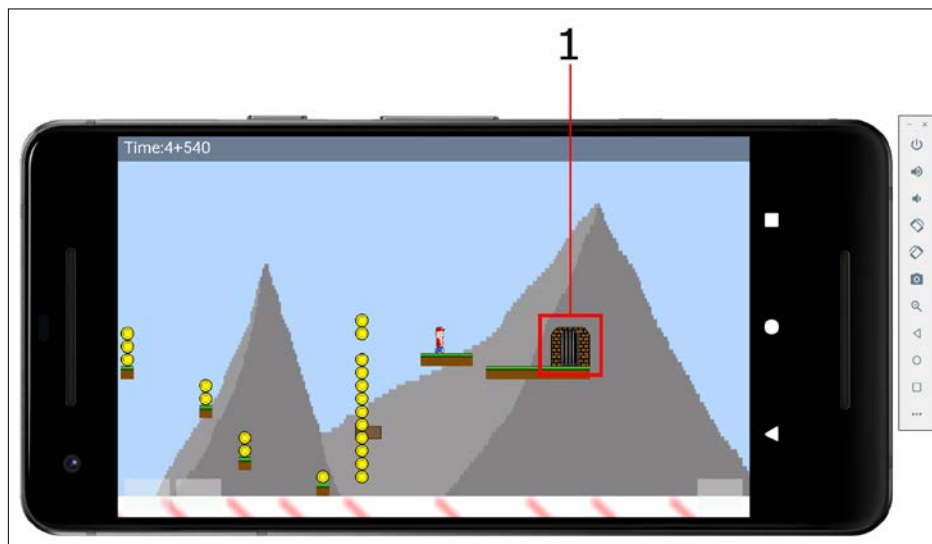
Take a look at this next image taken from the Underground level.



The levels each have their own distinctive background to set them apart from each other and there is also a good selection of decorative objects, platform designs and obstacles for each level. We will see more of the graphics soon. In the previous image notice the following things:

1. The time is in the top left-hand corner. The format is as follows: number of seconds taken so far + number of penalty seconds that can be removed by collecting all the coins.
2. Number two highlights the collectible coins.
3. Number three highlights an animated fire tile. The tile will flicker, making it obvious this tile is bad for the player's health. The slightest touch will end the level and send the player back to the home screen.
4. Number 4 is a Death tile. We need to stop the player accidentally getting out of the level into an eternal fall. By surrounding possible exits with Death tiles, we can end the game when the player falls or gets themselves into an impossible position. The tiles use a distinctive graphic to make level design and debugging clearer. In the finished game you would use a transparent (invisible) graphic or change it to something that matched the level.

This next image shows the Mountain level which is mainly a series of awkward jumps. The part highlighted number 1 is the level destination. When the player reaches the highlighted game object the game goes back to the home screen and if they got a faster time it will be updated.



The next level shown below is the City level. I have borrowed the background from the previous project.



In the image I have highlighted three rectangles. They are moving platforms. They go up and down a distance of 10 blocks. Once you see how to code them, it will be trivial to add horizontal movement or vary the distance they travel.

How we will build the platformer

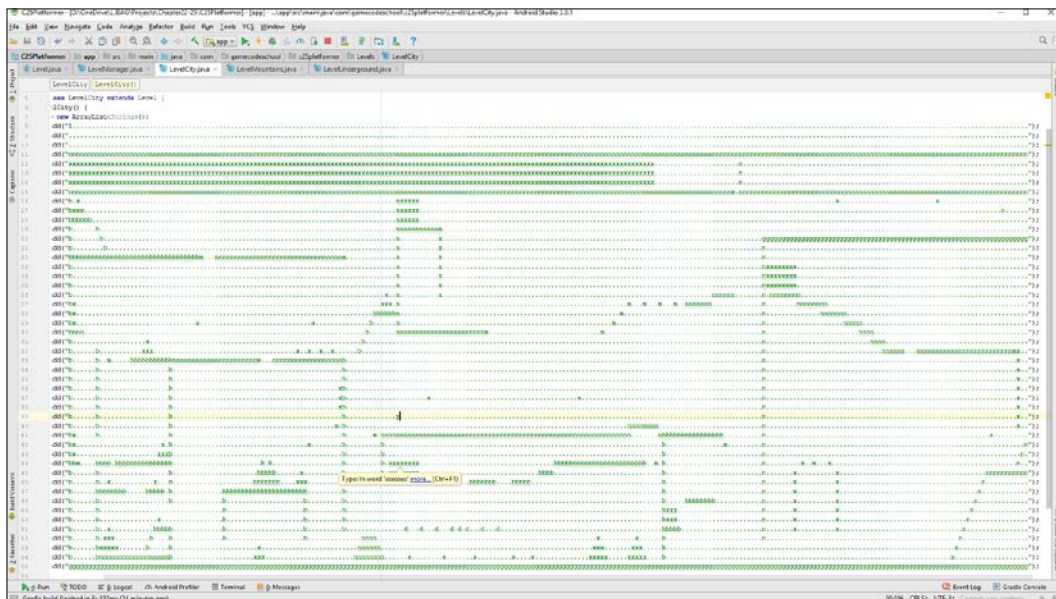
Many things about this project will use things we learned during the Scrolling Shooter project and many things will be completely new. By going through a few things in advance albeit not in great detail will give a clearer picture of how the project will eventually turn into the fully-fledged game by the end of chapter 25.

Level design files example

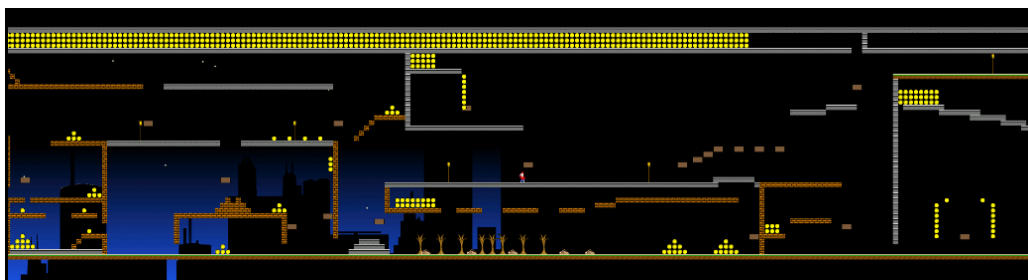
In all the previous projects we spawned objects like bats, balls, snakes, apples, aliens, bullets and lasers based on logic in our code. With this game we need a way to sit back a little and design each level. There are a few different ways we could have done this. We could draw the levels in a graphics application like Paint or Photoshop and then write code to turn a level design image into the game. Another option is to write an entire app that allows the user to select different game objects and then drag and drop them around the screen. The application could then save the design and use it in the game. Both these approaches, unfortunately, would mean a significant increase in the size of the book which is already getting quite chunky. The method I opted for was to lay out the level designs in text, in Android Studio. In the designs, different, alpha-numeric characters would represent different game objects. This is visual and allows the game-designer to create, test and tweak quite easily and is also quite simple to implement.

Take a look at this image of one of the level design files. Also open the Chapter 23/Levels folder in the download bundle. Open the files and you can examine them more closely than the image allows.

I don't provide the level files in the book because they are completely impractical to try and present them on the page. Remember that the download bundle is free and quick to get hold of on the book's page on the Packt website- in case you haven't got it already.



This next image is a zoomed-out view of the game that the game engine translated the previous text file into.

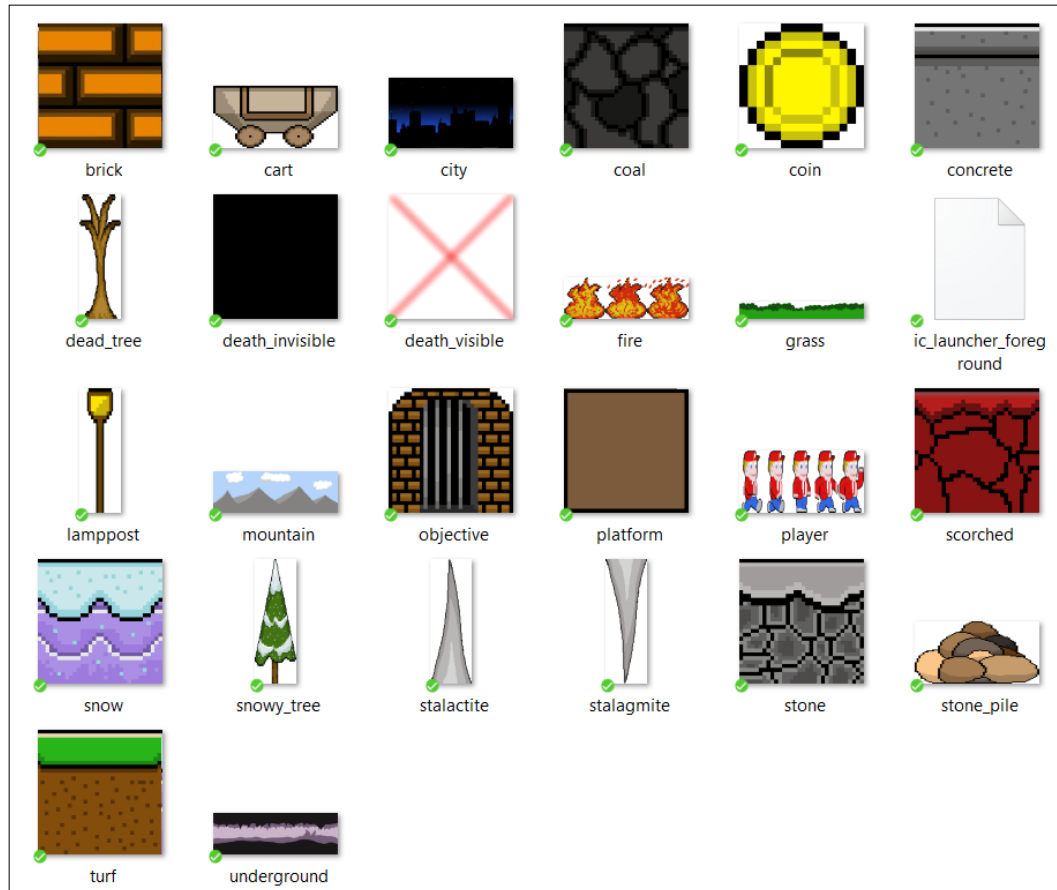


The zoomed-out effect was achieved using the Camera class that we will explore soon.

We will look at the level designs and which letters/numbers to use in greater detail in the next chapter.

The graphics

Here is all the graphics provided for this project.



Notice the graphics we will use as tiles for the player to walk on (coal, concrete, scorched, snow, stone, turf). And look at the graphics that will be purely decorative (no collisions or interactions), dead_tree, lamppost, snowy_tree, stalactite, stalagmite and the three backgrounds, city, mountain and underground. There are also two obstacle graphics. Obstacles are things that don't do anything but will need to be jumped over. These are cart and stone_pile. Notice the coin graphic which is the time-saving collectible. The graphic named platform is the moving platform mentioned earlier.

There are two graphics not mentioned so far. Look at a close up of `player` graphic.



The player graphic is a sprite-sheet. It is a series of frames of animation. Only one of the representations of Bob is shown on screen at any given frame of the game. But our code will loop through them one at a time to create an animated walking effect. This will be controlled by a class called **Animator**.

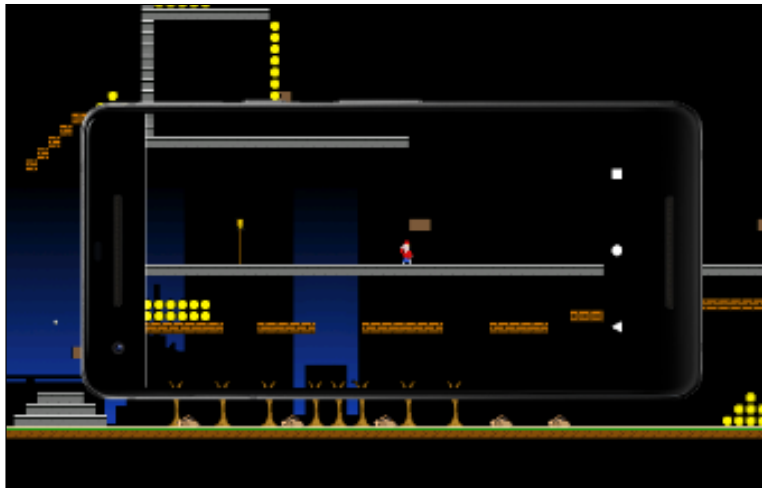
Now have a look at the fire graphic which is also a sprite-sheet and will create a flickering fire animation.



We have seen the graphics, we know how we will decide their layout although we haven't seen any code yet but how will we decide which part of the game world should be on the player's screen at any given frame?

Cameras and the real world

The answer to the question is a special class that can be thought of as the camera which is recording the gameplay. Look at the next image.



The image is a mock-up made in a simple graphics package that attempts to illustrate the camera in action. We can think of most of the game world existing off-screen and unseen and just the relevant portion being shown on-screen.

In the previous project the player's ship sat in the middle of the screen and the way we made it look like the ship was moving was a bit of a bodged job. The scrolling background did most of the work but the various `...AlienMovementComponent` based classes helped too. They took account of the direction the player was facing/flying and moderated (sped up or slowed down) their movement towards or away from the player accordingly. There is a better way.

In this project, all the game objects will have world coordinates. The coordinates held by each of the objects will have no connection to pixels on the screen. It could be simpler if you think about these coordinates as virtual metres. In fact, Bob is 2 units tall which is about the height of a not untypical human. Each and every piece of platform, scenery, collectable, Bob and even the background will have a position in the game world as well as a width and height (also in virtual metres) and not know anything about the screen resolution.

Whenever we move an object we will do so in virtual metres. Whenever we do collision detection we will do so in virtual metres. The virtual metre coordinates can still be held in a `RectF` and the width and height (in virtual metres) can still be held in a `PointF`. However, when the objects are drawn to the screen the `Camera` class will calculate whereabouts- if at all they get drawn. The camera will follow the player so as Bob moves around, and his virtual metres-based coordinates change the camera will reposition on-screen the objects drawn around him creating the illusion of moving through the game world.

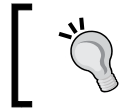
We get to see the code for this in the next chapter and it is surprisingly straight forward.

The slightly modified `ObjectSpec`

In the previous project, because of the lack of a camera we had slightly ambiguous values for width, height and speed. We used relative values that would cause them to scale to the screen size on the device it was running on. In this project, because we have the `Camera` we can use our new virtual metres. This will make the `ObjectSpec` class (which all the specification classes drive from) much clearer. We will see this class and code all the specification classes for the entire project, later this chapter.

New improved transform hierarchy

Also, in this project we will write better code for the Transform of the game objects. We crammed way too much into the Transform class of the Scrolling Shooter project. It was bad design, but it served the purpose of getting the project done without getting any bigger.



If the Scrolling Shooter project was an on-going thing that was going to be extended it would be worth re-writing the Transform class and using a Camera class as well.

For the Platformer project we will have a base/parent Transform class which has much less data in it and suits just the simple game objects like platforms and decorations and we will extend this class to create a BackgroundTransform and PlayerTransform with much more specific methods and data but also still allowing access to the base class's data and methods.

We will code the Transform class in the next chapter and the two extended classes across the rest of the project.

Project patterns

The Entity-Component pattern will be used for every game object in this project just as it was in the previous one. We will not need any spawn-related component classes, however, because starting positions will be determined by the level layout files and none of the objects need to respawn (as the aliens and lasers did).

We will still need a component class which handles what happens during the update phase of the game loop but as many of the objects don't move we will have an UpdateComponent interface instead of a MovementComponent interface. Same job with a new name.

We will still have a GraphicsComponent interface as we did in Scrolling Shooter. We will code these interfaces after we have coded the specification classes later this chapter.

We will still need to communicate from the GameState class to the GameEngine and will again have a EngineController interface to achieve this. Furthermore, we will use the Observer pattern in the exact same way as before to handle communicating the user's input with the classes that need to know about it.

We will also still be using a `GameEngineFactory` class to construct `GameObject` instances using component classes based on the various specification classes. This time the class will need to be a bit smarter to make sure each `GameObject` instance gets the correct type of `Transform`. Otherwise, this class will be very familiar. As already stated we will code all the specifications in a minute but the components themselves will take shape over the rest of the project.

BitmapStore

In this project, some of the levels will have hundreds of game objects. In all the previous projects the game object classes held a copy of the required bitmap. In the Snake game we drew lots of body segments and this was OK because the Snake class reused the body graphic over and over. The platforms, however will only know about themselves, not all the other platforms. This implies that if there is a platform that is 100 tiles long we will need 100 game objects with 100 copies of the graphic. That is not only a waste of memory and processing power but might even cause older Android devices to crash because they have run out of memory. In the next chapter we will code a `BitmapStore` class who's only job is to store and share bitmaps. Therefore, if there is a length of platform in the game that is 1000 blocks wide, if they are all the same object, perhaps turf, then there will only be one copy of the bitmap.

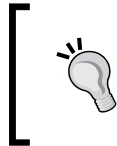
Levels and LevelManager

In the previous project we had the `Level` class. It defined which object types and how many of them were in the game. It used the `GameObjectFactory` class to construct the objects we wanted in the game. If we had wanted more divers and less chasers we would have made changes to `Level`. If we had invented a new type of alien or a smart-bomb for the player, we would have added it in the `Level` class. Most games have more than one level and this platformer is no exception.

To handle this, we will create a `Level` class and then extend it to create specific levels called `LevelCave`, `LevelMountain` and `LevelCity`. These `Level`-based classes will hold all the details of the specific level of the game. We will then create a `LevelManager` class that will understand how to use the `GameObjectFactory` class to load the required level when prompted to do so by the `GameState` class which will work in a similar way to the previous project.

Getting started with Bob was in a hurry

Now we can make a start with the project. Remember that you can do this step by step. All the instructions that follow assume you are doing it step by step, but you could just read chapter 22 and chapter 23 and then refer back to the information box at the start of this chapter to do a super-quick copy & paste job to get the project to its first runnable state.



Final warning! The author recommends you don't attempt to do the copy and paste job until you have thoroughly studied chapter 22 and 23 as there are key things you need to be aware of or the copy & paste won't work.

Creating the project and adding the assets

Create a new project called `Platformer` with an empty Activity called `GameActivity`, no layout file and without backwards compatibility; just as we have for every project in the book.

Grab the project's graphics from the download bundle; they are in the `Chapter 22/drawable` folder. Highlight the contents of this folder and copy them. Now right-click the `drawable` folder in the Android Studio solution explorer and select **Paste**.

Get the sound files from the download bundle in the `Chapter 22/assets` folder. Copy the entire `assets` folder including the folder itself. Now, using your operating systems file browser navigate to the `Platformer/app/src/main` and paste the `assets` folder into your project.

As usual, we want to make the game full screen and locked in landscape orientation. As a reminder, here is how to edit the `AndroidManifest.xml` file:

1. Make sure the `AndroidManifest.xml` file is open in the editor window.
2. In the `AndroidManifest.xml` file, locate the following line of code:
`android:name=".GameActivity">`
3. Place the cursor before the closing `>` shown above. Tap the *Enter* key a couple of times to move the `>` a couple of lines below the rest of the line shown above.
4. Immediately below `GameActivity` but before the newly positioned `>` type or copy and paste these two lines to make the game run full screen and lock it in the landscape orientation.

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
android:screenOrientation="landscape"
```

For more details refer to Chapter 1, section *Locking the game to full-screen and landscape orientation*.

Specifying all the game objects with GameObjectSpec classes

In the previous project the folder with all our class files was getting a bit crowded. This project will have even more classes, so we will help ourselves a little by separating things out into different packages. We will have a folder(package) for all the `GameObjectSpec` related classes and a folder for all the `Level` related classes. Everything else will remain in the usual folder.

When you create a new package, it is important to do so in the correct folder. The correct folder is the same one that we have been putting all the Java files in throughout this book. If you called this project `Platformer` it will be the `app/java/yourdomain.platformer` folder.

In the Android Studio solution explorer, right-click the `app/java/yourdomain.platformer` (NOT `yourdomain.platformer(androidTest)` and NOT `yourdomain.platformer(test)`), select **New | Package** and name it `GOSpec`.

We will now add a whole bunch of classes that will define all the objects of the game; their sizes, tags (for collision detection mainly), speed, starting location, associated graphic file, frames of animation and a list of components that define their behaviour.

There is one parent class, `GameObjectSpec` from which all the other specification classes extend. There are lots of them, 23 in total and I don't recommend you sit there and type them all in. They can all be found in the download bundle in the `Chapter 22/java/GOSpec` folder. You can highlight all the files, copy them and then right-click and **Paste** them all into the `GOSpec` folder/package you just created. What you might need to do manually is click into them and check that the package declaration at the top of each file was correctly auto-adjusted from my package name to yours.

If you used `Platformer` as the project title your package name should be:

```
package yourdomain.platformer.GOSpec;
```



If you check three or four files and the package was auto adjusted correctly then chances are good that they all were, and you don't need to check all 23.

While the specification classes are very similar, I thought it still advantageous to show all the code because they are worth examining. Flick through the next few pages of specification classes. When you look at them take note of the different sizes, speeds, graphics files, frames of animation and especially the different component names in the `components` array. Here is the complete listing from all the specification classes with a few notes on each.



In a professionally built game it would be more typical to load such things from text files with a single line of text per component or attribute. If you want to upgrade your project (wait until you have it all working in chapter 25 first) you can read about parsing a text file using a simple Web search. However, sorting the required data from the text file is an in-depth process (but not beyond a determined beginner). A simple Web search will provide good example code to get started. By doing things this way we keep all our work inside Android Studio and get to play with making our own packages.

The following list of 23 classes appears in the same order they will appear in Android Studio with the exception that I present the base/parent class to the other 22 classes first to aid understanding.

GameObjectSpec

This is the class that all the `...Spec` classes will extend. It provides a related member variable/object instance for each of the specifications a game object can have.

```
import android.graphics.PointF;

public abstract class GameObjectSpec {
    private String mTag;
    private String mBitmapName;
    private float mSpeed;
    private PointF mSize;
    private String[] mComponents;
    private int mFramesAnimation;

    GameObjectSpec(String tag,
                   String bitmapName,
                   float speed,
                   PointF size,
                   String[] components,
                   int framesAnimation ){
```

```
        mTag = tag;
        mBitmapName = bitmapName;
        mSpeed = speed;
        mSize = size;
        mComponents = components;
        mFramesAnimation = framesAnimation;
    }

    public int getNumFrames(){
        return mFramesAnimation;
    }

    public String getTag(){
        return mTag;
    }

    public String getBitmapName(){
        return mBitmapName;
    }

    public float getSpeed(){
        return mSpeed;
    }

    public PointF getSize(){
        return mSize;
    }

    public String[] getComponents(){
        return mComponents;
    }
}
```

The main method (the constructor) has the job of receiving the specifications from all the child classes and copying the values to the member variables/instances. Our code will then be able to handle instances of `GameObjectSpec` regardless of what the actual game object is; animated, controllable player through to dumb, unmovable tree.

All the other methods of this class are simple getter methods that the various parts of our program can call to find out more about the game object.

BackgroundCitySpec

Review the code for this class and then we can talk about it.



I will spend more time reviewing this game object because it is the first one we are looking at. In the remaining 22 I will just point out the occasional distinctive feature. It is still important, however, to glance at and make a mental note of the different specifications of the game objects.

```
import android.graphics.PointF;

public class BackgroundCitySpec extends GameObjectSpec {
    // This is all the specifications for the city background
    private static final String tag = "Background";
    private static final String bitmapName = "city";
    private static final int framesOfAnimation = 1;
    private static final float speed = 3f;
    private static final PointF size = new PointF(100, 70);
    private static final String[] components = new String [] {
        "BackgroundGraphicsComponent",
        "BackgroundUpdateComponent"};

    public BackgroundCitySpec() {
        super(tag, bitmapName, speed,
            size, components, framesOfAnimation);
    }
}
```

The `tag` is mainly used by the collision detection code we will write in *chapter 25*. Even the background which doesn't collide needs to be identified so that we can discount it from collision detection.

The `bitmapName` variable holds the name of the graphics file that is used to represent this game object.

The `framesOfAnimation` variable is set to 1 because this is not a sprite sheet. The player object on the other hand will be set to 5 because that's how many frames there are and the fire object's `framesOfAnimation` will be set to 3.

The `speed` variable in this project is different to that in the previous project because it is the speed in virtual metres per second rather than some convoluted measure relative to the player as it was in the last. Now we are making the jump to game-world based coordinates using a camera we can use these more real-world based size and speed measures.

Next up is `size` and you can consider the city to be 100 metres by 70 metres. When we code the `Camera` class in the next chapter we will see how these virtual metres translate to pixels and how we know which parts to show on the screen of the game at any given frame.

We also have a `String` array for a list of components just like we did in the previous project. The actual components themselves, however will be different to the previous project.

[ The exception to this is this object, the background, which has very similar components.]

Notice the two actual components we will need to code to bring this game object to life:

```
BackgroundGraphicsComponent
BackgroundUpdateComponent
```

Finally notice the constructor that does nothing other than pass all the specifications into the parent class constructor which as we saw stashes the values away in the appropriate variables.

BackgroundMountainSpec

This is the same as the previous accept it uses a different graphic, `mountain.png`.

```
import android.graphics.PointF;

public class BackgroundMountainSpec extends GameObjectSpec {
    // This is all the specifications for the mountain
    private static final String tag = "Background";
    private static final String bitmapName = "mountain";
    private static final int framesOfAnimation = 1;
    private static final float speed = 3f;
    private static final PointF size = new PointF(100, 70);
    private static final String[] components = new String [] {
        "BackgroundGraphicsComponent",
        "BackgroundUpdateComponent" };

    public BackgroundMountainSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```


BackgroundUndergroundSpec

This is the same as the previous accept it uses a different graphic.

```
import android.graphics.PointF;

public class BackgroundUndergroundSpec extends GameObjectSpec {
    // This is all the specifications for the underground
    private static final String tag = "Background";
    private static final String bitmapName = "underground";
    private static final int framesOfAnimation = 1;
    private static final float speed = 3f;
    private static final PointF size = new PointF(100, 70f);
    private static final String[] components = new String [] {
        "BackgroundGraphicsComponent",
        "BackgroundUpdateComponent"};

    public BackgroundUndergroundSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

BrickTileSpec

This is a regular platform tile. Bob will be able to walk on it, so we will need to perform collision detection on it. Its position in the game world never changes so it is technically inanimate. Note though that the relative position of the block on the screen to the camera will change and the brick graphic will need to be drawn accordingly but the Camera class will take care of all of that.

```
import android.graphics.PointF;

public class BrickTileSpec extends GameObjectSpec {

    private static final String tag = "Inert Tile";
    private static final String bitmapName = "brick";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(1f, 1f);
    private static final String[] components = new String[] {
        "InanimateBlockGraphicsComponent",
```

```

        "InanimateBlockUpdateComponent"};

    public BrickTileSpec() {
        super(tag, bitmapName, speed,
            size, components, framesOfAnimation);
    }
}

```

Finally notice the size is just one virtual metre wide and high.

CartTileSpec

This specification has the same components as the previous one but note we can assign any value for size we like. This mine cart object is two metres by one metre. The graphic must of course be drawn to suit the size otherwise it will appear squashed or stretched.

```

import android.graphics.PointF;

public class CartTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "cart";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(2f, 1f);
    private static final String[] components = new String[]{
        "InanimateBlockGraphicsComponent",
        "InanimateBlockUpdateComponent"};

    public CartTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}

```

Review the specifications and move on.

CoalTileSpec

Just another walkable tile.

```

import android.graphics.PointF;

public class CoalTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";

```

```
private static final String bitmapName = "coal";
private static final int framesOfAnimation = 1;
private static final float speed = 0f;
private static final PointF size = new PointF(1f, 1f);
private static final String[] components = new String [] {
    "InanimateBlockGraphicsComponent",
    "InanimateBlockUpdateComponent"};

public CoalTileSpec() {
    super(tag, bitmapName, speed, size,
        components, framesOfAnimation);
}
```

Review the specifications and move on.

CollectibleObjectSpec

Although this next object has quite a different use to say the brick object, notice that everything is the same apart from the graphic file and the tag. This is fine.

```
import android.graphics.PointF;

public class CollectibleObjectSpec extends GameObjectSpec {
    private static final String tag = "Collectible";
    private static final String bitmapName = "coin";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(1f, 1f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "InanimateBlockUpdateComponent"};

    public CollectibleObjectSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

We can detect a collision and determine that it is a collectible item from its tag and then handle what should happen.

ConcreteTileSpec

Just another walkable tile.

```
import android.graphics.PointF;

public class ConcreteTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "concrete";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(1f, 1f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "InanimateBlockUpdateComponent"};

    public ConcreteTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

DeadTreeTileSpec

This tile has minor differences to those we have seen so far. Firstly, it is quite large 2 metres by 4 metres. It also uses a `DecorativeBlockUpdateComponent` instead of an `InanimateBlockUpdateComponent`. The differences between these components in code is small but useful. As trees are just decoration in the background we will not bother to add or update a collider thus saving a few computation cycles per tree, per frame.

```
import android.graphics.PointF;

public class DeadTreeTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "dead_tree";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(2f, 4f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "DecorativeBlockUpdateComponent"};
}
```

```
        public DeadTreeTileSpec() {
            super(tag, bitmapName, speed, size,
                  components, framesOfAnimation);
        }
    }
```

Review the code and move on.

FireTileSpec

This is the first component we have seen that has more than one frame of animation. Our code will detect that an `Animator` object needs to be assigned to this game object and will manage a simple flickering effect on the fire tile using the frames from the sprite sheet `fire.png`.

```
import android.graphics.PointF;

public class FireTileSpec extends GameObjectSpec {
    private static final String tag = "Death";
    private static final String bitmapName = "fire";
    private static final int framesOfAnimation = 3;
    private static final float speed = 0f;
    private static final PointF size = new PointF(1f, 1f);
    private static final String[] components = new String[]{
        "AnimatedGraphicsComponent",
        "InanimateBlockUpdateComponent"};

    public FireTileSpec() {
        super(tag, bitmapName, speed, size,
              components, framesOfAnimation);
    }
}
```

Also notice that the `AnimatedGraphicsComponent` is used instead of the `InanimateBlockGraphicsComponent`.

GrassTileSpec

This another walkable tile.

```
import android.graphics.PointF;

public class GrassTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "turf";
```

```

        private static final int framesOfAnimation = 1;
        private static final float speed = 0f;
        private static final PointF size = new PointF(1f, 1f);
        private static final String[] components = new String[]{
            "InanimateBlockGraphicsComponent",
            "InanimateBlockUpdateComponent"};

        public GrassTileSpec() {
            super(tag, bitmapName, speed, size,
                components, framesOfAnimation);
        }
    }
}

```

Review the code and move on.

This next spec is for the blocks that will cause the player to instantly die and end the game. It helps the level designer prevent embarrassing bugs like falling out of the game world.

```

package com.gamecodeschool.c22platformer.GOSpec;
import android.graphics.PointF;

public class InvisibleDeathTenByTenSpec
    extends GameObjectSpec {

    private static final String tag = "Death";
    private static final String bitmapName = "death_visible";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(10f, 10f);
    private static final String[] components
        = new String[]{"InanimateBlockGraphicsComponent",
            "InanimateBlockUpdateComponent"};

    public InvisibleDeathTenByTenSpec() {
        super(tag,
            bitmapName,
            speed,
            size,
            components,
            framesOfAnimation);
    }
}

```

Review the code and move on.

LamppostTileSpec

This is the same as a tree that we discussed previously except the size and the graphic. Review the specifications and move on.

```
import android.graphics.PointF;

public class LamppostTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "lamppost";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(1f, 4f);
    private static final String[] components = new String[] {
        "InanimateBlockGraphicsComponent",
        "DecorativeBlockUpdateComponent" };

    public LamppostTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the code and move on.

MoveablePlatformSpec

This game object is new. Notice the MovableBlockUpdateComponent. We will code these to make these platforms move in the game world allowing the player to jump on them and be transported up and down.

```
import android.graphics.PointF;

public class MoveablePlatformSpec extends GameObjectSpec {
    private static final String tag = "Movable Platform";
    private static final String bitmapName = "platform";
    private static final int framesOfAnimation = 1;
    private static final float speed = 3f;
    private static final PointF size = new PointF(2f, 1f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "MovableBlockUpdateComponent" };

    public MoveablePlatformSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Note that we can still use a regular `InanimateBlockGraphicsComponent`. Just in case it is causing confusion, `inanimate` refers to the fact it doesn't use a sprite sheet (has no frames of animation), we can still change its location in the game world (move it) via the `MoveableBlockUpdateComponent`.

ObjectiveTileSpec

The objective game object is an important game object because it triggers the player reaching the end of the level. However, as with the collectible object all the significant functionality can be handled by the `PhysicsEngine` via its unique tag and no new components or features are needed for this game object's specification.

```
import android.graphics.PointF;

public class ObjectiveTileSpec extends GameObjectSpec {
    private static final String tag = "Objective Tile";
    private static final String bitmapName = "objective";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(3f, 3f);
    private static final String[] components = new String[] {
        "InanimateBlockGraphicsComponent",
        "InanimateBlockUpdateComponent"
    };

    public ObjectiveTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

PlayerSpec

This is perhaps unsurprisingly our most advanced game object. It has an `AnimatedGraphicsComponent` just as the fire tile does so that its sprite sheet graphic is used to animate its appearance. It also has a specific `PlayerUpdateComponent` that will be much more advanced than all the other `...Update...` components. It will handle things like moving, jumping and falling. The `PlayerInputComponent` will communicate with the `GameEngine` class to receive and translate the player's screen touches. The communication method will use the same `Observer` pattern that we learnt for the previous project.

```
import android.graphics.PointF;

public class PlayerSpec extends GameObjectSpec {
    // This is all the unique specifications for a player
```



```
private static final String tag = "Player";
private static final String bitmapName = "player";
private static final int framesOfAnimation = 5;
private static final float speed = 10f;
private static final PointF size = new PointF(1f, 2f);
private static final String[] components = new String [] {
    "PlayerInputComponent",
    "AnimatedGraphicsComponent",
    "PlayerUpdateComponent"};

public PlayerSpec() {
    super(tag, bitmapName, speed,
        size, components, framesOfAnimation);
}
```

Review the code and move on.

ScorchedTileSpec

Another simple walkable object.

```
import android.graphics.PointF;

public class ScorchedTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "scorched";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(1f, 1f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "InanimateBlockUpdateComponent"};

    public ScorchedTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

SnowTileSpec

Another simple walkable object.

```
import android.graphics.PointF;

public class SnowTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "snow";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(1f, 1f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "InanimateBlockUpdateComponent"};

    public SnowTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

SnowyTreeTileSpec

Just like the dead tree but lush and covered in snow.

```
import android.graphics.PointF;

public class SnowyTreeTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "snowy_tree";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(3f, 6f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "DecorativeBlockUpdateComponent"};

    public SnowyTreeTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

StalactiteTileSpec

This is another purely decorative game object.

```
import android.graphics.PointF;

public class StalactiteTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "stalactite";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(2f, 4f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "DecorativeBlockUpdateComponent"};

    public StalactiteTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

StalagmiteTileSpec

This is another purely decorative object.

```
import android.graphics.PointF;

public class StalagmiteTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "stalagmite";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(2f, 4f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "DecorativeBlockUpdateComponent"};

    public StalagmiteTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

StonePileTileSpec

A pile of stones the player can jump on.

```
import android.graphics.PointF;

public class StonePileTileSpec extends GameObjectSpec {
    private static final String tag = "Inert Tile";
    private static final String bitmapName = "stone_pile";
    private static final int framesOfAnimation = 1;
    private static final float speed = 0f;
    private static final PointF size = new PointF(2f, 1f);
    private static final String[] components = new String [] {
        "InanimateBlockGraphicsComponent",
        "InanimateBlockUpdateComponent"};

    public StonePileTileSpec() {
        super(tag, bitmapName, speed, size,
            components, framesOfAnimation);
    }
}
```

Review the specifications and move on.

Summary of component classes

Looking at the components you will notice the three main types each has. We will therefore need a graphics, update and input related interface. We will code them now. Throughout the rest of this project we will then code the classes that implement these interfaces and match the component classes we just saw in the `GameObjectSpec` related classes.

Coding the component interfaces

Now we will code an interface to suit each of the components that will make up all the different game objects we have just coded the specifications for. We will not code a different interface for each different component type, rather we will code generic types that relate to them all. Allowing all the different types of update and graphics component to be used polymorphically in our code – as we did in the previous project.

Therefore, we will need an update component interface and a graphics component interface.

 You can use copy & paste as you did in the previous section, just be sure to examine the code and understand it to avoid confusion later in the project.

Let's start with the graphics component.

GraphicsComponent

All graphics components will need to be able to do two things. This means they will need two methods. Note that the implementation of the actual components that implement these interfaces are free to add more methods- and some will. But they will be required to implement the required methods of the interface as a minimum.

Review and add the interface to your project.

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.PointF;

import com.gamecodeschool.platformer.GOSpec.GameObjectSpec;

interface GraphicsComponent {

    // Added int mPixelsPerMetre to scale the bitmap to the camera
    void initialize(Context c, GameObjectSpec spec,
                   PointF objectSize, int pixelsPerMetre);

    // Updated to take a reference to a Camera
    void draw(Canvas canvas, Paint paint,
              Transform t, Camera cam);
}
```

Any component that implements this interface will have to add an implementation for its two methods. Therefore, all the graphics related components will implement this interface we can then be sure that we can just call `initialize` and `draw` on any game object and be confident that they will handle initializing and drawing themselves in a manner appropriate to them.

There are still some errors due to missing classes. These include `Camera` and `Transform`. Don't try and auto-import them because we need to code them first. If you try and auto-import them, you might end up with Android classes that are different from the one that we want. All the classes you need to import at this stage are included in the code listing. Especially take note that we are importing our very own package for the first time. In the listing see this line of code highlighted.

```
import com.gamecodeschool.platformer.GOSpec.GameObjectSpec;
```

You will need to adjust this if you used a different domain name to `com.gamecodeschool`.

UpdateComponent

In the previous project we had several different ...Movement... related component classes and we coded a `MovementComponent` interface. The `update` method of `GameObject` then called `move` on each of the appropriate movement-related components. This interface is essentially for the same job but as our objects are more diverse using `UpdateComponent` as the name seemed to make sense. After all trees and platforms aren't technically moving. Add the `UpdateComponent`.

```
interface UpdateComponent {  
    void update(long fps, Transform t, Transform playerTransform);  
}
```

Every single object will have one of these and get called on each frame of the game.

The player's transform could have been left out of the argument list of the `update` method because it is only needed by the background (to see which way the player is moving). However, if we added enemies later they would likely need to know where the player is and what he is doing. So, I left it in rather than create another communication interface between the background and the player.

Coding the other interfaces

There are three more interfaces that we need to code. They are for communications between specific classes of the game. We need the `GameState` class to be able to trigger a new level via the `GameEngine` class and we will code the `EngineController` interface for that.

We will need the `GameEngine` class to broadcast to multiple input related classes whenever the player interacts with the screen and we will need these input-related classes to register themselves as observers just as we did in the previous project. For this we will code the `GameEngineBroadcaster` interface and the `InputObserver` interface.

Let's quickly add them now.

EngineController

Add the `EngineController` interface as shown next.

```
interface EngineController {  
    // This allows the GameState class to start a new level  
    void startNewLevel();  
}
```

In the next chapter we will make `GameEngine` implement this interface as well as pass a reference to `GameState`. `GameState` will then be able to call the `startNewLevel` method.

GameEngineBroadcaster

Add the code for the `GameEngineBroadcaster` interface.

```
interface GameEngineBroadcaster {  
  
    // This allows the Player and UI Controller components  
    // to add themselves as listeners of the GameEngine class  
    void addObserver(InputObserver o);  
}
```

This is the first step in wiring up the `GameEngine` to the classes that need to be able to handle input. Classes that receive a reference to this interface will be able to call the `addObserver` method and receive all the details of the player's screen interaction. We will make `GameEngine` implement this interface in the next chapter.

InputObserver

Add the code for the `InputObserver` interface as shown next.

```
import android.graphics.Rect;  
import android.view.MotionEvent;  
import java.util.ArrayList;
```

```
public interface InputObserver {  
  
    // This allows InputObservers to be called by GameEngine  
    //to handle input  
    void handleInput(MotionEvent event,  
                     GameState gs, ArrayList<Rect> buttons);  
}
```

Each of the classes that handle input will implement this interface and the required method (`handleInput`). They will also share an `InputObserver` reference via the `addObserver` method and our control system will be all set up and ready to go. We will do these final steps during the next chapter.

Summary

There was a lot of talk and theory in this chapter but hopefully it has set us up ready to make fast progress in the next. There was no runnable code in this chapter but in the next we will have a home screen with high scores and even be able to tap the screen to choose a level and see the beginnings of each of the playable levels.

23

The Singleton Pattern, Java HashMap, Storing Bitmaps Efficiently and Designing Levels

This is going to be a very busy and varied chapter. We will learn the theory of the **Singleton** design pattern.

We will be introduced to another of the classes of the Java Collections, `HashMap` in which we will see how we can more efficiently store and make available the wide variety of bitmaps that are required for this project. We will also get started on our new and improved `Transform` class, code the first of the component-based classes, code the all-new `Camera` class and make a significant start on some of the more familiar classes, `GameState`, `PhysicsEngine`, `Renderer`, `GameEngine` and more besides.

Here is a list of what to expect and the order to expect it.

- The Singleton pattern
- The Java `HashMap` class
- The memory problem and the `BitmapStore`
- Coding a basic `Transform`
- Coding the block-based component classes
- Creating the game levels
- Coding a stripped-down `GameObjectFactory`
- Coding the `GameState`

- Coding the `SoundEngine`
- Coding the `PhysicsEngine`
- Coding the `Renderer`
- Explaining and coding the `Camera` class
- Coding the `HUD`
- Coding the `UIController`
- Coding the `Activity`
- Coding the `GameEngine`
- Coding a stripped-down `LevelManger`
- Running the game for the first time

By the end of this chapter we will see the game in its first runnable state!



Throughout the course of this chapter there will be loads of errors in Android Studio because so many of the classes are interconnected and we can't code them all simultaneously. Note also that when there is an error in an interface file (and there will be) attempting to implement that interface will also cause an error in the implementing class. Stick with it until the end of this chapter as it all comes together, and we will see the first results on our screens. Be sure to read the information box at the top of chapter 22 as well as the one below for the quicker copy and paste approach you might like to take.

If you want to copy and paste the code read this next information box.



Feel free to copy and paste these classes if you prefer. All classes go in the usual folder except where I specifically point it out (for `Level` based classes). Be aware that my domain name is used in all the package names in the download bundle. The first package declaration in all the files will probably auto-update to your package name when you paste them into your project. However, when the `import` code refers to the packages that we have created `...GOSpec` (in the previous chapter) and `...Level` (later in this chapter) you will probably need to change the domain name and possibly the project name manually in some or maybe all of the files that `import ...GOSpec` and `...Level`.

If you want to type everything then you will just need to add a new class of the correct name by right-clicking the appropriate folder and selecting **New | Java Class**.

Let's talk about another pattern.

The Singleton pattern

The **Singleton** pattern is as the name suggests is a pattern that is used when we want only one of something. And more importantly, that we need to absolutely guarantee we only have one of something. The something I refer to is an instance of a class. Furthermore, the Singleton pattern is also used when we want to allow global access to some of its data or methods. The class can then be used from anywhere within a project and yet is also guaranteeing that all parts/classes of the project that access the class are using the exact same instance.

Part of the Singleton conundrum is simple. To make parts of it available to any other class you simply make the methods `public` and `static`.



See *chapter 8: Object Oriented Programming* for a reminder about `static` variables.



But how do we guarantee that only one instance can ever be created? We will look at the code next but as a look-ahead what we will do is create a class which has a `private` constructor. Remember that a `private` method can only be called from within the class itself. And a `public` and `static` method that creates an instance then returns a reference- provided an instance has not already been created- and if it has then a reference to the existing instance is returned. This implies that the class will hold a reference to its own instance. Let's look at this with some sample code.

The Singleton code

Here is the class declaration and it contains just one variable. A `private` and `static` instance of an object that is the same type as the class itself.

```
class SingletonClass {  
  
    private static SingletonClass mOurInstance;  
}
```

The name of the class is not relevant just that the name of the class is the same as the type being declared.

Next, we can look at the constructor method.

```
private SingletonClass() {  
    // This is only here to prevent instantiation  
}
```

It has the same name as the class as do all constructor methods but note it is `private`. When a method is `private` it can only be called by code within the class. Note that depending upon the needs of the project, it is perfectly acceptable to have some code in the constructor, just as long as the constructor is `private`.

This next method is default access and returns an object of type `SingletonClass`. Take a look over the code.

```
// Calling this method is the only way to get a SingletonClass  
static SingletonClass getInstance(Context context) {  
  
    if(mOurInstance == null) {  
        mOurInstance = new SingletonClass();  
    }  
  
    return mOurInstance;  
}
```

As the class is default access, any code in the same package can call the method. In addition, because it is `static`, it can be called without an instance of the class. Now look at the code in the body. The first line checks whether the `private` member, `mOurInstance` has been initialized with `if(mOurInstance == null)`. If the object has not been initialized (is `null`) then the `private` constructor is called, and the instance is initialized. The final line of code returns a reference to the `private` instance. Also note as will be the case in the platform game, it might not even be necessary to return an instance.

Now we add a method that actually does something. After all, a class must have a purpose.

```
static void someUsefulMethod() {  
    // Do something useful  
}
```

Note the previous method, `someUsefulMethod` is `static` so it too, like `getInstance` can be called without a reference to the class. This is why I said the `getInstance` method doesn't necessarily have to return an instance of the class.

Next, we add another method. Note that it is not `static`.

```
void someOtherUsefulMethod() {
    // Do something else useful
}
```

As `someOtherUsefulMethod` is not `static`, an instance of the class would be needed in order to call it. So, in this specific case `getInstance` would need to return a reference to the object.

Now let's see how we could use our new class in some other class of our project. First, we declare an instance of `SingletonClass` but as we cannot call the constructor we initialize it by calling the static `getInstance` method.

```
SingletonClass sc = SingletonClass.getInstance();
```

The `getInstance` method will check whether the member `SingletonClass` object needs to be initialized and if required calls the constructor. Either way, `getInstance` returns a reference to the private member instance of `SingletonClass` which initializes `sc`.

Now we can use the two methods that actually do something useful.

```
// Call the static method
SingletonClass.someUsefulMethod();

// Call the other method
sc.someOtherUsefulMethod();
```

The first method, `someUsefulMethod` is called without using a reference because it is `static`. The second method is called using the `sc` reference we initialized using `getInstance`.



It might surprise you to learn that the Singleton pattern is controversial. It's very use is discouraged even banned in some situations. There are a number of reasons for this but in one-person or small-team projects many of the objections are either not relevant or much less relevant. Android Studio even has a way to auto-generate a singleton. If you are interested in a discussion about the use of the Singleton pattern then a quick Google will bring up strong condemnations, spirited defences and even heated arguments about its use. I suggest you read this article because it gives a fairly balanced view on Singleton and from a game development perspective. <http://gameprogrammingpatterns.com/singleton.html>. Note that the article discusses Singleton in the context of a different programming language, C++ but the discussion is still useful.



If you are going for an interview for a programming job very soon you need to read the previous information box. Otherwise enjoy your Singletons.

Next, we will learn about the Java `HashMap` class and then we will get to code a Singleton for real.

More Java Collections – Meet Java HashMap

Java `HashMap`s are neat. They are part of the Java Collections and a kind of cousin to `ArrayList` which we have now used in two projects (this will be the third). They basically encapsulate really useful data storage techniques that would otherwise be quite technical for us to code successfully for ourselves.

We will get practical with `HashMap` in the next section when we discuss a problem regarding storing `Bitmap` instances in our `GameObject` instances. `HashMap` will be the second part (Singletons are the first part) of the solution to this problem.

I thought it would be worth taking a first look at `HashMap` on its own.

Suppose, we want to store the data of lots of characters from an RPG type game and each different character is represented by an object of type `Character`.

We could use some of the Java tools we already know about like arrays or `ArrayList`. However, Java `HashMap` is also similar to these things but with `HashMap` we can give a unique key/identifier to each `Character` object and access any such object using that key/identifier.



The term hash comes from the process of turning our chosen key/identifier into something used internally by the `HashMap` class. The process is called hashing.

Any of our `Character` instances can then be accessed with our chosen key/identifier. A good candidate for a key/identifier in the `Character` class scenario would be the character's name.

Each key/identifier has a corresponding object, in this case it is of type `Character`. This is known as a key-value pair.

We just give `HashMap` a key and it gives us the corresponding object. No need to worry about which index we stored our characters, perhaps Geralt, Ciri or Triss at, just pass the name to `HashMap` and it will do the work for us.

Let's look at some examples. You don't need to type any of this code just get familiar with how it works.

We can declare a new `HashMap` to hold keys and `Character` instances like this code:

```
Map<String, Character> characterMap;
```

The previous code assumes we have coded a class called `Character`.

We can initialize the `HashMap` like this:

```
characterMap = new HashMap();
```

We can add a new key and its associated object like this.

```
characterMap.put("Geralt", new Character());
```

And this:

```
characterMap.put("Ciri", new Character());
```

And this:

```
characterMap.put("Triss", new Character());
```



All the example code assumes that we can somehow give the `Character` instances their unique properties to reflect their internal differences elsewhere.

We can then retrieve an entry from the `HashMap` like this:

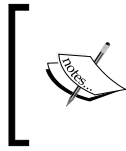
```
Character ciri = characterMap.get("Ciri");
```

Or perhaps use the `Character` class's methods directly like this:

```
characterMap.get("Geralt").drawSilverSword();

// Or maybe call some other hypothetical method
characterMap.get("Triss").openFastTravelPortal("Kaer Morhen");
```


The previous code calls the hypothetical methods `drawSilverSword` and `openFastTravelPortal` on the `Character` class.



The `HashMap` class also has lots of useful methods like `ArrayList`. See the official Java page for `HashMap` here: <https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html>.

Now we can use `HashMap` for real. Before we do let's look ahead in this project and we will see that we have a bit of a memory problem.

The memory problem and the BitmapStore

In all the projects since we introduced the `Bitmap` class back in *Chapter 13* the class representing the object in the game also held a copy of the `Bitmap`. It is slightly different in the previous project because each `GameObject` instance had a graphics-related component class of some type which held the `Bitmap`. It still amounts to the same thing because each game object still has/needed a copy of the appropriate `Bitmap`.

When we have just a few of each alien type as we did in the previous project this is not a problem but, in this project, we will have more than a hundred of some of the `Bitmaps` representing the tiles that make the platforms. At best this is inefficient and will waste memory, device power and make the game run more slowly and at worst (especially on older devices) the game will crash because it has run out of memory.

We need a way to share `Bitmap` instances between objects. This suggests a central store. Hey; what about that `HashMap` and `Singleton` thing we just learned about. How convenient.

Coding the BitmapStore class

All our game objects already have a specification that holds the name of the required `Bitmap` in a `String`. We have also just learned how we can store objects into a `HashMap` and retrieve them using a key. In addition, we know how to make a class available to all other classes using the `Singleton` pattern.

What we will do is code a `Singleton` class called `BitmapStore` which will hold all the required `Bitmap` instances using the `bitmapName` `String` (from the graphics-based component classes). We will however only store one of each `Bitmap`.

When the `initialize` method of the graphics-related component class executes instead of initializing its own `Bitmap` it will call the `addBitmap` method of the `BitmapStore` class. If the `Bitmap` in question has not already been added to the `HashMap` then it will do so. When the `Bitmap` needs to be drawn (during the `draw` method) each object will simply call the `getBitmap` method of the `BitmapStore` class. The `BitmapStore`, as we will see, will have two `HashMap`s, one for the regular `Bitmap` instances and one for reversed instances. The `BitmapStore` class will also have a `getReversedBitmap` method.

Create a new class called `BitmapStore`, add the following members and the constructor as shown next.

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Matrix;
import android.graphics.PointF;

import java.util.HashMap;
import java.util.Map;

class BitmapStore {
    private static Map<String, Bitmap> mBitmapsMap;
    private static Map<String, Bitmap> mBitmapsReversedMap;
    private static BitmapStore mOurInstance;

    // Calling this method is the only way to get a BitmapStore
    static BitmapStore getInstance(Context context) {

        mOurInstance = new BitmapStore(context);
        return mOurInstance;
    }

    // Can't be called using new BitmapStore()
    private BitmapStore(Context c) {
        mBitmapsMap = new HashMap();
        mBitmapsReversedMap = new HashMap();

        // Put a default bitmap in each of the maps
        // to return in case a bitmap doesn't exist
        addBitmap(c,
            "death_visible",
            new PointF(1, 1),
            128,
            true);
    }
}
```

There are three private members. Two `HashMap`s called `mBitmapsMap` and `mBitmapsReversedMap`. They will hold all the `Bitmap`s and reversed `Bitmap`s (when an object requires it) respectively. Notice that they use a `String` as the key and of course a `Bitmap` as its pair.

The third and final private member should be as expected. It is an instance of `BitmapStore` itself as discussed in the Singleton pattern. And, also as should be expected the constructor is private. It cannot be called from outside of the `BitmapStore` class.

As expected with the Singleton pattern we have a static `getInstance` method which calls the `BitmapStore` constructor if it hasn't done so already and then returns a reference to the one and only instance of the class.

Inside this constructor the two `HashMap` instances are initialized and the `addBitmap` method is called. We will code the `addBitmap` method shortly so there will be an error because the method doesn't exist yet. It is worth explaining the details of this call now, however.

If you look at the arguments in the call to `addBitmap` you will see, we pass in the `Context` (`c`), a `PointF` containing a size (which will allow `addBitmap` to scale the `Bitmap`) a somewhat arbitrary value of 128 and a `boolean` with a value of `true`.

The value of 128 is a value which has been predetermined as the number of pixels on the device which represents one metre of the game world. If the game world is to be scaled by our soon-to-be-coded `Camera` class, then it needs to know what to scale it to. When you see the `Camera` class later this chapter this will be clearer.

The final `boolean` indicates whether a reversed copy of the `Bitmap` is required as well.

We mentioned in the previous chapter while exploring the graphical assets that the `death_visible` graphic is for creating an inescapable wall around the level which will kill the player if touched. This helps avoid bugs like the player falling out of the level and then tumbling for eternity through space. The graphic is noticeable, so the level tester can clearly see that he has sufficiently wrapped the level during the development phase and then before the game is released it can be replaced with a blank graphic so that only the background is visible, but the player is still contained within where the level designer wants him.

All the other calls to `addBitmap` will be made by the `initialize` method within the graphic-related component classes.

Now add the `getBitmap` and `getBitmapReversed` methods.

```
static Bitmap getBitmap(String bitmapName) {

    if (mBitmapsMap.containsKey(bitmapName)) {
        return mBitmapsMap.get(bitmapName);
    } else {
        return mBitmapsMap.get("death_visible");
    }
}

static Bitmap getBitmapReversed(String bitmapName) {

    if (mBitmapsReversedMap.containsKey(bitmapName)) {
        return mBitmapsReversedMap.get(bitmapName);
    } else {
        return mBitmapsReversedMap.get("death_visible");
    }
}
```

All that these methods do is receive a `String` and then return the `Bitmap` that matches the key. The return statements are wrapped in an `if-else` block so that when a `Bitmap` does not exist the `death_visible` `Bitmap` is returned. This might at first seem odd. When we see how to lay out level designs we will see that we specifically have a way of adding these `death_visible` objects. And we will specifically request the `death_visible` `Bitmap` when it is required. By returning the `death_visible` `Bitmap` as a default, when we neglect to add a `Bitmap` the `death_visible` `Bitmap` will appear within the level when the game is tested, and we will know we have a `Bitmap` missing. In addition, it will prevent the game from crashing as it would if an uninitialized `Bitmap` was returned.

Add the `addBitmap` method shown next.

```
static void addBitmap(Context c,
                     String bitmapName,
                     PointF objectSize,
                     int pixelsPerMetre,
                     boolean needReversed) {

    Bitmap bitmap;
    Bitmap bitmapReversed;

    // Make a resource id out of the string of the file name
    int resID = c.getResources().getIdentifier(bitmapName,
        "drawable", c.getPackageName());
```

```
// Load the bitmap using the id
bitmap = BitmapFactory
    .decodeResource(c.getResources(), resID);

// Resize the bitmap
bitmap = Bitmap.createScaledBitmap(bitmap,
    (int) objectSize.x * pixelsPerMetre,
    (int) objectSize.y * pixelsPerMetre,
    false);

mBitmapsMap.put(bitmapName, bitmap);

if (needReversed) {
    // Create a mirror image of the bitmap
    Matrix matrix = new Matrix();
    matrix.setScale(-1, 1);
    bitmapReversed = Bitmap.createBitmap(
        bitmap,
        0, 0,
        bitmap.getWidth(),
        bitmap.getHeight(),
        matrix, true);

    mBitmapsReversedMap.put(bitmapName, bitmapReversed);
}
}
```

The `addBitmap` method should look quite familiar by now. The code loads the `Bitmap` from the `String` representing the `Bitmap`'s name. The `Bitmap` is scaled in the usual way using `createScaledBitmap`. The only difference to what we saw in the graphics-based component classes of the previous project is that the size of the object and the value representing the pixels-per-metre is used to determine the size.

The `Bitmap` is added to the `HashMap` and if a reversed copy of the `Bitmap` has been requested then the `Matrix` class is used to create a reversed copy and the `Bitmap` is added to the `HashMap` for reversed `Bitmaps`.

Add the `clearStore` method shown next.

```
static void clearStore() {
    mBitmapsMap.clear();
    mBitmapsReversedMap.clear();
}
```

The `clearStore` method does two things. It first uses the `clear` method of the `HashMap` class to empty the `mBitmapsMap` `HashMap` and then does the same thing for the `mBitmapsReversedMap` `HashMap`. This is needed because typically each level will use a different selection of graphics files and if you don't clear them then every time the player attempts a different level they will be left with unnecessary bitmaps in the store.

To be realistic about this, it really wouldn't matter even for a very basic device to have every single bitmap loaded at once- it's still way more efficient than we have been up until now as we used to load a bitmap for every instance of every game object. But if you added more levels and more object specifications then eventually your game would perform poorly so it is good practice to clear the store before loading the required bitmaps for the current level. This is the purpose of this method.

Coding the basic transform

In this project we will do a better job of the `Transform` class. In the Scrolling Shooter project, the `Transform` was packed full of variables and methods that many of the objects didn't need. It served its purpose to prevent the project getting even bigger and we got away with it because there was a limited number of objects in the game.

What we will code now is a very simple version of the `Transform` and it will contain only the members and methods needed by all game objects. When we need a `Transform` that does more specific things we can then extend this class and add the required extra members and methods.

Add a new class called `Transform`. Code the member variables and the constructor as shown next.

```
import android.graphics.PointF;
import android.graphics.RectF;

public class Transform {
    RectF mCollider;
    private PointF mLocation;
    private float mSpeed;
    private float mObjectHeight;
    private float mObjectWidth;
    private PointF mStartingPosition;
    private boolean mHeadingUp = false;
    private boolean mHeadingDown = false;

    private boolean mFacingRight = true;
    private boolean mHeadingLeft = false;
    private boolean mHeadingRight = false;
```

```
    Transform(float speed, float objectWidth,
              float objectHeight,
              PointF startingLocation) {

        mCollider = new RectF();
        mSpeed = speed;
        mObjectHeight = objectHeight;
        mObjectWidth = objectWidth;
        mLocation = startingLocation;

        // This tells movable blocks their starting position
        mStartingPosition = new PointF(
            mLocation.x, mLocation.y);
    }
}
```

The members should look quite familiar and self-explanatory. We have a `RectF` to represent the collider (`mCollider`), `PointF` instances for the current location and the starting location (`mLocation` and `mStartingPosition`), float variables for speed, height and width (`mSpeed`, `mObjectHeight` and `mObjectWidth`) and boolean variables for each direction the object could be heading.

In the constructor, all the variables representing position, size and speed are initialized. All the values come from the method's parameters. We will see later in the chapter that the `GameObjectFactory` class will have access to all the specifications needed for a level and will instantiate all the `Transform` instances by passing the data we have just seen to the `Transform` constructor. Just as it was in the previous project except this time `GameObjectFactory` will have more than one type of `Transform` to choose from.

Code `updateCollider` and `getCollider` methods shown next.

```
public void updateCollider() {
    mCollider.top = mLocation.y;
    mCollider.left = mLocation.x ;
    mCollider.bottom =
        (mCollider.top + mObjectHeight);

    mCollider.right =
        (mCollider.left + mObjectWidth);
}

public RectF getCollider() {
    return mCollider;
}
```

The `updateCollider` method uses the position of the object along with its width and height to make sure the collider is up-to-date so that collision detection can be done on its precise position. The `getCollider` method makes the collider available to the `PhysicsEngine` class so collision detection can be performed.

Add this bunch of getters and setters shown next.

```
void headUp() {
    mHeadingUp = true;
    mHeadingDown = false;
}

void headDown() {
    mHeadingDown = true;
    mHeadingUp = false;
}

boolean headingUp() {
    return mHeadingUp;
}

boolean headingDown() {
    return mHeadingDown;
}

float getSpeed() {
    return mSpeed;
}

PointF getLocation() {
    return mLocation;
}

PointF getSize() {
    return new PointF(
        (int) mObjectWidth,
        (int) mObjectHeight);
}
```

Although quite a long list the methods we just added are quite simple. They allow the instances of the `Transform` class to share and set the values of some of its private member variables.

Add these getters and setters next.

```
void headRight() {
    mHeadingRight = true;
    mHeadingLeft = false;
    mFacingRight = true;
}

void headLeft() {
    mHeadingLeft = true;
    mHeadingRight = false;
    mFacingRight = false;
}

boolean headingRight() {
    return mHeadingRight;
}

boolean headingLeft() {
    return mHeadingLeft;
}

void stopHorizontal() {
    mHeadingLeft = false;
    mHeadingRight = false;
}

void stopMovingLeft() {
    mHeadingLeft = false;
}

void stopMovingRight() {
    mHeadingRight = false;
}

boolean getFacingRight() {
    return mFacingRight;
}

PointF getStartingPosition(){
    return mStartingPosition;
}
```

These are the final methods for the `Transform` class. They make available to get and set, more of the members of the `Transform` class. Familiarize yourself with the method names and the variables they work with.

Coding the inanimate and decorative components

In the previous chapter we coded all the interfaces for our component classes. We coded `GraphicsComponent`, `UpdateComponent` and `InputComponent`. Now we will code the three simplest component classes that will be enough to complete quite a few of our specifications.

Let's start with the `DecorativeBlockUpdateComponent`.

DecorativeBlockUpdateComponent

The update-related components all implement just one method, `update`. Add a new class called `DecorativeBlockUpdateComponent`, implement `UpdateComponent` and add the empty update method as shown next.

```
class DecorativeBlockUpdateComponent implements UpdateComponent {
    @Override
    public void update(long fps,
        Transform t,
        Transform playerTransform) {
        // Do nothing
        // Not even set a collider
    }
}
```

Yes, the `update` method is meant to be empty as the decorative objects don't move or collide.

InanimateBlockGraphicsComponent

Add a new class called `InanimateBlockGraphicsComponent` and add the code shown next.

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.PointF;
```

```
import android.graphics.Rect;

import com.gamecodeschool. platformer.GOSpec.GameObjectSpec;

class InanimateBlockGraphicsComponent
    implements GraphicsComponent {

    private String mBitmapName;

    @Override
    public void initialize(Context context,
                           GameObjectSpec spec,
                           PointF objectSize,
                           int pixelsPerMetre) {

        mBitmapName = spec.getBitmapName();

        BitmapStore.addBitmap(context,
                               mBitmapName, objectSize,
                               pixelsPerMetre,
                               false);

    }
}
```



Note the highlighted import might need correcting if you are copy and pasting the class files.

Notice that despite this being a graphics-based component there is no `Bitmap` object, just a `String` to hold the name. The class implements the `GraphicsComponent` interface and therefore must provide the implementation for the `initialize` and `draw` methods.

In the code you just added you can see that it receives the required data in the parameters of the `initialize` method then retrieves the name of the `Bitmap` from the `GameObjectSpec` reference.

All the method needs to do is pass all the data on to the `addBitmap` method of the `BitmapStore` class and all the work of initializing a `Bitmap` and storing it for later use (if it hasn't already) is taken care of by `BitmapStore`. Just note that the name of the `Bitmap` is retained in the `mBitmapName` member variable so `InanimateBlockGraphicsComponent` can later access the `Bitmap` again using `getBitmap`.

Add the draw method to the `InanimateBlockGraphicsComponent` as shown next.

```
@Override
public void draw(Canvas canvas,
                  Paint paint,
                  Transform t,
                  Camera cam) {

    Bitmap bitmap = BitmapStore.getBitmap(mBitmapName);
    // Use the camera to translate the real world
    // coordinates relative to the player-
    // into screen coordinates
    Rect screenCoordinates = cam.worldToScreen(
        t.getLocation().x,
        t.getLocation().y,
        t.getSize().x,
        t.getSize().y);

    canvas.drawBitmap(
        bitmap,
        screenCoordinates.left,
        screenCoordinates.top,
        paint);

}
```

Inside the draw method the `getBitmap` method of the `BitmapStore` class is used to initialize a local `Bitmap` object. The `Bitmap` is drawn to the screen in quite a standard way using `drawBitmap` but before it is drawn the `Camera` reference `cam` is used to get the coordinates at which to draw this game object.

Remember that all game objects have world coordinates. The game objects don't know or care anything about the device's screen resolution. The `Camera` class (in the `worldToScreen` method) does a calculation based on the position of the camera to translate the object's world coordinates into appropriate screen coordinates. As we will see later in the chapter, the camera follows the player. So as the player moves through the game world the camera adjusts where on the screen (if anywhere) the game objects get drawn.

InanimateBlockUpdateComponent

Now add a new class called `InanimateBlockUpdateComponent` and code it as follows.

```
class InanimateBlockUpdateComponent
    implements UpdateComponent {

    private boolean mColliderNotSet = true;

    @Override
    public void update(long fps,
                      Transform t,
                      Transform playerTransform) {

        // An alternative would be to update
        // the collider just once when it spawns.
        // But this would require spawn components
        // - More code but a bit faster
        if (mColliderNotSet) {
            // Only need to set the collider
            // once because it will never move
            t.updateCollider();
            mColliderNotSet = false;
        }
    }
}
```

The class implements `UpdateComponent` and therefore has an `update` method. It has one member variable, a boolean called `mColliderNotSet`. In the `update` method the `mColliderNotSet` boolean is checked to test whether the `updateCollider` method on the `Transform` needs to be called. The boolean is then set to false to save a few CPU cycles for the rest of the game. As this object will never move in the game world (although it will move on the screen relative to the camera) this single call to `updateCollider` is sufficient.

Now we have completed three components we can make some levels that have objects that use these components.

Create the levels

Before we make some levels let's create another new package to put them in. As a reminder from the previous chapter, when you create a new package, it is important to do so in the correct folder. The correct folder is the same one that we have been putting all the Java files in throughout this book. If you called this project `Platformer` it will be the `app/java/yourdomain.platformer` folder.

In the Android Studio project explorer, right-click the **app/java/yourdomain.platformer** (NOT **yourdomain.platformer(androidTest)** or **yourdomain.platformer(test)**), select **New | Package** and name it **Levels**.



The step of creating a package is essential before you can copy and paste the class files.

We will now add some classes that will define all the layout of the levels of the game starting with a generic `Level` class which all the others can extend.

Don't add this code yet, just glance at the different letters and numbers in the layouts that will represent different game objects.

```
// Backgrounds 1, 2, 3(City, Underground, Mountain...)
// p = Player
// g = Grass tile
// o = Objective
// m = Movable platform
// b = Brick tile
// c = mine Cart
// s = Stone pile
// l = coal
// n = concrete
// a = lamppost
// r = scorched tile
// w = snow tile
// t = stalactite
// i = stalagmite
// d = Dead tree
// e = snowy tree
// x = Collectable
// z = Fire
// y = invisible death_invisible
```

Now let's see the actual levels comprising the letters and numbers starting with the base class.

Level

This class is really short. It will just be used as a polymorphic type, so we can use the extended classes in common code.

Create a new class called `Level` and add the following code.

```
package com.gamecodeschool.c22platformer.Levels;

import java.util.ArrayList;

public abstract class Level {
    // If you want to build a new level then extend this class
    ArrayList<String> tiles;
    public ArrayList<String> getTiles(){
        return tiles;
    }
}
```

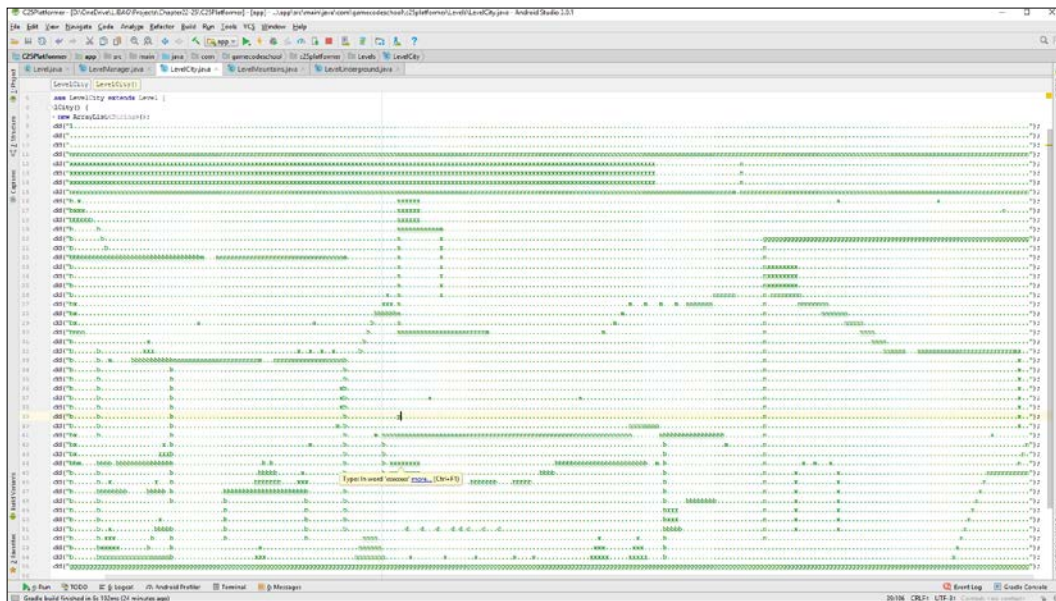


Change the highlighted package statement if copy & pasting

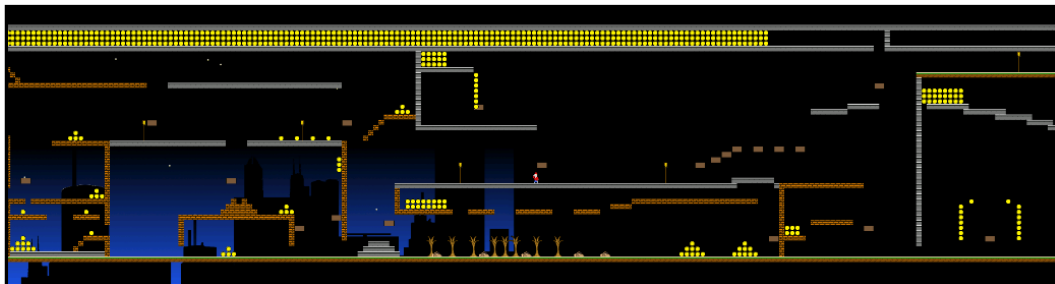
That's it. It just has an `ArrayList` called `tiles` to hold all the game objects and a getter called `getTiles` which unsurprisingly returns a reference to the `ArrayList`.

LevelCity

We have already seen these next couple of images in the previous chapter. They are shown again here for completeness. Copy the `LevelCity` file from the `Chapter 23/Levels` folder of the download bundle. It is not recommended that you try and manually add the code. Paste the file into the `Levels` folder/package of your project in the Android Studio project explorer window.



This is what the level looks like at the end of the project while the camera is zoomed-out.



Next add the Mountains level.

LevelMountains

Copy the `LevelMountains` file from the `Chapter 23/Levels` folder of the download bundle. It is not recommended that you try and manually add the code. Paste the file into the `Levels` folder/package of your project in the Android Studio project explorer window.

[illegible]

This is what the level looks like at the end of the project while the camera is zoomed-out.



Now for the final level.

[illegible]

A screenshot from the video game Super Mario Bros. The scene is a side-scrolling level with a dark sky and a light-colored ground. There are several platforms of different heights and widths. A brick block is visible on the left, containing a power-up. A small enemy is on the ground. The bottom of the screen shows a red and white checkered pattern, likely representing a hazard or a transition.

I spent quite a bit of time designing and testing the City level. It should be quite challenging and possible to keep replaying and improving on your best time. The Mountains and Underground levels I quickly threw together for the sake of showing how the level structure works and how easy it is to have multiple levels. They are intended as a project for the reader to improve or perhaps start again from scratch.

Coding a stripped down GameObjectFactory

The next step is to code the `GameObjectFactory` class. This class will look very similar to the class of the same name in the previous project but there will be a few differences that I will point out. We will code just enough in order to build the game objects that are ready to be built and we will revisit this class in the next chapter once we have finished coding all the component classes.

 In this and other upcoming classes remember to check that when you import your own classes (everything from the `GOSpec` package) make sure/change to the correct package name.

Add a new class called `GameObjectFactory` and add the following member variables and constructor method.

```
import android.content.Context;
import android.graphics.PointF;

import com.gamecodeschool.platformer.GOSpec.GameObjectSpec;

class GameObjectFactory {
    private Context mContext;
    private GameEngine mGameEngineReference;
    private int mPixelsPerMetre;

    GameObjectFactory(Context context,
                      GameEngine gameEngine,
                      int pixelsPerMetre) {

        mContext = context;
        mGameEngineReference = gameEngine;
        mPixelsPerMetre = pixelsPerMetre;
    }
}
```

The `GameObjectFactory` needs three members. A `Context` and an `int` with the number of pixels for every metre of the game world so it can pass them to the graphics-related component classes and a `GameEngine` reference, so it can pass it to any of the input-related component classes, so they can register as observers. These three members (`mContext`, `mGameEngineReference` and `mPixelsPerMetre`) are initialized in the constructor.

Next add the `create` method. We will come back to this method in the next chapter and add more code to the `switch` block.

```
GameObject create(GameObjectSpec spec, PointF location) {
    GameObject object = new GameObject();

    int mNumComponents = spec.getComponents().length;
    object.setTag(spec.getTag());

    // First give the game object the
    // right kind of transform

    switch(object.getTag()){
        case "Background":
            // Code coming soon
            break;

        case "Player":
            // Code coming soon
            break;

        default:// normal transform
            object.setTransform(new Transform(
                spec.getSpeed(),
                spec.getSize().x,
                spec.getSize().y,
                location));

            break;
    }

    // Loop through and add/initialize all the components
    for (int i = 0; i < mNumComponents; i++) {
        switch (spec.getComponents()[i]) {
            case "PlayerInputComponent":
                // Code coming soon
                break;
        }
    }
}
```

```
        case "AnimatedGraphicsComponent":
            // Code coming soon
            break;
        case "PlayerUpdateComponent":
            // Code coming soon
            break;
        case "InanimateBlockGraphicsComponent":
            object.setGraphics(new
                InanimateBlockGraphicsComponent(),
                mContext, spec, spec.getSize(),
                mPixelsPerMetre);
            break;
        case "InanimateBlockUpdateComponent":
            object.setMovement(new
                InanimateBlockUpdateComponent());
            break;
        case "MovableBlockUpdateComponent":
            // Code coming soon
            break;
        case "DecorativeBlockUpdateComponent":
            object.setMovement(new
                DecorativeBlockUpdateComponent());
            break;
        case "BackgroundGraphicsComponent":
            // Code coming soon
            break;
        case "BackgroundUpdateComponent":
            // Code coming soon
            break;

        default:
            // Error unidentified component
            break;
    }
}

// Return the completed GameObject
// to the LevelManager class
return object;
}
```

First, in the create method a new instance of `GameObject` is declared and initialized. Just as we did in the previous project we capture the number of components in the current specification and then call the `setTag` method on the `GameObject` instance. The `GameObject` can now be properly identified by a tag.

Next, we see something new compared to the previous `GameObjectFactory` from the Scrolling Shooter project. We `switch` based on the tag of the specification. There are three possible case statements that can be executed. One for "Background", one for "Player" and a default as well.

For now, we just add code to the default option that calls the `setTransform` method on the `GameObject` instance and passes in a new `Transform` reference. In the next chapter we will extend `Transform` twice to make special versions for the player and the backgrounds. This is how we make sure that every object gets the `Transform` it needs.

Next, we loop round a `for` loop once for each component in the specification. And just as we did in the previous project we initialize the appropriate component at each case statement by calling either `setGraphics` or `setMovement` on the `GameObject` instance and passing in a new component of the appropriate type according to the specification.

Also note I have added all the `case` statements to handle all the other components although most of them are currently empty. This will make it easy to show where the new code goes in the next chapter.

Coding a slightly commented-out game object

The game is coming together nicely, and we can now turn our attention to the `GameObject` class. Much will look familiar to the previous project. Create a class called `GameObject` and add the following member variables.

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.PointF;

import com.gamecodeschool.platformer.GOSpec.GameObjectSpec;

class GameObject {

    private Transform mTransform;

    private boolean mActive = true;
    private String mTag;

    private GraphicsComponent mGraphicsComponent;
    private UpdateComponent mUpdateComponent;

}
```



Check the highlighted import statement is correct for your project.

The members are almost the same as the previous project except we have an instance of an `UpdateComponent` replacing an instance of a `MovementComponent`.

Next add the following methods which are used to initialize the component-based and `Transform` instances that we just declared.

```
void setGraphics(GraphicsComponent g,
                Context c,
                GameObjectSpec spec,
                PointF objectSize,
                int pixelsPerMetre) {

    mGraphicsComponent = g;
    g.initialize(c, spec, objectSize, pixelsPerMetre);
}

void setMovement(UpdateComponent m) {
    mUpdateComponent = m;
}

/*
Uncomment this code soon
void setPlayerInputTransform(PlayerInputComponent s) {
    s.setTransform(mTransform);
}
*/

void setTransform(Transform t) {
    mTransform = t;
}
```

The methods we just coded are called from `GameObjectFactory` and pass in the specific instances of the `GraphicsComponent`, `UpdateComponent` and `Transform`.

Notice I have commented out `setPlayerInputTransform` as we haven't coded this `Transform` extended class yet. Commenting it out will enable us to run the project at an earlier opportunity.

Add the rest of the methods to the `GameObject` class.

```
void draw(Canvas canvas, Paint paint, Camera cam) {
    mGraphicsComponent.draw(canvas,
        paint,
        mTransform, cam);
}

void update(long fps, Transform playerTransform) {
    mUpdateComponent.update(fps,
        mTransform,
        playerTransform);
}

boolean checkActive() {
    return mActive;
}

String getTag() {
    return mTag;
}

void setInactive() {
    mActive = false;
}

Transform getTransform() {
    return mTransform;
}

void setTag(String tag) {
    mTag = tag;
}
```

The code we just added includes the key `update` and `draw` methods which are called each frame and the familiar bunch of getter methods for getting data from the `GameObject` instance.

Coding the GameState

The `GameState` class has exactly the same role as in the previous project. Obviously, however, the details of the state in this project is different. In the previous project we had score, high score lasers and aliens; in this one we have time, multiple fastest times and which level is being played. The `GameState` class also takes care of knowing (and sharing) the current state of paused, playing, drawing, thread running, etc.

Create a new class called `GameState` and add the member variables and constructor as shown next.

```
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;
    private EngineController engineController;

    private int mFastestUnderground;
    private int mFastestMountains;
    private int mFastestCity;
    private long startTimeInMillis;

    private int mCoinsAvailable;
    private int coinsCollected;

    private SharedPreferences.Editor editor;

    private String currentLevel;

    GameState(EngineController gs, Context context) {
        engineController = gs;
        SharedPreferences prefs = context
            .getSharedPreferences("HiScore",
                Context.MODE_PRIVATE);

        editor = prefs.edit();
        mFastestUnderground = prefs.getInt(
            "fastest_underground_time", 1000);
        mFastestMountains = prefs.getInt(
            "fastest_mountains_time", 1000);
        mFastestCity = prefs.getInt(
            "fastest_city_time", 1000);
    }
}
```

We have Boolean variables to represent whether the game thread is running, player has paused, game is over or currently drawing. As in the previous project we also have an `EngineController` reference so `GameState` can reinitialize a game/level directly.

Next up we have three `int` variables to hold the fastest time on each of the three levels. The `startTimeInMillis` variable will be initialized each time a level is attempted to record the time the level was started so it is possible to calculate how long the level took.

There are two more `int` members to hold the number of coins it is possible to collect in a level and the number of coins actually collected. They are `mCoinsAvailable` and `coinsCollected`.

The final two members in the previous code is an instance of `SharedPreferences`. Editor for writing new high scores and a `String` which will represent the current level to be played, City, Underground or Mountains.

Now add this quite long list of getters and setters to the `GameState` class.

```
void coinCollected() {
    coinsCollected++;
}

int getCoinsRemaining() {
    return mCoinsAvailable - coinsCollected;
}

void coinAddedToLevel() {
    mCoinsAvailable++;
}

void resetCoins() {
    mCoinsAvailable = 0;
    coinsCollected = 0;
}

void setCurrentLevel(String level) {
    currentLevel = level;
}

String getCurrentLevel() {
    return currentLevel;
}
```

```
void objectiveReached() {
    endGame();
}

int getFastestUnderground() {
    return mFastestUnderground;
}

int getFastestMountains() {
    return mFastestMountains;
}

int getFastestCity() {
    return mFastestCity;
}
```

A detailed description of each of the methods we just added would be somewhat laborious because they each do just one thing.

- Set a value(s)
- Return a value
- Call another method

It is however, well worth closely inspecting each method's name to aid understanding as we proceed.

Next, add three more methods to the `GameState` class for starting a new game, getting the current time and taking action when the player dies.

```
void startNewGame() {
    // Don't want to be handling objects while
    // clearing ArrayList and filling it up again
    stopEverything();
    engineController.startNewLevel();
    startEverything();
    startTimeInMillis = System.currentTimeMillis();
}

int getCurrentTime() {
    long MILLIS_IN_SECOND = 1000;
    return (int) ((System.currentTimeMillis()
        - startTimeInMillis) / MILLIS_IN_SECOND);
}

void death() {
```

```

        stopEverything();
        SoundEngine.playPlayerBurn();
    }

```

The `startNewGame` method calls the `stopEverything` method. We will code the `stopEverything` method soon. The next line of code uses the `GameController` reference to call the `startNewLevel` method on the `GameEngine` class. Once the `startNewLevel` method has done its work we call the `startEverything` method (which we will code soon) to get things going again. The reason we do these three steps is because otherwise we will be trying to update and draw objects at the same time as the `GameEngine` is also deleting and reinitializing them. This would be bound to cause a crash. The last thing we do in `startNewGame` is initialize the `startTimeInMillis` variable with the current time.

The `getCurrentTime` method shares the current time. Note that it takes the start time from the current time and divides the result by one thousand. This is because we want the player to see their time in seconds not milliseconds.

The death method simply calls `stopEverything` and then uses the `SoundEngine` to play a death sound.

Now code the `endGame` method and then we will discuss it.

```

private void endGame() {

    stopEverything();
    int totalTime =
        ((mCoinsAvailable - coinsCollected)
         * 10)
        + getCurrentTime();

    switch (currentLevel) {

        case "underground":
            if (totalTime < mFastestUnderground) {
                mFastestUnderground = totalTime;
                // Save new time

                editor.putInt("fastest_underground_time",
                             mFastestUnderground);

                editor.commit();
            }
            break;
        case "city":

```

```
        if (totalTime < mFastestCity) {
            mFastestCity = totalTime;
            // Save new time
            editor.putInt("fastest_city_time",
                mFastestCity);

            editor.commit();
        }
        break;
    case "mountains":
        if (totalTime < mFastestMountains) {
            mFastestMountains = totalTime;
            // Save new time
            editor.putInt("fastest_mountains_time",
                mFastestMountains);

            editor.commit();
        }
        break;
    }
}
```

The first line of code in `endGame` calls `stopEverything` so the game engine will halt updating and drawing.

Next, a local variable, `totalTime` is declared and initialized. The total time (as you might remember from *Chapter 22: Platform Game: Bob was in a hurry section*) is calculated by the total time the player took added to a penalty for each coin that the player failed to collect.

Next in the `endGame` method we enter a switch block where the condition is the `currentLevel` String. The first case is when the underground level has been played. The code uses an `if` statement to check if `totalTime` is less than `mFastestUnderground`. If it is then the player has a new fastest time. The `editor.putInt` and `editor.commit` methods of the `SharedPreferences.Editor` instance are then used to save the new record for posterity.

The next two case blocks do the same thing for the city then the mountains levels.

Now code the final methods for the `GameState` class.

```
void stopEverything() { // Except the thread
    mPaused = true;
    mGameOver = true;
    mDrawing = false;
}
```

```
private void startEverything() {
    mPaused = false;
    mGameOver = false;
    mDrawing = true;
}

void stopThread() {
    mThreadRunning = false;
}

boolean getThreadRunning() {
    return mThreadRunning;
}

void startThread() {
    mThreadRunning = true;
}

boolean getDrawing() {
    return mDrawing;
}

boolean getPaused() {
    return mPaused;
}

boolean getGameOver() {
    return mGameOver;
}
```

The final getters and setters control when the game engine does certain tasks like start/stop the thread and call update and/or draw. Familiarize yourself with their names and which members they interact with.

Let's make some noise.

Code the SoundEngine

Fortunately, the `SoundEngine` class holds no surprises. This is essentially the exact same class as we coded in the Scrolling Shooter project. The only exceptions are that we load different sound effects and the methods which play the sound effects have different names and play different sounds. We have also made it a `Singleton` to avoid the long parameter lists that we had in the previous project when we passed a `SoundEngine` reference into so many other classes/methods.

Add the following code for the member variables and the getInstance method.

```
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 static SoundPool mSP;
    private static int mJump_ID = -1;
    private static int mReach_Objective_ID = -1;
    private static int mCoin_Pickup_ID = -1;
    private static int mPlayer_Burn_ID = -1;

    private static SoundEngine ourInstance;

    public static SoundEngine getInstance(Context context) {
        ourInstance = new SoundEngine(context);
        return ourInstance;
    }
}
```

The getInstance method provides access to the full functionality of the class as well as calling the constructor method to initialize all the sound effects into a SoundPool. Add the constructor method.

```
public 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();
    } else {
        mSP = new SoundPool(5, AudioManager.STREAM_MUSIC, 0);
    }
    try {
        AssetManager assetManager = c.getAssets();
        AssetFileDescriptor descriptor;

        // Prepare the sounds in memory
        descriptor = assetManager.openFd("jump.ogg");
        mJump_ID = mSP.load(descriptor, 0);

        descriptor = assetManager.openFd("reach_objective.ogg");
        mReach_Objective_ID = mSP.load(descriptor, 0);

        descriptor = assetManager.openFd("coin_pickup.ogg");
        mCoin_Pickup_ID = mSP.load(descriptor, 0);

        descriptor = assetManager.openFd("player_burn.ogg");
        mPlayer_Burn_ID = mSP.load(descriptor, 0);

    } catch (IOException e) {
        // Error
    }
}

```

Now the sound files are loaded in to memory ready to play. Add the methods that play the sounds, as follows.

```

public static void playJump(){
    mSP.play(mJump_ID,1, 1, 0, 0, 1);
}

public static void playReachObjective(){
    mSP.play(mReach_Objective_ID,1, 1, 0, 0, 1);
}

public static void playCoinPickup(){
    mSP.play(mCoin_Pickup_ID,1, 1, 0, 0, 1);
}

public static void playPlayerBurn(){
    mSP.play(mPlayer_Burn_ID,1, 1, 0, 0, 1);
}

```


These methods are `public` and `static` so are accessible from anywhere without an instance of `SoundEngine`.

Any class can now play any of the sound effects.

Coding the physics engine (without collision)

The `PhysicsEngine` class is responsible, first, for updating all the game objects and secondly for detecting and responding to collisions. This next code handles updating the game objects in the `update` method and we will code an empty `detectCollisions` method ready to add more code in Chapter 25.

Add the `PhysicsEngine` class and then we will discuss the code.

```
import android.graphics.PointF;
import android.graphics.RectF;
import java.util.ArrayList;

class PhysicsEngine {

    void update(long fps,
                ArrayList<GameObject> objects,
                GameState gs) {

        for (GameObject object : objects) {
            object.update(fps,
                          objects.get(LevelManager.PLAYER_INDEX)
                              .getTransform());
        }

        detectCollisions(gs, objects);
    }

    private void detectCollisions(GameState gs,
                                  ArrayList<GameObject> objects) {
        // More code here soon
    }
}
```

The `update` method receives the current frames per second, an `ArrayList` full of all the `GameObject` references and a reference to the current `GameState`. It then loops through all the `GameObject` instances calling each and every `update` method after first checking that the `GameObject` is active. Finally, the `detectCollisions` method is called although for now we have left this method empty.

Coding a Renderer

As with many of the classes in this chapter, `Renderer` will be very similar to the earlier project, so we can zip through it and move on.

Create a new class called `Renderer` then add the following members and constructor.

```
class Renderer {
    private Canvas mCanvas;
    private SurfaceHolder mSurfaceHolder;
    private Paint mPaint;

    // Here is our new camera
    private Camera mCamera;

    Renderer(SurfaceView sh, Point screenSize){
        mSurfaceHolder = sh.getHolder();
        mPaint = new Paint();

        // Initialize the camera
        mCamera = new Camera(screenSize.x, screenSize.y);
    }
}
```

The `Renderer` has the `Canvas`, `SurfaceHolder` and `Paint` instances as we have come to expect. It also has a `Camera` instance called `mCamera`. We at last get to code the `Camera` class when we are done with `Renderer`.

In the constructor the `SurfaceHolder`, `Paint` and `Camera` instances are initialized.

Now add the `getPixelsPerMetre` and `draw` methods.

```
int getPixelsPerMetre(){
    return mCamera.getPixelsPerMetre();
}

void draw(ArrayList<GameObject> objects,
          GameState gs,
          HUD hud) {

    if (mSurfaceHolder.getSurface().isValid()) {
        mCanvas = mSurfaceHolder.lockCanvas();
        mCanvas.drawColor(Color.argb(255, 0, 0, 0));

        if(gs.getDrawing()) {
            // Set the player as the center of the camera

```

```
        mCamera.setWorldCentre(
            objects.get(LevelManager
                .PLAYER_INDEX)
                .getTransform().getLocation());

        for (GameObject object : objects) {
            if (object.checkActive()) {
                object.draw(mCanvas, mPaint,
                    mCamera);
            }
        }

        hud.draw(mCanvas, mPaint, gs);

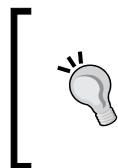
        mSurfaceHolder.unlockCanvasAndPost(mCanvas);
    }
}
```

The `getPixelsPerMetre` method uses the instance of the `Camera` class to return the number of pixels that represent a virtual meter in the game world. We will code the `Camera` class including this method next.

The code in the `draw` method prepares the surface, checks it is currently OK to draw then sets the camera to whatever the player's location is using the `setWorldCentre` method. We will code the `Camera` class including this method next. Then the code loops through all the objects checking which ones are active and drawing them. Then the HUD is drawn and the now familiar `unlockCanvasAndPost` method is called.

Coding the camera

This class is completely new to this project. The key to understanding it is to remember that all our game objects have sizes and positions in virtual meters. All the movement and collision detection will be done using these virtual meters. Here is the key-key point:



These virtual meters currently bare no relation to the pixels on the screen. The `Camera` class will know the screen's resolution and translate these virtual meters into pixel coordinates. Not all game objects will be on the screen every frame. In fact, most game objects will not be visible, most frames.

Add a new class called `Camera` and add the following member variables and the constructor.

```
import android.graphics.PointF;
import android.graphics.Rect;

class Camera {

    private PointF mCurrentCameraWorldCentre;
    private Rect mConvertedRect;
    private int mPixelsPerMetre;
    private int mScreenCentreX;
    private int mScreenCentreY;

    Camera(int screenXResolution, int screenYResolution){
        // Locate the centre of the screen
        mScreenCentreX = screenXResolution / 2;
        mScreenCentreY = screenYResolution / 2;

        // How many metres of world space does
        // the screen width show
        // Change this value to zoom in and out
        final int pixelsPerMetreToResolutionRatio = 48;
        mPixelsPerMetre = screenXResolution /
            pixelsPerMetreToResolutionRatio;

        mConvertedRect = new Rect();
        mCurrentCameraWorldCentre = new PointF();
    }
}
```

There are five, member variables as follows:

- The `PointF mCurrentCameraWorldCentre` instance will hold the central position (in virtual meters) the camera is centered on. This will be updated each frame and is the same as the position of the player.
- The `Rect mConvertedRect` instance will hold the four converted points that have been translated from virtual meters to pixel coordinates. This is the `Rect` that will be used by all the graphics-related component classes in their draw methods when they call `drawBitmap`. Look back to the `InanimateBlockGraphicsComponent` section if you want to confirm this.
- The `int mPixelsPerMetre` member is the number of pixels on the screen that represent one virtual metre.
- The `int mScreenCentreX` member is the pixel coordinate of the central pixel, horizontally on the screen.
- The `int mScreenCentreY` member is the pixel coordinate of the central pixel, horizontally on the screen.

It is the way we calculate and combine these five values that makes the class do its magic.

Next add the following getters and setters to the class.

```
int getPixelsPerMetreY() {
    return mPixelsPerMetre;
}

int getyCentre() {
    return mScreenCentreY;
}

float getCameraWorldCentreY() {
    return mCurrentCameraWorldCentre.y;
}

// Set the camera to the player. Called each frame
void setWorldCentre(PointF worldCentre) {
    mCurrentCameraWorldCentre.x = worldCentre.x;
    mCurrentCameraWorldCentre.y = worldCentre.y;
}

int getPixelsPerMetre() {
    return mPixelsPerMetre;
}
```

Four out of five of those methods simply return a value/reference to one of the members. We will see where they are used as we proceed. The `setWorldCentre` method however is more interesting. It receives a `PointF` reference as a parameter. The `setWorldCentre` method is called after the `update` method but before the `draw` method by `GameEngine` on every frame. The `PointF` that is passed in is the position of Bob in the game world (in virtual metres).

This next method is the meat of the whole class and it uses all the members and the position of Bob to translate virtual metres into pixels. Notice that it receives the object's horizontal and vertical positions (in virtual metres) as well as the objects width and height (also in virtual metres). Also notice it returns a `Rect` so that each game object's graphics-related component class can use it in the call to `drawBitmap`. Add the `worldToScreen` method and then we will examine its body.

```
// Return a Rect of the screen coordinates
// relative to a world location
Rect worldToScreen(float objectX,
                  float objectY,
                  float objectWidth,
                  float objectHeight) {
```

```

int left = (int) (mScreenCentreX -
                  ((mCurrentCameraWorldCentre.x - objectX)
                   * mPixelsPerMetre));

int top = (int) (mScreenCentreY -
                 ((mCurrentCameraWorldCentre.y - objectY)
                  * mPixelsPerMetre));

int right = (int) (left + (objectWidth
                           * mPixelsPerMetre));

int bottom = (int) (top + (objectHeight
                           * mPixelsPerMetre));

mConvertedRect.set(left, top, right, bottom);

return mConvertedRect;
}

```

Let's go through the body a line at a time. The first line of code declares and initializes a local `int` variable called `left`.

```

int left = (int) (mScreenCentreX -
                  ((mCurrentCameraWorldCentre.x - objectX)
                   * mPixelsPerMetre));

```

The initialization code uses a cast, `(int)`, to convert the `float` result to `int`. The calculation subtracts the objects horizontal position from the current focus of the camera which in turn is subtracted from the central horizontal pixel position. By multiplying the result by `mPixelsPerMetre` the world (virtual meter) positions are now a pixel coordinate. Note that at this stage we have only calculated the horizontal pixel position of one of four points.

This next line of code does the same thing except using the vertical values of the screen center, world center and object position.

```

int top = (int) (mScreenCentreY -
                 ((mCurrentCameraWorldCentre.y - objectY)
                  * mPixelsPerMetre));

```

The next two lines of code use `left` then `top` in conjunction with the width and height (respectively) to calculate the right and bottom pixel coordinates (respectively).

```

int right = (int) (left + (objectWidth
                           * mPixelsPerMetre));

int bottom = (int) (top + (objectHeight
                           * mPixelsPerMetre));

```

Let's invent some hypothetical positions for the player and an example object so we can see those calculations in action.

Suppose that the player is at (top left corner) the world location of 120 virtual meters horizontally and 90 meters vertically. The player is 1 meter wide and 2 meters tall. There is a platform at the world location of 116 meters horizontally and 89 meters vertically. The platform is 1 meter by 1 meter in size. If our calculations are to do their job, then they need to convert these sizes and positions into screen coordinates where the platform is to the left of the player by four times as much as it is above the player. This exercise will assume the screen is 1920 pixels wide by 1080 pixels high. First let's calculate the player's pixel coordinates.

Testing the formulas with hypothetical coordinates

The formula for the player's `left` (in English not in code) is as follows:



Remember that the world camera center coordinates are the same as the player's top-left coordinates and is updated each frame after `update` but before `draw` is called.

screen center x - ((camera world center x - player world center x) * number of pixels per meter)

$$= 960 - ((120 - 120) * 48)$$

$$= 960 - (0 * 48)$$

$$= 960$$

We can see that the left of the player is drawn in the horizontal center of the screen, 960 pixels.

Now for the `top` variable using the player's world coordinates.

screen center y - ((camera world center y - player world center y) * number of pixels per meter)

$$= 540 - ((90 - 90) * 48)$$

$$= 540 - (0 * 48)$$

$$= 540$$

We can now see that the player's top-left coordinates on the screen is 960, 540. This is the center of a 1920 x 1080 screen. Wherever in the game world the player moves, because `mCameraCurrentWorldCentre` is updated each frame, the player stays in the center of the screen.

Let's look quickly at how the right and bottom values are calculated and then we can run the hypothetical platform coordinates through the same formulas.

Here is the formula for the `right` variable.

$$\begin{aligned} &\text{Left pixel coordinate} + (\text{width in meters} * \text{pixels per meter}) \\ &= 960 + (1 * 48) \\ &= 1048 \end{aligned}$$

The `bottom` calculation uses the following formula:

$$\begin{aligned} &\text{Top pixel coordinate} + (\text{height in meters} * \text{pixels per meter}) \\ &= 540 + (2 * 48) \\ &= 636 \end{aligned}$$

The rectangle representing where Bob will be drawn will be at the following pixel coordinates:

Left: 960, Top: 540, Bottom: 1048, Right: 636.

The formula for the block's `left` (in English not in code) is as follows:

$$\begin{aligned} &\text{screen center x} - ((\text{camera world center x} - \text{block world center x}) * \text{number of pixels per meter}) \\ &= 960 - ((120 - 116) * 48) \\ &= 960 - (4 * 48) \\ &= 768 \end{aligned}$$

We can see that the left of the block is drawn in the horizontal center of the screen, 960 pixels.

Now for the `top` variable using the block's world coordinates.

screen center y - ((camera world center y - block world center y) * number of pixels per meter)

$$= 540 - ((90 - 89) * 48)$$

$$= 540 - (1 * 48)$$

$$= 492$$

Let's look quickly at how the right and bottom values are calculated. Here is the formula for the right variable.

Left pixel coordinate + (width in meters * pixels per meter)

$$= 768 + (1 * 48)$$

$$= 816$$

The bottom calculation uses the following formula:

Top pixel coordinate + (height in meters * pixels per meter)

$$= 492 + (1 * 48)$$

$$= 540$$

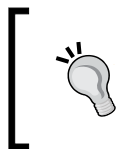
The rectangle representing where the block will be drawn will be at the following pixel coordinates:

Left: 960, Top: 540, Bottom: 1048, Right: 636.

Look closely at the results for converting the player's and the other object's world coordinates and you can see that they are within the scope of being able to accept them as correct. When we see hundreds of objects all neatly drawn in exactly the right place it will be certain the formulas work.

Finally, `mConvertedRect` is initialized with the four values using the `set` method and then returned to the calling code.

```
mConvertedRect.set(left, top, right, bottom);  
return mConvertedRect;
```



Try making up some more hypothetical object positions and calculating their pixel positions. Try some values that are some distance away (perhaps 50 metres) and notice how they are at a position that will not be seen on screen.



Now we have a system for converting all the floating-point world positions into integer pixel positions we can turn our attention to the HUD class.

Coding the Hud

The HUD in this game is no more complex than the previous game. We will define some `Rect` instances to draw the controls on the screen, we will rely on `GameState` to provide the time and fastest times for each level and we will make the button `Rect` `ArrayList` available so that `GameEngine` can pass them to our two classes that require them to handle the player's input.

Get started by adding a new class called `HUD` and add the following members and constructor method.

```
import android.content.Context;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
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 Bitmap mMenuBitmap;

    private int mTextFormatting;
    private int mScreenHeight;
    private int mScreenWidth;
    final float ONE_THIRD = .33f;
    final float TWO_THIRDS = .66f;

    private ArrayList<Rect> mControls;

    static int LEFT = 0;
    static int RIGHT = 1;
    static int JUMP = 2;

    HUD(Context context, Point size){
        mScreenHeight = size.y;
        mScreenWidth = size.x;
    }
}
```

```
mTextFormatting = size.x / 25;

prepareControls();

// Create and scale the bitmaps
mMenuBitmap = BitmapFactory
    .decodeResource(context.getResources(),
        R.drawable.menu);

mMenuBitmap = Bitmap
    .createScaledBitmap(mMenuBitmap,
        size.x, size.y, false);
    }
}
```

The class starts off with five members that we will use to control position and formatting of the parts of the HUD. Next there is an `ArrayList` for our `Rect` buttons and next there is the static variables `LEFT`, `RIGHT` and `JUMP` which the `UIController` (that we code next) and `PlayerInputComponent` (that we code next chapter) can use to identify what the player is trying to do.

In the constructor we initialize some of our formatting variables using the passed in screen resolution, call the `prepareControls` method and load and scale the `Bitmap` that is used for the menu background.

Now we can code the `prepareControls` method that we just called.

```
private void prepareControls(){
    int buttonWidth = mScreenWidth / 14;
    int buttonHeight = mScreenHeight / 12;
    int buttonPadding = mScreenWidth / 90;

    Rect left = new Rect(
        buttonPadding,
        mScreenHeight - buttonHeight - buttonPadding,
        buttonWidth + buttonPadding,
        mScreenHeight - buttonPadding);

    Rect right = new Rect(
        (buttonPadding * 2) + buttonWidth,
        mScreenHeight - buttonHeight - buttonPadding,
        (buttonPadding * 2) + (buttonWidth * 2),
```

```

        mScreenHeight - buttonPadding);

    Rect jump = new Rect(mScreenWidth - buttonPadding
        - buttonWidth,
        mScreenHeight - buttonHeight - buttonPadding,
        mScreenWidth - buttonPadding,
        mScreenHeight - buttonPadding);

    mControls = new ArrayList<>();
    mControls.add(LEFT, left);
    mControls.add(RIGHT, right);
    mControls.add(JUMP, jump);
}

```

In the previous code we initialize our remaining formatting members relative to the screen resolution in pixels. We then use them to position our three `Rect` objects that represent the buttons and then add them to the `mControls` `ArrayList`.

Next add the draw method which just like the HUD in the previous project will be called each frame of the game to draw the HUD. Notice the usual suspects are passed in as parameters to enable the method to do its job.

```

void draw(Canvas c, Paint p, GameState gs){

    if(gs.getGameOver()){

        // Draw the mMenuBitmap screen
        c.drawBitmap(mMenuBitmap, 0,0, p);

        // draw a rectangle to highlight the text
        p.setColor(Color.argb (100, 26, 128, 182));
        c.drawRect(0,0, mScreenWidth,
            mTextFormatting * 4, p);

        // Draw the level names
        p.setColor(Color.argb(255, 255, 255, 255));
        p.setTextSize(mTextFormatting);
        c.drawText("Underground",
            mTextFormatting,
            mTextFormatting * 2,
            p);

        c.drawText("Mountains",
            mScreenWidth * ONE_THIRD
            + (mTextFormatting),

```

```
        mTextFormatting * 2,
        p);

    c.drawText("City",
        mScreenWidth * TWO_THIRDS
        + (mTextFormatting),
        mTextFormatting * 2,
        p);

    // Draw the fastest times
    p.setTextSize(mTextFormatting/1.8f);

    c.drawText("BEST:" + gs.getFastestUnderground()
        + " seconds",
        mTextFormatting,
        mTextFormatting*3,
        p);

    c.drawText("BEST:" + gs.getFastestMountains()
        + " seconds", mScreenWidth * ONE_THIRD
        + mTextFormatting,
        mTextFormatting * 3,
        p);

    c.drawText("BEST:" + gs.getFastestCity()
        + " seconds",
        mScreenWidth * TWO_THIRDS + mTextFormatting,
        mTextFormatting * 3,
        p);

    // draw a rectangle to highlight the large text
    p.setColor(Color.argb(100, 26, 128, 182));
    c.drawRect(0, mScreenHeight - mTextFormatting * 2,
        mScreenWidth,
        mScreenHeight,
        p);

    p.setColor(Color.argb(255, 255, 255, 255));
    p.setTextSize(mTextFormatting * 1.5f);
    c.drawText("DOUBLE TAP A LEVEL TO PLAY",
        ONE_THIRD + mTextFormatting * 2,
        mScreenHeight - mTextFormatting/2,
        p);
}
// else block follows next

}
```

The method is long and might seem complicated at first glance but there is nothing we haven't seen before. There is one thing to note, however. All the code is wrapped in an `if` block with a condition of `gs.getGameOver`. So, all the code we just added runs when the game is over. We will add the `else` block which follows this `if` block in a moment.

The code inside the `if` block draws the background, level names and fastest times as well as the message to tell the player how to start the game. Clearly, we don't want these things on the screen while the game is being played.

Still inside the `draw` method add the `else` block that follows the `if` block which will execute while the game is being played.

```
// else block follows next
else {
    // draw a rectangle to highlight the text
    p.setColor(Color.argb(100, 0, 0, 0));
    c.drawRect(0,0, mScreenWidth,
               mTextFormatting,
               p);

    // Draw the HUD text
    p.setTextSize(mTextFormatting/1.5f);
    p.setColor(Color.argb(255, 255, 255, 255));
    c.drawText("Time:" + gs.getCurrentTime()
               + "+" + gs.getCoinsRemaining() * 10,
               mTextFormatting / 4,
               mTextFormatting / 1.5f,
               p);

    drawControls(c, p);
}
```

In the `else` block we draw a transparent rectangle across the top of the screen which has the effect of highlighting the text that is drawn on top of it. Then we draw the current time. The slightly convoluted formula (`gs.getCoinsRemaining() * 10`) has the effect of calculating (and displaying) the time penalty based on how many coins the player still needs to collect. The final line of code in the `draw` method (but still inside the `else` block) calls the `drawControls` method. This is separated out purely to stop the method getting any longer than it already is.

Here are the final two methods of the HUD class. Add the `drawControls` and `getControls` methods.

```
private void drawControls(Canvas c, Paint p){
    p.setColor(Color.argb(100,255,255,255));

    for(Rect r : mControls){
        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 mControls;
}
```

The `drawControls` method loops through the `mControls` `ArrayList` and draws each button as a transparent rectangle. The `getControls` method simply returns a reference to `mControls`. `GameEngine` will use this method to pass `mControls` to the other classes that need it.

Coding the `UIController` class

This class will have the same purpose as the class of the same name did in the previous project. It will also look very similar too.

Add a new class called `UIController` and add the member variables, constructor and `addObserver` method.

```
import android.graphics.Point;
import android.graphics.Rect;
import android.view.MotionEvent;

import java.util.ArrayList;

class UIController implements InputObserver {

    private float mThird;

    private boolean initialPress = false;
```

```

    UIController(GameEngineBroadcaster b, Point size) {
        // Add as an observer
        addObserver(b);

        mThird = size.x / 3;
    }

    void addObserver(GameEngineBroadcaster b) {
        b.addObserver(this);
    }
}

```

The float `mThird` variable will help us to divide the screen up vertically into thirds. The player will then be able to tap a portion of the screen to choose the level that they want to play. The `initialPress` Boolean is used in a workaround to avoid a bug/glitch. Sometimes when the game ends the menu will immediately register the player's touch causing a new level to start instantly rather than allowing them to view the menu screen at their leisure. By ignoring the very first touch we avoid this problem.


 This is a hack and not a good solution to go into a finished game. Unfortunately, the solution would require at least another chapter and I am running out of space. In chapter 26 I suggest some resources to continue your learning. Be sure to look into the **State** pattern to improve all the projects and as a solution for this hack.

In the constructor we call the `addObserver` method and initialize the `mThird` variable to a third of the screen's width. In the `addObserver` method the code uses the `GameEngineBroadcaster` reference to add an observer to `GameEngine`.

We need to add an observer each time a new game is started, and the game objects are rebuilt because the observer list is cleared. The `addObserver` method allows us to re-add an observer rather than just add it in the constructor.

Now add the `handleInput` method which receives the `MotionEvent`, the `GameState` and the `ArrayList` which contains the button positions.

```

@Override
public void handleInput(MotionEvent event,
    GameState gameState,
    ArrayList<Rect> buttons) {

    int i = event.getActionIndex();
    int x = (int) event.getX(i);

```



```
int eventType = event.getAction()
    & MotionEvent.ACTION_MASK;

if (eventType == MotionEvent.ACTION_UP ||
    eventType == MotionEvent.ACTION_POINTER_UP) {

    // If game is over start a new game
    if (gameState.getGameOver() && initialPress) {

        if (x < mThird) {
            gameState.setCurrentLevel("underground");
            gameState.startNewGame();
        } else if (x >= mThird && x < mThird * 2) {
            gameState.setCurrentLevel("mountains");
            gameState.startNewGame();
        } else if (x >= mThird * 2) {
            gameState.setCurrentLevel("city");
            gameState.startNewGame();
        }
    }
    initialPress = !initialPress;
}
}
```

As we did in the previous project we handle input by accessing the event at the index which triggered the current action. There is an if - else if - else if structure where the code detects which level was pressed. These are the only touches that need to be handled. The buttons will be handled by the `PlayerController` component class. After an `ACTION_UP` has been detected, notice also that the rest of the code is wrapped in an if condition that tests whether the game is over and `initialPress` is false. Outside this structure all `ACTION_UP` events will switch the value of `initialPress`. This means that the last stray touch of game-play won't dismiss the menu screen as soon as it starts.

Code the Activity

This class is just like the one we have been coding for virtually the entire book. Take a look at the code and then add this class to your project.

```
import android.app.Activity;
import android.graphics.Point;
import android.os.Bundle;
import android.view.Display;
```

```
public class GameActivity extends Activity {

    GameEngine mGameEngine;

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

        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();
    }
}
```

Everything is very familiar except you can see that I have renamed the resume and pause methods of the GameEngine class to startThread and stopThread respectively.

Now we can code the GameEngine class (including startThread and stopThread) and we will then be one small step away from running our hard work from the last two chapters.

Code the GameEngine

Now we get to the class that glues all the others together. In the code that follows we will declare and use an instance of LevelManager that we will code immediately after GameEngine. All the rest of the code will at last be error free and nearly ready to execute.

Add the `GameEngine` class, its members and constructor as shown next.

```
import android.content.Context;
import android.graphics.Point;
import android.util.Log;
import android.view.MotionEvent;
import android.view.SurfaceView;

import java.util.concurrent.CopyOnWriteArrayList;

class GameEngine extends SurfaceView
    implements Runnable,
    GameEngineBroadcaster,
    EngineController {

    private Thread mThread = null;
    private long mFPS;

    private GameState mGameState;
    UIController mUIController;

    // This ArrayList can be accessed from either thread
    private CopyOnWriteArrayList<InputObserver>
        inputObservers = new CopyOnWriteArrayList();

    HUD mHUD;
    LevelManager mLevelManager;
    PhysicsEngine mPhysicsEngine;
    Renderer mRenderer;

    public GameEngine(Context context, Point size) {
        super(context);
        // Prepare the bitmap store and sound engine
        BitmapStore bs = BitmapStore.getInstance(context);
        SoundEngine se = SoundEngine.getInstance(context);

        // Initialize all the significant classes
        // that make the engine work
        mHUD = new HUD(context, size);
        mGameState = new GameState(this, context);
        mUIController = new UIController(this, size);
        mPhysicsEngine = new PhysicsEngine();
        mRenderer = new Renderer(this, size);
    }
}
```

```

        mLevelManager = new LevelManager(context,
            this, mRenderer.getPixelsPerMetre());
    }
}

```

There are a few new things happening in the code you just saw. First notice that we need to import the `CopyOnWriteArrayList` class. This is a version of the `ArrayList` class that works simultaneously on multiple threads without crashing the game. This structure will be used to hold the `InputObservers`. `CopyOnWriteArrayList` is slower than the regular `ArrayList` class but as it is holding just two items that are accessed infrequently (compared to the `GameObject` instances) it won't cause a performance issue. The `GameObject` instances will be held in a regular `ArrayList`.

Notice as usual we have the members such as `mThread` and `mFPS` for our main game loop and measuring the frames per second. We also declare instances for many of the classes we have been slaving over for the past two chapters.

In the constructor we initialize all the instances. Look at the way we initialize our two Singletons (`BitmapStore` and `SoundEngine`) using the `getInstance` method. The other initializations are just as we have seen before and should expect based on the classes we have been coding. The exception is the instance of `LevelManager`. Look at the parameters we pass in, a `Context`, `this`, and the number of pixels per meter (supplied by the `Renderer` class). We will see how we put all these things in to action shortly. Let's finish the `GameEngine` class.

Code the `startNewLevel` method and the `addObserver` method.

```

public void startNewLevel() {
    // Clear the bitmap store
    BitmapStore.clearStore();
    // Clear all the observers and add the UI observer back
    // When we call buildGameObjects the
    // player's observer will be added too
    inputObservers.clear();
    mUIController.addObserver(this);
    mLevelManager.setCurrentLevel(mGameState.getCurrentLevel());
    mLevelManager.buildGameObjects(mGameState);
}

// For the game engine broadcaster interface
public void addObserver(InputObserver o) {
    inputObservers.add(o);
}

```

First, the static method `Bitmap.clearStore` is used to make sure there are no `Bitmap` instances already in the store. Then, the `inputObservers` `ArrayList` is cleared. The reason we need to do this each new game/level attempt is because each call to `buildGameObjects` deletes any existing game objects and rebuilds new ones to suit the current level. This includes the player that will be composed of a `PlayerInputController` component class that will have registered as an input observer. Therefore, if we leave the old input observers in the `ArrayList` then there will be a reference to an object which doesn't exist. Calling it will cause an instant crash.

After we have cleared the observers for the reason just stated we then need to add the `UIController` instance's observer by calling `addObserver`. The `UIController` instance remains the same throughout the life of the application but we must add it each level of course because we have just cleared it.

To end the method, we call the `setCurrentLevel` and `buildGameObjects` methods of the soon-to-be-coded `LevelManager` class.

The `addObserver` method allows the `InputObservers` to register themselves and be added to the `inputObservers` `ArrayList`.

Next add the familiar `run` method.

```
@Override
public void run() {
    while (mGameState.getThreadRunning()) {

        long frameStartTime = System.currentTimeMillis();

        if (!mGameState.getPaused()) {
            mPhysicsEngine.update(mFPS,
                                  mLevelManager.getGameObjects(),
                                  mGameState);
        }

        mRenderer.draw(mLevelManager.getGameObjects(),
                       mGameState,
                       mHUD);

        long timeThisFrame = System.currentTimeMillis()
                               - frameStartTime;

        if (timeThisFrame >= 1) {
            final int MILLIS_IN_SECOND = 1000;
```

```

        mFPS = MILLIS_IN_SECOND / timeThisFrame;
    }
}

```

Most of this we have seen before, but this is what happens in run, step by step:

- The current time is stored in `frameStartTime`.
- If the game is not currently paused, then the `PhysicsEngine` instance's `update` method is called.
- The `Renderer` instance's `draw` method is called
- The amount of time that the previous two steps took is calculated and stored in `timeThisFrame`
- The current frame rate is calculated and stored in `mFPS` ready for use during the next frame of the game loop.

Add the `onTouchEvent` method along with the `stopThread` and `startThread` methods.

```

@Override
public boolean onTouchEvent(MotionEvent motionEvent) {
    for (InputObserver o : inputObservers) {
        o.handleInput(motionEvent,
                      mGameState,
                      mHUD.getControls());
    }
    return true;
}

public void stopThread() {
    mGameState.stopEverything();
    mGameState.stopThread();
    try {
        mThread.join();
    } catch (InterruptedException e) {
        Log.e("Exception",
             "stopThread()" + e.getMessage());
    }
}

public void startThread() {
    mGameState.startThread();
    mThread = new Thread(this);
    mThread.start();
}

```

The `onTouchEvent` method loops through all the registered observers calling their `handleInput` method.

The `stopThread` method stops the thread (and everything else) via the `GameState` instance.

The `startThread` method starts the thread using the `GameState` instance.

Now we can move on to the final class for this chapter.

Coding the LevelManager class

We are nearly there. Coding this class will leave us with an executable project!

Create a new class called `LevelManager` and add the following members and the constructor.

```
import android.content.Context;
import android.graphics.PointF;
import android.util.Log;

import com.gamecodeschool.platformer.GOSpec.*;
import com.gamecodeschool.platformer.Levels.*;

import java.util.ArrayList;

final class LevelManager {

    static int PLAYER_INDEX = 0;
    private ArrayList<GameObject> objects;
    private Level currentLevel;
    private GameObjectFactory factory;

    LevelManager(Context context,
                  GameEngine ge,
                  int pixelsPerMetre){

        objects = new ArrayList<>();
        factory = new GameObjectFactory(context,
                                         ge,
                                         pixelsPerMetre);
    }
}
```

First, be aware of the two highlighted `import` statements that will need checking/correcting to your full package name.

We can tell we are getting close because we have declared the `ArrayList` that will hold all the `GameObject` instances (objects) and an instance of `GameObjectFactory` that will build them all.

We also have an instance of `Level` called `currentLevel` ready to be filled up with all the alpha-numeric characters that represent our level designs. The other member, `PLAYER_INDEX` is static and will help us insure that we keep a check on the position of the player in the objects `ArrayList`. In the constructor, objects and factory are initialized.

Add the `setCurrentLevel` method.

```
void setCurrentLevel(String level){
    switch (level) {
        case "underground":
            currentLevel = new LevelUnderground();
            break;

        case "city":
            currentLevel = new LevelCity();
            break;

        case "mountains":
            currentLevel = new LevelMountains();
            break;
    }
}
```

This method allows the `UIController` class to change the current level by passing in a `String` which the code in the method body uses to create a new `Level` reference of one of the extended classes (`LevelUnderground`, `LevelCity` or `LevelMountains`).

The next code is long but not difficult. We glimpsed some of it (the comments) before in the earlier section *Create the levels*. Very similar to previous project it just adds a case to create the appropriate type of object based on the alpha-numeric character that is returned by the nested `for` loops which go through the level design a character at a time. Objects that aren't available yet are commented out for later convenience.

Add the `buildGameObjects` method.

```
void buildGameObjects(GameState gs){
    // Backgrounds 1, 2, 3(City, Underground, Mountain...)
    // p = Player
    // g = Grass tile
```



```
// o = Objective
// m = Movable platform
// b = Brick tile
// c = mine Cart
// s = Stone pile
// l = coaL
// n = coNcrete
// a = lAmpost
// r = scoRched tile
// w = snoW tile
// t = stalacTite
// i = stalagmIte
// d = Dead tree
// e = snowy trEe
// x = Collectable
// z = Fire
// y = invisible death_invisible

gs.resetCoins();
objects.clear();
ArrayList<String> levelToLoad = currentLevel.getTiles();

for(int row = 0; row < levelToLoad.size(); row++ )
{
    for(int column = 0;
        column < levelToLoad.get(row).length();
        column++){

        PointF coords = new PointF(column, row);

        switch (levelToLoad.get(row)
            .charAt(column)){

            case '1':
                //objects.add(factory.create(
                // new BackgroundCitySpec(),
                // coords));
                break;

            case '2':
                //objects.add(factory.create(
                // new BackgroundUndergroundSpec(),
                // coords));
                break;

            case '3':
                //objects.add(factory.create(
```

```
        // new BackgroundMountainSpec(),
        // coords));
        break;

    case 'p':
        //objects.add(factory.create(new
        // PlayerSpec(),
        // coords));
        // Remember the location of
        // the player
        //PLAYER_INDEX = objects.size()-1;
        break;

    case 'g':
        objects.add(factory.create(
            new GrassTileSpec(),
            coords));

        break;

    case 'o':
        objects.add(factory.create(
            new ObjectiveTileSpec(),
            coords));

        break;

    case 'm':
        //objects.add(factory.create(
        // new MoveablePlatformSpec(),
        // coords));
        break;

    case 'b':
        objects.add(factory.create(
            new BrickTileSpec(),
            coords));

        break;

    case 'c':
        objects.add(factory.create(
            new CartTileSpec(),
            coords));

        break;

    case 's':
        objects.add(factory.create(
            new StonePileTileSpec(),
            coords));
```

```
        break;

    case 'l':
        objects.add(factory.create(
            new CoalTileSpec(),
            coords));
        break;

    case 'n':
        objects.add(factory.create(
            new ConcreteTileSpec(),
            coords));
        break;

    case 'a':
        objects.add(factory.create(
            new LamppostTileSpec(),
            coords));
        break;

    case 'r':
        objects.add(factory.create(
            new ScorchedTileSpec(),
            coords));
        break;

    case 'w':
        objects.add(factory.create(
            new SnowTileSpec(),
            coords));
        break;

    case 't':
        objects.add(factory.create(
            new StalactiteTileSpec(),
            coords));
        break;

    case 'i':
        objects.add(factory.create(
            new StalagmiteTileSpec(),
            coords));
        break;

    case 'd':
        objects.add(factory.create(
            new DeadTreeTileSpec(),
```

```
        coords));  
        break;  
  
    case 'e':  
        objects.add(factory.create(  
            new SnowyTreeTileSpec(),  
            coords));  
        break;  
  
    case 'x':  
        objects.add(factory.create(  
            new CollectibleObjectSpec(),  
            coords));  
        gs.coinAddedToLevel();  
        break;  
  
    case 'z':  
        //objects.add(factory.create(  
        // new FireTileSpec(),  
        // coords));  
  
        break;  
  
    case 'y':  
        objects.add(factory.create(  
            new  
            InvisibleDeathTenByTenSpec(),  
            coords));  
  
        break;  
  
    case '.':  
        // Nothing to see here  
        break;  
  
    default:  
        Log.e("Unhandled item in level",  
            "row:"+row  
            + " column:"+column);  
        break;  
    }  
}  
  
}  
  
}
```

Each case starts with `objects.add...` and then calls the `create` method of the `GameObjectFactory` instance passing in the specification class of the new object to create and the coordinates at which to create it. The `GameObjectFactory` class returns the newly created object of the desired type and it is added to the `objects ArrayList`. All the specifications where we haven't yet coded all the component classes are commented out to avoid errors/crashes.

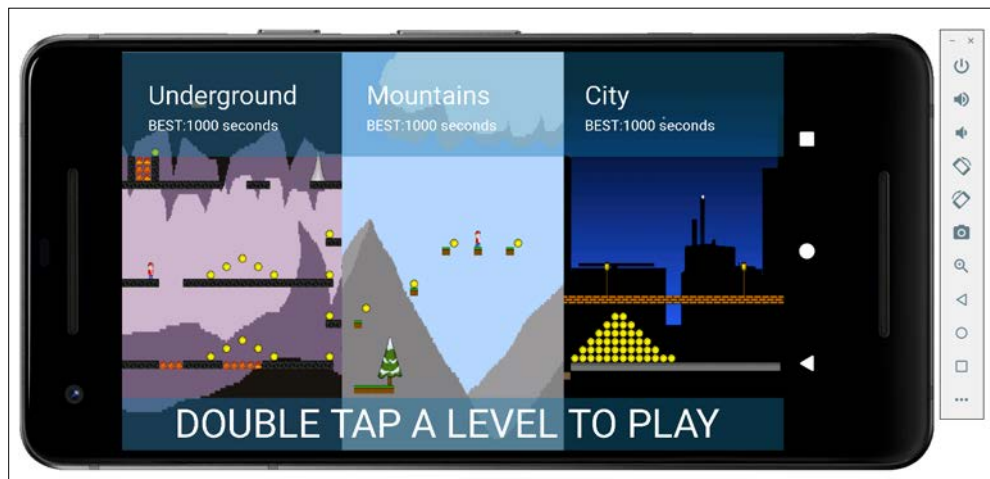
Add the final method to the `LevelManager` class.

```
ArrayList<GameObject> getGameObjects() {  
    return objects;  
}
```

This method returns a reference to the `ArrayList` that holds all the `GameObject` instances. You might remember this was called from the main game loop (`run` method) in the `GameEngine` class.

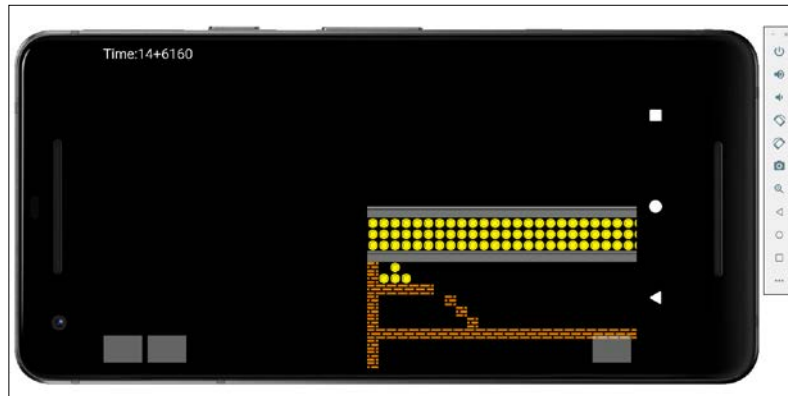
Running the game

If you execute the game, you will see the menu.

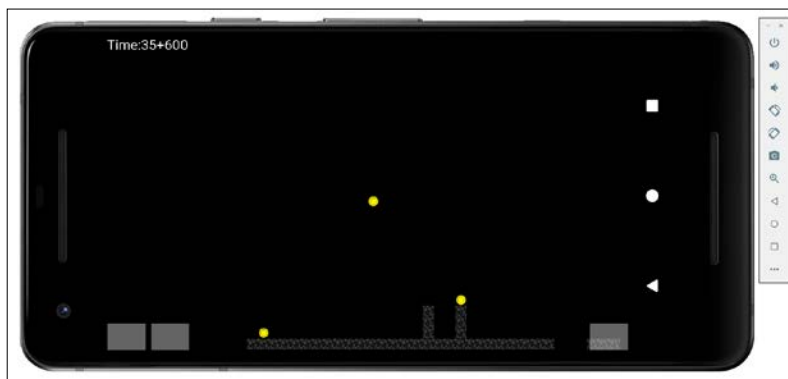


The menu allows you to start each of the three levels. But as there is currently no way to complete or fail a level you will need to quit the game and restart it if you want to see the progress on all three.

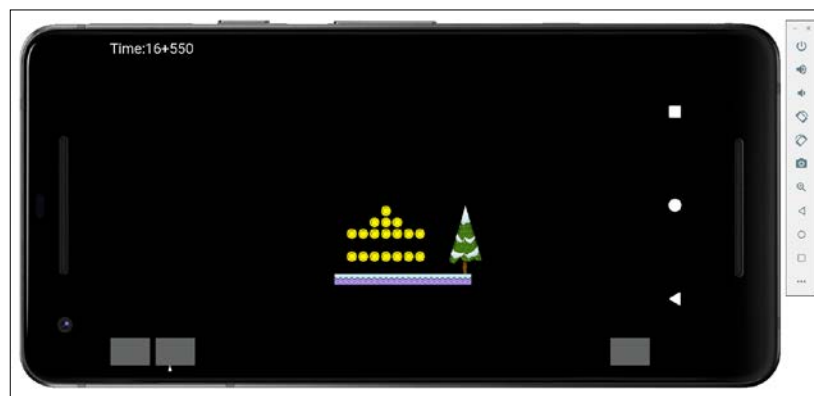
This is the City level so far.



This is the Underground level so far.



This is the Mountains level so far.



Note the camera is centred around 0,0 because `PLAYER_INDEX` equals zero and the first object in the `objects ArrayList` is at 0,0. You could count the characters in the level designs and then change `PLAYER_INDEX` in the `LevelManager` class to equal this number. Then the camera will focus on it and you can see more of each level when you run the game. When the player is added to the game the camera will centre around Bob.

Summary

That was a big chapter, but much was achieved. We were introduced to the Java `HashMap` class and the Singleton pattern which together enabled us to code the `BitmapStore` class to save memory by avoiding duplicate copies of `Bitmap` instances. We coded the component classes for all the platforms that Bob will walk on as well as the decorative component classes that will handle things like trees and stalagmites.

We coded a new `GameObjectFactory` that was like the previous project except it handles multiple different types of `Transform`. We saw and added the code for the level designs. They are represented by extending a parent `Level` class.

We also coded the slightly familiar `GameObject`, `GameState`, `SoundEngine`, `PhysicsEngine` and `Renderer` classes.

We learned about and coded the all-new `Camera` class which tracks the player's position, keeping him in the centre of all the action and translating floating point world coordinates to integer pixel coordinates.

Next, we coded the `HUD` and the `UIController` which allows the player to choose a level and draws buttons on the screen when the game is playing.

Finally, we added the main `Activity` class, `GameEngine` class and `LevelManager` which handles choosing and loading levels to play.

In the next chapter we will add a running, jumping and falling Bob character to the game along with a parallax scrolling background. Although we will need to wait until chapter 25 to implement the collision detection that makes him playable and interactive with the other game objects.

24

Sprite-sheet animations, Controllable Player and Parallax Scrolling Backgrounds

In this chapter we will look at how basic animation occurs by "flipping" through different images to create a moving effect. This is the same technique used to make a simple cartoon. Of course, we will then need to code a solution to achieve this simple animation in our game objects and will do so by coding an `Animator` class and adding one to any graphics-related component class that needs it. We will also implement another scrolling background but slightly differently to the scrolling background in the previous project because this time it will need to be positioned both horizontally and vertically, each frame, relative to the position of the camera.

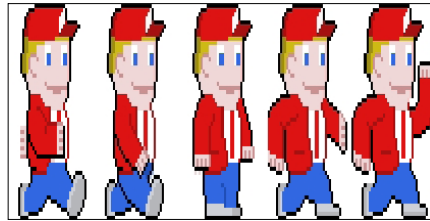
- Code the `Animator` class
- Code the component classes related to the player
- Code the component classes related to backgrounds
- Update the `LevelManager` and `GameObjectFactory` to handle the new game objects and their components.

Let's get started with the animations.

Animating sprite-sheets

Sprite-sheet animations work by quickly changing the image drawn. Exactly like a child may draw the frames of a stick-man moving on the corner of a book, and then quickly flicking through it to make it appear to move. Just like a cartoon.

The frames of Bob's animation are already contained within the `player.png` file shown again for convenience next.



All we need to do is loop through them one at a time while the player is moving.

This is quite straightforward to implement. We will make a simple animation class called `Animator` that handles the function of keeping time and returning the next frame of the sprite sheet when asked. We can then initialize a new `Animator` object for any `GameObject` that needs to be animated. In addition, when they are being drawn in the `draw` method, if the object is animated, we will handle it slightly differently.

In this section, we will also see how to use the variables of the `Transform` that track which way a game object is facing. It will enable us to reverse the sprite sheet depending on the way the player (or any future animated objects) is headed.

Let's start by making the `Animator` class.

Coding the Animator

Add the members and a constructor to a new class called `Animator`.

```
class Animator {
    private Rect mSourceRect;
    private int mFrameCount;
    private int mCurrentFrame;
    private long mFrameTicker;
    private int mFramePeriod;
    private int mFrameWidth;
    Animator(float frameHeight,
            float frameWidth,
```

```
        int frameCount,
        int pixelsPerMetre) {

    final int ANIM_FPS = 10;

    this.mCurrentFrame = 0;
    this.mFrameCount = frameCount;
    this.mFrameWidth = (int)frameWidth * pixelsPerMetre;

    frameHeight = frameHeight * pixelsPerMetre;

    mSourceRect = new Rect(0, 0,
        this.mFrameWidth,
        (int)frameHeight);

    mFramePeriod = 1000 / ANIM_FPS;
    mFrameTicker = 0L;
    }
}
```

Let's look at all those variables one at a time.

- The `mSourceRect` member is a `Rect` that will hold the four corners of the frame to be animated. Put another way- it will hold the pixel coordinates from within the sprite-sheet that represent the image from the sprite-sheet to be drawn next. How `mSourceRect` is initialized and updated with these values will become clear as we go ahead.
- The `mFrameCount` member is an `int` which will hold the number of frames of animation contained in the sprite-sheet. This will be important to divide the sprite-sheet up into frames. The number of frames, you might recall, is held in the specification class (`GOSpec`). It will be passed from specification to factory to graphics-related component class and finally to `Animator` to initialize `mFrameCount`.
- The `mCurrentFrame` variable is an `int` and you can probably guess that it will hold the number of the frame currently being drawn.
- The long `mFrameTicker` member will hold the amount of time that the current frame has been used for.
- The `int mFramePeriod` member holds a value that is the number of milliseconds a frame should be displayed for.
- The `int mFrameWidth` member will hold the width in pixels of a frame of animation. In this project all the frames will be of equal width.

At this point we have all the members necessary to calculate which frame to show and where within the sprite-sheet the current frame is. Let's look more closely at the constructor that we just coded to see how we initialize these members ready for use.

Here is the signature of the constructor again.

```
Animator(float frameHeight,  
         float frameWidth,  
         int frameCount,  
         int pixelsPerMetre)
```

Notice the values that are passed in as parameters. The height and width of the frame being passed in might be surprising because you would perhaps expect the `Animator` to have to work that out? But actually, the height and width of a frame of animation is the exact same value as the height and width of the game object and is therefore available from the specification class. The frame count is also straight from the specification class and is decided simply by counting the number of frames in the sprite-sheet. Flip back to the image of Bob and you can count there are five. The number of pixels-per-meter you might recall is passed from the camera into the factory and after, shared to the `Animator`. Now we have a lot of data to work with let's see what we do with it.

The next two lines of code in the constructor declares and initializes another local variable called `ANIM_FPS` which will represent how many frames of the sprite-sheet should be flipped through every second. We use 10 which is fairly arbitrary but works. Feel free to experiment with higher or lower values. The next line of code initializes `mCurrentFrame` to 0 ready to draw the first frame. Using 0 for the first frame as we will see shortly makes the calculations simpler than if we had used 1.

```
final int ANIM_FPS = 10;  
  
mCurrentFrame = 0;
```

Next in the constructor `mFrameCount` is initialized with the `frameCount` parameter and `mFrameWidth` is initialized by multiplying the `frameWidth` parameter by the `pixelsPerMetre` parameter. Remember `frameWidth` is a size in virtual meters so this calculation gives us the number of pixels from the sprite-sheet that holds a frame.

```
this.mFrameCount = frameCount;  
this.mFrameWidth = (int)frameWidth * pixelsPerMetre;
```

This next line of code uses the same formula to get the number of pixels in the height of a frame.

```
frameHeight = frameHeight * pixelsPerMetre;
```

And now we see how we initialize the pixel coordinates of the first frame of animation. Look at each of the arguments that initialize the `Rect`.

```
mSourceRect = new Rect(0, 0,
    mFrameWidth,
    (int)frameHeight);
```

The top-left of the sprite-sheet is 0, 0. And `mFrameWidth, frameHeight` completes the `Rect` exactly one frame's width across and one frame's width down.

The `mFramePeriod` member is initialized by dividing 1000 by the number of frames required each second. Now `mFramePeriod` holds the value of 100. So, when we code the next method we will see how we change the frame every 100 milliseconds. The `mFrameTicker` is set to 0 ready to start counting to 100.

```
mFramePeriod = 1000 / ANIM_FPS;
mFrameTicker = 0L;
```

Next add the `getCurrentFrame` method which completes the `Animator` class.

```
Rect getCurrentFrame(long time) {
    if (time > mFrameTicker + mFramePeriod) {
        mFrameTicker = time;
        mCurrentFrame++;
        if (mCurrentFrame >= mFrameCount) {
            mCurrentFrame = 0;
        }
    }

    // Update the left and right values of the source of
    // the next frame on the spritesheet
    this.mSourceRect.left = mCurrentFrame * mFrameWidth;
    this.mSourceRect.right = this.mSourceRect.left
        + mFrameWidth;

    return mSourceRect;
}
```

This method, as the name suggests is how we find out which region/coordinates form the sprite-sheet to draw. It returns a `Rect` which defines the region/coordinates. Remember it doesn't have access to the sprite-sheet itself that is in the `BitmapStore` class- it just knows the height, width, number of frames etc.

The graphics component class will set up an `Animator` by calling its constructor during the `initialize` method and then get the appropriate region by calling this method (`getCurrentFrame`) while obtaining the sprite-sheet itself from the `BitmapStore`. Let's analyse the code in the `getCurrentFrame` method.

First notice that the current time is passed in as a parameter. The `if` condition then uses the test `time > mFrameTicker + mFramePeriod` to determine if 100 milliseconds has passed. If it has the code inside the `if` block is executed.

Inside the `if` block the `mFrameTicker` is reset to the current time and the frame is incremented to the next one.

```
mFrameTicker = time;
mCurrentFrame++;
```

This nested `if` calculates if the last frame has already been used and if so moves the frame back to zero.

```
if (mCurrentFrame >= mFrameCount) {
    mCurrentFrame = 0;
}
```

The last two lines of code use the current values of the members to calculate the coordinates of the frame and load it into the `Rect`.

```
this.mSourceRect.left = mCurrentFrame * mFrameWidth;
this.mSourceRect.right = this.mSourceRect.left
    + mFrameWidth;
```

The left-hand coordinate is calculated with the current frame number multiplied by the width of a frame and the right-hand side is calculated by adding the width of a frame to the previous calculation. The top and bottom values never change.

Finally, a reference to the `Rect` is returned to the calling code (the `draw` method of the graphics-related component).

Now we have an `Animator` class ready to use we will code some more components to bring some more game objects to life.

Coding the player's components and transform

Since Bob is the most significant object that uses an Animator, let's code all the necessary classes to get the player game object working. There are three components, AnimatedGraphicsComponent, PlayerInputComponent and PlayerUpdateComponent.

Before we code the components, we will also need a more advanced version of the Transform class.

PlayerTransform

Create a new class called `PlayerTransform` and add the member variables along with the constructor method. Notice the class extends `Transform` so it will also have access to the methods and members of the simpler `Transform` class we coded in *Chapter 23: Coding the basic Transform*.

```
import android.graphics.PointF;
import android.graphics.RectF;

import java.util.ArrayList;

class PlayerTransform extends Transform {

    private ArrayList<RectF> mColliders;

    private final float TENTH = .1f;
    private final float HALF = .5f;
    private final float THIRD = .3f;
    private final float FIFTH = .2f;
    private final float FEET_PROTRUSION = 1.2f;

    private RectF mHeadRectF = new RectF();
    private RectF mRightRectF = new RectF();
    private RectF mFeetRectF = new RectF();
    private RectF mLeftRectF = new RectF();

    private boolean mJumpTriggered = false;
    private boolean mBumpedHeadTriggered = false;

    private boolean mGrounded;
```

```
PlayerTransform(float speed,
                float objectWidth,
                float objectHeight,
                PointF startingLocation) {

    super(speed, objectWidth,
          objectHeight,
          startingLocation);

    mColliders = new ArrayList<RectF>();
    // Load up the colliders ArrayList with
    // player specific colliders
    mColliders.add(mFeetRectF);
    mColliders.add(mHeadRectF);
    mColliders.add(mRightRectF);
    mColliders.add(mLeftRectF);
}
```

There are quite a few new members let's talk about them. The first is an `ArrayList` of `RectF` objects called `mColliders`. This will hold a whole bunch of different colliders for the player. It will be necessary to figure out where on the player a collision occurred and take different action accordingly. The `mCollidersArrayList` will hold all these colliders. The player will still have access to the usual collider via the parent `Transform` class and this will be used to see if a collision has occurred initially. If it has then `mColliders` will be used to see more specifically where a collision has occurred.

```
private ArrayList<RectF>mColliders;
```

These next five final float variables were arrived at through testing the game-play, specifically the collisions. They define values that we will use in calculations with the colliders when we are initializing them.

```
private final float TENTH = .1f;
private final float HALF = .5f;
private final float THIRD = .3f;
private final float FIFTH = .2f;
private final float FEET_PROTRUSION = 1.2f;
```

Here are the four extra colliders. One for the head, right, left and feet.

```
private RectF mHeadRectF = new RectF();
private RectF mRightRectF = new RectF();
private RectF mFeetRectF = new RectF();
private RectF mLeftRectF = new RectF();
```

In addition to the extra colliders we have some extra boolean values that will help us acknowledge that a specific collision event has occurred so other classes can respond to them. For instance, if the head collides with a platform, the `PhysicsEngine` can set `mBumpedHeadTriggered` to `true`. The update-related component class will then be able to detect this and respond accordingly- perhaps by ending a jump that was in progress.

```
private boolean mJumpTriggered = false;
private boolean mBumpedHeadTriggered = false;

private boolean mGrounded;
```

The constructor does a few things, let's look at them. The first thing the constructor does is to call the super-class constructor so that `Transform` can set itself up in the way we discussed in chapter 23.

```
super(speed, objectWidth,
      objectHeight,
      startingLocation);
```

Next the `mColliders` `ArrayList` is initialized and the four `RectF` instances are added to it.

```
mColliders = new ArrayList<RectF>();
// Load up the colliders ArrayList with
// player specific colliders
mColliders.add(mFeetRectF);
mColliders.add(mHeadRectF);
mColliders.add(mRightRectF);
mColliders.add(mLeftRectF);
```

Now add the `getColliders` method.

```
public ArrayList<RectF> getColliders() {
    updateColliders();
    return mColliders;
}
```


The `getColliders` method first calls the `updateColliders` method and then returns the `ArrayList` (to `PhysicsEngine`) so they can be used for collision detection. We don't update all the colliders every frame to save time. The collider on the regular transform detects a collision and that triggers the call to `getColliders` and only then the detailed colliders are updated.

Next add the `updateColliders` method.

```
public void updateColliders(){

    PointF location = getLocation();
    float objectHeight = getSize().y;
    float objectWidth = getSize().x;

    // Feet
    mColliders.get(0).left = location.x
        + (objectWidth * THIRD);

    mColliders.get(0).top = location.y
        + objectHeight - (objectHeight
        * TENTH);

    mColliders.get(0).right = location.x
        + objectWidth - (objectWidth
        * THIRD);

    mColliders.get(0).bottom = location.y
        + objectHeight + (objectHeight
        * FEET_PROTRUSION);

    // Head
    mColliders.get(1).left = location.x
        + ((objectWidth * THIRD));

    mColliders.get(1).top = location.y;
    mColliders.get(1).right = location.x
        + objectWidth - (objectWidth
        * THIRD);

    mColliders.get(1).bottom = location.y
        + (objectHeight * TENTH);
```

```
// Right
mColliders.get(2).left = location.x
                        + objectWidth - (objectWidth
                        * TENTH);

mColliders.get(2).top = location.y
                    + (objectHeight * THIRD);
mColliders.get(2).right = location.x
                    + objectWidth;

mColliders.get(2).bottom = location.y
                    + (objectHeight - (objectHeight
                    * HALF));

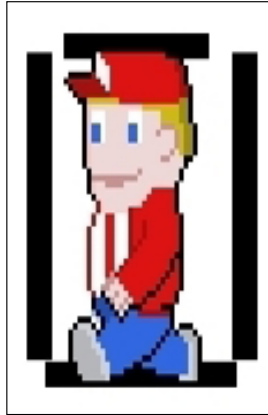
// Left
mColliders.get(3).left = location.x;
mColliders.get(3).top = location.y
                    + (objectHeight * FIFTH);
mColliders.get(3).right = location.x
                    + (objectWidth * TENTH);

mColliders.get(3).bottom = location.y
                    + (objectHeight - (objectHeight
                    * FIFTH));
}
```

The `updateColliders` method initially looks long and imposing but it is not that bad. Notice the comments indicating which part of the code updates which `RectF`- // Feet, // Head, etc.

At the top of the method the height, width and location are initialized into local variables and these local variables are used along with the `final float` member variables to initialize the four `RectF` instances. The formulas for calculating the colliders look tricky but they are just small rectangles on the head, right, left and feet of the rectangle which is the player.

This next image shows where these calculations approximately place these colliders.



The exact calculations were arrived at not through math but through trial and error while testing the game.

Now add this list of getters and setters to complete the `PlayerTransform` class.

```
// Called by handle input component to
// let us know a jump has been triggered
void triggerJump() {
    mJumpTriggered = true;
}

// Called by movement component to let transform
// know that movement component is aware
// jump was triggered
void handlingJump() {
    mJumpTriggered = false;
}

// Used by movement component to find
// out if jump has been triggered
boolean jumpTriggered() {
    return mJumpTriggered;
}
```

```
void setNotGrounded(){
    mGrounded=false;
}

void triggerBumpedHead() {
    mBumpedHeadTriggered = true;
}

void handlingBumpedHead() {
    mBumpedHeadTriggered = false;
}

boolean bumpedHead() {
    return mBumpedHeadTriggered;
}

void notGrounded() {
    mGrounded = false;
}

void grounded() {
    mGrounded = true;
}

boolean isGrounded() {
    return mGrounded;
}
```

Look through the methods you just added. There is nothing technical. Each of the methods either sets or provides access to one of the member variables. Through these methods the `PhysicsEngine` and the `PlayerUpdateComponent` will coordinate what has had a collision and how it is responded to.

Let's move on to the components.

AnimatedGraphicsComponent

Create a new class called `AnimatedGraphicsComponent` and add the member variables along with the constructor as shown next.

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
```

```
import android.graphics.PointF;
import android.graphics.Rect;

import com.gamecodeschool.platformer.GOSpec.GameObjectSpec;

class AnimatedGraphicsComponent implements GraphicsComponent {

    private String mBitmapName;
    private Animator mAnimator;
    private Rect mSectionToDraw;

    @Override
    public void initialize(Context context,
                          GameObjectSpec spec,
                          PointF objectSize,
                          int pixelsPerMetre) {

        // Initialize the animation
        mAnimator = new Animator(
            objectSize.y,
            objectSize.x,
            spec.getNumFrames(),
            pixelsPerMetre);

        // stretch the bitmap by the number of frames
        float totalWidth = objectSize.x *
            spec.getNumFrames();

        mBitmapName = spec.getBitmapName();
        BitmapStore.addBitmap(context,
            mBitmapName,
            new PointF(totalWidth, objectSize.y),
            pixelsPerMetre, true);

        // Get the first frame of animation
        mSectionToDraw = mAnimator.getCurrentFrame(
            System.currentTimeMillis());
    }
}
```

All the graphics-related component classes have a `String` to hold the name of a `Bitmap` but this one has two extra members. A `Rect` called `mSectionToDraw` which will be used to hold the coordinates of the current frame of animation and it also declares an instance of `Animator` called `mAnimator`.

Inside the `initialize` method `mAnimator` is initialized and the `Bitmap` is added to the store in the same way it was with the platforms and scenery in Chapter 22. The one subtle difference is that the final argument to the `addBitmap` call is `true` which will trigger `BitmapStore` to also store a reversed copy of the `Bitmap`.

The final line of code uses the `Animator` class's `getCurrentFrame` method to initialize the first frame of animation into `mSectionToDraw`.



Note the highlighted import that will cause an error if copy & pasting.

Add the required draw method.

```
@Override
// Updated to take a reference to a Camera
public void draw(Canvas canvas,
                  Paint paint,
                  Transform t,
                  Camera cam) {

    // Get the section of bitmap to draw
    // when an object is in motion
    // OR if it is a object with
    // zero speed(like a fire tile)
    if (t.headingRight() ||
        t.headingLeft() ||
        t.getSpeed() == 0) {

        // Player is moving so animate/change the frame
        mSectionToDraw = mAnimator.getCurrentFrame(
            System.currentTimeMillis());
    }

    // Where should the bitmap section be drawn?
    Rect screenCoordinates = cam.worldToScreen
        (t.getLocation().x,
         t.getLocation().y,
         t.getSize().x,
         t.getSize().y);
```

```
if (t.getFacingRight()) {
    canvas.drawBitmap(
        BitmapStore.getBitmap(mBitmapName),
        mSectionToDraw,
        screenCoordinates,
        paint);
} else
    canvas.drawBitmap(
        BitmapStore.getBitmapReversed(mBitmapName),
        mSectionToDraw,
        screenCoordinates,
        paint);
}
```

The first `if` statement in the `draw` method checks to see whether the player is heading either right or left. Notice the subtle difference between heading and facing. If the player is heading right or left, then the player is in motion so we want to let the `Animator` have the opportunity to update the current frame of animation, so we call `getCurrentFrame` inside the `if` statement. You might have also noticed a third part in the `if` statement condition `|| t.getSpeed == 0`. This will mean that static objects like the fire tile will constantly call for an updated frame even though they don't ever head in any direction.

Next the `draw` method uses the `Camera` class's `worldToScreen` method to determine the rectangle at which to draw the player.

Finally, an `if-else` block determines which way the player is facing to choose whether to use the regular `Bitmap` from the `BitmapStore` or the reversed version. Notice the call to `drawBitmap` uses `mSectionToDraw` and `screenCoordinates` to determine which portion of the `Bitmap` to use and where to draw it.

PlayerInputComponent

This is quite a long class but there is nothing we haven't already learned about. Let's break it in to chunks to code it.

Create a new class called `PlayerInputComponent`, add the two, member variables and the first two methods.

```
import java.util.ArrayList;

class PlayerInputComponent implements InputObserver {

    private Transform mPlayersTransform;
```

```

private PlayerTransform mPlayersPlayerTransform;

PlayerInputComponent(GameEngine ger) {
    ger.addObserver(this);
}

public void setTransform(Transform transform) {
    mPlayersTransform = transform;
    mPlayersPlayerTransform =
        (PlayerTransform) mPlayersTransform;
}
}

```

First notice that the class implements `InputObserver` so it can communicate with the `GameEngine` via its `ArrayList` of registered observers. There are two, member variables, one, of type `Transform` and one of the more specific type, `PlayerTransform`.

The first method is the constructor and all we need to do is call the `addObserver` method using the `GameEngine` reference that was passed as a parameter. Once we have implemented the `handleInput` method the class is then set up ready to receive data when the player touches the screen.

We then added the `setTransform` method. This method initializes both of the members. This is achieved in a straightforward manner with `mPlayersTransform` and by casting `(PlayerTransform)` for `mPlayersPlayerTransform`. Now we have these references we can notify all the different parts of the player's transform when specific screen touches are received.

The `handleInput` method is quite long so let's split it up. Add the outline of the `handleInput` method as shown next.

```

// Required method of InputObserver interface
// called from the onTouchEvent method
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);

    if(!gameState.getPaused()) {
        switch (event.getAction() & MotionEvent.ACTION_MASK) {
            // Case statements go here

```



```
    }  
}
```

The method starts by storing the pointer index and the horizontal and vertical locations of the touch in the `i`, `x` and `y` variables respectively.

The `if` statement checks that the game is not paused. If the game is paused this class is not interested in any input. Next there is a `switch` and the condition gets the type of action (`ACTION_UP` etc.). We will now handle all the case statements.

Add all the case statements along with the nested `if` and `else - if` blocks shown next.

```
// Case statements go here  
case MotionEvent.ACTION_UP:  
    if (buttons.get(HUD.LEFT).contains(x, y)  
        || buttons.get(HUD.RIGHT)  
        .contains(x, y)) {  
  
        // Player has released  
        // either left or right  
  
        mPlayersTransform.stopHorizontal();  
    }  
    break;  
  
case MotionEvent.ACTION_DOWN:  
    if (buttons.get(HUD.LEFT).contains(x, y)) {  
        // Player has pressed left  
        mPlayersTransform.headLeft();  
    } else if (buttons.get(HUD.RIGHT).contains(x, y)) {  
        // Player has pressed right  
        mPlayersTransform.headRight();  
    } else if (buttons.get(HUD.JUMP).contains(x, y)) {  
        // Player has released  
        // the jump button  
        mPlayersPlayerTransform.triggerJump();  
    }  
    break;
```

```

case MotionEvent.ACTION_POINTER_UP:
    if (buttons.get(HUD.LEFT).contains(x, y)
        || buttons.get(HUD.RIGHT)
            .contains(x, y)) {

        // Player has released either
        // up or down
        mPlayersTransform.stopHorizontal();
    }
    break;

case MotionEvent.ACTION_POINTER_DOWN:
    if (buttons.get(HUD.LEFT).contains(x, y)) {
        // Player has pressed left
        mPlayersTransform.headLeft();
    } else if (buttons.get(HUD.RIGHT).contains(x, y)) {
        // Player has pressed right
        mPlayersTransform.headRight();
    } else if (buttons.get(HUD.JUMP).contains(x, y)) {
        // Player has released
        // the jump button
        mPlayersPlayerTransform.triggerJump();
    }
    break;

```

Just as we did when we coded the `PlayerInputController` in the Scrolling Shooter project we handle four cases: `ACTION_UP`, `ACTION_DOWN`, `ACTION_POINTER_UP` and `ACTION_POINTER_DOWN`.

In the `ACTION_UP` and `ACTION_POINTER_UP` cases the code that checks whether the player's finger was in the left or right buttons at the moment they left the screen. If it was then `mPlayersTransform.stopHorizontal` is called and the `Transform` class will set the necessary variables to influence the `PlayerUpdateComponent` the next time update is called.

In the `ACTION_DOWN` and `ACTION_POINTER_DOWN` cases the `if` and `else - if` blocks detect presses for left, right and jump. When these occur the `headLeft`, `headRight` and `triggerJump` methods (respectively) are called on the `PlayersTransform` reference.

Now we can code the `PlayerUpdateComponent` that responds to all these effects on the `Transform` and `PlayerTransform`.

PlayerUpdateComponent

Create a new class called `PlayerUpdateComponent`. To get it started, add the member variables and the empty update method. Of course, it implements `UpdateComponent`.

```
import android.graphics.PointF;

class PlayerUpdateComponent implements UpdateComponent {

    private boolean mIsJumping = false;
    private long mJumpStartTime;

    // How long a jump lasts
    // Further subdivided into up time
    // and down time, later
    private final long MAX_JUMP_TIME = 400;
    private final float GRAVITY = 6;

    public void update(long fps,
                      Transform t,
                      Transform playerTransform) {

    }

}
```

There are four, member variables. The `isJumping` variable will be `true` when the player is executing a jump and `false` otherwise. The `long mJumpStartTime` will be initialized each time the player starts a jump. It can then be measured and controlled. The `MAX_JUMP_TIME` variable which was initialized to 400 means that a jump will last for just under half a second (500 milliseconds = half a second). The `GRAVITY` variable which is initialized to 6 means that there will be an invisible force pushing down on the player that moves it at 6 virtual meters per second.



In the signature of the `update` method, the player sending its transform to itself, twice, is imperfect design on my part. The solution is to have a more flexible and a bit more complex design. For example, each component could have a method that runs after all the objects have been created by the factory but before the game starts (a bit like `initialize` on all the `GraphicsComponents` but after all objects are built). The `StartUp` method could be passed a reference which allows it to query for any aspect of any other object. All the `GameObject` components could then request a reference to all the different `Transform`(or anything else it needs) instances from whatever objects it requires. By running this method before the game starts there would be no performance cost because it runs before game-play begins, all the components from all the game objects could initialize private member references to whatever they need and fewer parameters would be needed in the other methods of the components. This is for the reader to explore when they are ready.

Now add the first of the code inside the `update` method shown next.

```
// cast to player transform
PlayerTransform pt = (PlayerTransform) t;

// Where is the player?
PointF location = t.getLocation();
// How fast is it going
float speed = t.getSpeed();

if (t.headingLeft()) {
    location.x -= speed / fps;
} else if (t.headingRight()) {
    location.x += speed / fps;
}

// Has the player bumped their head
if (pt.bumpedHead()) {
    mIsJumping = false;
    pt.handlingBumpedHead();
}
```

First, we get a reference to the `PlayerTransform` by casting to `(PlayerTransform)` the `Transform` reference passed in as a parameter. The `pt` object can now be used to do anything on the `PlayerTransform`. Including access all the values that the `PlayerInputComponent` has been manipulating.

We then get a local reference to the location using `t.getLocation` and declare plus initialize the local variable `speed` by using `t.getSpeed`.

Following this there is an `if - else if` structure. The `if` executes when the player is heading left and moves the player to the left relative to the frames per second. And the `else if` executes when the player is moving right and moves the player to the right.

The final part of the code we just added checks whether the player has bumped their head. This will have been triggered by the `PhysicsEngine` in the `collisionDetection` method. If `pt.bumpedHead` returns `true` then `mIsJumping` is set to `false` and the code calls `pt.handlingBumpedHead`. The effect of calling `handlingBumpedHead` is that the `PlayerTransform` class will return `false` the next time `bumpedHead` is called; until the player bumps their head again.

Add this next block of code which runs when a jump is triggered. Remember that when the player presses the jump button, the `PlayerInputComponent` class sets a boolean in the `PlayerTransform` so that `pt.jumpTriggered` (in the next code) will return `true`.

```
// Check if jump was triggered by the player
// and if player
// is NOT ALREADY jumping or falling
// (because we don't want to jump in mid air)
// If you want double jump (or triple etc.)
// allow the jump when not grounded,
// and count the number of non-grounded
// jumps and disallow jump when
// your preferred limit is reached
if (pt.jumpTriggered()
    && !mIsJumping
    && pt.isGrounded()) {

    SoundEngine.playJump();
    mIsJumping = true;
    pt.handlingJump();
    // Thanks for the notification
    // I'll take it from here
    mJumpStartTime = System.currentTimeMillis();
}
```

If a jump has been triggered, the player is not already jumping, and the player is on the ground (`isGrounded`) then four things happen.

1. A sound is played
2. The variable `mIsJumping` is set to true to inform the code that follows
3. The method `pt.handlingJump` is called. This causes the `PlayerTransform` to return false the next time `pt.jumpTriggered` is tested and preventing this block of code from executing again until the next jump
4. The variable `mJumpStartTime` is set to the current time so the code that follows can begin measuring a jump.

Add the final part of the `update` method as shown next. Be sure to study the new code and then we will go through it.

```
// Gravity
if (!mIsJumping) {
    // Player is not jumping apply GRAVITY
    location.y += GRAVITY / fps;
} else if (mIsJumping) {
    // Player is jumping
    pt.setNotGrounded();

    // Still in upward (first part) phase of the jump
    if (System.currentTimeMillis()
        < mJumpStartTime + (MAX_JUMP_TIME / 1.5)) {
        // keep going up
        location.y -= (GRAVITY * 1.8) / fps;
    }
    // In second (downward) phase of jump
    else if (System.currentTimeMillis()
        < mJumpStartTime + MAX_JUMP_TIME) {
        // Come back down
        location.y += (GRAVITY) / fps;
    }
    // Has the jump ended
    else if (System.currentTimeMillis()
        > mJumpStartTime + MAX_JUMP_TIME) {

        // Time to end the jump
        mIsJumping = false;
    }
}

// Let the colliders know the new position
t.updateCollider();
```

The key to understanding the code just added is to look at its structure. There is an outer `if - else if`. Here it is again with all the code stripped out.

```
if(!isJumping){
    ...
}
else if(mIsJumping){
    ...
}
```

We can see that the `if` block executes when the player is not jumping and the `else if` block executes when the player is jumping. In the block when the player is not jumping a single line of code executes that moves the player down based on the gravity and the current frames per second.

The `else if` block is where the stages of the jump are handled. The first line of code inside the `else if` block calls `pt.setNotGrounded`. Now look again at just the structure inside the main `else if` block. Here it is again with the code stripped out.

```
if (System.currentTimeMillis()
    < mJumpStartTime + (MAX_JUMP_TIME / 1.5)) {
    ...
}
else if (System.currentTimeMillis()
    < mJumpStartTime + MAX_JUMP_TIME) {
    ...
}
else if (System.currentTimeMillis()
    > mJumpStartTime + MAX_JUMP_TIME) {
    ...
}
```

That `if - else if - else if` structure defines the three phases of a jump. The first, when the duration of the jump is less than two thirds (`MAX_JUMP_TIME / 1.5`) through the jump. The second when the duration is between two thirds but less than the maximum duration. And the third when the duration has exceeded the maximum allowed.

This is what happens in each of the first, second and third phases:

1. The vertical position is moved up(remember going up is a lower pixel value position) by a fraction of the gravity (`location.y -= (GRAVITY * 1.8) / fps`). Remember that when a jump is executing the player is not pushed down by gravity.

2. The vertical position is moved down(remember going down is a higher pixel value position) by a fraction of gravity (`location.y += (GRAVITY) / fps`). The code that moves the player down each frame by full gravity does not execute again until the jump has finished (`mIsJumping = false`).
3. The jump is stopped until the next time one is triggered and the whole process starts again.

All Bob's components are ready to roll. However, we won't see them function fully until we add the collision detection in the next chapter.

Coding a parallax background

Much of the code that makes the backgrounds work is the same as it was in the scrolling shooter project. However, we will be coding a background-specific transform that extends `Transform` and we will also be using the `Camera` class to tweak the position of the backgrounds and make them create a parallax effect as well as vertical movement(compared to just horizontal in the previous project).

We will see the tricks and the code required to achieve a parallax scrolling background effect. The parallax effect is when different layers of backgrounds are moved at different speeds to achieve the effect of motion and depth. By moving the front layer(s) faster than the back the distance/depth effect is achieved.

To be clear, the back layer is the background and the front layer is all the platforms and other game objects.

Video games didn't invent this technique and the first modern use of the parallax effect dates to early cinema. The famous Disney production *Snow White* used this effect as well as another trick where layers of backgrounds are moved in opposite directions to achieve a rotating effect.

The first time the effect was used in a video game was Atari's *Moon Patrol*. Why not take a look at the video in this link: <https://www.youtube.com/watch?v=M3CrqmhDk2A>

Notice the foreground which is plain green with the occasional pot-hole, scrolls at a different(faster) speed to the background hills. It is also possible with more modern hardware (like our Android phones) to have transparent parts to the layers and overlap them with other layers to create a more pleasing effect.

Coding the BackgroundTransform

Create a new class called `BackgroundTransform` and make it extend `Transform`. Add the members and constructor shown next.

```
import android.graphics.PointF;

class BackgroundTransform extends Transform {

    private float xClip;
    private boolean reversedFirst = false;

    public BackgroundTransform(
        float speed,
        float objectWidth,
        float objectHeight,
        PointF startingLocation) {

        super(speed, objectWidth,
            objectHeight,
            startingLocation);
    }
}
```

We have the same members as the equivalent class from the previous project. Furthermore, on the surface we have the same parameters. The obvious difference is that this class is specific for a background but also that height, width and speed are in virtual metres and will therefore be possible to manipulate its appearance by the `Camera` class when it comes to drawing the background to the screen. The final line of code in the constructor calls the super-class to get `Transform` set up.

Add the following getter and setter methods to complete the `BackgroundTransform` class.

```
boolean getReversedFirst() {
    return reversedFirst;
}

void flipReversedFirst() {
    reversedFirst = !reversedFirst;
}

float getXClip() {
    return xClip;
}
```

```
void setXClip(float newXClip) {  
    xClip = newXClip;  
}
```

These getter and setter methods are the same ones that used to be part of the Transform class in the previous project when everything was a bit squashed together. Now they are neatly present in this more specific BackgroundTransform class.

Coding the BackgroundGraphicsComponent

This at a glance looks very similar to the class in the previous project but there are a few differences.

Add the two methods of the BackgroundGraphicsComponent including the empty draw method. We can then code the draw method a chunk at a time.

```
import android.content.Context;  
import android.graphics.Bitmap;  
import android.graphics.Canvas;  
import android.graphics.Paint;  
import android.graphics.PointF;  
import android.graphics.Rect;  
  
import com.gamecodeschool.platformer.GOSpec.GameObjectSpec;  
  
class BackgroundGraphicsComponent implements GraphicsComponent {  
  
    private String mBitmapName;  
  
    @Override  
    public void initialize(Context context,  
                           GameObjectSpec spec,  
                           PointF objectSize,  
                           int pixelsPerMetre) {  
  
        mBitmapName = spec.getBitmapName();  
  
        BitmapStore.addBitmap(context,  
                               mBitmapName,  
                               objectSize,  
                               pixelsPerMetre,  
                               true);  
  
    }  
}
```

```
    }

    @Override
    public void draw(Canvas canvas,
                     Paint paint,
                     Transform t,
                     Camera cam) {

    }

}
```



Note the highlighted import statement that might need amending.

The only member variable is the `mBitmapName` `String`. In the constructor it is initialized from the passed in `GameObjectSpec` reference and passed on to the `BitmapStore` to get the `Bitmap`, scaled and stored in the `HashMap`.

Next add the first part of the code inside the `draw` method.

```
// Cast to a background transform
BackgroundTransform bt = (BackgroundTransform)t;

Bitmap bitmap = BitmapStore.getBitmap(mBitmapName);
Bitmap bitmapReversed = BitmapStore
    .getBitmapReversed(mBitmapName);

int scaledXClip = (int)(bt.getXClip()
    * cam.getPixelsPerMetre());

int width = bitmap.getWidth();
int height = bitmap.getHeight();
PointF position = t.getLocation();

float floatstartY = ((cam.getyCentre() -
    (cam.getCameraWorldCentreY() - position.y) *
    cam.getPixelsPerMetreY())));
```

```

int startY = (int) floatstartY;
float floatendY = ((cam.getyCentre() -
    (cam.getCameraWorldCentreY() -
        position.y - t.getSize().y) *
        cam.getPixelsPerMetreY())));

int endY = (int) floatendY;

```

The differences in how we handle the first part of the draw method for this background compared to the Scrolling Shooter project are as follows:

- Firstly, is that the two `Bitmap` references are retrieved from the `BitmapStore`.
- Secondly, we declare a local variable called `scaledXClip` and initialize it by multiplying `xClip` from the `BackgroundTransform` by `pixelsPerMetre` from the `Camera` class. The background "knows" how big it is in the virtual world but only the camera "knows" what this translates to in pixels.
- Thirdly, when the `floatStartY` and `floatEndY` variables are initialized they are done so using world coordinates (`getyCentre` and `getCameraWorldCentre`) translated to pixel coordinates by the `Camera` class. In the scrolling Shooter project, we simply assigned zero to `startY` and the screen height (in pixels) + 20 to `endY`.

Next add the final part of the code to the draw method. Add it immediately after the code we just discussed.

```

// Position the reversed bitmap
Rect fromRect2 = new Rect(width - scaledxClip, 0, width, height);
Rect toRect2 = new Rect(0, startY, scaledxClip, endY);

// Draw the two bitmaps
if (!bt.getReversedFirst()) {
    canvas.drawBitmap(bitmap, fromRect1, toRect1, paint);
    canvas.drawBitmap(bitmapRevesed, fromRect2, toRect2, paint);
} else {
    canvas.drawBitmap(bitmap, fromRect2, toRect2, paint);
    canvas.drawBitmap(bitmapRevesed, fromRect1, toRect1, paint);
}

```

This code works the same as it did in the Scrolling Shooter project except that `scaledXClip` is used instead of `xClip`.

Coding the BackgroundUpdateComponent

There is just one method required and it is the update method as dictated by the UpdateComponent interface. Add a new class called BackgroundUpdateComponent and code it as shown next.

```
class BackgroundUpdateComponent implements UpdateComponent {

    @Override
    public void update(long fps,
                      Transform t,
                      Transform playerTransform) {

        // Cast to background transform
        BackgroundTransform bt =
            (BackgroundTransform) t;

        PlayerTransform pt =
            (PlayerTransform) playerTransform;

        float currentXClip = bt.getXClip();

        // When the player is moving right -
        // update currentXClip to the left
        if (playerTransform.headingRight()) {
            currentXClip -= t.getSpeed() / fps;
            bt.setXClip(currentXClip);
        }

        // When the player is heading left -
        // update currentXClip to the right
        else if (playerTransform.headingLeft()) {
            currentXClip += t.getSpeed() / fps;
            bt.setXClip(currentXClip);
        }

        // When currentXClip reduces either
        // extreme left or right
        // Flip the order and reset currentXClip
        if (currentXClip >= t.getSize().x) {
            bt.setXClip(0);
            bt.flipReversedFirst();
        } else if (currentXClip <= 0) {
            bt.setXClip((int) t.getSize().x);
            bt.flipReversedFirst();
        }
    }
}
```

```
    }  
    }  
}
```

As with the `BackgroundGraphicsComponent` the `BackgroundUpdateComponent` works the same way it did in the Scrolling Shooter project except that now the code uses its own dedicated `BackgroundTransform` class.

Updating the levelManager, GameObjectFactory and GameObject

Now we can add the player (Bob), animated fire tiles and a scrolling background to the game.

In the long switch block of the `LevelManager` class uncomment the following highlighted code to build backgrounds, player and the animated fire tiles.

```
switch (levelToLoad.get(row)  
        .charAt(column)) {  
  
    case '1':  
        //objects.add(factory.create(  
        // new BackgroundCitySpec(),  
        // coords));  
        break;  
  
    case '2':  
        //objects.add(factory.create(  
        // new BackgroundUndergroundSpec(),  
        // coords));  
        break;  
  
    case '3':  
        //objects.add(factory.create(  
        // new BackgroundMountainSpec(),  
        // coords));  
        break;  
  
    case 'p':  
        //objects.add(factory.create(new  
        // PlayerSpec(),
```

```
        // coords));
        // Remember the location of
        // the player
        //PLAYER_INDEX = objects.size()-1;
        break;

    case 'g':
        objects.add(factory.create(
            new GrassTileSpec(),
            coords));
        break;

    case 'o':
        objects.add(factory.create(
            new ObjectiveTileSpec(),
            coords));
        break;

    case 'm':
        //objects.add(factory.create(
        // new MoveablePlatformSpec(),
        // coords));
        break;

    case 'b':
        objects.add(factory.create(
            new BrickTileSpec(),
            coords));
        break;

    case 'c':
        objects.add(factory.create(
            new CartTileSpec(),
            coords));
        break;

    case 's':
        objects.add(factory.create(
            new StonePileTileSpec(),
            coords));
        break;

    case 'l':
        objects.add(factory.create(
```

```
        new CoalTileSpec(),
        coords));
    break;

case 'n':
    objects.add(factory.create(
        new ConcreteTileSpec(),
        coords));
    break;

case 'a':
    objects.add(factory.create(
        new LamppostTileSpec(),
        coords));
    break;

case 'r':
    objects.add(factory.create(
        new ScorchedTileSpec(),
        coords));
    break;

case 'w':
    objects.add(factory.create(
        new SnowTileSpec(),
        coords));
    break;

case 't':
    objects.add(factory.create(
        new StalactiteTileSpec(),
        coords));
    break;

case 'i':
    objects.add(factory.create(
        new StalagmiteTileSpec(),
        coords));
    break;

case 'd':
    objects.add(factory.create(
        new DeadTreeTileSpec(),
        coords));
```



```
        break;

    case 'e':
        objects.add(factory.create(
            new SnowyTreeTileSpec(),
            coords));
        break;

    case 'x':
        objects.add(factory.create(
            new CollectibleObjectSpec(),
            coords));
        gs.coinAddedToLevel();
        break;

    case 'z':
        //objects.add(factory.create(
        // new FireTileSpec(),
        // coords));

        break;

    case 'y':
        objects.add(factory.create(
            new InvisibleDeathTenByTenSpec(),
            coords));

        break;

    case '.':
        // Nothing to see here
        break;

    default:
        Log.e("Unhandled item in level",
            "row:"+row
            + " column:"+column);
        break;
}
```

Nothing new is happening in this code we are simply calling the `create` method of the `GameObjectFactory` class and passing in a different `GameObjectSpec` instance depending on what we want to build.

To enable the `GameObjectFactory` class to deal with these new objects first add this new highlighted code so the correct type of transforms is given to the new object types.

```
// First give the game object the
// right kind of transform
switch(object.getTag()){
    case "Background":
        // Code coming soon
        object.setTransform(
            new BackgroundTransform(
                spec.getSpeed(),
                spec.getSize().x,
                spec.getSize().y,
                location));
        break;

    case "Player":
        // Code coming soon
        object.setTransform(
            new PlayerTransform(spec.getSpeed(),
                spec.getSize().x,
                spec.getSize().y,
                location));
        break;

    default:// normal transform
        object.setTransform(new Transform(
            spec.getSpeed(),
            spec.getSize().x,
            spec.getSize().y,
            location));
        break;
}
```

And finally, for the `GameObjectFactory` class add the highlighted case statements to properly construct the required components when they are present in the components array.

```
...
case "PlayerInputComponent":
    // Code coming soon
    object.setPlayerInputTransform(
        new PlayerInputComponent(
            mGameEngineReference));
    break;
case "AnimatedGraphicsComponent":
    // Code coming soon
    object.setGraphics(
        new AnimatedGraphicsComponent(),
        mContext, spec, spec.getSize(),
        mPixelsPerMetre);
    break;
case "PlayerUpdateComponent":
    // Code coming soon
    object.setMovement(new PlayerUpdateComponent());
    break;
case "InanimateBlockGraphicsComponent":
    object.setGraphics(new
        InanimateBlockGraphicsComponent(),
        mContext, spec, spec.getSize(),
        mPixelsPerMetre);
    break;
case "InanimateBlockUpdateComponent":
    object.setMovement(new
        InanimateBlockUpdateComponent());
    break;
case "MovableBlockUpdateComponent":
    // Code coming soon
    break;
case "DecorativeBlockUpdateComponent":
    object.setMovement(new
        DecorativeBlockUpdateComponent());
    break;
```

```

case "BackgroundGraphicsComponent":
    // Code coming soon
    object.setGraphics(
        new BackgroundGraphicsComponent(),
        mContext, spec, spec.getSize(),
        mPixelsPerMetre);
    break;
case "BackgroundUpdateComponent":
    // Code coming soon
    object.setMovement(new BackgroundUpdateComponent());
    break;

default:
    // Error unidentified component
    break;
}
...

```

This leaves just one error in our code which is the call to `setPlayerTransform`. Let's uncomment that method in the `GameObject` class and we will be able to run the game. Here is the code that needs to be uncommented.

```

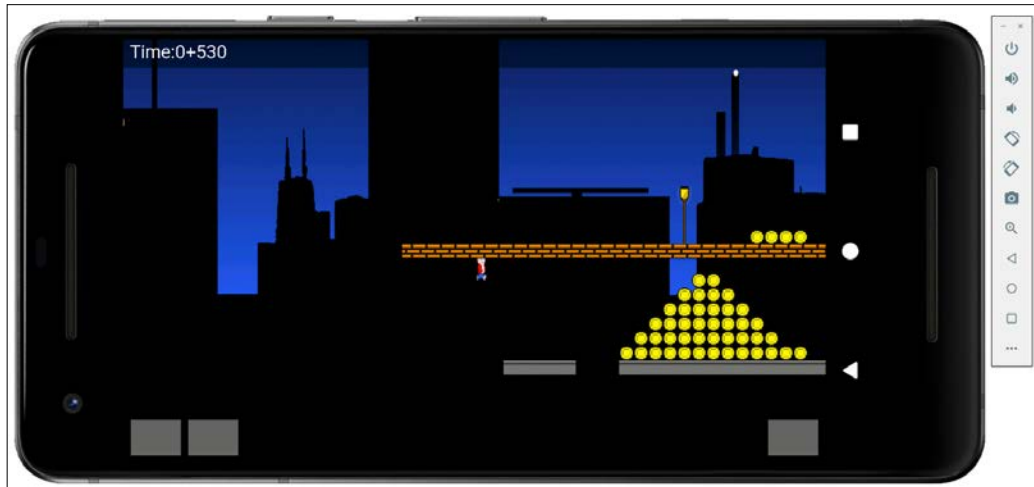
/*
Uncomment this code soon
void setPlayerInputTransform(PlayerInputComponent s) {
    s.setTransform(mTransform);
}
*/

```

 Obviously, you will also need to remove the text `Uncomment this code soon` to avoid an error.

Running the game

You can now run the game and select your preferred level. This is a screen shot of the City level as the player begins to fall through the platforms.



The player falls but he can walk and there is a pretty, scrolling parallax effect with the background and the platforms.

Summary

We are very nearly done. We learned how to animate a sprite sheet and create a parallax effect. We then put both things into action in the game.

We have one more game object to implement, the moving platforms and we also need to implement collision detection in the `PhysicsEngine` class. We will complete this game in the next chapter.

25

Intelligent Platforms and Advanced Collision Detection

This chapter is the final chapter of the final project and shorter than most. We will however, add quite a bit to the game. First, we will code an intelligent platform—one that moves up and down of its own accord. This greatly increases the potential for interesting level designs and looks good too. The final thing we will add is the collision detection. This will be more advanced than in any of the other projects as we will need to respond differently depending on which collider of the player collides with various different game objects. In addition, the collision code will need to communicate with the movement code to make the whole system behave as needed.

Chapter to-do list:

- Coding a moving platform
- Coding the collision detection
- Possible improvements
- Running the completed game

Let's wrap this thing up!

Coding the moving platform component class

All we need to do to bring this new object to life is code a `UpdateComponent`. We already have the specification in the `GOSpec` package and a regular graphics-related component class has already been coded.

MoveableBlockUpdateComponent

Create a new class called `MoveableBlockUpdateComponent`, extend `UpdateComponent` and add the required update method as shown next.

```
import android.graphics.PointF;

class MoveableBlockUpdateComponent implements UpdateComponent {

    @Override
    public void update(long fps,
                      Transform t,
                      Transform playerTransform) {

        PointF location = t.getLocation();
        if (t.headingUp()) {
            location.y -= t.getSpeed() / fps;
        } else if (t.headingDown()) {
            location.y += t.getSpeed() / fps;
        } else {
            // Must be first update of the game
            // so start with going down
            t.headDown();
        }

        // Check if the platform needs
        // to change direction
        if (t.headingUp() && location.y <=
            t.getStartingPosition().y) {

            // Back at the start
            t.headDown();
        } else if (t.headingDown() && location.y >=
            (t.getStartingPosition().y
             + t.getSize().y * 10)) {

            // Moved ten times the height downwards
            t.headUp();
        }

        // Update the colliders with the new position
        t.updateCollider();
    }
}
```

That's quite a detailed method but after everything we have seen in the `PlayerUpdateComponent` it shouldn't be too hard to break it down and see what is happening.

The method receives the usual parameters, frame rate, `Transform` of the object and the player's `Transform`.

Load up the object's current location into a local variable.

```
PointF location = t.getLocation();
```

If the object is going up decrease the vertical value of position, if going down increase it and if not moving start it heading down.

```
if (t.headingUp()) {
    location.y -= t.getSpeed() / fps;
} else if (t.headingDown()) {
    location.y += t.getSpeed() / fps;
} else {
    // Must be first update of the game
    // so, start with going down
    t.headDown();
}
```

Next, check if the platform has reached its maximum height and if it has start it heading down, check if the platform is at its lowest point and if it is start it heading up.

```
// Check if the platform needs
// to change direction
if (t.headingUp() && location.y <=
    t.getStartingPosition().y) {

    // Back at the start
    t.headDown();
} else if (t.headingDown() && location.y >=
    (t.getStartingPosition().y
     + t.getSize().y * 10)) {

    // Moved ten times the height downwards
    t.headUp();
}
```


Finally, the collider is updated with the new position.

```
// Update the colliders with the new position
t.updateCollider();
```

The moving platform is now ready to be built by the factory.

Update LevelManager and GameObjectFactory

Now we have a new object ready to go we can quickly update the LevelManager and GameObjectFactory. Uncomment this code in the LevelManager class.

```
case 'm':
    //objects.add(factory.create(
    // new MoveablePlatformSpec(),
    // coords));
    break;
```

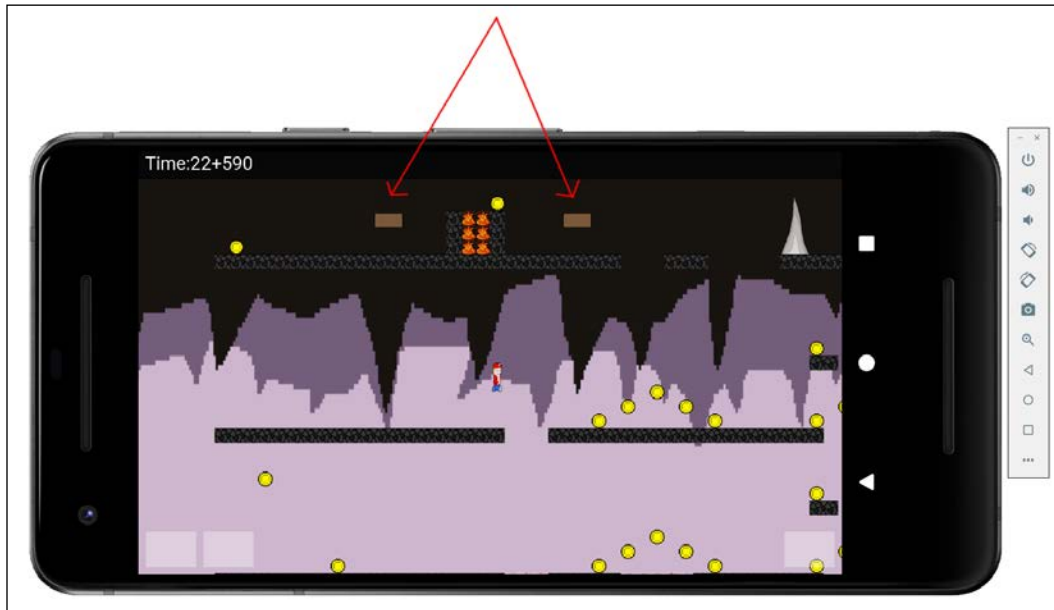
Add this highlighted line of code to the GameObjectFactory class.

```
...
case "InanimateBlockUpdateComponent":
    object.setMovement(new
        InanimateBlockUpdateComponent());
    break;
case "MovableBlockUpdateComponent":
    // Code coming soon
    object.setMovement(new
        MovableBlockUpdateComponent());
    break;
case "DecorativeBlockUpdateComponent":
    object.setMovement(new
        DecorativeBlockUpdateComponent());
    break;
...
```

The LevelManager will now request moving platforms from the factory and the factory knows how to construct the moving platforms.

Running the game

You can now see the moving blocks in all the levels. Here is the Underground level just to prove they are now active.



Unfortunately, Bob still falls straight through everything and he has no chance of picking up any coins either.

Coding the detectCollisions method: Part 1

All the code for handling collisions will go in the `detectCollisions` method. There is quite a large amount of code so let's split it up into two parts.

And once we have added the code for each part we will further sub-divide it to make sure we know how it works.

To get started, add the following code. Study it in detail as you proceed and note where the comment `// More code here next is.`

```
private void detectCollisions(GameState gs,
    ArrayList<GameObject> objects) {

    // More code here soon
    boolean collisionOccurred = false;

    // Something collides with some part of
    // the player most frames
    // so, let's make some handy references
    // Get a reference to the players position
    // as we will probably need to update it
    Transform playersTransform =
        objects.get(LevelManager.PLAYER_INDEX)
            .getTransform();

    PlayerTransform playersPlayerTransform =
        (PlayerTransform) playersTransform;

    // Get the players extra colliders
    // from the cast object
    ArrayList<RectF> playerColliders =
        playersPlayerTransform.getColliders();

    PointF playersLocation =
        playersTransform.getLocation();

    for (GameObject go : objects) {
        // Just need to check player collisions
        // with everything else
        if (go.checkActive()) {
            // Object is active so check for collision
            // with player - anywhere at all
            if (RectF.intersects(go.getTransform()
                .getCollider(),
                playersTransform
                .getCollider())) {

                // A collision of some kind has occurred
                // so let's dig a little deeper
                collisionOccurred = true;
            }
        }
    }
}
```

```

        // Get a reference to the current
        // (being tested) object's
        // transform and location
        Transform testedTransform =
        go.getTransform();
        PointF testedLocation = testedTransform
            .getLocation();

        // Don't check the player against itself
        if (objects.indexOf(go) !=
            LevelManager.PLAYER_INDEX) {

            // Where was the player hit?
            for (int i = 0; i
                < playerColliders.size(); i++){

                // More code here next

            }

        }

    }

}

}

if (!collisionOccurred) {
    // No connection with main player
    // collider so not grounded
    playersPlayerTransform.notGrounded();
}
}
}

```

Let's break down the contents of `detectCollisions` so far.

Explaining part 1

Look at the first line of code. This code creates a new boolean called `collisionOccurred` and sets it to `false`.

```
boolean collisionOccurred = false;
```

A new local Transform reference is initialized to the player's Transform.

```
Transform playersTransform =  
    objects.get(LevelManager.PLAYER_INDEX)  
        .getTransform();
```

A new PlayerTransform object is initialized as a reference to PlayerTransform part of the player's PlayerTransform by casting the result returned from getTransform.

```
PlayerTransform playersPlayerTransform =  
    (PlayerTransform) playersTransform;
```

A local ArrayList of RectF objects is initialized as a reference to the player's detailed colliders using the getColliders method.

```
ArrayList<RectF>playerColliders =  
    playersPlayerTransform.getColliders();
```

A local PointF is initialized as a reference to the player's location using getLocation.

```
PointF playersLocation =  
    playersTransform.getLocation();
```

At this point we know just about everything we need to know about the player and the information is easily accessible from the playersTransform, playersPlayerTransform, playerColliders and playersLocation local variables. It is true we could have called getColliders, getLocation etc. whenever we needed to but making local copies like this makes the code more readable and perhaps slightly faster.

The next code in part 1 of the detectCollisions method is this (with comments stripped out):

```
for (GameObject go : objects) {  
  
    if (go.checkActive()) {  
  
        if (RectF.intersects(go.getTransform()  
            .getCollider(),  
            playersTransform  
                .getCollider())) {
```

The enhanced `for` loop goes through each of the `GameObject` instances in the `objects ArrayList`. Inside the `for` loop the current `GameObject` is checked to make sure it is active before going any further. If it is active the third line of code checks whether the current `GameObject` has collided with the player's main (all encompassing) collider. If it hasn't collided with the main collider then we know it hasn't collided with any of the more specific colliders either. If it has then we need to find out the specific details of this collision.

We set `collisionOccurred` to `true`. We respond to this variable at the end of the method.

```
collisionOccurred = true;
```

Now we make two local references to the current `GameObject` being tested. The first to its `Transform` and the next to its location. It is true that we could call `getTransform` and `getLocation` whenever we need them but doing it this way makes for more readable code and probably slightly faster as well.

```
Transform testedTransform =
    go.getTransform();

PointF testedLocation = testedTransform
                        .getLocation();
```

This next `if` statement checks that the current `GameObject` is not the player because we don't want to test the player against the player.

```
if (objects.indexOf(go) !=
    LevelManager.PLAYER_INDEX) {
```

Now we enter the main `for` loop that will do the bulk of the work. Notice the `for` loop goes through each of the player's detailed colliders from the `playerCollidersArrayList`.

```
for (int i = 0; i < playerColliders.size(); i++){

    // More code here next
    ...
```

In part two we will add a `switch` block which will detect the specific combinations of collisions and respond accordingly. The `switch` block will be added right after the comment `// More code here next`.

After the switch block that we will code next and right at the end of the `detectCollisions` method we check to see if any collisions occurred. If no collision occurred, we set the player's `PlayerTransform` to `notGrounded` which corresponds with the logic from the `PlayerMovementComponent` in the `move` method.

```
if (!collisionOccurred) {  
    // No connection with main player  
    // collider so not grounded  
    playersPlayerTransform.notGrounded();  
  
}
```

Let's code the last part of the `detectCollisions` method.

Coding the `detectCollisions` method: Part 2

Add the following code after the highlighted comment. Notice the first line of code is an `if` statement that checks for a collision between the current `GameObject` and the current player collider from the `ArrayList` of the player's detailed colliders.

```
// More code here next  
if (RectF.intersects(testedTransform  
    .getCollider(),  
    playerColliders.get(i))) {  
  
    // React to the collision based on  
    // body part and object type  
  
    switch (go.getTag() + " with " + " " + i) {  
        // Test feet first to avoid the  
        // player sinking in to a tile  
        // and unnecessarily triggering  
        // right and left as well  
        case "Movable Platform with 0":// Feet  
            playersPlayerTransform.grounded();  
            playersLocation.y =  
                (testedTransform.getLocation().y)  
                - (playersTransform.getSize().y);  
            break;  
  
        case "Death with 0":// Feet  
            gs.death();  
    }
```

```
        break;

    case "Inert Tile with 0":// Feet
        playersPlayerTransform.grounded();
        playersLocation.y =
            (testedTransform.getLocation().y)
            - (playersTransform.getSize().y);
        break;

    case "Inert Tile with 1":// Head
        // Just update the player to a suitable height
        // but allow any jumps to continue
        playersLocation.y = testedLocation.y
            + testedTransform.getSize().y;

        playersPlayerTransform.triggerBumpedHead();
        break;

    case "Collectible with 2":// Right
        SoundEngine.playCoinPickup();
        // Switch off coin
        go.setInactive();
        // Tell the game state a coin has been found
        gs.coinCollected();
        break;

    case "Inert Tile with 2":// Right
        // Stop the player moving right
        playersTransform.stopMovingRight();
        // Move the player to the left of the tile
        playersLocation.x = (testedTransform
            .getLocation().x)
            - playersTransform.getSize().x;
        break;

    case "Collectible with 3":// Left
        SoundEngine.playCoinPickup();
        // Switch off coin
        go.setInactive();
        // Tell the game state a coin has been found
        gs.coinCollected();
        break;
```



```
case "Inert Tile with 3":// Left
    playersTransform.stopMovingLeft();
    // Move the player to the right of the tile
    playersLocation.x =
        testedLocation.x
        + testedTransform.getSize().x;

    break;

// Handle the player's feet reaching
// the objective tile
case "Objective Tile with 0":
    SoundEngine.playReachObjective();
    gs.objectiveReached();
    break;

default:

    break;

}
}
```

Let's break down the contents of the second (and final) part. The following code:

```
switch (go.getTag() + " with " + "" + i) {
```

Creates a `String` which is tested by each of the `case` statements for a specific combination of objects and player colliders making contact. Let's look at each case statement in turn.

Explaining part 2

The first case checks for a movable platform colliding with the feet. When this occurs, the player's vertical location is tweaked to align the underside of the player with the top of the platform. This ensures the player stands realistically on top of the platform and can't sink into it.

```
case "Movable Platform with 0":// Feet
    playersPlayerTransform.grounded();
    playersLocation.y =
        (testedTransform.getLocation().y)
        - (playersTransform.getSize().y);

    break;
```

This next case detects the player's feet colliding with a death tile. The response is a simple one and the `GameState` class reference calls the `death` method to handle ending the game.

```
case "Death with 0":// Feet
    gs.death();
    break;
```

Next, we deal with an `Inert Tile` (such as a regular platform) colliding with the player's feet. The response is to line up the underside of the player with the top side of the platform.

```
case "Inert Tile with 0":// Feet
    playersPlayerTransform.grounded();
    playersLocation.y =
        (testedTransform.getLocation().y)
        - (playersTransform.getSize().y);
    break;
```

If the player's head bumps into a regular platform, this next case sets the player's height, so the head cannot begin to sink in to the underside of the platform. Next the `triggerBumpedHead` method is called so that the logic in the `PlayerUpdateComponent` can handle the progression of the player's jump.

```
case "Inert Tile with 1":// Head
    // Just update the player to a suitable height
    // but allow any jumps to continue
    playersLocation.y = testedLocation.y
        + testedTransform.getSize().y;

    playersPlayerTransform.triggerBumpedHead();
    break;
```

This next case executes when the right-hand side of the player touches a collectible. A sound effect is played, the collectible is set inactive and will disappear then the `GameState` class is notified that a collectible has been collected by calling `coinCollected`.

```
case "Collectible with 2":// Right
    SoundEngine.playCoinPickup();
    // Switch off coin
    go.setInactive();
    // Tell the game state a coin has been found
    gs.coinCollected();
    break;
```

If the player's right-hand side touches an inert tile, then this next `case` statement will prevent the player sinking into it. It does this by lining up the right-hand side of the player with the left-hand side of the object.

```
case "Inert Tile with 2":// Right
    // Stop the player moving right
    playersTransform.stopMovingRight();
    // Move the player to the left of the tile
    playersLocation.x = (testedTransform
        .getLocation().x)
        - playersTransform.getSize().x;
    break;
```

This next case executes when the left-hand side of the player touches a collectible. A sound effect is played, the collectible is set inactive and will disappear then the `GameState` class is notified that a collectible has been collected by calling `coinCollected`.

```
case "Collectible with 3":// Left
    SoundEngine.playCoinPickup();
    // Switch off coin
    go.setInactive();
    // Tell the game state a coin has been found
    gs.coinCollected();
    break;
```

If the player's left-hand side touches an inert tile, then this case statement will prevent the player sinking into it. It does this by lining up the left-hand side of the player with the right-hand side of the object.

```
case "Inert Tile with 3":// Left
    playersTransform.stopMovingLeft();
    // Move the player to the right of the tile
    playersLocation.x =
        testedLocation.x
        + testedTransform.getSize().x;
    break;
```

The final case statement (apart from default) tests whether the player's feet touch the objective. This means the player has completed the level. A sound effect is played and the GameState is notified via the objectiveReached method.

```
// Handle the player's feet reaching
// the objective tile
case "Objective Tile with 0":
    SoundEngine.playReachObjective();
    gs.objectiveReached();
    break;
```

The entire game is now completed!

Running the game

You can now play the game. It is much more pleasing (probably necessary) to run it on a real device because running and jumping with mouse clicks on the emulator is almost impossible.

Summary

In this short chapter we coded the moving platforms and applied the glue that makes the whole thing work; the collision detection. I hope you have enjoyed building these six games. Why not take a quick look at the final short chapter about what to do next?

26

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 easily 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 in-depth 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, upload this image etc.

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.

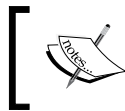
[ If you want a check-list for publishing take a look at this <https://developer.android.com/distribute/best-practices/launch/launch-checklist.html> but you will find the process intuitive (if very drawn out).]

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 devalue the many months I spent putting this work together.

If you have been helped, inspired or used my assets I would be grateful for a credit, but it is not required.



I would love to hear from you if you have published a game using the techniques, code, assets etc 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 diversifying by reading this article: <http://gamecodeschool.com/blog/making-games-where-do-i-start/>

If you are hooked with 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, think how you will create each game object, how will they behave, what game states need to be monitored and how will the code be structured.

With the "keep learning" step you will find that as you make a game you suddenly realise 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 almost inevitability why not study some patterns right away. One great source is *Head First: Java Design Patterns*. In fact, I would specifically suggest reading about the **State** pattern that 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 the website `GameProgrammingPatterns.com`. This site is packed full of 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.

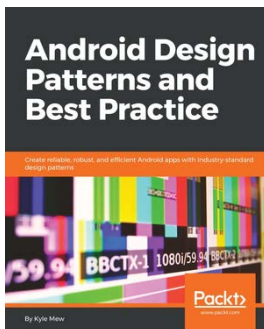
- gamecodeschool.com
- facebook.com/gamecodeschool
- twitter.com/gamecodeschool
- youtube.com/channel/UCY6pRQAXnwviO3dpmV258Ig/videos
- plus.google.com/114785498572480147747/posts
- linkedin.com/in/gamecodeschool

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 do is enough work to get it out of you.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Android Design Patterns and Best Practice

Kyle Mew

ISBN: 978-1-78646-721-8

- Build a simple app and run it on real and emulated devices
- Explore the WYSIWYG and XML approaches to material design provided within Android Studio
- Detect user activities by using touch screen listeners, gesture detection, and reading sensors
- Apply transitions and shared elements to employ elegant animations and efficiently use the minimal screen space of mobile devices
- Develop apps that automatically apply the best layouts for different devices by using designated directories
- Socialize in the digital word by connecting your app to social media
- Make your apps available to the largest possible audience with the AppCompat support library



Mastering Firebase for Android Development

Ashok Kumar S

ISBN: 978-1-78862-471-8

- Learn about Firebase push notifications and write backend functionalities
- Identify the root cause of an application crash and diagnose and fix bugs
- Store different Multipurpose Internet MailExtension(MIME) type files
- Explore web hosting and connect the Firebase functions to the host website
- Send push notifications and understand the deep integration of analytics tools and cohorts
- Market and monetize your application using Firebase Adwords and Admob
- Build a secure authentication framework while enhancing the sign-in and onboarding experience for end users

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 195, 196

access modifier

about 172

class access, in nutshell 173

default 172

public 172

used, for controlling class 172

used, for controlling variable 173, 174

used, for methods 174, 175

Accessors 178

activity

coding 654, 655

Activity class 105

AlienChaseMovementComponent

about 542-547

Code AlienLaserSpawner 548

interface, implementing 549

interface, implementing in 548

interface, implementing in

GameEngine 548-550

AlienDiverMovementComponent 549, 551

AlienHorizontalSpawnComponent 551, 552

AlienPatrolMovementComponent 552-556

aliens

GameEngine, updating 557

GameObjectFactory, updating 559

level, updating 558

spawning 557

alien's components

adding 541

AlienChaseMovementComponent 542-547

AlienDiverMovementComponent 549, 551

AlienHorizontalSpawnComponent 551, 552

AlienPatrolMovementComponent 552-556

AlienVerticalSpawnComponent 556, 557

AlienVerticalSpawnComponent 556, 557

Android

current versions, detecting 269

versions, handling 268

working 9, 10

Android Coordinate system

about 113

drawing 114

plotting 114

Android developers 3

Android emulator

Sub' Hunter game, executing 30-32

Android games

about 2, 3

combining, with Java 2, 3

development 4

Java, using 3, 4

Android lifecycle

about 227

onCreate method 229

onDestroy method 229

onPause method 229

onResume method 229

onStart method 229

onStop method 229

phases 227-229

run method, coding 230-233

used, for starting and stopping thread 230

Android Studio

Editor window 26, 27

Project panel 25, 26

setting up 10-18, 24

AnimatedGraphicsComponent 681-684

apple

coding 366
using 369, 370

Apple constructor 367-369

Application Programming Interface (API) 10

array example mini-app creating 305-307

ArrayList 371-374

array notation syntax 304

array of bullets

spawning 313-317

arrays

about 307, 373
dynamic array example 307-309
out of bounds exception 313

B

BackgroundGraphicsComponent

coding, in sprite-sheets 695-697

background's components

about 506
BackgroundGraphicsComponent 519-522
BackgroundMovementComponent 523, 524
BackgroundSpawnComponent 524
coding 518, 519

background's empty component classes

BackgroundGraphicsComponent 493, 494
BackgroundMovementComponent 494
BackgroundSpawnComponent 494
coding 488

BackgroundTransform

coding, in sprite-sheets 694, 695

BackgroundUpdateComponent

coding, in sprite-sheets 698

Ball Class

about 237, 238
Ball constructor, coding 240, 241
Ball helper method, coding 243-245
Ball update method, coding 242, 243
game engine communication 239
realistic-ish bounce, coding 245-247
rectangles, representing with RectF 239
RectF getter method, coding 241
squares, representing with RectF 239

using 247-250

variables, coding 240

Bat class

about 250, 251
constructor, coding 252, 253
helper method, coding 254
update method, coding 254, 255
using 256
variables, coding 251, 252

Bat input handling

coding 257, 258

Bitmap class 104

Bitmaps

about 380, 381
Matrix class 381
rotating 379, 380

BitmapStore

about 606
class, coding 606-611

bitwise operators

reference link 147

Bob class

about 321
adding, to collision detection 327-329
coding 322-326
drawing, to screen 329, 330
graphic file, adding to project 322
printDebuggingText method, coding 332
sound effects, adding 330
spawnBullet method, coding 333, 335
startGame method, coding 335
teleport, activating 331, 332
using 326, 327

Bob Was in A Hurry platform game

about 566-569, 576
assets, adding 576
cameras 572, 573
graphics 571, 572
level design files example 569-571
new improved transform hierarchy 574
project, creating 576
real world 572, 573
slightly modified ObjectSpec 573

broadcaster interface

coding, in Scrolling Shooter project 444

bugs 79

Bullet class

- coding 297-300
- spawning 300-302

Bullet Hell game

- about 6
- AndroidManifest.xml file, modifying 288
- classes, creating 288
- executing 318, 336
- initiating 287
- planning 286, 287

BulletHellGame class

- BulletHellGame constructor, coding 292, 293
- BulletHellGame methods, coding 293, 294
- coding 290
- draw and onTouchEvent, drawing 294
- member variables, coding 291, 292
- pause, coding 295
- printDebuggingText, coding 295
- resume, coding 295

C

camel casing 63

camera

- coding 640-644
- formulas, testing with hypothetical coordinates 644-646

Canvas class

- about 103, 104, 105
- Activity content, setting 107
- and Bitmap class 104
- objects, initializing 107
- objects of classes, preparing 106
- using 106

Canvas Demo app

- about 108
- Bitmap initialization, exploring 110
- coding 109
- Color.argb, explaining 112, 113
- new project, creating 108
- screen, drawing on 111

casting 71

chaining 75

class

- about 161
- code, viewing 161

- object, declaring 162-165
- object, initializing 162-165
- object, using 162-165

classes mini-app

- about 166
- first class 170
- first class, creating 166-169

class implementation 162

collision detection

- bat and ball example 280
- coding 280
- detectCollisions method, coding 711, 716, 719, 720
- four walls example 281, 282
- game, executing 721

collision detection options

- about 262
- crossing number algorithm method 265, 266
- radius overlapping method 264, 265
- rectangle intersection method 262, 263

collisions

- detecting 560-563
- detection, options 262
- handling 262
- optimizations 266
- Pong game, options 267
- RectF intersects method, using 268

collisions optimizations

- about 266
- multiple hitboxes 266
- neighbour checking 266, 267

compiling process 9

component interfaces

- coding 485, 593
- EngineController 596
- GameEngineBroadcaster 596
- GraphicsComponent 594, 595
- GraphicsComponent interface 486
- InputComponent interface 486
- InputObserver 596
- MovementComponent 487
- SpawnComponent 487
- UpdateComponent 595

concatenation 71, 72

constructor 163, 179

continue keyword 144

control flow blocks
 combining 143
control flow statements 128

D

Dalvik EXecutable (DEX) code 9
Dalvik Virtual Machine (DVM) 9
deadlock 220
debugging Information
 printing 80, 81
decorative component
 coding 615
 DecorativeBlockUpdateComponent 615
design pattern 403, 408
detectCollisions method
 coding, in collision detection 711-720
do while loop 128
draw method, PongGame class
 about 215, 217
 printDebuggingText method,
 adding 214, 215

E

else keyword 138-140
encapsulation
 about 159, 171
 about 183-187
 class, controlling with access modifier 172
 methods, with access modifier 174, 175
 objects, setting up with constructor 179-181
 private variables, accessing
 with getters 176-178
 private variables, accessing
 with setters 176-178
 static methods 181, 182
 this keyword, using 181
 variable, controlling with access
 modifier 173, 174
entity-component pattern
 about 469
 coding 469, 470
 composition, over inheritance 472
 diverse object types, managing 469
 generic GameObject, used for code
 structure 471

enumeration 375, 376
errors 79
expression 71

F

fields 97, 162
for loop
 about 128, 132
 using, to draw Sub' Hunter grid 133-136

G

game
 testing 100
GameEngine
 coding 655-660
GameEngine class
 updating 534-537
game engine, Snake game
 coding 348
 constructor, coding 350, 351
 draw method, coding 355, 356
 members, coding 348, 349
 newGame method, coding 352
 onTouchEvent, coding 356, 357
 pause method, coding 357
 resume method, coding 357
 run method, coding 352, 353
 update method, coding 354
 updateRequired method, coding 353, 354
game loop
 about 217, 218
 phases 217
game loop implementation, with threads
 about 224
 activity lifecycle 226
 Android lifecycle 227
 run method, implementing 224
 Runnable, implementing 224
 thread, coding 225
 thread, starting 225, 226
 thread, stopping 225, 226
GameObject class
 about 502-505
 coding 627-629
 Transform class, used 495-501

- GameObjectFactory**
 - building 525-531
 - coding 624-627
 - updating, in intelligent platform 710
- game objects 468**
- game objects, specifying with**
 - GameObjectSpec classes**
 - about 577
 - BackgroundCitySpec 580, 581
 - BackgroundMountainSpec 581
 - BackgroundUndergroundSpec 582
 - BrickTileSpec 582
 - CartTileSpec 583
 - CoalTileSpec 583
 - CollectibleObjectSpec 584
 - ConcreteTileSpec 585
 - DeadTreeTileSpec 585
 - FireTileSpec 586
 - GameObjectSpec 578, 579
 - GrassTileSpec 586, 587
 - LamppostTileSpec 588
 - MoveablePlatformSpec 588, 589
 - ObjectiveTileSpec 589
 - PlayerSpec 589
 - ScorchedTileSpec 590
 - SnowTileSpec 591
 - SnowyTreeTileSpec 591
 - StalactiteTileSpec 592
 - StalagmiteTileSpec 592
 - StonePileTileSpec 593
- GameState class**
 - coding 629-635
 - communicating, to GameEngine 414
 - finishing 422-424
 - high score, loading 419-421
 - high score, saving 419-421
 - passing, from GameEngine to other classes 414
 - special button, pressing 422
 - used, for controlling Scrolling Shooter project 413
 - using 425, 426
- garbage collection 342**
- German support**
 - adding, to snake game 360

- graphics**
 - adding 365
- GraphicsComponent interface 486**

H

- handleInput method**
 - coding, in Scrolling Shooter project 447-450
- heap 341, 343**
- HUD class**
 - building, for control button and text display 429, 430
 - drawControls, coding 435, 436
 - draw method, coding 433, 434
 - getControls, coding 435, 436
 - prepareControls method, coding 431-433
 - using 438, 439
- HUD (heads-up-display)**
 - about 38
 - coding 647-651

I

- if keyword 138**
- ImageView class 104, 105**
- inanimate component**
 - coding 615
 - InanimateBlockUpdateComponent 618
 - InanimateBlockGraphicsComponent 615-617
- infinite loop 130**
- inheritance**
 - about 159, 160, 188-190
 - mini-app, creating 190-194
- InputComponent interface 487**
- InputObserver interface**
 - coding, in Scrolling Shooter project 445
- instance variable 162**
- intelligent platform**
 - game, executing 711
 - GameObjectFactory, updating 710
 - LevelManager, updating 710
 - moving platform component class, coding 707
- interface**
 - about 415, 197, 198
 - coding 416

- example 415
- GameState class, coding 418, 419
- implementing 416, 417
- reference, passing 417, 418
- solution, implementing 416
- used, for granting partial access to class 415

J

jargon

- handling 61, 62

Java

- working 9, 10

Java Arrays

- about 302-304
- using, as objects 304

Java code

- decision, creating 125, 126

Java Hashmap 604-606

Java methods

- about 43, 44
- overriding methods 45

Java packages

- importing, for class addition 52, 53
- using 51

Java syntax 62

Java variables

- about 62-64
- primitive types 64-66
- reference variables 66
- types 64

K

keywords 62

L

Level class

- coding 531-534

LevelManager

- updating, in intelligent platform 710

LevelManager class

- coding 660, 661, 666

levels

- creating 619
- LevelCity file 620, 621

- Level class 620

- LevelMountains 622

- LevelUnderground 623, 624

local variable 97, 162

locking 220

loop

- about 128
- do while loop 131
- for loop 132
- nested loop 133
- while loop 129, 130

M

Matrix class

- head to face, inverting 381, 382
- head to face, rotating 383, 384

member variable 97, 162

memory 340

memory problem 606

method access

- summary 175

method body 91, 92

method definition 44

method names 89, 90

method overloading

- about 95
- example 92
- project, creating 92

method overloading mini-app

- coding 93, 94
- executing 95, 96

methods

- about 83, 84
- linking up 53-59
- modifier 87
- parameters 90, 91
- return type 88, 89
- revisting 97, 98

method signature 85-87

modifier 87

MotionEvent class

- reference link 146

MoveableBlockUpdateComponent 708, 709

MovementComponent 487

moving platform component class

- coding, in intelligent platform 707

- multidimensional Array mini app**
 - creating 309-312
- Multitouch UI controller**
 - coding, in Scrolling Shooter project 447
 - creating, in listener 447
- Mutators 178**

N

- nested loop 132**
- newGame method**
 - Random based code, adding 99, 100
- nextInt method 98, 99**
- nth dimension**
 - entering, with arrays 309

O

- Object-Oriented Programming (OOP)**
 - about 47-49, 157, 158, 188, 189, 190
 - classes 48-50
 - encapsulation 159
 - features 160
 - inheritance 159, 160
 - instances 49, 51
 - objects 48, 49, 51
 - polymorphism 160
- objects**
 - setting up, with constructor 179-181
- object specifications**
 - about 477
 - AlienChaseSpec 479, 480
 - AlienDiverSpec 481
 - AlienLaserSpec 481, 482
 - AlienPatrolSpec 482, 483
 - BackgroundSpec 483, 484
 - coding 479
 - parent class, coding 477, 478
 - PlayerLaserSpec 484
 - PlayerSpec 485
- Observer pattern**
 - about 442
 - coding, in Scrolling Shooter project 444
 - in Scrolling Shooter project 442, 443
 - Scrolling Shooter project 442, 443
- onTouchEvent method**
 - coding 147, 148

- Open-World Platformer game 8**

- operators 126, 128**
 - about 69
 - addition operator (+) 70
 - assignment operator (=) 70
 - decrement operator (--) 70
 - division operator (/) 70
 - increment operator (++) 70
 - multiplication operator (*) 70
 - reference link 71
 - subtraction operator (-) 70
- Oracle Java tutorials**
 - reference link 66
- overriding methods 45**

P

- packages 51**
- Paint class 104, 105**
- parallax background**
 - coding, in sprite-sheets 693
- parameters 90, 91**
- Particle class**
 - coding, in Scrolling Shooter project 453, 454
- particle system**
 - adding, in Scrolling Shooter engine 459, 461
 - drawing, with Renderer in Scrolling Shooter engine 459, 461
- ParticleSystem class**
 - coding, in Scrolling Shooter project 455-459
- particle system explosion**
 - implementing, in Scrolling Shooter project 452
- PhysicsEngine**
 - coding 638
 - updating 537
- platformer**
 - building 569
- PlayerInputComponent 684-687**
- player's components**
 - about 506
 - coding, in spirte-sheets 675
 - GameObject/Component reality
 - check 524, 525
 - LaserMovementComponent 515-517
 - LaserSpawnComponent 517
 - PlayerInputComponent 511-515

- PlayerMovementComponent 508-510
- PlayerSpawnComponent 510
- StdGraphicsComponent 506, 507
- players components**
- player's empty component classes**
 - coding 488
 - LaserMovementComponent 492
 - LaserSpawnComponent 492
 - PlayerInputComponent interface 490-492
 - PlayerLaserSpawner interface 490-492
 - PlayerMovementComponent 489
 - PlayerSpawnComponent 490
 - StdGraphicsComponent 488, 489
- player's transform**
 - coding, in sprite-sheets 675
- PlayerTransform 675-681**
- PlayerUpdateComponent 688-693**
- polymorphic methods 194**
- polymorphism**
 - about 160, 194
 - abstract class 195, 196
 - interface 197, 198
- PongActivity class**
 - about 202
 - coding 202-204
- Pong engine**
 - BulletHellActivity, coding 289, 290
 - Bullet Hell engine, testing 296, 297
 - BulletHellGame class, coding 290
 - reusing 289
- Pong game**
 - about 5
 - collisions, options 267
 - executing 234
 - initiating 198
 - planning 199
 - playing 282
 - project, setting up 199, 200
 - running 259
 - sound, adding 277
 - SoundPool, initializing 278, 279
 - sound variables, adding 277
- PongGame class**
 - coding 205-207
 - draw method, coding 213, 214
 - member variables, adding 208-210
 - startNewGame method, coding 212, 213
 - SurfaceView class 215, 216
- PongGame constructor**
 - coding 211, 212
- primitive types**
 - about 65
 - boolean 66
 - float 66
 - int 65
 - long 65
- private variables**
 - accessing, with getters 176-178
 - accessing, with setters 176-178
- project patterns**
 - about 574
 - BitmapStore 575
 - LevelManager 575
 - Levels 575

R

- Random class**
 - about 98, 99
 - reference link 99
- random numbers**
 - generating, to deploy sub 98
- real-time strategy (RTS) 166**
- reference type 340**
- reference variables**
 - about 66
 - array references 67
 - object/class references 67
 - string references 66
- Renderer**
 - coding 639, 640
 - updating 538, 539
- Renderer class**
 - building, for handling drawing 436, 437
 - using 438, 439
- Runnable interface**
 - reference link 222

S

- scope 96, 97**
- screen touches 144-146**

Scrolling Shooter engine

- particle system, adding 459, 461
- particle system, drawing with
 Renderer 459, 461

Scrolling Shooter game

- about 7, 8
- executing 539, 666-668

Scrolling Shooter project

- about 404-407
- broadcaster interface, coding 444
- controlling, with GameState class 413
- executing 439, 451, 464
- GameActivity class, coding 409, 410
- GameEngine, BroadCaster making 445, 446
- GameEngine class 410,-412
- handleInput method, coding 447-450
- InputObserver interface, coding 445
- listener, creating 447
- manifest, editing 409
- Multitouch UI controller, coding 446
- Observer pattern 442, 443
- Observer pattern, coding 444
- Particle class, coding 453, 454
- ParticleSystem class, coding 455-459
- particle system explosion,
 implementing 452
- physics engine, building 462, 463
- starting 408
- structure 407
- testing 429
- UIController, using 450, 451

Simple Factory pattern

- about 473, 474
- scenarios 475, 476

Singleton code 601-604

Singleton pattern 599, 601

Snake class

- checkDinner method, coding 392, 393
- coding 384, 385
- constructor, coding 386-388
- detectDeath method, coding 391, 392
- draw method, coding 393-395
- move method, coding 389-391
- reset method, coding 388
- switchHeading method, coding 395, 396
- using 396, 398

Snake Clone game 6, 7

snake game

- about 343-345
- ArrayList 371, 372
- arrays, using 371
- code, amending 362
- completed game, running 399, 400
- creating, in different languages 359
- empty classes, adding 346
- enhanced for loop 373
- executing 358
- executing, in German 363, 364
- executing, in Spanish 363, 364
- finishing 399
- full screen, making 345, 346
- game engine, coding 348
- German support, adding 360
- landscape, making 345, 346
- reference link 344
- SnakeActivity, coding 346, 347
- sound effects, adding 348, 384
- Spanish support, adding 360
- string resources, adding 361

sound effects

- generating 274-276

sound engine

- building 426
- files, adding to project 426
- SoundEngine class, coding 427, 428
- SoundEngine class, using 428, 429

SoundEngine

- coding 635-638

Soundpool class

- about 269, 270
- initializing 270-272
- Java Method chaining 270
- sound files, loading into memory 272
- sound, playing 273
- sound, stopping 274

sounds

- playing 280

Spanish support

- adding, to snake game 360

SpawnComponent 488

sprite-sheets

- animating 670
- animator, coding 670-674

- BackgroundGraphicsComponent,
 - coding 695-697
- BackgroundTransform, coding 694, 695
- BackgroundUpdateComponent, coding 698
- executing 706
- GameObjectFactory, updating 699, 703, 705
- GameObject, updating 699, 703, 705
- levelManager, updating 699, 703, 705
- parallax background, coding 693
- player's components, coding 675
- player's transform, coding 675
- stack 341, 343**
- static methods 181, 182**
 - mini-app, creating 183-187
- string identifier 361**
- string resources**
 - adding, to snake game 361
- subclass 190**
- Sub' Hunter game**
 - about 4, 5
 - actions flowchart/ diagram 38, 39
 - actions flowchart/ diagram, code
 - comments 39
 - Bitmap, initializing 116-118
 - boom method, coding 152
 - canvas, initializing 116-118
 - code, mapping out with comments 40-43
 - deploying 29, 30
 - drawing 115, 116
 - executing 155, 156
 - executing, on Android emulator 30-32
 - executing, on real device 32, 33
 - for loop, used for drawing grid 133-136
 - graphics, drawing 115
 - grid lines, drawing 118-120
 - HUD, drawing 120-122
 - ImageView, initializing 116-118
 - initiating 18-24
 - locking, to full-screen and landscape
 - orientation 28, 29
 - Paint, initializing 116, 118
 - planning 35-37
 - printDebuggingText method,
 - upgrading 122, 123
 - shot, drawing on grid 153, 154
 - structuring, methods used 45-47
 - takeShot method, coding 149, 150
 - takeShot method, explaining 150-152
 - testing 81
 - text, drawing 115
 - variables, declaring 72
 - variables, initializing 72
- superclass 190**
- switch keyword**
 - about 141, 142
 - example 142, 143
- syntax**
 - handling 61, 62

T

- this keyword**
 - about 183
 - using 181
- Thread class**
 - reference link 221
- threads**
 - about 219
 - aspects 219
 - catch exception handling 223, 224
 - game loop, implementing 224
 - issues 219-222
 - Java try 223
- Transform class**
 - coding 611-615

U

- UIController**
 - using, in Scrolling Shooter project 450, 451
- UIController class**
 - coding 652-654

V

- variable access**
 - summary 174
- variables**
 - about 59, 162, 341
 - declaring 68
 - initializing 68, 69

- using 68
- using, with operators 69
- variables, Sub' Hunter game**
 - declaring 73, 74
 - initializing 76-79
 - planning 72, 73
 - screen sizes and resolution, handling 74-76

W

- warning** 79
- while loop**
 - about 128-130
 - exit, determining 130

X

- XML (extensible markup language)** 29

