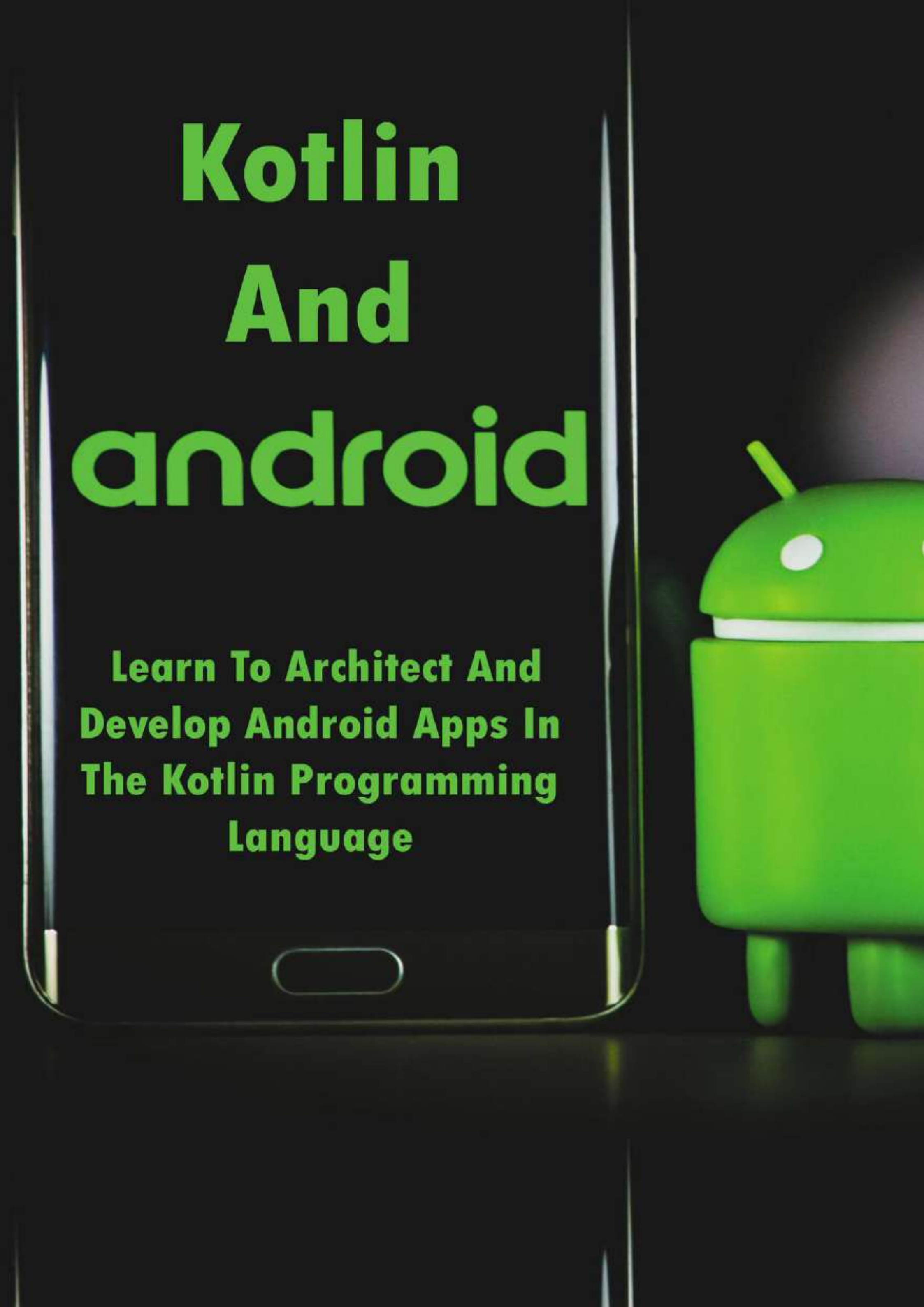
Kotlin And android

Learn To Architect And Develop Android Apps In The Kotlin Programming Language



Section 1
Getting Started With Android Studio

Android addresses a major expected market.It is additionally the most open of the“large” telephone and tablet stages. You can compose a program for an Android and let your companions have a duplicate, hush up about it or put it on special in an application store. Android telephones and tablets are similarly modest and this makes it simpler to get everything rolling.What is far and away superior,every one of the instruments you really want to make an Android application are free. You don’t have to pay anything to make, or circulate, your Android applications. To sell them utilizing a notable commercial center there may something to pay– there is a one-time charge of $25 to enroll for Google Play, yet you don’t need to utilize a specific conveyance method. All that stands among

you and your Android application is your creative mind and programming capacity. I can’t do a lot to work on your creative mind, yet I can assist with the programming side of things. In the event that you are new to Android programming this is the spot to start. In this book I will show you the basics of Android programming. Not the tips and deceives, but rather how to ponder what is happening. You’ll be acquainted with the overall rules that will make it feasible for you to dominate anything that you experience that is new later on. It is preposterous to expect to cover all of Android in one book as the subject is extremely enormous. Rather we center around the rudiments of making a User Interface (UI) as all applications must have some method of cooperating with a user.
There are numerous ways of making an Android application however Google’s Android Studio is a simple to utilize Android IDE – Integrated Development Environment – and it is presently the suggested method of doing the job.
Before Android Studio you needed to go through the Eclipse IDE and set the SDK and

different bits of programming required. This was easy, however Android Studio wipes out additional means and it makes programming Android simple. Set forth plainly, it is the method of things to come thus worth your interest in learning it.
With the arrival of Android Studio Google halted work on the Eclipse module and this implies that Android Studio truly is the best way to create applications from now on.

The Language Choice

With the arrival of Android Studio 3 you currently have a decision of programming in Java or Kotlin. The benefit of Java is that it is a notable and all around upheld language. To obtain the ability then you likely could be in an ideal situation beginning with Android Programming In Java*: Starting with an App ISBN: 978-1871962550*
Kotlin might be a be another dialect yet it is

now all around upheld for the straightforward explanation that it is 100% viable with Java. The Android libraries are totally written in Java, however Kotlin can utilize them without any issues. It is this that makes Kotlin Android improvement conceivable. Likewise, you’re not limited to Kotlin in a venture. You can add Java code to your new Kotlin venture and you can add Kotlin code to a current Java project.

Put essentially, there is next to no danger implied in moving to Kotlin and there is a great deal to be acquired. Kotlin is a lot less difficult and cleaner language than Java. It has had the advantage of perceiving how Java developed and keeping away from those slip-ups. Kotlin endeavors to cut to the chase. In Java you will more often than not think of some code over and throughout and it very well may be tedious and conceals the effortlessness of what you are attempting to do. At whatever point this happens Kotlin changes the language with the goal that you can communicate what you are doing compactly. Software engineers moving from Java to Kotlin by and large see that they like

this is on the grounds that they completely finish less composing.Developers who just realize Kotlin don’t have the foggiest idea how fortunate they are!
Kotlin does things another way from Java and keeping in mind that you can get the language as you go you may jump at the chance to peruse Programmer’s Guide To Kotlin ISBN*:978-1871962536*.It isn’t required, as long as you probably are aware Java or another item situated language, you can get Kotlin as you foster your Android applications however I suggest learning the better places of the language sometime. It pays off to know your language.
The manner in which Kotlin is utilized and the manners in which it changes Android programming specifically are presented as we come. In any case, the last section is a concise gander at the significant impacts of Kotlin on Android programming. If you want a quick overview before you start then read the final chapter, but in many ways it makes moresense to read itas a summary after you have encountered the ideas incontext.
Most importantly, except if you have a

promise to Java, you likely should begin new undertakings in Kotlin and convert existing tasks to Kotlin a cycle at a time.

What You Need to Know

You should have the option to program in an advanced article situated language. Java would be best as it is nearest to Kotlin, yet C++, C#, Visual Basic or anything comparable are close enough in soul to Java for you to have the option to adapt. You may well have to gaze things upward with regards to the points of interest of specific highlights of Kotlin, yet more often than not it ought to be self-evident, or clear with the assistance of a couple comments.
It isn’t important to be a specialist developer in light of the fact that for a ton of Android programming you are basically utilizing the elements and offices gave. That is, a ton of Android writing computer programs is simply

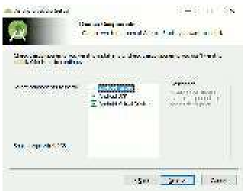
an issue of following the rules.
However, assuming you desire to deliver something remarkable and valuable you will eventually need to add something of your own– and here imagination and ability are required. So you won’t have to be a specialist software engineer to begin,yet you really want to become one when you make your astounding app.
Fortunately practice is a decent instructor thus figuring out how to benefit as much as possible from Android Studio will really assist you with figuring out how to code.

Making a Start

I ’m not going to invest a ton of energy disclosing how to introduce Android Studio in a bit by bit manner as the Android site works effectively and it is bound to be exceptional. It is worth, nonetheless, going over the fundamental principles.
https://developer.android.com/studio/
The installer will download all that you really

want including the JDK. **Windows:**

1. Launch the downloaded EXE file,
android-studio-pack <version>.exe.
2. Follow the arrangement wizard to introduce Android Studio. **Mac OS X**:
1. Open the downloaded DMG file,
android-studio-group <version>.dmg
2. Drag and drop Android Studio into the Applications folder. **Linux:**
1. Unpack the downloaded ZIP file,
android-studio-group <version>.tgz,
into a suitable area for your applications.
2. To dispatch Android Studio, explore to the
android-studio/receptacle/
catalog in a terminal and execute studio.sh.
You might need to add android-studio/container/
to your PATH ecological variable so you can begin Android Studio from any directory. Accept any defaults that the arrangement program offers you– except if you have a valid justification not to.It introduces Android Studio, yet the SDK and the virtual gadget framework that allows you to test your application.



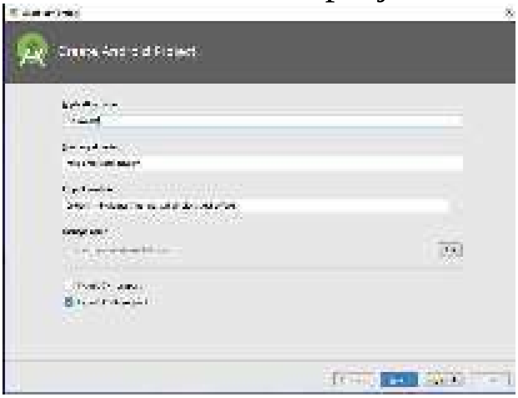
In many cases Android Studio simply introduces with no problem. Now you ought to have the option to run Android Studio. If not the most probable reason for the issue is the JDK thus re-establishment is a best first option.

Your First Program

You can select to begin Android Studio later the establishment. You will likely not get directly to Android Studio whenever it first beginnings as it downloads updates to itself and to the Android SDK. You simply must be patient.
When it at last gets moving you will see the Android Studio welcome screen:
If you have as of now made a few projects you may well see them recorded in Recent

projects.
Assuming this is your first project select theoption: Start

anewAndroidStudioproject

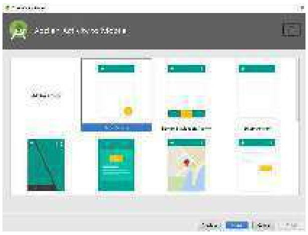


You can overlook the subtleties of the new task for the occasion.You should simply supply a name for your application– HelloWorld for this situation. Additionally ensure you have Include Kotlin support ticked– this is the thing that makes the undertaking use Kotlin rather than Java. Acknowledge different defaults that Android Studio has filled in for you.

When you click Next you are allowed the opportunity to pick what gadgets you are

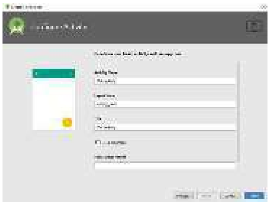
focusing on. Again essentially acknowledge the defaults:
Most of the time you will need to make applications that sudden spike in demand for an adaptation of Android that catches the greatest market however assuming this isn’t a worry then it tends to be smarter to choose a later Android version.

The next page lets you select a template for your project. In this case change the selection to Basic Activity. This gives you some additional generated code which makes the app easier to create an app that looks right. Every Android application consists of at least one Activity and this template generates a projectwith asingleActivityready for youto customize:



On the nextpage youcan assigncustomnames

for the variouscomponents of
yourprojectthatthetemplategenerates.Forarealp
names that were meaningful but in this case
you can accept the defaults:



Finally you can tap the Finish button and stand by as Android Studio makes every one of the documents you want. Indeed, even a straightforward Android project has heaps of documents so again everything takes time.

First Look

When everything is prepared you will see Android Studio for the first time. As long as everything has worked you ought to ultimately, it requires around three minutes or more, be given a perspective on your new undertaking getting going in the Layout

Editor:

Problems?

If you get any mistake messages then the odds are good that your venture hasn’t got done with being handled. Trust that the movement will stop. Assuming that you check out the status line at the lower part of the window you will see a message saying“Gradle Build Finished” when Android Studio has wrapped up with your new project. If you actually have issues it merits attempting the File,Invalidate Caches/Restart order. This normally works for“Missing styles” and comparable errors.

The IDE

Although there resembles a ton to dominate in Android Studio ’s UI, the majority of it you will just visit at times. The vital things to see are that moving from left to right you have:

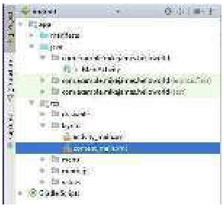
☐ The Project window
☐ The instrument Palette and the Component Tree window ☐ The Layout Editor
☐ The Attributes window

Most of the time you will utilize the Project window and the Attributes window. You will likewise see various editors relying upon what kind of record you have chosen. For this situation you have of course a design document, content_main.xml,chose and thus you have a format supervisor in the screen.

Before we go into design, which is one of the primary subjects of this book, you actually must know a little with regards to the record construction of a task so you can explore to its distinctive parts.

Basic Project Structure

When the venture has wrapped up building each of the records made can be seen by opening the Projects tab. The most compelling thing to see is that there are a large number of organizers and files:



It appears to be practically unfathomable that the least difficult Android application you can make includes so many files.
Don’t freeze. The greater part of the documents that have been made are auto-produced and more often than not you don’t have to know at least
something about them,not to mention open or alter them. Indeed opening
and altering auto-created records truly is definitely not a decent idea.
So how about we center around the documents that make a difference to us.
For our straightforward program there are just two significant
documents. One of them decides the conduct

of the Activity:
MainActivity.kt

The other decides the visual appearance, or View, of the app:

content_main.xml
You can set which Activity is the one that the framework begins, yet naturally it is the single action that you made and named when you set up the venture. You can change the default names yet for the second leave them as they are.

Despite this being a Kotlin project, the java registry, according to your perspective, is the place where the majority of the development of your application happens, so ensure you know where it is. The res catalog is the place where you store each of the assets, formats, bitmaps, and so on, that your application needs.

So while things look muddled right now the main two venture records that make a difference to you, and your undertaking, are MainActivity.kt in the java envelope and content_main.xml in the res folder.

The two different organizers in the java organizer are worried about making tests for your program. This isn’t something that we want to stress over when initially beginning to compose Android apps.

Anatomy of an Activity

An Android application is comprised of at least one Activity classes. You can consider an Activity being something like a page total with HTML to figure out what showcases and JavaScript to figure out what it does. For the situation of an Activity the format is controlled by the XML record in asset (res) organizer, this is frequently called the View, and the conduct is dictated by the Kotlin or Java code in the java folder.
The XML can be considered as a markup language similar as HTML or

XAML. It characterizes an underlying format for the screen when the application first runs. It is feasible to produce new format parts at runtime from the Java record. Truth be told, assuming you truly need to, you can get rid of the XML record and produce everything from code, yet as you will find the XML markup approach is a lot of the most ideal way to do the occupation as a result of the accessibility of the Layout Editor.

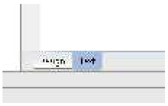
So to be 100% clear in a Kotlin project:

☐ The kt document contains the code that makes your application act specifically ways.
☐ The.xml format document contains a meaning of the underlying UI, the View, of your app.

Hello Layout Editor

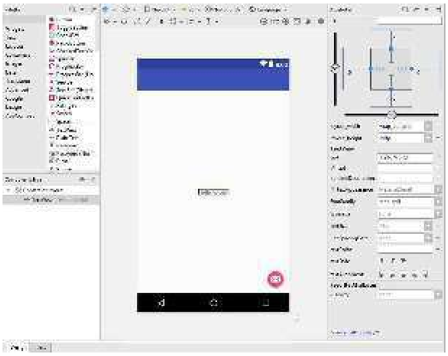
Let ’s investigate the two records that have been created for our underlying Hello World application starting with the XML design. Double tap on content_main.xml document in

the Project tab and the record will open (assuming it isn't as of now open). Assuming it is as of now open you can likewise choose its tab showed simply over the editorial manager region. You can choose any record that is open for altering by choosing its tab. You can work with the XML straightforwardly to characterize where every one of the buttons and text go, and later you will figure out how to alter it when things turn out badly or to adjust it. Be that as it may, Android Studio gives you an exceptionally decent intelligent manager– the Layout Editor and this is worth using.
As you become more encountered exchanging between a plan view and a XML view will turn out to be natural. Consider the intuitive proofreader an exceptionally simple method of creating the XML that in any case would take you ages to get right. Assuming that you check out the base left you will see two tabs– Design and Text:



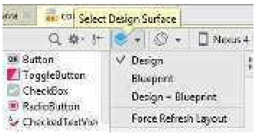
You can switch between altering the XML as text, and altering it in the intuitive Layout Editor basically by tapping on the tab. Assuming you presently click on the tab the window will show the Layout Editor however show restraint whenever you first do this it may take a couple moments.

The Layout Editor looks excessively a lot to take in when you first see it yet you will rapidly become accustomed to it. On the left is a Palette of all of the components or controls - buttons, text, checkboxes and so on - that you can place on the design surface:

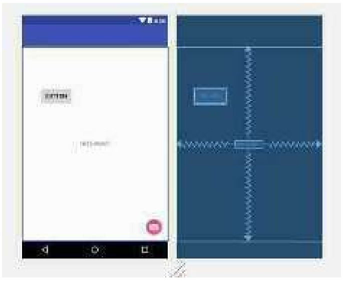


In the center is the plan surface and this defaults to the screen size and presence of the Nexus 5.You can choose different gadgets to work with.

There are, indeed, two perspectives on the format that you can utilize, the plan and the outline. As a matter of course you are shown the plan view yet you can show either view utilizing the menu at the upper left of the plan area.

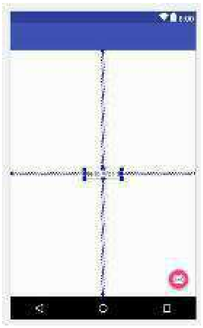


You can show the two perspectives together yet as a rule accessible screen region is the primary issue and showing only one is the most ideal choice. The plan view shows you the design as a nearby estimation to how it will show up on a genuine gadget. The outline view doesn’t attempt to deliver the UI all things considered however it gives you will more design data to assist you with situating and size components.Use whichever you are most cheerful with.

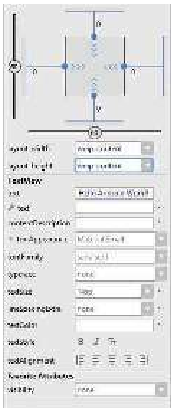


On the left, beneath the Palette, you have the Component Tree which shows you the design of your format, that is the way unique UI parts are contained inside others. It shows you the design of the XML record in a more straightforward to utilize structure. You can utilize the Component Tree as a simple method of to choose individual UI parts by tapping on their names. You can likewise move UI parts onto the Component Tree to situate them precisely inside the hierarchy. On the right you have the Attributes window that can be used to set the attributes, such as width, height, color and so on of any component in the layout. In the event that you have utilized any simplified Layout Editor then this will appear to be natural and assuming you have battled with itemized design utilizing a markup language, be it

HTML, XAML or XML, you will see the value in how simple the Layout Editor makes building and testing a UI. For the situation of our example program the main part is a solitary TextView previously containing the text“Hi World”. A TextView is the standard part to utilize when all we need to do is to show some static text.



You can alter the hello text assuming you need to. Select the TextView part either on the plan or in the Component Tree and utilize the Attributes window to observe its Text quality. Change this to peruse “Hi Android World”:



Use the text field without the spanner symbol.The properties with the spanner symbol close to them are utilized to set qualities that main show in the Layout Editor. For this situation the text field without the spanner symbol is the one that controls what shows up in your application at runtime.
You can utilize the Layout Editor to make any UI you want to and you truly don’t need to engage in the XML that relates to the design– except if things turn out badly or you want to accomplish something so complex that the Layout Editor doesn’t uphold it.

Inspecting the XML

The Layout Editor will consequently produce the XML expected to make the format for yourself and adjust it as you change the layout.
If you truly need to see the XML then you should simply choose the Text tab at the lower part of the Layout Editor window:

```
<?xml version=“1.0” encoding=“utf-8”?>
<android.support.constraint.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"


xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width=“match_parent” android:layout_height=“match_pare
nt”
app:layout_behavior=“@string/appb ar_scrolling_view_behavior”
tools:context=“com.example.mikejames.helloworld1.MainActivity”
tools:showIn=“@layout/activity_main”>

<TextView
android:layout_width=“wrap_ content”
android:layout_height=“wr ap_content”
android:text=“Hello World!” app:layout_constraintBotto
m_toBottomOf=“parent” app:layout_constraintLeft_ toLeftOf=“parent”
app:layout_constraintRight _toRightOf=“parent” app:layout_constraintTop_
toTopOf=“parent”/>

</android.support.constraint.ConstraintLayout>
```

You should find it genuinely straightforward – read the <TextView> tag for instance– yet pass on it to the Layout Editor to make and alter it.
The amounts beginning with @ are generally

references to things characterized somewhere else in asset records, a greater amount of this in section 11.
We will get back to the Layout Editor and the XML it creates in numerous later chapters.

The Kotlin

If you double tap on the MainActivity .kt document, or select the MainActivity.kt tab, you will see the code it contains.A portion of the code may be covered up however you can investigate it to by tapping the + button to grow it.

The significant piece of the code is:

```
class MainActivity : AppCompatActivity() {
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
```

You can disregard the guidelines that keep the setContentView work in light of the fact that these set up the guideline“additional items” that each Android application currently upholds– a drifting ActionBar. There are two different capacities beneath the onCreate work yet overlook these for the occasion. They execute highlights you didn’t actually look for which can be helpful, yet not when you are simply getting started. The onCreate work is the main thing that is important right now. This capacity is considered when your application is run and it is relied upon to make the view and do the activities the Activity is concerned with. As our Activity doesn’t actually do anything much the main thing onCreate needs to do is first call the acquired OnCreate strategy, super.onCreate, to do every one of the standard things and afterward utilize the setContentView capacity to choose the XML record that decides the format of the Activities screen.

The line:

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

is the most important of all and really the only one that actually does anything.ItgetstheresourceobjectRthatrepresen definedby the XML file created by the Layout Editor and makes it the current ContentView, i.e. it is what is displayed on the screen. In other words, it makes the

connection between the layout you have defined using the Layout Editor and stored in the XML file, and the user interface that appears on the device's screen.
You might be perplexed concerning why you altered an asset record called content_main.xml but the Kotlin is stacking an asset document called activity_main.xml The appropriate response is that to make broadening your application more straightforward Android Studio makes two design records,
activity_main.xml that makes the"standard" controls that

are shown and content_main.xml that you use to plan your custom UI. Obviously, activity_main.xml contains a reference to content_main.xml. This makes things more muddled for the fledgling however it is a disentanglement later.
We have more to find out with regards to the asset object R yet you can see that its fundamental job is to shape a connection between your Java code and the assets that have been made as XML records by the

Layout Editor. As this is everything our Activity does this is all the code we want. While I concur it is not really an“movement” it is sufficient to see the
fundamental blueprint of an Android application and to perceive how to make it run– which is our next job.

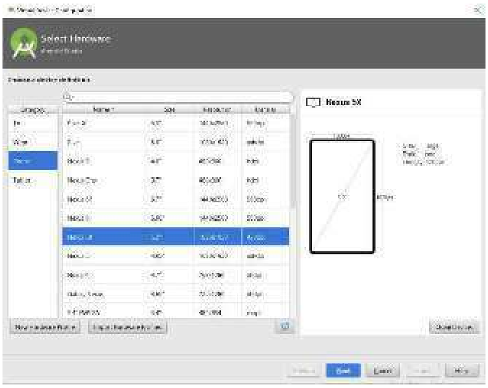
Getting Started with the Emulator

There are two unmistakable methods of running an Android application utilizing Android Studio. You can utilize the emulator or a genuine Android gadget. In the end you should find how to run an application on a genuinely associated Android gadget in light of the fact that the emulator just permits you to test a subset of things and it is slow. However, for the second running your first application on an emulator is sufficient to get started.
All you need to do is click the green run

symbol in the top toolbar– or utilize the Run,Run“application” menu thing. At the point when you do this interestingly it will take some time for the application to be aggregated. Ensuing runs are a lot quicker because of advancements presented in Android Studio 3.
When your application is fit to be ordered you will see a discourse box seem which permits you to either choose a running emulator or start one going:

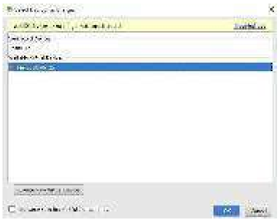
If no emulators are recorded then you should make one. Select the Create New Emulator button. This will introduce an exchange box where you can choose the gadget you need to test on.
The default is the Nexus 5 running Nougat API 25 and for a first test you should utilize this. Assuming that you really want different gadgets you can utilize the AVD (Android

Virtual Device) Manager to characterize them.
If you see a message on the right of the screen“HAXMis notinstalled” then it is a good idea to click the Install Haxm link just below. HAXM is a gas pedal that is utilized on Intel machines to make the Android Emulator run quicker. You needn’t bother with it yet it speeds things up:



You can acknowledge every one of the defaults in this previously run. You can screen the stacking of the imitating in the Run window which shows up naturally at the lower part of Android Studio.You might see a few admonitions show up– these can generally be ignored.

The entire interaction takes some time whenever you first do it. Later whenever the emulator first is as of now running and the moment run include attempts to re-run your application with the base changes:



Finally, make sure to delay until the Android working framework is stacked and you see the natural home screen before you begin to ponder where your application is. Even when it is loaded it is a good idea to give it a few seconds for Android Studio to notice it and to upload your app:

For our situation this isn ’t especially noteworthy– simply the words“Hi Android World!”, however when you think about the excursion voyaged it definitely should impress.
From this point you would now be able to alter the code or the format and run the application again to see the impacts of the changes. Assuming that anything turns out badly and you get wrecked then basically erase the project and make it again from scratch.
You actually have a ton to find regarding how to broaden the application and make it helpful, yet the experience has begun.

Summary

☐ Android Studio makes making Android applications significantly more straightforward than different methodologies and it is presently the main authority method for doing the job.

☐ An application has somewhere around one Activity and this characterizes a screen format, the View, and a conduct. An Activity doesn’t must have a UI, however much of the time it has one.

☐ To make a straightforward application utilize the Basic Activity format without any additional items selected.

☐ The screen design is constrained by a XML markup record, Main_Activity.xml, put away in the res catalog. There is additionally content_main.xml, which is the place where we place our custom UI controls.

☐ Android Studio gives a simplified Layout Editor that permits you to make a UI without working straightforwardly with the XML.

☐ The conduct of the application is constrained by a Kotlin record,MainActivity.kt for our situation,put away in the java organizer.You can alter the code in the Kotlin document straightforwardly in Android Studio. The code needs to load and show the design characterized in the XML file.

☐ To run an application you really want either an emulator based AVD or a genuine Android gadget associated with the machine.

☐ When you run the application you can choose which AVD or which equipment gadget is utilized to test it. When you initially start just utilize the default AVD, a Nexus 5.

☐ You can change and yet again run your application without restarting the AVD or any genuine equipment associated with the machine.

Chapter 2 Activity and User Interface

So you know how to make an Android

application, yet do you truly know how it functions? In this section we check out how to make a (UI) and how to attach it to the code in the Activity.
We found in Chapter 1 how to utilize Android Studio to construct the most straightforward application. In transit we found that an Android application comprises of two sections– an Activity and a View. The Activity is the code that accomplishes something and the View gives the (UI). You can consider this duality being like the HTML page and the JavaScript that rushes to cause it to accomplish something, or as a XAML structure and the code behind.
The key thought is that an Activity is the code that works with a UI screen characterized by the View. This isn’t exactly precise in that an Activity can change its view so one piece of code can uphold various perspectives. In any case, there are benefits to utilizing one Activity for each view in light of the fact that,for instance, this how the Android back button explores your application– from Activity to Activity. A complex application almost consistently

comprises of numerous Activities that the client can move between like pages yet a straightforward application can oversee very well with only one Activity. There is no rigid guideline regarding the number of Activities your application must have, yet it must have least one.
If you are contemplating whether an Activity can exist without a View the appropriate response is that it can, yet it doesn’t check out as this would pass on the client with no real way to cooperate with your application. Exercises are dynamic when their View is introduced to the user. It truly is an extraordinary improvement to think as far as an Activity as a solitary screen with a client interface.
If you need something to run without a UI then what you need is an assistance or a substance supplier which is past the extent of this book. It is also worth making clear at this early stage that an Activity has only one thread of execution– the UI thread – and you need to be careful not to perform any long running task because this would block the UI and make your app seem to freeze. That is, an

Activity can just do each thing in turn and this incorporates cooperating with the client. Assuming you compose a program utilizing a solitary action and it does a confounded estimation when the client clicks a

button then the movement can not react to any extra snaps or whatever occurs in the UI until it completes the calculation.
In many cases the entire motivation behind the Activity that is related with the UI is to take care of the UI and this is the thing that this book is for the most part about.
Also notice that making extra Activities doesn't make new strings. Just a single Activity is dynamic at some random time, a greater amount of this some other time when we consider the Activity lifecycle in detail. In this book we will focus on the single screen UI Activity since it is the most widely recognized application building block you will experience, and it is even where most complex applications start from.

The MainActivity

There is one movement in each undertaking that is assigned as the one to be dispatched when your application begins. Assuming you use Android Studio to make another Basic Activity application called SimpleButton and acknowledge every one of the defaults, the startup Activity is called MainActivity of course. You can change which Activity begins the application by changing a line in the application’s manifest.
The Manifest is a project file we haven't discussed before because if you are using Android Studio you can mostly ignore it and allow Android Studio to construct and maintain it for you, but it is better if you know it exists and whatitdoes.
The Manifest is put away in the application/shows registry
and is called AndroidManifest.xml.
It is a XML record that tells the Android framework all that it has to know about your application, including what authorization it needs to run on a device.
Specifically it records the exercises as a whole and which one is the one to use to begin the app.

If you open the produced Manifest, by double tapping on it, you will see somewhat way down the file:

```
<activity android:name=“.MainActivity”

android:label=“SimpleButton” >
```

This characterizes the Activity the framework has made for yourself and the lines just underneath this characterize it as the startup Activity:

```
<intent-filter>
<action android:name=“android.intent.action.MAIN”/>
<category android:name=“android.intent.category.LAUNCHER”/>
</purpose filter>

</activity>
```

Notice that the choice of which Activity starts the app has nothing to do with what you call it, i.e. calling it MainActivity isn't enough. For the second you can depend on Android Studio to care for the Manifest for you. Much of the time you will possibly have to alter it straightforwardly when you really want to address a blunder or add something advanced.

Inside the Activity

The produced Activity has one class, MainActivity, which holds the techniques as a whole and properties of your activity.
It likewise has three created methods:

☐ onCreate
☐ onCreateOptionsMenu
☐ onOptionsItemSelected

The last two are clearly associated with the working of the OptionsMenu, which is a significant point however one that can be overlooked for the occasion.Not movements of every sort need to have an OptionsMenu and you could even erase these strategies to help a choices menu. All three of these techniques are occasion controllers. That is they are called when the occasion that they are named after happens. The entire of an Android application is an assortment of only occasion controllers and their assistant functions.

The main strategy produced by Android Studio is onCreate.This is an occasion overseer and it is considered when your application is made and is the place where we do all the instatement and setting up for the whole application. It is likewise commonly the spot we show the application’s primary UI screen.

Let’s look again at the initial two lines of produced code for onCreate, which are the most important:

```
abrogate fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
```

The onCreate occasion overseer is passed a Bundle object called savedInstanceState. This is expected to safeguard state data between summons of your application and we will perceive how this is utilized later. For this situation no information has been saved thus savedInstanceState is invalid– however you actually need to give it to the acquired onCreate strategy.You will glean some significant knowledge more with regards to Bundle in Chapter 12.

The last guidance calls setContentView, which is a strategy that has various diverse over-burden structures. For this situation we pass a number that demonstrates which XML document depicts the design to be utilized for the view. The setContentView strategy utilizes this to make every one of the parts of your UI as characterized in the XML record. That is, this is the association between the format you made utilizing the Layout Editor and the design that shows up on the gadget’s screen.

It merits looking somewhat nearer at the way that the design document is determined on the grounds that this is an overall way that Android allows you to get to assets. There is an entire section about assets later yet it is as yet worth a presentation now.

The R object is built by the framework to permit you to get to the assets you have put in the asset catalogs.For instance,in onCreate the utilization of R.layout.activity_main returns a number worth, its id, that permits the setContentView technique to find the activity_main XML design document. Overall all assets are found through the R object, consider it a file of resources.

View and ViewGroup

So far so great, however it is critical to understand that what occurs next it that the XML record is delivered as a bunch of View objects. That is, Java protests that are largely sub-classes of the View object. The whole UI and illustrations framework is executed as a pecking order of parts got from the View class.

If you have utilized practically any GUI structure, AWT, Swing, XAML, and so on, this thought won’t be unfamiliar to you. For instance, a button is a class gotten from View and to make a button you should simply make an example of the Button class. You can obviously make however many buttons as you like essentially by making more instances.

This leaves open the topic of where the button shows up in the layout?

The response to this is that there are ViewGroup objects which go about as compartments for other View objects. You can set the place of the kid View objects or simply permit them to be constrained by different design rules, a greater amount of which later. You can select to make the whole UI in code by making and working with occurrences of View articles and this is an exhibited thing in Chapter 7.

So to be 100% clear all of the UI objects are characterized in code and each UI object, like a button, has a class with a comparative name that allows you to make the UI in code. Truth be told this is the best way to make the UI, yet there are alternate methods of determining it. Rather than composing code to make the UI you can indicate what you need in a XML document and afterward utilize provided code to show it. This is the thing that setContentView does– it peruses the XML record you indicate and makes protests that execute the UI.

This implies you could make the UI by physically composing a XML file that characterizes view items and how they home one inside another and depend on the framework to make the view object progressive system for you. Albeit this is conceivable, it is a lot more straightforward to utilize the Layout Editor to make the XML document and afterward permit the framework to make the items for you from the created XML record. That is, you can relocate a button onto the Layout Editor and it will naturally produce the XML expected to make it and determine where it is and every one of different subtleties you set.

That is, at the surface level there are three methods for making the UI: 1. You can compose code to produce the fundamental objects.
2. You can compose XML labels and utilize the framework to change the XML over to the equivalent objects.

3. You can utilize the Layout Editor to intelligently make the UI and create the XML record which is then changed over into the Java objects required. You can imagine this as:

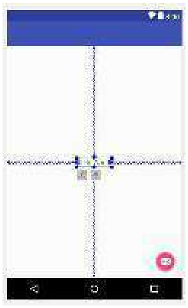
simplified format - > XML - > View objects

Being ready to work with the UI in an intuitive manager is one of the extraordinary benefits of utilizing Android Studio and, regardless of whether you know how to alter the XML design record straightforwardly, it’s anything but an element you ought to disregard. It is almost consistently a smart thought to utilize the format proofreader as the initial step and apply any changes, if essential, to the XML record later.

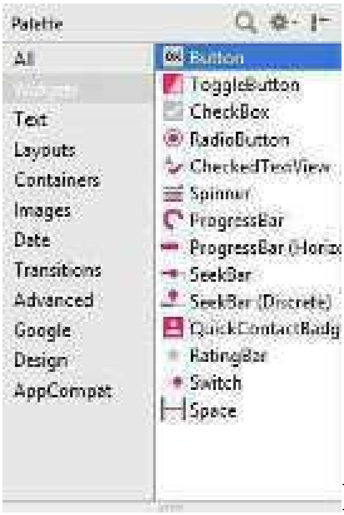
Creating Our First UI

To perceive how each of this fits together

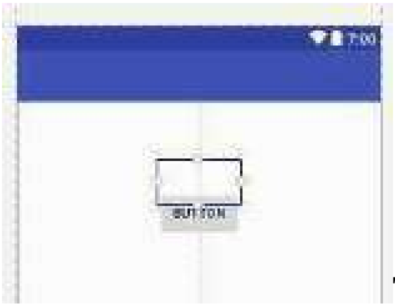
how about we add a button and a textView object. You most likely definitely realize that a button is for squeezing and you can figure that a textView is utilized to show the client some text.
First eliminate the Hello World text that is produced naturally when you make another clear Activity. Load the content_main.xml document by opening it from the task view. The content_main.xml is the place where you make all of your UI. There is another format document, however as clarified in Chapter 1 this essentially gives the standard components in a UI like the AppBar. Notice that on account of this you can see UI parts in the Layout Editor that you can’t alter - they have a place with the other format document. To eliminate“Hi World” you should simply choose it and press the erase key: Notice that there is a fix order, Ctrl-Z, assuming you erase something by mistake.



Next select the button in the Widgets segment of the Palette by tapping on it:



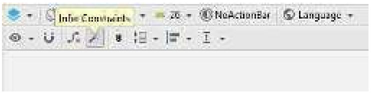
If you currently place the cursor over the plan region you will find that as you move it different arrangements are demonstrated by lines:



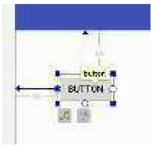
To situate the button essentially click and a full button, complete with the default inscription Button, will appear.

However, just dropping the button on the plan surface isn ’t exactly enough. Assuming you simply do this the button will be situated however with no situating imperatives. Assuming you have a go at running the program you will observe that the button here and there disappears on the emulator or genuine gadget.The explanation is that without any imperatives to its situating applied it ascends to the highest point of the screen and is taken cover behind other controls.
This conduct is normal for the default ConstraintLayout, a greater amount of which later. For the second we simply need to apply

a few requirements to the button.
The most straightforward arrangement is to tap on the Infer limitations symbol and let Android Studio work out how to position the button:

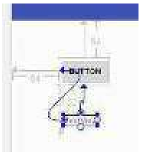


When you click this button imperatives are added by where the Button is situated. Right now precisely what requirements you wind up applying matters not exactly the basic truth that there are a few. In the screen dump underneath the button is obliged to be a decent separation from the top and right-hand side. Notice that you can situate the button and afterward click the Infer limitations button to set the requirements expected to keep the button in its location:



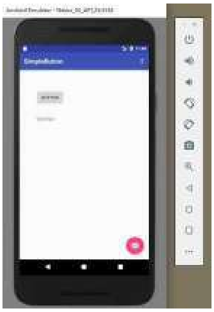
Now you have a button on the UI all set, we

should add a TextView Widget in the very same manner– click on its symbol in the Palette, position in the Layout Editor and snap to set the position. Notice that Android Studio gives situating data to assist you with adjusting parts of the UI. Again you really want to tap the Infer requirements symbol to situate the TextView.
The simplest thing to do is position the TextView where you want it and then clicktheInferconstraintsicontoset theconstraintsneededforthatposition:



Notice that Android Studio gives situating data to assist you with adjusting parts of the UI. The least complex way to deal with making a format is to put the parts where you need them and afterward click the Infer limitations symbol to apply imperatives to fix their position. By and by these requirements may not be by and large what you want however it gives you a beginning stage.

That’s the long and short of it and indeed making a total complex UI is only business as usual, simply a question of picking parts from the Palette and situating them on the plan surface.
If you presently run the program, by tapping the green Run symbol (allude back to Chapter 1 assuming you don’t have the foggiest idea how to do this) you will see your new UI:

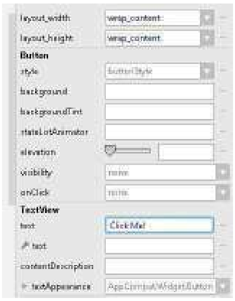


Obviously it sits idle, regardless of whether you can tap the button. The button clicks, however there is no code associated with the button click occasion to express what ought to occur, something we will manage very soon.

Properties & Attributes

Our next task is to change the caption on the button. Recall that objects have properties and methods.Things like caption text,background color and so on for UI widgets are represented as properties. You can change properties in code, or at design time you can use the Attributes window on the right-hand side of the screen.
What is the difference between a property and an attribute?
The reality is that all of the UI objects are code objects and quantities like text are properties of the object but when you specify a quantity in XML it is known as an attribute. You could say that you set attributes in the XML to set the properties of code objects.
If you select the button and examine the Attributes window on the right you will find the button’s text attribute. This currently contains the value “Button”. If you change

this to“Click Me!” and re-run the app you will see that the Button’s caption has changed:



layout_width	wrap_content
layout_height	wrap_content
Button	
style	buttonStyle
background	
backgroundTint	
stateListAnimator	
elevation	
visibility	none
onClick	none
TextView	
text	Click Me!
text	
contentDescription	
textAppearance	AppCompat.Widget.Button

You can set the initial properties of any of the widgets that you have placed in the UI. There are a great many properties and we need to spend some time looking at some of them. However, for the moment the important thing is that you see how easy it is to change a property using the Attributes window. As you might guess, the property that you changed results in a change in the XML file defining the layout. Recall that the general principle is that the Layout Editor creates the XML file that you could have created by hand without any help from the Layout Editor. In this sense, the Layout Editor doesn’t add anything to the process,other than being much easier.

Events

Now we want to do something when the button is clicked.Android supports a complete event-driven UI. So what we need to do next is define a method, or function, that is called when the button is clicked. Java’s way of implementing events is complicated but now much simplified in Java 8. Kotlin makes event handlers comparatively easy, but still slightly more difficult than in languages such as C#.

There are a number of different ways to specify an event handler, but the simplest is to use the Layout Editor to generate the XML needed for the system to hookup the event handler to the event. This is not an approach that you can use all of the time. It only works for the click event, but it gets you started.

If you have used something like Visual Studio,it is also worth pointing out that Android Studio doesn’t automatically create event handlers for you.In the case of Android Studio you have to create a function and then

assign it as the onClick handler.
Using the Layout Editor approach, method click event handlers are simply public methods of the current Activity with the signature:

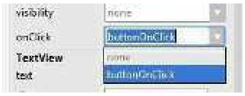
```
fun buttonOnClick(v: View){
```

Just in case you have missed or forgotten what a function's signature is: *A function's signature is the number and types of the parameters it accepts. Functions with the same name but different signatures are considered to be different functions.In this case the function takes a single parameter,which is a View, and returns Unit, i.e. nothing.*
You can call the method anything you like, but in most cases it helps to specify exactly what the event it handles is. In this case we want to handle the button's onClick event–which occurs when the user clicks on the button with a mouse or more likely taps on the button using a touch sensitive device.
Load the MainActivity.kt file into the code editor and add the following method:

```
fun buttonOnClick(v: View){

// do something when the button is clicked
```

}
This needs to be added directly following the onCreate method, or anywhere that makes it a method of the MainActivity class.

With all this code out of the way, now switch back to the Layout Editor, select the button, find the onClick property in the Attributes window and enter buttonOnClick:



At the mome nt the IDE doesn’t notice Kotlin methods that are suitable event handlers so you have to type them in, but this might well change in the near future. Notice that you don’t type in parameters, just the name of the method.
That’s all there is to it. You define your event handler with the correct signature and return type and set the appropriate onClick property in the Attributes window.
When it comes to other types of event you have to do the job in code– the XML/Layout Editor method only works for onClick.

Connecting the Activity to the UI

Now we have an event handler hooked up to the button click event we usually want to do something that affects the UI as a result. Let’s suppose that when the button is clicked we want to change the text displayed in the TextView widget to“I’ve Been Clicked!”. We can do this by changing the TextView widget’s text property to the new text. The only problem is how do we find the TextView widget in code? This is a fairly standard problem when you use a markup language to define a UI. The markup language defines widgets, or other UI objects, and the code has to have a way of making the connection to those UI objects. For example, in JavaScript you make use of the getElementById method to retrieve a DOM object corresponding to a particular HTML element. In Android we do something similar. First make sure you follow the idea that all of the XML generated by the Layout Editor gets converted into a set of objects, one for each component or View placed on the design surface. These objects have the same range of properties as you saw in the Attributes window and have methods to get other things done. All you need to do is find a way to reference one of them.

In the case of the View object that caused the event this is very easy as it is passed to the event handler as the only argument of the call. So if the event handler is:

```
fun buttonOnClick(v: View){
```

and the event handler is only hooked up to the button, then you can be 100% sure that v is the button object when the event occurs. If you want to change the button’s caption you could just use its text property to change its value.

Note: this is a Java property as all Android classes are Java classes. This means

they don’t have properties but have get and set methods which retrieve and modify *property values. Kotlin automatically converts such get and set properties to properties that can be used without reference to get and set methods. In other words in Java you would have to write v.setText(“New Text”); but in Kotlin you can write v .text=”New Text”. Kotlin automatically converts the assignment to a call to setText. So if you are a Java programmer get out of the habit of calling getter and setters and just use assignment.*

So, in principle, all you have to write is:

```
v. text=“I’ve Been Clicked!”
```

However, this doesn ’t work because v is declared to be a general View object which doesn’t have a text property– not all View objects have any text to display.

To use the Button ’s text property we have to cast v to its correct type, namely a Button.
Note:Casting is where you tell the system the type of the object you are working with. If classB is a subclass of classA then you can treat a classB object as a classA - after all it

has all of the methods and properties that classA does by inheritance.
However, if you want to make use of a property or method that only classB has then you need to cast the reference to the classB object to make its type clear.
For example, assuming classB inherits from classA:

```
val myObject:classA = classB()
```

creates an instance of classB but myObject is declared to be of type classA. This is fine but you can only access the methods and properties of the classA object.

However, if you try:

```
myObject.classBMethod()
```

then it will fail if classBMethod only exists in classB.
To use the classB method you have to cast myObject to its real type: `myObject as classB`
You can store a reference to the cast in a new variable:

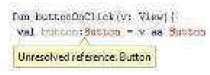
```
val myClassBObject:classB = myObject as classB
```

and then call the method:

```
myClassBObject.classBMethod()
```

or you can just do the cast on the fly at the cost of an extra pair of parentheses: `(myObject as` *classB).classBMethod()*

If you simply change the code to cast the v object to a Button object, i.e. (Button) v, you will discover that Android Studio flags an error by showing Button in red. If you hover over the red symbol you will see the exact error message:



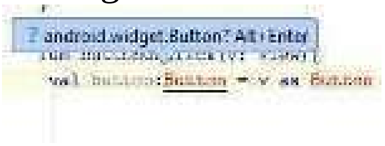
```
fun buttonOnClick(v: View){
  val button:Button = v as Button
```

Unresolved reference: Button

This is because you have used a class without importing it. You may see other classes, View, for example highlighted in red as which classes are imported by default depends on the exact project template you are using.

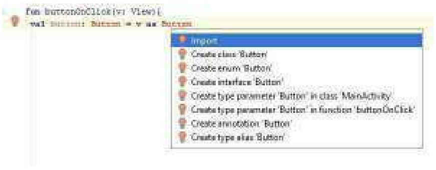
Any class that you use has to be listed at the top of the program in an import statement.Whenever you see a“Cannot resolve symbol” error message the most likely cause is that you haven’t imported the class.

This can be a tedious business but Android Studio has some help for you.If you click on the error symbol you will see a blue hint message:



```
? android.widget.Button? Alt+Enter
    val button:Button = v as Button
```

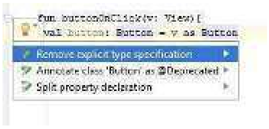
If you look at the hint message it suggests pressing Alt+Enter which is always good advice because it produces a list of possible fixes for the problem:



```
fun buttonOnClick(v: View){
    val button: Button = v as Button
```

Import
Create class 'Button'
Create enum 'Button'
Create interface 'Button'
Create type parameter 'Button' in class 'MainActivity'
Create type parameter 'Button' in function 'buttonOnClick'
Create annotation 'Button'
Create type alias 'Button'

You can implement the fix simply by selecting it from the list. In this case you have to add the class definition to the start of the program. import android.widget.Button
You can do this manually, i.e. you can type it in, or just select the first option. The import is added and in a moment the red errors disappear. If you hover over the corrected Button class name you will also see a light bulb:



```
fun buttonOnClick(v: View) {
    val button: Button = v as Button
```

Remove explicit type specification
Annotate class 'Button' as @Deprecated
Split property declaration

Android Studio offers you hints on improving your code even when there are no errors– look out for the light bulbs. If you click on this one it will offer to remove the explicit type specification which is perfectly reasonable as:

val button = v as Button

can be considered more idiomatic Kotlin with the use of type inference. However the second option is to add a @Deprecated annotation which is unlikely at best. Android Studio is full of hints and offers to make your code better at every turn– you don't have to accept them.

Now we have the button object we can use its text property:

button.text =“I’ve Been Clicked”

Remember: No need to use get and set. In Java you would have to write button.setText(“I’ve Been Clicked”);

The complete event handler is:

fun

```
buttonOnClick(
v: View){ val
button = v as
Button
button.text =“I’ve Been Clicked”

}
```

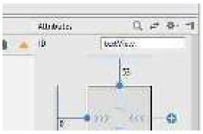
Now if you run the program you will see the button’s caption change when you click the button.
This is a common pattern in making your UI work – event handlers change the properties of View objects to modify the UI.
Notice that this approach only works if you know the type of the object that caused the event and called the event handler. If your event handler is

only connected to a single component then you do know the type of the object that caused the event. If it isn’t, or if you want to modify the properties of a View object that isn’t the subject of the event, then you have to find it.

This is exactly what we are going to do next.

Finding View Objects

Now suppose we want to do something to one of the other components in the View. In this case we have to find the object that represents the component without the help of the event handler’s argument.
For example how do we find the TextView that we placed below the button? Kotlin makes this very easy.
Every View object defined in the XML file has an id. You can assign an id manually but the Layout Editor automatically assigns an id to each component you place on the design surface. You can see the id of an object by selecting the object and then looking at the top right of the Attributes window:



You can also use this window to enter a new

value for the id i.e. you can use it to “rename” controls.
In this case you can see that our TextView object has been assigned an id of “textView”. In fact this is in reality an integer constant used to identify the View object in the layout but Kotlin makes use of this identifier to create an Activity property of the same name. The rule is that for every View object defined in the XML file that has an id Kotlin creates an Activity property of the same name. That is, in this case the id of the TextView is textView and so Kotlin automatically creates a property

```
this.textView
```

or, as you can generally drop this, just:

```
textView
```

When the program runs each of these properties is set to reference the View object created with the same id. That is, the textView property references the TextView object.
This makes working with the UI very much easier. For example, to set the text property of the TextView object all we have to do is:

```
textView.text=”You clicked my button”
```

As you type “text” notice that Android Studio will prompt you to import a definition derived from the XML file– accept this and you will be able to refer to all of the components in the XML file by their id.

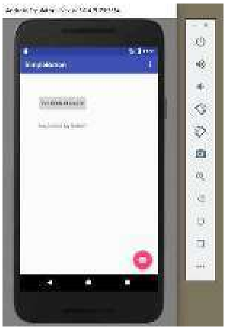
This adds the import:

import kotlinx.android.synthetic.main.content_main.* The complete event handler is:

```
fun
buttonOnClick(
v: View) { val
button = v as
Button
button.text = “I’ve Been
Clicked” textView.text = “You
clicked my button”

}
```

Notice that the button variable is a local variable that hides the button property that Kotlin automatically created for you from the XML ids.

If you now run the program you will see that you are informed twice of the fact that this very important button has been clicked:



You may think that this is all very small stuff and nothing like a real app, but this is how building a UI works in Android.
You now know how to design a single screen app using the widgets available in the Layout Editor’s Toolbox, how to hook them up to handle their click events, how to find the object that represents them and how to call the methods that modify them.
Apart from the fine detail of how each of the widgets works – radio buttons, checkboxes and so on– you now have the general outline of how to build a single screen app.

Summary

☐ An Activity is the unit of the Android app and it roughly corresponds to one screenful of user interface plus the code to make it work.
☐ In most cases you will create an Activity for each UI screen you want to present to your user.
☐ Only one Activity from your app is running at any given time. ☐ An Activity is single-threaded and runs on the UI thread. ☐ You can set which Activity starts the app in the Manifest. Android Studio sets this to MainActivity by default.

☐ The Activity has events corresponding to different stages in its lifecycle. The onCreate event is called when the app first starts and this is where you perform all initialization.

☐ You can also restore the app’s state from previous runs at this point. ☐ The Activity then loads a View or ViewGroup object to create its user interface.
☐ You can create View/ViewGroup objects in three possible ways: in code, using XML or using the Layout Editor to generate the XML.

☐ The Layout Editor is far the easiest way to create a UI. ☐ By opening the XML file you can use the Layout Editor to place widgets corresponding to View objects on the design surface. ☐ You can use the Attribute window to set the attributes of each widget.

☐ The XML file that the Layout Editor creates is used by the Activity to set its UI by creating objects that correspond to each of the View objects placed using the Layout Editor.

☐ When you reference a class that isn’t defined in the file, i.e. most of them, then you need to add an import statement to the start of the code.

☐ If you use Alt+Enter when the cursor is positioned within any word that is displayed in red then Android Studio will help you fix the problem.

☐ You can hook up onClick event handlers defined within the current Activity to the widgets using the Attributes window.
☐ An onClick event handler is just a public

function with the signature myEventHandler(v:View):Unit.

☐ The View object parameter is sent to the View object that raised the event. This can be used to access the properties/methods of the View object that the user interacted with.

☐ To access other View objects directly you can make use of the fact that Kotlin converts all of the id attributes assigned by the Layout editor into Activity property initialized to reference the corresponding View object.

Chapter 3
Building a Simple UI

By this point in you understand how the Activity and the View fit together to create a simple application, but the Android UI is more complicated than most because of its need to cope with a range of very different screen sizes and orientations. In this chapter we look at the problem of layout and working with the UI framework and on the way we’ll build a calculator app.

When building an Android app you will spend far more time than you could possibly imagine on perfecting the UI. So it is important that you master the basics so that you can move on to code that does more interesting things.

The learning curve with any UI framework is more or less the same. First you have to find out what constitutes an application that you can run i.e. where is the code? In Android’s case this is an Activity.

Next you have to work out how UI components are represented, how you can create them and how to hook up the UI with the code. In Android’s case this is a matter of a hierarchy of View objects and hooking up with the code is a matter of finding the objects representing each UI component and adding event handlers.

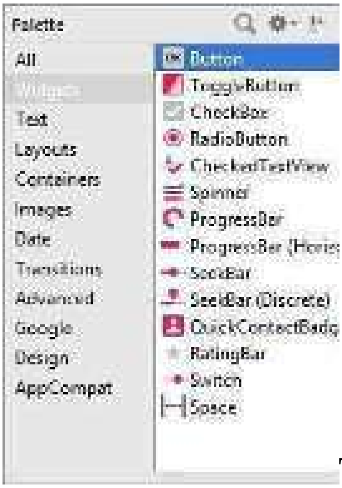
Once you have the basics you have to start exploring what components you have been provided with to build a UI. In general this varies from the extremely simple - the

Button, for example - to almost complete applications in themselves - the Listview, for example. It would take a long time to master all of them, but what most programmers do is make sure that they can use the basic components and then find out about the bigger more sophisticated components when needed. The good news is that once you know how one component, even the simplest, works then most of it generalizes to bigger more complicated things.
We also have to worry about how to lay out the UI – how to size and position sets of components. Android is particularly sophisticated in this respect because being a mobile operating system it has to contend with a wide range of screen sizes and even orientation changes while an app is running. This is not a simple topic and we will have to consider it in more detail later, but for the moment let’s just take a look at the easier aspects of screen layout.

Using the Layout Editor is the simplest and most productive way to work so let’s continue to concentrate on this method of creating a UI. This chapter is mostly about how to use the Layout Editor with the default layout and the challenges of creating a UI.

What's in the Palette

Start Android Studio and create a new simple basic activity project called UItest. This is going to be our UI playground for the rest of this chapter. Accept all the defaults, apart from selecting a Basic Activity, and
wait while the project is created. Make sure you have checked Include Kotlin support on the first dialog page.
If you now open the file content_main.xml in the app/res/layout folder then the Layout Editor will open and you will see the familiar rendering of the default layout.
Now it is time to look at the Palette in more detail:

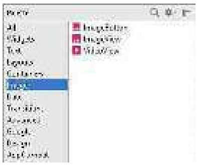


The top four sections of the Palette hold the most important UI components with a further five sections with more specialized ones.

1. The Widgets section contains the most frequently used components
– Buttons, Checkboxes and so on. This is the set of components you need to learn to use first.

2. Text Fields are a set of text input components which all work in more or less the same way.
3. Layouts are containers for other components that provide different layout

rules.
4. Containers are like mini-layouts in that you generally put other components inside them and the container“looks after” them:
5. Images and media are containers for specific types of resources such as images and videos.



6. Date and time are widgets concerned with date and time entry and display.
7. Transitions perform limited animation between different components.
8. Advanced doesn’t really mean advanced– more bigger and complex components such as the number picker.
9. Google is for services provided by Google and consists of MapView and AdView at the moment.
10. Design is a collection of more advanced controls such as tabbed pages.
11. AppCompat are controls that provide up-to-date controls for older versions of

Android.

The Button an Example

Where else should we start – the Button is almost the“Hello World” of UI construction. If you know how to work with a Button you are well on your way to understanding all of the possible components. The good news is that we have already met and used the Button in Chapter 2 and discovered how to work with it in code. However, there is still a lot to find out.
Generally there are three things you need to discover about using any component.
1. How to make it initially look like you want it to. This is a matter of

discovering and setting properties using the attributes provided in the Layout Editor.

2. How to modify the way a component looks

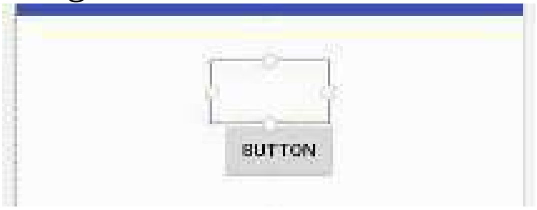
at runtime. This is a matter of finding out how to work with properties in code.

3. How to hook up the events generated by the component to the code. Setting properties sounds easy, but there are different types of properties and these have different appropriate ways of allowing you to interact with them. The first thing we have to find out about is how to position a component.

Positioning– the ConstraintLayout

Before you continue with the project select and delete the default “hello world” text– it makes trying things out easier to have a clean design surface.
There are two views of your layout you can opt for– design and blueprint. The design view looks like your UI when it runs on a real device. It is useful for getting an overall feel and impression of what the UI really looks

like. The blueprint view shows you the UI in skeleton form. This makes it quicker and easier to work with when you are positioning things. You can view both at the same time but this is mostly a waste of screen space. Some users prefer the blueprint view because it seems to be light weight others prefer the design view because it is complete. Use whichever you find workable but there is no doubt that the blueprint view is faster and suffers from less lag as you try to position components.
Next click on the Button in the palette and drag it onto the layout.

Notice the way the layout information changes as you move the button around the design surface:


It is important to realize that this positioning information is only to help you locate UI components relative to the existing layout.With the new constraint layout where you drop a component has no effect on where is will show when the app is run.In fact if you simply drop a UI component on the design surface without applying any constraints then when you run the app the component will drift up to the top left-hand corner.In fact things are slightly more complicated because there is a facility that will automatically add constraints when you drop the component that realizes the alignments shown at the moment you drop.However this can be turned off as we will see.The key thing to realize is that the alignments displayed are not constraints.

Using the constraint layout the only thing that affects where a component is displayed are the constraints you apply.

So how do you apply a constraint?

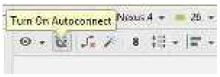
You can have the Layout Editor suggest them for you automatically or you can apply them manually.

Automatic constraints are the easiest to use and there are two ways to get the editor to apply constraints dynamically:

1. Autoconnect mode
2. Infer Constraints

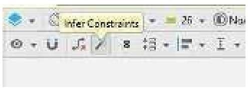
They do slightly different things and you need to learn to make them work together.

The Autoconnect mode seems the most useful when you first meet it but in practice it can be confusing and error prone.However, it is worth trying out. To turn Autoconnect on simply click the Autoconnect icon on at the top of the Layout Editor:



With Autoconnect on,when you place a component on the design surface the editor attempts to work out suitable constraints based on where you drop the component. If it fails to apply any constraints then it will try again if you drag the component to another location. It will only apply constraints to satisfy the alignments that are indicated in the Layout Editor. If you drop a component at an arbitrary point on the screen with no alignments indicated then no constraints will

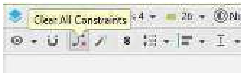
be applied.
Once it has applied constraints, these are not altered if you modify the component. For example if you drop a component in the middle of the screen then constraints are applied to place the component 50% of the way down the screen and 50% from the left. If you then move the component the percentages are changed to position it but the actual constraint use wont change to anything more appropriate if you move it to say the top left hand corner. Once a constraint has been applied its type will not change. The Infer Constraints option is actually easier to use and reasonably effective. All you have to do is position all the components where you want them and then click the Infer Constraints button:



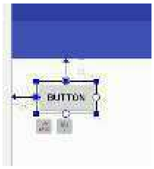
When you do this constraints will be calculated based on where each component is. This gives you more time to position the components and you can always delete all the constraints and click Infer Constraints again to recompute them. Only constraints that are necessary to fix the position of a component are added – existing constraints are not modified.This means you can use Infer Constraints to make sure that your layout has enough constraints to make it work– if none are added it was OK.

Also notice that once Infer Constraints has added a constraint the type of constraint wont change if you drag the component.

A good strategy is to switch off Autoconnect and use Infer Constraints every time you place a new component on the design surface. This allows you to build up a layout one component at a time and check each new set of constraints. You can then modify the newly added constraints and move on to the next component.
Where things get difficult is when you have a multi component layout and need to make radical changes. Often it is easier in this case to delete all of the constraints using the Clear All Constraints icon, and start again:



For a component that has been placed on its own close to the top and lefthand edge of the screen, constraints will be added that fix its distance from the left and top:

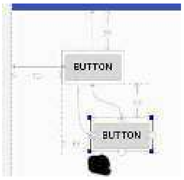


From now on when the button is displayed it will position itself at 36 from the left and 33 from the top.These constraints are applied no matter what the size of the physical screen is. This is the sense in which they are constraints rather than absolute positioning.

It is worth knowing that the actually positioning is achieved by setting the button’s margin properties to 36 and 33. The constraint layout may be the most sophisticated of the layout components but it positions other components using properties that have been in use from the start of

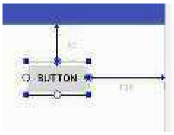
Android.

You can see that things aren't quite as simple as positioning relative to the edge of the screen if you place a second button on the surface and move it closetothe first.NowifyouclicktheInferConstraintsbuttont thatareappliedaretakenrelative tothefirstbutton:



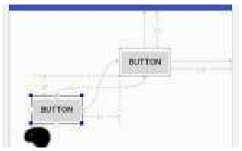
You can see that in this case the second button is positioned 88 from the left edge of the first button and 49 below it. Once again these constraints will be obeyed no matter how large or small the physical screen the UI is displayed on.What is more if you move the first button the second button will keep its position relative to the first. In fact if you click and drag the first button, the second will move with it.
If you play with the positioning in the Layout

Editor you will quickly get the idea.
The big problem with inferring constraints is that the system sometimes gets it wrong. Rather than let the Layout Editor infer constraints incorrectly you can place them manually. All you have to do is drag a line from the constraint circles on the component to the edge you want to use as a reference point.
For example to position the button relative to the right side of the screen simply drag the circle on the right of the button to the right side of the screen and then move the button where you want it:

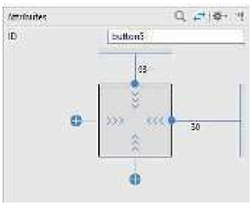


There are quite a few different types of constraint that you can apply and we will go into these in detail in Chapter 5 where we examine layouts in depth. For the moment this almost all you need to know.
Constraints can be applied automatically or manually and they often set the distance from

some point on the component to a point on some other component.
For example, if you want to place a component relative to another then simply drag the constraint point on the first to an edge of the second. In the case shown below manual constraints have been applied because constraint inference would have placed constraints on the left-hand edge of the screen:



It is also worth knowing at this early stage that the Attributes window can help you in a number of ways. The first is that it has a diagram that shows you the currently applied constraints, and lets you modify them and set distances exactly:



If you click on one of the constraint lines then you can enter a value for the distance. You can also click on the dot and delete the constraint completely or on the + and add a constraint.

As you might guess, there are attributes which let you set the location and margins for each position using the editor.
To see all of the attributes you have to click on the View all attributes icon at the top right of the Attributes window– the double arrow. The attributes listed in the window are not a simple representation of the properties that you might set in code to achieve the same results.They are organized and simplified to make using the Layout Editor easier. For example, if you look in the fully expanded Attributes window you will see:
Layout

Top
which gives the distance from the top edge and

Contraints
top_toTopOf

which specifies the object that the positioning is
relative to. If you look in the XML file you will see:

```
app:layout_constraintTop_toTopOf="parent"
android:layout_marginTop="348dp"
```

All layout attributes start with layout_name where the name gives the positioning affected. The mapping from the Attributes window to the XML properties is fairly obvious.
It is also worth understanding that in allowing you to set the position of a component by simply dragging it to the location you want, the Layout Editor is working out how to set multiple properties correctly.In the example above it sets the top_toTopOf and MarginTop attributes.You could do this manually to get the same effect but the Layout Editor does it simply from where you have positioned a component.
This is why it is easier to let the Layout Editor set the attributes for you. The ability to set which other component to position against i.e. which component to set the position

relative to means you can build sets of components all aligned to one that is aligned to the container, so that they all move together as a group.
You can also set components to align with the left and right side of the screen and allow for rotation from portrait to landscape. You can also set constraints which divide the available space up by percentages – more of this in Chapter 5.

The constraint layout is the preferred layout component.The reason is that it is fast and efficient compared to the usual alternative of putting one layout inside another.Unless you have a good reason not to, it is the one to use.

However, all this said it is very easy to get into a complete mess with the layout in the Layout Editor. If components go missing then chances are they are on top of each other.The easiest way to sort this problem out is to go to the Attributes window and manually reset one of the positioning properties.

It is helpful to notice the following:

☐ Use positioning relative to another component if it makes logical sense. That is if you have a text entry component then it make sense to position its Accept Button relative to its right-hand side.

☐ If you position everything relative to the container parent then you effectively have an absolute layout.

☐ If the screen size changes then it is possible that components will overlap one another if there isn’t enough space. Always make sure your layouts have plenty of unnecessary space.

☐ A good strategy when working with a group of components is to pick one that you position relative to the container, then position all other components relative to it to ensure that you can move the group and keep alignments.

☐ Remember that some components can change their size as well as location and this can modify the position of components positioned relative to them.

The Constraint Layout component is used by default but you can change this for any of the other layout components– Relative, Linear, Table, Grid and Frame. There are also other container components which can be used in place of these standard layout components.

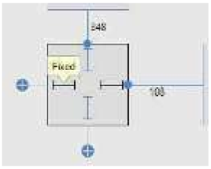
One thing worth knowing at this early stage is that components have layout properties that are provided by their container,so the set of properties that we have looked at in connection with the Constraint layout component are unique to it. That is if you use another Layout then you have to learn its layout properties from scratch.

Again this is another topic we have to return to.

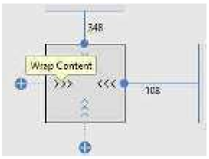
Sizing

In comparison to positioning, sizing a component is almost trivial. All components

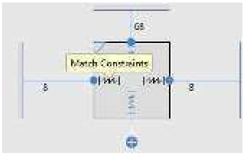
have a Height and Width property and these correspond to their drawn height and width when the component is actually rendered on the screen.
You may have noticed that there are what look like sizing handles in the corners of the components that you place in the Layout Editor. If you drag any of these, the component will change its size. However, the components content will not be resized. All that happens is that the content has some space around it.If the content gets bigger or smaller the component stays the same size.If the content it too big then it won’t fit in the component.
You can see the space is fixed in the Attributes window. The lines shown insidethecomponentrelatetothesizeofthecompc
its content. Straight lines indicate a fixed size:



An alternative to fixed size components is to allow them to automatically resize to fit or wrap their contents. To set this all you have to do is click on the inner straight lines which change to <<< to suggest springs which allow the component to change its size:



The third option is Match Constraints which lets thecomponent change it size to fill the availablespace.



This auto-sizing behavior is set by the layout_width and layout_height properties. You can modify this by typing in an exact size e.g. 100dp into the value box next to the property.

Also notice that you can set a maximum and a minimum size for most components.

Setting a maximum will result in the content being truncated if it doesn’t fit. Setting a minimum is often the best plan because then the component will increase in size when necessary.

The Component Tree

If you look to the bottom left corner of Android Studio in Design mode you will see a window called Component Tree. This is almost self explanatory and hardly needs explanation other than to draw your attention to it.Notice that you can see the tree structure of the layout starting at the Layout container. You can see that the default layout is ConstraintLayout and you can see all of the other components correctly nested within their parent containers:
Notice that you can select and delete layout elements directly in the tree. This is useful when you have a complex layout that has gone so wrong that you are finding it hard to select components reliably. You can also

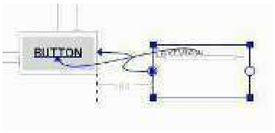
move elements around in the tree to change the way that they are nested.

A Simple Button Example – Baseline Alignment

As a simple demonstration let ’s first place a Button on the Layout Editor and use the Infer Constraints button to apply some constraints. If you find it difficult get a rough position and then enter the exact margins then always remember that you can move to the Attributes window and enter them directly.
Next place a TextView widget on the Layout Editor.
In this case the alignment we want is for the text to be on the same line as the text in the Button. This is a baseline alignment and one of the more sophisticated alignments but if you have the Layout Editor zoomed so that you can see the full area of the screen the

chances are all you will be able to do is align to the top or bottom of the Button.
If you look at the TextView’s representation in the Layout Editor you will see that there are two small icons below it. The first removes all of the constraints from the component, and the second makes the text baseline, a small elliptical box, appear inside it:
This represents the text in the TextView and you can use it to set up a baseline constraint.

All you have to do is select the TextView and hover the cursor over the elliptical box until is highlighted. Next drag from the elliptical box to the baseline of the text in the button, or any component that you want to align the baseline with:



That ’s all you have to do.After this the text in the button and the text in the TextView will share a common baseline.If you move the Button then the TextView will move with it.If you want to remove the baseline alignment then all you have to do is select the TextView, hover over the elliptical box, and click when the remove baseline constraint appears.

Orientation and Resolution

One of the biggest challenges in creating apps for Android is allowing for the range of screen sizes and orientations. Mobile app development is distinctly different from desktop development because of this need to deal with differing screen sizes. Even when your app is running on a fixed device, the user can still turn it though 90 degrees and change the screen dimensions in a moment and then turn it back again. So your app has to respond to these changes. Fortunately Android makes is easier for you by providing facilities that specifically deal with resolution and orientation changes, but you need to come to terms with this situation as early as possible in your progress to master Android programming.

Android Studio has lots of tools to help with the problem of variable layout. For example, it lets you see your layout in a range of devices, orientations and resolutions. This makes it so much easier.

We can test our layout in landscape mode simply by selecting the Landscape option:
The relative positions that you set in portrait mode are applied in landscape mode without modification and so it is important that you work on their specification so that the UI looks good in either orientation. If this isn’t possible then you can provide a separate layout for each orientation. You can also see what your UI looks like on the range of supported screen sizes by selecting from the list of devices that appears when you click on the Device In Editor button:
For simple layouts this is probably enough but Android has more up its sleeve to help

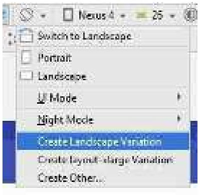
you work with different sized screens. In particular you can make use of“fragments” and create a separate layout for each screen orientation and resolution. Fragments are advanced and the subject of a separate book,*Android Programming: Mastering Fragments & Dialogs.*On the other hand, using different layouts for each orientation and resolution is fundamental to Android UI design and you need to at least know it exists at this early stage.

The idea is quite simple– provide a layout resource for each orientation and resolution you want to support. The system will automatically select the resource layout needed when the app is run on a real device. How this works is simple, but Android Studio has some helpful features that shield you from the underlying implementation. It is a good idea to first discover how it works and then move on to see how Android Studio does it for you.

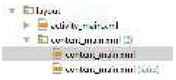
If you create a new layout file called activity_main.xml, i.e. exactly the same name as the portrait layout file you have already created, but in the folder

format land this will be utilized consequently when the client pivots the gadget into scene mode. That is, the design document utilized in any circumstance is dictated by the name of the envelope it is put away in.With this new design which applies just to a scene direction you would now be able to make a format that may be utilized in this orientation. What happens in general is that when your app starts the system picks the best available layout file for the screen according to what has been stored in theresourcefolders,i.e.layoutforportraitandlayo landforlandscape.. When the gadget is pivoted the framework restarts your application and burdens the best accessible design. In the event that you just have a picture format then it is utilized for all directions, yet assuming you additionally have a scene design document then it will be utilized for scene mode.
What has recently been portrayed is by and large what occurs.To recap, there are organizers for every goal and direction and you can make different forms of format assets all with a similar name. The framework

consequently picks the organizer to use to stack the design fitting to the equipment at runtime.
Android Studio, be that as it may, attempts to introduce this to you in a less complex manner. The undertaking view conceals the way that various organizers are utilized and essentially shows you the various adaptations of the asset records with a qualifier affixed - (land) for a scene document etc. Let's perceive how this works.
To add a scene format to the button and message design given above just utilize the Orientation in Editor button:

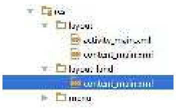


Initially the picture design is utilized as the layout for the scene format. Ensure you are altering the content_main.xml document before you make the scene format dependent on it. This guarantees that this is the document that is utilized as the layout for the scene form of the file. The scene variant of the record will contain all of the UI parts that the format record does. This implies that your code can collaborate with it in the standard manner without realizing that it is working with an alternate design. All the code thinks often about is that there is a button or there is a textView– not where they are put for sure direction is in use. If you analyze the Project window you will see the new design document listed:

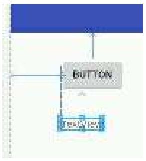


You can see that there are currently two content_main.xml records and one has (land) added later its name. As currently clarified this is an improvement that Android Studio offers you.

If you change to the Project Files view you will see that truth be told the new format asset is in another registry, design land:



Android Studio 's view is more straightforward, however when you really want to realize where records are truly put away this view is useful.
At first the new scene design is as old as picture design, yet you can alter it to make it more reasonable for the direction. For instance, we should put the text under the button in scene mode:



Now assuming you run the application utilizing the test system you will at first see the picture screen, however assuming that you turn the test system to scene mode utilizing the menu to one side, you will see a delay where the representation format is apparent and afterward it will change to the scene layout.
If you view the application on a scope of

gadgets you can pivot to scene and see your custom scene layout.
Overall the best arrangement is to make a total picture design and produce a scene format as late as conceivable in the turn of events so you don’t need to rehash UI tweaks. Also notice that the auto-exchanging of designs causes a possible issue. At the point when the format is exchanged your application is restarted and this implies that it can lose its present status. It is as though your application has quite recently been begun by the user.
To take care of this issue we want to investigate the application lifecycle and find how to safeguard state, which is shrouded in Chapter 13.

You can likewise utilize similar way to deal with supporting distinctive screen goals. The thought is something similar– various format XML documents are given, one for every goal, and the framework consequently picks which one to utilize. This is somewhat more convoluted than managing direction since you may well need to give diverse goal

pictures to make the design look great. Favoring this later we have taken a gander at assets and asset management.
This is only our first glance at a specific part of application improvement that makes Android more confounded than producing for a decent screen size and it turns out not to be so exceptionally troublesome as it would first appear.

A First App– Simple Calculator

To act as an illustration of building a basic application utilizing the ConstraintLayout and the Layout Editor, how about we fabricate a Calculator App. A portion of the procedures utilized in this application are a little past what we have covered up to this point however you ought to have the option to follow and the time has come to show what more than a solitary button application looks like.

This is an exceptionally straightforward mini-computer - everything it does is structure a running total. It has only ten numeric buttons and a showcase. It takes the qualities on every one of the buttons and adds them to a running aggregate.There are no administrator buttons,add,deduct or clear
– however you could add them to expand its functionality.

Start another Basic Activity project called ICalc or anything you desire to call it. Acknowledge all of the defaults.

The rule of activity is that we will set up a framework of ten buttons. Set each button to 0 through 9 as a text name. We are then going to allot similar onClick overseer to every one of the buttons. All it will do is recover the text inscription appearing on the button,convert it to a whole number,add it to the running total, and afterward store it back in the TextView in the wake of changing over it back to a String.
Put essentially, when a button is tapped the occasion overseer is called which recovers

the button’s name as a digit and adds it to the running all out on display.

Code

So the code is decently easy.
We really want a private property to keep the running all out in: hidden var complete = 0
Next we want an onButtonClick work which will be utilized to deal with the onClick occasion for the buttons in general.Allude back to Chapter 2 assuming you don’t be aware of straightforward occasion handling.

The button that the client clicks is passed as a View object as the main contention to the capacity and we can utilize this to get its text caption: fun

onButtonClick
(v: View) { val
button = v as
Button
val bText = button.text.toString()

Now that we have the button ’s subtitle, 0, 1, 2, etc, we can change it over to a number and add it to the running total:

val esteem =
bText.toInt()
absolute += value

Finally we set the TextView's text property to the all out changed over to a String:

textView.text= total.toString()
}

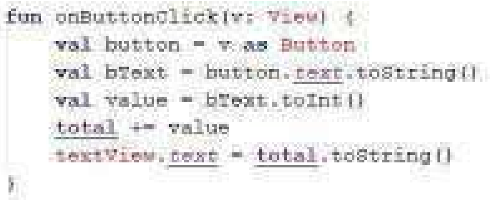
The total occasion overseer is:

private var complete = 0

fun
onButtonClick
(v: View){ val
button=v as
Button
val
bText=button.text.toString()
val value=bText.toInt()
total+=value
textView.text= total.toString()

}

Put this code inside the MainActivity class as one of its strategies, for instance right toward the end not long before the last shutting }. When you enter this code you will see a considerable lot of the classes and techniques in red:



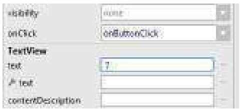
```
fun onButtonClick(v: View) {
    val button = v as Button
    val bText = button.text.toString()
    val value = bText.toInt()
    total += value
    textView.text = total.toString()
}
```

This is on the grounds that they should be imported to be utilized by the task. You can do this physically, adding the essential import articulations toward the beginning of the record, however it is a lot more straightforward to put the cursor into every last one of the images in red and press Alt+Enter and select Import Class if fundamental. This ought to be turning out to be natural by now.
Notice that the blunder hailed in textView can’t be cleared– it is a genuine mistake. We haven’t at this point characterized the UI and textView doesn’t exist. It will once we make the UI, so for the second overlook the error.

Layout

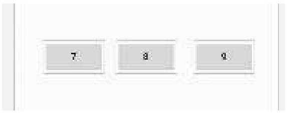
Now we direct our concentration toward the design. Open the format document, erase the default text and spot a solitary button on the plan surface at the upper right of the screen with a little space. We want to alter its text characteristic and we can utilize an easy route to get to it. Assuming that you double tap on

it you will be moved to the text property window where you can enter 7 as its text attribute.
While in the Attributes window observe the onClick quality. Set this to onButtonClick, the occasion overseer you have only written:

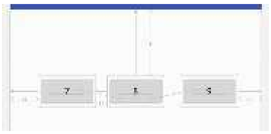


visibility	none
onClick	onButtonClick
TextView	
text	7
text	
contentDescription	

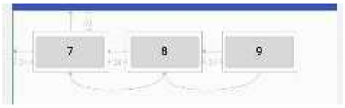
To make a network of buttons you need to rehash this interaction nine additional occasions. You could do this by hauling each button onto the plan surface and physically set the requirements expected to find every one comparative with the buttons previously positioned. Notwithstanding, we can utilize the one Button we have as of now made to make the rest utilizing duplicate and paste. Select the button in the Component Tree and utilize the Copy order and afterward glue two additional Buttons onto the plan surface. The duplicates will be put on top of the main Button so drag them to shape a harsh line of three Buttons. Try not to stress at this stage over definite situating, we just need to do this later the requirements are set up. Change the text property of each button to understand 7, 8 and 9 respectively:



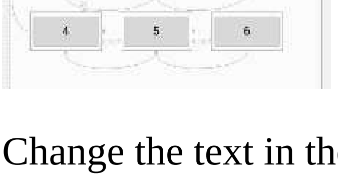
If you attempt to utilize Infer Constraints you will find that it works yet the imperatives applied are not really as consistent as you may want. For this situation the requirements are sensible however notice that the upward place of the line is set by the center button and the even situation by the first button:



There are numerous ways of setting the limitations yet it is helpful to have the option to situate the whole cluster of buttons utilizing the upper left-hand button. So position it with imperatives to the top and left of the screen. The second button in its column can have a limitation set to situate it to the right of the principal fasten and adjust its base to the lower part of the first button. The third button in the line has similar requirements as the button on its left:

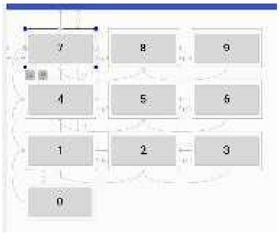


Now is additionally a fun opportunity to situate the buttons precisely either by hauling or by utilizing the Attributes window. Continue this example with the following line, don't attempt to reorder a
whole line on the grounds that the limitations become convoluted. Reorder
three additional buttons. Apply requirements concerning the main column,
however for this situation position the primary button in the line comparative
with the upper left button:



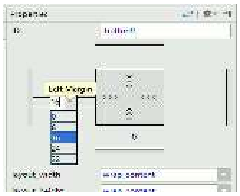
Change the text in the buttons to 4, 5 and 6 as

shown.
The last line is something similar yet its first button is situated comparative with the principal button in the subsequent line. At long last add the solitary zero button and position it comparative with the principal button in the third row:



Make sure that you change the text of the last four buttons to 1, 2, 3 and 0 respectively. Check the limitations cautiously as it is extremely simple to commit an error.
The rule is that the primary button in each line is situated upward by the
button above and different buttons in the line are situated comparative with
the first button.
This is a ton of work!
Nowwe need to make thegrid moreregular. Select each buttonin turn and

make sure its caption is correct and that it has the onclick event handler set.
Thenforeachbuttonset thehorizontal margins usingtheAttributes window
to 16 horizontally and to 0 vertically. The only exception to this is the first



button in each row that needs to be set to 0 horizontally and 16 vertically - if you think about it then it should make logical and fairly simple sense:

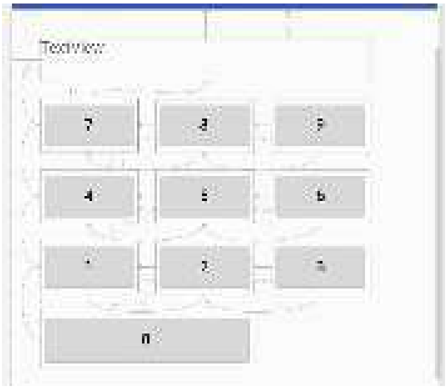
Now you have a matrix of buttons accurately positioned.
Try to ensure that each Button you add is comparative with one of

different Buttons and ensure that you have set them all to a similar onClick overseer. You can make sure that you have done this effectively by hauling one of the Buttons and

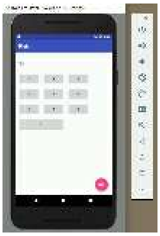
check whether they all follow! Assuming a couple don’t then take a stab at repositioning them comparative with one of different Buttons. Assuming you find any situating troublesome zoom in.
If you get into a wreck erase every one of the limitations and start over.

To make this look more like a keypad select the 0 key and size it so it possesses a twofold width.
Finally add a TextView at the top. You can situate it comparative with the top line and the 7 key and afterward size it so it traverses the total arrangement of buttons. To do this you should erase the limitations on the 7 key and set its top comparative with the lower part of the TextView and its left to agree with the left of the TextView.

Don't forget to position the TextView relative to the top of the screen:



Now you ought to have the option to move the whole square when you move simply the TextView to a new location.
It is great to realize that you can fix an activity by utilizing Ctrl-Z. In the event that you find the Layout Editor too hard to even consider working with to make this format, and it tends to be troublesome, you may like to set the design credits utilizing the Attributes window.
Select the TextView and utilize the Attributes window to change the text dimension to 18p or more – you will think that it is under the textAppearance field.
Finally run the program in the usual way and you will be please to discover your first app does actually calculate things!



Try pivoting the emulator to see what it resembles in scene mode. Not terrible yet as cautioned, as the application restarts you lose any computation you were performing since the application is restarted when the direction changes.
If you can, give it a shot on a genuine Android gadget. This allows you to discover what it truly feels like. Likewise examine what the format resembles on a scope of screens and orientations.
This isn't a very remarkable adding machine, yet you could cause it into a great one.

☐ Right to adjust the text in the display
☐ Add a + and– button and execute the action
☐ Allow the client to enter multi-digit numbers

☐ Add a decimal point key
☐ Add * and/keys
☐ Add an unmistakable key

Summary

☐ You can make the XML design document manually, however utilizing the Layout Editor is easier.
☐ It is as yet helpful to know how the XML record functions so you can alter it when the Layout Editor lets you down in some way. ☐ The Layout Editor alters the manner in which parts examine the Layout mode by changing a part’s attributes.
☐ How you position a part relies upon the Layout you are using. ☐ ConstraintLayout allows you to situate parts comparative with one another or to the container.
☐ The Layout Editor might change numerous qualities to situate the part where you need it.
☐ Use the Component Tree to choose parts that are hard to choose in the Layout Editor.

☐ You can utilize the Attributes window to choose and straightforwardly set any attribute.
☐ If you see as situating or measuring troublesome in the Layout Editor take a stab at zooming in. ☐ You can utilize Android Studio to see what your application resembles on different screen sizes and orientations.

☐ Different directions and goals can be obliged by making extra design records all with a similar name. The framework will pick which one to use at runtime.

☐ You can reorder parts in the Layout Editor to rapidly develop rehashed UI designs.
☐ A solitary occasion controller can be appended to numerous components.

Chapter 4
Android Events

Working with Android Studio makes

assembling the UI simple with an intelligent supervisor, yet you actually need to discover how to deal with the things it isn't exactly so great at. We currently need to discover how to function with any occasion, not simply onClick. Despite the fact that most controls can be utilized effectively utilizing only the onClick occasion, there are different kinds of occasion that can't be dealt with basically by setting an onEvent property in the Attribute Window.

In this section we examine profundity at Android occasion taking care of. As in the past, the accentuation will be on utilizing Android Studio and the Layout Editor to get as a large part of the work done as possible. Fortunately Kotlin's help for passing capacities as boundaries makes it all a lot simpler. In any case, this doesn't imply that you can totally overlook the way that Java handles the issue. Kotlin needs to work with libraries written in Java and depend in transit of executing occasion controllers.You don't need to dive excessively deep,however you do have to know how Kotlin utilizes the Java object-arranged occasion mechanism.

How Java Passes Functions

If you would rather not know how things work you can avoid this part until you stall out with an occasion overseer and afterward return and read it.
Events in Java are slightly more complicated than other languages because originally everything in Java was an object and the only way a function can exist is as a method, i.e. as part of an object. When you need to set up an event handler you need to specify the function to be called when the event happens. Youcan’tpass a function as a parameter to another function as Java doesn’tsupportfunctiontypes.
The solution is to definean Interface that has one abstract method–a SAM or Single Abstract Method. The abstract method is the function that will be used as the event handler. To pass it you have to create an instance of the Interface,i.e. an object that has

the method defined, and pass it.

An interface resembles a class yet it basically characterizes the capacities that a class needs to help.Assuming a class acquires or executes an interface then you need to compose code for each capacity characterized in the interface.You can consider an interface a determination for a rundown of capacities you need to write to execute the interface.Both Kotlin and Java support interfaces.

It merits saying at this beginning phase that not all occasion overseers in the Android SDK are carried out as SAMs.

Some are classes or interfaces that have more than one unique technique. That is they are objects that wrap up a bunch of related occasions. Both Java and Kotlin have furnished simpler methods of working with SAMs, yet these don’t work when the item being referred to has numerous theoretical strategies. At the point when you experience the present circumstance you need to make a full case of the class being referred to– fortunately there are simple methods of doing this as well.

Android Events

How occasions are executed in Android observe a guideline pattern. Any item that can produce an occasion will have a seteventListener technique which can be utilized to join an onEventListener item to the occasion overseer list.

The name of the strategy and the article incorporates the name of the occasion. So there is a setOnClickListener strategy which takes an OnClickListener object as its just parameter.

Notice that this naming framework is altogether show and not piece of the Java language.
The OnEventListener items are completely gotten from an interface of a similar name which characterizes a solitary conceptual technique which goes about as the occasion handler.

For instance, OnClickListener is an interface which has the single strategy onClick(View v) defined.
To utilize it as an occasion overseer you need to make an example that carries out onClick and afterward add it to the occasion controller list utilizing setOnClickListener.
In Java this is a straightforward yet drawn-out process, assuming you utilize the central method of carrying out an occasion controller, that includes a ton of standard code. First you would need to make another class that executed the Interface. Then you would have to create a new instance of this class and pass it to setOnClickListener.

The Kotlin Object

In Kotlin you don ’t need to initially make a class and afterward make a case, you can go straightforwardly to the new article you require utilizing the capacity to pronounce objects directly.
You can announce an item utilizing very

much like language structure to proclaiming a class. For example:

```
object myObject { fun

myMethod() {
. ..
}
}
```

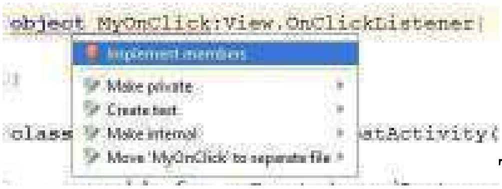
makes an article occasion called myObject with a solitary strategy. Notice that this gives you an article without characterizing a class. Objects proclaimed in this manner can carry out an interface similarly as a Class.

```
object
myObject:myInterface {


. ..
}
```

So to make a case that carries out OnClickListener we should simply begin the affirmation of the article with:

```
object MyOnClick:View.OnClickListener{
}
```

Android Studio will carry out the techniques characterized in the interface for you. You should simply put the cursor in the red line that flags that you haven’t executed the interface and afterward press Alt and Enter. Select Implement individuals from the menu that appears:


```
object MyOnClick:View.OnClickListener{
class                                   stActivity(
```

The created code is not difficult to understand:

```
object MyOnClick: View.OnClickListener { abrogate fun onClick(p0:
View?) {
TODO(“not implemented”)
}

}
```

The question mark toward the finish of View is the main thing that may confound you. This proclaims p0 to be a nullable kind. Java doesn’t utilize Kotlin’s non-nullable sorts and leads and thus anything passed into a Kotlin work that is a Java object must be treated as a nullable kind– subsequently the question mark. It is great practice not to change nullable sorts over to non-nullable sorts without making sure that it isn’t really invalid– see Chapter 17 for more information. For this situation it is basically impossible that that p0 can be invalid so we can project it“dangerously” to Button which is non-nullable:

```
object MyOnClick:View.OnClickListener{ supersede fun onClick(v:

View?) {
(v as Button).text = “You Clicked Me”
}
}
```

If, under any condition v is invalid, an exemption will be thrown. Now that we have our occasion of OnClickListener we can set it as an occasion overseer for the Button b:

```
b.setOnClickListener(MyOnClick)
```

You can likewise utilize an item articulation to make and pass the article in one step.
An item articulation is a mysterious article that you make in the typical manner, yet without providing a name

```
object {

fun myMethod() {
. ..
}
}
```

An item made in this manner can carry out an interface or acquire from a class:

```
object:myInterface{
}
```

So you can compose the past model as:

```
b.s etOnClickListener(object: View.OnClickListener { abrogate fun

onClick(p0: View?) {
(p0 as Button).text = “You Clicked Me”
}
})
```

This is the easiest method of doing the work, however now and then it is valuable to characterize the article so it very well may be reused. There is likewise an inconspicuous contrast in the thing factors are open contingent upon where the article is made– see the part on conclusion later in this chapter.

This technique for making occasion overseers works for an occasion. All you need to do is:

1. make an example of the OnEventListener object and give executions to all of the theoretical methods.
2. Use the setOnEventListener capacity to add the OnEventListener you have recently made to the rundown of occasion handlers. If there are numerous occasion overseers characterized in the OnEventListener you essentially carry out them all and pass the whole object.

In many cases, nonetheless, there is just a solitary dynamic technique (SAM) and for this situation there are considerably easier methods of accomplishing the equivalent result.

Passing Functions In Kotlin

We have examined the fundamental way of passing an object that hosts event handlers as methods, but there are three different, althoughrelated, ways of passing an event handler when the event object is a SAM, i.e. only defines a single event handler:

Function References
Anonymous Functions
Lambda Expressions

Of the three, the lambda articulation is the most usually utilized and encountered. Notice that Kotlin gives alternate ways of characterizing and work with capacities including capacity types, augmentation work, infix capacities and more.

Function References

Kotlin upholds capacities that don ’t have a place with an article. Truth be told, the adaptability of Kotlin’s way to deal with capacities is one the purposes behind needing to utilize it.
So in Kotlin it is entirely legitimate to write:

```
fun myFunction(){

…
}
```

outside of a class definition. Obviously, to remain viable with Java this high level capacity is truth be told a technique for a class, yet one that is created by the compiler to have a name that is gotten from the bundle and the document name.

You can expressly set the name of the class used to contain high level capacities with the
`@file:JvmName(“class name”)`
comment.Notice likewise that Kotlin allows

you to call the high level or bundle level capacity without giving the class name.

That is:

```
myFunction()
```

is legal.
Kotlin additionally has a capacity reference administrator :: which can be

utilized to pass practically any capacity as a boundary in one more capacity or store a reference to the capacity in a variable. This makes it conceivable to utilize any capacity as an occasion handler.

For instance, assuming you characterize the capacity at the bundle level:

```
fun clickEvent(v: View?) {
(v as Button).text = “You
Clicked Me”

}
```

then, at that point, you can write:

```
b.setOnClickListener(::clickEvent)
```

If the capacity is characterized as a technique inside a class you need to write:

```
b.setOnClickListener(this::clickEvent)
```

The most recent form of Kotlin will permit you to drop the this for strategy references.

Notice that despite the fact that this looks as though you are utilizing and passing a reference to a capacity, what’s going on is that the capacity is being changed over into an occurrence of the SAM That is indicated by the boundary’s sort. That is, it’s anything but a reference to a capacity that is passed, however an item built utilizing the capacity. Each time you elapse the capacity, another article is built from the capacity. More often than not this doesn’t have any effect yet you should know about it in the event that you are utilizing the capacity reference administrator to pass a capacity on different occasions. Each time you use it another item carrying out the SAM is made and this uses memory quicker than you may expect.

Anonymous Functions

An unknown capacity is actually what its name recommends – a capacity with no name characterized. You basically write: `fun(parameters){body of function}` You can store a reference to a mysterious capacity in a variable of the right kind and pass it to another capacity.For example:

```
b.setOnClickListener(fun(v: View?) {

(v as Button).text = “You Clicked Me” })
```

You don ’t need to utilize the reference administrator on the grounds that the compiler comprehends that you need to pass the mysterious capacity. It changes the mysterious capacity over to an occasion of the SAM determined by the boundary’s sort. You could save the mysterious capacity in a variable and afterward pass the variable. For example:

```
val clickEvent=fun(v: View?) {

(v as Button).text = “You Clicked Me” }
b.setOnClickListener(clickEvent)
```

It is hard to see any benefit of doing this, nonetheless, other than assuming you are utilizing a similar capacity more than once.

The Lambda

First, what is a lambda?

A lambda is a function that you can define using special notation . According to this perspective, you don’t actually require lambda as

you can do all that you need to utilizing the reference administrator and unknown functions.
Indeed, a lambda is similar as an unknown capacity that you can characterize all the more without any problem. As the capacity doesn’t have a name, it is an unknown capacity and it can likewise be put away in a reasonable variable and passed to another function.

You characterize a lambda by providing boundaries and some code: {parameter list - >code… }
For example:
{a:Int,b:Int - > a+b}
is a lambda that will add its two boundaries

together.

Note that a lambda can ’t have a return articulation – the last worth that the lambda figures is naturally returned. In the past model the lambda consequently returns an Int which is a+b.
A lambda acts like an articulation and you can store a lambda in a variable:
```
var sum={a:Int,b:Int - > a+b}
```
You can utilize the lambda by calling it as a function:
```
sum(1,2)
```
which returns 3.
There are a couple of disentanglements of the lambda language structure that can make them look baffling until you become accustomed to them. If a lambda has no boundaries then you can forget about them and the bolt. So: `{1+2}`
is a lambda that profits 3.
At its most outrageous a lambda can essentially return a value:
```
{0}
```
These principles can make lambda articulations look exceptionally unusual in your code and this may make it harder to

peruse. Try not to go for the briefest and most minimized articulation ensure your code is not difficult to understand.

Events Using Lambdas

To characterize an occasion overseer for an occasion you should simply to utilize the SetEventListener technique with a lambda as its boundary. The lambda is the occasion dealing with function.

For example:

```
button.setOnClickListener({ view - >
(view as Button).text = “You Clicked Me”
})
```

sets the lambda:

`{view - > (view as Button).text = “You Clicked Me”}` as the occasion overseer. Notice that you don’t need to determine the sort of the boundary on the grounds that the compiler can conclude it from the kind of the boundary that the setOnClickListener takes. There is one final syntactic rearrangements. Assuming a capacity

acknowledges a solitary capacity as its main boundary you can overlook the parentheses:

```
b.s etOnClickListener {view->(view as Button).text =“You Clicked Me”}
```

This is the structure Android Studio utilizes for any occasions it creates in layouts. Its possibly advantage is that it looks more as though the code of the occasion overseer is essential for the strategy it is being characterized in.

You can likewise store the lambda in a variable and use it later, yet for this situation the compiler can’t work out what the boundary and return types are thus you need to determine them.

For example:

```
val clickEvent=

{view:View - > (view as Button).text = “You Clicked Me”}
b.setOnClickListener(clickEvent)
```

Notice that, similarly as with different techniques, the lambda is changed over to an occurrence of the occasion object before it is passed to the setOnEventListener method.

Lambdas are utilized for all Android Studio produced occasion controllers and it is the standard method of doing the work. Where conceivable it is the strategy utilized in all further models in this book.

Closure

Closure is one of those themes that sounds as though it will be troublesome. Lambda articulations are not the same as most capacities in that they approach each of the factors that have a place with the strategy that they are proclaimed in.
The way that the lambda approaches the factors in the encasing technique has the abnormal outcome that you could compose the occasion controller as:

```
val message="You Clicked Me"
button.setOnClickListener {view - > button.text = message}
```

This might look odd however it works. Assuming you don't believe that it is odd then you

haven’t saw that the occasion overseer, the lambda, will be executed when the Button is clicked and this is probably going to be well later the encasing strategy has gotten done and every one of its factors not longer exist– but the lambda can in any case utilize message to set the Button’s text.
The framework will keep the worth of message so the lambda can utilize it. This is the pith of a conclusion – the protecting of factors that have left extension so a lambda can in any case get to them.
Unlike Java the factors caught by the Kotlin conclusion don’t need to be final.
Notice that accessing message within the lambda makes it look as if the lambda is naturally still part of the code it is being defined in and not a “detached”functional entity that runs at some time in the distant future– which is what it really is.

Now we come to an unobtrusive point.
The factors caught by a Kotlin conclusion are divided among all elements that catch them. That is, they are caught as references to the variable. This possibly matters when more

than one lambda is in use.
For instance place two Buttons on the plan surface and in the onCreate occasion controller add:

```
var i=0

button.setOnClickListener {view - > button.text =
(++i).toString()} button2.setOnClickListener {view - > button2.text =
(++i).toString()}
```

Notice that the snap occasion overseer for each button catches the variable I inside its conclusion and both offer a similar variable. If you click the first buttonyouwillsee1asitscaptionandifyouthencli otherbuttonyou willsee2asitscaption.The lambdas are sharing a solitary caught variable.
Lambdas are by all account not the only element that you can use to characterize a capacity complete with a conclusion. Neighborhood capacities passed by reference, protests that carry out SAMs and unknown capacities all have terminations that work similarly. Work references don’t have conclusion since they are characterized away from where they are used.
For instance, utilizing a neighborhood named

function:

```
var i=0
fun eventHandler(v:View){

button.text = (++i).toString()
}
button.setOnClickListener(::eventHandler)
```

Notice that this doesn’t work for an overall named capacity or technique. It must be a neighborhood function.
Similarly for a mysterious function:

```
var i=0
button.setOnClickListener(fun (v:View){
button.text = (++i).toString() })
```

This works due to a similar conclusion. For instance, utilizing an object:

```
var i=0
button.setOnClickListener(object: View.OnClickListener { supersede fun
onClick(p0: View?) {
button.text = (++i).toString() }


})
```

the capacity characterized in the article approaches I by means of a closure. When you initially meet the possibility of a conclusion it can appear to be exceptionally peculiar and surprisingly superfluous. Nonetheless, when a capacity is characterized inside a strategy the technique frames its

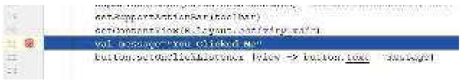
neighborhood setting and it is extremely regular that it ought to approach the nearby factors that are in scope when it is defined.

Closure is helpful for occasion overseers yet it is especially valuable when utilized with callbacks provided to long running capacity. The callback normally needs to deal with the aftereffect of the long running capacity and approaching the information that was current when it was made is regular and useful. There are risks,in any case,to depending on a conclusion.Not all that was in scope at the time the capacity was proclaimed can be ensured to in any case exist. Assuming a neighborhood variable is set to reference an item then it will be remembered for the conclusion and the variable will exist, yet there is no assurance that the article it referred to will in any case exist at the time that occasion controller is executed.

Using Breakpoints

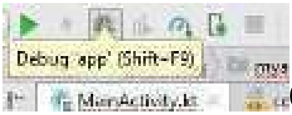
The most straightforward method for making sure that the occasion overseer is called is to embed a breakpoint on the primary line of the onClick method.

Breakpoints are a basic investigating device and you really want to figure out how to utilize them as ahead of schedule as possible. To put a breakpoint essentially click in the“edge” close to the line of code. A red mass appears:

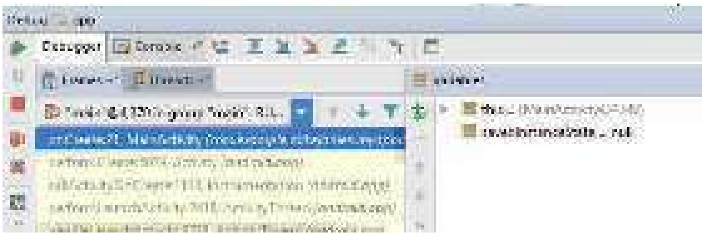


```
[illegible]
setSupportActionBar(bar)
setContentView(R.layout.activity_main)
val message="You clicked me"
button.setOnClickListener {view -> button.text = message}
```

Now when you run the program utilizing the Debug symbol or the Run, Debug order the program will stop when it arrives at any line of code with a breakpoint set.



Once the program stops you can see where it has reached in its execution and you can analyze what is put away in each of the factors in use:

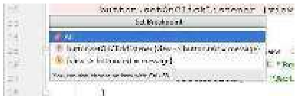


You can likewise restart the program or step through it – see the symbols at the highest point of the investigate window. As you venture through you will see the qualities in the factors change. Any bug will be found at the primary spot you observe an error between what you hope to find and what you really find.

Lambdas represent a specific issue for breakpoints on the grounds that you frequently just have a solitary line to work with:

```
button.setOnClickListener {view - > button.text =
(++i).toString()}
```

If you attempt to set a breakpoint on this Android Studio springs up a message that allows you to choose where you need to set the breakpoint – on the setOnClickListener or the lambda:



There is significantly more to find out about investigating however until further notice this little acquaintance is enough with save you a ton of time.

Modern Java Event Handling

If you keep to Kotlin code then you presently realize all you really want to regarding how to characterize an occasion controller. Notwithstanding, assuming you want to work with existing Java code you will experience various alternate methods of doing the job. Java 8 presently has lambdas and these are utilized to characterize occasion handlers. Before Java 8 there were two fundamental methods of making an occasion overseer. One way was to get the Activity to carry out the interface. This transformed the

Activity into an occasion taking care of article and it needed to carry out the occasion dealing with work as a strategy. The even overseer was then set using:

```
setOnClickListener(this);
```

Which passes the Activity to the part’s occasion controller list.When the even happens the occasion taking care of capacity inside the Activity is called.

Although utilizing the Activity as the occasion taking care of item is experienced the most well-known method of doing the occupation is to utilize a nearby mysterious class:

```
Button button= (Button)findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {


@Override
public void onClick(View view) { Button
b=(Button) view; b.setText(“you
clicked me”);
}
});
```

You can see that the OnClickListener object is made in one stage and the onCLick occasion controller characterized similarly as utilizing a Kotlin object. Truth be told the two are straightforwardly same and the Kotlin object is gone along into a mysterious class.

Fortunately you possibly need to stress over these things is assuming you stray from Kotlin.

Summary

☐ In Java you can’t pass a capacity to set up an occasion controller you need to pass an item that contains the capacity as a method.

☐ Events are for the most part dealt with by Single Abstract Method SAM interfaces. The technique proclaimed in the SAM is the capacity that does the occasion handling.

☐ For every occasion and each control that can produce that occasion there is a SAM of the structure onEventListener and the item has a seteventListener strategy which can be utilized to join the occasion handler.

☐ You can make a SAM in various ways the most broad of which is to utilize a Kotlin object. This can be utilized in any event, when the occasion isn’t carried out as a SAM.

☐ There are three elective methods of carrying out a SAM other than utilizing an object:

1. Function References
2. Anonymous Functions
3. Lambda Expressions

☐ Function references can be utilized to pass a bundle or nearby level capacity or a method.
☐ A mysterious capacity works similarly however you needn't bother with the reference operator.

☐ A lambda is a more limited method of composing an unknown capacity and it is the standard method of executing occasion handlers.

☐ The last syntactic rearrangements is that assuming the setListener work has a solitary capacity boundary then you can drop the parentheses.

☐ Local articles, neighborhood work

references, unknown capacities and lambda are altogether likely to terminations which make the factors that are available to them at the hour of their statement open when they are run.

☐ Breakpoints are the most effective way to investigate your program and they become fundamental when you begin to carry out a few occasion controllers. At the point when run in investigate mode a breakpoint will stop your program so you can analyze the substance of variables.

☐ If you can confine your regard for Kotlin you can make occasion overseers utilizing only items and lambdas. Assuming you really want to comprehend Java code then there are various alternate methods of doing the occupation incorporating carrying out the interface in the Activity and utilizing mysterious neighborhood classes.

Chapter 5

Basic Controls

We have effectively utilized some UI controls in past sections, yet presently we have found how occasions work the time has come to analyze how they work and how you can change the way they look.

Basic Input Controls

The term control comes from the possibility that the client “controls” your program utilizing them. Gadget or part are additionally terms that are utilized to mean exactly the same thing. The Android people group appears to be extremely messy about terminology.
The fundamental info controls are:

Buttons
Text Fields
Checkboxes
Radio Buttons

Toggle Buttons
Switches

If you have utilized other UI structures a considerable lot of these will be known to you and you can presumably jump to simply the ones that interest you.

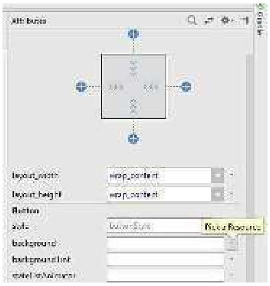
TextView also has to be included in the full list of basic controls, but unlike the Text Field it cannot be modified by the user, i.e. it is an output control only.
Starting right toward the start, we should return to the Button in somewhat more detail.

Button Styles and Properties

There are two essential sorts of button, Button and ImageButton. The ImageButton is similarly as the Button yet it has a src (source) property which can be set to a picture which will be shown as the button symbol and it doesn’t show any text.

The primary credits that you work with on account of a button are things like foundation, which can be set to a shading or a realistic. You can spend a long

time changing the manner in which buttons yet examine, the principle the main significant quality of a button is its onClick overseer. A button is in your UI to be clicked. If you place a Button on the plan surface you can utilize the Attributes window to redo it. There are two unmistakable ways you can change the manner in which a control looks.You can set a style or adjust an attribute. Setting the style alters a bunch of qualities to given standard qualities. You can set a style and afterward change individual ascribes to change it. Regularly when you initially start work on a UI you will set individual credits simply later to define a cognizant style to be applied to the whole UI. Set the foundation of the Button to dim dark by tapping on the ellipsis at the right of the attribute:

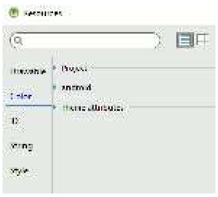


layout_width wrap_content
layout_height wrap_content
Button
style buttonStyle
background
backgroundTint
stateListAnimator

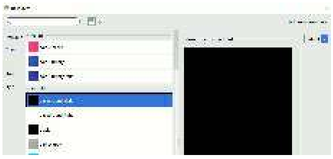
The Resources window that seems gives you admittance to every one of the assets that your program can utilize.Assets are a major piece of programming in Android and we cover them exhaustively in later chapters.

You could enter a shading for the foundation straight by indicating the worth of RGB and alternatively Alpha, its straightforwardness. Utilizing an asset, notwithstanding, implies that assuming you change the asset esteem later on all of the UI parts that utilize it change.

There are various kinds of asset you can choose from, and there are assets characterized by the task, the Android framework and any Themes you may be utilizing.Now and again you can't see these three classifications since they are ventured into long lists:

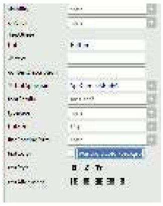


For this situation you need to choose a shading so select the Color tab and afterward the android list. Look down and you will see a bunch of predefined colors that you can utilize. Find background_dark and select it:

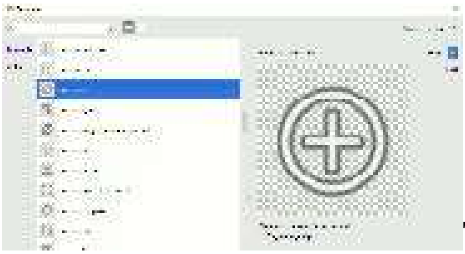


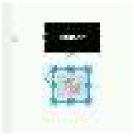
Obviously, you currently can ’t see the text in the Button. To change its shading look down in the Attributes window until you can see TextView. The Button is a composite article and it has a TextView object within it that shows the Button’s text. In the event that you drop down the textAppearance menu you will see textColor.

Simply choose an elective light tone to use:



If you presently place an ImageButton, found lower in the range in Images, on the plan surface, the Resources window opens immediately. This is to permit you to choose a “drawable”, or symbol, for the ImageButton to show. Look down until you find ic_menu_add.At the point when you select it this turns into the src property for the button:

The outcome is two fastens that look something like:



You can invest a lot of energy investigating ways of utilizing a button 's ascribes to style it similarly as you need it to look.
To join a tick occasion overseer on the two buttons you can essentially characterize a capacity in the Activity and set the onClick property on each button. You can track down instances of utilizing buttons this way in prior chapters.

Now that we know how to deal with occasions utilizing lambdas it merits applying this more modern method. As we need to allot similar occasion overseer to each of the three buttons we really want to make an occurrence of OnClickListener utilizing a lambda.

All you need to do is enter:

```
val myOnClickListener = { view: View? - >
val b = (view as Button)
b.text =“You
Clicked Me”


}
```

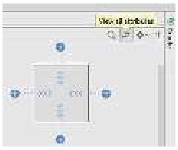
Notice that you currently need to supply the kind of the boundary as the framework can't derive it. All that still needs to be done is to set a case on each of the buttons:

```
button.setOnClickListener(myOnClickListener)
imageButton.setOnClickListener(myOnClickListener)
```

Now when you click on both of the buttons myonClickListener is called.
Note that saving the lambda in a variable for reuse isn’t more effective than replicating out the lambda each time it is required– another occurrence of onClickListener is made for each Button. Its main benefit is that there is a solitary capacity that you really want to change and keep up with rather than two.
It is likewise important that, while a button has no capacity in life other than to be clicked, every one of different controls can likewise react to an onClick occasion. That is you can utilize anything as a button assuming you need to.

All Attributes

If you have checked out any instance of utilizing Buttons in the documentation, or then again on the off chance that you take a gander at marginally more seasoned Android code, you might find a few ascribes that you can't get to utilizing Android Studio. This is on the grounds that the Attributes window just shows you the most utilized traits. To see them all you really want to tap as soon as possible bolt symbol at the top or lower part of the window.



The huge issue with it is that you go from a tiny subset to a mind-boggling list. The most ideal way to manage this is to know what you need to change and observe the specific name of the characteristic that does the work. This is simpler to say than to do much of the time. For instance, assume you need to set a symbol inside a standard text button so it presentations to one side of the message that the button contains. This gives off an impression of being unimaginable assuming you limit yourself to the qualities introduced

in the underlying property window.
If you look into the subtleties of the traits that Button acquires you will find:

drawableLeft
drawableRight
drawableStart
drawableEnd
drawableBottom
drawableTop

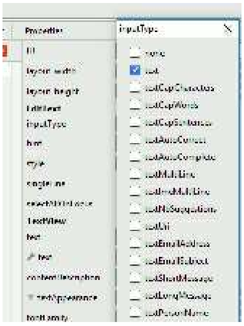
which are not in the Attributes window list. These allow you to display a drawable, i.e. a graphic or icon,at the location specified relative to the text. Once you know the name of the characteristic you can think that it is in the
rundown of all attributes:
Click on the ellipsis button close to it and select a drawable of your decision to show the realistic to one side of the text:

Text Fields

Text Fields are the manner in which you get the client to enter some text for you to process and as you can figure they structure a significant piece of most UIs.
There are a great deal of Text Fields gave in the Toolbox, however they all work similarly varying just in the sort of info they anticipate that the user should type. Throughout creating applications you will experience the greater part of them eventually,however you truly just need to know how to utilize one of them to see how they all work.
If you have utilized another UI Framework then you will have experienced some type of the Text Field previously, yet the Android control is marginally divergent in that it by and large offers the client a redid virtual console alluded to as an IME (Input Method Editor). You can construct your own IMEs, however for the second how about we simply utilize the ones gave by the system.
If you need the Android emulator to utilize

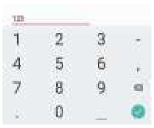
the IME that a genuine Android gadget would utilize then you really want to deselect the Hardware Keyboard Present choice when you are making the AVD (Android Virtual Device). Assuming the emulator utilizes the host machine’s console you don’t see the IME at all.

The main thing to clear up is that despite the fact that there give off an impression of being many sorts of Text Field controls in the Toolbox,they are on the whole instances of the EditText control with its inputType property set to a specific value.
When you place one on the plan surface you will see that it is of type EditText and in the event that you look down to its inputType property you can change the kind of info the Text Field will handle.
When you utilize a specific inputType the client is given a virtual console that is reasonable for composing a value:



```
Properties                inputType
ID                          none
layout_width              ✓ text
layout_height               textCapCharacters
EditText                    textCapWords
inputType                   textCapSentences
hint                        textAutoCorrect
style                       textAutoComplete
singleLine                  textMultiLine
selectAllOnFocus            textImeMultiLine
TextView                    textNoSuggestions
text                        textUri
text                        textEmailAddress
contentDescription          textEmailSubject
textAppearance              textShortMessage
fontFamily                  textLongMessage
                            textPersonName
```

For instance assuming you select a numeric kind you will give the client a worked on IME that main shows numeric keys:



You likewise get an activity key at the base right of the console that the client can press to finish the activity - Send on account of a SMS message, for instance. To choose the activity button for a specific console you should utilize the All Attributes view and select a setting for the imeOptions property: For instance setting it to actionSend powers a Send button, the green paper dart button, to be displayed:

There are numerous different properties that you can use to tweak a Text Field, however there is one standard assignment worth clarifying exhaustively– composing a controller for the onEditorAction event.

The onEditorAction Event

Returning to the EditText control, how about we add an overseer for the Send button. This gives one more freedom to an illustration of the overall course of adding an occasion handler.
First spot an EditText for an email on the plan surface and utilize the Attributes window as depicted in the last area to add a

Send button to the IME that springs up when the client enters message, that is track down imeOptions and select actionSend.
Before you can deal with any new occasion you need to find the name of the occasion audience interface and the setOn technique for that occasion. For the EditorAction occasion the audience interface is called OnEditorActionListener and the setOn strategy is setOnEditorActionListener. With this data we can continue as in the past and utilize a lambda to execute the occasion controller. For this situation we should do the occupation inside the setOnEditorActionListener as the occasion overseer may be required by this one control:

```
editText.setOnEditorActionListener {v, actionId, occasion >

process the event
true
}
```

Notice that now our occasion controller has three boundaries however we actually don't need to supply their sort as the compiler can construe them. The last worth genuine is the return value.
Now all that remains is to compose the

occasion controller, onEditorAction.You can look into the subtleties of the occasion overseer in the documentation:

```
@Override
public boolean onEditorAction(

TextView textView,
int i,
KeyEvent keyEvent)
```

For this situation textView is the control that the occasion is related with, I is the activity id and keyEvent is invalid except if the enter key was used.

If the action has been “consumed”,i.e. acted upon, then the routine should return true and no other handlers will get a chance to process it.In general, events can be passed on to other controls that contain the source of the event.

For a straightforward model how about we add a TextView and move the message that the client enters when they select the send button. Envision that the situation is really sending an email or sms.
We test to check whether the client chose the

send button or another button and assuming they did we move the text:

```
editText.setOnEditorActionListener { v, actionId, occasion - > if (actionId
==
EditorInfo.IME_ACTION_SEN D) {

textView.text = editText.text }
true


}
```

Notice the utilization of the EditorInfo static class to get the whole number id comparing to the send activity. The EditorInfo class has heaps of valuable constants and methods.
If you run the application you will find that you can enter an email address into the EditText field with the assistance of the console and when you press the Send button the location is moved to the TextView.

CheckBoxes

A CheckBox is a genuinely clear UI component. It shows a little name, constrained by the text property with or without a tick mark close to it. The client can choose or deselect as numerous checkboxes as desired. In many cases you try not to manage the condition of a CheckBox until the client squeezes another control,generally a major button stamped Done or comparative. Then you can discover the state of each CheckBox by simply using theisChecked method which returns true or false:

For instance, assuming you have a CheckBox with id checkBox then you can find its state when a button some place on the view is clicked using:

```
var checked = checkBox.isChecked
```

Notice that you can utilize the isChecked technique as though it was a Kotlin property as not exclusively are get and set techniques changed over to a property yet additionally strategies beginning with is returning a boolean.

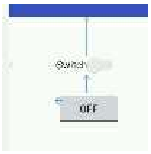
The CheckBox likewise upholds the onClick occasion which can be utilized to deal with changes to its state, and you can set up the onClick occasion overseer utilizing the Attributes window as on account of a Button. So to deal with the CheckBox change of express you should simply set its onClick occasion controller to:

```
checkBox.setOnClickListener { v - > checked =
checkBox.isChecked }
```

Obviously the occasion controller would regularly accomplish something other than store the state in a variable.
If you really want to adjust a CheckBox esteem then, at that point, utilize the setChecked or the flip methods.

Switches and Toggle buttons

Switches and Toggle buttons are only CheckBoxes in another configuration. They store one of two states and they change state when the client taps on them, very much like a CheckBox:



You can actually look at the condition of a Switch/Toggle button utilizing the isChecked strategy and you can utilize its onClick occasion to screen when its state changes. The main genuine contrast is that you can utilize the textOn and textOff to set what is shown when the switch/flip is on or off.

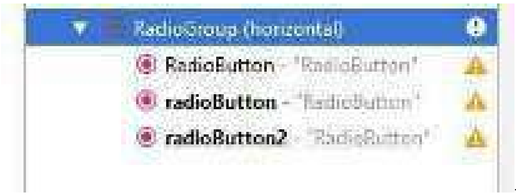
Radio Buttons

The last “straightforward” input control is the RadioButton. This works like a CheckBox in that it very well may be in one of two states, however the enormous contrast is that a bunch of RadioButtons works in a gathering and just one of them can be chosen at a time. *The justification for the expression “radio button” is that, in the beginning of hardware,* *vehicle radios had mechanical tuning buttons organized in a line which let the driver rapidly select a station by squeezing a button. At the point when you squeezed another button the current button sprung up so just one button was squeezed out of nowhere, ensuring that you simply paid attention to one station at a time.*

The main entanglement in utilizing RadioButtons is ensuring you assemble them accurately.To do this we need to utilize a RadioGroup holder which is utilized to hold each of the buttons that cooperate. There are various compartments used to bunch controls,however the most fundamental of these is the RadioGroup.

Using Android Studio you can make a gathering of RadioButtons by first setting a RadioGroup holder on the plan surface and afterward putting however many RadioButtons inside the holder as you require. Assuming a gathering of RadioButtons doesn’t function as you expect,the odds are not every one of the buttons are inside the RadioGroup.

The least demanding method for checking, make and alter a gathering of RadioButtons is to utilize the Component Tree window where you will actually want to see precisely how they are settled.You can likewise add RadioButtons to the holder by hauling to the Component Tree window.


RadioGroup (horizontal)
RadioButton - "RadioButton"
radioButton - "RadioButton"
radioButton2 - "RadioButton"
All the RadioButtons inside a RadioGroup consequently work so just each button can be chosen in turn and you don’t need to accomplish any additional work to execute this behavior.

To discover which button is chosen you can utilize the isChecked technique as on account of the CheckBox. Truth be told you can work with a bunch of RadioButtons in the very same manner as a bunch of CheckBoxes, with the main contrasts being the utilization of the RadioGroup and the way that just one button can be chosen at any one time.

You can utilize the onClick occasion to recognize when any button has been changed and the setChecked or the flip techniques to alter the condition of a button.

Summary

☐ The essential controls that make up the majority of the

straightforward Android UI are: Buttons
Text Fields
CheckBoxes
Radio Buttons
Toggle Buttons
Switches

☐ Each control is tweaked utilizing its ascribes and occasion handlers.
☐ Some credits are stowed away from you by Android Studio except if you select the All

Attributes button.

Chapter 6 Working With Layouts

The decision of Layout is imperative to an Android UI. The Layout is the thing that permits you to position and by and large organize different parts. A decent comprehension of what is on offer in every one of the accessible Layouts can have the effect between a simple and a troublesome UI, according to both the perspective of the software engineer and the client. This is particularly the situation to help a scope of devices.

All of the classes and articles that make up the Android UI are gotten from a similar base class, the View. That is, a Button is a View as are the Layout classes.Be that as it may, Layouts appear to act in totally different ways to the straightforward Button and this brings up the issue what precisely is a Layout?

Understanding Layouts

A Layout is a compartment for other View-inferred objects.At the point when the Layout is approached to deliver itself, it delivers all of the View objects it contains and organizes them inside the space of the showcase it occupies.
The default Layout utilized by Android Studio is the ConstraintLayout and we have as of now checked out utilizing it in prior parts, however it isn’t the main Layout you can use with the Layout Editor.
There are six presently upheld Layouts:

☐ ConstraintLayout
☐ GridLayout
☐ FrameLayout
☐ LinearLayout

☐ RelativeLayout
☐ TableLayout

The ConstraintLayout was new in Android Studio 2.2. It is at present the prescribed design to utilize, and it is the default in Android Studio 3. It very well may be considered as an improvement on the RelativeLayout. On a basic level, it very well may be utilized to make any format that you can carry out utilizing a blend of different designs.It is asserted that ConstraintLayout based UIs are simpler to

constructandfaster because they are "flat" i.e. do notuse multiple layouts nestedinsideoneanother.
Android Studio truly doesn’t need you to utilize any format other than ConstraintLayout as the underlying holder thus it doesn’t allow you to erase the default design. It will allow you to supplant a RelativeLayout in a current task with a ConstraintLayout, yet you can’t erase or supplant it by some other format. The explanation is that supplanting one format by

another is troublesome on account of every one of the progressions in the upheld properties.
You can alter the XML record straightforwardly to supplant one design by another, however you should re-alter all of the design properties. You can supplant a vacant design by one more by altering the XML record and this is as of now the best way to do the job.
Despite the way that ConstraintLayout is the prescribed format to use there are as yet numerous Android projects that utilization the first designs and a few software engineers essentially would rather avoid the ConstraintLayout. Henceforth it merits knowing how the less complex formats work.
Among the first designs, RelativeLayout and LinearLayout are the most utilized, with FrameLayout coming a far off third. The last two, TableLayout and GridLayout, are appropriate for particular sorts of UI and in Android Studio 3 are pretty much unsupported in the Layout Editor so you need to work straightforwardly with their properties.Hence they are best avoided.

Before checking out these other options and how to function with them, it merits having the opportunity to grasps with the fundamental thoughts of formats and the standards they share in like manner. Then we will look at the Frame, Linear and Relative layouts because they are still important. ConstraintLayout, nonetheless, is so significant on the grounds that it is the favored format type for the future that it gets a section all to itself.

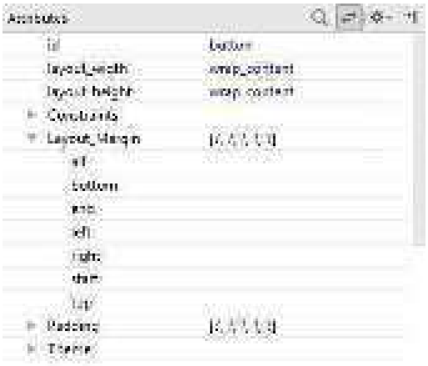
Layout Properties

Mostly you will more often than not think about the properties that are applicable to a control as having a place with the control, yet a design can take care of its business in numerous ways and requires the control to have bunches of unmistakable properties to decide how it is situated.As such,the properties that a control needs to work with a design rely upon the format picked and this makes things hard to organize.

The least complex yet impossible method of executing this is demand that each control carried out each property utilized by each format, even the ones not as of now being utilized. This is obviously inefficient. The answer for the issue really utilized is that every format characterizes a settled class, gotten from LayoutParams, that has each of the properties it needs the control to characterize.The control that will be put inside the format makes an

occasion of the proper LayoutParams class thus differs the boundaries it approaches relying upon the design compartment it ends up in. That is, instead of defining every possible property that any layout could want, a UI component that can be placed in a layout uses the appropriate LayoutParams class to“import” the properties it needs for the layout it finds itself in. This means that a control has two types of property– its own and those that it gets from LayoutParams. Thus in the Layout Editor where the properties are represented by XML attributes a control’s attributes are shown in two groups:

☐ Attributes that belong to the object.
☐ Attributes that are required by the Layout object.
You can tell Layout attributes because they are of the form layout_name in

the XML file. You can see them in the Attributes window in the Layout Editor:

Attributes

id	button
layout_width	wrap_content
layout_height	wrap_content
▸ Constraints	
▾ Layout_Margin	[illegible]
all	
bottom	
end	
left	
right	
start	
top	
▸ Padding	[illegible]
▸ Theme	

So in the screen dump the layout_margin attributes are supplied by the ConstraintLayout.LayoutParams object, but the Padding attribute is something that the Button supports. In other words, which layout attributes you see depends on what sort of layout the control is, but the other attributes belong to the control and are always listed. It is also worth knowing at this early stage that the Layout Editor often presents a simplified set of layout attributes which it then maps onto a larger and more confusing set of layout_ attributes in the XML.

Width and Height

The exact set of Layout attributes that you see depends on the Layout you use. However, there are two that all Layouts support:

☐ layout_width

☐ layout_height

You might think that the width and height of a control were attributes that should belong to a control,but here things are more subtle.A control doesn’t necessarily have a fixed size. It can, for example, ask the Layout to give it as much space as possible, in which case the Layout sets the size of the control. This is the reason why controls have layout_width and layout_height and not just width and height. You can set these properties to any one of

three possible values:
☐ a fixed size, e.g.24px
☐ wrap_content, which sets the size so that it just fits the control's content without clipping
☐ match_parent, which lets the control become as big as the parent Layout can allow
If you use the mouse to drag the frame of a control in the Layout Editor then what happens depends on the control and the Layout.
In most cases the default set by the Layout Editor is wrap_content and it will ignore any attempts you make to interactively size a control. Indeed, in most cases trying to interactively resize a control doesn't change the layout_width or layout_height properties.However,depending on the Layout in use you might appear to change the size of the control due to the setting of other layout properties. More of this when we deal with particular Layout types.
The point is that the layout_width and layout_height are not necessarily the only attributes that control the final displayed size of a control. One thing is fairly certain, if you want to set a fixed size for a control then you

need to type the values into the Property window.

Units

If you are going to enter a fixed size or a location you need to know how to do it. Android supports six units but only two, both pixel-based units, are used routinely:

px– pixel
dp– density-independent pixel

The unit that it is most tempting to use when you first start creating an app is px, the pixel, because you generally have one testing device in mind with a

particular screen size and resolution. This is not a good idea if you want your app to look roughly the same as screen resolution changes. For this you need the density-independent unit,dp,because it adjusts for the screen resolution. If the device has a screen with 160 pixels per inch then 1dp is the same as 1px. If the number of pixels per inch changes then dp to px changes in the same ratio. For example, at 320 pixels per inch 1dp is the same as 2px.

By using density-independent pixels you can keep controls the same size as the resolution changes.
Notice that this does not compensate for the screen size. If you keep the number of pixels fixed and double the resolution then the screen size halves. A control on the screen specified in px would then display at half its original size. A control specified in dp would display at its original size but take up twice the screen real-estate.

Using dp protects you against screens changing their resolution, not their physical size.
not their physical size.

inch tablet, no matter what resolution it has.
As well as pixel-based measures there are also three real world units: mm– millimeters
in– inches

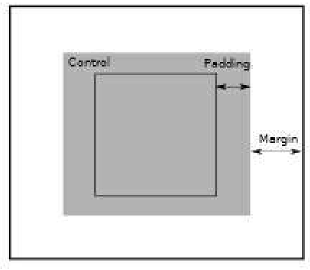
pt– points 1/72 of an inch
All three work in terms of the size of the screen, and the number of
pixels a control uses is related to the screen resolution.If the screen has
160 pixels per inch then 1/160 in=1 px and so on. Notice that once again
these units protect you against resolution changes,but not changes to the
actual screen size. Your button may be 1 inch across on all devices, but
how much of the screen this uses up depends on the size of the screen
the device has. The danger in using real world units is that you might well
specify a fractional number of pixels and end up with an untidy looking
display.
The final unit is also related to pixels but is tied to the user's font size
preference:
sp – scale-independent pixel
This works like the dp unit in that it is scaled with the device's resolution
but it is also scaled by the user's default font size preference. If the user

sets a larger font size preference then all sp values are scaled up to match.
Which unit should you use? The simple answer is that you should use
dp unless you have a good reason not to, because this at least means
that if you have tested your UI on a device of size x it should work
reasonably on all devices of size x,no matter what the resolution.
Android Studio defaults to working in dp whenever you enter a value
without a unit or when you interactively size or move a control.

A Control is Just a Box

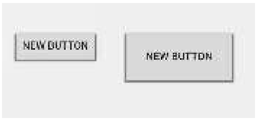
As far as a Layout is concerned, a control is just a rectangle. Its size is given by layout_width and layout_height and these can be set by the control or, more often, by the Layout. Once the Layout knows the size of

the control it can position it according to the rules you have established using the Layout’s properties.
If you want to know the position that a control has been assigned then you can use its Top and Left properties. This gives you the position of the top left- hand corner of the control’s rectangle. You can work out where the other corners are by using Width and Height properties,but to make things easier there is also Right and Bottom property.Notice that the position of the top left-hand corner of the rectangle is always relative to the Layout it is in. That is,the position is not an absolute screen position. It is also worth knowing that controls also support padding, dead-space inside the control. This is space left between the outside edge and the content. In addition some, but not all, layouts support margins, dead-space outside a control that can be used to add space between controls:



Notice that padding is a property of the control and margin is a layout property. You can set each margin or padding on the left, right, top or bottom individually or specify a single value to be used for all of them.

In theory, padding is used to put space around the content of a control, but it can also be used simply to make the control bigger when its dimensions are set to wrap its contents.For example,the Button on the left has zero padding and the one on the right has a padding of 30dp all round:



Similarly, margins are used to put space around a control, but they can be used to position one control relative to another or to its container. This is how RelativeLayout and ConstraintLayout work.

Gravity

Gravity is often regarded as mysterious, partly because of its name and partly because there are often two gravity properties in play. Basically, gravity just sets where in a dynamic layout something is positioned.

Simple gravity settings are:
top
bottom
left
right
center
center_vertical
center_horizontal

The meaning of all of these is obvious in that the object just moves to the specified position. However, things get a little complicated if you try to set an object to display at the left when the size of its container has been adjusted to fit, i.e. it is already as far to the left and the right as it can be. You can also set multiple gravity options. For example, you can set left and right at the same time and this just centers the object horizontally.

What makes gravity even more complicated is that there are settings that change the size of the object affected:
fill
fill_vertical
fill_horizontal
In each case the object grows to fill the specified
dimension. There are also two clipping settings:
clip_vertical
clip_horizontal
These work with the top, bottom, left and right to clip an object to fit the container. For example, if you set gravity to top and clip_vertical then the object will be positioned at the top of the container and its

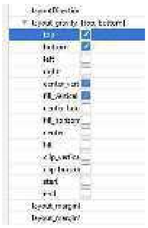
bottom edge will be clipped.
Most of the time you will simply use gravity settings like center or top. If you try to use complicated combinations then things tend not to work as you might expect.
The final complication, which in fact turns out to be quite straightforward, is that controls have a gravity property and Layouts provide a
layout_gravity property. The dIfference is very simple. The gravity property sets what happens to the contents of a control and the layout_gravity sets how the control is positioned in the Layout container. For example, if you have a Button and you set its gravity property to top then the text within the button will be moved to align with the top. If, on the other hand, you set the Button’s layout_gravity to top the whole Button moves to the top of the Layout container. Notice that not all Layouts provide a layout_gravity property to their child controls.

The FrameLayout

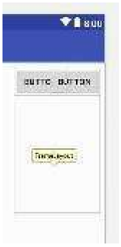
The FrameLayout is the simplest of all the Layouts. It really doesn ’t do very much to position the controls it contains and its intended use is to host a single control, i.e. it really does just act as a frame around a control.

The Layout Editor no longer provides as much help with FrameLayout as it did and if you have used an earlier version you may well find the new behavior frustrating. Indeed, the only way you can make a FrameLayout the main layout control is to edit the XML file and replace ConstraintLayout by FrameLayout. If you want to try out a FrameLayout, simply drag and drop it onto the default ConstraintLayout and use it there.

When you drop a control in a FrameLayout it is positioned at the top left and you cannot drag it to a new position within the layout. To set a control’s position you have to find the layout_gravity attribute, not the gravity attribute, and set one or more of top, bottom, left, right, center_horizontal, and center_vertical.


layoutDirection
layout_gravity [top, bottom]
top ✓
bottom ✓
left
right
center_vertical ✓
fill_vertical ✓
center_horizontal
fill_horizontal
center
fill
clip_vertical
clip_horizontal
start
end
layout_margin
layout_margin

You can use this to position multiple controls in the same FrameLayout, but notice that if the total size of the FrameLayout changes the different controls may welloverlap:



If two controls overlap in a FrameLayout, they are drawn in the order in which they were added. In other words, the last one added is drawn on top of the others. This aspect of the FrameLayout makes it useful if you want to display multiple controls and switch which one is visible. Simply put all the controls into the FrameLayout and select one to be visible using its Visible property. More commonly a FrameLayout is used simply as a placeholder for a component of a layout that isn’t specified until some time later. For example, if you make use of a Fragment, see the companion book Android Programming*:Mastering Fragments & Dialogs*,to create part of a UI or read in a list of things to display, then often you need a container for the new component to be present in your static layout. A FrameLayout does the job very efficiently.

☐ Use a FrameLayout when you need the simplest Layout that will hold one or a small number of components without the need for much in the way of positioning or sizing.

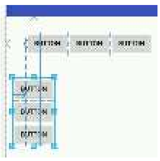
LinearLayout

The next layout we need to consider is the LinearLayout. This is a simple layout that can be used to do a great deal of the basic work of

organizing a UI. In fact once you start using LinearLayout, it tends to be the one you think of using far too often. You can use a LinearLayout as the base Layout, replacing the default ConstraintLayout that Android Studio provides, or you can place a LinearLayout within the ConstraintLayout.
In Android Studio LinearLayout occurs twice in the Palette– once as a vertical and once as a horizontal LinearLayout. The difference, however, is just the setting of the orientation property to horizontal or vertical.In other words, you can swap a horizontal and vertical linear layout with a simple property change.
The horizontal LinearLayout acts as a row container and a vertical LinearLayout acts as a column container. You can use nested LinearLayouts to build up something that looks like a table, but if this gets very complicated it is better to use ConstraintLayout. Nesting layouts like this is also inefficient as the rendering engine has to compute the layout multiple times to get it right. The advice is to use a ConstraintLayout and avoid nesting.
If you place a LinearLayout on the

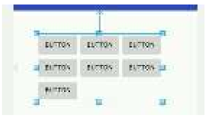
ConstraintLayout then you can position it like any other control. If you then place other controls inside it then they will stack up horizontally to form a row or vertically to form a column.
This sounds easy but there are lots of ways to use a LinearLayout.

For example, if you put a horizontal and a vertical LinearLayout in the ConstraintLayout then how they behave depends on what you set their
layout_widthandlayout_heightto.Ifyousetittow
Layouts act like a horizontal and vertical panelof controls, i.e. you can move all of the controls as a block:



It can be very difficult to be sure what you are dragging in the Layout Editor as it is easy to pick up one of the Buttons rather than the layout. Make use of the Component Tree window to select the layout and to make sure that the Buttons are in the layout you think they are. Things get more interesting when you nest one LinearLayout inside another to create a table. For example, you can create a calculator style keypad by nesting three horizontal LinearLayouts inside a single vertical LinearLayout.

421



That is, place a vertical LinearLayout on the screen and then place three horizontal LinearLayouts within it. Within each horizontal LinearLayout place three buttons. If you have difficulty doing this, use the Component Tree tomakesurethatthecomponentsarecorrectlynest the layout_width and layout_height to wrap_content, otherwise the LinearLayouts willoverlap:

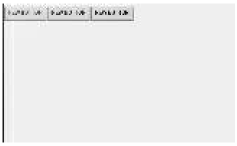
This is easier to arrange than using the

ConstraintLayout. The final Button is just placed into the vertical LinearLayout and it forms a row all of its own. ☐ LinearLayout is a useful grouping device whenever you need a row or column of controls.

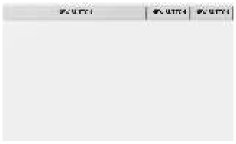
Layout_weight

There ’s one last mystery of the LinearLayout to discuss, layout_weight, a layout property that only the LinearLayout supports. If you assign a layout_weight to any of the controls in a LinearLayout then the controls are adjusted in size to fill any unused space in proportion to their weights. The really important part of this description is“unused space”. What happens is that Android first computes the layout ignoring any weight assignments.
This means that the controls are set to the sizes you specified. Next the system determines what space remains unused in the containing LinearLayout. This is then distributed between the controls that have nonzero values of layout_weight in

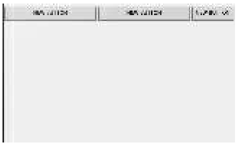
proportion to their weights. For example, suppose we have a horizontal LinearLayout with three Buttons all set to wrap_content.The screen has been rotated to provide a lot of unused space for the example:



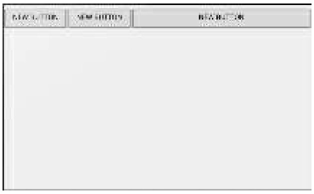
You can see that there is a lot of unused space over to the right. If we now set the first Button’s layout_weight to 1 it will be allocated all of that unused space:



If you now set the second Button ’s layout_weight to 1 then the unused space will be shared between the first two Buttons equally:



You can guess what would happen If we now set the third Button ’s layout_weight to 1, the space would be shared equally and all three buttons would be the same size. If, however, the first button was given a weight of 2 then the unused space would be shared out in the ratio 2:1:1 and so on. More interestingly what do you think would happen if you assigned a fixed width to the third Button? The answer is simple. If the third Button’s layout_weight is zero then it is set to the width specified and the other two buttons get the unused space.For example setting the third Button to 350dp gives:



However, if the third button has a layout_weight set then it will probably change its width because it gets a share of the unused space just like the other buttons.In other words,when you set a non-zero layout_weight a control can change its size even though you have set a specific size for it. This leads to the idea of“measured size” and“actual size”.

In the case of the third Button its measured size is 350dp but if its layout_weight is non-zero then its actual size on the screen will be different– it will be allocated some of the unused space.

When you are working with components in code the Width and Height properties will give you the actual width and height of the control. The MeasuredWidth and MeasuredHeight properties will give you the measured width and height before any adjustment by the Layout has been performed. Finally, it is worth pointing out that if you want to be sure that the three Buttons are the same size you have to set their widths to 0dp and weight to 1 (or the same value). Why is this necessary? When you set the widths to zero all of the space is unused and the system will divided it equally between each one. You can also set their widths to some constant minimum value and then let the weight mechanism share out the unused space.

RelativeLayout

The RelativeLayout was the most used in the past and it is still worth knowing about because you will meet it in existing apps and you might have to use it if ConstraintLayout doesn’t work for you. It was the one that was favored by Android Studio until the ConstraintLayout was introduced.

It is a complex and sophisticated layout component and you might think that you should prefer simpler alternatives if at all possible. For example, you can often use a number of LinearLayouts to do the work of a single RelativeLayout. The most commonly quoted rule is that you should try to design your UI using the smallest number of Layouts. In particular, deep nesting of Layouts, i.e. one Layout inside another, slows things down because the system has to dig deep into each layer of Layout and this can take a lot of work. The rule is:

☐ Prefer a shallow sophisticated Layout to a deep nest of simpler ones. You can usually replace a set of nested LinearLayouts with a RelativeLayout or a ConstraintLayout.

For such a capable Layout, RelativeLayout has only a few Layout properties. They fall into two groups:
Properties concerned with positioning the control relative to the parent container
Properties concerned with positioning relative to another control.

At least one of the controls in the Layout has to be positioned relative to the parent container to give the rest of the Layout a position. However, any number of controls can be positioned relative to the parent container if this fits in with what you are trying to achieve.

The RelativeLayout attributes are presented and organized by the Attributes window slightly differently to the way they are represented as attributes in the XML or in code. This description applies to the way Android Studio presents them. Refer to the documentation for the XML or programmatic constants.

Edge Alignment

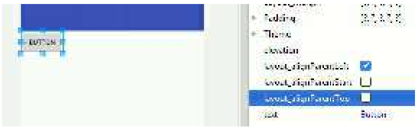
The principle is that you can specify the alignment of any pair of edges, one in the parent and one in the child. This will move the child control so that its edge lines up with the parent edge, For example, top edge to top edge. If you specify two pairs of edges then you can change the size of the control as well as positioning it. Forexample, top to top and bottom to bottom makes the child control the same height as the parent.

Layout Relative to Parent

The fundamental parent layout attributes are:
layout_alignParentTop

layout_alignParentLeft
layout_alignParentBottom
layout_alignParentRight

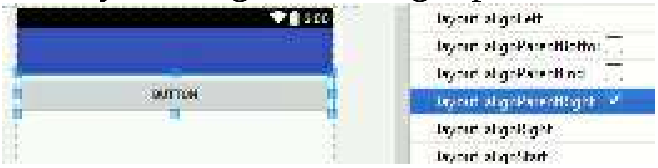
which adjust the relating edge of the control with that of the parent container.
For instance: setting layout_alignParentLeft moves the passed on side of the control to the left half of the parent:



Padding
Theme
elevation
layout_alignParentLeft ✓
layout_alignParentBottom ☐
layout_alignParentTop ☐
text Button

This works with practically no resizing of the control. Assuming you select two restricting arrangements, top and base or left and right, then, at that point, both the edges are moved and the control is resized.

For instance, setting layout_alignParentLeft and layout_alignParentRight produces:



layout_alignLeft
layout_alignParentBottom
layout_alignParentEnd
layout_alignParentRight ✓
layout_alignRight
layout_alignStart

You can likewise adjust to the focal point of the parent with:
layout_centerInParent
layout_centerVertical
layout_centerHorizontal

Layout Relative to Another Component

The Layout Editor used to plan a confounding arrangement of XML properties that situated a part comparative with another part.The most recent form no longer plays out this disentanglement. Rather you can either permit the Layout Editor to work out how to set position comparative with another part or you can work with the crude credits in the Attributes window.
The fundamental thought behind each of the situating ascribes comparative with another part is that you just stock the name of the other part to the applicable attributes:

layout_alignTop
layout_alignLeft
layout_alignBottom

layout_alignRight

For example:
layout_alignRight= button1
sets the right-hand edge of the control to line up with the right-hand edge of button1.As continually, adjusting two edges top/base and left/right changes the size of the control.
The attributes:

layout_above
layout_below
adjust the lower part of the control to the highest point of the referred to control and the base with the top respectively.

Similarly the attributes:
layout_toLeftOf
layout_toRightOf

adjust the left/right of the control with the right/left of the referred to control. At long last the attribute:
baseline

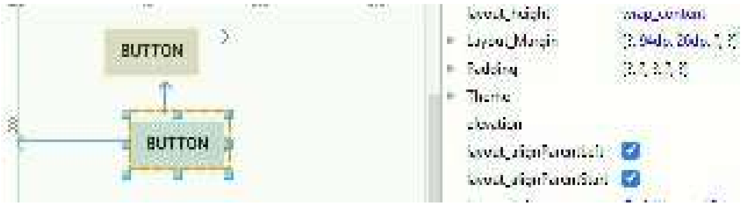
adjusts the text benchmark in the parent and youngster controls. In API 17 a new feature was added to take account of the direction that text

should flow within a layout, this caters for languages such as Arabic that start at the right-hand side and go towards the left. As a matter of course Layout_Direction is left-to-right and the beginning edge is as old as left edge and the end edge is as old as right edge. If Layout_Direction is set right-toleft then start is the same as right and end is the same as left. You can set startpadding to control the cushioning on the left or right contingent upon the design direction
set. The entirety of the left/right ascribes have a beginning/end version.

Margin Offsets

So far nothing remains at this point but to adjust sets of edges. How would you determine definite positions comparative with another control or the parent?

The appropriate response is to utilize the normal example of setting the edges of the control. On the off chance that you adjust top edges however set a top edge of 10dp on the kid then the highest point of the youngster control will be 10dp lower than the parent control:



layout_height	wrap_content
Layout_Margin	[illegible]
Padding	[illegible]
Theme	
elevation	
layout_alignParentLeft	✓
layout_alignParentStart	✓

So the edge arrangement is utilized to determine the overall course of one control to another and the edges set give the specific offset.

RelativeLayout and the Layout Editor

With all of this comprehended you would now be able to perceive how the Layout Editor allows you to create a RelativeLayout.As you move a control around the plan surface, the closest other control or the parent Layout is chosen as the parent to use for situating, the nearest edges are utilized for arrangement and that edge is set to the distance between the parent and

child.
This functions admirably however it can once in a while be hard to get the Layout Editor to pick the control or the Layout as you move a control around. You can generally utilize the Attributes window to physically set this assuming it demonstrates too hard to even think about setting interactively.
Also notice that in the event that you drag an edge of one control near arrangement with the edge of another control then this will bring about that edge being adjusted and the control changes its size. For instance, assuming that you drag the right half of a control to the right size of the Layout then the width of the control changes. This conduct can confound fledglings utilizing the Layout Editor as it gives off an impression of being feasible to resize controls by hauling an edge, yet more often than not the control snaps back to its unique size when delivered. Obviously, it possibly resizes when the edge you are hauling lines up with a comparing edge on another control.
It must be said again that the Layout Editor isn’t as simple to use with the RelativeLayout

as in past forms of Android Studio. For instance, it no longer supplies intuitive situating criticism as far as pixel counterbalances and it no longer uses a worked on arrangement of format properties which map onto the genuine XML properties. These things may change in later forms, however with the accentuation on ConstraintLayout it appears unlikely.
Once you comprehend the way that the RelativeLayout works then, at that point, utilizing the Layout Editor turns out to be a lot more straightforward thus does utilizing RelativeLayout to make a complex UI.

Summary

☐ You can utilize diverse Layout holders to make UIs. Every Layout has its own offices for how youngster controls are situated and sized.

☐ The default in Android Studio 3 is ConstraintLayout. ☐ The main choices are

FrameLayout, LinearLayout and RelativeLayout. ☐ Each Layout has its own arrangement of format properties to

control situating and estimating of a control. Kid controls have an occasion of the format properties class to advise the Layout how to position and measure them. All Layouts support layout_width and layout_height.

☐ You can indicate position utilizing various units, yet much of the time use dp (thickness free pixels) as this works the same way on screens of a similar size however unique resolutions.

☐ As far as the Layout is concerned, a control is only a square shape, width by tallness, situated utilizing top and left.

☐ All controls have cushioning properties which determine additional room around the control’s content.
☐ Some Layouts give layout_margin properties that set additional room around the outside of the control.

☐ Gravity simply sets the simple positioning of an object - top, bottom, right, left. Every control has a gravity property which sets the position of its content, e.g. the text in a Button. Some Layouts have a layout_gravitypropertythatsetshowacontrolwil
☐ The FrameLayout is the most straightforward of all Layouts and simply has layout_gravity for situating. As a rule it holds a solitary control and it is regularly utilized as a placeholder.
☐ The LinearLayout can be utilized to coordinate controls as a line or a segment. Just as gravity, the LinearLayout additionally upholds the detail of a control's weight. Later the deliberate size of each control is dictated by the Layout, the excess unused space is allotted to the controls in similar extents as their allocated weights.
☐ Complex designs can be made by settling LinearLayouts inside one another to create a segment of lines or a line of segments. This has brought about the LinearLayout being the most used.
☐ The overall guideline is to attempt to choose a Layout that outcomes in the littlest

settling of Layout compartments. It is subsequently better to utilize a solitary RelativeLayout or ConstraintLayout rather than profoundly settled LinearLayouts.

Chapter 7 The ConstraintLayout

The ConstraintLayout was new in Android Studio 2.2 and it utilizes an extra library. The help library is viable with all variants of Android back to Gingerbread (2.3, API level 9) thus you can utilize the ConstraintLayout except if you plan focusing on gadgets running sooner than Gingerbread. ConstraintLayout was acquainted in a work with make design more receptive to screen size changes, and to work on the productivity of format by making it conceivable to try not to settle designs. It is, basically, a further developed RelativeLayout and on the off

chance that you have perused the part on the RelativeLayout quite a bit of what follows will appear to be natural. The Layout Editor has been changed to function admirably with the ConstraintLayout at the expense of making different Layouts harder to work with. This may change as Android Studio keeps on growing yet with the accentuation on ConstraintLayout being the answer for all that this appears progressively unlikely.
In the past the ConstraintLayout was immature and didn't function admirably in all circumstances. In Android Studio 3 the help library has improved alongside the Layout Editor. It currently appears to be sensible to base all of your future applications on the ConstraintLayout. *At the hour of composing the default variant of ConstraintLayout is 1.0.2. This backings all highlights including chains and rules yet not hindrances and gatherings. The current documentation recommends that the Layout Editor upholds obstructions and gatherings yet assuming you utilize the default rendition of ConstraintLayout they aren't and they don't show up in the setting menu.*

If you need to utilize these highlights you need to guarantee that your venture is utilizing 1.1.0 or later. This is at present in beta however could well arrive at a last form before the following variant of Android Studio.To utilize it you want to alter the build.gradle record to read:

conditions {
execution fileTree(dir: 'libs', include: ['.jar']) execution*
'com.android.support:appcompat-v7:26.1.0'
execution'com.android.support.constraint:

requirement layout:1.1.0-beta3 '

Onlythelastlineneedschangingi.e.from1.0.2to1
beta3.Youwillneed
toresynctheprojectandthenyouwillseegroupan
contextmenu.

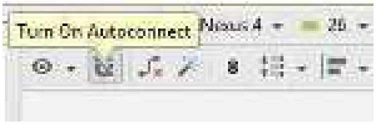
If you have a current Layout then you can request that Android Studio convert it to ConstraintLayout. You should simply right tap on the Layout in the Component Tree and select Convert to ConstraintLayout.This can be utilized to diminish a settled design to a solitary"level" ConstraintLayout,however be cautioned that by and by it regularly gets things exceptionally off-base. Regularly all that you can say of a changed over format is that it

gives a beginning stage to re-implementation. Let’s investigate how the ConstraintLayout functions in the Layout Editor in more detail than in past chapters.
Using the ConstraintLayout implies the main thing that influences where a part shows are the requirements you apply. So how would you apply an imperative? There are two methodologies and we have appear them both momentarily prior parts. You can have the Layout Editor recommend them for you consequently, or you can apply them manually.

Automatic Constraints

Automatic imperatives, which we met in Chapter 3, should make things simple. There are two methods for getting the manager to apply limitations dynamically:
Autoconnect mode– for a solitary component
Infer Constraints– for the whole layout

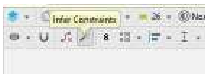
They do marginally various things and you want to figure out how to make them work together.
To turn Autoconnect on essentially click its symbol at the highest point of the Layout Editor:



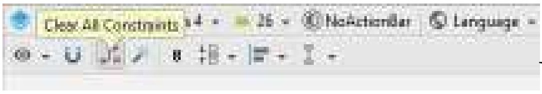
As previously clarified in Chapter 3, Autoconnect applies imperatives to a solitary part that you are either setting on the plan surface
interestingly,or to a part that doesn’t as of now have requirements applied that you drag to another area.At the point when a requirement is added,Autoconnect doesn’t endeavor to transform it assuming that you move the part to where an alternate limitation may be more proper.The other issue previously referenced is that Autoconnect doesn’t add imperatives that seem

self-evident. It adds imperatives for focusing and setting near an edge, however it doesn’t

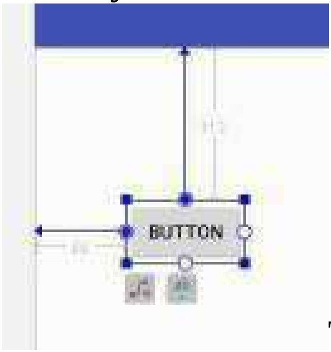
add any requirement comparative with another part or to by and large position the component.
Right now Autoconnect is a genuinely frail element and scarcely worth turning on in most cases.
The Infer Constraints option is, in many ways, easier to use and more powerfulthanAutoconnect.Allyou have to dois positionthecomponents where youwant them andthen click the Infer Constraints icon:



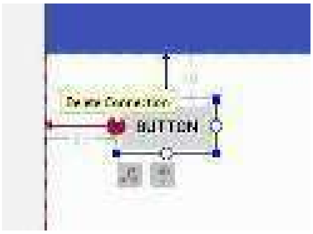
Only limitations that are important to fix the place of a part are added – existing imperatives are not adjusted. This implies you can utilize Infer Constraints to ensure that your design has an adequate number of requirements to make it work. On the off chance that none are added it was OK. It likewise implies that assuming you click Infer Constraints a subsequent time nothing changes regardless of whether you have moved parts.To get another arrangement of requirements you need to erase the limitations you need recomputed and afterward click Infer Constraints. You can delete all of the constraints in a layout using the Clear All Constraints button:



You can get all free from the imperatives on a specific part by choosing it and tapping on the red cross symbol that shows up beneath it:

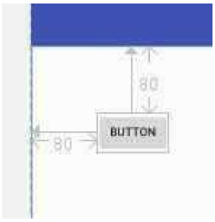


To clear asingle constraintsimply hover themouseover thecircle that marks the location of the constraint until it turns red and then click it:



In this manner you can specifically erase imperatives and yet again apply the Infer Constraints activity, or basically physically apply a more fitting constraint.
Infer Constraints works in an extremely basic manner. It applies limitations to parts as per what they are nearest to.This methodology brings about a format that works however it probably won’t be the most sensible for future modification.
A decent system is to turn off Autoconnect and utilize the Infer Constraints choice each time you place another part on the plan surface. This permits you to develop a design each part in turn and actually look at each new arrangement of imperatives. You would then be able to adjust the recently included imperatives and move to the following component.
Where things get troublesome is the point at

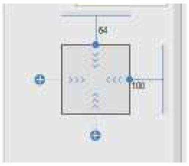
which you have a multi-part design and need to roll out revolutionary improvements. Regularly it is more straightforward for this situation to erase the requirements in general and start again.
As longas suitable constraints are in place, the ConstraintLayout works much like the RelativeLayout. For a component that has been placed on its own close to the top and left-hand edge of the screen, constraints will be added that fix its distance from the left and top:



From now on when the button is shown it will situate itself at 80dp from the left and 80dp from the top. These requirements are applied regardless the size of the actual screen is.

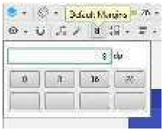
It merits realizing that the genuine situating is

accomplished, as on account of the RelativeLayout, by setting the button’s edge properties to 80dp. However, in contrast to what occurs in the RelativeLayout. if you move the Button closer to the right-hand edge, then the constraint will not change to one relative to that edge. When an imperative is set it stays set except if you erase it and apply another one. All that happens when you drag a part is that the worth of the imperative changes, not its type.
It is additionally worth realizing that the Attributes window has an outline that shows you the right now applied limitations and allows you to alter them and set distances exactly:

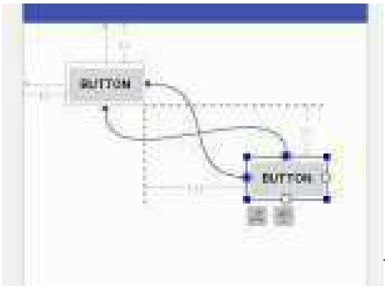


If you click on one of the requirement lines, you can enter an incentive for the distance. You can likewise tap on the X that shows up and erase the limitation completely.
You can likewise set the default edge that is

utilized for new imperatives utilizing the default edge symbol in the menu bar:



Just like a RelativeLayout, you can set requirements comparative with the parent or comparative with different controls in the format. Have a go at putting a second button on a superficial level and moving it near the first. Presently assuming you click the Infer Constraints button,the imperatives that are applied are taken comparative with the first button:



You can see that for this situation the subsequent button is situated 114 from the left edge of the main button and 57 beneath it.

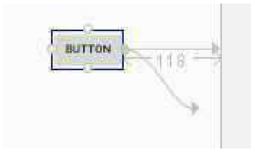
Once again these limitations will be submitted to regardless of how huge or little the actual screen the UI is shown on. Likewise, assuming you move the main button, the subsequent button will keep its position comparative with the first. This is by and large like the RelativeLayout and, as long as you set the right requirements, anything you can do with RelativeLayout should be possible with ConstraintLayout.

If you play with the situating in the Layout Editor you will rapidly get the idea.

The huge issue with the Layout Editor naturally applying imperatives is that it regularly misses the point. Working out how one control should be fixed relative to another reallyrequires someintelligence and,at the moment, the LayoutEditordoesn’thaveit.

Manual Constraints

You can place constraints on positioning manually. All you have to do is drag a line from the constraint circles on the component to the edge you want to use as a reference point. For example, to position the button relative to the right side of the screen simply drag the circle on the right of the button to the right side of the screen and then move the button where you want it:

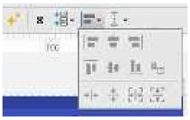


You can blend the programmed and manual setting of limitations and this is regularly the most ideal way to work. By and large, you are determining the imperative as the separation from some point on the part to a point on some other component.
For instance, to put a part comparative with another then essentially drag the requirement point on the first to an edge of the second.
For the situation displayed beneath manual imperatives have been applied in light of the fact that limitation induction would have put requirements on the lefthand edge of the screen:

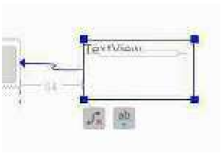
At this point it merits making a little, yet significant, point extremely clear. While you can physically set negative edges in the RelativeLayout, you can’t utilize them in ConstraintLayout.You can’t drag a part so it needs a negative offset– the Layout Editor stops your drag at nothing.Assuming you set a negative edge in the edge properties it will show as negative however be treated as zero.

What this means in practice is that if you align the left side of a component to the left side of another, then the only type of constraint you can apply shifts the second button to the right. Assuming you attempt to move it to the left then the negative edge that would result is overlooked. To comprehend this attempt it in the Layout Editor and you will observe that you can’t drag the lower button in the past screen dump past the left half of the top button. Similarly, you can’t drag the lower button over the lower part of the upper button.This has a specific rationale, however it is additionally prohibitive on the grounds that it implies you can’t set an imperative that interfaces the

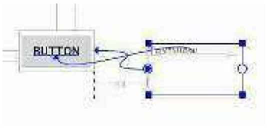
highest point of a part to the highest point of a part that is beneath it.
You can utilize left-side to left-side requirements to left-adjust parts and right-side to right-side limitations to right-adjust parts, etc. To assist you with doing this there is an arrangement instrument palette:



To utilize it select every one of the parts that you need to adjust and afterward click the ideal arrangement apparatus. It is critical to understand that utilizing these arrangement instruments essentially applies the proper imperatives. You could accomplish similar outcomes by applying the imperatives yourself manually.
As well as having the option to adjust various sides of a control,you can likewise set a pattern limitation so text lines up. This was clarified exhaustively in Chapter 3,however set forth plainly– assuming you click on the second symbol that seems when you select a part, a circular box appears:



Hover the cursor over the curved box until is featured and haul from the circular box to the pattern of the text in the part that you need to adjust with:

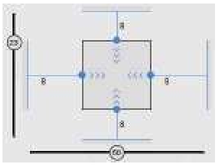


Bias Constraints

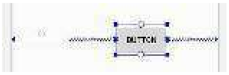
So far the ConstraintLayout hasn ’t actually given anything new, yet it has a more modern requirement that doesn’t have a partner in the RelativeLayout. Alluded to as“inclination” this kind of limitation works as far as proportions, communicated as rates and shown as crisscross lines:
To utilize it you need to make two requirements that “battle” one another. For

instance, assuming you drag one imperative to the left and its partner to the right, the outcome is a flat inclination used to situate the control at the ideal extent of the format size. You can accomplish an upward predisposition by hauling requirements from the top and the bottom.

As you drag the control around the screen, the divisions or rates are refreshed in the Attributes window. You can utilize the sliders that seem to set the percentages:



Things are somewhat more muddled in that you can likewise determine an edge which gives a general part to the predisposition imperative. Assuming you investigate the showcase in the Attributes windows displayed above, you can see that the limitations have a worth of 8 showed close by them. This is the default edge alloted when you made the limitation.As you drag the part around the screen you can’t situate it nearer to the sides than the edge set in that direction. For example, if you edit the left margin to be 100, then the constraint display acquires a straight portion 100 units long. Assuming you currently attempt to move the button to one side, you will observe you can’t draw nearer than 100 units. Notice that it is the distance between the predefined edges that is split in the proportion set by the predisposition. If you want the entire screen to be used, then set the margins to zero:



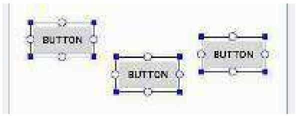
Chains

An as of late acquainted element is the capacity with set inclination imperatives between parts just as to the parent compartment. That is, you can set sets of limitations between parts that work similarly as a
predisposition requirement. In doing as such

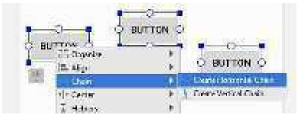
you make what is currently called a“chain” of parts. The justification behind acquainting chains is with make it simpler for ConstraintLayout to make the kind of thing that you would have used

LinearLayout for, specifically lines or segments of parts with corresponding spacing.

Creating a chain is a marginally secretive interaction. To make an even chain of say three Buttons the primary errand is to organize them generally into a line:

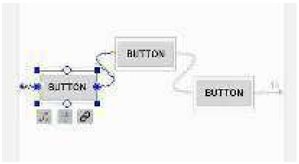


If they aren ’t in a sensible line the limitations will be applied independently to every part.Select each of the three by hauling a marquee around the three and afterward right snap while every one of the three are chosen and utilize the Chain, Create Horizontal Chain:



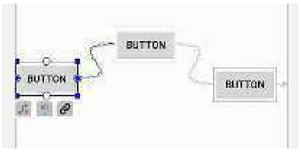
To make an upward chain you orchestrate the parts in an upward section, select them all and right snap and utilize the Chain,Create Vertical Chain order.All that works similarly however pivoted by 90 degrees.

If everything goes to design and the parts are recognized as a likely chain, the unique imperatives will be applied and you will see a chain symbol between the internal components:



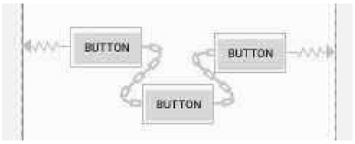
Notice that the default format for the chain is Spread which conveys the perspectives equitably, assessing edges. There are three elective design modes which are chosen by tapping on the chain symbol that seems when you select a part that is important for a chain.

If you click this once the format changes to Spread Inside, which puts the first and last part hard against the compartment and disseminates the others equitably in the space available:



If you click a second time the design changes to Packed which puts every part as near one another as could be expected, taking into account edges, and afterward puts them as a gathering focused in the container.

You can situate the gathering utilizing the predisposition setting of the main part in the chain. Notice that the inclination setting has no impact in the other chain design modes:

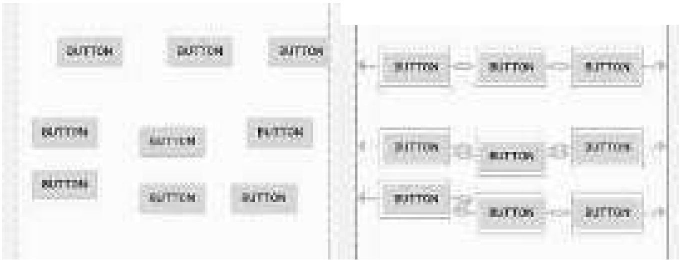


You can set a format weight utilizing

layout_constraintHorizontal_weight and layout_constraintVertical_weight. In spread or spread inside mode this conveys the space with respect to the loads and emulates the manner in which loads work in a LinearLayout.

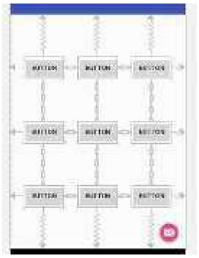
A Chained Keypad

To show how helpful chains are, we should carry out a keypad of the sort that we utilized in the number cruncher project toward the finish of Chapter 3.
First spot nine buttons on the plan surface in an unpleasant 3 by 3 grid:

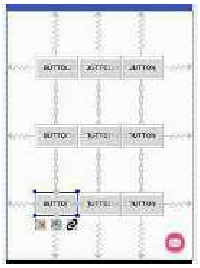


Selecteachrowin turnandusetheChain,CreateHorizontalChainco toconverteachoneintoachainofthreebuttons:

Then select the three chains, i.e. all nine buttons, and use the Chain, Create VerticalChaincommandtocreatethegrid:

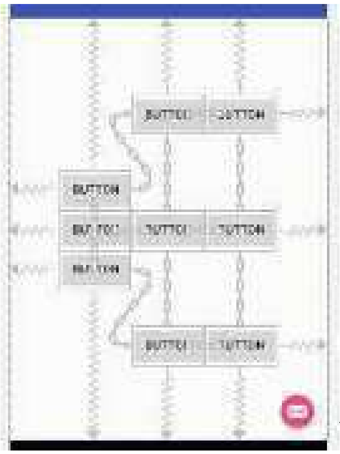


If you need to change the design to Packed basically choose the main button in each line and snap the chain symbol until you get a pressed row:

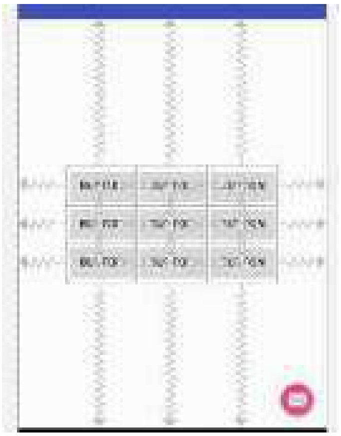


To get a stuffed segment is more troublesome on the grounds that the chain symbol just controls the principal chain that a part is in. At the hour of composing the best way to set

stuffed on the sections is to alter the XML file.
The chain style is constrained by the primary part in the chain. Assuming you check out the XML for the main button in the primary column you will see:

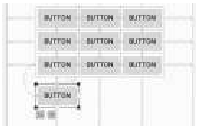
app:layout_constraintHorizontal_chainStyle="packed"
This is liable for setting the principal line to stuffed design. To set the primary section to stuffed you need to change the XML to read:
app:layout_constraintHorizontal_chainStyle="packed"
app:layout_constraintVertical_chainStyle="packed"/> If you do this you will see the first column in packed format:


You need to change the XML for the button at the highest point of the subsequent line and at the top third line to

read:
app:layout_constraintVertical_chainStyle="packed"/>
If you change the top row button's XML correctly you should see a packed array of buttons:


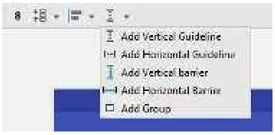
The large issue currently is that to situate the matrix of Buttons you need to change the predisposition settings of the main button in each line, which sets the even inclination for that line, and the settings of the principal button in every segment, which sets the upward inclination. There is no single setting that will situate the whole network, and this doesn't make situating easy.
What about adding the 10th Button?
It might have been added to the principal

segment yet it is similarly as simple to add it now and set a limitation from the lower part of the last Button in the primary segment and from the right and left half of the button:



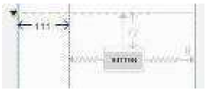
Guidelines

The last situating apparatus you have available to you is the rule. You would now be able to relocate a level or vertical rule on the plan surface and use it to situate different parts. The main thing about rules is that they don’t exist as a View item, or whatever else in the last format. A rule is an item in the Layout Editor and any limitations that you set utilizing it are changed over to situating that makes no reference to the rule when the application is run.
To add a rule you should simply utilize the Guidelines tool:



Once the rule has been added, you can drag it to the ideal area and afterward position parts comparative with maybe it was a part by its own doing. You can set any limitation that you can use with a part, including predisposition. Notice that assuming you move a rule any parts of the UI that are compelled to is will likewise move– this can be very useful.

You can erase a rule by choosing it and squeezing erase. Right now the large issue with rules is that, while the Layout Editor shows the area as you drag, there is definitely not a simple method for entering a worth to situate it exactly:

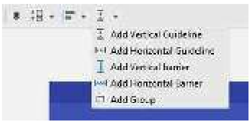


You can enter a precise position assuming you grow the Attributes you can enter a careful worth in the guide_begin quality which you will find under the Constraints set. While rules are another situating instrument, and you can never have too much, there isn't anything you can do with a rule that you can't manage without one. Likewise, considering that rules are situated totally on the screen, they don't give any offices to changing the format as the screen size changes.

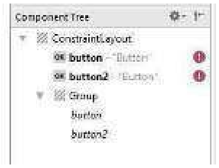
Groups

Notice that this is just accessible in ConstraintLayout 1.1.0 or later – see the beginning of the section for more information.

A Group enables you to make gatherings of parts. Right now this just gives a straightforward method for changing the perceivability of a bunch of parts. You can put a gathering on the plan surface utilizing the Add Group menu command.



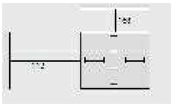
The Group is least demanding to use in the Component Tree.You can drag quite a few parts and drop them on the Group. Notice that there is no sense wherein the parts are offspring of the Group, it’s anything but a compartment. The Group essentially keeps a rundown of the parts that it controls the perceivability of:



In the model displayed over, the Group controls the perceivability of button and button2. In the event that the Group’s perceivability property is set to undetectable then button and button2 don’t really show in the UI however they actually occupy design room. Assuming you set perceivability to gone then they are imperceptible and don’t occupy room in the UI. The Group office is a minor accommodation as you could accomplish the very outcome by composing code that sets the perceivability of every one of the parts. In any case, it very well may be valuable as a method of getting sorted out a complex UI with various arrangements of parts that can be made gone or apparent with a solitary command.

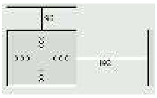
Sizing

In contrast with situating, estimating a part is practically minor yet with some intriguing twists.
You can see how the component is sized in the Attributes window. The type of lines shown inside the component indicate the size of the component and its relation to its content. Straight lines indicate a fixed size:

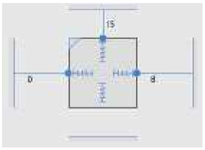


This auto-sizing behavior is set by the layout_width and layout_height properties. You can modify this by typing in an exact size, e.g. 100dp, into

thevalueboxnexttotheproperty.
An option in contrast to fixed size parts is to permit them to
consequently resize to fit or wrap their substance. To set this you should simply tap on the inward straight lines which change to <<< and permit the part to change its size:

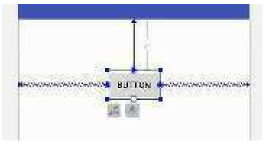


This conduct is constrained by layout_width and layout_height properties set to wrap_content and it is the most normal estimating conduct utilized in Android. That is, most parts change their size to suit their contents. There is a third chance– the size of the part can be set by limitations. Assuming that you click once again on the inner lines they change from <<< to a spring-like graphic:

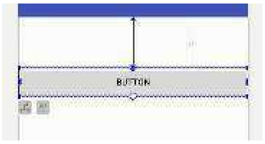


Using Match Constraints you can set measuring freely in the flat or vertical. implying that limitations can be utilized to set the width or the stature of the part. To set the width you need to set a limitation on the left and an imperative on the right and to set a stature you set a requirement on the top and one on the base. Match Constraints is set by setting a decent size of 0dp.

For instance, assuming you have a Button and physically apply a requirement from the right and one from the left to the parent and have wrap_content set as the flat measuring then the Button stays at its unique size, and inclination imperatives are added to split the accessible space in the Button’s position:

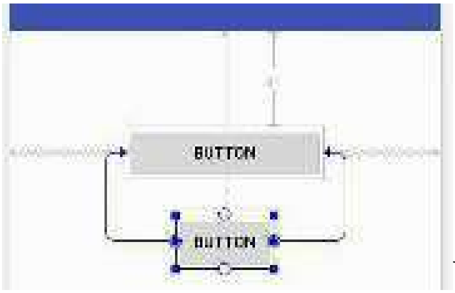


If you leave the limitations set up and change the level estimating to Match Constraints then the button resizes to utilize the even space accessible to it:



The thought is exceptionally basic – the imperatives control the place of the part, or the size of the component.
You can apply Match Constraints to parts that are obliged comparative with different parts just as the parent container.

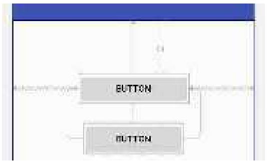
For instance, assuming you have two Buttons and you associate the passed on side of one to the left half of the other and the equivalent for the right side then the compelled Button will situate itself to agree with the center of the other Button:



You can likewise set the arrangement guide utilizing the predisposition control toward cause the

Button to have a rate offset.

If you leave the limitations set up and change the even estimating to Match Constraints, then, at that point, the obliged button resizes to be a similar size as the other button:

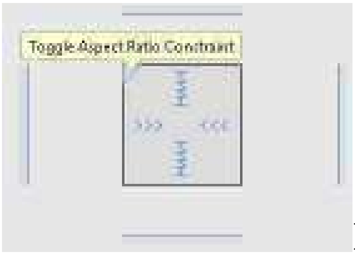


Again, the requirements can be utilized to set either the position or the size. A similar guideline works for chains of parts. In the event that you have a chain set to Spread or Spread Inside, yet not Packed, you can set MatchConstraints and the parts will resize to occupy the space. You can set a

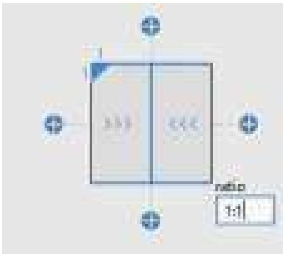
portion of the parts in the chain to Match Constraints, and just those parts will resize to occupy the space allotted to them.

For instance, in this chain of three Buttons just the subsequent two Buttons are set to Match Constraints and the first is set to Wrap

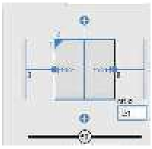
Content:
The last measuring choice can be utilized to set the size of one aspect as a proportion of the other. To make this work, one of the aspects should have its size set in another manner– either by a Wrap Content, a decent size or a constraint.
The aspect that will be set as a proportion must be set to Match Constraints and afterward you click the little triangle toward the side of the Attributes window to flip Aspect Ratio Constraint.
For instance, assuming you place a Button on the plan surface and set its width to Wrap Content, and its tallness to Match Constraint, then, at that point, a little triangle will show up in the top left:



If you click on the triangle, you will set a default proportion for width:height of 1:1.



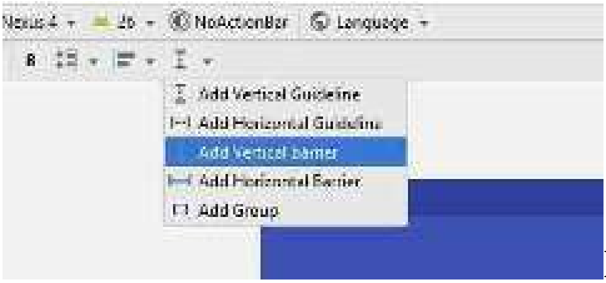
If you type in a worth of say 1:2 then the stature will be set to double the width, which thus is set by the contents:
You can set one of the aspects utilizing limitations. For instance assuming you physically put both ways requirements on the Button and afterward set its flat size to Match Constraints, the Button will be pretty much as wide as the screen. Assuming you currently set Match Constants on the stature and set the Aspect proportion to 2:1 you will get a Button that is just about as wide as the screen and half as high.



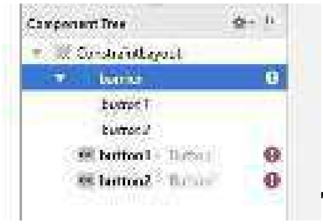
Barriers

Notice that this is just accessible in ConstraintLayout 1.1.0 or later – see the beginning of the part for more information.
Barriers take care of numerous ConstraintLayout issues that would somehow or another be hard to handle, yet they are not something you are probably going to utilize each day.
A Barrier is a brilliant rule. You place a Barrier on the plan surface and afterward intuitive quite a few parts onto it– most effectively inside the Component Tree.This works similarly as the Group part in that the Barrier isn’t a holder of any kind, yet essentially keeps a rundown of parts that it will work with. The Barrier additionally has a bearing and positions itself to line up with the part in its rundown that is the most extreme

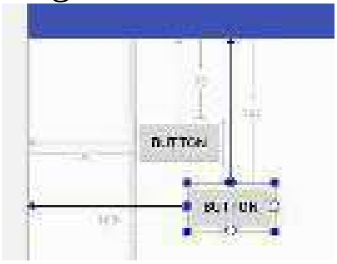
toward that path. That is, assuming the Barrier is set to right it positions itself on the right edge of the part in its rundown that is farthest to one side. If you think of the groupof components that the Barrier has in its list as beingsurrounded by a box that just contains them, a bounding box, then another way of thinking of this is that the Barrier will position itself on one of the edges of the bounding box–the right-hand edge in our example. Like a Guideline you can situate different parts comparative with it by setting imperatives to it. You can see that this permits you to situate parts so they are to the extreme right, left, top or lower part of a group. For instance, utilize the Add Vertical boundary menu choice to put a Vertical Barrier on the plan surface. At this stage you wont have the option to see this is on the grounds that its default bearing is left and it accepts up position as far to one side as possible.



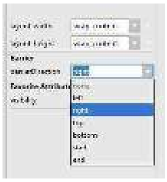
Next place two Buttons on the plan surface and, utilizing the Component Tree simplified them onto the barrier:



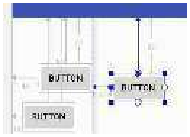
The default Barrier heading is passed on thus it will presently situate itself to be on the left hand edge of the Button that is uttermost to the left. You can see the Barrier as a gray shaded strip in the diagram below:



If you move the two Buttons you will see that the Barrier consistently secures itself to the left most edge in the gathering. You can change this conduct to one more edge by choosing the Barrier in the Component Tree and afterward utilizing the Attributes window to change barrierDirection to right say:



With this change the Barrier positions itself on the right most edge of the group. Now if you place a third Button on the design surface and constrain it to be to the right of the Barrier, then it will always stay to the right of the rightmostedgeofthecomponentsintheBarrier'sl



If you move the two Button 's around you will before long get the possibility that this Barrier is a method of setting parts to one side of the furthest right part of the gathering whichever one this is.Without a Barrier object you would need to place the Button's into a Frame and afterward position different parts comparative with the Frame.This would work yet it makes a settled format.The

Barrier approach keeps the ConstraintLayout flat.
The Barrier is helpful at whatever point you have a gathering of parts that change their size because of client information or information or district or whatever. You can utilize a Barrier to split the screen region into something adequately enormous to oblige the gathering and afterward the remainder of the parts can situate themselves in what is left over.

Constraint Attributes

As you would figure, there are ascribes basically the same as those utilized by the RelativeLayout which set the limitations for a ConstraintLayout. To see each of the properties you need to tap on the View all ascribes symbol at the upper right of the Attributes window– the twofold arrow.
For instance, assuming that you examine the Attributes window you will see:
Constraints

Left_toLeftOf

This prompts you to supply the id of the other control being utilized in the

arrangement– left-hand side to left-hand side for this situation. There are comparable imperatives for different potential outcomes.For example:

Bottom_toBottomOf
thus on.
It is likewise worth arrangement that, in permitting you to set the place of a part by essentially hauling it to the area you need, the Layout Editor is

working out how to set different properties accurately. You could do this physically to get a similar impact, yet the Layout Editor does it just from where you have situated a part. For this reason it is simpler to let the Layout Editor set the properties for you.

Troubleshooting

The ConstraintLayout gives a method of

making responsive designs that change in accordance with the size of the screen. Notwithstanding, doing this viably is troublesome. The measure of knowledge required works out in a good way past what the Infer Constraints device has. It might even be past a human. A less difficult, however more work-concentrated way, is to give a different design to profoundly unique screen sizes and use requirements to make little adjustments.

The ConstraintLayout library utilizes a straight imperative solver to work out the places of the parts as a whole. This then produces a layout that is used in your app that is intended to be fast and efficient. As of now the framework and the Layout Editor are going through quick change and advancement. There are many elements that don’t exactly fill in as promoted, and many elements don’t have the offices expected to make them simple to use. However, ConstraintLayout is the method of things to come by the Android advancement team. Given the upside of delivering a solitary

design that provides food for an assortment of screen sizes and directions it merits persisting with.Be that as it may, it is extremely simple to get into a total wreck in the Layout Editor. One exceptionally normal issue is for parts to clearly disappear. This is typically on the grounds that they are on top of one another or situated off the screen. The most straightforward method for figuring this issue out is to go to the Attributes window and physically reset one of the situating properties.
You will likewise find the accompanying tips useful:

Use situating comparative with another part in the event that it seems OK. That is, assuming you have a text passage part then it seem OK to situate its acknowledge button comparative with its right-hand side.
If you position everything comparative with the parent holder then you adequately have an outright design that indicates the specific and unchanging place of everything.
If the screen size changes then it is conceivable that parts will cover each other in

the event that there isn’t sufficient room. Continuously cause your designs to have a lot of extra space. A good strategy when working with a group of components is to pick one that you position relative to the container, then position all other components relative to it to ensure that you can move the group and keep alignments.
Remember that a few parts can change their size too as area and this can adjust the place of parts situated comparative with them. Automatically produced limitations now and again work, however they are seldom pretty much as legitimate as a bunch of physically made imperatives. If you plan to make use of the layout in the future, then it is worth creating a set of manually applied constraints. Use Infer Constraints gradually as you add parts and afterward physically change what it makes to be more logical.
It is more sensible to deliver a different design asset for little to medium-sized screens and one for enormous screens.

Summary

☐ The ConstraintLayout is the format of things to come and the new Layout Editor was made to work best with it.

☐ There are two programmed imperative devices– Autoconnect which works out requirements for a solitary part, and Infer Constraints which works out any missing limitations for the whole design.Neither one of the apparatuses is as of now especially useful.

☐ You can get all free from the limitations in a format, or only those on a solitary part or an imperative at a time.
☐ Constraints can be applied from a part to the parent compartment and these carry on like outright positioning.
☐ You can set default edges to make situating parts more regular.
☐ Components can be situated comparative with another component.
☐ You can't set negative margins.

☐ You can adjust text baselines.
☐ Bias requirements permit you to set places that partition the space accessible in a predetermined proportion.
☐ Chains give a portion of the highlights of a LinearLayout.For instance, you can utilize a chain to disperse parts across the accessible space.
☐ Guidelines can be added to a layout and components can be positioned relative to them, even though they don’t appear in the final layout.
☐ Groups can be created which allow you to set the visibility of all of the components in one go.
☐ You can set the size of a component absolutely or to be determined by the content. In addition, you can allow a pair of constraints to determine the size.
☐ An Aspect Ratio Constraint can set one dimension as the ratio of another.
☐ A Barrier is a smart guideline that tracks a specified group of components and is positioned at the most extreme edge of the group in a specified direction. You can position other components relative to the

Barrier.

Chapter 8 Programming The UI

If you want to be a really good Android programmer,not only do you need to know how to create a UI, but also how the UI is created. To be really confident in what you are doing, you need to understand some of the inner workings of the Android graphics system. This is also essential if you want to modify the UI in code and work with menus.

A UI Library

There are lots of different UI construction kits for Java and other languages, AWT, Swing, Qt, MFC, WPF and on, and you might think that mastering them all would be a difficult, if not impossible, task. In fact it is a lot easier than you might think because most UI libraries use the same general approach and the Android UI library, which doesn't seem to have a given name, is no different. Let's take a careful look at how it works.

An Activity has a window associated with it and this is usually the entire graphics screen of the device it is running on. In other words, an Activity can allow other objects to draw on the device's screen. However, rather than simply providing direct access to the graphics hardware, there is an extensive set of classes that make building a UI and performing graphics operations easier.

Before we look at general graphics we need to first find out how the UI is constructed.

The View

The basis of all UI components and general 2D graphics is the View class. This is a general-purpose class that has lots and lots of methods and properties that determine how it will display the component or other graphics entity it represents. It also takes part in the event handling system, which means Views can respond to events. There are View classes that implement all of the standard components that you make use of in the Android Studio Layout Editor,i.e.Button,TextView and so on.

Every View object has an onDraw method that can draw the graphic representation of what it represents onto a Canvas object which is essentially a bitmap with drawing methods. What happens is that the Activity calls the View’s onDraw method when it needs to update the UI and passes it a Canvas object that it then renders to the screen– you don’t have to worry about how the Canvas is rendered to the screen at this level. You can think of this as, “every View object knows how to draw itself”.
To summarize:

An Activity can be associated with a View object.
When the Activity needs to draw its UI it

calls the View object’s onDraw method e.g.view.onDraw(Canvas).

The View object then draws on the Canvas whatever it needs to, whether a button, text or something else.
The Activity then displays the Canvas object on the screen.

An Activity can only be associated with a single View object, which determines what is drawn on the screen. This might seem a bit limited but, as you will see, it is far from limited because View objects can be nested within one another.

Using setContentView

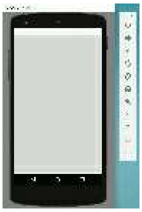
How do you set a View object to show in the Activities window? The answer is that you use the Activities setContentView method, which is what we have been doing all along.

To see this in action, start a new Basic Activity project
and change onCreate to read:

```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState) val b = Button(this)
setContentView(b)

}
```

Don’t forget to use Alt+Enter to add the import statements needed to allow you to use the Button class, and don’t leave any code in onCreate that would use other View objects such as the menu.

The first instruction creates a Button object, which is a subclass of View, and the second sets this as the Activities View. If you run this program what you will see is a gray area that fills the entire screen:



Yes, this is the button! You can even click it although, with no event handler, nothing

happens.
To make this button a tiny bit more interesting we can customize it by setting properties.For example:

```
val b = Button(this)
b.text="Hello
Button"
setContentView(
b)
```

If you run this you will see a button that fills the screen with the caption "HelloButton".

Don ’t bother setting any layout properties in code because at the moment there is no layout in force so they will be ignored. How to activate a layout is our next topic.

The ViewGroup

If an Activity can only show a single View object, how can we ever create a complex UI

with multiple buttons, textViews and other components? The answer, and you probably already guessed it, is that there are Layout, or ViewGroup, objects which can be used to host other View objects. You already know about using Layouts in the Layout Editor or in an XML file, but they, like all UI elements, correspond to particular classes that do the actual work.
A ViewGroup can display multiple View objects. So in nearly all cases the View object that is associated with an Activity is a Layout View. When the Activity asks the Layout View to render itself, by calling its onDraw method, the Layout calls the onDraw method of each of the View objects it contains and puts them together to make a single result. Of course,it also performs a layout operation positioning and sizing the View objects it contains.

So a Layout does two things:
it hosts other View objects
it performs the layout function after which it is named.

To see this in action try:

```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState) val linLayout= LinearLayout(this)

val b = Button(this)
b.text="Hello Button"
linLayout.addView(b)
setContentView(linLayout)

}
```

If you run this program you will see a button at the very top left of the screen.

The first instruction creates a LinearLayout object. This is a subclass of View that can contain other View objects and it organizes them in a left to right or top to bottom way depending on the setting of its orientation property. Next we create a button object and then use the standard addView method of the LinearLayout to add it to the layout.
All Layouts have an addView method, which can be used to add multiple View objects.

You can add more buttons to see how the default LinearLayout works: override fun onCreate(savedInstanceState: Bundle?) {

```
super.onCreate(savedInstanceState)
val linLayout= LinearLayout(this)

val b1 = Button(this) b1.text="Hello
Button 1"
```

```
linLayout.addView(b1)

val b2 = Button(this) b2.text="Hello
Button 2"
linLayout.addView(b2)

val b3 = Button(this) b3.text ="Hello Button 3"
linLayout.addView(b3) setContentView(linLayout )

}
```

Programming Layout Properties

At the moment we are relying on default settings for the properties,and the layout properties in particular,of the View objects we are creating. However, in practice you could spend the time and lines of code to set all of the properties needed to create any user interface and layout you wanted to.

You now know how to create a UI completely in code. All you have to do is create all of the objects you need, set their properties and add them to suitable layout objects. This is a little more complicated than you might think because each layout type has a different set of layout properties. Exactly how this is done is easy enough, but if you don’t want to know about it at this stage you can skip ahead as it doesn’t change any of the general principles.
As explained in the previous chapter, each type of Layout has an associated class derived from LayoutParams called layout.LayoutParams where layout is the name of the Layout class. For example, LinearLayout has the LinearLayout.LayoutParams class, which is used to define all of the layout properties that a View object can use when added to a LinearLayout object.

You can probably guess how to make use of the LayoutParams class. All you do is create a correctly initialized instance of the appropriate LayoutParams class and use setLayoutParams on any View object you want to customize.

For example, to set the height and width in a LayoutParams object we could use:

```
val LP= LinearLayout.LayoutParams(100,100)
```

There is a constructor for all of the LayoutParams classes that accepts just the width and the height properties. Once you have a LayoutParams object you can assign it to any View object by setting the View object’s
LayoutParams property:

```
b3.layoutPar
ams =LP
linLayout.ad
dView(b3)
```

With this change the third button in our previous layout will be exactly 100 by 100 pixels.
Notice that the constructor works in pixels, i.e. px, instead of deviceindependent pixels, dp.You can also use constants for MATCH_PARENT and WRAP_CONTENT. For example:

```
val
LP=LinearLayout.LayoutParams(WRAP_CONTENT,WRAP_CONTENT)
```

There is also a constructor that allows you to set the weight. Other layout properties have to be set using properties after the constructor has done its job. Some properties might have to set using set property methods. For example:

```
LP.setMargins(20,20,20,20)
```

which sets the left, top, right and bottom margins accordingly:

More complex Layout objects have correspondingly more complex LayoutParams that you have to spend time setting up.

So to be clear– there are properties such as text that you set directly on the View object, but there are also Layout properties that you have to set on an appropriate LayoutParams object, which is then set as the View object’s LayoutParam property.

The View Hierarchy

Notice also that a Layout can contain other Layouts and so the set of View objects that make up a UI is structured like a tree, the View hierarchy. When the screen is redrawn each View object is asked to draw itself and this is done for all View objects in the hierarchy from top to bottom. Normally the View hierarchy is drawn just once when the Activity loads. If an area of the screen is obscured by another graphic for any reason, the redraw is clever enough not to draw the entire View hierarchy. It only redraws View objects that intersect with the invalidated area of the screen. The View hierarchy is also involved in passing events between objects and in determining which component has the current focus.

XML Layout

So far the principles of the graphic system are simple enough. Every control or component corresponds to a View object and you can build a UI by creating View objects in code and adding them to Layouts. You control the way the View objects are arranged using the LayoutParams object or by directly setting properties. An Activity will draw its View hierarchy to the screen when it needs to.OK, this is how to create a UI in code,but so far we have been building a UI using the Layout Editor. How does this relate to the View hierarchy?

The Layout Editor creates an XML file which describes the View hierarchy that you want to create. The way that this works is fairly obvious. Each tag in the XML file corresponds to a View object for which you want to create an instance.For example:

```
<LinearLayout>
</LinearLayout>
```

creates an instance of a LinearLayout object. Nesting tags within a layout indicates that the objects created need to be added to the layout as child Views. For example:

```
<LinearLayout>

<Button />
</LinearLayout>
```

creates a LinearLayout object and a Button object and then adds the Button object to the

LinearLayout using its addView method. You can see that the XML captures the idea of the View hierarchy perfectly.

To set object properties all you have to do is is use the corresponding attributes in the XML. For example to set the button’s text:

```
<Button


android:text=“New Button”
/>
```

Layout parameters are set using properties prefixed with layout_property. For example:

```
<Button


android:layout_width= “wrap_conte
nt”
android:layout_height=“wrap_conte
nt”


/>
```

That is really all there is to it. The XML defines a hierarchy of objects and their properties and the system reads the file and creates the objects.This use of XML as an object instantiation system is not an uncommon one. Of course, the XML created by the Layout Editor looks a lot more complicated than the examples above, but this is mainly because of the number of attributes it defines. The basic idea is still the same.

Inflation Theory

The final big question to be answered is how does the XML get converted into a real object hierarchy? The answer to this is to use an“inflater”. To inflate a layout is Android jargon for instantiating the objects defined by an XML file. You normally don’t have to call an inflater because the system does it for you behind the scenes, but you can if you want to. For example, to inflate a layout you would use an instance of the LayoutInflater. Normally you wouldn’t create a fresh instance. Instead you can simply use an existing one supplied by the system using the LayoutInflater property of the Activity. Once you have the LayoutInflater you can use one of its many inflate methods to create a View object hierarchy as specified by the XML.

Which method you use depends on where the XML is stored. You can simply supply a resource id for an XML file included in the res directory. For example, to inflate the usual activity_main.xml layout you can use:

```
val inf = layoutInflater
val myView = inf.inflate(R.layout.activity_main,null)
setContentView(myView)
```

The second parameter of inflate can be used to provide a View object to act as the root container for the inflated View hierarchy. The container is just used as a reference“parent” for the purposes of calculating the layout. That is, it simply provides the container that everything has to fit into according to the layout rules. Of course this is entirely equivalent to the usual:

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

which calls the LayoutInflater and sets the view in a single instruction. The only reason that you would manually inflate an XML layout is if you wanted to do something clever, such as put one layout together with another or in some way manipulate the View hierarchy.

Notice that there are other types of inflater objects, e.g. the Menu inflater, which do the same job of converting XML to instantiated objects with the given properties. We will come back to these more specialized inflaters when we look at menus in the next chapter. There is also a version of the inflate method: `inflate(R.layout.activity_main,root, true/false)` which will inflate the XML resource using root as its container for the purposes of layout if the last parameter is false, and it will add the inflated View to the root if the last parameter is true.

Finding View objects

One problem we have to solve if you want to work with the View hierarchy created by an inflater is finding View objects in the hierarchy. In the example where we built the View hierarchy in code it was easy to keep track of a button or a textView by simply keeping a reference to when it was created. An inflater simply returns the View hierarchy without an easy way to get at a particular object, a button say. One way of solving the problem would be to "walk" the View tree. A ViewGroup object, e.g. a Layout, not only has an addView method but also a range of methods that allow you to access the objects it contains. Each child object is assigned an integer index– think of it like an array. The method:

```
getChildAt(i)
```

will return the child object at index i. You can also use:

```
getChildCount()
```

to find out how many child objects are stored in the container.

Using these methods you can search the hierarchy for the View object you want but how do you know which one it is? The answer is that all View objects have an id property which should identify them uniquely. The id property is set as part of the XML file.

To avoid you having to work out an id value, the standard way of setting an id is to define a resource within the XML file:

```
<Button
android:id="@+id/my_button"
```

When the XML file is inflated the @+

symbol is inteRpreted as “create a resource”.An integer id is generated using the generateViewId method and this is used to both create the id property and to add a my_button property to the id property of the R object, R.id.

If you are using Kotlin to work with the XML file it automatically converts all of the string labels on the ids to Activity properties and then makes them reference the objects that the inflater creates. To allow this to happen you have to enable the kotin-android-extensions plugin– which is enabled by default in a Kotlin project. You can then specify which layout files you want to create properties for using:

import kotlinx.android.synthetic.main.*layout*.* So to import properties for all of the View created by the two standard XML files main.activity_main.xml and main.content_main.xml you would use:

```
import
kotlinx.android.synthetic.main.activity_main.* import
kotlinx.android.synthetic.main.content_main.*
```

These are automatically added to your project file when it is created. Any other layout

resource files that you create will also be added automatically so that their ids are properties also.
The only minor complication is that when you set an id using the Layout Editor it will auto generate the @+id/ for you.So in the Attributes window you will see my_button not @+id/my_button which is what is entered into the XML file. This is helpful, but it can be confusing.
There is a lot more to say about resources, but for the moment this is enough to understand what is going on. Resources deserve a chapter all to themselves and you’ll come to it after we’ve looked at menus.
What all this means is that not only do you get an autogenerated id value, but also a way to get this value into running code. You could use the getChildAt methods to step through all of the View objects in the hierarchy, but it is much easier to use:

```
findViewById<Button>(R.id.my_button)
```

which returns the object in one instruction. If you are not using Kotlin’s conversion of ids to properties then this is the only sensible way to work and Java programmers make use

of the findViewById before they can work with any View object in code.
The general method, in Kotlin, is to inflate the XML, set it as the content of the View, and use the generated Activity properties to work with any View object you want to.

How to Build a UI?

You now have two distinct approaches to building a UI. You can do the whole job in code or you can create an XML layout. In practice it is usually easier to use the Layout Editor to generate the XML file for you. You can, however,mix the two approaches and change a UI “on the fly”. For example you can load a UI by implicitly or explicitly inflating an XML file and then

writing code to create and add other components or even remove View objects from the layout. To remove View objects you simply use the removeView or removeViewAt methods of the ViewGroup object.

There are other UI components such as menus that the Layout Editor doesn ’t support. In this case you have to work with the XML or create the UI in code. This means you do need to know the inner workings of the View hierarchy even though the Layout Editor is the easiest way to create a UI.

Summary

☐ All of the UI components are derived from the View class.
☐ An Activity can host and display a single instance of the View class set by one of its setContentView methods.
☐ You can create instances of View objects in code and set them to be displayed by the Activity.

☐ A Layout or ViewGroup object is a View object that can contain many View objects, so creating a sophisticated layout that the Activity can display.

☐ Each Layout has its associated LayoutParams class, which is used by each View object it contains to control how it treats it within the layout.

☐ You generally have to create an instance of the LayoutParams class, set the parameters you want to determine, and then set the instance as the LayoutParams of each View object that needs to use it via the LayoutParams property.

☐ The use of Layout containers results in a View hierarchy, i.e.
hieayoiewojewhhedpybheiv

☐ You can also code the View hierarchy using XML. Each XML tag corresponds to a View object and they are nested to define the hierarchy. The properties of the objects are set within the XML as attributes. Layout properties are treated in the same way but with layout_ as a prefix.

☐ When the time comes for the View hierarchy to be assigned to the Activity, the

XML file is converted into a nested set of View objects by the use of an inflater method. This simply reads the XML and converts each tag to an object and sets the object’s properties.

☐ To find particular View objects in an inflated hierarchy the usual approach in Kotlin is to generate properties corresponding to the ids.If you don’t want to do this then you have to use use the findViewById method.

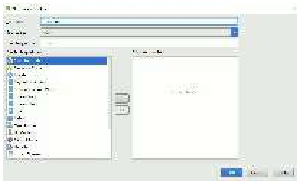
Chapter 9 Menus – Toolbar

A UI isn ’t just made up of buttons and other widgets or components; the menu is still a useful way of letting the user select what happens next. Android’s menu system is easy to master.We also need to find out about the Toolbar implementation of the action bar. There is a menu Layout Editor in Android

Studio 3.0 and it works quite well, but it is still worth knowing how to create the XML yourself. The basic principles of menu creation are the same as for building a UI in that a menu is a collection of View objects. You can create the View objects in code or you can use an XML file and a special inflater, a MenuInflater, to convert it into the objects. Defining a menu is more or less the same process every time, although the way in which you use the menu varies according to where and when the menu is shown, but even this follows roughly the same steps. Let’s look a the general idea first.

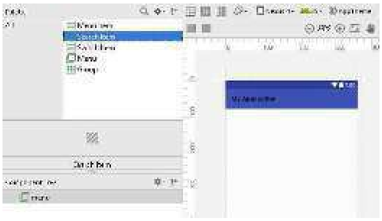
Creating a Menu Resource

Menus are generally defined by a menu resource file, which is an XML file whichis rendered to createthemenu.All menu resources arestoredin the app\res\menu directory. Ifyou right click on this directory you can select the New, Menu resource option and type in a name–all lowercase as usual:



You can ignore the Available qualifiers for the moment. The idea is that each resource you create will be used in a given situation according to the qualifiers you select. In this way you can create custom menus for particular languages for example. More about this idea in Chapter 11.

When you click the OK button the resource file is created and it will be opened in the layout editor. This in principle should allow you to edit the menu using drag-and-drop just like a general layout, but at the moment it is very limited and tends to create XML that doesn’t work. You can place a menu item onto the menu and customize some of its properties, but this is about as far as it goes:



You can work either with the menu editor or the XML directly. You will discover that the XML Editor suggests auto-completions so you aren’t entirely without help even if you have to abandon the menu editor.
Before we can create a menu we need to know something about the XML tag.and the corresponding menu items./

The Menu Tree

A menu is a hierarchy of options. The top-level menu presents a set of items. If any of the items is itself a menu, i.e. a submenu, then it can contain more items.
Android menus make use of three objects, and hence three XML tags:

<menu>
<item>

<group>

At the top level the <menu> tag inflates to a Menu object which is a container for menu items and group elements. The <item> tag inflates to a MenuItem object which is a single option in the menu. It can also contain a <menu> tag which can in turn contain more <item> tags to create a submenu. The <group> tag doesn’t inflate to an object. Instead it sets the group id of all of the items it contains. You can set various properties of a group of items in one operation i.e. they act as a group.
There are a range of attributes that can be used with <item> and <group> and not all can be used in every situation. The three that you need to know about are:

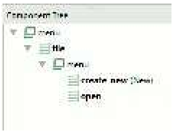
id - an integer that you use to identify the menu item title - a string that determines what the menu item displays icon - a drawable image used when the menu item can be displayed as an icon.

With all this explained let ’s define a menu with a single File item and a submenu consisting of two items, New and Open:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android=

"http://schemas.android.com/apk/res/android">
<item
android:id="@+id/file" android:title="File"> " android:title="Open"
/>

<!- "file" submenu -->
<menu>

<item
android:id="@+id/create_ne w" android:title="New" />

<item
android:id="@+id/open

</menu>
</item>
</menu>
```

You can see in the example that we have a top-level menu item that we don't bother giving an id which contains a single <item> tag. This displays the text File when the menu is displayed. In case you have forgotten the notation"@+id/name" automatically creates the id resource name and sets it to a newly generated integer value. You can then use name in code to find and work with the menu item.

The <item> tag contains another <menu> tag which in turn contains two more items that correspond to New and Open file menu options.You can see the structure in the Component Tree:



The Component Tree can also be used to do some limited editing of the menu by dragging-and-dropping to modify how items are nested. Exactly how this simple menu looks depends on how it is displayed, but it defines a menu with a single top-level File option which when selected displays a submenu of two other options, New and Open.

Displaying a Menu

There are four different ways you can display a menu:

1. Action bar such as the App Bar

2. Context menu
3. Contextual action Bar (CAB)
4. Popup

The action bar or App bar was introduced in Android 3 as a replacement for the original options menu and has become the standard primary menu for apps. With Android 5 a new way of implementing it was introduced, namely the Toolbar, which is a standard widget that can be edited using the Layout Editor. This makes it easier to integrate and work with. If you want to use the ActionBar in earlier versions of Android you need to make use of the Support Library – this is automatically included when Android Studio creates a project for you.

The context menu is a popup menu that appears in response to a long click on a component of the UI.

The Contextual Action Bar appears at the top of the screen when the user long-clicks on a UI element and it is supposed to be used to provide actions that are appropriate for the item that has been selected. It too needs the Support Library to work with older versions of Android.

The popup menu can be displayed in response to almost any user action you care to use. It appears next to the View object that causes it to be displayed. For example you could have a button that displays a popup when clicked or a popup could appear when a user types something into a text field. It is difficult to know exactly when to use a popup menu. Logically the next step would be to use the XML file we just created to display either a CAB or a popup, but it is better to start with the default project type as it generates the code and menu resource file needed to implement an action bar, the Toolbar, and it is used in most applications. The remaining menu types are the topic of the next chapter.

Using the Toolbar

If you start a new Basic Activity project called MenuSample then you will discover that it automatically creates a main_menu resource file and the code needed to display it as a Toolbar.

The Basic Activity template uses the support library to make it possible in earlier versions of Android. This is why the MainActivity class in your project has to derive from the AppCompatActivity class and not the more basic Activity class:

```
class MainActivity : AppCompatActivity() {
```

and why we need the imports:

```
import
android.support.design.widget.Snackbar
import android.support.v7.app.AppCompatActivity
```

The Toolbar is defined in activity_main.xml as a custom widget:

```
<android.support.design.widget.AppBarLayout
android:layout_width="match_parent"


android:layout_height= "wrap_content"
android:theme="@style/AppTheme.AppBarOverlay">
app:popupTheme="@style/AppTheme.PopupOverlay"
style="@style/AppTheme" />

<android.support.v7.widget.Toolbar
android:id="@+id/toolbar"
android:layout_width="match _parent"
android:layout_height="?attr/a ctionBarSize"
android:background="?attr/colorPrimary"

</android.support.design.widget.AppBarLayout>
<include layout="@layout/content_main" />
...
```

Notice that the Toolbar ’s style is set to AppTheme. This is important as the style selected changes the way the Toolbar is

displayed and can stop it from displaying altogether.
Also notice the tag:

```
<include layout="@layout/content_main" />
```

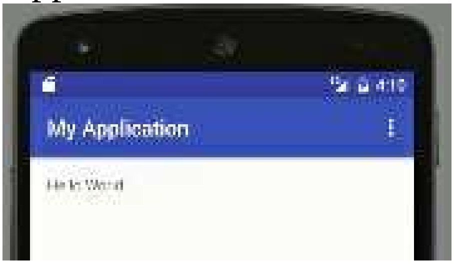
This loads the layout that you design in content_main.xml. As explained in earlier chapters, the layout files are split into two parts– activity_main.xml which defines the layout that should be common to all Android apps, and content_main.xml which is used for the layout specific to your app. The two files are automatically merged together when activity_main.xml is loaded.
When the Activity loads, onCreate runs and inflates the layout in activity_main.xml and the included content_main.xml. This is enough for the Toolbar to display, but for it to be used as an action bar menu by the system we need to add:

```
setSupportActionBar(toolbar)
```

This is generated for you automatically and you will find it in the MainActivity.java file in onCreate:

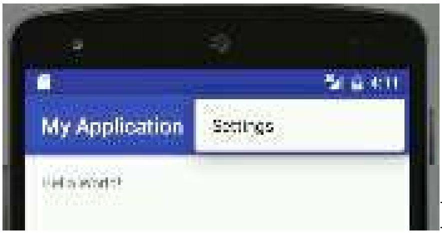
```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)
```

Now if you run the app you will see the familiar Hello world message and the default App bar:



In fact, this toolbar is so familiar you may not even have realized that it is a menu. It has the name of the app to the left and a three dot icon to the right.

If you select the three dot icon the Settings menu item appears:



If you take a look at the menu_main.xml file you can see the definition of this menu:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
```

```
tools:context="com.example.mikejames.myapplication.MainActivity”>

<item
android:id=“@+id/action_settings”
android:orderInCategory=“100”
android:title=“Settings”
app:showAsAction=“never” />

</menu>
```

You can see that this defines a single menu with a single item with the title “Settings”. There are some new attributes being used in the item tag.The showAsAction attribute is important for the way the Toolbar,or action bar,

works.By default the system places menu items into the overflow area that is only revealed when the user selects the three dot icon or more generally the action overflow icon. However, for items that you would like to give the user more direct access,you can set the showAsAction attribute.

This can be set to any of the following:
ifRoom - show if there is room
never - never show in the visible area
withText - show with text
always - always show even if it means

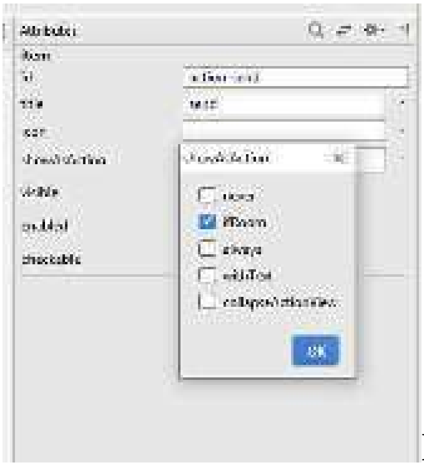
overlapping other items collapseActionView - show a collapsed view of the item.

As you can see, the Settings item in the default Toolbar is set to never show.
The showAsAction attribute works with the orderInCategory attribute to determine the order in which items are shown.

To see this in action let’s add another item, one to perform a Send, to the end of the menu_main.xml file before the final </menu> tag:

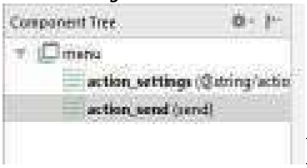
```
<item android:id=“@+id/action_send”
android:title=“Send”
app:showAsAction=“ifRoom” />
```

You can do the same job using the Menu editor. Simply drag-and-drop a Menu Item from the Palette to the Component Tree and use the Attributes window to customize it:

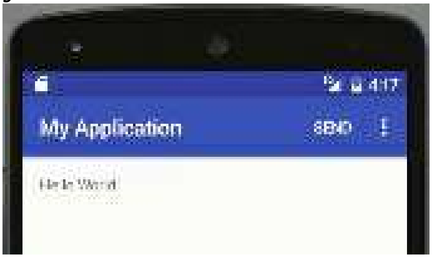

Attributes
item
id action_send
title send
icon
showAsAction
visible
enabled
checkable

showAsAction
[ ] never
[✓] ifRoom
[ ] always
[ ] withText
[ ] collapseActionView
OK

No matter how you do the job the result is the same:


Component Tree
menu
action_settings (@string/actio
action_send (send)

Now if you run the app you will see:

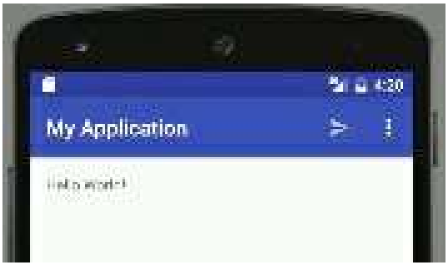


The new Send item will be displayed as long as there is room. If there isn ’t it will appear when the user selects the three dots icon.
It is usual to show toolbar items as icons so

change the item tag to read:

```
<item android:id= “@+id/action_send”
android:title=“Send” app:showAsAction=“ifRoom”
android:icon=“@android:drawable/ic_menu_send”/>
```

which specifies one of the many supplied icons. You can also use the property window to select ic_menu_send in the Resources window:



You can carry on adding menu items to the Toolbar and customizing how they display given different screen widths. The general idea is the same for all menus.

Creating the App Bar

So far we have just looked at the menu_main.xml file and the XML specification of the action bar menu. There is also some code generated to actually create the menu displayed in the App bar. The Activity will fire a CreateOptionsMenu event when it is ready to display the App bar - recall that before Android 3 there was an options menu rather than an action bar.

All the onCreateOptionsMenu event handler has to do is inflate the XML file that defines the menu:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
menuInflater.inflate(R.menu.menu_main, menu) return
true

}
```

The onCreateOptionsMenu is called once when the Activity starts.Before Android 3 it was called each time the menu was displayed on the screen, but now the App bar is always on display. How to change a toolbar is discussed later.

All the event handler has to do is use the appropriate inflater object to create the menu from the resource file. The new View hierarchy of menu and item objects is added to the menu object passed to the event handler. This is important because it means that your new menu items are added to any that are already there. This allows other Activities and Fragments to add items to a common toolbar.

Where's My Toolbar?

If you try any of this out then there is a chance that you will do everything correctly and yet your Toolbar will not show.
There are only two common reasons for this:

1. You are targeting and using an early version of Android which doesn’t support the action bar. This is unlikely, especially if you are using the Support Library.

2. Much more common is that you are using a theme that doesn’t support the type of action

bar you are using.
The solution is easy. For Android 5 or later, or when using the Support Library,select one of the AppCompat themes if you want to make use of the new features introduced with Android 5. As always, Android Studio generates the correct XML for the styles that work with the Toolbar.

Responding to Menu Events

You can attach click event handlers to the various items in a menu in the usual way. This is something that is often overlooked because there is a simpler and standard way of doing the job. However, it is worth seeing the direct method if only to convince yourself that a menu is just another part of the View hierarchy that your application displays. If you aren’t fully familiar with how to attach a click event handler to a View object, refer back to Chapters 4 and 5.

When you inflate a menu resource in onCreateOptionsMenu what happens is that the View hierarchy that is created when you inflate the XML is added to the current menu. This is passed into the onCreateOptionsMenu event handler in the menu parameter:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
```

Next you use the inflater and inflate the resource:

```
menuInflater.inflate(R.menu.menu_main, menu)
```

After this, menu contains the View hierarchy, that is the entire menu as created so far.
At this point you might be thinking that you can access the menu items using

Activity properties that Kotlin adds. You can’t and you can’t import the menu XML file as you can with a layout. Instead you have to use findViewById however you can’t do this because the menu hasn’t yet been added to the complete View hierarchy. The menu is only added to the View hierarchy after the onCreateOptionsMenu event handler finishes. That is, the menu’s View hierarchy is built up in menu and this is added to the app’s full View hierarchy only when the onCreateOptionsMenu event handler finishes.

To allow you to find the menu item while in onCreateOptionsMenu, the menu object has its own findItem method. So to find the MenuItem that corresponds to the item with id action_send you would use:

```
val mItem = menu.findItem(R.id.action_send)
```

Now that you have the MenuItem object corresponding to the Send item you can add a click event handler:

```
mItem.setOnMenuItemClickListener {item→>

process event
fals
e
}
```

Now the event handler will be called when the Send menu item is selected by the user. The event handler should return true if it has consumed the event and false if it wants to pass it on to other handlers.

The entire onCreateOptions Menu method is:

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
menuInflater.inflate(R.menu.menu_main, menu) val
mItem = menu.findItem(R.id.action_send)
mItem.setOnMenuItemClickListener {item
>
process event false
}
return true

}
```

You can also add an event handler for the menu in the XML resource using the android:onClick attribute. The event handler has to have the same signature as the one demonstrated above,i.e.it has to return a boolean and have a single MenuItem parameter.
For example:

```
fun myOnClick(
item:MenuItem):
Boolean { return true

}
```

and:

```
<item android:id= “@+id/action_send” android:title=“Send”
app:showAsAction=“ifRoom”
android:icon=“@android:drawable/ic_menu_send”
android:onClick=“myOnClick”

/>
```

You can use this method to connect as many individual event handlers as you

require for each of the menu items.
This is not the way it is usually done. It tends

not to be a good idea to attach event handlers to the click events of each menu item.

Instead the Activity has an onOptionsItemSelected event handler method which is called when any of the items in the menu is selected. Obviously this saves a great deal of effort because you just have to implement a single event handler– and you don’t even have to hook it up to the menu.

Android Studio automatically generates an event handler ready for you to use:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean { return
when

(item.itemId) {
R.id.action_settings -> true
else -> super.onOptionsItemSelected(item)

}
}
```

This just handles the single autogenerated Settings menu option, but you can see the general principle. The event handler is passed the menu item that has

been selected - the actual object not just its id. You can then use the menu item’s ItemId property to retrieve the id and you can then test it against the ids that you assigned in the resource file.

So in our simple example with a Settings and a Send item we might rewrite the generated event handler as:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean { return
when


(item.itemId) {
R.id.action_settings ->
true R.id.action_send -> {

perform send action
true
}
else -> super.onOptionsItemSelected(item)

}
}
```

You can see the overall thought – test for every one of the thing’s id in every one of the provisos and return valid assuming you have handled the thing occasion. This is the standard method of handling thing occasions in menus, that is with a solitary occasion controller and a, potentially huge, when statement.

Changing Menus in Code

A menu is only a View progression thus you can make changes to it very much like any View order by adding and redoing View objects. Nonetheless, menus have some additional contemplations since they are shown in a marginally unique manner to the remainder of the UI. The issue is that the menu things are not in every case part of the View progressive system. They are made when the menu is shown. This implies that you may attempt to alter them before they are available thus cause an application crash. The way to adjusting a menu on the fly is the onPrepareOptionsMenu occasion controller. This is called not long before the menu is shown and the menu View objects are remembered for the View chain of importance. The onCreateOptionsMenu occasion possibly fires once when the menu is made,however the onPrepareOptionsMenu is called each time the menu is redisplayed. Consequently you can utilize it to make changes to the menu.

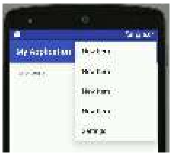
Finding a straightforward illustration of its utilization is troublesome as we will find, so we should simply add another thing through the
onPrepareOptionsMenu. Select a reasonable area in the class and right snap, select Generate and afterward Override strategy. You can choose onPrepareOptionsMenu from the rundown and Android Studio will make a stub for you:

```
abrogate fun onPrepareOptionsMenu(menu: Menu?): Boolean { return

super.onPrepareOptionsMenu(menu)
}
```

Now we should simply utilize the add strategy to add another thing. There are various over-burden adaptations of add that permit you to determine the thing exhaustively. The easiest is simply add(CharSequence) which adds a thing with the predefined title:

```
supersede fun onPrepareOptionsMenu(menu: Menu?): Boolean {
menu?.add("New
Item”)
return super.onPrepareOptionsMenu(menu)

}
```

Now assuming you run the program you will find that each opportunity you select the Settings menu two or three New Items are added to the menu:



What is continuing? The appropriate response is that each time the menu is drawn the onPrepareOptionsMenu is called. For instance, assuming you make the Settings menu be shown, this overwrites the menu show and thus it must be redrawn and onPrepareOptionsMenu is called.
If you need to alter the menu in this manner you really want to check in the event that it has effectively been made.As such, you really want to actually look at the situation with the menu to check whether the thing you need to add is now there.

A somewhat more practical model is to add and eliminate a menu thing relying upon the setting of a checkbox. Add a checkbox and change the technique to read:

```
abrogate fun onPrepareOptionsMenu(menu: Menu?): Boolean { if

(checkBox.isChecked()) {
menu?.add(Menu.NONE, 10, Menu.NONE,“New Item”)
} else {
menu?.removeItem(10)
}
return super.onPrepareOptionsMenu(menu)
}
```

Notice that every one of the things added are given id 10 – there can be more than one menu thing with a similar id. If the checkbox isn't checked then the menu item with id 10 is removed. Assuming that there is no menu thing with id 10 nothing occurs and assuming there are more than one simply the first is eliminated. Utilizing this you can add different New Items and eliminate them by basically seeing the Settings menu which refutes the menu display.
In a more reasonable application you wouldn’t trust that the menu will be nullified by a client activity. You would call invalidateOptionsMenu() when you needed

the onPrepareOptionsMenu to be called.So maybe a superior model is to add a button that calls invalidateOptionsMenu to refresh the activity bar:

```
button.setOnClickListener { view-> invalidateOptionsMenu() }
```

and have the onPrepareOptionsMenu possibly add the thing assuming it isn't as of now in the menu:

```
abrogate fun onPrepareOptionsMenu(menu: Menu?): Boolean { if


(checkBox.isChecked()) {
if (menu?.findItem(10) == invalid) { menu?.add(Menu.NONE, 10,
Menu.NONE,"New Item")


}
} else {
menu?.removeItem(10)
}
return super.onPrepareOptionsMenu(menu)
}
```

Finally, how would you set different properties of the MenuItem you have added? The appropriate response is that add returns the MenuItem made. So to set the new thing to show in the activity bar you would utilize something like:

```
val menuItem= menu?.add(Menu.NONE, 10, Menu.NONE, "New Item")
menuItem.setShowAsAction(MenuItem.SHOW_AS_ACTION_AL WAYS)
```

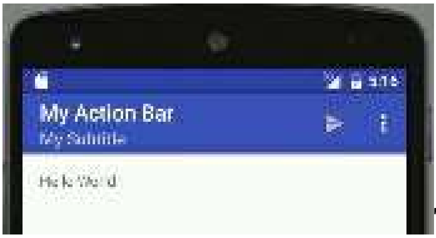
Dynamically adjusting the menu involves

monitoring its present status and really at that time changing it.

Controlling the Toolbar

There are two or three inquiries that actually remain. You can perceive how to add things, submenus and for the most part control what the Toolbar shows, Nonetheless shouldn’t something be said about controlling when it is shown or what it looks like? The response to these inquiry depends on working with the ActionBar object and as you would expect this isn’t an overall thing to all menus. ActionBar has heaps of highlights for showing menus in various ways and to cover it totally would take an excessive amount of room.However,it merits knowing a portion of the fundamental customizations that you can apply. You can get the ActionBar object in an Activity utilizing supportActionBar or actionBar properties. When you have the

ActionBar object you can utilize its show() and stow away() techniques to show/conceal it as required: supportActionBar?.hide()
Similarly you can change the title and caption showed in the activity bar: supportActionBar?.title =“My Action Bar”
supportActionBar?.subtitle =“My Subtitle”



There are numerous different properties and strategies that you can use to alter the manner in which the activity bar looks and behaves.

Things that merit gazing upward are: utilizing a split Toolbar; Up Navigation; Action Views; Collapsible Action Views; Action suppliers; and Navigation tabs.

Summary

☐ A menu is an order of View protests very

much like any UI element. ☐ You can make a menu in code, however utilizing a menu XML asset document is the most widely recognized method of doing the job. ☐ There are Menu articles and labels which go about as compartments for Item protests and tags.
☐ Submenus can be made by settling Menu objects inside other Menu objects.
☐ Menu things can have a title, symbol, and numerous different qualities which oversee how they are displayed.
☐ There are four distinct ways you can show a menu: App bar/Toolbar; setting menu; logical activity mode; popup.

☐ To utilize the App bar/Toolbar and the relevant activity mode in prior renditions of Android you really want to utilize the AppCompatActivity and the other help classes.

☐ To show a Toolbar you should simply utilize the onOptionsCreate occasion controller to make a menu, as a rule by swelling a menu resource.

☐ Overall you can deal with click occasions on every menu thing or, all the more ordinarily, you can utilize a thing's snap occasion overseer that the framework gives that reacts to a tick on any of the items.

☐ For the Toolbar the thing's snap occasion controller is characterized in the Activity as
abrogate fun onOptionsItemSelected(item: MenuItem): Boolean

Chapter 10 Menus – Context and Popup

As well as the generally useful Toolbar which fills in as an application 's principle menu, there are three other regularly experienced menus– the setting menu, the context oriented activity menu, and the popup menu. They share the essential Android way to deal with menus while having some unmistakable characteristics.
The setting menu is a drifting menu that

presentations orders that take the thing clicked as their subject. For instance, choosing a line in a table may raise a setting menu that permits you to erase or move the thing. The setting menu is straightforward and genuinely simple to carry out, yet there is a slight turmoil brought about by the presentation of the Contextual Action mode, a setting menu as an activity bar which is upheld in Android 3 or more. A popup menu shows a rundown of things and is moored to the View that summoned it. It tends to be utilized, for instance, to give extra decisions to tweak an action.

The Context Menu

Now that we have the standards of menu development surely knew it is not difficult to broaden what you are familiar the activity bar to different sorts of menu. To make a setting menu is exceptionally simple once you know the essential mechanism.
You can enlist any View object in the UI to

react to a long snap. At the point when any enlisted View object gets a long snap, the Activity’s onCreateContextMenu occasion overseer is called. This is the place where you make the menu that will be displayed as the setting menu. Notice that you can enlist as many View objects as you like and every one will react to a long snap by onCreateContextMenu being called. This implies that to show an alternate menu for each View object you want to test to see which one has caused the event.
So the formula is:

1. Create a XML menu resource
2. Register all of the View protests that you need to trigger the setting menu utilizing registerForContextMenu

3. Override the onCreateContextMenu occasion overseer and utilize a menu inflater to make the menu

For a basic model make another menu asset, right snap in the res/menu registry and select the new asset menu record choice. Call it mycontext.xml and enter the following:

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

<item
<item
```

android:title="MyItem1" android:id="@+id/myitem1"/>

```
android:title= "MyItem2"
android:id="@+id/myitem2"/>
</menu>
```

Next place a button and a checkbox on the UI and in the OnCreate strategy add the lines as displayed below:

```
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

registerForContextMenu(button)
registerForContextMenu(checkBox)


}
```

You can see what's going on here, We observe the Button and CheckBox items and register them utilizing registerForContextMenu.Later this a long snap on either UI part will trigger the onCreateContextMenu event.

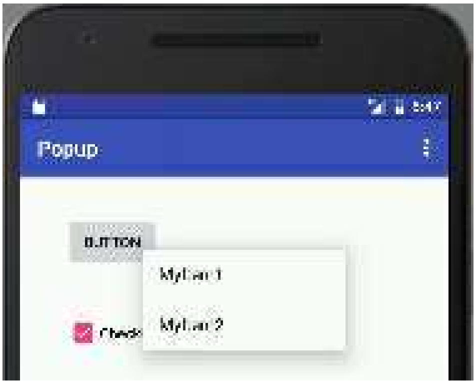
We want to add an overseer for this occasion and we should allow Android Studio to produce the code for us. Assuming you right-click at a reasonable spot in the Java code for the Activity and select the Generate and afterward the Override Methods choices in the menu that seems you will see a rundown of techniques you can override:

Select onCreateContextMenu and Android Studio will produce a stub for you. You can observe the technique you are searching for by composing in its name character by character and permitting Android Studio to show you matches in the rundown. Select the one you need when you can see it.
All you need to do in this hit is expand the menu asset record or in any case make a menu object in code:

```
supersede fun onCreateOptionsMenu(menu: Menu): Boolean {
```

```
menuInflater.inflate(R.menu.menu_main, menu) return true

}
```

If you currently run the program you will see that a similar setting menu seems when you long snap on either the Button or the CheckBox:



Obviously assuming you truly need this to be a setting menu then you really want to test the View object passed into the onCreateContextMenu occasion controller and burden the menu asset that relates to the fitting menu for the View object.
How would you deal with the setting menu thing determination? n pretty much the same way concerning the Toolbar.

When any of the things in the setting menu is chosen the Activity 's onContextItemSelected occasion overseer is called. So you should simply supersede this and utilize the thing's id to control what ought to occur. For example:

```
supersede fun onContextItemSelected(item: MenuItem): Boolean { return
when

(item.itemId) {
R.id.myitem1 → {
myitem1 action
true

}

R.id.myitem2 → {
myitem2 action
true

}
else - >
super.onContextItemSelected(item)
}
}
```

The main additional detail is the accessibility of the ContextMenuInfo object which is passed to the onCreateContextMenu overseer and can be recovered from the thing in the occasion controller utilizing item.getMenuInfo(). This contains extra information such as which line in a list view has been long clicked. Precisely what data is

incorporated relies upon the View object that produced the event.
If you need to see a setting menu in real life add a TextView to the UI and try:

```
supersede fun
onContextItemSelected(item: MenuItem): Boolean { return when

(item.itemId) {

R.id.myitem1 - > {
textView.text =
“item1” true

}

R.id.myitem2 - > {
textView.text =
“item2” true

}
else - >
super.onContextItemSelected(item) }
}
```

Contextual Action Bar

The setting menu is not difficult to utilize Be that as it may later Android 3 the Contextual Action Bar menu is liked as it expands the conduct of the move bar,assuming control over the App bar position at the highest point of the screen. However, it works, and is executed, freely of the App bar.

When a client long snaps a View object a relevant activity bar (CAB) shows up at the highest point of the screen, rather than close by the related View object as the setting menu does, and it offers the client to chance to play out various activities. This might be the method for getting things done, yet it is more convoluted in light of the fact that it expects you to carry out more code. Contextual activity mode can be utilized in before forms of Android through the Support Library which Android Studio consequently incorporates for you.
Unlike the setting menu you don’t simply enlist UI parts that trigger the menu. You need to call the startActionMode strategy to show the relevant activity mode menu, and this implies you need to compose a long snap occasion controller. Notice that it is dependent upon you what client activity triggers the relevant activity mode, however it is almost consistently a long click.

The means to making a context oriented activity bar are:

1. First make an occurrence of ActionMode.Callback which contains strategies for various stages in the menu’s lifecycle.
2. You need to at minimum abrogate the onCreateActionMode and this generally calls a menu inflater to produce the menu.
3. To deal with occasions on the menu you likewise need to supersede onActionItemClicked.
4. To cause the menu to seem you need to call
startSupportActionMode and pass it the case of
ActionMode.Callback you made earlier.

Notice that ActionMode.Callback is the primary illustration of an occasion object that we can’t carry out utilizing a lambda for the basic explanation it characterizes four occasion overseers. As such it’s anything but a SAM and we need to carry out an item to pass to the setListener method.
For instance, assuming that you have a button

and you need a long snap occasion to trigger the menu you really want to write in the Activity's OnCreate something like:
button.setOnLongClickListener { view-> bogus } So far all we have executed is a void long snap occasion controller for the button. To make the long snap occasion show a menu we first need an occurrence of the ActionMode.Callback interface.
You just need to enter object:ActionMode.callback and afterward you can utilize the create code choice to execute the interface:

```
val mycallback=object : ActionMode.Callback { abrogate fun
onActionItemClicked(
mode: ActionMode?, thing: MenuItem?): Boolean { TODO("not
implemented")
}


supersede fun onCreateActionMode(
mode: ActionMode?, menu: Menu?): Boolean { TODO("not implemented")
}

supersede fun onPrepareActionMode(
mode: ActionMode?, menu: Menu?): Boolean { TODO("not implemented")
}
abrogate fun onDestroyActionMode( mode:
ActionMode?) {
TODO("not implemented")
}


}
```

The Callback object has four techniques

every one of which is called as a component of the menu’s life cycle. You really must supplant the TODOs with code that profits the right sort for each method.
You need to ensure that when you import the class you import the rendition from the help library:

```
import android.support.v7.view.ActionMode
```

As you would well theory, to cause the menu to seem you need to fill in the subtleties for the onCreateActionMode method:

```
abrogate fun onCreateActionMode(
mode: ActionMode?, menu: Menu?): Boolean {
mode?.menuInflater?.inflate(R.menu.mycontext, menu) return true
}
```

All we do is swell the menu asset and the framework adds it to the menu. You can likewise finish up the subtleties of different strategies– you in all probability will need to add something to onActionItemClicked, however this doesn’t include anything new. Finally we want to enact the logical activity menu in the button’s onLongClick occasion handler:

```
button.setOnLongClickListener { view
>
startSupportActionMode(mycallbac
k) true

}
```

Now when you long click the button a new context action bar menu appears above the usual App bar:

Notice that this acts somewhat better in that the menu stays on the screen until the client taps the left bolt button in the corner. You can permit the client to make numerous determinations or excuse the menu when the client chooses one choice. In this sense the logical activity bar is more complex and adaptable than the basic setting menu.
The total code to make the relevant activity bar show up, with the article changed to a mysterious item withing the setOnLongClickListener is: abrogate fun onCreate(savedInstanceState: Bundle?) {

```
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)
```

```
button.setOnLongClickListener {
view - > startSupportActionMode(object : ActionMode.Callback {
supersede fun onActionItemClicked(

mode: ActionMode?,
thing:
MenuItem?):
Boolean {
return false
}
supersede fun onCreateActionMode( mode:
ActionMode?,
menu: Menu?): Boolean { mode?.menuInflater?.inflate(R.menu.mycontext,
menu) return true
}

supersede fun onPrepareActionMode( mode: ActionMode?,
menu: Menu?): Boolean { return false

}

abrogate fun onDestroyActionMode(mode: ActionMode?) { }
})
true
}
```

This presentation has quite recently start to expose how context oriented menus can be utilized however the overall standards observe the thoughts of the overall Android menu and large numbers of the points of interest of the activity bar.

The Popup Menu

The last menu is so basic by correlation with the rest it is not really important to disclose how to carry out it! What is more troublesome is to say when it ought to be utilized. It positively shouldn’t be utilized as a substitute for a setting menu where the tasks influence the substance that it is“connected” to. On account of a popup it appears to be that its normal use is to refine an activity choice by giving boundaries that adjust it.

To show a popup menu, you should simply launch a PopupMenu object, set its onMenuItemClick occasion overseer, make the menu by blowing up an asset document lastly utilize its show strategy. Where on the screen the PopupMenu shows relies upon the View object you pass while making the instance.

The means are:

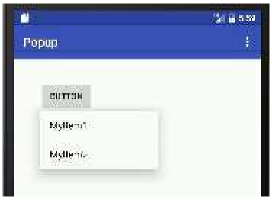
1. Create an occurrence of PopupMenu and pass the View object you need to use to position the menu.
2. If you need to deal with thing

determination, supersede the onMenuItemClick method.
3. Use the case’s blow up technique to make the menu.
4. Use the case’s show strategy to show the menu.

A popup menu is generally displayed in light of some client activity thus we want an occasion controller to make the popup in. Assuming that you place a button into the UI you can characterize its snap occasion controller as:

```
button.setOnClickListener { view - >
val popup = PopupMenu(view.context, view)
popup.inflate(R.menu.mycontext) popup.show()

}
```

ThePopupMenuconstructoraccepts the context andtheViewobjectit will be displayed next to. Usually this is the View object that the user clicked or interacted with i.e. the button in this case. Next we inflate the menu as usual by using the Popups own inflater which adds the menu to the Popup. Finally we call Show which displays thePopup:



The menu is quickly excused assuming the client taps on a thing or on some other piece of the display.
Obviously on the off chance that you were truly utilizing the popup menu you would likewise deal with the thing click event:

```
popup.setOnMenuItemClickListener { thing >
textView.text =
item.title false

}
```

As before you would utilize a when to find which thing the client had chosen and follow up on that selection.
The total code is:

```
abrogate fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

button.setOnClickListener { view - >
val popup = PopupMenu(view.context, view)
popup.inflate(R.menu.mycontext)
popup.setOnMenuItemClickListener { thing >
textView.text =
item.title false
}
popup.show()
}
}
```

Summary

☐ The setting menu is summoned for all View objects in the UI that are enlisted utilizing registerForContextMenu(View).

☐ When the client long taps on any enrolled object
onCreateContextMenu is called and you can utilize this to show the menu. At the point when the client chooses any of the things in the menu, the framework
onContextItemSelected occasion controller is called.

☐ The logical activity bar menu is the most muddled to execute. You need to make an ActionMode.Callback object total with techniques to make the menu and handle occasions from its items.

☐ To show the menu you call the startSupportActionMode(Callback) indicating the Callback object to be used.

☐ If you need to utilize the relevant activity mode menu on prior variants of Android you need to utilize the Support Library classes.

☐ The popup menu is the most straightforward to execute. Basically make a Popup object, swell the menu asset, and utilize its show technique to show the menu.

Chapter11 Resources

So far we have disregarded assets, yet the subject can be overlooked no more. Assets fill too significant a need in Android. They not just make limitation more straightforward, they are critical to making applications that adjust to the gadget they are being run on.

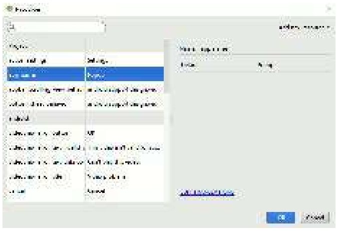
Why Use Resources?

Any limited scale information that your application utilizes – strings, constants, designs, format documents– should be generally included as assets rather than hard coded into your application. For instance, up to this point we have entered any text that the UI required straightforwardly as a string property estimation. For a button this approach has meant setting the Textproperty bytypinginastringsuchas "HelloWorld". Android Studio does it best to save you this effort. If you run the Warnings and Errors "linter" by clicking on the red icon in the Layout Editor, then for any string you have entered as a literal into the Attributes window you will see awarning:

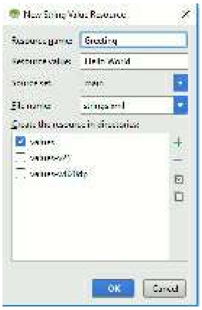
While you shouldn ’t freeze about each notice

that the linter shows you, this one is a smart thought on the grounds that an asset can be reused in different pieces of the program and it is not difficult to change without jumping into the code.
A lot greater benefit is that you can give elective assets, which are naturally chosen by your application as per the conditions. For instance, for this situation you can give strings in another dialect and they will be utilized naturally as indicated by the district where the application runs. This is an extremely incredible method for making custom applications that work anyplace on the planet and on any device.

So how would we make an asset? It relies upon the sort of the asset however in Android Studio there are two different ways of making a string resource.
The first is to click the "..." at the right of the property entry box which makes theResourcesdialogpopup:



This rundowns existing assets characterized by the framework, the venture and the topic being used. You can likewise utilize the Add New Resource dropdown to make extra assets.For this situation a string resource:



New String Value Resource

Resource name:	Greeting
Resource value:	Hello World
Source set:	main
File name:	strings.xml

Create the resource in directories:

- [✓] values
- [ ] values-v21
- [ ] values-w820dp

OK Cancel

All you need to do is enter an asset name and its worth – the remainder of the sections can be left at their defaults, their significance will turn out to be clear as we proceed.
When you click OK you will see the Attributes window with the entry: @string/Greeting
This is a reference to your recently made

asset. You will likewise see the string that you have determined show up as the button's inscription. You would now be able to reuse the string asset by entering @string/Greeting anyplace the"Hi World" string is required–obviously it must be utilized with a property that acknowledges a string. Assuming you change the meaning of an asset then clearly each of its uses are updated.
The second method of making a string asset is to alter the asset XML document. Assuming you check out the record strings.xml in the res/values catalog you will find it contains:

```
<resources>

<string name= "app_name">Resources1</string>
<string name="hello_world">Hello world</string>
<string name="action_settings">Settings</string>
<string name="Greeting">Hello World</string>

</resources>
```

You can see the welcome world string that we have quite recently added and some different strings that Android Studio makes naturally for you, including one more welcome world string which is utilized in the at first produced UI. You can alter the asset record straightforwardly or you can utilize the Resources exchange to make and alter existing string resources.

What are Resources?

☐ Resources are any information that your program needs to use. Resources are arranged into your program’s executable fit to be appropriated

to your end clients as a feature of your application. For a string this doesn ’t appear to be especially valuable, later everything you could straightforwardly utilize a string steady in code, however for different sorts of asset it tends to be the best way to get the information remembered for the application’s executable.

For instance, drawables, a scope of various kinds of illustrations objects, can be incorporated as assets. Assuming you incorporate a .gif document as an asset then that .gif record is fused into your application by the compiler. This implies you don’t need to give the gif as a different document and you don’t need to stress over where it is put away. Notice, in any case, that adding a .gif record to your application builds its size.

All assets are put away in organizers inside the res registry. The organizer name gives the kind of the asset and, as we will see, can indicate when the asset is to be used.
You can see a rundown of asset types in the documentation yet the usually utilized ones are:

☐ drawable/
Any realistic– normally .png, .gif or .jpg yet there are a scope of other less often utilized sorts. The standard is– assuming it’s in any sense a realistic it has a place in drawable.

☐ design/
We have been utilizing formats since we began programming Android and XML format is an asset assembled into your application prepared to be used.

☐ menu/
The XML records that determine a menu are additionally assets and aggregated into your app.

☐ values/
XML files that define simple values such as strings, integers and so on. You can imagine these as characterizing constants for your

program to utilize. In spite of the fact that you can put any combination of significant worth characterizing XML into a solitary record in the qualities/index, it is common to bunch esteems by information type and additionally use.

Typically:
• arrays.xml for composed arrays
• color.xml for shading values
• dimens.xml for dimensions
• strings.xml for strings
• styles.xml for styles.

You can likewise incorporate subjective documents and discretionary XML documents in crude/and xml/. There are two registries managing liveliness, illustrator/and anim/.There is likewise an index where you can store symbols appropriate for various screen goals,however a greater amount of this when we take a gander at contingent resources.

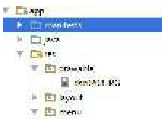
Drawables

Drawables are somewhat unique in relation to different assets in that there is no XML record that characterizes them. To make and utilize a drawable you should simply duplicate the realistic record that you need to use into the drawable/index. You needn't bother with XML to characterize the asset in light of the fact that the asset's id is only the document name.
The most ideal way to clarify how drawable assets work is through a straightforward model. To make a jpg bitmap asset you should simply duplicate the jpg record into the right asset index. Android Studio

naturally makes the drawable/catalog for yourself and you should simply duplicate the bitmap document into it– however how?
The primary comment is that the record name for a drawable asset can just hold back lower case letters and digits. Assuming the first document name doesn't adjust to this example you can rename it later you have replicated it into the directory.
There are two genuinely simple ways:

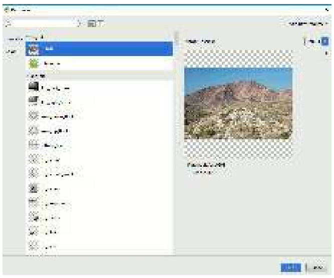
1. You can observe the record in the typical record registry construction and use reorder to glue it into the index showed in Android Studio.

2. Alternatively you can right tap on the drawable organizer in Android Studio and select Show in Explorer which opens the envelope as a standard document framework organizer to which you can duplicate the bitmap record in any capacity that you like, intuitive for instance:



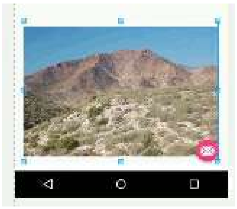
Once you have the record, dsc0208.jpg for this situation, in the drawable/catalog you can utilize it. Place an ImageView control on the UI utilizing the Layout Editor.The Resources window opens naturally for you select the drawable you need to use.
If you need to change the drawable sometime in the not too distant future, find its src property in the Attributes window and snap on the ... button at the extreme right and the Resources window will open.

Select the Project tab and look down until you can see the name of the record you have recently replicated into the drawable organizer and select it:



You will see that @drawable/dsc0208.jpg has been entered as the worth of the src property–

you might have entered this string straightforwardly without the assistance of the Resources window. You will likewise see the picture showed in the ImageView control:



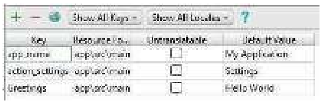
That ’s practically everything to drawable/assets, however there is something else to find out with regards to the various kinds of drawables that you can store there. This is best talked about when we cover designs overall in the following chapter.

Values

The simplest of the asset envelopes to utilize is likely qualities/,yet it additionally will in general be the least frequently utilized. The strings.xml record will in general be utilized, yet the others are underutilized.The

justification behind is that the framework will in general incite you to enter strings as assets and there is an undeniable benefit to string assets in that they permit simple confinement. Nonetheless,placing constants of different kinds in assets is a decent idea.
Although the documentation just notices string, clusters, shadings, aspects and styles, you can incorporate a more extensive scope of information types in the XML file:

☐ Bool
<bool name=“resourcename”>true</bool>
☐ Integer
<integer name=“resourcename”>1234</integer>
☐ String
<string name=“resourcename”>A String</string>



Key	Resource Fo...	Untranslatable	Default Value
app_name	app\src\main	☐	My Application
action_settings	app\src\main	☐	Settings
Greetings	app\src\main	☐	Hello World

There is likewise a string manager which you can use to deal with an assortment of strings including interpretations to different dialects. Its utilization is genuinely self evident:

There are additionally two exhibit types:

☐ Integer Array

```
<integer-cluster name= “resourcename”>
<item>123</item>
<item>456</item>

</number array>
```

☐ Typed Array

```
<array name= “resourcename”>
<item>resource</item>
<item>resource</item>


</array>
```

Dimension is likewise two straightforward worth resources:

☐ Dimension is basically a number worth complete with a units

designator. For example:

```
<dimen name=“resourcename”>10px</dimen>
```

Obviously you can utilize any of the standard Android estimation scales– pt, mm, in, etc. You use aspects anyplace that you want to determine a size or area in a specific arrangement of units.

Finally Color allows you to put together the shadings you are utilizing in your app:

☐ Color gives simple to utilize names to hex

shading
codes. For example;

```
<color name=“resourcename”> #f00 </color>
```

characterizes a red tone. You can indicate a shading utilizing any of:

```
#RGB, #ARGB, #RRGGBB or #AARRGGBB
```

where every one of the letters addresses a solitary hex person with R= Red, G=Green, B=Blue and A=Alpha (transparency).

IDs

Ids are esteem assets and can be set up very much like some other worth asset. For instance, to set up an id for a button you may use:

```
<item type=“id” name=“resourcename”/>
```

Notice that this is somewhat not the same as other worth assets in that you don’t need to offer a benefit. The explanation is that the framework

gives an exceptional whole number worth to every id. You don ’t regularly have to characterize an id in a XML record all of its

own on the grounds that ids can be made on the fly inside other XML files.
Putting a + before an asset id makes the asset without having to unequivocally do the work.
For example:

```
<Button
android:text="“@string/Greeting”
android:id=“+@id/button2”
```

makes the button2 asset. On
the other hand:

```
<Button
android:text=“@string/Greeti
ng” android:id=“@id/button2”
```

will possibly work on the off chance that you have characterized button2 in an asset document in the qualities/folder.
Obviously in Android Studio you can essentially type in an id in the
property window say and the framework will naturally give the +@id to
make it auto-make. Additionally remember that the Kotlin module changes
over ids into properties that reference the View objects.So in the model
above there will be a button2 property that

references the Button object.

Accessing Resources in Code– The R Object

For a significant part of the time you truly don ’t have to waste time with getting to assets in code on the grounds that the task is finished for you consequently. For instance, in the event that you relegate a string asset to a button’s text:

```
<Button
android:text=“@string/Greetin
g”
```

then the framework recovers the string asset and sets the button ’s text property to it when the format is swelled. You don’t need to effectively make everything work. In any case, in some cases assets are required inside the code of an application and you need to expressly recover them.

When your application is assembled by Android Studio it consequently makes an asset id,a one of a kind whole number,for each asset in your res/catalog.These are put away in a produced class called R - for Resources. The registry structure beginning with res/is utilized to produce properties for the R object that permits you to observe the id that relates to any asset. This implies that an asset id is constantly named something like R.*type*.*name*. For example:

R.string.Greeting
recovers the asset id for the string asset with asset name“Hello” that is: <string name=“Greeting”>Hello World</string>
Notice that the asset id is a number and not the string it
distinguishes. So how would you convert an asset id to the asset value?

The main comment is that you don ’t generally need to. There are numerous techniques that acknowledge an asset id as a boundary and will get to the asset for your benefit. It is normally essential to recognize when a technique is content with an asset id and when you need to pass it the real asset value.

If you do have to pass the resource, or you want to work with the resource, then you have to make use of the Resources object. This has a scope of gettype(resource_id) strategies that you can use to get to any asset. For instance, to get the string with the asset name“Hello” you would write:

```
var myString=resources.getString(R.string.Greeting)
```

and myString would contain “Hi World”.Assuming you are not with regards to the movement you may need to utilize applicationContext.resources.

The main issue with utilizing the Resources object is attempting to work out which get techniques you really need.
There are likewise utility strategies that will return any part or all of its asset name given the id:

☐ getResourceEntryName(int resid)

Returns the passage name for a given asset identifier

☐ getResourceName(int resid)
Returns the complete name for a given asset identifier
☐ getResourcePackageName(int resid)
Returns the bundle name for a given asset identifier
☐ getResourceTypeName(int resid)
Returns the sort name for a given asset identifier.
There are additionally a few techniques that will interaction the asset just as basically recover it. For instance, assuming you have a crude XML asset,

getXml(int id) returns a XmlResourceParser that allows you to manage the XML recovering labels, ascribes, etc.

Conditional Resources

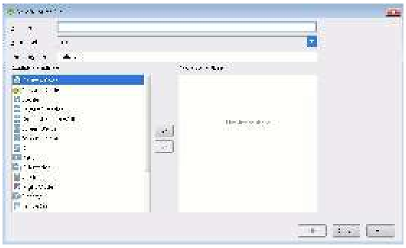
So far assets have recently been a method for getting information into your application. You may have imagined that assuming there

was one more method for doing the work then it very well may be comparable. Notwithstanding,Android Resources are vital to altering your application so it works with the wide scope of gadget types and client that an Android application needs to adapt to. The cunning part is that you can utilize contingent assets to give a bunch of assets modified for the current gadget at runtime.
The thought is basic. First you give a default asset record. This is the one that is situated in the proper index,/res/values say, however presently you use qualifiers as a component of the catalog name.
For instance, you can make a registry called/res/values-es which is planned to give asset esteems to the application when running on a gadget set to a Spanish language region. What happens is that first any qualities that you have characterized are taken from the XML records in the qualities registry, these are viewed as the default. Then, if the application is running on a Spanish language gadget the XML records in qualities es are handled and any assets with a similar name supplant the ones gave by the defaults.

You can see that this gives a genuinely simple method for ensuring that your application presents a UI in the nearby language, however there are a greater number of qualifiers than just area, and these permit you to alter the application for different elements of the gadget. You can even pile up qualifiers as a feature of envelope names to get more exact focusing on. For instance, the index esteems es-little would possibly be utilized assuming that the language was Spanish and the screen was like a low thickness QVGA screen. The main thing to be cautious about is that the qualifiers are utilized in the request where they are recorded in the documentation. There are qualifiers for district, screen thickness, size and direction, gadget type, night v day, contact screen type, console accessibility and stage adaptation (API level). You can think that they are totally recorded in the documentation, yet much of the time this is pointless in light of the fact that Android Studio assists you with making qualified resources.

If you select an asset catalog, values say, and

right snap you will see a New,Values asset document in the setting menu. On the off chance that you select this choice then the New Resource File discourse opens.You can utilize this to make an asset file
– enter its name - and you can apply any qualifiers you need to by essentially choosing from the rundown of accessible quantifiers to the left:



If you select Locale then you will be permitted to enter any subtleties of the qualifier.On account of Locale you can choose the dialects and districts you need to apply from a list:

Using the New Resource File exchange saves you a ton of time looking into qualifiers and language/area codes. It is a lot less difficult than hand coding the indexes and XML documents. Be that as it may, you should know about how Android Studio presents qualified assets to you in the undertaking program. For instance, assuming you make another strings document in the qualities registry and select Locale and es for the Spanish language in Any Region then this makes an index res/esteem es and a strings.xml record inside it. So presently you have:

values/strings.xml

and:

values-es/strings.xml

This isn’t what the undertaking program shows naturally which is the Android view. It shows the new document as being in the qualities index inside another envelope called strings.xml.

The project browser is trying toshow you the resource files with allofthe similar files, i.e. strings, all grouped together in a single directory, but with qualifiers displayed to the right. In the case of locale it even shows the country flag as anicon:

This is a reasonable method of getting sorted out things and it apparently makes working with different asset documents more straightforward.For this situation you can see that the two strings.xml documents are shown as minor departure from strings.xml.
You can see the genuine record structure by choosing Project Files starting from the drop menu in the upper left-hand corner.

Select the Android view to get back to the recognizable perspective on your project. Working with Android Studio you can, generally, disregard the real record and index structure and basically add extra asset records utilizing the New Resource File exchange, or for adding regions the Translation Editor depicted in the following segment. Every so often, notwithstanding, you should find a

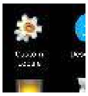
document and physically alter the index structure.Specifically,there are no orders that permit you to alter the qualifiers related with an asset document. The most straightforward arrangement is to erase the document and once again make it with new qualifiers if vital.It is additionally worth focusing on that you can make custom format documents similarly and that the Layout Editor likewise has offices to permit you to clone a current picture design as a scene design, for example.

A Simple Localization

As we have as of now made a Spanish strings.xml record it is trifling to give a Spanish variant of the good tidings string.You should simply alter the Spanish strings.xml to read:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

<string name="Greeting">Hola Mundo</string>
</resources>
```

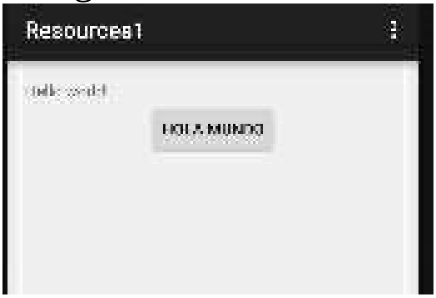
Now all you have to do is run the app and change the locale of the emulator



or the real Android device. The simplest way to change the locale is to use theCustomLocaleapp:
When you run this you can see the current

area and select another one:

When you run the application in the emulator you will see the string asset recovered from the Spanish strings.xml file:



Notice that while the button ’s text has changed the default strings.xml has been utilized for different strings in the app.
It is vital to ensure that every one of the assets you use have a default esteem. Assuming you miss an asset out of a district

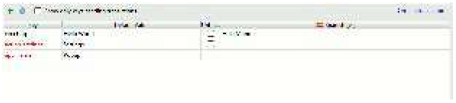
document that doesn’t have a default, your application will not run.

Android Studio Translation Tools

Android Studio likewise has some extra instruments to permit you to see restrictive assets. The World symbol showed in the Layout Editor permits you to choose an area. It merits utilizing this to check that your application great examines other locales:

However, new areas possibly show in the drop-down list assuming they have been added utilizing the Translation Editor. Assuming you have physically added the Spanish area utilizing the directions prior add it in the Translation Editor any interpretations that you have as of now added will be retained.

If you are attempting to keep an application that upholds various dialects then the simplest method for keeping up with it is to utilize the Edit Translations choice. This presentation an editorial manager for string assets that shows every region rendition on the equivalent line:



You can utilize this to enter interpretations of default strings and you can make new string assets across each of the areas with a solitary activity– enter the key, default and interpretations. You can likewise utilize the world symbol at the upper left to make new districts and this is by a wide margin the most effective way to do the work. There is even a connection to arrange an interpretation from Google Translate.

Summary

☐ Use assets for as much information as possible. It makes it more straightforward to change things.
☐ You can utilize the Property window to make and utilize assets or you can alter the XML directly.

☐ There is a wide scope of asset types. Notwithstanding designs and menus, you are sure to utilize values and drawables.

☐ Drawables are an illustration of an asset that doesn’t utilize a XML record. You just duplicate the illustrations asset documents into the drawables directory.

☐ You can store straightforward information types in the qualities asset catalog and this is frequently a preferred method for getting things done over utilizing constants in the code.

☐ The R object is consequently created and its construction emulates the asset index structure.For every asset the R object has a number asset id.

☐ Many techniques take an asset’s id and recover the asset for you. To unequivocallygettoanassetincodethen,atthatpo provides.

☐ Conditional assets permit you to give assets that suit the current gadget– area, screen goal thus on.
☐ Conditional assets work by applying qualifiers to the names of the registries that hold the asset files.

☐ Android Studio gives an improved visible of restrictive assets that bunches all minor departure from an asset document in a similar hierarchical envelope.This doesn’t compare to the document structure, however it is simpler.

☐ You can utilize the New, Resource File order to make contingent resources.
☐ The Layout Editor allows you to choose

which district asset is utilized so you can work straightforwardly with the confined layout. ☐ If you need to restrict your application then, at that point, utilize the Android Studio Translation Editor.

Chapter 12 Bitmap Graphics

Android illustrations is a gigantic subject, yet you need to begin some place. In this section we look a straightforward 2D bitmap designs, which is regularly all you really want, and find the essential standards of Android graphics.

Android Graphics

Graphics support in Android is broad – from the basic showcase of pictures and 2D

illustrations through liveliness and on to full 3D delivering. To cover everything needs a different book, and surprisingly then there would be themes left immaculate. This part is a prologue to the base you really want to be aware of 2D bitmap designs in Android. It isn’t all you really want to know to take care of business, however it is sufficient to kick you off on numerous normal straightforward illustrations undertakings. With this establishing you’ll likewise be in a situation to search out a portion of the extra data you want to do the more uncommon things.
The critical distinction between this record of Android illustrations and the normal methodology you find in different reports is that it focuses on the standards just as the how-to. Before the finish of this section you ought to have a reasonable thought of what the distinctive Android illustrations classes are everything about.

The Bitmap

You could say that the bitmap is the establishment of any utilization of Android designs. The explanation is that regardless of how a realistic is determined or made, it is a bitmap that is in the end made and shown on the screen.

There are numerous methods of making or acquiring a bitmap.We have effectively perceived how a bitmap document,.jpg,gif or.png can be remembered for the drawable/asset registry and showed in an ImageView control.Inmanycasesabitmapcan beacquiredbyloadingafile, readinginacollectionofbytes,oreventakinga photowiththeAndroid camera.
In this case we will essentially make a Bitmap directly:
val b = Bitmap.createBitmap(500, 500, Bitmap.Config.ARGB_8888) A bitmap is basically a square shape of pixels. Every pixel can be set to a given tone however precisely what tone relies upon the sort of the pixel. The main two

boundaries give the width and the tallness in pixels. The third boundary indicates the kind of pixel you need to utilize. This is the place where things can get convoluted. The detail ARGB_8888 implies make a pixel with four channels ARGB - Alpha, Red, Green,Blue, and assign eight pieces of capacity to each channel. As four eights are 32 this is 32-bit designs. The alpha channel allows you to set an opacity.

There are various pixel designs you can choose, and there is consistently a compromise between the goal and the measure of room a bitmap possesses. Notwithstanding, ARGB_8888 is an extremely normal choice.

Now you have a bitmap how would you be able to manage it? The appropriate response is truly a lot!
Most of the strategies for the Bitmap object are worried about things that change the whole picture. For example:

```
b.eraseColor(Color.RED)
```

sets every one of the pixels to the predetermined shading, red for this situation.

Using
the setPixel and getPixel techniques you can get to any pixel you need to

and perform practically any illustrations activity you want to.You can likewise work with the pixel information at the piece level.

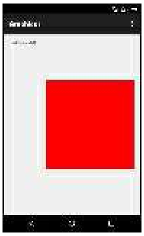
The ImageView Control

How would you be able to see a Bitmap that you have recently made? The basic answer is to utilize an ImageView control.Thisisn’ta popular approach,however,because it isn’t as flexible as alternatives such as overriding the onDraw event handler. All things considered, the ImageView control is extremely simple to utilize and adequate for some tasks.
Start another venture and spot an ImageView into the UI, tolerating the defaults. The Layout Editor wont let you place a void ImageView on the plan surface so select any

drawable as an impermanent filler. As an exhibition of how you can utilize the ImageView to show a Bitmap, change the onCreate occasion controller to read:

```
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)

setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

val b = Bitmap.createBitmap(500, 500, Bitmap.Config.ARGB_8888)
b.eraseColor(Color.RED)
imageView.setImageBitmap(b)

}
```

If you run the program you will see a genuinely enormous red square seem where the ImageView control has been placed:



Notice the ImageView control has been consequently resized to show the Bitmap. The ImageView control has a scope of techniques similar
to setImageBitmap that makes it helpful for

showing a scope of various sorts of graphic.

Canvas

You can simply utilize a Bitmap and work with individual pixels, however this is definitely not an extremely“undeniable level” method for making designs.What we really want are some bigger scope strategies that draw helpful things like lines, square shapes, circles thus on.
The Bitmap object doesn’t have this kind of strategy, however the Canvas object does and you can utilize it to draw on any Bitmap. That is you make a Canvas object, partner it with a Bitmap, and you can utilize the Canvas item’s techniques to draw on the Bitmap. There are bunches of various Canvas drawing strategies, and surprisingly an office to change the essential direction framework to anything you want to utilize, yet we need to begin some place so a basic model first.

You can make a Canvas and join it to a

bitmap in one operation:

```
val c = Canvas(b)
```

Now when you utilize the drawing techniques for the Canvas they draw on the Bitmap b. There are many events when a control or item gives a Canvas object joined to a Bitmap all prepared for you to draw on.

It is likewise essential to realize that, at first, the direction arrangement of the Canvas object is set to pixels as controlled by the Bitmap. That is, assuming the Bitmap is width by stature pixels, the default coordinate framework runs from 0,0 at the upper left-hand corner to width,*height at the base right*.

Using Canvas can be just about as basic as calling a technique like drawLine to define a boundary between two focuses. The main slight confusion is that you need to utilize the Paint object to indicate how the line will be drawn. Overall Paint controls how any line or region is drawn by the Canvas methods.

A common Paint object is something like:

```
val paint = Paint()
paint.setAntiAlias(true)
paint.STROKEWidth=6f
paint.color=Color.BLUE
paint.Style=Paint.Style.STROKE
```

After creating the Paint object we set AntiAlias to true, i.e turn it on. This creates a higher quality but slightly slower rendering.Next we set the width of the line to 6 pixels, color to blue, and sets it as a stroke, i.e. a line rather than an area fill. If you are puzzled by the 6F in the setStrokeWidth it is worth saying that this is how you specify a float constant.

Once made you can utilize a Paint object as regularly as it is required and you can change it and reuse it. You can likewise utilize a current Paint object to make another Paint object which you then modify.

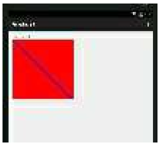
A First Graphic

Now we have a Paint object we can draw a line:

```
c.drawLine(0f, 0f, 500f, 500f, paint)
```

This draws a line from 0,0 to 500,500 and as the Bitmap is 500 by 500 pixels

this draws a diagonal line across the entire bitmap. Again the numeric values all end in f because the method needs float values, for reasons that will become clearlater:



All you need to do now is get to know the diverse drawing techniques gave.Here we utilize a couple of the most common.
First we really want to set the Paint object we made before to a FILL style so we draw strong squares of shading and not simply outlines:

```
paint.Style=Paint.Style.FILL
```

To draw a yellow circle all we want is:

```
paint.color=Color.YELLOW c.drawCircle(400f,
200f, 50f, paint)
```

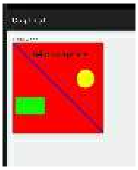
The initial two boundaries give the place of the middle and the third is the circle’s radius.
To draw a green rectangle:

```
paint.color=Color.GREEN
c.drawRect(20f, 300f, 180f, 400f, paint)
```

The initial two boundaries give the upper left-hand corner and the following two the base right-hand corner.
Finally we should add some text:

```
paint.color=Color.BLACK
paint.textSize=50f
c.drawText(“Hello Graphics”,0,14,90f,80f,paint)
```

The Paint object has various properties that let you set the text style and style. In this model we basically set the text size. The drawText strategy takes a string, the beginning and end places of characters in the string that will be shown,and the directions of the beginning location.
If you put this all together and run it you get this very tasteful graphic:



The total code to make this is:

```
abrogate fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)
val b = Bitmap.createBitmap(500, 500, Bitmap.Config.ARGB_8888)
b.eraseColor(Color.RED)
val c = Canvas(b)

val paint = Paint()
paint.setAntiAlias(true)
paint.strokeWidth=6f
paint.color=Color.BLUE
paint.Style=Paint.Style.STROKE
c.drawLine(0f, 0f, 500f, 500f, paint)
paint.style=Paint.Style.FILL
paint.color=Color.YELLOW c.drawCircle(400f,
200f, 50f, paint)

paint.color=Color.GREEN
c.drawRect(20f, 300f, 180f, 400f, paint)

paint.color=Color.BLACK
paint.textSize=50f
c.d rawText("Hello Graphics",0,14,90f,80f,paint)
imageView.setImageBitmap(b)

}
```

You can utilize this to evaluate other Canvas illustrations methods. It merits bringing up that there is a bunch of drawBitmap techniques that you can use to draw a current Bitmap onto the Bitmap related with the Canvas. This may appear to be something peculiar to do, yet it is perhaps the most helpful drawing operation since you can utilize it to carry out basic sprite animation.

Transformations

Now we come to one of the more modern choices that Canvas offers. Prior to anything is drawn, the directions you supply are followed up on by a change framework. Naturally this is set to the personality grid which leaves the directions as they were. Nonetheless, the accompanying strategies change the change matrix:

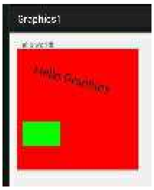
☐ rotate(float degrees)
turn about the beginning through degrees
☐ rotate(float degrees, float px, float py)
pivot about the point px,py through degrees
☐ scale(float sx, float sy)
scale about the beginning by sx and sy
☐ scale(float sx, float sy, float px, float py) scale about the point px,py by sx and sy
☐ skew(float sx, float sy) slant by sx and sy
☐ translate(float dx, float dy) interpret by dx and dy

There are likewise a few properties and techniques that let you work straightforwardly with the change matrix:

☐ matrix
set or get the change matrix
☐ concat(Matrix matrix)
increase the change grid by the lattice provided
If you definitely know how networks and change frameworks work then this will be appear to be very direct. If not, there are a great deal of traps holding on to entangle you. The fundamental one, that inconveniences pretty much everybody right away, is that the request where you do things matters. An interpretation followed by a turn isn't exactly the same thing as a pivot followed by an interpretation. Attempt it on the off chance that you don't trust me. Another is that these changes change the direction framework and influence nothing you have effectively drawn. They possibly change what happens when you draw something later the change has been applied.
For instance, we can pivot the text in the past example:

```
paint.color=Color.GREEN
c.drawRect(20f, 300f, 180f, 400f, paint) c.rotate(15f)
paint.color=Color.BLACK
paint.textSize=50f
c.drawText("Hello Graphics",0,14,90f,80f,paint)
```

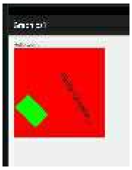
In this case there is a 15 degree rotation after the rectangle has been drawn
and before the text is drawn. The result is that the rectangle stays where it
was but the text is rotated:



After the turn all that you draw will be at 15 degrees.
Notice that the rotation is about the origin, i.e. 0,0 the top left-hand corner. If you want to rotate about a different point, usually the center of some object, you need to use rotate(d,x,y).
For instance to turn the square shape about its middle you would use:

```
c.rotate(45f,100f,350f)
paint.color=Color.GREEN
c.drawRect(20f, 300f, 180f, 400f, paint)
```

where 100,350 is the center of the rectangle:



You can likewise see that the text is presently in an alternate position and not just pivoted about the beginning. This is on the grounds that the 15 degree pivot is added to the 45 degree turn around the focal point of the square shape. To keep your changes isolated and non-communicating you need to make sure to return the change framework to the personality later you have transformed it to something else.

How would you hinder the lattice to the character? You could use:

```
c.matrix= Matrix()
```

as Matrix() makes a personality framework of course, or you could just use: `c.matrix= null`

in light of the fact that setting it to invalid makes the grid be reset. In any case, expecting to roll out an improvement to the change framework and afterward reestablish the first is such a typical activity that Canvas supports:

```
c.save()
```

which saves the current grid and:

```
c.r estore()
```

which reestablishes the current lattice. For example:

```
c.s ave()
c.rotate(45f,100f,350f)
paint.color=Color.GREEN
c.drawRect(20f, 300f, 180f, 400f, paint)

c.restore()
c.rotate(15f)
paint.color=Color.BLAC
K paint.textSize=50f
c.drawText(“Hello Graphics”,0,14,90f,80f,paint)
```

With the save before the first pivot, the reestablish before the second decouples the two transformations.

A Logical Approach to Transforms

One way to deal with monitoring changes is to draw everything focused on the beginning, and afterward decipher, scale and turn it to its last position. For instance, to draw the rectangle:

```
c.drawRect(20f, 300f, 180f, 400f, paint)
```

pivoted through 45 degrees you would initially draw a unit square fixated on the origin:

```
c.drawRect(- 0.5f, - 0.5f, 0.5f, 0.5f, paint)
```

then you would then scale it to its ideal size:

```
c.scale(160f,100f)
```

and turn it:

```
c.rotate(45f)
```

Finally you would move it to its right location:

```
c.t ranslate(100f, 350f)
```

If you evaluate these means, you will find that you don’t get what you anticipate.The explanation is that we have been changing the article: draw a square, scale the square, pivot it and move it to the ideal location. However, the Canvas changes don’t change graphical items yet the

direction framework. You can promptly see that this implies you should draw the square last later you have played out all of the transformations. Indeed this is the rule:

☐ do all that you would have done to the mathematical shape in the converse request while changing the direction system. So the right change grouping is:

```
c.save()
c.translate(100f,
350f) c.rotate(45f)
c.scale(160f,100f)
c.drawRect(- 0.5f, - 0.5f, 0.5f, 0.5f, paint) c.restore()
```

Notice that when you do a scale this applies to any strokeWidth you may set i.e. twofold the scale and a strokeWidth of 1 turns into a powerful strokeWidth of 2.
You can generally work out the change arrangement you want by
considering the graphical item, working out the changes expected to transform it to what you need and applying them in the opposite order. Some software engineers take to this thought and think it is awesome and best way to do sensible precise illustrations. Some embrace it a tad, and others draw things where they are required in the size and direction needed.

Setting Your Own Coordinates

When you initially connect a Canvas to a Bitmap the direction framework is as far as the quantity of pixels in the Bitmap. Frequently, nonetheless, you need to work with an alternate direction framework. For instance, you should work with the beginning in the center and the x and y coordinate going from - 1 to +1.

You can set any direction framework you want to work with utilizing reasonable transformations.

If your direction framework runs from xmin to xmax and from ymin to ymax you can apply it to the material using:

```
c.scale(width/(xmax-xmin),height/(ymax
ymin)) c.translate(- xmin,- ymin)
```

where width and tallness are the size in pixels of the bitmap.

Using this detailing the y coordinate increments down the screen, as did the first pixel coordinates.

If you need the y direction to increment up the screen then, at that point, utilize the transformation:

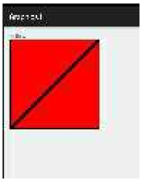
```
c.scale(width/(xmax-xmin),- stature/(ymax
ymin)) c.translate(- xmin,- ymax)
```

and notice the change to ymax in the second line.

So,for example, if you wanted to drawa graph using coordinates between 0,0 in the bottom left corner and 10,10 in the top right, i.e. y increasing up the screen,youwoulduse:

```
c.save()

val xmax=10f
val xmin=0f val
ymax=10f val
ymin=0f val
width=500f
val height=500f

c.scale(width/(xmax-xmin), - tallness/(ymaxymin)) c.translate(- xmin, -
ymax)
paint.setStrokeWidth(.4f)

c.drawLine(0F, 0F, 10F, 0F, paint) c.drawLine(0F, 0F, 0F, 10F, paint)
c.drawLine(0F, 0F, 10F, 10F, paint)

c.restore()
```

This draws tomahawks and a 45 degree line:



Notice that when you change the direction framework any remaining estimations change too. Henceforth the stroke width must be set to 0.4 as it is presently not as far as pixels.

Simple Animation

To wrap this section up we will invigorate a ball ricocheting around a Canvas, or a Bitmap depending your perspective. This may appear to be an odd subject to end on, particularly since we won’t do the occupation in the manner that most Android developers would go with regards to it. Indeed Androidhasa range of different animation facilities– View animation, Value animation and so on. Nonetheless, not a solitary one of them shows the essential way that unique designs work

and before you continue on to learn more complex methods of making activity it is a smart thought to discover how things work at the most minimal level.
This model shows you something liveliness, yet additionally about the issues of making dynamic illustrations of any sort in the Android UI. One admonition– don’t accept this is everything to be aware of Android movement or that this is the most ideal way to do things.
To quicken something in the easiest and most direct manner you should simply draw the shape, change the shape, eradicate the old realistic, and draw it again.
In many frameworks this is generally accomplished at the least conceivable level by utilizing a clock to call an update work which deletes the shape, does the update to the shape and afterward draws it at its new area. You can adopt this strategy in Android, yet for different reasons it isn’t the status quo normally done. It must be conceded that there are some slight challenges, yet beating them isn’t hard and is very instructive.

To perceive how everything functions we should simply skip a “ball” around the screen. This is pretty much the“hi world” of basic 2D sprite-based graphics.
So start another Android Studio venture and spot an ImageView on the plan surface. This is the main UI component we need.
We want a bunch of items and qualities that are available from various strategies and that have a lifetime as old as application. The easiest method of accomplishing this is to set up private properties:

```
private val b = Bitmap.createBitmap(width, height,

Bitmap.Config.ARGB_8888) private val c: Canvas=Canvas(b) private val
paint: Paint = Paint()
```

First we make a bitmap and partner it with a Canvas. The Paint object is made to try not to need to make an occasion each time we update the graphics.
Notice that different pieces of the program will have to get to width, tallness of the play area:

```
private val width = 800 private
val stature = 800
```

We are additionally going to require properties to record the ball’s position, its range and speed. For effortlessness we should simply utilize the default pixel directions of the Bitmap:

```
private var x = 463f
private var y = 743f
private var vx = 1f private
var vy = 1f private var r =
30f
```

Now we have these factors characterized we can continue on with the OnCreate capacity and set up the shade of the play region and the Paint object used to draw the ball:

```
c.drawColor(Color.WHITE)

paint.setAntiAlias(false)
paint.Style = Paint.Style.FILL
```

You might be wondering why AntiAlias is set to false, i.e. turned off. The reason is that its dithering algorithm makes it hard to remove a graphic by redrawing it in the background color. Try changing false to true in the final programtoseewhattheproblemis.

We likewise need to set the bitmap we are attracting on to the display:

```
imageView.setImageBitmap(b)
```

Now we are on the whole prepared to begin drawing the animation.

Timer and Threads

Now we come to the inward operations of the animation.
We really want a Timer object that runs a capacity each so many milliseconds: `val clock = Timer()`
The clock object has a scope of timetable capacities which run a capacity, really a strategy in a TimerTask object, at various occasions. The one we want is:
`timer.schedule(TimerTask,delay,repeat)`
which pursues the TimerTask postpone milliseconds and each recurrent milliseconds later that. The timings aren't exact and it could take longer than indicated for the TimerTask to be run.

The simplest way to createthe TimerTask is to use anobject expression. You can'tuse a lambda because the TimerTask is an object with a constructor and some additional methods i.e. itisn't a SAM:

```
timer.schedule(object : TimerTask() {
abrogate fun run() { update()
}
}
, 0, 10)
```

Thiscreates anew TimerTask andoverrides its runmethod. The run method is called when the Timer is triggered. All it does is to call the new function update, which we have yet to write, that does the update to the ball's position etc. The final two parameters specify a 0 millisecond delay in triggering the first call and then10 milliseconds as the repeat period. That is, update will be called every 10 milliseconds or so. If the processor is busy doing something elseitcouldbemorethan10millisecondsbetween The update work is reasonably easy:

```
fun update() {
paint.color=Color.WHITE
c.drawCircle(x, y, r, paint) x = x +
vx
y = y + vy
if (x + r >=
width) vx =
vx if (x - r <=
0) vx = - vx
if (y + r >=
tallness) vy =
vy if (y - r <=
0) vy = - vy
paint.color=Color.RED
c.drawCircle(x, y, r, paint)
imageView.invalidate(
)

}
```
First it sets the shading to white and draws the ball, a circle. This deletes the ball at its old position, recollect the foundation is white. Next it refreshes the situation by adding the speeds toward every path. To ensure that the ball skips we test to check whether it has arrived at a limit and assuming it has its speed is switched. At last, the shading is set to red and the ball is drawn at the new position.
If the function was to stop at this point then everything compiles and runs, but you won't see the ball move. The reason is simply that the UI is drawn once at when the program is initially run and then only when it is necessary becausethe user hasinteracted withitortheorientationhas changed,etc.As a result the bitmap displayed by the ImageView object would be changed every 10 milliseconds, but it would not be redisplayed. To make the UI update we want to call the ImageView's negate strategy which essentially tells the UI to redraw it. Notwithstanding, assuming you put this in

toward the finish of the update work you get a blunder message something like:

android.view.ViewRootImpl$CalledFromWro Only the first string that made a view chain of importance can contact its views.

The justification behind this is that the Timer object utilizes another string to run the TimerTask. This is regularly what you need to occur, however for this situation it is an issue. It is an issue that regularly happens in making a complex Android application and something you must figure out how to adapt with.
If you are new to stringing, this clarification may help.
When you run an Activity it is alloted a solitary string or execution. A string is a unit of execution and it is the thing that submits to your guidelines. In a total framework there are many strings of execution – some running and some suspended. The working framework picks which strings get to run such that endeavors to make them all appear to be making progress. The single string that the

Activity gets is for the most part called the UI string since its occupation is simply to deal with the UI.It reacts to occasions from the Activity
like OnCreate and from the client like a Button click. At the point when the UI string reacts to an occasion it submits to the occasion controller and afterward returns to sitting tight for the following occasion. This is the sense wherein each Android application is essentially an assortment of occasion controllers that the UI string executes when the comparing occasion occurs. The large issue is that the UI occasion is truly just cheerful when it sits around aimlessly. Then it just waits for an event and processes it at once. This makes the client think your application is extremely responsive on the grounds that snaps and other information are followed up on without a moment's delay. If you give the UI thread a long task to do, for example you write a lot of processing into an event handler, then it isn't just waiting for the user to do something, and the user starts to think that your app is slow and sluggish. At the super the UI string can be kept 100%

occupied with accomplishing something and afterward the whole UI appears to freeze up.

In short the UI string ought not be utilized for serious calculation or anything that takes in excess of a couple of milliseconds. The method for accomplishing this is to utilize different strings. This is a principle subject of Android Programming: Structuring Complex Apps.

The UI string makes the UI and to keep away from issues of
synchronization just the UI string can interface with the UI. That is, just the UI string can get to the UI. This is a genuinely normal way to deal with executing a UI and not in the slightest degree extraordinary to Android.

So what happens is that the Timer attempts to run its TimerTask and this thus runs the update work, however utilizing the string the Timer runs on rather than the UI string. All is well until the last guidance of update, which endeavors to utilize a technique that has a

place with an ImageView article and this it can’t do on the grounds that it isn’t the UI string.Thus the mistake message.
At this point numerous Android developers surrender and attempt something else altogether.A portion of these methodologies do enjoy benefits,see the Handler class for instance for a decent other option. In any case, the Android system gives a technique to simply such a situation: runOnUiThread(Runnable)
This is a strategy for the Activity item and you can utilize it from any string that approaches the Activity article’s techniques to run a capacity on the UI string. If the thread using it happens to be the UI thread then no harm done, the function is just called. On the off chance that it isn’t the UI string then the call will be conceded until the UI string is accessible and afterward the capacity will be run. As consistently the capacity shouldn’t keep the UI string occupied for a really long time or
the UI will become languid or even freeze completely.
The Runnable is an Interface that has a solitary run strategy that is the capacity that is

executed on the UI string– this implies it is a SAM (Single Abstract Method) and we can utilize a lambda to improve the code:

```
runOnUiThread { update() }
```

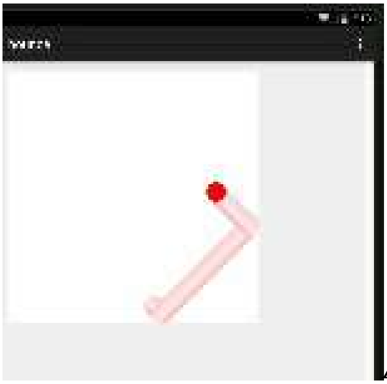
This guarantees that update is run on the UI string. Assembling this all gives:

```
timer.schedule(object : TimerTask() {
abrogate fun run() {
runOnUiThread { update() } }
}

, 0, 10)
```

This resembles a wreck of settling and wavy supports, yet you ought to have the option to follow the logic.

Now when you run the program you will see the red ball bob gradually and easily around the screen.How great the movement is depends what you

run it on. On the emulator is can be slow and sporadic; on a genuine gadget it ought to be fine:

*Animation complete with a path to show how the ball moves*

Now you know no less than one method for permitting a non-UI string collaborate with the UI. There are such countless ways of executing activity that this is only one of many beginning stages, yet with a comprehension of this one the others will appear to be more straightforward.To utilize this methodology,the design of this showing project could be improved. For instance, the ball definitely should be a Ball class total with its position and speed properties and its update strategy. This way you gain the advantages of article direction and you can enliven loads of balls around the screen with almost no extra effort.

Listing

The total posting of the activity program is:

```
import android.os.Bundle

import
android.support.design.widget.Snackbar import
android.support.v7.app.AppCompatActivity import
android.view.Menu
import android.view.MenuItem
import
kotlinx.android.synthetic.main.activity_main.* import
android.graphics.Bitmap
import
android.graphics.Canvas
import android.graphics.Color import
android.graphics.Paint
import
kotlinx.android.synthetic.main.content_main.* import java.util.*

class MainActivity : AppCompatActivity() {
private val width = 800 private val tallness = 800

private var x = 463f private var y = 743f private var vx = 1f private var vy =
1f private var r = 30f

private val paint: Paint = Paint()
private val b = Bitmap.createBitmap(width, height,
Bitmap.Config.ARGB_8888) private val c: Canvas = Canvas(b)

abrogate fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

c.drawColor(Color.WHITE)
paint.setAntiAlias(false)
paint.Style = Paint.Style.FILL imageView.setImageBitmap(b)
val clock = Timer()
timer.schedule(object : TimerTask() {

supersede fun run() {
runOnUiThread { update() }
}
}
, 0, 10)

} imageView.invalidate(
)

fun update() {
paint.color = Color.WHITE c.drawCircle(x, y, r, paint) x = x + vx
y = y + vy
if (x + r >= width) vx = vx if (x - r <= 0) vx = - vx if (y + r >= stature) vy =
vy if (y - r <= 0) vy = - vy paint.color = Color.RED c.drawCircle(x, y, r,
paint)


}
```

There is such a long way to go regarding designs it is hard to choose things you want to know. To discover more with regards to how the standard UI functions you want to investigate the OnDraw occasion and how to make your own View object that render designs. You really want to look into Android’s vector illustrations utilizing shapes and way. You really want to be familiar with the various kinds of liveliness that are accessible and in the end you really want to learn
about OpenGL and its help for equipment sped up 2D and 3D graphics.

Summary

☐ The subject of Android designs is colossal and there is in every case more than one method for moving toward any errand. This section is a first gander at what you may call UI-based graphics.

☐ The Bitmap is the essential designs object. It comprises of a square shape of pixels that can be set to any color.
☐ The ImageView is a broadly useful UI part that can be utilized to show a scope of designs objects including a Bitmap.
☐ You can draw on a Bitmap utilizing a Canvas object which has an enormous number of various drawing methods.
☐ The tone and drawing style utilized by numerous individuals of the Canvas techniques is dictated by the properties of a Paint object. ☐ The Canvas object additionally upholds changes which can be utilized to alter where a realistic is drawn, its size, pivot, etc.
☐ Transformations can be utilized to

normalize the drawing of illustrations objects at the origin.

☐ Transformations can likewise be utilized to change the default pixel coordinate framework to anything you need to use.

☐ Simple movement is conceivable utilizing only a Bitmap, Canvas and an ImageView.
☐ Only the UI string can adjust the UI.
☐ The Android Timer can be utilized to enliven 2D illustrations, however you have guarantee that it runs the code on the UI string using the runOnUIThread method.

Chapter 13 LifeCycleOfAnActivit

One of the things that you need to become acclimated to when programming for a versatile stage is that your application can be closed down and restarted absent much by way of caution. This is the kind of thing that

frequently makes amateurs and developers from different stages commit errors.You need to figure out how to adapt to this beginning stop presence and that, for an Android application, it is an intense life simply attempting to remain alive.
I as of late experienced an Android application, made by a huge notable organization, that requested that I fill in a structure. I was going to press the Submit button when I unintentionally shifted the telephone and the screen auto-rotated. When I looked again there was a clear structure! I needed to begin once again and, being a developer,Icouldn'tresisttestingtoseeifthedata vanishedfromtheform when I rotated the phone a second time–it did! The programmers fromthis high-profile company had no idea about creating an Android app beyond the basics.
Don’t fall into a similar snare. Look into application lifetime, state and continuing information. It might seem like an exhausting subject yet it is imperative to the working of a UI.

Lifetime and State

Mostprogrammersareusedto theidea thattheirapplication willbestarted by the user, used and then terminated by the user. If the application is restartedthenitisusually uptothe usertoloadwhatevercontexttheyneed by way of documents, etc. Sometimes an application has to automatically retain state information from run to run. For example, a game might keep track of which level a player had reached and the cumulative score, but this is about as complicated as things get.

For an application running on a cell phone things are altogether different. The actual idea of the gadget implies that any application could be hindered at any second by an approaching call or the client settling on a telephone decision. Somewhat this need to make applications“interruptible” has been taken up by cell phone working frameworks as a more broad rule. Android, for instance, will stop your application running since it needs the memory or needs to save battery life. It even ends and restarts your application assuming a media reconfiguration is required.

For instance, as we found in a past section, in the event that the client changes the direction of the gadget then your application is ended and restarted. This isn't exactly a restart without any preparation in light of the fact that the framework saves and reestablishes some state data naturally, yet precisely how this works is something we need to find out. The bottom lineis that when youprogram under Android–and mostother mobile operating systems–you have to care about the life cycle of your app andyou have to take steps to ensure thatits stateis preserved so it canseem to the user that nothing at all has happened– even though your app has effectively been hit on the head and disposed of beforebeing dragged back to continuefromwhereitwas.
Being an Android application is a risky reality and not in any way like a Windows or a Linux application which can act as though they have the machine all to themselves.

The Life Cycle of an App

The various states that an Android application can be in and the changes between them can appear to be convoluted– that is on the grounds that they are. Assuming you are as yet suspecting as far as a work area application that beginnings, runs and is ended by the client, this degree of intricacy can appear to be superfluous– and maybe it is. However, these are the principles that we need to play by.
The Activity class has a bunch of overrideable occasion overseers for every one of six expresses an Activity can be in. These work two by two, organizing the periods of the Activity:

☐ onCreate and onDestroy section the whole existence of the
Activity in memory and can be viewed as at the furthest level. This pair is considered

when the application is stacked into memory or dumped from memory and section the whole lifetime of an Activity. At the point when it is first stacked the onCreate is set off and when the application is discarded onDestroy is set off. You unmistakably need to utilize these two to set up and obliterate any assets which are required for the whole lifetime of the application. In any case, if your application is taken out by the framework it will call onDestroy not long prior to emptying the Activity and onCreate when it reloads it. This implies that onCreate might be called when you want to reestablish the condition of the Activity so the client doesn’t see any interruption.

☐ onStart and onStop section any period that the application is visible. It could be that the Activity is behind, say, a modular exchange box. The Activity is noticeable however not interfacing with the client. This pair of occasions can be set off different occasions during the whole lifetime of the app.
Simple applications can generally overlook the onStart and onStop occasions in light of

the fact that the Activity is as yet in memory and doesn’t lose any assets or state.

The fundamental utilization of onStart and onStop is to offer the application a chance to screen any progressions that may influence it while not interfacing with the client.To befuddle the issue much more there is likewise an onRestart occasion which happens before the onStart occasion yet provided that this isn’t whenever the Activity first has terminated the onStart - that is this is a genuine restart.

☐ onResume and onPause section the period that the Activity is in the frontal area and connecting with the user.
Again this pair of occasions can happen on numerous occasions during the whole lifetime. The onResume occasion happens when the Activity gets back to the closer view and going about its standard business. The onPause occasion happens when the client switches away to another application for example.

The Simple Approach

At this point you reserve each privilege to be mistaken for such countless changes of status and reacting to every one. The most compelling thing to stress over is the finished restart of your application which triggers an onDestroy and an onCreate. This is the one in particular that annihilates the present status of the application,the others are essentially openings for your application to decrease the heap on the framework, or to save some client information for good measure. As the onDestroy is generally a chance to tidy up assets to keep away from releases, most basic applications truly just need to deal with the onCreate event.

It is enticing to believe that this beginning and halting is very much like a work area application being ended and afterward the client choosing to utilize it once more, so that it’s fine for the application to get going as though it was being run interestingly. On account of an Android application this isn’t what the client expects by any means. Your application can be ended by the framework without the client knowing at least something about it. At the point when the client attempts to recapture admittance to an application they will by and large anticipate that it should carry on from where they left it. It may be a complete restart as far as you are concerned, but the user just switched to another app for a few minutes and expects to find yours as they left it.

What this implies is that you can’t think about onCreate as though it was the constructor for your application.At the point when onCreate is terminated it very well may be whenever your program first has at any point run or it very well may be a restart for certain things that you made on a past start still in presence. For straightforward applications you can frequently basically make everything again as though it was the initial time your application had run, and sometimes you really want to test the savedInstanceState to check whether you are in reality restarting. Notice likewise that regularly these occasions additionally will more often than not happen in successions. For instance, an application that has recently gotten the onPause occasion is probably going to proceed to get the onDestroy occasion on the grounds that the framework will eliminate it to let loose memory.It is a slip-up to attempt to contemplate groupings of events

and ask which one ought to do a specific introduction or tidy up. Simply contemplate the express that your application is moving into and place the vital code into that occasion handler.

Lifecycle Explorer

There is no more excellent method for feeling alright with the lifecycle and its occasions than to compose a short demo program that shows you when they occur.
Start another Android Basic Activity project,

tolerating all defaults, call it Lifecycle and afterward acknowledge every one of the defaults to begin rapidly. In the Layout Editor eliminate the“Hi World” string and spot a TextView on the plan surface. Next resize it so it fills the vast majority of the space and eliminate its default text passage, ensure it is obliged and has a huge wiggle room around it:

The code we want is genuinely simple, the main stunt is to make sure to call every one of the framework gave occasion overseers that you have superseded.Assuming you don’t do this the application just ends when you run it. The OnCreate occasion overseer actually needs to develop the UI, however presently we get a reference to the Java object addressing the TextView into a worldwide variable so the occasion controllers can get to it:

```
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)

setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

textView.append( “Create\n”)
```

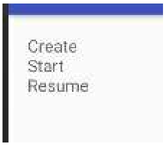
The remainder of the program essentially abrogates every one of the occasion overseers thus, calls the first occasion controller and afterward adds an instant message to textView:

```
abrogate fun onStart() { super.onStart()

textView.append(“Start\n”)
}

abrogate fun onPause() { super.onPause() textView.append(“Pause\n”)
}

abrogate fun onResume() { super.onResume()
textView.append(“Resume\n”)
}

abrogate fun onStop() { super.onStop() textView.append(“Stop\n”)
}

supersede fun onRestart() { super.onRestart() textView.append(“Restart\n”)
}

abrogate fun onDestroy() { super.onDestroy()
textView.append(“Destroy\n”)
}
```

An extremely straightforward and drawn-out kind of program.

Trying It Out

If you currently run this program you can utilize it to discover when the existence cycle occasions occur. You may be amazed to discover that when you previously run the program you get:



```
Create
Start
Resume
```

If you consider it briefly, this isn ’t nonsensical as the application is being stacked, becoming apparent and in the forefront, and henceforth the proper occasions are terminated in turn. Beginners regularly accept that the existence cycle occasions some way or another supersede one another. That is, assuming an onCreate has been terminated then this is the huge occasion in the Activity’s life thus the

others will not occur. This isn’t true and you want to ensure that you put activities into the occasion overseers that truly are fitting to the existence cycle state. For instance, assuming you put something in the onResume occasion controller ensure you understand that it will be terminated when the application initially fires up just as when it simply being resumed. If you attempt different things, such as squeezing the Home key and choosing another application, then, at that point, you will see other occasion arrangements, obviously just when you continue the Lifecycle app.
For example, pressing the Home key, then showing the task managerby long pressing the Homekey and reselecting your app results in: Pause, Stop as the app is removed from the foregroundand then Restart,Start, Resumeas the appis loaded, becomes visibleand then resumes theforeground and interacts with the user.
You can give different activities a shot yet there is one thing you should test– changing the direction. If you are using the emulator then press CtrlF11. At the point when the

screen direction transforms you will see that the TextView has been cleared and Create,Start,Resume have been added. This is because when you change orientation the app is completely stopped and then completely restarted, i.e. it is as if the app was being run from scratchforthefirsttime. This assertion is practically obvious– yet not quite.

Retaining State – the Bundle

When you change direction your application is halted and restarted.At the point when this happens the TextView is reset to its default state when the application loads. This depiction of what's going on is maybe what you may anticipate. Be that as it may, this isn't the finished story.

The framework attempts to assist you with the issue of having your application halted abruptly and restarted. It will naturally hold the province of UI components that can be changed by the client, and it consequently reestablishes them when the application begins. So on a basic level you can at first overlook the issue of an application restart on the grounds that the framework reestablishes your UI. This is the explanation that some Android developers accept that everything is“typical” and there is no compelling reason to study the lifecycle of an application. This is valid from the outset, yet later your application will advance past what the framework gives by default.

Automatically saving and reestablishing the UI ’s state is the thing that the savedInstanceState boundary in the onCreate occasion controller is all about:

```
abrogate fun onCreate(savedInstanceState: Bundle?) {
```

A Bundle is a bunch of key/esteem sets which is utilized to save the qualities put away in UI components when the application is halted by the framework. It stores id/esteem sets and when the application is restarted the Bundle is utilized to introduce the qualities in the

relating UI components. Notice that assuming the client stops your application by eliminating it from the new applications list then the savedInstanceState is obliterated,the application truly begins once again once more,and the onCreate isn’t passed a Bundle to reestablish the UI. At the end of the day, savedInstanceState possibly reestablishes the UI when the application has been halted by the system.
It is likewise worth seeing that the reestablish will work to an elective format stacked due to a setup change.For instance,it will reestablish state to a scene variant of a format just as the first representation rendition. The only thing that is in any way important is that the current design has View objects with the right ids.
Atthis pointyou are probably wondering why the TextView object wasn't restoredby thesystemwhenthedevice was rotated? Thesimpleanswer is thataTextViewobjectisn'tintendedforuserintera itissupposedto just be used to show static text labels and so the system doesn't save and restore it.
You can see the programmed save and

reestablish in real life assuming you add an EditText input field on the plan surface of the Lifecycle Explorer. Presently assuming you enter some message into the EditText field it will be held on the off chance that you pivot the gadget. Notwithstanding, assuming you press and hold the Home key,eliminate the application and afterward start it once more you will see that the EditText field is clear again:

The text in the EditText field at the lower part of the screen is protected during a screen rotation.

☐ The overall standard is that any UI component that can be altered by the client is naturally saved and reestablished. Any progressions that your code makes or that the client makes in complex UI parts are lost except if you find ways to protect them.

Saving Additional UI Data

The framework will save and reestablish the condition of the UI components that the client can change, however it won’t store any that your code changes. It additionally doesn’t consequently save and reestablish whatever other information that the client or your code might have produced that isn’t inside the UI. In these cases you need to keep in touch with some code that saves the qualities and reestablishes them. There are loads of methods of saving the condition of an application as it is begun and halted by the framework. One of the least complex is to utilize the Bundle object that the framework uses.

The framework fires the onSaveInstanceState occasion when it is going to add information to the Bundle and save it. To save some extra information you should simply abrogate the default occasion overseer. For instance, assume you need to save and reestablish the information in the TextView in the Lifecycle Explorer. First you need to save the data:

```
supersede fun onSaveInstanceState(outState: Bundle?) {
super.onSaveInstanceState(outState)

outState?.putCharSequence(“myText”,textView.text)
}
```

Notice that we save the text content of the textView object as the worth and utilize the key“myText”. Much of the time it would be smarter to make a string consistent for the key. The key can be any identifier you want to utilize, yet it includes to be exceptional inside the Bundle as it is utilized to recover the information you have put away in the Bundle.

Now to recover the information and spot it into the TextView we want to change the onCreate occasion handler:

```
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)

setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

textView.append( “Create\n”)
if (savedInstanceState != null){
textView.text=savedInstanceState.getCharSequenc
e(“myText”)
}
```

This gets going in the typical manner however presently we verify whether savedInstanceState has any information.Assuming that it does we recover the put away text utilizing the key“myText”.

There is zero excuse not to utilize the onCreate occasion overseer along these lines, however the framework likewise fires an onResetoreInstanceState occasion when it is going to play out its own“programmed” reestablish of the UI for you. You can abrogate this occasion controller to keep the code that reestablishes the application’s state kept out of the onCreate occasion handler.

For instance, you could have written:

```
supersede fun onRestoreInstanceState(savedInstanceState: Bundle?) { if
(savedInstanceState != invalid) {
textView.text =
savedInstanceState.getCharSequence(“myText”)

}
super.onRestoreInstanceState(savedInstanceState)


}
```

Do make sure to call the super.onRestoreInstanceState assuming you need the framework to reestablish the remainder of the UI in the standard manner. There are put and get techniques for a scope of standard

information types. All simple types, byte, integer and so on, are supported, as are strings, arrays and other simple data structures. You can likewise make your own classes that can be put away in a Bundle by executing the Parcelable interface. Notice you can save subjective information, and not only information for the design. You can likewise make your own Bundle occurrence and utilize it for information stockpiling, diligence and for moving information to different pieces of a program. There are numerous standard classes the Bundle doesn't uphold. In these cases you need to make your own courses of action to save information to storage.

Often all you really want to do to ensure that your application keeps up with its state between framework restarts is to utilize the savedInstanceState Bundle object.This methodology likewise diminishes a large part of the reasonableness of lifecycle the board to carrying out the onSaveInstanceState and onRestoreInstanceState occasion controllers. This is such a ton easier than stressing over all of the diverse life cycle events. As an activity, you would now be able to return to the iCalc model in Chapter 3 and make its presentation and current worth continue through a screen rotate.

Complex UI Elements

One of the snares sitting tight for you is the issue of precisely what is naturally saved and reestablished.For instance,toward the beginning of this part we had the narrative of the application that lost the client’s structure information when it was rotated.
Given what we presently know about the auto-saving and reclamation of client modifiable UI components, you may be considering how this could occur? The appropriate response is that the developers of

the application had presumably become used to the programmed perseverance of UI state and didn’t try to make sure that turn had no impact. It had an impact in light of the fact that the structure was being downloaded from a site and showed in a WebView control. A WebView control is persevered by the framework, however it reloads the page when it is reestablished. This implies that on a revolution the structure was reloaded as unfilled and the client’s information was lost.

☐ You generally need to make sure that things fill in as you anticipate. Continuously test what befalls your UI on a rotation.

Advanced State Management

For fulfillment it is actually important that there are a lot more ways to deal with keeping up with state. Later you should find how to store bunches of client information

locally for longer term determination, and this is frequently enough to carry out state the board however an arrangement change.
There are additionally further developed state the executives issues when you come to utilize Fragments,the subject of Android Programming*: Mastering Fragments and Dialogs*.For this situation you can utilize retainedInstance to ask the framework not to obliterate a whole Fragment. This implies that every one of the information put away in the Fragment is held despite the fact that the Activity might be eliminated from memory. This makes it conceivable to utilize a Fragment as a store of state.
A definitive in getting sure that things going as you need is to deal with the arrangement change yourself. You can do this by rolling out an
improvement to the show. Assuming you do this then it is dependent upon you to roll out the improvements required when the onConfigurationChanged occasion happens.You could,for instance,select to quicken buttons and other UI objects into new positions or simply disregard the need to

reconfigure altogether.
This is completely exceptional and for most applications you can get by utilizing only the onSaveInstanceState and onRestoreInstanceState occasion handlers.

Summary

☐ Android, in the same way as other versatile OSs, will eliminate your application from memory and restart it as it needs to.

☐ The client won’t know that your application has been annihilated and reproduced and will anticipate that it should proceed from where they left off.

☐ The framework signals changes in state to your Activity through a confounded arrangement of occasions. You really want to see how these work to make your Activity continue effectively later the different degrees of suspension.

☐ onCreate ought not be viewed as

the“constructor” for your application since it is considered when the application is reestablished just as when it is run for the first time.

☐ The framework will store the condition of any client modifiable UI parts and reestablish it when the Activity resumes.
☐ The information is put away in an extraordinary example of a Bundle, a bunch of key/esteem sets, called savedInstanceState.

☐ The framework makes you when it is about aware of save and reestablish information from savedInstanceState by terminating the onSaveInstanceState and onRestoreInstanceState occasion handlers.

☐ You can supersede both of these occasion controllers to save and reestablish additional information in the savedInstanceState Bundle.

☐ For some straightforward applications you can for the most part disregard the lifecycle occasions and focus on utilizing the

onSaveInstanceState
and onRestoreInstanceState occasion overseers to persevere data.

☐ You should consistently make sure that UI and different components are endured through a suspension of your application. You can test utilizing a turn design change.

☐ There are other further developed methods of saving state which you should find later on. You can't utilize a Bundle for everything.

Chapter 14 Spinners

Working with Android Studio makes assembling the UI simple with an intelligent manager, the Layout Editor, yet you actually need to discover how to deal with the things it isn't exactly so great at.In the following two parts of our investigation of Android we take a gander at spinners and pickers, the subsequent stage up from buttons and text controls.

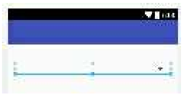
Spinners are what are alluded to as drop-down records or something almost identical in other UIs. They permit the client to pick from a rundown of potential things. Pickers, which are the subject of Chapter 15, additionally permit the client to pick a thing, yet for this situation the things are all the more barely characterized– a date, a period or a number.

The Spinner and the Layout Editor

☐ The spinner presents a bunch of options in contrast to the client and allows them to choose one.

Putting a Spinner into your undertaking is

pretty much as simple as utilizing the instrument Palette in the Layout Editor,however you can’t move away without some code to make everything work. Specifically, you want to characterize the rundown of things that the Spinner will show when the client actuates it.
At the point when you place a Spinner on the plan surface all you get is a clear control with a non-working dropdown icon:



Right now the Layout Editor just offers insignificant help for setting up the Spinner for certain information to show. The easiest kind of thing to show the client is a rundown of text things, and you may well think that the most immediate method of doing this is to utilize a String cluster. It is, yet things are somewhat more convoluted than this. They are basic, nonetheless,on the off chance that you make an asset to

determine the String exhibit on the grounds that for this situation the framework will do all things required to stack the data.

Setting the substance of the Spinner utilizing a String cluster in code is a decent method for perceiving how the Spinner functions, and this is our main thing next, yet it isn't the way it normally happens.
Androidprovidesacomprehensivesystemofreso
strings,imagesand lots of XML files. The idea is that you can produce customized versions of you application just by changing the resource files, i.e. without having to modifyyourcode.Fordetailedinformationonusi
resourcesreferbackto Chapter11.
For the situation of a Spinner you could set up an asset that was a String cluster that gave the rundown of things to be shown. To make a variant of your application for an unfamiliar market you could get the rundown meant make an asset in the new language.
Resources are a good thought and you should utilize them for generally fixed strings and fixed information overall. Up until this point, the models have would in general stay away from assets to simplify everything, except for

a Spinner you want to know how to code up a String cluster asset and use it.
Android Studio ends up being beneficial asset support however in certain spaces it is deficient. For instance, in an ideal world the framework would assist you with making a String or String exhibit asset, however right now it just gives assistance to straightforward string esteems. It will assist you with making a String asset, yet offers no help for a String cluster, for which we must choose between limited options however work with the XML file.
Find the record strings.xml in the res/values catalog and open the document strings.xml.Add to this the String cluster definition: <string-exhibit name=“country”>

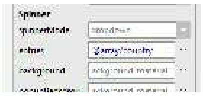
```
<item>Canada</item>
<item>Mexico</item>
<item>USA</item>

</string-array>
```

The significance of the XML is self-evident and this is the benefit of an intelligible markup language.

If you have explored resources using the

Resource window which appears when you select the three dots option in the Attributes window you might worrythatthisnewresource,i.e.SpinnerList, doesn'tappear.Italsodoesn't appear in the Resource editor that you can select while editing the XML file.

The explanation is that, as of now,Android Studio doesn ’t uphold the task, altering or formation of String exhibits other than physically. Nonetheless, utilizing the new asset is genuinely simple. The id of the asset is @array/nation and this must be composed into the passages property in the Attributes window as the Resource picker doesn’t uphold exhibits at the moment:



Spinner
spinnerMode dropdown
entries @array/country
background [illegible]

If you enter the id effectively you will see the principal thing show up in the Spinner inside the proofreader. At the point when you run the application you will see each of the passages when you drop down the list:

For straightforward arrangements of choices that can be indicated at configuration time this is all you want to be aware of how to stack a rundown of things into a Spinner. Practically speaking, notwithstanding, a Spinner ordinarily must be stacked with a rundown at runtime and afterward you want to be aware of the ArrayAdapter.

Introducing the ArrayAdapter

For the situation of UI gadgets that show arrangements of things, Android has a considerably more broad component to adapt to the distinctive kind of things you could show. Gadgets that show arrangements of things by and large work with an illustration of an“connector”.For additional on

connectors overall see Chapter 15.
A connector fundamentally takes a rundown of items and converts them into something that can be shown in the gadget. By and large, you can make custom Adapters to do shrewd things with arrangements of objects of your own so they show properly. As a rule, in any case, you can get by with simply utilizing the gave worked in adapters.

For the situation of the spinner the most regular decision is the ArrayAdapter. This takes a variety of objects of any kind and makes them reasonable for show by calling their toString() technique.However long the toString() strategy produces what you need to find in the Spinner then everything ought to work.
For the situation of a variety of Strings calling the toString() technique on each cluster component may seem like pointless excess, yet it is the value we pay to fabricate instruments that can adapt to more muddled situations.

So the game plan is to make a String exhibit with the things we need to show, utilize this to introduce an ArrayAdapter article, and afterward connect the ArrayAdapter to the Spinner.

Creating the cluster is easy:

```
val country = arrayOf(“Canada”, “Mexico”, “USA”)
```

The ArrayAdapter constructor can appear to be convoluted. It appears to be considerably more confounded in light of the fact that ArrayAdapter utilizes generics to permit you to determine the kind of every component in the array. *If you haven’t utilized generics previously,all you really want to know is that,as a general rule, generics are a method of making helpful classes, frequently assortments of items that can be of a sort you indicate.You realize you are working with a conventional when you need to indicate the sort it is to work with utilizing <type>.*

So instead of creating a special array Adapter for each type of array, an IntArrayAdapter, a StringArrayAdapter and so on, you simply have to specify the type as <int> or <String> when you use the generic ArrayAdapter type. For instance, to make an ArrayAdapter for a variety of Strings you would use: val

stringArrayAdapter=ArrayAdapter<String>(constructor parameters) The example is something similar for every one of the constructors and for various exhibit types. There are many ArrayAdapter constructors, yet they all need some fundamental data. They need to know the current setting, typically this, and the format to be utilized to show the rundown and the variety of information items.
The main troublesome one is the design to be utilized to show the rundown. This sounds like a ton of difficult work until you find that the framework gives some essential standard designs that you can only use. For our situation the format is:
andRoid.R.layout.simple_spinner_dropdown_item
Notice that this is really a whole number that decides the design asset to utilize and nothing more confounded. Assembling it all gives:

```
val stringArrayAdapter=ArrayAdapter<String>(

this,
andRoid.R.layout.simple_spinner_dropdown _item, country)
```

If you are considering what nation is, recall that we characterized a String cluster called country earlier.
The last advance is to indicate the

ArrayAdapter to use in the Spinner.
We can utilize its connector property:

```
spinner.adapter=stringArrayAdapter
```

You might have done the occupation in one line and the onCreate occasion handler:

```
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)
val country = arrayOf("Canada", "Mexico", "USA")
spinner.adapter=ArrayAdapter<String>(
this,
andRoid.R.layout.simple_spinner_dropd own_item, country)
}
```

If you run the program you will see a similar drop-down list we created utilizing only the Layout Editor.

Handling the Selection

The following inquiry is, how would you find that the client has made a selection?
The straightforward answer is that we need to attach to the Spinner's occasions and the OnItemSelectedListener is the interface that

has the onNothingSelected and onItemSelected occasion controllers. As the interface characterizes two occasion controllers it’s anything but a SAM thus we need to utilize an object.

If you enter the accompanying line:

```
val onSpinner=object:AdapterView.OnItemSelectedListener{
```

You would then be able to utilize the right snap Generate, Override Methods choice to make a stub:

```
val onSpinner=object:AdapterView.OnItemSelectedListener{ supersede fun


onNothingSelected(p0: AdapterView<*>?) {
TODO(“not implemented”)
}
abrogate fun onItemSelected(p0: AdapterView<*>?,
p1: View?, p2: Int, p3: Long) { TODO(“not implemented”)
}


}
```

It additionally adds an import for AdapterView.

You can see without a moment’s delay that you need to carry out two occasion handlers:

☐ onItemSelected– set off when the client chooses an item

☐ onNothingSelected– set off when the Spinner has no things or the client deselects

everything items
You may be astounded to see two events of <*> in the produced code. The two occasions are conventional and, as currently clarified, they can work with a scope of various kinds. The <*> is a sort projection which basically permits any sort to be utilized. For instance List<*> is a rundown of any sort. Obviously, the real kind isn’t known until runtime thus every component of the List is treated as an Any sort and it’s dependent upon you to project it to something more explicit. Kotlin adds checks to ensure your projects are safe. Let’s gander at the onItemSelected occasion overseer in more detail: supersede fun onItemSelected(p0:

```
AdapterView<*>?,
p1: View?,
p2: Int, p3:
Long)
```

What is this p0:AdapterView that has abruptly showed up? As clarified in before sections,each of the apparent parts of the UI compare to View objects of some sort. An AdapterView is the View object that compares to one of the showed things in the Spinner. You could, for instance, utilize the

AdapterView passed to the occasion controller to alter the presence of the showed item.
The p1:View boundary is only the offspring of the AdapterView that was really clicked.Note that a thing can be made out of more than one View item. Finally the p2:int boundary and the p3:long give the place of the view that was clicked in the connector and the column id of the thing that was selected.
You can populate a Spinner from an information base. For this situation the column id gives the data set line number, which isn't really as old as position in the Spinner. For a basic ArrayAdapter the position and id are something similar. Much of the time the main boundary you will be keen on is the int position which gives you the thing the client chose. For instance, place a TextView on the plan surface and change the
onItemSelected occasion overseer to read:

```
abrogate fun onItemSelected(p0: AdapterView<*>?,

p1: View?, p2:
Int, p3: Long)
```

```
{

textView.text=p2.toString()
}
```

All that happens is that the position boundary is shown in the TextView. Finally to associate the occasion dealing with object to the Spinner we really want to add it utilizing its onItemSelectListener property:

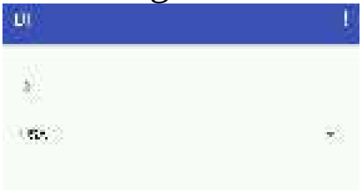
```
spinner.onItemSelectedListener=onSpinner
```

Obviously you could characterize the occasion taking care of protest and allocate it in one guidance. Assembling this gives the new onCreate:

```
abrogate fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

val country = arrayOf( “Canada”, “Mexico”, “USA”)
spinner.adapter=ArrayAdapter<String>(
this,
andRoid.R.layout.simple_spinner_dropdo wn_item, country)

spinner.onItemSelectedListener =
object:AdapterView.OnItemSelectedListener { supersede fun
onNothingSelected( p0:
AdapterView<*>?) {

}

abrogate fun onItemSelected( p0: AdapterView<*>?, p1: View?, p2: Int, p3:
Long) {

textView.text = p2.toString() }
```

}

If you currently run the program you will see something like:



The position showed relates to the component in the exhibit that has been chosen, counting from zero of course.

Most of the time is it enough to have the list number of the chose thing, however the AdapterView object has a couple of techniques that empower you to get the chosen thing. To audit the chose thing you could utilize the situation to record the first ArrayAdapter, or even the first String array,

however these aren ’t for the most part accessible to the occasion controller.So to recover the thing you would utilize the getItemAtPosition technique.For instance, to show the nation name you would adjust the setText call to:

textView.text = p0?.getItemAtPosition(p2).toString() Notice that the chose thing is returned as an article type and you need to project it before you can do anything with it.

There are different strategies that can be utilized to recover data or control the chose component, yet for most basic applications the itemAtPosition property is all you truly need.

Creating an ArrayAdapter from a Resource

To make an ArrayAdapter from an asset you want to utilize a static technique for the ArrayAdapter class, createFromResource. This simply needs you to determine the specific situation, the asset id, and the Spinner design. You should simply to supplant the formation of

stringArrayAdapter:
spinner.adapter=ArrayAdapter.createFromResource(

this,
R.array.country
,
andRoid.R.layout.simple_spinner_dropdown_ item)

With this change everything functions as in the past, however presently to change the things that show up in the Spinner you essentially alter the XML file.
You could likewise utilize the assets object to recover the String exhibit and afterward continue as though the String cluster had been characterized in code: val country=resources.getStringArray(R.array.country)

Changing The List

There are bunches of further developed things that you can do with Spinners, yet these aren’t experienced that regularly and principally happen when attempting to construct a custom client experience. The one thing that does occur often is the need to dynamically change the list of items. There are many slight variations on this, but essentially what you do is change the String array and then call the adapter's notifyDataSetChange method. For example, if you want to change Mexico, i.e. element one, to Greenlandyouwoulduse:

```
country[1]=“Greenland”
(spinner.adapter as ArrayAdapter<String>).notifyDataSetChanged()
```

You need to expressly give the connector property a role as it is excessively intricate for the compiler to infer.

The ArrayAdapter likewise has add, clear, eliminate and embed techniques which can be utilized to change the basic information, however for this to work the item holding the information must be modifiable.
You can’t adjust a String cluster thusly. What you want rather is an ArrayList.
If you change the presentation of country to:

```
val country= mutableListOf(“Canada”, “Mexico”, “USA”)
```

you can add“Greenland” to the furthest limit of the things using: (spinner.adapter as ArrayAdapter<String>).add(“Greenland”) Notice that for this situation the ArrayAdapter constructor utilized changes from one that acknowledges

a cluster to one that acknowledges a List of objects.
You can generally discover what number of things there are utilizing the count property.
How do you modify a list of items that are created using a resource? This is a tricky question because the ArrayAdapter creates a String array to hold the data which means you can't use the methods that modify a list. There are various methods of fixing this, however the most straightforward is to build your own List from the asset directly:

```
val country= mutableListOf(

*resources.getStringArray(R.array.country))
spinner.adapter=ArrayAdapter<String>(
this,
andRoid.R.layout.simple_spinner_dropd own_item, country)
```

With this form of the ArrayAdapter you can by and by utilize the add, and different techniques,to adjust the rundown of things.Notice the utilization of the Kotlin spread administrator to unload the String cluster into individual Strings so mutableListOf works correctly.

Summary

☐ Spinners are a method of introducing a rundown of choices for the client to choose from.

☐ Spinners can be mind boggling as far as their format and what they show, however the most straightforward model is to work with a variety of Strings.

☐ The variety of Strings can be made in the code or inside an asset file.
☐ The variety of Strings must be changed over into an ArrayAdapter object to be utilized with the Spinner.
☐ The ArrayAdapter gives a View object to every thing showed in the Spinner.

☐ There are two different ways (at any rate) to stack a String exhibit asset into an ArrayAdapter – utilizing its createFromResource or by stacking the asset as String exhibit and afterward continuing as before.

☐ Loading the String cluster enjoys the benefit that you can transform it into a List, which can be altered in code by adding or erasing components. You can’t change the length of a String array.

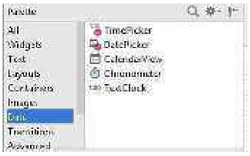
☐ To discover what the client has chosen basically utilize the onItemSelected occasion handler.
☐ To recover the thing that the client has chosen utilize the getItemAtPosition(position) method.

Chapter 15 Pickers

Android right now upholds three Pickers for dates, times and general numbers and they are significant ways of getting client input. Notwithstanding, they have experienced such countless amendments that they need straightforward documentation or rules how to utilize them. We should clear up the confusion.

Working with Pickers

☐ A picker is a“dial” that you can use to choose one of a foreordained arrangement of values.
In this sense it is a ton like the Spinner shrouded in the past part, however one that has a confined arrangement of choices.
There are two methods for utilizing a Picker, as a gadget, or as a discourse box. You will track down the TimePicker and the DatePicker, prepared to put on the plan surface, in the Date&Time part of the Palette.
The NumberPicker, notwithstanding, is a lot of lower down, in the Advanced section:

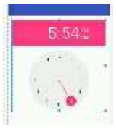


Although now and again you likely will need to make a Dialog box and should utilize a perplexing DialogFragment to wrap the Dialog and deal with its lifecycle, and there are numerous circumstances where utilizing the crude gadget will do the work alright. For additional on DialogFragment see:

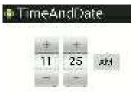
Android Programming: Mastering Fragments and Dialogs.

TimePicker

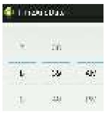
The TimePicker is an extremely simple method for beginning. Assuming you make another Android project, called TimeAndDate and acknowledge each of the defaults you can put a TimePicker,recorded top of the Date and Time segment of the Palette,on the plan surface very much like some other gadget, and measure and find it as required. In its default setup it shows in the Layout Editor utilizing the style android:timePickerStyle and has a simple clock face and an info region that can be utilized to establish the point in time by hauling the hands on the clock face:



Pickers are mind boggling gadgets that have numerous parts, every one of which can change appearance relying upon how they are styled and which adaptation of Android you are focusing on. While you can choose from a scope of topics, they don't all work with all SDK/APIs and they may to be eliminated sooner or later. For instance, before API 14 this subject was used:



API 14 took on the Holo topic and the picker looked like this:



What follows is the thing that you get assuming you just acknowledge the defaults. Notwithstanding, it is critical to understand that precisely what the client sees relies upon the form of Android they are utilizing. For example your up-to-date app running on the latest Android might look like the clock TimePicker but it you run it on a pre-Lollipop API such as Jelly Bean (API 18) then you will see the Holo-themed version as above. To help more seasoned forms then it is vital to make sure to test utilizing them. To utilize something that appears as though the Holo-themed form on all adaptations then you should simply track down timePickerMode in the Attributes window and utilize the drop down rundown to choose spinner or clock contingent upon the sort you need.
For simple in reverse similarity use similarity library v7 appcompat, which is remembered for your undertakings consequently by Android Studio, and stick with the AppCompat topics.

TimePicker in Code

To interface with the TimePicker you should simply use the get/set techniques for Hour and Minute. You can likewise programatically change the 12 hour/24 hour mode.Notice that the time is constantly returned in 24-hour structure regardless the gadget's mode is.
Although the currentMinute and currentHour properties are expostulated, the choices which do the very same thing, moment and hour, don't deal with prior renditions of Android. So for the second it appears to be desirable over utilize the censured techniques and overlook the alerts. For instance to set the TimePicker you would use:

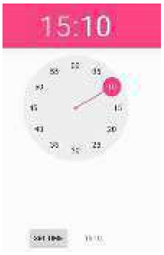
```
timePicker.setIs24HourView(true
) timePicker.currentMinute=10
timePicker.currentHour=13
```

The possibly task remaining is sorting out some way to find when the client has chosen a time.
You could give a button which the client needs to snap to affirm the new time. For instance, place a Button and a TextView on

the plan surface and add the accompanying Button click occasion handler:

```
button.setOnClickListener { view - >

textView.text= timePicker.currentHour.toString() + “:” +
timePicker.currentMinute.toString() }
```

This gives a way to the client set the time:



In many cases the troublesome aspect in utilizing a Picker isn ’t setting it up or getting the information from it, yet in handling that information into a structure in which your program can utilize it. For this situation we basically convert the time into a somewhat organized string
representation.

Updating the Time

What about getting an update each time the client changes the TimePicker? This requires an
OnTimeChanged occasion which
occasion controller for an

can be carried out with an OnTimeChangedListener interface as a lambda.
You can figure that when the time is changed by the client, the onTimeChanged strategy is called and the TimePicker that set off the occasion is passed as view, and its hour and moment setting as hourOfDay and moment, and everything necessary is to set the occasion overseer utilizing the setOnTimeChangedListener method.
For instance, to move the new an ideal opportunity to the TextView utilized in the past model you would use:

```
timePicker.setOnTimeChangedListener { timePicker, h, m - >

textView.text=h.toString() + “:” + m.toString()

}
```

Now assuming you run the program you will see the TextView change each time the client adjusts the TimePicker by whatever method. The full program, including the code for the button and the occasion controller is shown below:

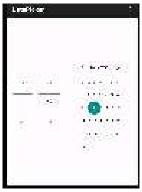
```
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

timePicker.setIs24HourView(true) timePicker.currentMinute = 10
timePicker.currentHour = 13

button.setOnClickListener { view - >
textView.text =
timePicker.currentHour.toString() + ":"

+ timePicker.currentMinute.toString( }

timePicker.setOnTimeChangedListener { timePicker, h, m
- > textView.text = h.toString() + ":" + m.toString()

}
}
```

DatePicker

There are two date gadgets in the current Android framework, DatePicker and Calendar view. The DatePicker has been being used since API 1 however Calendar view was just presented in API 11. The fundamental contrast between the two is the way that they look. As they work similarly and the DatePicker is more adaptable this is the gadget we will utilize. Likewise whenever you have encountered the manner in which TimePicker works there is very little to add to cover its date counterpart. To see the DatePicker in real life essentially start another task called Date and acknowledge the defaults as a whole. Place a DatePicker on the plan surface at the upper left-hand corner. The DatePicker went through similar changes with the presentation of Material Design as did the TimePicker. However long you utilize the most recent API and the default design it will show in a schedule format:

As is the situation with the TimePicker, assuming you run your application on a more seasoned pre-Lollipop variant of Android then you will see the Holo-themed spinner version:

You can opt to display the Holo spinner version in all Android versions by setting the datePickerMode to spinner.As an alternative you can also opt to also show a full Calendar in spinner mode by setting CalendarViewShown to true. The spinnerShown property can also be set to false to remove the spinners:



As well as playing with the manner in which the DatePicker looks, you can likewise set and get each of the pieces of a date using:
☐ DayOfMonth
☐ Month
☐ Year
You can likewise set and get greatest and

least dates that the gadget will
show utilizing properties.
When it comes to interfacing with the gadget you can set up a button to permit the client to mark the calendar similarly as with the TimePicker or you can utilize the OnDateChanged occasion to follow the worth.Doing this follows similar strides concerning the OnTimeChanged occasion however with a little contrast - there was no setOnDateChangedListener technique in early forms of Android.Rather there is an init technique which can be utilized to mark the calendar and the occasion controller. Anyway as the last boundary of init is SAM you can utilize a lambda and you can put the lambda outside of the capacity call enclosures i.e.

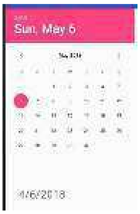
```
init(y,n,d){event handler}
```

For instance, on the off chance that you add a TextView to the lower part of the plan surface and the accompanying code for the occasion overseer then you can see the date change each time the client makes a change:

```
datePicker.init(2018, 4, 6)
{ datePicker, y, m, d - >
textView.text = m.toString() +“/” + d.toString() + “/” + y.toString() }
```

which sets the year, month and day and the

occasion
controller. Assuming you run the application you will see:

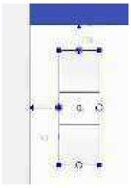


This tells you without a moment ’s delay, May is chosen and the date is 4/6/2018, that the months are numbered beginning with Jan at zero not 1. The arrangement is to add one to the month number.

Number Picker

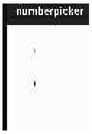
You may be stressed that the NumberPicker will be inconvenience when you notice that it is in the Advanced part of the tool kit range! Truly it is extremely simple to utilize, you should simply choose it and spot it on the plan surface. To give it a shot beginning another venture called Number and

acknowledge the defaults in general.
Assuming you disapprove of the delivering of the NumberPicker just disregard the mistakes and construct the undertaking. The mistakes should then go away.
If you run the app then you will see the NumberPicker styled for the latest API which in this case looks very odd because there is nothing loaded into thespinner:



The justification for this is that the NumberPicker is somewhat more muddled than different Pickers in that it permits you to set what the spinner displays.
There are two particular manners by which you can set the reach that is shown, as a couple of max/min esteems, or as the qualities put away in an array.
For instance, assuming you simply need the NumberPicker to show 0 to 9 you may use:

```
numberPicker.maxValue = 9
numberPicker.minValue = 0
```



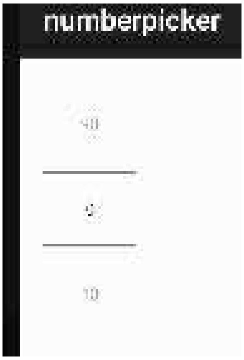
If you don ’t need the number spinner to fold over you can use: `numberPicker.wrapSelectorWheel=false`
If you want to give the user the choice of 0, 10, 20 and so on up to 90 you first have to initialize a string array of the correct size for these values. For this situation, the distinction among MaxValue and MinValue properties in addition to 1 gives the quantity of components in the list.

To make the cluster we want to utilize the Array constructor with a lambda that instates it:

```
val values=Array(10,{i-> (i*10).toString()})
```

Once we have the variety of qualities to show it tends to be relegated to the Picker utilizing its setDisplayedValues method:

```
numberPicker.maxValue=9 numberPicker.minValue=0
numberPicker.displayedValues=values
```

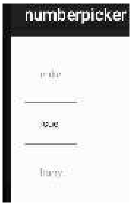


You might have seen that the exhibit used to determine the qualities is a String cluster. This means the NumberPicker, regardless of its name, can permit the client to pick from a rundown of discretionary strings that you can set.

For example:

```
val values=arrayOf( “mike”,”sue”,”harry”)
numberPicker.maxValue=2 numberPicker.minValue=0
numberPicker.displayedValues=values
```

produces:



When it comes to recovering the information you can utilize the getValue strategy,which consistently returns a number. This is either the record in the String cluster, or the genuine worth assuming you are not utilizing a String exhibit, of the thing the client picked.

If you need to get a live update of the worth the client chooses you can utilize the OnValueChange occasion.The occasion handler:

```
public void onValueChange(

NumberPicker picker,
int
oldVa
l, int
newV
al)
```

furnishes you with the NumberPicker object that the occasion happened on as picker and the list of the old and new qualities. The main issue is getting the qualities from the String cluster that was utilized at first. This is likely not open from the occasion controller and may not exist any more drawn out. The arrangement is to utilize the NumberPicker’s getDisplayedValues, which returns a String exhibit of qualities as of now stacked in the NumberPicker.

For instance to move the worth to a TextView

```
numberPicker.setOnValueChangedListener { numberPicker, old, new
- >
val values =numberPicker.displayedValues textView.text=values[new]
}
```

This uses the picker to get the variety of showed values, which it then, at that point, moves to the TextView utilizing the newVal as an index. Truth be told given that qualities is as of now characterized and liable to be open to the lambda through conclusion we could write:

```
numberPicker.setOnValueChangedListener { numberPicker, old, new
- >


textView.text=values[new]  }
```

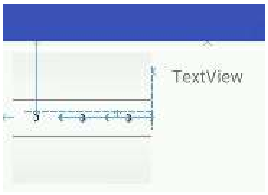
Notice anyway that this would come up short assuming the rundown of things in the NumberPicker had changed.

Now when you run the program the TextView is refreshed when any progressions are made:

That’s pretty much everything to fundamental utilization of the NumberPicker.

Multi-Digit Input

If you want to create a multi-digit input –for hundreds, tens, units, say–then simply use three NumberPickers. This is more interesting than it initially shows up to powerfully follow the current worth across more than one NumberPicker. For instance, to construct a three-digit input you first need to put three NumberPickers on the plan surface with ids numberPicker1, numberPicker2 and numberPicker3.Place numberPicker1onthefarleft, then numberPicker 2 and thennumberPicker3. Additionally add a TextView some place convenient:



You could instate every one of the NumberPickers thusly, however it is enlightening to utilize a variety of NumberPickers to do the job:

```
val nps=arrayOf(numberPicker1,numberPicker2,numberPicker3)
```

Note that you can make a variety of object of any sort utilizing arrayOf. Now we have a variety of NumberPicker objects we can introduce them all similarly, yet first we want the variety of values:

```
val values=Array(10,{i->i.toString()})
```

As we are utilizing 0 to 9 this should be possible as a list without utilizing an exhibit, yet this makes the model more broad. Presently we have the variety of qualities we can instate the NumberPickers:

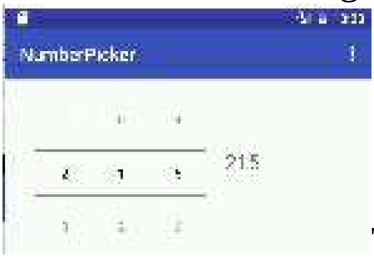
```
for( I in nps.indices){

nps[i].maxValue=values.size-1
nps[i].minValue=0
nps[i].displayedValues=values
nps[i].setOnValueChangedListener(onValueCh

anged)
}
```

Notice that a similar occasion overseer is utilized for all of the NumberPickers. Sometimes this is the method for getting things done, in others it is smarter to have an occasion overseer for each widget.

The following little issue is the manner by which to refresh the worth showed in a TextView when one of the NumberPickers changes its worth. Again the easiest answer for a model is to get the qualities from every one of the NumberPickers utilizing a for loop:

```
val onValueChanged= {picker:NumberPicker,old:Int,new:Int-> var


temp=““
for(i in nps.indices){
temp+=values[nps[i].value]
}
textView.text=temp


}
```

For this situation we don’t utilize any of the occasion technique’s boundaries we just utilize the qualities exhibit which is accessible because of conclusion and query the qualities that each NumberPicker is as of now set to.

If you run the program, you ought to have the option to adjust what is shown in the TextView in a reasonable three-digit place esteem way: 

The total program is:

```
import android.os.Bundle
import
android.support.design.widget.Snackbar import
android.support.v7.app.AppCompatActivity import android.view.Menu
import
android.view.MenuItem import
android.widget.NumberPicker

import
kotlinx.android.synthetic.main.activity_main.* import
kotlinx.android.synthetic.main.content_main.*

class MainActivity : AppCompatActivity() {
supersede fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

val values=Array(10,{i->i.toString()})
val nps=arrayOf(numberPicker1,numberPicker2,numberPicker3)
val onValueChanged= {picker:NumberPicker,old:Int,new:Int-> var temp=““
for(i in

nps.indices){
temp+=values
[nps[i].value]

}
textView.text=temp
}
for( I in
nps.indices){
nps[i].maxValue=values.size-1
nps[i].minValue=0
nps[i].displayedValues=values
nps[i].setOnValueChangedListener(onValueChanged)
}
```

There is considerably more to say about the Pickers and how to modify them, yet the strategies clarified here are the most well-known, and customization by and large just includes the utilization of genuinely clear properties and methods.
The greatest exclusion here is the utilization of the Pickers inside a DialogFragment. This is a major theme that is shrouded in Android Programming: Mastering Fragments and Dialogs.

Summary

☐ TimePicker, DatePicker and NumberPicker give simple to utilize methods of getting client input without utilizing an on-screen keyboard.

☐ Pickers are generally presented as exchange boxes or discourse parts. This is a decent method for utilizing them, however they are additionally helpful in the event that you don’t utilize a dialog.

☐ With the presentation of Material Design with Android Lollipop, the appearance of the Pickers has changed.For simple in reverse similarity use similarity library v7,appcompat,which is remembered for your activities naturally by Android Studio, and stick with the AppCompat themes.

☐ If you stay with these defaults your application will utilize Material Design on Lollipop and later, yet the Holo subject for prior renditions of Android. Assuming that you stray from AppCompat topics things tend not to work.

☐ It is conceivable and exceptionally simple to supplant the Material Design with the Holo look on all forms of Android. Essentially set the timePickerMode or potentially datePickerMode properties to spinner. The NumberPicker consistently shows as a spinner.

☐ Use the OnXChangedListener occasion overseer to react to client input.
☐ The DatePicker in spinner structure can likewise show a schedule utilizing the CalendarViewShown property.

Chapter 16 ListView

ListView is presumably the most regularly utilized UI part in an Android application. It

is easy to utilize, however you really want to will grasps with the possibility of an“connector”, and getting what is happening pays dividends.
For a scope of reasons, one of the most widely recognized things you really want to do in an Android UI is to show a rundown of things that the client can choose from. We have as of now checked out the essential Picker, however the ListView is a more broad classification of“picker”. If you know about work area advancement then you likely consider records that the client can choose from as being like a drop-down list. Android and versatile gadgets frequently need more than simply a little list. Because of the restricted screen size, it is generally expected the situation that the client must be shown an outline of many various things. At the point when the client chooses a thing they are then given more subtleties of the thing, generally called a subtleties view.
Displaying records and different assortments of information is normal that Android has an instrument that makes it more straightforward once you see how it all works.

The key thought is that to show an assortment, every thing in the assortment has some way or another to be changed over to a suitable View object.It is the View object that the compartment shows for every thing of information in the collection.

Understanding the Adapter

Displaying an assortment of things has various similitudes regardless the assortment of things are, for sure holder is utilized. The compartment has different places that are noticeable on the screen for showing things. For instance, a ListView has level openings, one for each thing, and a GridView has a 2D matrix of spaces. Every compartment acknowledges a View article and shows it in a space. For instance, you could give the ListView TextView articles and it would basically show text in every one of its slots. You may be asking why not simply supply a bunch of Strings to the ListView and let it work out how to show the Strings as text? Things could be coordinated along these lines and it would be easier, however provided that you needed to

show Strings. To show a rundown of pictures, say, then, at that point, you would require a ListView that got pictures thus on.
It is considerably more adaptable to give the holder a bunch of arranged View objects since then the compartment basically needs to show the View object without playing out any conversions.
All holders that get from the AdapterView class utilize connectors to supply what they show as far as View objects. Just as the ListView and GridView, they incorporate the Spinner, Gallery and StackView. We have as of now took a gander at the utilization of the Spinner and its related connector, so this time it is the turn of a further developed UI component– the ListView and its adapter.
This approach also has the advantage that you can provide the container with a View object that is a complete layout, i.e. it could itself be

a container with lots of View objects to display. For example, you could supply a ListView with a View object that contains an image, and a TextView to create a multimedia list of pictures and text. Obviously the compartment isn’t doing the transformation from the information object to the View object – you must do it. This is the place where the connector comes in. The ListView and the GridView compartments both utilize the ListAdapter class as their fundamental adapter.

Extending the ListAdapter Class

Overall you need to take the ListAdapter class and extend it to make your own custom Adapter which produces a custom View object for the holder to use in every one of its spaces. The compartment requests the View object to be utilized to show thing I or utilizing whatever ordering suits the specific holder. The Adapter returns the View object and the holder shows it– without stressing what it is for sure it compares to. This may sound muddled, yet it ends up being extremely straightforward in practice. However, to make things significantly less complex there is additionally an ArrayAdapter which allows you to show a solitary text thing for every component of a variety of discretionary items. How can this possibly work if the object in the array can be anything? The first thing to point out is that ArrayAdapter is a generic class and can accept an array of any type as long as you specify it when you create the ArrayAdapter instance. The second highlight note is that Array connector calls every thing's toString technique to get some text to show in the holder, which is exceptionally basic yet in addition extremely prohibitive. Truth be told, it is very simple to alter what the ArrayAdapter presentations, and this makes it more flexible than you may expect and thus certainly worth getting to know.

Before continuing a speedy outline is helpful:

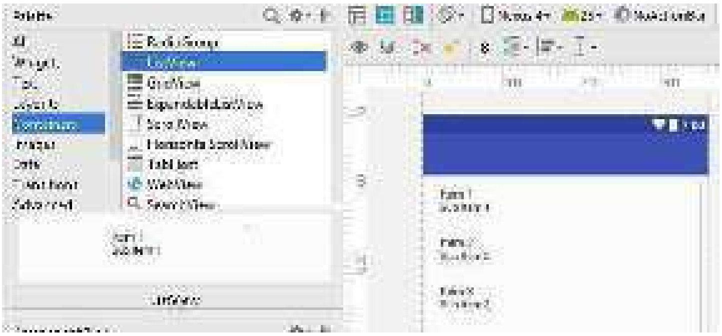
☐ Containers like ListView and GridView show View objects in a specific plan– as an upward rundown or as a 2D framework in these two cases respectively.

☐ An Adapter changes over the information that will be shown in each space into a reasonable View object.
☐ For complex information and showcases you really want to make a custom Adapter.
☐ In many cases the ArrayAdapter, a predefined custom Adapter for changing Arrays over to TextView objects, can be used.
☐ The ArrayAdapter, at its generally essential, can supply TextView objects to a holder from a variety of any kind basically by calling the item’s toString methods.

Using the ArrayAdapter

Rather then, at that point, beginning with a

model that is totally broad, it merits taking a gander at how the ArrayAdapter is utilized related to a ListView.
Start another Android Studio project called ListViewExample dependent on a Basic Activity and acknowledge every one of the defaults. For a basic model all we will do is show a rundown of names in a ListView.
First delete the usual“Hello World” textView. In the Layout Editor scroll down the Palette until you can see the Containers and place a ListView on the design surface. At the moment the listView isn’t always automatically assigned an id. If this is the case type listView into the id in the Attributes window. If you do this then a dummy content will be generated for you:



You will see that, for the sake of allowing you to work with the layout, the Layout Editor shows you the ListView filled with some two-item text objects.

Our Adapter is going to be simpler than this dummy display with just a single line of text. If you run the program at this early stage you won’t see anything in the ListView– it will be blank. The text that you see in the Layout Editor is just to help you visualize the UI – there is no Adapter associated with the ListView and hence when you run it there is nothing to display.

Our next task is to create an ArrayAdapter object to supply the ListView with something to display. First, however, we need a String array to hold the text we are going to display. For simplicity we might as well add the code to the onCreate method.

To create a simple String array we can use:

```
val myStringArray = arrayOf( “A”, “B”, “C”)
```

Feel free to think up something more creative than A, B, C. In the real world the Strings would probably be read from a file or a database, etc. Now we can create the

ArrayAdapter.To do this the constructor needs

the context, usually this, and a layout to use to display each String and the String array:

```
val myAdapter= ArrayAdapter<String>(
this,
andRoid.R.layout.simple_list _item_1, myStringArray)
```

Notice the way that the type of the array is specified as <String>. If you are not familiar with generics then you need to look up how it all works. Also notice the use of the standard supplied layout simple_list_item1. You can create your own layouts and we will see how this is done in a moment.
Finally we need to associate the adapter with the ListView:

```
listView.setAdapter(myAdapter)
```

The complete onCreate method is:

```
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)


setContentView(R.layout.activity_main)
setSupportActionBar(toolbar)

val myStringArray = arrayOf( “A”, “B”,
“C”) val myAdapter =
ArrayAdapter<String>(

this,
andRoid.R.layout.simple_list _item_1, myStringArray)

listView.setAdapter(myAdapter)
}
```

You will also have to remember to add import statements for each of the classes used– ListView and ArrayAdapter– easily done with Alt+Enter. If you now run the program you will see a neat list with each array element displayed on a line:

So far this doesn ’t look impressive but the ListView gives you some basic facilities. For example, if you increase the number of elements in the array: `var myStringArray =`

`arrayOf( “A”, “B”, “C”, “D”, “E”, “F”, “G”, “H”, “I”, “J”)` you will discover that you can automatically scroll through the list using the usual flick gesture.

Working with the

Data

The whole point of showing the user a list of items is so that they can interact with it.You can manipulate the data on display in various ways and handle events when the user selects an item.

Get Selection

Perhaps the most important thing is to deal with the user selecting an item. The usual way of doing this is to write a handler for the OnItemClickListener, which passes four parameters:

```
onItemClick(AdapterView parent,View view,int position, long id)
```

The AdapterView is the complete View displayed by the container, the View is the View object the user selected, the position is the position in

the collection, and the id is the item’s id number in the container. For an ArrayAdapter the id is the same as the array index.

You can use this event to find out what the user has selected and modify it. For example the event handler:

```
listView.setOnItemClickListener({parent, view,


position, id -> (view as
TextView).text="selected"


})
```

sets each item the user selects to“selected”.

It is important to know that changing what the View object displays doesn’t change the data stored in the associated data structure. That is, in this case setting a row to“selected” doesn’t change the entry in the String array. You can also set the selection in code using:

```
listView.setSelection(position)
```

where position is the zero-based position of the item in the list, and you can scroll to show any item using:

```
listView.smoothScrollToPosition(position)
```

A subtle point worth mentioning is that you can’t make use of the View object that is passed to the event handler to display the selection in another part of the layout. A View object can only be in the layout hierarchy once. In most cases this isn’t a problem because you can usually manually clone the View object. For example, in this case the View object is a TextView and so you can create a new TextView and set its Text property to be the same as the one in the list:

```
val w = TextView(applicationContext)
w.text = (view as
TextView).text
```

This can be more of a nuisance if the View object is more complex.

Changing the Data

One of the slightly confusing things about using adapters is the relationship between what is displayed and what is in the underlying data structure. You can change the data, but if you want to see the change in the container you have to use an adapter notify method to tell the adapter that the data has changed.

For example, if you change an element of the array:

```
myStringArray[0]="newdata"
```

then nothing will show until you use:

```
(listView.adapter as ArrayAdapter<String>).notifyDataSetChanged()
```

Notice that you have to cast the ListAdapter in adapter to an ArrayAdapter<String> to call the notify method.

There is a second way to change the data using the ArrayAdapter itself. This provides a number of methods to add, insert, clear, remove and even sort the

data in the adapter. The big problem is that if you use any of these then the underlying data structure associated with the adapter has to support them. For example, the add method adds an object onto the end of the data structure but, with the program as currently set up, if you try: `myAdapter.add(“new data”)`
the result will be a runtime crash. The reason is that in an array has a fixed size and the add method tries to add the item to the end of the array, which isn’t possible.
If you want to add items to the end of an array-like data structure, you need to use a List and not just a simple array– we met this idea before in Chapter 14 in connection with

Spinners. A List can increase and decrease its size. For example we can create a List from our existing String array: val myList=mutableListOf(*myStringArray)

and you can associate this new List with the adapter instead of the String array:

```
val myAdapter = ArrayAdapter<String>(

this,
andRoid.R.layout.simple_list_ite m_1, myList)
```

Following this you can use:

```
myAdapter.add(“new data”)
```

and you will see the new data at the end of the displayed list. You may have to scroll to see it.

As long as you are using a List you are safe to use all of the adapter data modifying methods:

```
add(item)
addAll(item1,item2,item3...)
clear() //remove all
data
insert(item,position)
remove(item)
```

You can also make use of:

```
count() // number of elements

getItem(position) // get item
```

```
getItemId(position) //get item id
getPosition(item)
```

A Custom Layout

So far we have just made use of the system provided layout for the row. It is very easy to create your own layout file and set it so that it is used to render each row, but you need to keep in mind that the only data that will be displayed that is different on each row is derived from the item’s toString method.

The simplest custom layout has to have just a single TextView widget which is used for each line. In fact this is so simple it has no advantage over the system supplied layout so this is really just to show how things work.

Use Android Studio to create a new layout in the standard layout directory and call it mylayout.xml. Use the Layout Editor or text editor to create a layout with just a single TextView object. Create a new layout and accept any layout type for the initial file. You can then place a TextView on the design surface.You won’t be able to delete the layout,however,as the editor will not allow you to do it. Instead you need to switch to Text view and edit the file to remove the layout:

```
<?xml version=“1.0” encoding=“utf-8”?>

<TextView
xmlns:android="http://schemas.android.com/apk/res/android"
android:text="TextView"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:id="@+id/textView" />
```

Notice that you need the xmlns attribute to make sure that the android namespace is defined.

To use the layout you simply provide its resource id in the ArrayAdapter constructor:

```
val myAdapter = ArrayAdapter<String>(

this,
R.layout.mylay
out, myStringArray)
```

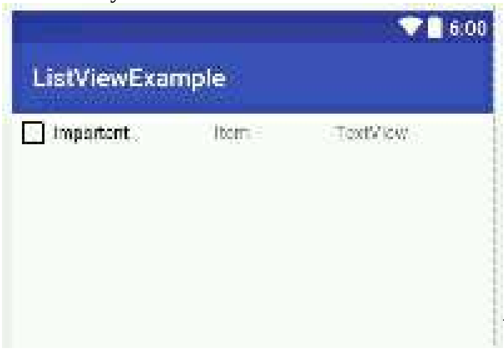
If you tRy this you won ’t see any huge difference between this and when you use the system layout android.R.layout.simple_list_item_1.

The next level up is to use a layout that has more than just a single TextView in it. The only complication in this case is that you have to provide not only the id of the layout but the id of the TextView in the layout that you want to use for the data. For example, create a layout with a horizontal LinearLayout and place a CheckBox, and two TextViews. The simplest way to do this is to place the LinearLayout in the default ConstraintLayout and use the Layout Editor to design the layout. Then start a new layout resource and copy and

past the LinearLayout XML tag and all it contains as the base layout in the new file:

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

xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="368dp"
android:layout_height="wrap_cont ent"
android:descendantFocusability="blocksDescendants"
android:orientation="horizontal"
tools:layout_editor_absoluteX="0dp" tools:layout_editor_absoluteY="25
dp">

<CheckBox
android:id="@+id/checkBox2" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:layout_weight="1"
android:text="CheckBox" />
```

```
<TextView
android:id="@+id/textView"
android:layout_width="wrap_content"
android:layout_height="wrap_content" android:layout_weight="1"
android:text="TextView" />
android:text="TextView" />

<TextView
android:id="@+id/textView2"
android:layout_width="wrap_content"
android:layout_height="wrap_content" android:layout_weight="1"

</LinearLayout>
```



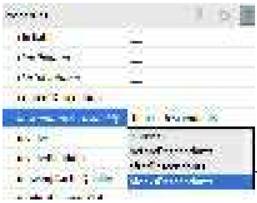
You can use this layout by creating the ArrayAdapter with:

```
val myAdapter = ArrayAdapter<String>(
this,

R.layout.mylay out,
R.id.textView 2,
myStringArray )
```

assuming that the TextView you want the data to appear in is textView2. The resulting ListView example is a little more impressive than the previous example

Notice that each of the View objects in the layout gives rise to a distinct instance per line. That is, your layout may only have had one CheckBox but the ListView has one per line. This means that when the user selects the line you can retrieve the setting of the CheckBox, say. It also means that a ListView can generate several View objects very quickly and this can be a drain on the system. There are a few things that you need to know if you are going to successfully handle onItemClick events. The first is that your layout can't have any focusable or clickable Views. If it does then the event isn't raised and the handler just isn't called. The solution is to stop any View object in the container from being focusable by adding:

android:descendantFocusability= "blocksDescendants" to the LinearLayout, or use the Property window to set it to blocksDescendants:

With this change the event handler should be called, but now you need to keep in mind that the View object passed as view in:

```
listView.setOnItemClickListener({ parent, view, position, id ->
```

is the complete View object for the row and not just the TextView. That is, in the case of the example above it would be the LinearLayout plus all of its children.
If you are going to work with the View object, you have to access the objects it contains and you can do this is in the usual way.
For example:

```
listView.setOnItemClickListener({ parent, view, position,

id->
view.findViewById<TextView>(R.id.textVi ew).text="Selected"

})
```

Notice that you can use findViewById in the View that is returned.

A Custom ArrayAdapter

If you only want to display a String in each row you can use the standard ArrayAdapter and the object’s toString method.You can even customize the object’s toString method to display something different from the default, but it is still just a String.
If you have an array of general objects and want to display multiple items from each object, or items which are not Strings, then you need to create a custom ArrayAdapter. This isn’t difficult, although there are one or two more advanced points to take note of.
For this first example, let’s keep it as simple as possible. First we need some objects to hold the data we are going to display– a record type, say, with a field for name and one for number in stock. You could also add a photo of the item, but in the spirit of keeping it simple a String and int are enough.
If you know other languages you might be

thinking that we need a struct or something similar. In Java or Kotlin there are no structs. If you want to create a record you create an object with the necessary properties. However Kotlin supports data classes which are designed for the task and in many ways are a superior “record” type to what you find in other language.
Start a new project called CustomList and accept all the defaults. Remove the default text and add a ListView.
In Java it is a rule that every new class you create has to be in a separate folder. You can follow this convention in Kotlin but it is often easier to add utility classes to the file you are working in. You can always separate them at a later date.

We are going to create a data class complete with properties that we can use to store data:

```
data class MyData (var myTitle:String, var myNum:Int)
```

The new class has two public fields, myTitle and myNum, and a primary constructor allowing us to initialize these fields.
For example, in the onCreate method you can

add:

```
val myDataArray=arrayOf(
MyData("item1", 10),
MyData("item2", 20),
MyData("item3", 30)

)
```

You might need to add more data than this to try out the ListView properly but this at least gets us started.

Now we have some data to display we need to add the custom adapter. Once again we can create a new class called MyAdapter in the same file. MyAdapter has to inherit from ArrayAdapter:

```
class MyAdaptor( val mycontext: Context?,
val resource: Int,
val objects: Array<out MyData>?) :


ArrayAdapter<MyData>(mycontext, resource, objects) {
```

This looks complicated but it is the code you get using Alt+Enter on the partially declared class. The <out MyData> is the compiler protecting you

from an unlikely problem but it is still worth playing by its rules. Notice also that we have to change the generated context parameter to mycontext to avoid a name clash.

Notice that we want the generic ArrayAdapter to work with MyData objects. Most of the methods of ArrayAdapter will work perfectly well with arrays of arbitrary objects.
The primary constructor that we have just defined will also automatically create read only properties for each of the parameters that start val.This is another of Kotlin’s simplifications and we don’t need to define a more elaborate constructor.
Notice that with this constructor our adapter is used in the same way as in the previous example, that is we supply context, resource id, and the array of data. These values are also passed on to the constructor of the super class i.e. ArrayAdaptor.
We now reach the key part of the customization, overriding the adapter’s getView method. This is the core of the functionality of the adapter. Each time the ListView needs to display a new row, it calls the adapter’s getView method and expects to get back a View object that it can display as

the row.

To override the method you can use Android Studio to generate some code. Right-click in the adapter class and select Generate, Override method and then select getView. The generated code isn't particularly helpful, but at least it gets the method signature correct:

```
override fun getView( position: Int,
convertView: View?,
parent: ViewGroup?): View { return

super.getView(position, convertView, parent)
}
```

It really doesn 't matter how getView generates the View object it is going to return, but the most common way of doing the job is to inflate a layout file. To give the inflater something to work with, right-click on the res/layout folder and select New,Layout file. Call the file mylayout, change the LinearLayout to horizontal, and add two TextViews with ids, title and number.Feel free to change the layout to make things look pretty - it won't change the code you need to write:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout

xmlns:android="http://schemas.android.com/apk/res/andr oid"
xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools"
android:layout_width="368dp"
android:layout_height="wrap_c ontent"
android:descendantFocusability="blocksDescendants"
android:orientation="horizontal"
tools:layout_editor_absoluteX="0 dp"
tools:layout_editor_absoluteY= "25dp"
layout_height="wrap_content" layout_width="match_parent">

<TextView android:id="@+id/title"
android:layout_height="wrap_content" android:layout_weight="1"
android:layout_width="wrap_content" android:text="TextView" />

<TextView
android:id="@+id/number"
android:layout_height="wrap_content"
android:layout_weight="1"
android:layout_width="wrap_content"
android:text="TextView" />


</LinearLayout>
```

Our first task is to get an inflater and inflate the layout file:

```
val inflater=(mycontext as
Activity).layoutInflater val
row=inflater.inflate(resource, parent,false)
```

Notice that we make use of the resource id we stored when the constructor ran and we use the parent View object passed in to the getView method. The only purpose the parent

View object serves is to allow the system to lay out the resource in a known container. The final false parameter tells the inflater not to add the resource generated object to the parent – this is a job for the ListView. Before this happens we have to put the data into the View object. To do this we need to find the two TextView objects that we placed into the layout and this is just a matter of using the familiar findViewById pattern:

```
val
title=row.findViewById<TextView
>(R.id.title) val
number=row.findViewById<TextView>(R.id.numbe
r)
```

Once you have the View objects you need to change, you can use the position parameter to get the data from the array of objects that was set by the constructor:

```
title.text= objects?.get(position)?.myTitle number.text=
objects?.get(position)?.myNum.toString()
```

All we need to do now is return the row View object:

```
return row
}
```

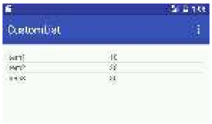
The complete myAdapter class is:

```
class MyAdaptor(val mycontext: Context?,
val resource: Int,
val objects: Array<out MyData>?) :
ArrayAdapter<MyData>(mycontext, resource, objects) { override fun
getView(position: Int,
convertView: View?, parent:

ViewGroup?): View { val inflater=(mycontext as
Activity).layoutInflater val
row=inflater.inflate(resourc e,parent,false) val
title=row.findViewById<TextVi ew>(R.id.title) val
number=row.findViewById<TextView>(R.id.number ) title.text=
objects?.get(position)?.myTitle number.text=
objects?.get(position)?.myNum.toString() return row
}


}
```

Now all we have to do is write some code that makes use of the new class and this is exactly the same as the code that made use of the standard ListView:

```
val myAdapter=MyAdaptor(this,R.layout.mylayout,myDataArray)
listView.adapter=myAdapter
```

Don’t forget to put a ListView component on the main layout:



If you run the program you will now see a list consisting of two TextViews, each with something different to display on each line. In a real app you probably wouldn’t create a new class for two text items, overriding the toString method would be easier,but the principles are the same no matter what the multiple View objects created by the adapter are.

Reuse, Caching and General Layouts

We have a working custom adapter class but there are some things we can do to make it better. The first relates to efficiency. If you recall, it was pointed out that a big list of objects could result in the creation of a lot of View objects.In practice,however,we really only need the number of View objects that correspond to rows actually being displayed on the screen.

To avoid having to dispose of and create new View objects all the time, the ListView gives you the opportunity to recycle the View objects you have already created. This is what the convertView parameter in the getView method is all about. If it is null you have to inflate and create a new View object. If it is non-null then it is a View object ready to be used and you don’t have to create a new one.

Modifying the previous example to make use of convertView is easy:

```
val row:View
if(convertView==null){
val inflater=(mycontext as Activity).layoutInflater
row=inflater.inflate(resource,pare
nt,false)
}else{

} row=convertView
```

Notice that this is null safe as the compiler checks that row gets a value one way or another.

This is a speed-up worth making and saves having to create lots of View objects and dispose of them. However, we are still looking up the child View objects every time we want to change the data:

```
val
title=row.findViewById<TextView
>(R.id.title) val
number=row.findViewById<TextView>(R.id.numbe
r)
```

This is also very wasteful and can be avoided with the application of the ViewHolder pattern. All we have to do is save the references to the children in an object,and store this in the parent View’s tag property. Then the next time we see the parent View we don’t have to find the children we are looking for - they are stored in the parent’s tag as a ViewHolder object.

First we need to create a ViewHolder data class:

```
data class ViewHolder(var title:TextView,var number:TextView)
```

Notice that this has fields capable of holding references to our two TextView objects. The logic of the getView method is now to also create and store a ViewHolder object if we are not recycling a View object:

```
val row: View
if (convertView == null) {


val inflater = (mycontext as
Activity).layoutInflater row =
inflater.inflate(resource,
parent, false)
val viewHolder =ViewHolder (

row.findViewById<TextVi
ew>(R.id.title),
row.findViewById<TextView>(R.id.number
)) row.tag=viewHolder


}
```

Notice that we have stored the references to the TextViews in the viewHolder and stored this in the row’s Tag field– this is what tag

fields are generally used for.
If we do have a recycled View object, we need to get the viewHolder object:

```
} else {
row = convertView
}
```

Finally, no matter where the viewHolder object came from, we just use it: (row.tag as ViewHolder).title.text = objects?.get(position)?.myTitle (row.tag as

```
ViewHolder).number.text =
objects?.get(position)?.myNum.toString()
return row
}
```

With this change we have avoided creating a View object each time and we have avoided having to look up the child objects each time, a very useful saving in time and resources.

Finally there is one last embellishment to apply. At the moment the layout for the row has to have the ids of the TextView objects set to title and number. It is much better to let the user set these in the constructor:

```
class MyAdaptor( val mycontext: Context?,

val resource: Int,
val resTitle:Int, val
```

resNumber:Int,
val objects: Array<out MyData>?) :

ArrayAdapter<MyData>(mycontext, resource, objects) { This constructor has two extra parameters used to specify the id numbers of the two ViewText objects in the layout. These are automatically created properties of the class.
Finally we need to change the getView method to use the two new private variables:

val viewHolder =ViewHolder (
row.findViewById<TextView>(res Title),
row.findViewById<TextView>(re sNumber))

With these changes the adapter can be used as:
val myAdapter = MyAdaptor(this,

R.layout.mylayou
t, R.id.title,
R.id.number,
myDataArray)

and the user is free to use any ids for the layout as long are there are two TextViews.
There are aspects of the ListView that haven’t been covered. In particular, what do you do if your data structure isn’t an array? The ArrayAdapter can handle Lists by simply overriding the appropriate

constructor.Anything more complex and you will have to create a custom Adapter. In most cases, however, this isn’t necessary because most data structures can be mapped onto an array.
There are a range of formatting topics not covered,the header and separator for example, but these are relatively easy. More complicated are the multi- selection options and any custom list displays such as a list with multiple columns or pop-out detail windows.

Custom Adapter

Putting together all the code introduced so far gives:

```
class MyAdaptor(
val mycontext: Context?,
val resource: Int,
val resTitle: Int, val
resNumber: Int,
val objects: Array<out MyData>?):

ArrayAdapter<MyData>(mycontext, resource, objects){ override fun
getView(
position: Int,
convertView: View?,
parent: ViewGroup?): View { val
row: View
if (convertView == null) { val inflater = (mycontext as
Activity).layoutInflater row =
inflater.inflate(resource,
parent, false)
val viewHolder = ViewHolder(
row.findViewById<TextView >(resTitle),
row.findViewById<TextVie w>(resNumber))
row.tag = viewHolder

} else {
row = convertView
}

(row.tag as ViewHolder).title.text = objects?.get(position)?.myTitle (row.tag
as ViewHolder).number.text =
objects?.get(position)?.myNum.toString()
return row
}
}

data class ViewHolder(var title: TextView, var number: TextView)
```

Summary

☐ Containers like ListView work together with an adapter to display data. ☐ The adapter accepts a data structure– usually an array or a list– and converts any data item into a View object that represents the data. ☐ You can handle a selection event to find out what the user has selected.
☐ The View object can be complex with an

outer container and any number of child objects.
☐ The basic ArrayAdapter uses each object’s toString method to provide the data to display in a ViewText object.

☐ If you want to display something other than the result of calling toString, you need to implement a custom ArrayAdapter. To do this you have to override the inherited getView method.

☐ The ListView is clever enough to provide you with View objects to recycle– although you don’t have to if you don’t want to. ☐ The ViewHolder pattern can make your use of nested View objects more efficient.

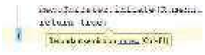
Chapter 17 Android The Kotlin Way

After seeing Kotlin at work making your Android code shorter and, more importantly, clearer, it is time to gather the ideas together. This chapter is a collection of the ways Kotlin makes Android easier in comparison with Java. If you are not a Java programmer many of these differences will not seem impressive. This chapter is at the end of the book because it can serve as a reminder to a Java programmer learning Android what a difference Kotlin makes. It could also be read first by a fairly experienced Android programmer wanting to know what their Android Java idioms look like in Kotlin.
If you want a good grounding in Kotlin I can do no better than recommend my own Programmer's Guide To Kotlin ISBN 978-1871962536.This chapter is more about how Kotlin affects your approach to Android programming than a general Kotlin tutorial.

What You No Longer Have To

Type

☐ Semicolons, type specifiers and new
The first joy of Kotlin is that you no longer have to type a semicolon to mark the end of each line. You can type a semicolon if you want to and Android Studio doesn't mark it as an error, but it does suggest that you might want to remove it:



```
[illegible]
return true;
```

[illegible]

In Kotlin the end of a line is the end of a line and you don't need a special additional symbol to mark it. However, if you include more than one statement on a line, you do need to indicate where each one ends by placing a semicolon between them.

If you are a long time Java programmer you might well find it difficult to stop typing semicolons– eventually you will.

The second obvious simplification is that in most cases you no longer have to specify a type when it is obvious.
For example in place of:
```
String myString=new String();
```
you can now write
```
var myString=String()
```

Java beginners have long been confused by the need to write String… new String and now we don’t have to. If you do need to specify a type you do it as a trailing qualifier:
```
var myString:String=String()
```

This example also highlights the third simplification – you no longer write new in front of a constructor. After all new signified that the function call was a constructor, but what is and what is not a constructor is fairly obvious from the class declarations.

var & val

Of course, one new thing is that you have to use either var or val when you declare a variable.

If you write:

```
var myVariable=1
```

then you will get a true variable that you can read and
write. If you use:

```
val myVariable=1
```

then you get a read-only variable which you cannot use on the left of an assignment.

In most cases you should use var for simple types and val for objects. A more general principle,however,is to always use val unless you are forced to use var by the nature of the algorithm.Notice that all val promises is that you cannot assign to the variable– you can modify the properties of any object that it references.This is read-only rather than immutable.

No More get & set

Kotlin classes have properties that come complete with get and set mutator functions.Properties declared using var have default getters and setters and those declared using val have only a getter i.e.val properties are readonly.

All properties are accessed via getter and setter functions and, unlike in Java, you don’t have to explicitly create them– the compiler will do the job for you.

It will also automatically put get or set in front of the property ’s name so that you can use Java properties implemented in this way without having to modify property names. In Java you might have a setText and a getText method.You can still use these in Kotlin:

```
var
myString=view.getText()
view.setText()=myString
```

But in many cases you can simply write:

```
var
myString=view.text()
view.text=myString
```

In other words, the mutator methods have been converted into properties with the same name, but with a lower case first letter.

Kotlin can also deal with Booleans that are named starting with is. For example, instead of:

```
button.isShown()
```

you can write:

```
button.isShown
```

In most cases, when you see a set or get you can simply use the equivalent property, but not always. There are many odd problems that can crop up to stop the transformation from getters/setters to properties. Consider the well known and used:

```
setOnClickListener
```

If this was converted to onClickListener this would confuse the issue with the OnClickListener interface.In this case you have to use the setOnClickListener method but as it accepts a SAM– an interface that defines a Single Abstract Method it can be written using a lambda:

```
button.setOnClickListener { view -> instructions}
```

At the moment the best advice is to try to use any get/set methods as properties and see what Android Studio supplies as an auto-complete. The range and quality of auto-complete features is likely to improve as Kotlin support is developed.

View Objects As Properties

In Java finding a View object uses the findViewById method, and this is still needed in Kotlin in many cases. It is even easier to use in Kotlin because it is implemented as a generic extension function and you can avoid having to cast to the correct type:

```
val button = findViewById<Button>(R.id.my_button)
```

However, if you are using Kotlin to work with the XML file it automatically converts all of the string labels on the ids to Activity properties and then makes them reference the objects that the inflater creates. You can then specify which layout files you want to create properties for using:

import kotlinx.android.synthetic.main. *layout*.* As you enter the id of a View object defined in the XML file Android

Studio will ask if you want to import the “synthetic” definition it has created.

So.to import properties for all of the View created by the two standard XML files main.activity_main.xml and main.content_main.xml you would use:

```
import
kotlinx.android.synthetic.main.activity_main.* import
kotlinx.android.synthetic.main.content_main.*
```

What this means is that you can simply use variables with the same name as the id string assigned to the View object so, for example, instead of having to use findViewById for the R.id.my_button object you can simply use the button property as if you had executed:

```
val button = findViewById<Button>(R.id.my_button)
```

Event Handlers

With the whole of Chapter 4 about event handling, only a brief summary is given here.

In Java event handlers are methods that belong to event listener objects often defined using an interface. In many cases there is a single event handler defined in a single interface and this is a SAM– Single Abstract Event.
For example the View.OnClickListener interface is defined as:

```
public interface OnClickListener { void
onClick(View var1);
}
```

and you need to create an object which implements this interface to pass to the setOnClickListener method. The Kotlin compiler accepts a lambda expression, an anonymous local function or a function reference of the correct type and compiles it to an object that implements the interface with

the function.Notice that the setOnClickListener is passed an object and not a lambda or a function.
For example:

button.setOnClickListener {view -> button.text=”Clicked”} Using a lambda is the most common way of creating an instance of the event listener object for a SAM.

There are instances where the event listener object isn’t a SAM . It might not even be defined as an interface. Some event listeners are defined as classes with a mix of virtual and implemented methods. The implemented methods are often utility functions that allow you to do things such as cancel event handling or modify it in some way. Even when event listeners are interfaces, they can define multiple related event handlers and so do not qualify as a SAM.

For example, in Chapter 10 we meet the ActionMode.Callback which is an interface with four event handling methods defined. In this case the simplest solution is to use a local object that implements the interface: val mycallback=object : ActionMode.Callback { override fun

onActionItemClicked(
mode: ActionMode?, item: MenuItem?): Boolean { TODO(“not implemented”)
}

```
override fun onCreateActionMode(
mode: ActionMode?, menu: Menu?): Boolean { TODO(“not implemented”)
}

override fun onPrepareActionMode(
mode: ActionMode?, menu: Menu?): Boolean { TODO(“not implemented”)
}
override fun onDestroyActionMode( mode:
ActionMode?) {
TODO(“not implemented”)
}


}
```

The same approach works if the event listener is defined as a class with

some implemented methods and some virtual methods. Simply create an object that implements the class and all of the virtual methods.

Data

Kotlin essentially reuses Java’s data types including arrays and strings, but it augments them and has its own approach to them. However, essentially a Kotlin array is a Java array and a Kotlin String is a Java String. You create an array using:

```
var a=arrayOf(1,2,3)
```

which creates an array of three
integers. For larger arrays you can
use:

```
var array=Array(size, initializer)
```

where size gives the number of elements and initializer is a lambda expression that used to initialize the array.For example:

```
var a = Array(1000, { i -> i * 2 })
```

If you want the equivalent of an ArrayList in Java use List or MutableList. Kotlin's Strings are much like Java's but it is worth knowing that they support templating.For example:

```
var s = "Name $name and Address $address"
```

will insert the values of the variables– name and address– into the string. Although Kotlin doesn't have a record or structure type, it does have data classes.If you create a class with a primary constructor something like:

```
class MyPersonClass{

var name:String=““
var age:Int=0


}
```

then you get a class with two properties initialized as
specified. If you also add the modifier "data" in front:

```
data class MyPersonClass{ var
name:String="“
var age:Int=0

}
```

you also get some auto-generated methods including, equals, copy, hashcode, toString and componentN. The componentN methods are particularly useful as they provide destructuring:

```
var myDataObject=MyDataClass(“Mickey”,89)
var (myName,myAge) = myDataObject
```

which unpacks the properties in to the separate variables.
A special case of destructuring is the spread operator *. If you have a function that accepts a variable number of arguments you can pass it an array using;

```
val a=arrayOf(1,2,3)
val list =asList(*a)
```

Null Safety

Perhaps one of the most subtle features of Kotlin you have to get to know and use is its

null safety.
References can be either non-nullable or nullable.
If you declare a variable in the usual way you get a non-nullable: `var myVariable:sometype= something` and you cannot set the variable to null because:

```
myVariable=null
```

throws a compiler error.
The compiler tracks operations that could generate a null, and flags any operation that could possibly set a non-nullable variable to null. For example, if a function could return a null you cannot assign it to a nonnullable. This often occurs if you are trying to use a Java function which, of course,doesn't support non-nullable types.
If you need a nullable reference then you have to explicitly declare it using ?
as in:

```
var myVariable:sometype?=something
```

Now myVariable can be set to null:

```
myVariable=null
```

and this works without a compiler error or warning.

Kotlin tracks your use of nullables and makes

sure you don’t assign a nullable to a nonnullable without checking that it isn’t null.If you do use a nullable without checking that it is safe,the compiler will warn you and refuse to compile the program.
There are various operators that make life easier if you are working with nullable types.

The protected call administrator ?. will possibly get to a property assuming the item is non-invalid. For example:

```
var myVariable=myObject.myProperty
```

won’t arrange assuming myObject is a nullable. Utilizing the protected call operator:

```
var myVariable:Int?=myObject?.myProperty
```

takes care of business and myVariable is set either to the worth of the property or to invalid on the off chance that myObject is invalid. Notice that myVariable must be nullable for everything to fall into place. To play out an activity assuming something is non-invalid you can utilize the let method:

```
a?.let {a=a+1}
```

the square of code in the wavy sections is possibly executed in the event that an is non-null.

Finally there is the safe cast **as?** **which will return null if the cast isn't possible. So:**

```
variable as? type
```

will assess to invalid assuming the cast to type isn’t possible.

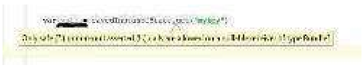
Java Types and Null

When working with Kotlin then you can keep nulls under control. All the variables you use can be non-nullable and the only cost of this is that you have to initialize them when they are declared or soon after. In principle, it is very difficult of a null to occur in pure Kotlin, but when you are working with Android you cannot avoid interworking with Java and null is a possible value for any Java variable. Not only this but the type being used by a Java method that you have to call may not be a Kotlin type at all. In this case Android Studio studio shows them as T! meaning they could be referenced by a nonnull or nullable variable, i.e. a T or a T?

Similarly, Java assortments can be treated as impermanent or permanent and can be nullable or non-nullable. The IDE shows these as (Mutable) Collection <T>!. At long last a Java exhibit is displayed as Array<(out) T>! implying that it very well may be a variety of a sub-kind of T nullable or non-nullable.
Given that Java types are nullable you have two options in dealing with this. You can set the sort to a Kotlin non-nullable of the same kind, or you can save the nullable kind and check for nulls. For instance, the savedInstance Bundle passed into the onCreate technique is a Java object thus it is a nullable:

```
supersede fun onCreate(savedInstanceState: Bundle?)
```

If you attempt to utilize one of its strategies in the typical manner then you will see the accompanying message:



```
var value = savedInstanceState.get("mykey")
Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of type Bundle?
```

Changing to the protected strategy call:

```
var value= savedInstanceState?.get("mykey")
```

makes the mistake message disappear, however presently esteem is

gathered to be a nullable. For this situation esteem is Any?. Notice that if either get is invalid or “mykey” doesn’t exist esteem is set to null.

At this point any invalid qualities are not causing an issue. Anyway assuming we cast the worth to a non-nullable sort things can go wrong: `var value= savedInstanceState?.get("mykey") as Int`

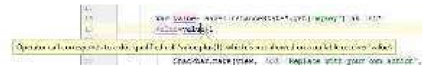
Now esteem is a non-nullable Int and keeping in mind that this aggregates it will cause a run time exemption assuming the outcome is to attempt to allocate an invalid to a none nullable.The compiler adds a declaration that the worth is non-invalid to secure you and assuming it is you will see a runtime exemption that incorporates the message:

```
Caused by: kotlin.TypeCastException: invalid can't be cast to non-invalid
sort kotlin.Int
```

Alternatively you could project to a nullable type:

```
var value= savedInstanceState?.get("mykey") as Int?
```

Now esteem is a nullable Int and it both accumulates and runs without a special case regardless of whether the outcome is invalid. Obviously assuming you currently attempt to utilize esteem you actually have the issue that it very well may be invalid yet presently the compiler prompts you to manage this:


[illegible]

The compiler won ’t say anything negative on the off chance that you check for an invalid prior to attempting to use

value:

```
var value= savedInstanceState?.get(“mykey”) as Int?
if(value!=null)value=value+1
```

A shorthand method of checking is to utilize the let method:

```
value?.let{value=value+1}
```

which possibly assesses the articulation assuming the variable is non-null. The issue is by all accounts that once you have an invalid in the framework it is hard to get it out and you need to test and respond to it in all of the code where it makes a difference. The main option is to supplant it with a non-invalid outcome that acts effectively. You can do this in Kotlin utilizing the amusingly named Elvis administrator ?:. For example:

```
value?:0
```

is zero assuming worth is invalid.As a general rule, assuming you have a worth that will fill in for an invalid then you ought to kill the invalid when possible.

For instance, for this situation we can utilize the Elvis administrator to make esteem a non-nullable type:

```
var value= (savedInstanceState?.get(“mykey”)?:0) as Int
```

Now esteem is a non-nullable Int and assuming that Java returns an invalid for savedInstance or the consequence of get then it has a worth of zero. In many cases it is ideal to eliminate the invalid at its first appearance. So for instance, we can make savedInstanceState safe:

```
val safeSavedInstanceState=savedInstanceState?:Bundle()
```

If savedInstanceState is invalid safeSavedInstanceState is a recently started up Bundle.This likewise permits you to utilize cluster access instead of the get. Notice anyway that this doesn’t dispose of the invalid issue as:

```
var value= safeSavedInstanceState[“mykey”] as Int
```

```
value=value+1
```

will gather, however you will produce a runtime special case assuming mykey isn’t in the Bundle as this then, at that point, returns an invalid. So you actually need to make sure that outcome isn’t null:

```
var value= (safeSavedInstanceState[ “mykey”]?:0) as Int
```

Nulls are a typical reason for runtime accidents and it merits investing energy working out what ought to occur assuming something you depend

on in code is invalid. Kotlin doesn’t tackle the invalid issue, however it gives you the instruments to settle it.

Kotlin Aims to Help

There are a lot more ways that Kotlin, the compiler and the module to Android Studio, attempt to make life more straightforward. There are beyond any reasonable amount to cover exhaustively and you can expect both

code finishing and code provoking to improve as Kotlin support in Android Studio is improved.
Notice that Android Studio Takes us past unadulterated Kotlin in the scope of offices it gives. Take, for instance, the programmed restricting of properties of the Activity to the ids appointed to parts in the XML records. This isn’t important for the Kotlin language, however it is a lot of a piece of Android programming with Kotlin.
In many cases you should see that the messages and prompts that are given by Android Studio give you enough direction to work out what choices are on offer and what the answers for issues really are. Always set aside effort to peruse the prompts and messages as the appropriate response is regularly gazing you in the face.
If you might want to find out about Kotlin, then, at that point, I prescribe my book:
Programmer’s Guide To Kotlin ISBN:978-1871962536

Summary

☐ Kotlin is completely viable with Java and gives numerous
rearrangements to utilizing the Android Java libraries.

☐ If you are a Java software engineer there are numerous thing you need to escape the propensity for composing – particularly semicolons.

☐ In Kotlin there are no fields, just properties complete with get and set techniques. These guide to Java fields with get and set methods.
☐ The Kotlin module changes over the ids alloted in the XML format document to properties of the Activity.
☐ Event overseers are not difficult to make utilizing either lambdas or objects.
☐ Data classes are an extremely simple method for making“record” like information structures.
☐ Kotlin acquaints apparatuses with

assistance you oversee nulls from an unadulterated Kotlin program.
☐ When Kotlin connects with Java code then nulls are an unavoidable truth and you need to manage the likelihood that a nullable kind is for sure null.

Index