Building Mobile Applications

CS 346 Application Development

Steve Jobs, Apple CEO, introduces the original iPhone at Macworld San Francisco in 2007.

Smartphone Design

- Smartphones as personal, portable computing.
- Originally a mashup of other devices.
  - “An iPod, a phone, an internet communicator”.
  - Mobile phone category evolved over 2-3 years.
- What makes them unique?
  - Touch-screens! Touch input, customizable output.
  - Optimized for simple, ad hoc interaction.
  - A single device for all your needs (data).
- Design concerns
  - Processing efficiency, battery life.
  - Security! Applications needed to be sandboxed.

The first iPhone, introduced in Jan 2007, and available for sale in June of that year. Apple sold more than 6 million phones before replacing this model with the iPhone 3G in 2008.

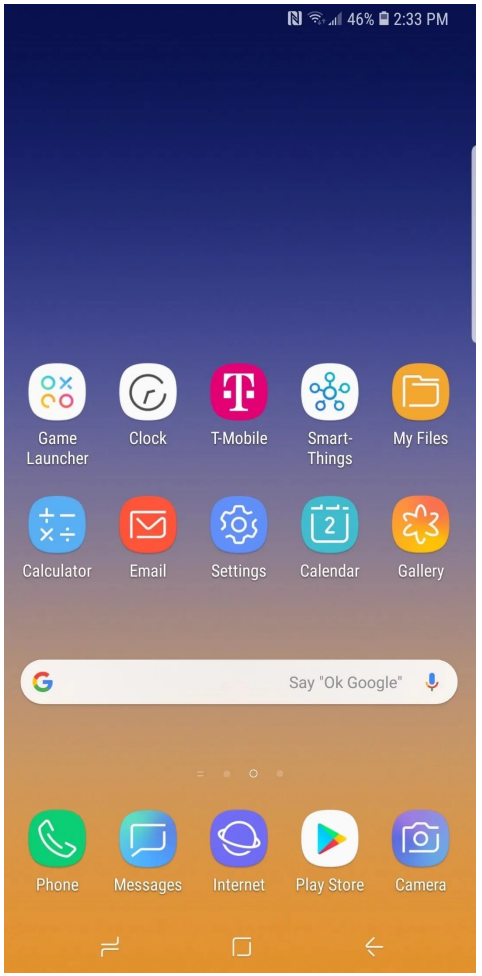
Android

- History of Android
  - Founded by Andy Rubin in 2003 to build a camera OS.
  - Pivoted to phone OS in 2004, sold to Google in 2005.
  - By Dec 2006 Google was testing phones w. keyboards.
  - Redesigned for touch-screens before phones launched.
- Android is the world's "most popular OS".
  - Based on Linux kernel; portions are open source.
  - Ships on different devices e.g., TV boxes, tablets, phones.
  - "Billions of Android devices".
- Features
  - Comparable features to iPhone.
  - Tight integration with Google services.

The first Android phone was the HTC Dream, which launched in October 2008 – approximately 18 months after the first iPhone.

Android Features

- Graphical User Interface
  - Applications presented as pages of icons.
  - An application usually runs full-screen.
    - Forward/backward screen navigation within an application.
    - Navigate through running applications.
  - Custom UI displays
    - Side-by-side applications, Live-regions
- Tight integration with Google applications
  - Gmail, Google docs, other services.
  - Google search, “Ok Google” voice chat.
- Wider range of hardware
  - Many vendors, who produce a wider range of devices.



Getting Started

How to create an Android project.

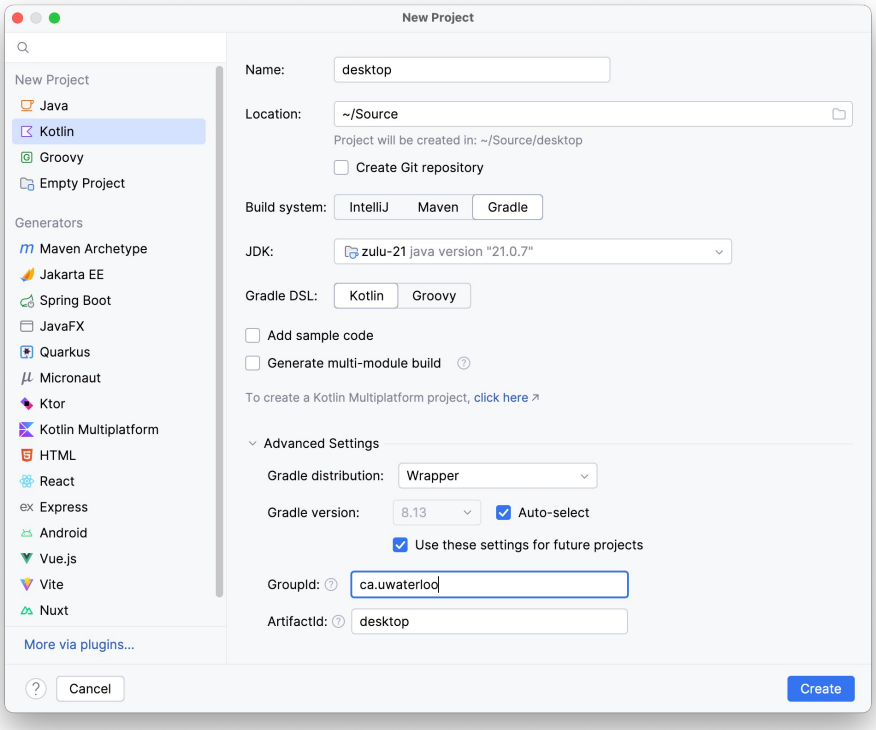
Step 1: Create a Project

An Android project is simply a Gradle project with specific dependencies.

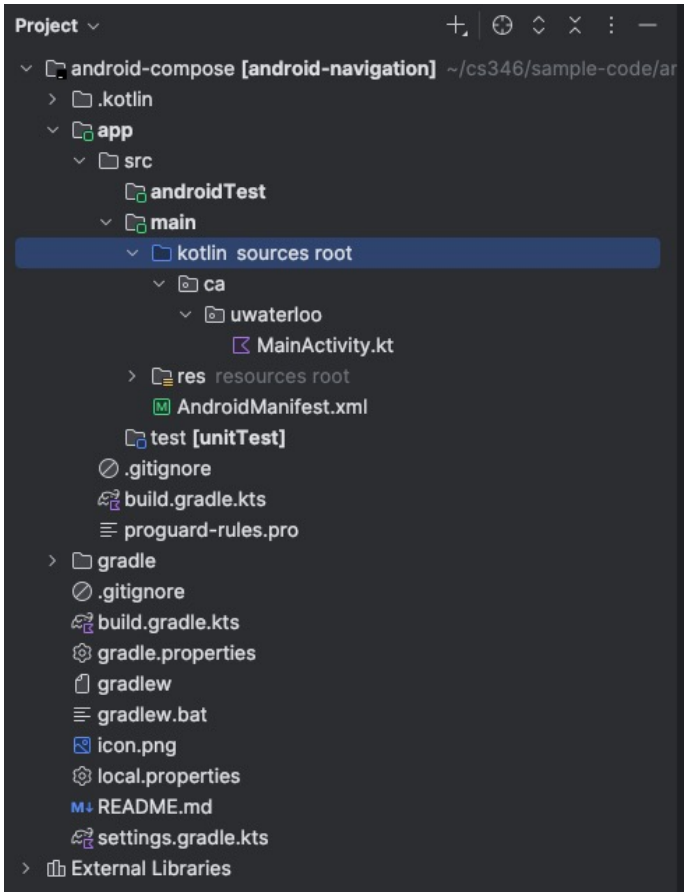
- Requires an IDE with the Android plugin installed.
- IntelliJ IDEA or Android Studio are both fine.

See course website:

Reference > Getting Started > Gradle project



Step 2: Check the directory structure



Unlike a desktop project, an Android project should be completely usable after you walk through the creation wizard!

Do NOT modify your starting project structure.

Differences

- There are `androidTest` and `Test` folders for unit tests (see later slides).
- `AndroidManifest.xml` in `src/main`.
- Resources under `main/res` folder.
- Top-level source file is `MainActivity.kt`.

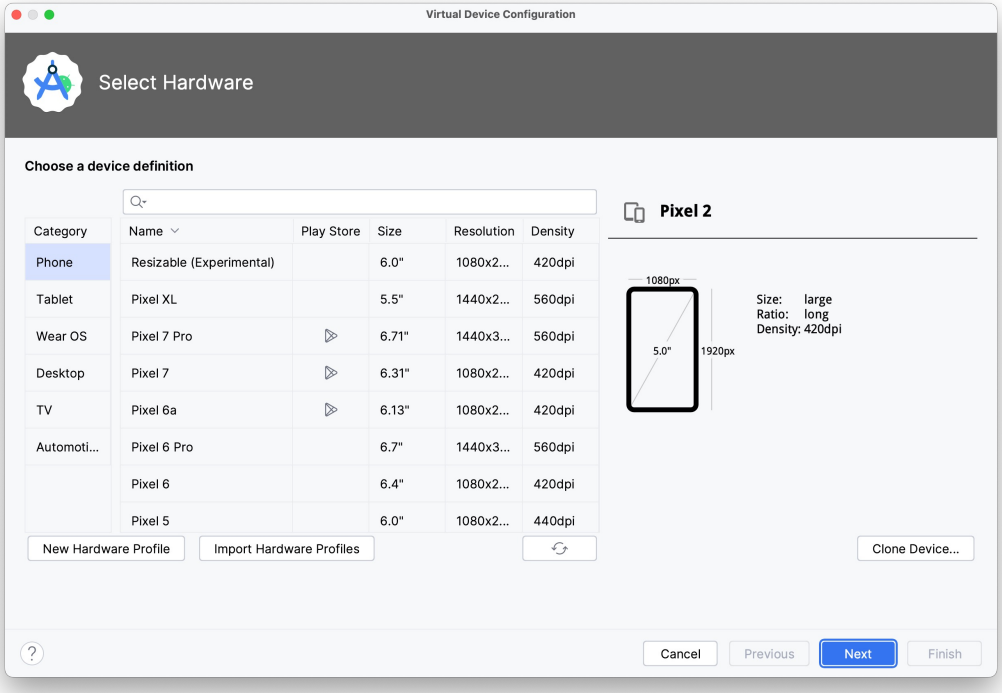
Step 3: Dependencies

```
[versions]
agp = "8.10.1"
kotlin = "2.0.0"
coreKtx = "1.15.0"
junit = "4.13.2"
junitVersion = "1.2.1"
espressoCore = "3.6.1"
lifecycleRuntimeKtx = "2.8.7"
activityCompose = "1.9.3"
composeBom = "2024.10.01"
composeNavigation = "2.8.3"
serialization = "1.7.2"

[libraries]
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx" }
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit", version.ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso", name = "espresso-core", version.ref = "espressoCore" }
androidx-lifecycle-runtime-ktx = { group = "androidx.lifecycle", name = "lifecycle-runtime-ktx", version.ref =
"lifecycleRuntimeKtx" }
androidx-activity-compose = { group = "androidx.activity", name = "activity-compose", version.ref = "activityCompose" }
androidx-compose-bom = { group = "androidx.compose", name = "compose-bom", version.ref = "composeBom" }
androidx-ui = { group = "androidx.compose.ui", name = "ui" }
androidx-ui-graphics = { group = "androidx.compose.ui", name = "ui-graphics" }
androidx-ui-tooling = { group = "androidx.compose.ui", name = "ui-tooling" }
androidx-ui-tooling-preview = { group = "androidx.compose.ui", name = "ui-tooling-preview" }
androidx-ui-test-manifest = { group = "androidx.compose.ui", name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-test-junit4" }
androidx-material3 = { group = "androidx.compose.material3", name = "material3" }
navigation-compose = { module = "androidx.navigation:navigation-compose", version.ref = "composeNavigation" }
kotlinx-serialization-json = { module = "org.jetbrains.kotlinx:kotlinx-serialization-json", version.ref = "serialization"}
```

- You will have a large number of starting dependencies!
- Add more as needed through the version catalog.

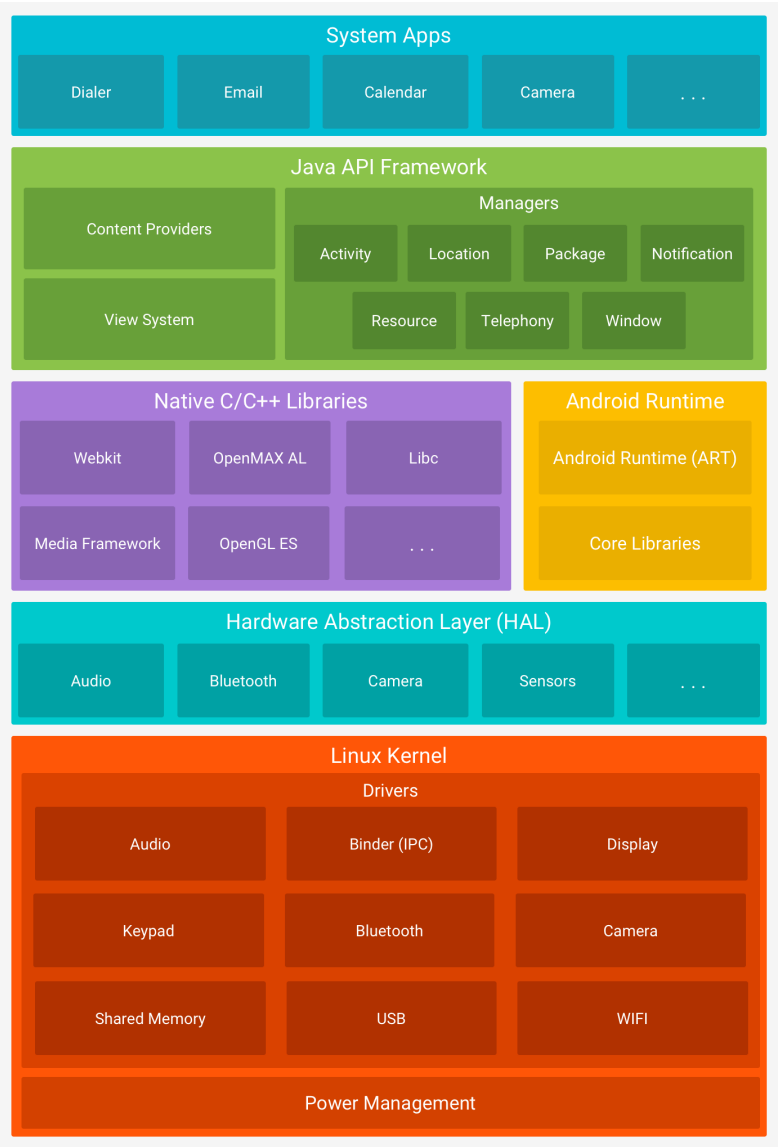
Android Device Manager



Tools > Android > Android Device Manager

Architecture

How is Android designed?



This is the Android operating system stack.

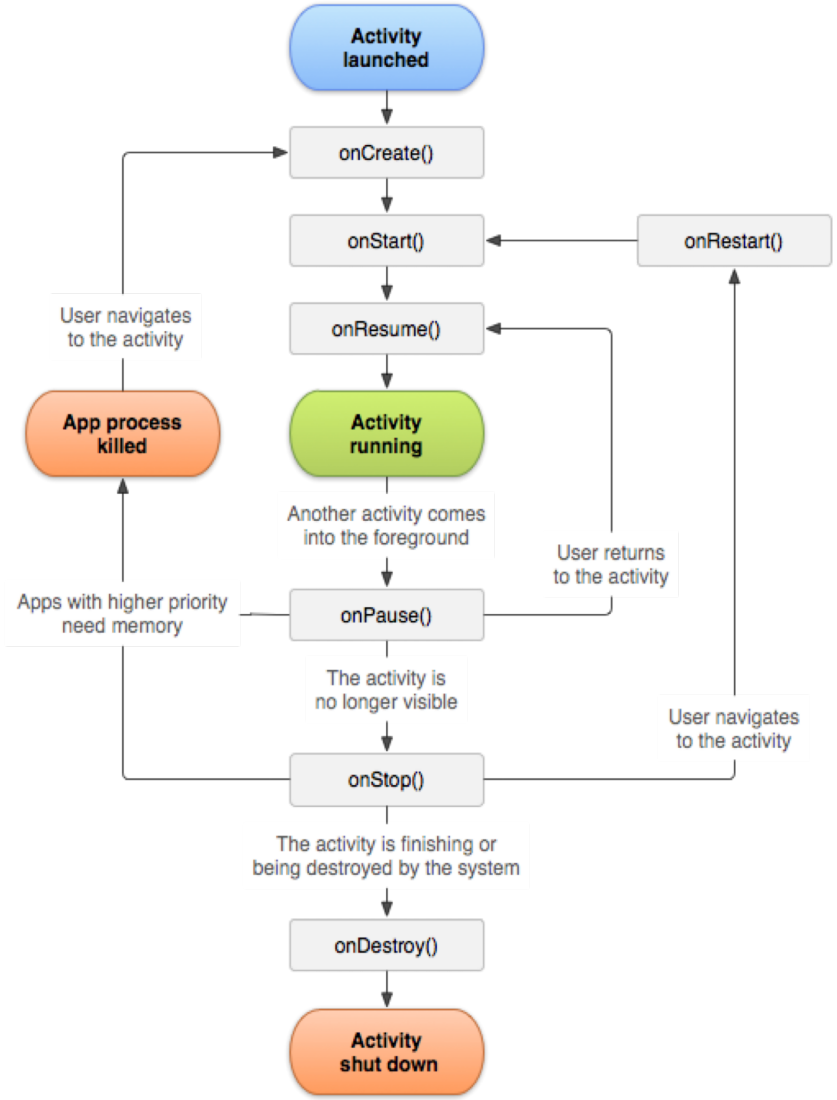
This is a layered architecture: hardware at the bottom, user applications at the top.

- **System Applications**: Applications that are bundled with the OS or written (by us). They leverage the next layer & cannot communicate directly with anything further down the stack.
- **Java API Framework**: Google repurposed some Java libraries to provide services to the OS. Recently, portions have been rewritten in Kotlin.
- **Native C++ Libraries**: high-performance libraries that the higher-level frameworks leverage e.g., graphics, media.
- **HAL/Linux**: Lowest level drivers, hardware access.

Application Design

A typical Android application contains multiple components, including some combination of:

Component	Description
Activities	Screens, each with its own state and lifecycle
Fragments	Portions of a screen that can be managed separately
Services	Provides long-running operations in the background
Content Providers	Shares data with other applications.
Broadcast Receivers	Listens for system events e.g., phone call, airplane mode



Key takeaway: your application needs to support being paused or stopped (typically by saving data for later).

Activity Details

- Applications consist of one or more running activities, each one corresponding to a screen.
- You can think of an activity as a visible screen with state information.
- An activity can be one of the following running states:
  - The activity in the **foreground**, typically the one that user is interacting with, i.e., running.
  - An activity that has **lost focus** but can still be seen is visible and active.
  - An activity that is completely **hidden, or minimized** is stopped. It retains its state (it's basically paused) BUT the OS may choose to terminate it to free up resources.
  - The OS can choose to **destroy** an application to free up resources.

Activity Lifecycle

There are three key loops that these phases attempt to capture:

- **The entire lifetime of an activity** happens between the first call to `onCreate(Bundle)` through to a single final call to `onDestroy()`. Setup is done in `onCreate()`, and all remaining resources are released by `onDestroy()`.
- **The visible lifetime of an activity** happens between a call to `onStart()` until a corresponding call to `onStop()`. During this time the user can see the activity on-screen, though it may not be in the foreground.
- **The foreground lifetime of an activity** happens between a call to `onResume()` until a corresponding call to `onPause()`. During this time the activity is in visible, active and interacting with the user. An activity can frequently go between the resumed and paused states e.g. when the device goes to sleep.

Warning: Data loss on rotation

- Activities can be restarted when
  - The OS decides that it needs to reclaim resources (*uncommon*),
  - You rotate the device (*common!*)
- Restarting activities means relaunching and losing data.
- How do you avoid this?
  - Save and restore data manually
    - Override the onPause() and onResume() methods and manage a Bundle of data.
  - Use a ViewModel as a base class for your custom ViewModel.
    - Android will automatically save and restore VM data!!
    - https://developer.android.com/topic/libraries/architecture/viewmodel

Application Structure

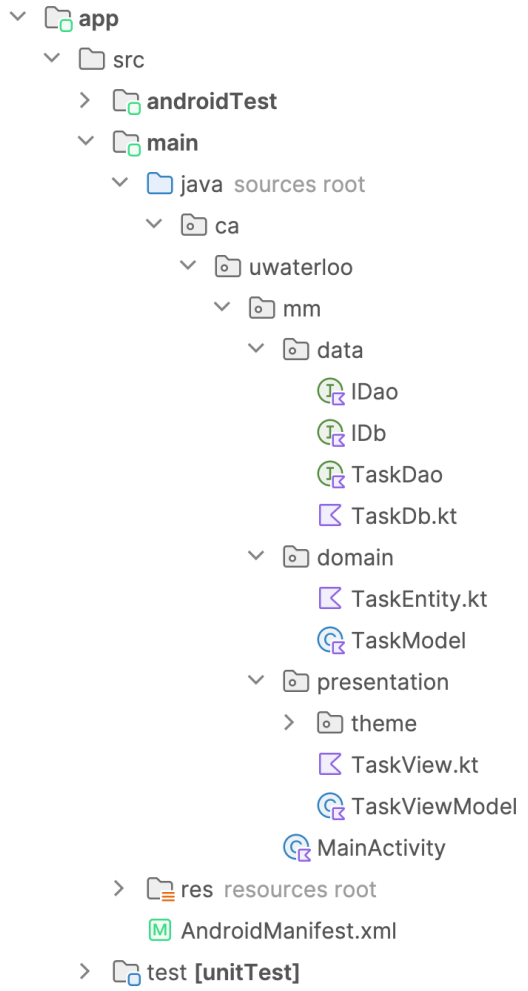
What does it look like again?

Project Structure

Your application structure should look the same as discussed in the Architecture lecture, with data/, domain/ and presentation/ layers.

Differences compared to a desktop application:

- Your entry point is the `MainActivity` class.
- Android stores resources in the `res` folder structure. There is an API to load them.
- `Manifest` file describes your project structure.



MainActivity

MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()

        val database= getRoomDatabase(this)
        val taskModel = TaskModel(database.taskDao())
        val viewModel = TaskViewModel(taskModel)

        setContent {
            MMTheme {
                TaskView(viewModel) // top-level View/Composable
            }
        }
    }
}
```

MainActivity is a class that extends ComponentActivity.

Activities have built-in methods that mirror their lifecycle: onCreate(), onStart(), onStop() and so on.

The onCreate() method is the first method that is called when the MainActivity is instantiated and serves as the entry point for your application.

GitHub: demos > mm-android

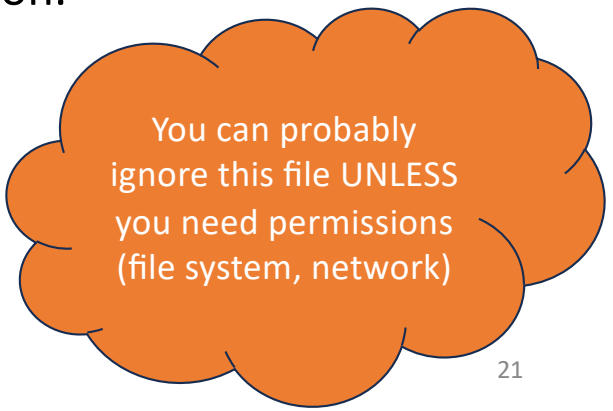
Application Manifest

Every Android project has a single `AndroidManifest.xml` file

This is an XML file that describes your application structure.

- It lists components and properties required to compile, install and run your application. e.g.,
  - Identifies the `MainActivity` which launches on startup i.e., `main` method.
  - Identifies the name and icon to use for your application.
  - Location of resources to include.
  - Permissions that the application requires

- See Application Manifest Overview



AndroidManifest.xml

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

    <application
            android:allowBackup="true"
            android:dataExtractionRules="@xml/data_extraction_rules"
            android:fullBackupContent="@xml/backup_rules"
            android:icon="@mipmap/ic_launcher"
            android:label="@string/app_name"
            android:roundIcon="@mipmap/ic_launcher_round"
            android:supportsRtl="true"
            android:theme="@style/Theme.Mmandroid"
            tools:targetApi="31">
        <activity
                android:name="ca.uwaterloo.mm.MainActivity"
                android:exported="true"
                android:label="@string/app_name"
                android:theme="@style/Theme.Mmandroid">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

GitHub: demos > mm-android

Activity: View relationship

We use View classes for screens. How do they relate to Activities?

1. Each View is an Activity (early-Android).
   - Every screen is represented by a corresponding Activity.
   - You would use an Intent (message to the OS) to swap between them.
   - This is not recommended! It's very slow. ✗
2. You fewer Activities, each is configurable using Fragments (old Android)
   - You have few Activities, but each one is composed of pieces called Fragments.
   - You write logic to load the Activity, then load suitable fragments.
   - Not recommended! Faster, but still generally very slow. ✗
3. One Activity, and you just choose your View to show (new Android)
   - Use your MainActivity as a container. Each view is a single top-level composable!
   - Navigation code/libraries just chooses which View to launch. ✓

presentation/
TaskView.kt

```
@Composable
fun TaskView(viewModel: TaskViewModel) {
    val items by viewModel.getAll().collectAsState(initial = emptyList())

    Scaffold(
        topBar = {
            Toolbar(
                addHandler = { viewModel.showAddDialog = true },
                editHandler = { viewModel.showEditDialog = true },
                deleteHandler = {
                    val task = viewModel.selectedTask ?: return@Toolbar
                    viewModel.delete(task)
                    viewModel.selectedTask = null
                }
            )
        },
        bottomBar = {  },
    ) { padding ->
        Box(
            modifier = Modifier.fillMaxSize().padding(padding)
        ) {
            if (items.isEmpty() && !viewModel.showAddDialog && !viewModel.showEditDialog) {
                Text(
                    "No tasks available. Add a task using the + button.",
                    modifier = Modifier.align(Alignment.Center).padding(16.dp)
                )
            } else {
              // …
            }
```

The presentation layer communicates with the domain layer. i.e. TaskViewModel and TaskEntity classes.

None of this is Android-specific; it's straight Compose code.

GitHub: demos > mm-android

```
/*
 * Android ViewModel
 * This class holds state for our Application Composable function.
 * The built-in ViewModel survives screen rotation automatically.
 */

class TaskViewModel(val taskModel: TaskModel) : ViewModel() {
    var selectedTask by mutableStateOf<Task?>(null)
    var showAddDialog by mutableStateOf(false)
    var showEditDialog by mutableStateOf(false)

    fun getAll(): Flow<List<Task>> {
        return taskModel.getAll()
    }

    fun getById(id: Int): Task {
        return runBlocking {
            taskModel.getById(id)
        }
    }

fun deleteAll() {
    viewModelScope.launch {
        taskModel.deleteAll()
    }
}

// …
```

GitHub: demos > mm-android

```
domain/
  TaskViewModel.kt
```

The domain layer communicates with the data layer.

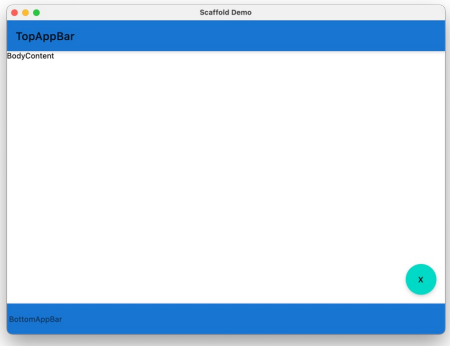
None of this code is Android specific.

We'll review the application in more detail in the database lecture.

Android-Specific Composables

What Compose functionality is specific to mobile development?

Composable: Scaffold



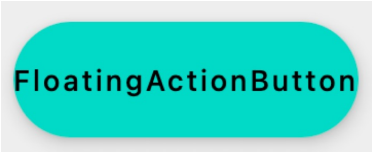

```
@Composable
fun ScaffoldDemo() {
    val materialBlue700= Color(0xFF1976D2)
    val scaffoldState = rememberScaffoldState(rememberDrawerState(DrawerValue.Open))
    Scaffold(
          scaffoldState = scaffoldState,
          topBar = {
              TopAppBar(title = {Text(“TopAppBar")}, backgroundColor = materialBlue700)
          },
          floatingActionButtonPosition = FabPosition.End,
          floatingActionButton = { FloatingActionButton(onClick = {}){Text(“X”)} },
          drawerContent = { Text(text = "drawerContent") },
          content = { Text("BodyContent") },
          bottomBar = {
              BottomAppBar(backgroundColor = materialBlue700) {Text(“BottomAppBar”)}
          }
    )
}
```

Composable: Image

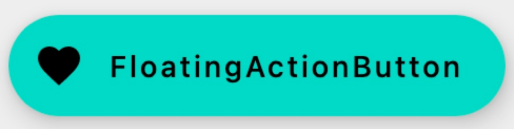
```
@Composable
fun ImageResourceDemo() {
    val image: Painter = painterResource(id = R.drawable.composelogo)
    Image(painter = image,contentDescription = "")
}
```

Composable: Floating Action Buttons

```
@Composable
fun FloatingActionButtonDemo() {
    FloatingActionButton(onClick = { /*do something*/}) {
        Text("FloatingActionButton")
    }
}
```

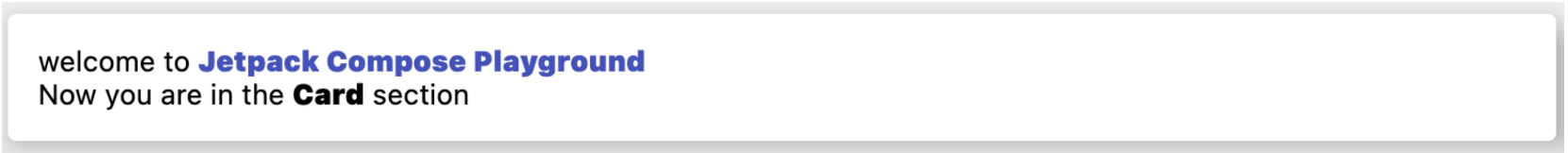



```
@Composable
fun ExtendedFloatingActionButtonDemo() {
    ExtendedFloatingActionButton(
        icon = { Icon(Icons.Filled.Favorite,"") },
        text = { Text("FloatingActionButton") },
        onClick = { /*do something*/ },
        elevation = FloatingActionButtonDefaults.elevation(8.dp)
    )
}
```



Composable: Card

```
@Composable
fun CardDemo() {
    Card(
        modifier = Modifier.fillMaxWidth().padding(15.dp).clickable{ },
        elevation = 10.dp
    ) {
        Column(modifier = Modifier.padding(15.dp)) {
            Text("Jetpack Compose Playground")
            Text("Now you are in the Card section")
       }
    }
}
```



Finding More Composables

All of the other composables work as well! The amazing thing about Compose is that you can copy/paste composables between platforms.

List of Composables
https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary

Sample Code
https://foso.github.io/Jetpack-Compose-Playground/
https://developer.android.com/jetpack/compose/components

Managing State

Android-specific issues.

What is unique about Android?

The OS has control over applications at a deep level.

- Application components only communicate through the OS via intents.
- The OS can launch and control specific application components.
  - e.g., Your application can use a Photo Capture screen from a different application.

- The OS was designed around devices with very limited resources.
  - Rotating the device will cause the UI to be reloaded. ⚠
    - Pre-compose? The UI was completely reloaded, and UI state is lost.
    - Compose? This forces recomposition.
  - The OS may terminate your application if it needs resources. ⚠
    - You need to handle this as well, otherwise you will lose data!

Managing Compose State

```
@Composable
fun ChatBubble(
  message: Message
) {
  var showDetails by rememberSaveable { mutableStateOf(false) }

  ClickableText(
       text = AnnotatedString(message.content),
       onClick = { showDetails = !showDetails }
  )

  if (showDetails) {
       Text(message.timestamp)
  }
}
```

Caveats

`rememberSaveable` stores data in a Bundle

- this is a special Android specific data structure to hold values.
- It only works for primitives!

To store anything more complex, you may need additional APIs.

- e.g., making a class Parcelable.
- See Ways to store state

Navigation

Android-specific navigation.

Jetpack Navigation Concepts

- A **navigation graph** describes all of the possible screen destinations and connections between then.
- A **destination** is a node that you can navigate to. This can be a composable (screen), or a dialog, or a different navigation graph (for complex user interfaces).
- A **route** identifies a destination and defines how to navigate to it.

Jetpack Navigation Library

The Navigation library represents the user's path as a stack of destinations. You can use this to move forward/backwards through navigation history.

Core classes:

- **NavController**: provides APIs for core functionality.
- **NavHost** is a composable that displays the contents for the current destination (determined by the navigation graph).
- **NavGraph** describes all possible destinations and the connections between them.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                val navController = rememberNavController()
                NavHost(
                    navController = navController,
                    startDestination = ScreenA
                ) {
                    composable<ScreenA> {
                        ScreenAView(navController)
                    }
                    composable<ScreenB> {
                        val args = it.toRoute<ScreenB>()
                        ScreenBView(navController, args)
                    }
                    composable<ScreenC> {
                        val args = it.toRoute<ScreenC>()
                        ScreenCView(navController, args)
                    }
                }
            }
        }
    }
}
```

See: GitHub > sample-code > android-navigation

Interactivity

Handling screen events, key presses.

Interaction Styles

What types of interaction do we need to support on a mobile device?

1. Multi-touch for primary input.
    - Tapping on widgets to activate e.g. touch a text widget to enter text; touch a button to activate it.
    - Dragging and other gestures.
2. Keyboard input as secondary.
    - Soft-keyboard (on-screen).

Multi-touch Widgets

This is *exactly* the same as desktop. You override the handler functions for the widgets, providing it with a lambda function that is executed when the event fires.

```
FloatingActionButton(onClick = { /* something */ }) {
    Text("FloatingActionButton")
}
```

Touch Gestures

You can apply gesture modifiers to make the composable listen to gestures.

```
var log by remember { mutableStateOf("") }
Column {
   Box(
        Modifier
            .size(100.dp)
            .background(Color.Red)
            .pointerInput(Unit) {
                detectTapGestures { log = "Tap!" }
                detectDragGestures { _, _ -> log = "Dragging" }
            }
    )
}
```

Key Gestures

```
@Composable
fun SimpleFilledTextFieldSample() {
    var text by remember { mutableStateOf("Hello") }

    TextField(
        value = text,
        onValueChange = { text = it },
        label = { Text("Label") }
    )
}

@Composable
fun SimpleOutlinedTextFieldSample() {
    var text by remember { mutableStateOf("") }

    OutlinedTextField(
        value = text,
        onValueChange = { text = it },
        label = { Text("Label") }
    )
}
```

Testing

What's different about testing on Android?

What's unique?

Desktop testing: Development and deployment systems are the same.

Android testing: development and deployment hardware are different.

- `src/test` – local unit tests that run on your computer. Android not required. Should be used for generic/simple unit tests.
- `src/androidTest` – run on an Android device (or VM) using that hardware. Can test Android specific APIs and functionality.

You can use both folders but be aware that the tests execute in different environments.

Reference

- Google. 2025. Android Developer Portal.
- Google. 2025. Compose Lifecycle.
- Google. 2025. Guide to App Architecture.
- Google. 2025. State and Jetpack Compose.