Mobile Application Development

Lecture 1

Pranab Roy

Outline

Introduction

Mobile Application Development - Issues and Trade-offs

Android – an Overview

Android – Usage and Application

Android – Released Versions

Android – Usage Data

Android – Systems Architecture

Android – Four Main Components and Other Features

Outline

Android and ioS

Establishing Mobile Application Development Environment

Xcode – an Overview

Android Studio

Gradle

Android Studio Layout

Android Layout Editor

Gradle usage for Android and Android Cloure

Android Test Framework and Debuggers

Android – Managing Libraries

Introduction

Smartphones and tablets have become commonplace in the past decade and are now the predominant means whereby users produce and consume content over the Internet.

By 2012 smartphones and tablets had begun to eclipse desktop PC sales, particularly in emerging economies which represent a lion share of the growth .

Along with these ubiquitous, powerful hand-held touch-enabled computing devices, came their respective application ecosystems:

Google's Play Store and **Apple's iTunes App Store**.

At their annual World-Wide Developer's Conference in June 2016,Apple announced that the number of apps available in the App Store exceeded 2 million.

Google's Play Store boasts slightly more apps than Apple.

These two ecosystems remain the most popular means by which end users are introduced to new life enriching experiences on their devices

Introduction

Both offers what are properly referred to as **native applications**, or simply apps in everyday vernacular.

Native applications are programs that are implemented by software developers, compiled into binary bundles, and published on the respective application ecosystems.

End users can purchase, download and install these applications on their devices.

Native apps are not the only way to get new user experiences onto smartphones and tablets.

These range from applications that are implemented as **web applications** on the one side, to **downloadable native apps** on the other.

In between these two extremes there are also a significant number of **hybrid approaches**

- **frameworks** that allow developers to implement applications in a **common tool suite**
- Then generate **buildable projects** for the various **target native platforms**

Introduction

Figure 1.1 offers a wide spectrum of exemplar techniques and tooling available to software developers that can be used to implement applications for mobile devices.

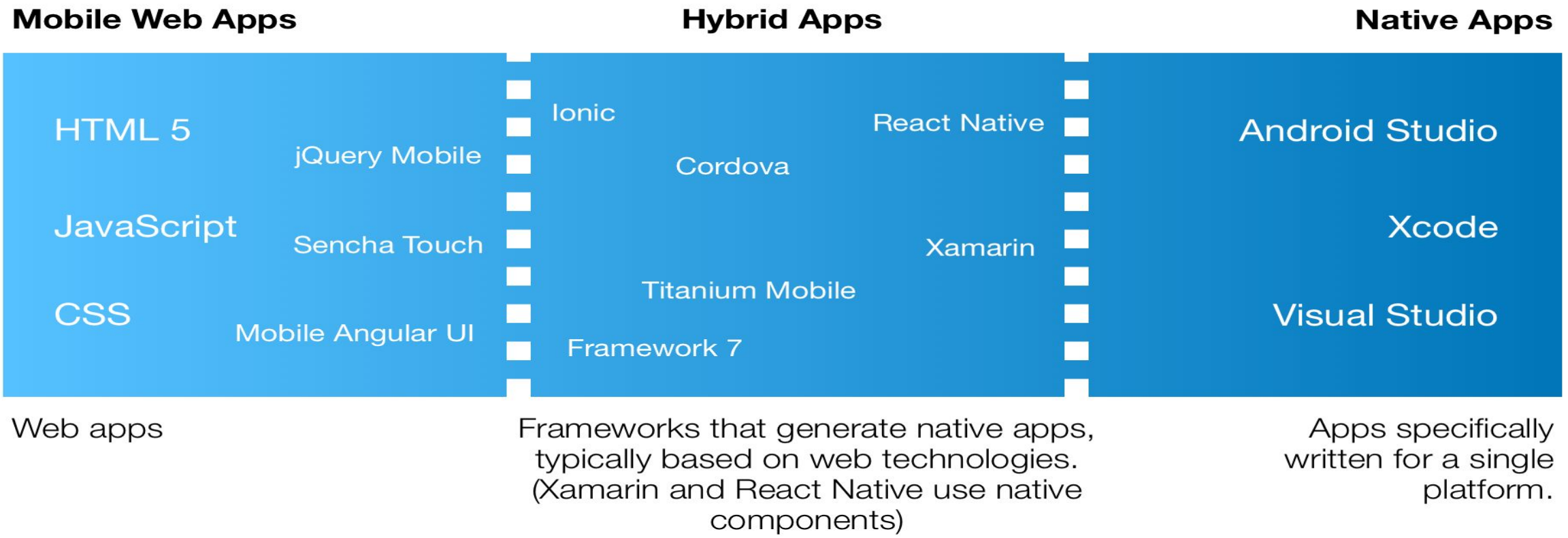


Figure 1.1: New content and experiences can be published on mobile devices via a spectrum of techniques, ranging from web apps to native apps.

Mobile Application Development – Issues and Trade -offs

When deciding how a particular mobile application should be developed,
there are a number of trade-offs to consider.

Mobile web apps are typically easier on the budget

a single responsive web app can be implemented that renders quite well on a variety of smartphone, tablet, and desktop client devices.

Mobile web apps are also much easier to manage - As changes can be made on the server side and immediately made available to all client devices using the application.

Mobile Application Development – Issues and Trade -offs

Native mobile apps on the other hand, target a very specific software stack (most often iOS, iPadOS1 or Android). The programming languages and frameworks used are different, and require a separate implementation for each target platform.

To update a native app, the developer must submit the updated binary to the respective app store.

While the level of testing and vetting that happens in the approval process varies across (and within!) ecosystems, the simple fact is that the app may not become available to users for a number of hours, or even days.

Even then, the update is not made until each individual client device downloads and installs the update.

Mobile Application Development – Issues and Trade -offs

Native apps on the other hand are advantageous for at least a couple of reasons.

First and foremost, web apps are implemented via the browsers available on the client devices.
From a user experience perspective, this results in a sort of “lowest common denominator” like situation in which a number of the device’s underlying capabilities are simply not available to the application via the browser,
and browser features vary across implementations.

Hence developers will approach the implementation in a way in which the app will function on all devices / browsers,which means certain capabilities will not be used.

While HTML5 has begun to change this with its tighter integration with the underlying device features, conventional wisdom is that native mobile apps deliver a far more optimal end user experience than mobile web apps.

A second often overlooked advantage that native apps have over web apps is that of being discoverable.
Since the advent of the Apple’s App Store in 2008, users have very quickly been conditioned to find new app experiences via searching the respective app store on their devices.

Popular apps can often be located and installed within a few key strokes.

This fact is one of the reasons that most popular consumer facing experiences on devices today are experienced via a native app, even when there is a parallel companion web app available.

Mobile Application Development – Issues and Trade -offs

Hybrid frameworks attempt to take the best of both web apps and native apps and make them available to developers.

Like web apps, hybrid frameworks address the market fragmentation problem that challenges native app developers,

by making it possible to develop the app once in the framework's software stack, and via tooling, rendering it into a native app bundles that can be published in the app stores.

While significant progress is being made in this area (such as React Native and Flutter), many of these frameworks are based on HTML5, CSS3, JavaScript and the resulting apps in the app stores are almost immediately recognizable as apps that were not developed using the native SDKs.

Mobile Application Development – Issues and Trade -offs

In summary, there is no single approach for developing mobile apps that is optimal in every situation.

As always, the best approach depends on the unique circumstances (budget, target audience, timeline, etc.) of a given project.

A general rule of thumb is that anytime you have a “captive audience” (e.g. a situation in which users have no choice but to use your app), the web app, or hybrid approach is worth looking into.

This often occurs when apps are targeting a specific set of users within an organization, such as an enterprise or university community.

On the other hand, whenever your audience consist of the general population of consumers (who already have over 2 million apps to choose installing over yours!) building a beautiful, easy to discover, easy to use native app for each of the target platforms will often be a requirement.

Android – An Overview

Welcome to the Android World! Android is by now the most popular mobile operating system (OS) in the world, and you can find Android phones wherever you go

In October 2003, Andy Rubin and a few others founded the Android Inc.

In August 2005, Google acquired Android Inc. which was only 22 months old at that time. Andy Rubin stayed and continued to be responsible for the Android project.

After years of R&D, Google released the first version of Android OS in 2008.

However, ever since then Android has faced lots of backlashes. Steve Jobs insisted that Android was an iOS knockoff that stole the ideas of iOS, and he threatened to destroy Android at all costs.

Android OS is based on Linux, but it was removed from the Linux kernel main branch by Linux team in 2010. Since all the apps in Android were developed with Java in the early days, Oracle also sued Google for intelligence infringement. . .

However, all of these couldn’t stop Android from taking up the market share rapidly.

Google made Android an open OS which means that any OEM or person can get the source code of Android OS for free, and can use and customize the OS freely

Android – An Overview

Samsung, HTC, Motorola, Sony-Ericsson, etc., all released their Android phones.

After being released for 2 years, Android had already overtaken Nokia Symbian which had been the leader of the smartphone mobile OS market for more than 10 years.

At that time, millions of new Android devices were activated every day.

Today, Android has more than 70% of global smartphone OS market share.

Just think that about 7 out of every 10 individuals' phone can run the app you write in Android.

Android – An Overview

More than 20 versions of Android have been released so far, and Google has built a holistic ecosystem during the past years.

OEMs, developers, and users all play an important role in this ecosystem, and they work together to push the evolution of Android.

Developers make a crucial part of this ecosystem.

no matter how good the OS is, if there are not enough apps available, users wouldn't feel like using the OS.

Who will use an OS that does not support Facebook, Messenger, or Instagram?

Furthermore, Google Play is the marketplace for Android apps

If you can make high-quality apps that can attract users, then you can get good return from your apps.

If your app is popular, you can even become an independent developer to support yourself or even create a startup!

Android – An Overview

Android is developed by a consortium of developers known as the Open Handset Alliance and commercially sponsored by Google.

It was unveiled in November 2007, with the first commercial Android device, the HTC Dream, being launched in September 2008.

Most versions of Android are proprietary.

The core components are taken from the Android Open Source Project (AOSP), which is free and open-source software (FOSS) primarily licensed under the Apache License.

When Android is installed on devices, ability to modify the otherwise FOSS software is usually restricted, either by not providing the corresponding source code or preventing reinstallation through technical measures, rendering the installed version proprietary.

Android – Usage and Application

Most Android devices ship with additional proprietary software pre-installed,

most notably Google Mobile Services (GMS)which includes core apps such as Google Chrome, the digital distribution platform Google Play, and associated Google Play Services development platform.

Over 70 percent of Android smartphones run Google's ecosystem; some with vendor-customized user interface and software suite,

TouchWiz and later *One UI* by Samsung, and *HTC Sense*.

Competing Android ecosystems and forks include
Fire OS (developed by Amazon),
ColorOS by OPPO,
OriginOS by vivo
OxygenOS by **OnePlus**
MagicUI and EMUI 8.2 by Honor
LinageOS – successor of **CyanogenMod (**earlier open source OS based on Android)

However, the "Android" name and logo are trademarks of Google which imposes standards to restrict the use of Android branding by "uncertified" devices outside their ecosystem.

Android – Released Versions

In September 2008, Google released the first version of Android—Android 1.0.

In the following years, Google kept updating Android at an amazing speed. v2.1, v2.2, and v2.3 helped Android to grab a huge market share.

In February 2011, Google released Android 3.0, which was dedicated to tablet and was one of the few versions that didn't get traction.

Soon, in October of the same year, Google released Android 4.0 which stopped treating phone and tablet differently and could be used in both phone and tablet.

In 2014, Google released Android 5.0 which claimed to have the most drastic changes in the history of Android. That version used ART to replace Dalvik virtual machine which greatly boosted the speed of apps.

The concept of Material Design also came from that version to help optimize the UI design.

Google also released OS that can be used in specific devices such as Android wear, Android Auto, and Android TV at the same time.

After that, Google accelerated the speed of releasing Android OS updates and releases a new version every year. By 2019, Android was already at v10.0 which was also the latest version

Android – Released Versions

Version number	Codename	API	Market share
2.3.3–2.3.7	Gingerbread	10	0.3%
4.0.3–4.0.4	Ice Cream Sandwich	15	0.3%
4.1.x	Jelly Bean	16	1.2%
4.2.x		17	1.5%
4.3		18	0.5%
4.4	KitKat	19	6.9%
5.0	Lollipop	21	3%
5.1		22	11.5%
6.0	Marshmallow	23	16.9%
7.0	Nougat	24	11.4%
7.1		25	7.8%
8.0	Oreo	26	12.9%
8.1		27	15.4%
9	Pie	28	10.4%
10	Q	29	
11	Red Velvet Cake	30	
12	Snow Cone	31,32	
13	Tiramisu	-	

Android – Usage Data

As of May 2021, Android has over three billion monthly active users, the largest installed base of any operating system,

and as of January 2021, the Google Play Store features over 3 million apps.

Android 12, released on October 4, 2021, is the latest version.

Roles of a Mobile Developer

Responsibilities

Designing an Application

Building

- User Interface(UI),
-Writing Codes for functionalities,
- Integrating Application Program Interface
- Developing Animations

Maintaining

- Writing Tests
- Releasing Apps
- Uploading Updates

Team works

- Other Developments
- Project Functions
- Quality Assurance
- Collaborative Design

Version Control

- Track and Manage Changes
- Work collaboratively and simultaneously to a project by enabling access to the most recent version of a software code [Github]

Planning
Meeting
Research and Studies on Trends and Development

Mobile Apps

Design

User Experience

Server Technology

Continuous Integration

Android – Systems Architecture

Roughly, Android can be divided into four layers:

Linux kernel layer,
system lib layer,
application framework layer, and
the applications layer

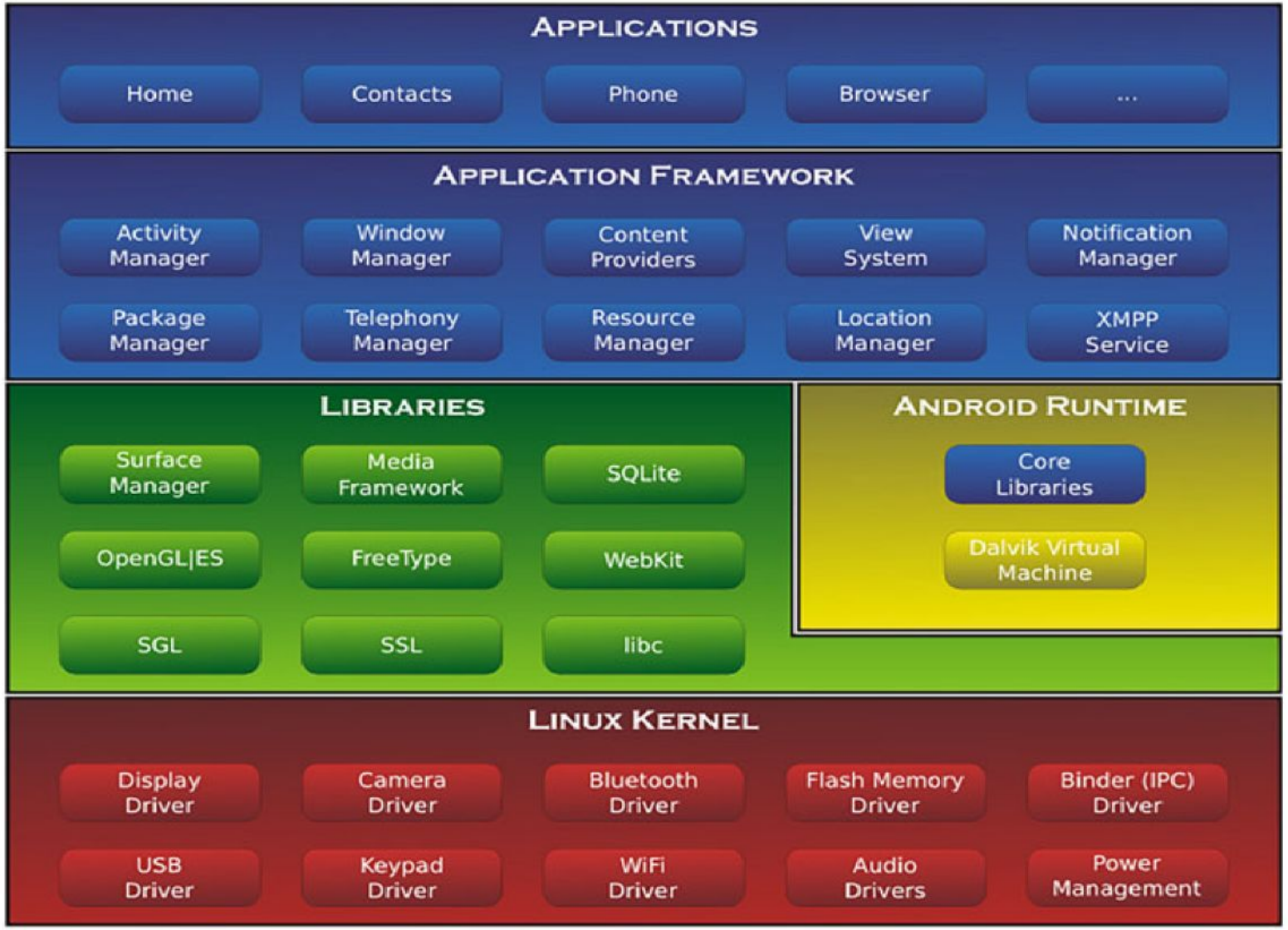


Fig. 1.1 Android system architecture.

Android – Systems Architecture

1. Linux Kernel Layer

Android is based on Linux kernel, and this layer provides the drivers for the various hardware of Android devices, for example, display driver, audio driver, camera driver, blue-tooth driver, Wi-Fi driver, battery management, etc.

2. System Libraries Layer

This layer provides primal support to the system with some C/C++ libraries.
For example, SQLite provides database support for the database, OpenGL/ES provides support for 3D graphics, Webkit lib provides support for browser kernel, etc.

Android runtime is also in this layer, which provides some core libraries to allow developers to develop Android apps with Java.
The Dalvik virtual machine is also in the runtime (replaced with ART after version 5.0), and it allows the apps
to run in their own processes and have their own virtual machine instances.
Compared with Java virtual machine (JVM), Dalvik and ART have been specifically optimized for mobile devices which usually have less memory and computing power.

Android – Systems Architecture

3. Application Framework Layer

This layer provides all kinds of APIs that developers use to build the apps. The Android system apps utilize these APIs.

4. Applications Layer

All the apps are belonging to this layer, for example, system apps like Contact, Message, and the apps being downloaded from Google Play, and of course the apps those are going to be built.

Four Main Components

The four main components of Android application development are

Activity,
Service,
BroadcastReceiver, and
ContentProvider.

Activity is used to display the UI of the app, and everything you can see is hosted in Activity.

Unlike Activity, Service is invisible, instead, it can run in the background even if the user exits the app.

BroadcastReceiver allows your app to receive broadcast messages from apps like Phone and Message. Of course, the app can also send broadcast messages.

ContentProvider can be used to share data between different apps.

You may need the help of ContentProvider to read the contacts in the system Contact app.

Rich System Widgets

Android provides plenty of system widgets to help developers build beautiful UIs.

Of course, you can always create your own widgets to meet your special requirement.

SQLite Database

Android is embedded with SQLite database, which is a lightweight and superfast relational database.

It supports standard SQL syntax and can be accessed using APIs provided by the system which make CRUD operations super-easy.

Powerful Multimedia Support

Android provides very powerful multimedia support like music, videos, voice recording, taking photos, etc.

You can use the supported APIs to make miscellaneous apps.

iOS and Android

At a surface level, Android and iOS share many similarities.

- Both are highly popular smartphone software platforms.
- Both are controlled solely by a single large tech company, Apple and Google, respectively
- Both continue to evolve at a fairly rapid rate, which means as a mobile developer, you'll always be on the learning curve.
- Both have large healthy app ecosystems in which developers can publish their apps.
- Both platforms have attracted a large community of active developers.

iOS and Android

There are also a number of significant differences that have ramifications on how developers approach them

• While Google has exclusive control of the Android software stack, it makes it freely available to a large number of device manufacturers.
On the other hand, Apple has exclusive control of both the iOS software stack as well as the hardware platforms it runs on.

• The Android software stack has been open-sourced by Google. The iOS platform is Apple proprietary.

• Native Android development can be done on a variety of desktop operating systems (Windows, macOS, or Linux), but native iOS development can only be done on macOS.

• While both Apple and Google now use humans to vet (e.g. to make sure they meet the store's requirements / guidelines) the apps that get published in their respective ecosystems,

the Apple review process takes significantly longer and is usually more rigorous.

• Android development is done in either Java or Kotlin[2] (and C/C++ via Native Development Kit/NDK support).
iOS development is done in Swift or Objective-C.

Establishing Mobile App Development Environment

For command line interface (CLI) oriented developer who insists on carrying out all software development tasks from the command line, there are specific ways to make that happen for mobile app development.

However, for the rest, iOS and Android development is carried out using the official integrated development environments Apple and Google provide which are as follows:[3]:

- Xcode 12.3: Apple's IDE for apps running on iOS, watchOS, macOS, and tvOS.

- Android Studio 4.1.1: Google's IDE for apps running on Android on phones/tablets, Wear OS, TV, Android Auto and Android Thing

Establishing Mobile App Development Environment

Besides features commonly found in other modern IDEs like:

code completion, code refactor, code analysis,

these IDEs support seamless integration with source control management programs(like git).

However, one of the most important features is the ability of these IDEs to connect and run and debug any app on either a device simulator or a physical device.

To enable this feature, additional “modules” may have to be installed into the respective IDE.

Xcode

Xcode is only supported on computers running an up-to-date version of Apple's macOS operating system.

The software can be freely downloaded via the MacStore app that is preloaded on every macOS installation.

Developers can use Xcode to develop and run/debug apps on simulators or physical iOS devices.

However, to publish apps on the App Store, developers have to pay an annual $99 developer fee.

Xcode supports iOS development in both Swift and Objective-C.

The Swift language (currently version 5) was introduced by Apple in 2014 and has since been open-sourced.

Objective-C, a derivative of the stalwart C programming language enhanced with object-oriented syntactical sugar, has long beer Apple's flagship programming language.

Both languages are currently supported by Xcode, and developers can mix and match modules written in both languages within a single project.

Xcode IDE

The Xcode IDE is a flexible IDE with support for the typical features one would expect in a modern IDE

Figure 1.2 highlights the various functional panes that the IDE window is divided into when editing code.

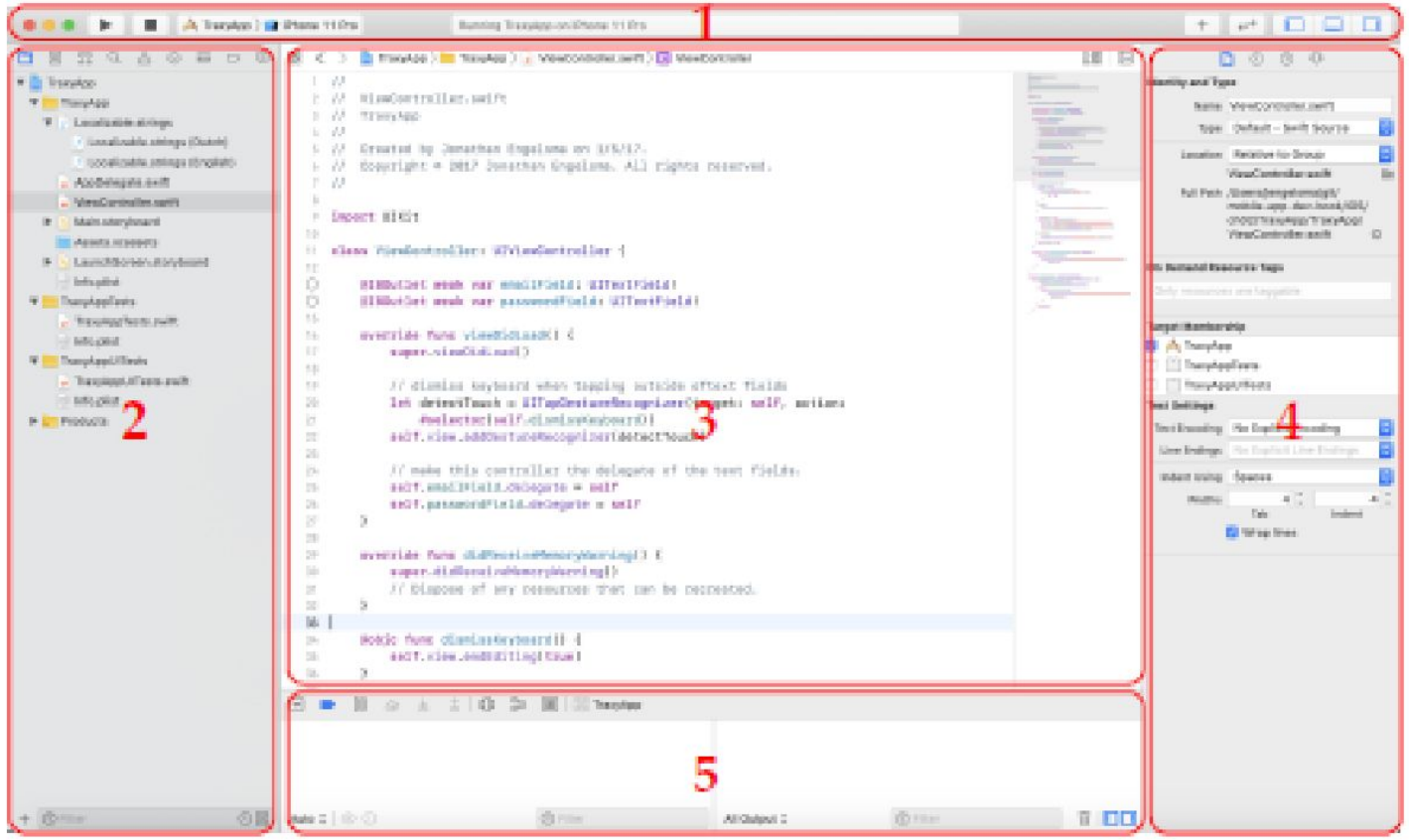


Figure 1.2: The Xcode integrated development environment.

• **Toolbar (1):** The toolbar displayed at the top of the IDE provides you
buttons for starting /stopping the simulator,
the status of an ongoing build, and
a variety of buttons that allow you to display / hide various panes within the IDE.

• **Nav Area (2):** The navigation area allows us to navigate through the various project artifacts that our app is built from,
including source code,
image asset catalogs,
view layouts, etc.

The nav area is actually populated by a tab like view that also provides us information on unit test outcomes, build errors, and the run-time stack and break points during debug sessions.

• **Editor (3):** The editor area takes the bulk of the Xcode window. If you are writing or editing code, the editor pane will be filled with a code editor that supports
syntax highlighting, auto completion, etc.

If you are working on the user interface of an app, the editor pane will be filled with an interactive scene editor (named Interface Builder) that allows us to construct a user interface in a visual drag and drop manner.

• **Utility Area (4):** The utility area is a tabbed view of various inspectors.

What gets displayed by these inspectors is entirely dependent on what is currently selected within the editor
pane.

The inspectors let you manipulate attributes of items selected in the editor,
such as
UI controls within a view.

It is possible also to manipulate connections between view artifacts and code, layout constraints, etc.

• **Debug Area (5):** The debug area gives the developer interactive access to the internal state of a running program when running in debug mode.

The Xcode debugger provides you with a structured view of your app’s current context in the pane on the left, and a command line console like input/out interface on the right.

Interface Builder

iOS developers have a couple of different options for implementing user interfaces.

The most common approach would be to use Xcode's Interface Builder.

Interface Builder is a visual scene editor that allows the construction of user interfaces in a drag and drop fashion, Interface Builder stores the view hierarchies constructed in an Apple proprietary XML format.

There are two different types of Interface Builder files:

• *.xib files: Pronounced as "nib" files for historical reasons, these files contain a single scene or view hierarchy.

This format predates the storyboard format introduced by Apple in 2011.

• *.storyboard files: Storyboard files can contain multiple scenes as well as the transitions (also known as segues) among them.

By default, newly created projects in Xcode utilize storyboard files.

Integrated Debugger and Simulators

Apple's LLDB debugger

a more recent replacement of the gdb used in older versions of Xcode, is tightly integrated within the IDE, and capable of debugging apps via simulated iOS devices or debugging the apps onboard actual physical devices.

Figure 1.3 shows Xcode in debug mode along with an attached desktop iOS simulator.

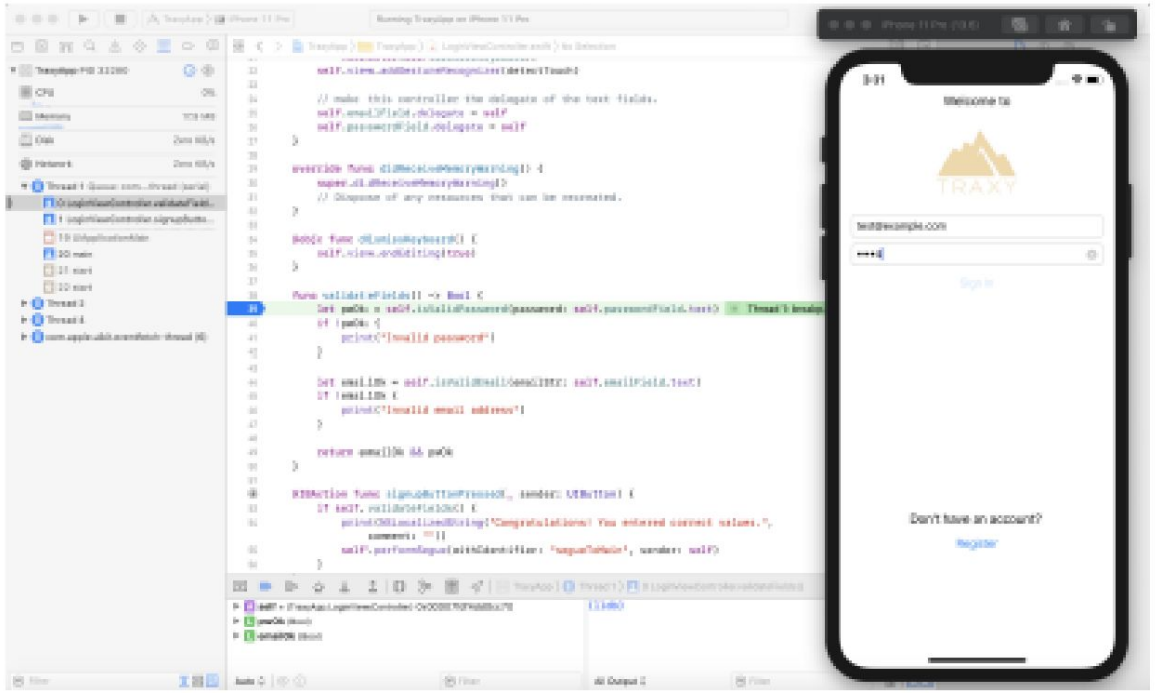


Figure 1.3: Xcode in debug mode attached to a simulator running an app.

Integrated Debugger and Simulators

Simulators that run on the Mac desktop are available for a variety of Apple products, including various models of the iPhone, iPad, Apple Watch and Apple TV product families.

In addition, they are available in multiple versions of each of the respective operating systems,

so developers can simulate fairly reasonable job of running their code on a simulator of any device an end user might have, without actually having physical possession of the device.

the iPhone’s camera and Bluetooth functionality is not supported by the simulator.

Testing an app that utilizes these features would need to be tested on a physical device.

To debug an app on a physical device, such as an iPhone, the developer must register as a developer (free account or paid) via Apple’s developer site (https://developer.apple.com)
And
connect the device to the Mac using the USB cable

Xcode Documentation

Apple provides an extensive set of documentation for its iOS frameworks and tools in the form of developer guides and API reference documentation.

The complete set can be accessed within Xcode via the application menu Window Documentation and API Reference or alternately with keyboard shortcut 

Reference documentation can also be retrieved contextually in a ”quick help” window when editing code by moving the pointer over any program identifier holding  and clicking with the mouse.

The documentation can be downloaded for off-line reference via the application menu (Xcode Preferences ... and select Documentation on the Component tab.).

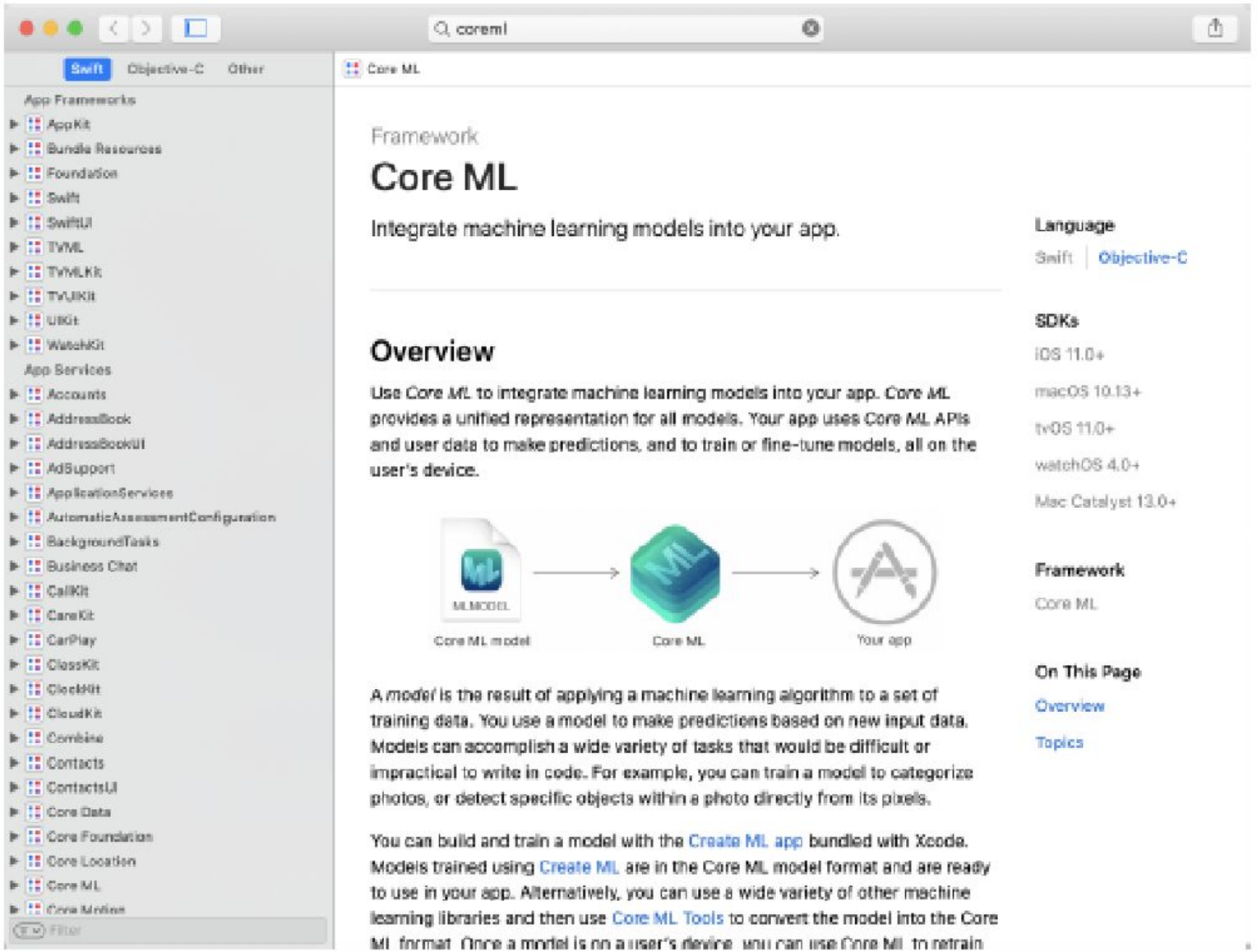


Figure 1.4: The Xcode Document and API Reference

Managing Third Party Components with CocoaPods

A vibrant open-source community has coalesced around the iOS platform, offering everything from components that simplify network programming to beautiful custom user interface controls.

Most production apps today, will incorporate a significant number of open-source components.

Originally inspired by Ruby gems, CocoaPods (https://cocoapods.org) has emerged as the de facto industry standard for dependency management of Swift and Objective-C projects.

Referred to as pods, these components can be specified within a configuration file (Podfile) and automatically downloaded and integrated into your app's build process.

Today there are thousands of Objective-C and Swift libraries available in the form of pods, coming from individual open-source developers as well as from large companies such as Google and Facebook.

Unlike gradle, the nearest analog to CocoaPods in the Android world, which manages dependencies as well as the build process,

CocoaPods only addresses external dependency management.
The actual build process is still managed and implemented by Xcode.

Android Studio

Android Studio is a multi-platform IDE that runs on the three major operating systems: Windows, Linux, and macOS.

It is available to developers as a free download.

The IDE supports the development, execution and debugging of Android applications on emulators and physical Android devices.

Android Studio is powered by the IntelliJ platform whose architecture clearly separates program editing from program building[5].

This design decision enables JetBrains (the company that develops IntelliJ) to reuse its editor front-end to create other IDEs (PyCharm, CLion, WebStorm, etc.).

For instance, while reusing most of same editor front-end, CLion builds C/C++ programs by incorporating CMake as its build system.

Android Studio

The Android Developer Tool team had several options for their choice of program builder modules to use in Android Studio.

To complement IntelliJ's core framework extensibility and flexibility, the team was looking for a build system that was also extensible and flexible.

They ultimately settled on Gradle as Android Studio build system.

The Android Developer Tool team has so much confidence in Gradle that they guarantee the same build results regardless how Gradle was invoked.

Whether it runs from the IDE, command line, or development server, in all cases, Gradle delivers the same build results

Gradle

Gradle is a build automation tool for multi-language software development.

It controls the development process in the tasks of compilation and packaging to testing, deployment, and publishing.

A build automation tool is used **to automate the creation of applications**.

Supported languages include Java (as well as Kotlin, Groovy, Scala), C/C++, and JavaScript.

It also collects statistical data about the usage of software libraries around the globe.

Gradle builds on the concepts of Apache Ant and Apache Maven, and introduces a Groovy- and Kotlin-based domain-specific language contrasted with the XML-based project configuration used by Maven.

Gradle uses a directed acyclic graph to determine the order in which tasks can be run, through providing dependency management. It runs on the Java Virtual Machine

Android Studio

Since its first release at Google I/O 2013 (Google's annual developer conference), Android Studio has improved significantly, and it is now available as an open-source IDE.

Productivity enhancing features found in IntelliJ, such as: code refactoring, auto intention, smart completion,
automatically carry over to Android Studio.
Multi-language support is another strong feature inherited from IntelliJ.
Developers who need to interface Java code with C or C++ can easily do so in Android Studio.

NDK support in Android Studio has improved significantly as well:
JNI header editing, compiling, and debugging are seamlessly integrated into Android Studio.

More recently, a Kotlin plugin for IntelliJ enables Java and Android developers to write code in Kotlin, a more recent language designed by JetBrains.

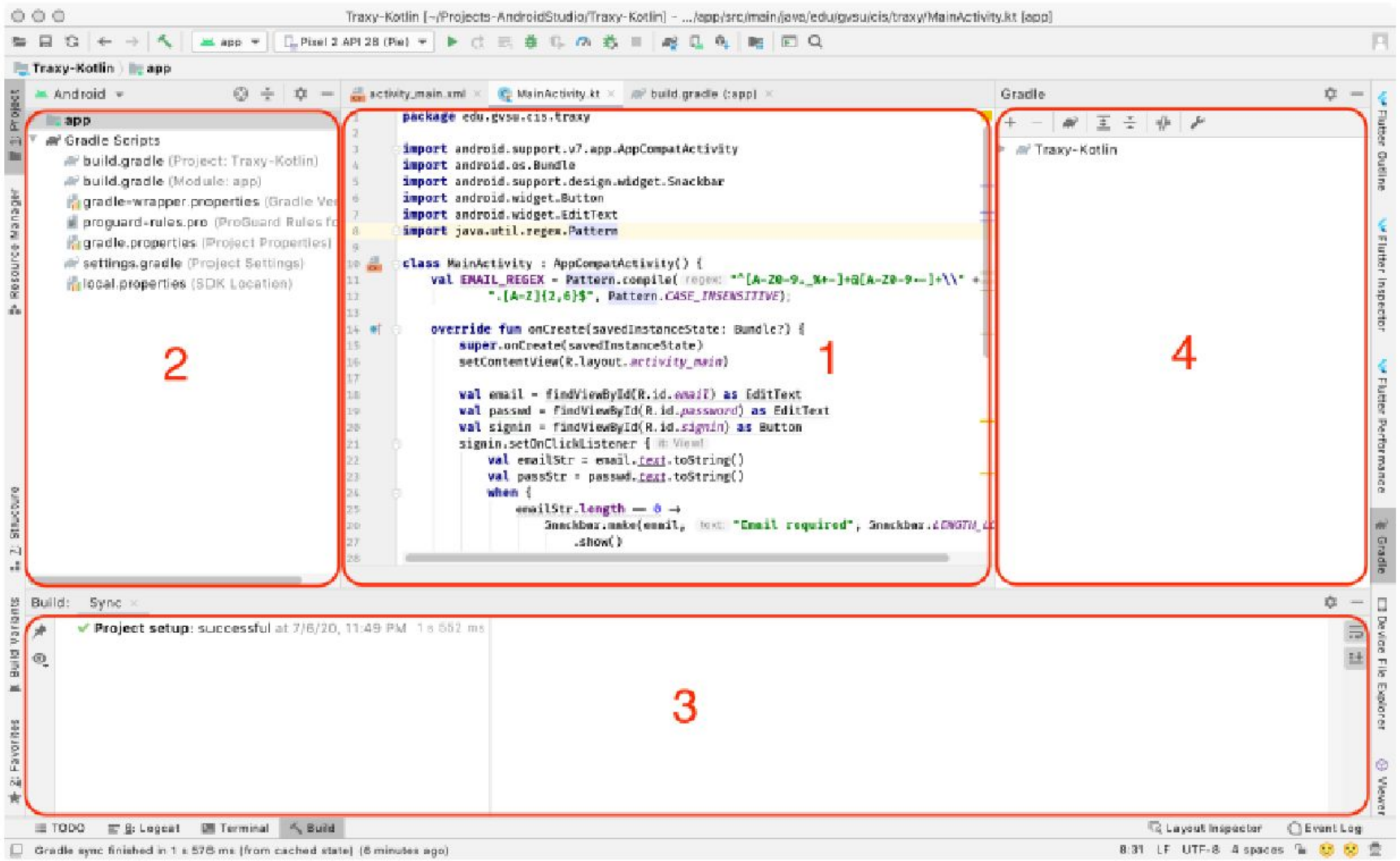


Figure 1.5: The Android Studio IDE.

Android Studio Layout

Figure 1.5 shows the workspace with the Editor panel at the center (labeled 1) and three currently expanded (labeled 2, 3, and 4) collapsible panels.

The overall layout of the Android Studio window is very similar to other IDEs,

where the top menu/tool bar holds commonly-used buttons like open file, save file, copy, cut, paste, run, debug, etc.

Visual elements that may be unique to IntelliJ/Android Studio are the collapsible panels.

Android Studio workspace is organized into several collapsible panels.

On the left and right, a vertical strip that holds vertical tabs of collapsible panels.

Similarly, the bottom screen shows a horizontal strip that holds horizontal tabs of collapsible panels

Android Studio Layout

Just above the project panel (2) there is a pull-down menu to select different layouts of the project panel.

In Figure 1.5, the current selection is “Android”.

Other selections of interest include - Project, Packages, and Production.

The upper right corner of each collapsible panel shows an icon with an arrow pointing to the collapse side.

For instance, a panel with the - icon indicates it collapses to the left gutter/vertical strip.

The tabs on left, bottom, and right gutters of the workspace show the name of panels collapsible to that corresponding side.

For instance, the figure shows that the following eight panels are collapsible to the bottom: Debug, TODO, Messages, Version Control, Terminal, Android Monitor, Event Log, and Gradle Console.

To show/hide any of these collapsible panels:

- Click the upper right collapse/hide icon to collapse the panel
- Click the tab on the gutter to expand a collapsed panel

Android Studio Layout

While developing Android apps, you will be using the following panels most of the time:

- **Run/Debug**: debugger control (step over, step into, step out, etc.), variable inspector, expression watcher and other debug related features.

- **Version Control**: source code version control using a particular utility (git, CVS, Mercurial, etc.).

- **Android Monitor**: Android LogCat, Monitor CPU/Memory/Network/GPU activity, etc.

Android Studio Layout

After a new Android project is initialized and created, the **Project** (collapsible) panel shows how various components of the project are organized.

The tree structure inside this panel is normally shown with all its nodes collapsed.

After expanding some of these nodes, you will see more details as shown in Figure 1.6.

- The **manifests** directory contains an XML file (AndroidManifest.xml) that describes the various components and permissions of the application.
- The **java** directory holds all the Kotlin (and Java) source files. These files are further organized into subpackages.
- The **res/drawable** directory holds all the images (bitmap, shape drawables, gradient backgrounds, icons, state list drawables, and vector assets)

Figure 1.6: Android Studio project structure.

Android Studio Layout

• The res/layout directory holds user interface layout descriptions. These are XML definitions of the view hierarchies of various screens displayed by the app.
• The res/menu directory holds the menu structure definitions.
• The res/values directory holds key-value pairs of several resource types: colors, strings, and styles.

An Android project may include additional resource subdirectories besides the ones shown in the figure.

For instance, animations can be defined declaratively in XML files under res/anim.

Layout Editor

Android Studio supports a visual editor referred to as the layout editor that is somewhat analogous to Xcode's Interface Builder.

Using the layout editor, screens can be designed by dragging Android widgets from the layout editor's widget palette and onto the screen.

Android supports a variety of layout managers, including the most recent addition, the ConstraintLayout

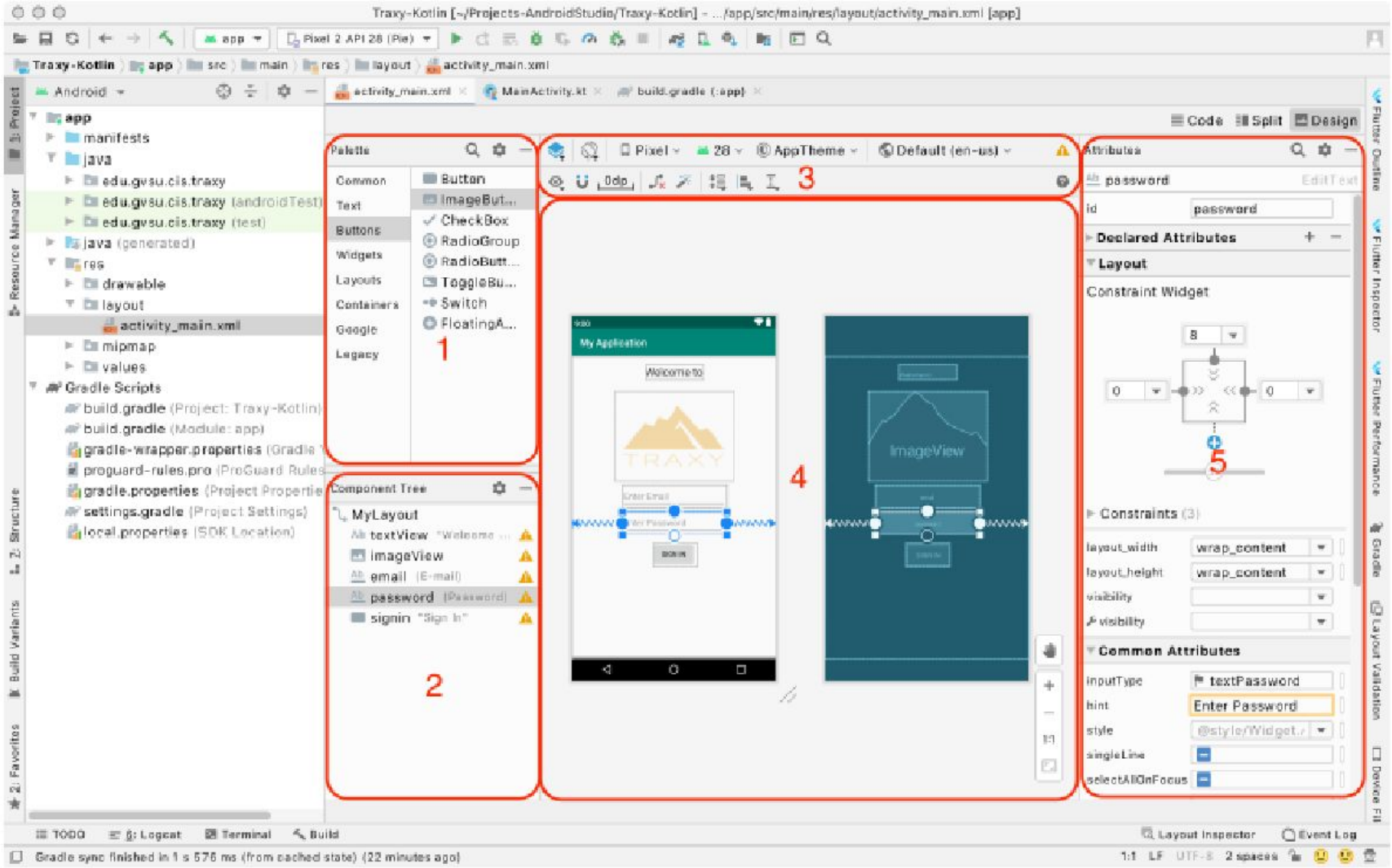


Figure 1.7: Android Studio's Layout Editor.

Layout Editor

With reference to the five annotated areas of the figure,
widgets can be dragged from the palette (1) onto the design editor (4).

The design editor shows both the design as well as blueprint view.

The component view(2) shows the view hierarchy of the layout.

The toolbar(3) above the design editor lets the user to configure the appearance of the layout

To modify the selected theme, or envision what the layout will look like on a particular device.

The properties panel (5) shows the property editor of the currently selected widget.

The Android documentation uses the term widget to refer to user interface components

Gradle – Use for Android

Using a more powerful plugin mechanism provided by the Gradle build system, Android developers can conveniently use just one IDE to develop Android apps, even if the app incorporates functionalities from non-Android modules.

Gradle manages such heterogeneous apps by organizing a project into modules.

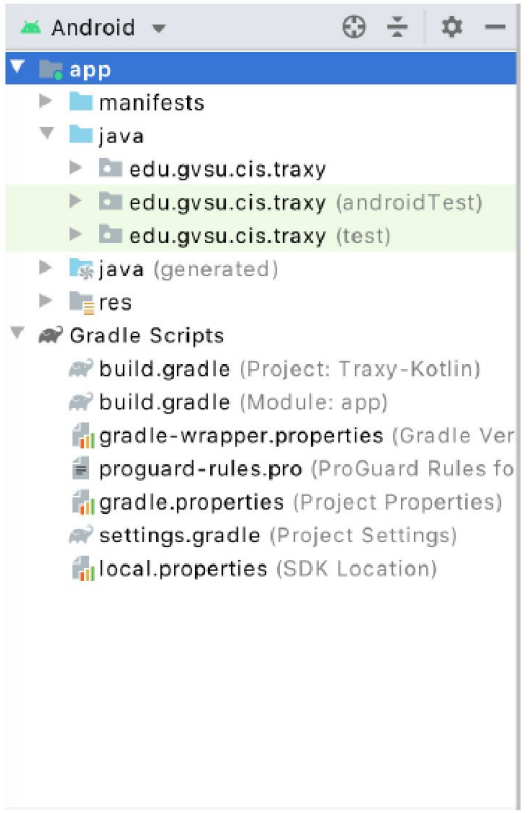
Each module typically requires a separate plugin.

All the build settings of a gradle module are specified in a build.gradle file.

For simple projects, Android Studio creates projects with two gradle.build files, which can be found under the **Gradle Scripts** node in the Project panel (left collapsible panel).

In Figure 1.6 each build.gradle file is tagged with its scope: “(Project: Traxy)” and “(Module: app)”.

Gradle – Use for Android



For Android beginners, keeping up the build environment up-to-date can be a daunting task;
they have to keep the following three interrelated components in sync with each other:

• The Gradle Plugin version is set in the build.gradle file of the project itself (the parent the Android module).
The sample setup in uses Android Gradle Plugin 3.5.2.
If the version is too old compared to the SDK being used, Android Studio will complain with an error message. The most common fix to this problem is update the build tool to the latest release.

• The Gradle executable plugin is set in another text file gradle-wrapper.properties.

When opening an existing project after upgrading to a newer Android Studio release, you may also be prompted to
upgrade the Gradle binary to a specific version. This additional step is required because newer Android Plugin for Gradle may require features only available in newer Gradle binary executable.

Gradle – Use for Android

Gradle is both a program builder and a dependency manager. This dual role of Gradle lessens the number of third-party installs for Android developers.

As seen previously, the lack of a dependency management feature in Xcode has resulted in the emergence of the CocoaPods dependency management utility.

build.gradle at outer layer

First look at the build.gradle file at the outer layer as shown below:

```
buildscript {
    ext.kotlin_version = '1.3.61'
    repositories {
        google()
        jcenter()
    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.5.2'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$
kotlin_version"

    }
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}
```

First, the two repository closures all use google() and jcenter() methods.

These two methods are used to specify the code repository that this project is going to use.

The google repo is Google's Maven repo and jcenter is the repo mainly for third party open sources libs.

With these two methods, we can easily use any libs in the google and jcenter repo.

Next, in the dependencies closure, classpath specifies Gradle plugin and Kotlin plugin.

Why need to specify Gradle plugin? This is because Gradle wasn't specifically created for building Android projects but for other types of projects written in Java, C++, etc.

If we want to use it to build the Android project, we need to specify in Gradle to use com.android.tools.

build:gradle:3.5.2 plugin. The series number at the end is the plugin version number and should be the same as the current Android Studio version number.

The Kotlin plugin just means that the current project is written with Kotlin.

If you use Java to develop Android project, then there is no need to use this plugin.

That’s everything for the outer layer build.gradle.

Usually, you don’t need to modify this file unless you want to make some global configuration changes

build.gradle (app directory)

Next, let us look at the build.gradle under the app directory, code should be the same as follows:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
  compileSdkVersion 29
  buildToolsVersion "29.0.2"
  defaultConfig {
    applicationId "com.example.helloworld"
    minSdkVersion 21
    targetSdkVersion 29
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner "androidx.test.runner
AndroidJUnitRunner"
  }
  buildTypes {
    release {
      minifyEnabled false
      proguardFiles getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
    }
  }
}

dependencies {
  implementation fileTree(dir: 'libs', include: ['*.jar'])
  implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$
kotlin_version"
  implementation 'androidx.appcompat:appcompat:1.1.0'
  implementation 'androidx.core:core-ktx:1.1.0'
  implementation 'androidx.constraintlayout:
constraintlayout:1.1.3'
  testImplementation 'junit:junit:4.12'
  androidTestImplementation 'androidx.test.ext:junit:1.1.1'
  androidTestImplementation 'androidx.test.espresso:espresso-
core:3.2.0'
}
```

The first line applies a plugin which has two values to choose from:

com.android.application means this is an application module;
com.android.library means this is a lib module.

The biggest difference between them is that the application module can run independently while lib module will need to be loaded in other apps so that it can run.

The next two lines apply the kotlin-android plugin and kotlin-android-extensions plugin.

If you want to use Kotlin to develop Android app, then you must add the first plugin.

The second plugin provides some useful extensions of Kotlin

Android closures

Next is a large android closure in which we can configure everything needed to build the project.

Among them, compileSdkVersion is used to specify the SDK version used to compile the project and here we set it to SDK 29 which is Android V10.0. If there is a newer version, Android Studio will notify you.

Android closures

Then we can see a defaultConfig closure within the Android closure which can specify more details about the project configuration

In this closure, applicationId is the unique identifier of the app and cannot be duplicated.

By default, it will use the package name we specified when we created the project.

minSdkVersion is used to specify the oldest version of Android that this project is compatible with, and it is set to 21 which is Android 5.0.

targetSdkVersion specifies the target Sdk version which means that you've already extensively tested the app in this version and the system can open some new functionality or features to the app

Android closures – buildtypes, Debug, release

buildTypes closure. This closure is used to specify the configuration for the build file which usually has two sub-closures: debug and release.

Debug closure is used to specify the configuration for the debug version of installer

and release closure will specify the configuration for production version of installer.

In the release closure, minifyEnabled is used to specify if we need to obfuscate the code or not.

proguardFiles is used to specify the file with obfuscating rules.

Here it specifies two files. The proguard-android-optimize.txt file is under <AndroidSDK>/tools/proguard directory and has general obfuscation rules for all projects.

The proguard-rules.pro is under the root directory of current project which can be used to write the obfuscating rules for the current project.

It is worth noting that all the installers generated by directly running Android Studio build and run are debug version

Android closures -dependencies

Next is the dependencies closure which can specify all the dependencies of the current project.

There are three types of dependency:
local binary dependency, local library module dependency, and remote binary dependency.

Local binary dependency can add dependency to local jars or directories;
local library dependency can be used to add dependency to local lib modules;

remote dependency can be used to add dependency to the open-source projects in jcenter.

In dependencies closure, implementation fileTree is a local binary dependency,

It will add all the files under libs directory with .jar suffix to the current project’s build path.

Implementation is for remote binary dependency and android.appcompat: appcompat:1.1.0 is a standard remote dependency lib.

Androidx.appcompat is the domain name which is used to distinguish from libs from other companies;

appcompat is the project name which is used to differentiate from the libs in the same company;

1.1.0 is the version number that can differentiate the same lib with different versions.

After adding this, Gradle will check if cache of this lib exists locally,

if not, then it will download the lib and add to the build path.

Android Emulators

A virtual device that allows developers to run the app they are developing without a physical device on hand.

Test Framework

A project created by Android Studio is automatically configured for test automation.

In addition
to the “main” package that holds the application logic of the app,

Android Studio creates
two additional packages designed specifically for testing.

Assuming “main” package is edu.cis.gvsu.myapp,
the other two packages for the test code are:

- edu.cis.gvsu.myapp.test: test code that runs on local JVM that requires no Android dependency.
- edu.cis.gvsu.myapp.androidTest: test code that requires an Android device to run.

This is also known as **Instrumentation Test**.

Debuggers

Seasoned developers know how to use code debuggers by setting up breakpoints, using the debugger common toolbar icons (Step Over, Step Into, Step Out, Step Return), using expression watchers and variable inspections.

In addition to these common tasks in a typical debugging sessions, Android Studio also provides features which may not be frequently used by most developers:

• Conditional breakpoints: a breakpoint that becomes active only when its associated Boolean expression evaluates to true (at runtime).
To attach a condition, right-click on the breakpoint marker on the left gutter of the editor to display the screen in Figure 1.8.
• Logged breakpoints: a breakpoint that also prints debugging output to Android Logcat.

This is a convenient way of logging messages without having to recompile code.

• Attach the debugger to an active process: a feature that is useful when we need to reinspect the state of our app at the time (or after) a bug occurs. Activate this feature from Run □Attach to Process

• Smart Step Into (Run Smart Step Into or Up arrow + F7) is a variant of Step Into with an additional feature for stepping into a **selected method**, making the overall stepping process faster and more convenient

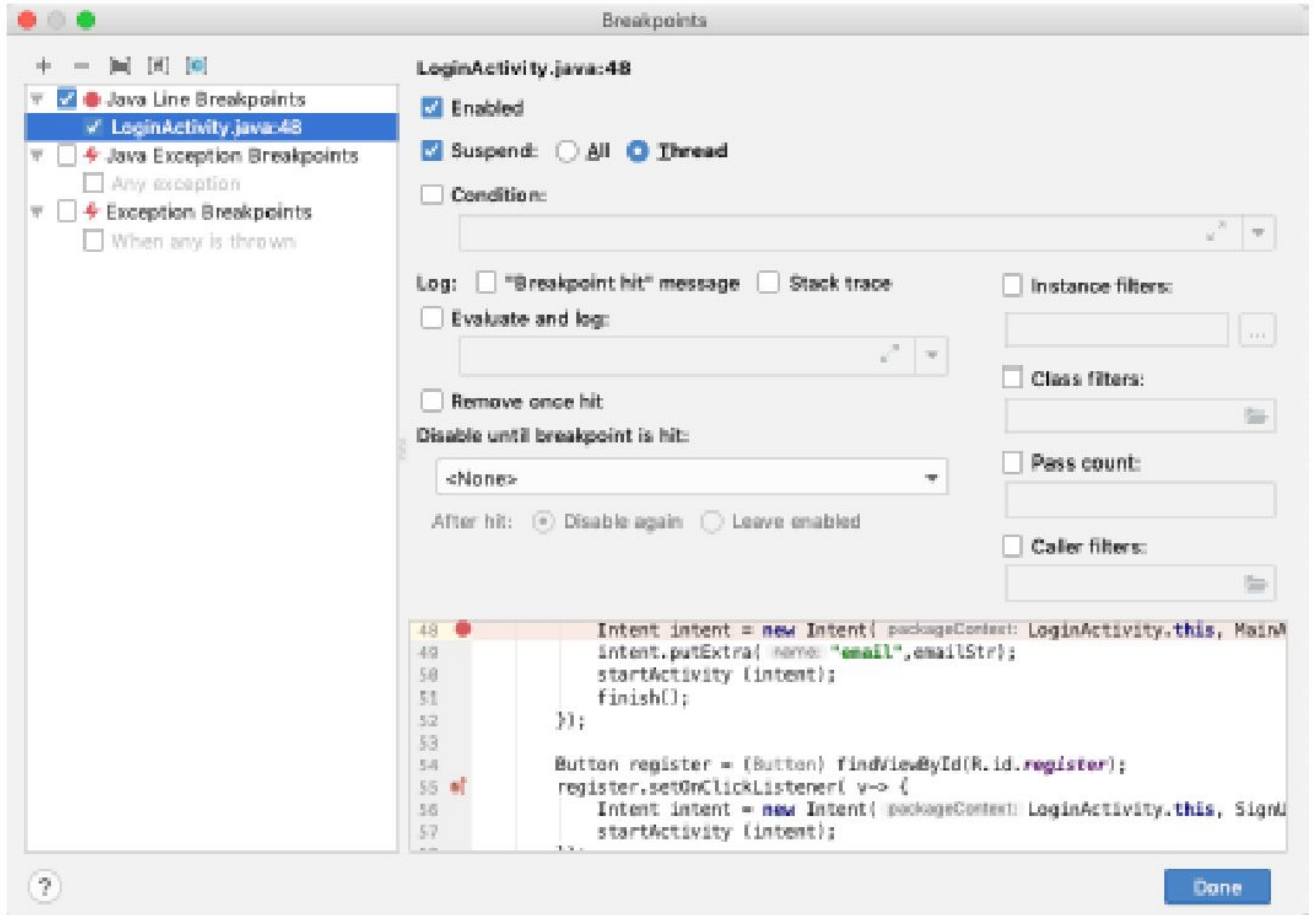


Figure 1.8: Debugger breakpoints

Managing Libraries

Libraries are now inseparable components in Android development.
Most of these libraries enhance the features provided by the core Android framework.

The naming convention of these libraries follow the guidelines specified by the Apache Maven Project.

The name consists of three parts:
group id,
artifact id, and
version number.

Library Name - com.android.support:design:28.0.0

its group id is com.android.support,
artifact id is design,
and currently at version 28.0.0

Android Support Libraries

The support libraries had been the main venue for the Android team to add new features

while at the same time maintaining backwards compatibility with older API levels.

The ”vx” infix (”v4”, ”v7”, ”v8”, etc.) in the library name indicates the lowest API level supported by the library.
For instance com.android.support:cardview-v7:25.0.0 is a library with backwards compatibility
to API Level 7.

Prior to API level 28, Android developers had to include one or more Android Support Libraries.
These libraries are under the group id com.android.support;

API level 28 the support libraries are now called AndroidX which is part of Jetpack

Thank You