Mobile Development

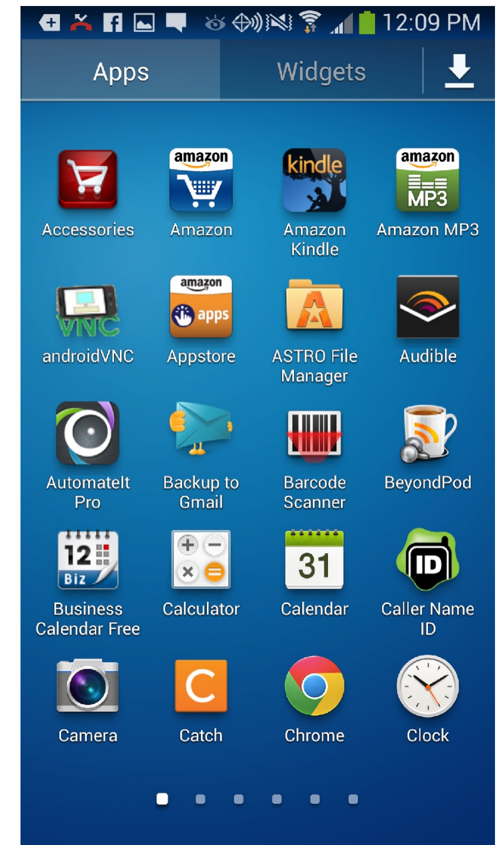
CS 346: Application Development

Features

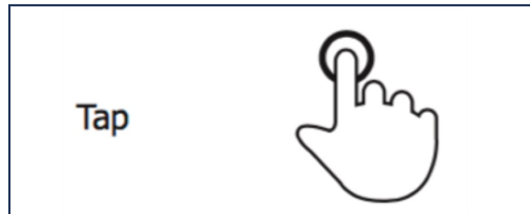
- Smartphones are portable, designed for ad hoc interaction**.**
- Interaction is **touch-based**. Keyboard input is secondary to touch. No assumption of physical buttons.
- Designed around a **single foreground application**, running **full-screen** (not resizable or movable).
- Restrictive environment:
  - Limited memory.
  - Slow CPU/IO.
  - Applications paused when not in the foreground.

Interactive Graphical Elements

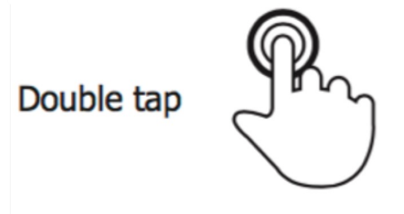
- Window contents are a combination of text, images, and interactive elements.
- Mobile applications tend to have fewer controls or on-screen widgets compared to desktop.
- Interaction is typically by gestures (touch and swipe on regions of the screen). Direct manipulation is emphasized.
- Challenges?
  - Screen size and
  - Difficulty interacting with small elements by touch.

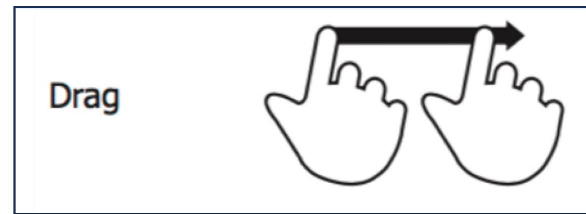


Click



Right-click





Drag

We use a relatively small number of gestures to interact with our phones. The highlighted ones loosely correspond to mouse events on desktop.

Toolkits

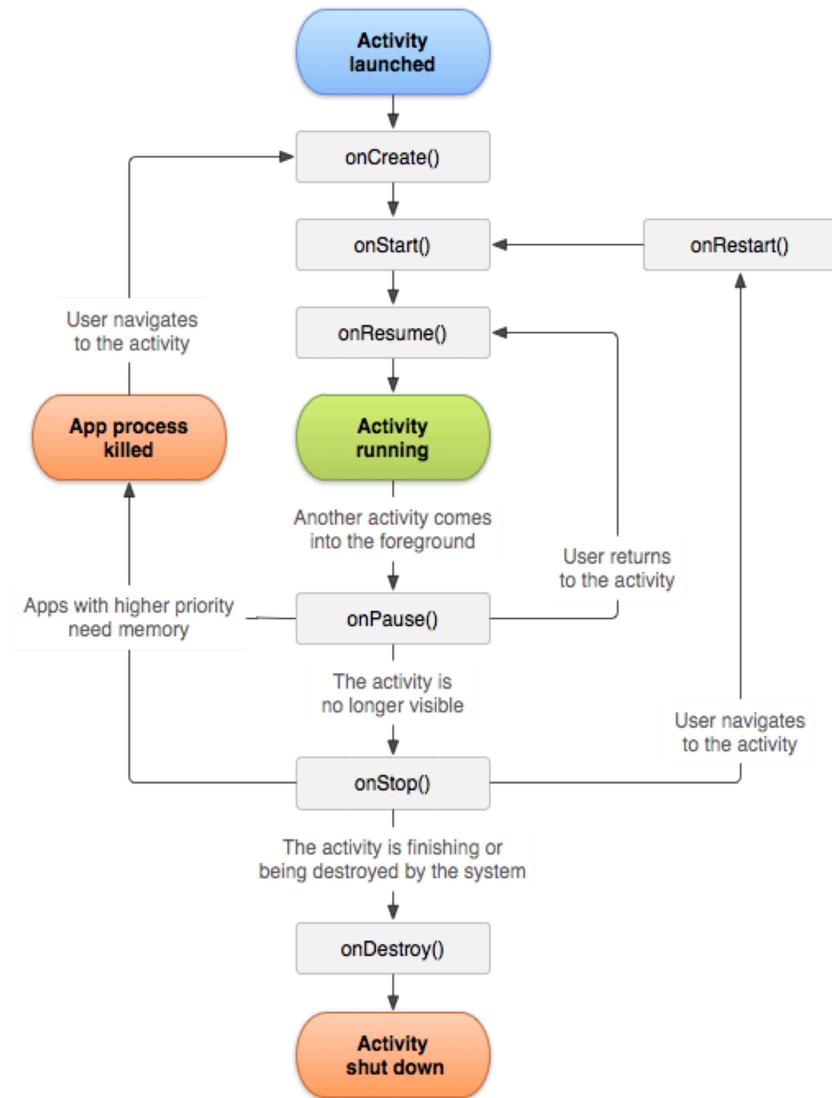
- In a desktop OS, we might have a widget or GUI toolkit to provide advanced features for building applications (e.g. creating and managing application windows, providing reusable widgets like buttons, lists, toolbars.
- Android has two toolkits, both provided by Google:
  - **Android Views/XML**: the original imperative toolkit (*deprecated*).
  - **Jetpack Compose**: a declarative toolkit, recommended.
    - This is the same toolkit we discussed last class.

Structure

How is a program structured?

Activities

- 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.



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

Activity Lifecycle

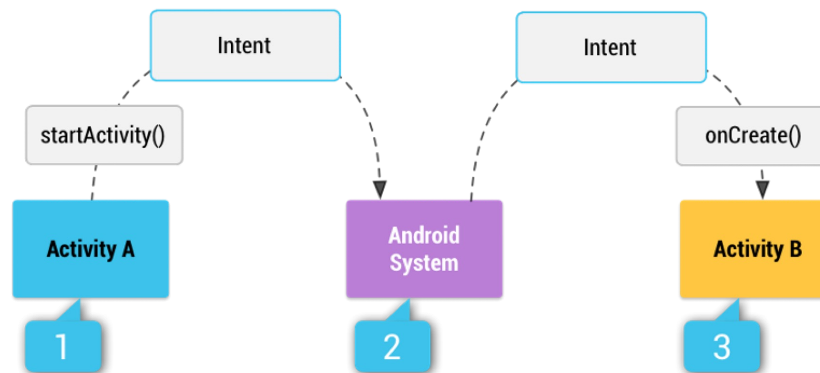
There are three key loops that these phases attempt to capture:

- **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.

Intents

An **intent** is an asynchronous message, that represents an an operation to be performed. This can include activating components, or activities.

- The `startActivity(Intent)` method can be used to **start a new activity**. It takes a single argument, an Intent, which describes the activity to be executed. This is the simplest way to support multiple screens.

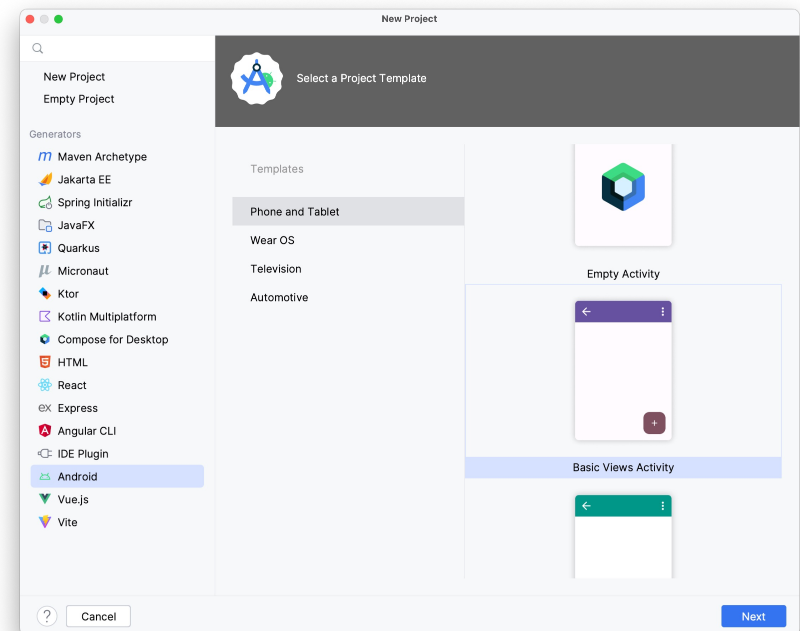


Jetpack Compose on Android

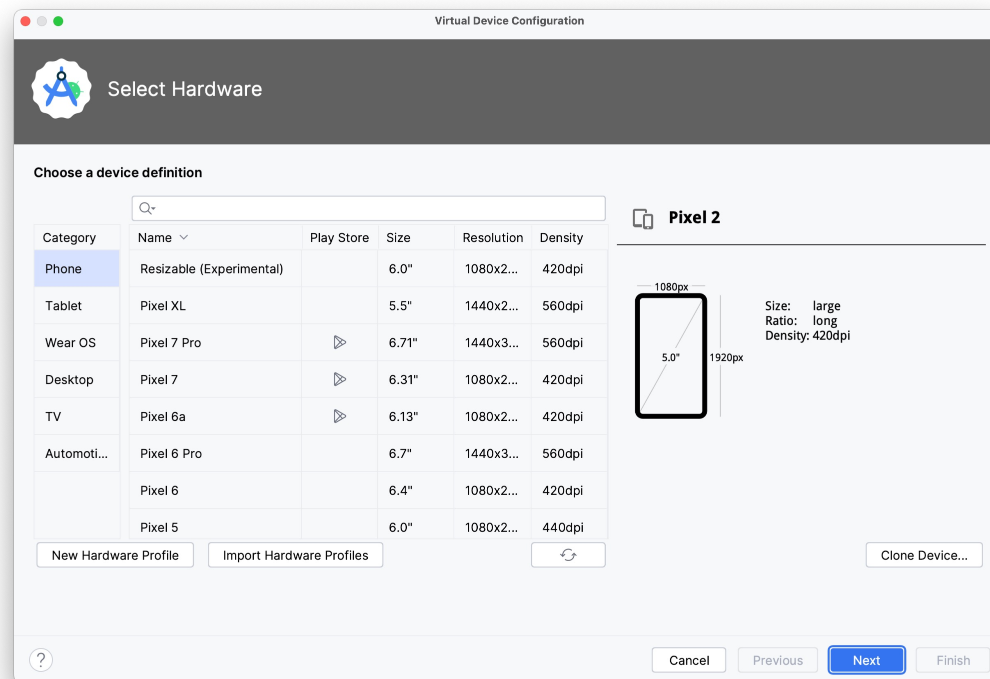
Using the GUI toolkit for mobile development.

Creating a Project

- Android uses Gradle projects. You can create an Android project in **IntelliJ IDEA** or **Android Studio.**
  - Make sure that you and your team use the same IDE (and plugins).
- Most project templates will create a simple project with one activity.
  - Try “Basic Views” for a simple Activity.
- The project structure that is created will match a standard Android project, with source and res folders.



Android Device Manager



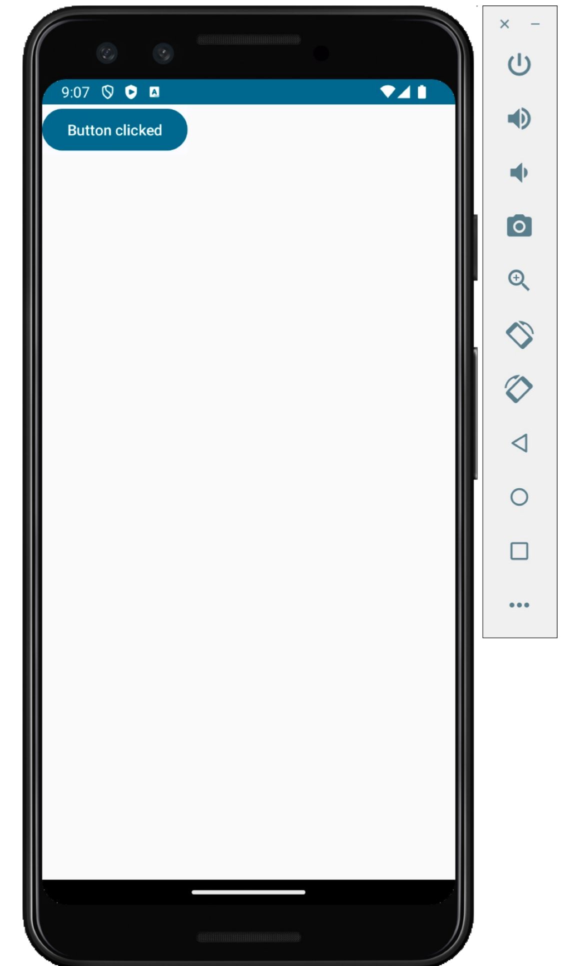
Tools > Android > Android Device Manager

Main Activity (Entry Point)

- `MainActivity` is a class that extends `ComponentActivity`. It's the default name given to the activity that gets launched on application startup.
- The `onCreate()` method is the first method that is called when the MainActivity is instantiated and serves as the entry point for your application.

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      // Compose user interface goes here
    }
  }
```

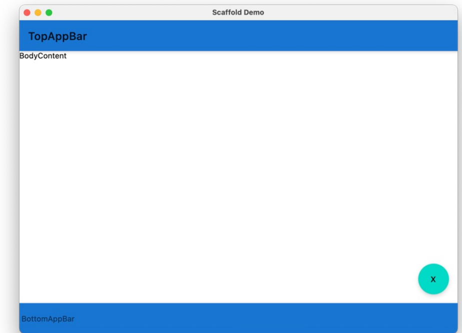
```kotlin
// an activity is essentially a screen
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface() {
                    ButtonExample()
                }
            }
        }
    }
}


@Composable
fun ButtonExample() {
    var text = remember { mutableStateOf("Hello world!")}
    Button(onClick = { text.value = "Button clicked"}) {
        Text(text = text.value)
    }
}
```



Android-Specific Composables

What Compose functionality is specific to mobile development?

Composable: Scaffold



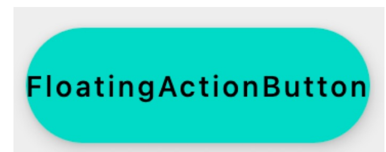

```
@Composable
fun ScaffoldDemo() {
    val materialBlue700= Color(0xFF1976D2)
    val scaffoldState = rememberScaffoldState(rememberDrawerState(DrawerValue.Open))
    Scaffold(
          scaffoldState = scaffoldState,
          topBar = {
              TopAppBar(title = {Text("TopAppBar")}, backgroundColor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

Composable: Image

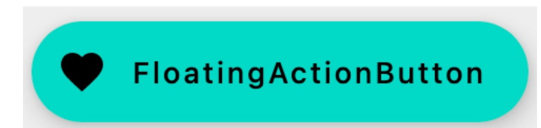
```
@Composable
fun ImageResourceDemo() {
    val image: Painter = painterResource(id = R.drawable.composelogo)
    Image(painter = image,contentDescription = "")
}
```

Composable: Floating Action Buttons

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




```kotlin
@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")
       }
    }
}
```

welcome to **Jetpack Compose Playground**
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

Navigation

CS 346: Application Development

What is navigation?

- Navigation refers to **switching screens**.
- Intents allow you to switch between Activities. What if you want to switch composables?
  - Navigation components support Compose.
  - Limited to Android only.
- Alternatives
  - Voyager – works on Android, iOS, desktop.
  - Simpler to setup and use.

```
class HomeScreen : Screen {

    @Composable
    override fun Content() {
        val screenModel = rememberScreenModel ()
        // ...
    }
}

class SingleActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            Navigator(HomeScreen())
        }
    }
}
```

Voyager code to load a composable screen.

Jetpack Navigation

The Navigation component handles many navigation patterns, from button clicks to complex patterns like app bars and the nav drawer. It provides animations, deep linking, back-and-up actions.

Concept	Purpose	Type
Host	A UI element that contains the current navigation destination. When a user navigates, the app swaps destinations in and out of the navigation host.	NavHost
Graph	A data structure that defines all the navigation destinations within the app and how they connect.	NavGraph
Controller	The central coordinator for managing navigation between destinations.	NavController
Destination	A node in the navigation graph. When the user navigates to this node, the host displays its content.	NavDestination
Route	Uniquely identifies a destination and any data required by it. You can navigate using routes. Routes take you to destinations.	

```
setContent {
    TypeSafeComposeNavigationTheme {
        val navController = rememberNavController()
        NavHost(
            navController = navController,
            startDestination = ScreenA
        ) {
            composable<ScreenA> {
                Column(
                    modifier = Modifier.fillMaxSize(),
                    verticalArrangement = Arrangement.Center,
                    horizontalAlignment = Alignment.CenterHorizontally
                ) {
                    Button(onClick = {
                        navController.navigate(ScreenB(
                            name = null,
                            age = 25
                        ))
                    }) {
                        Text(text = "Go to screen B")
                    }
                }
            }
```

```
            composable<ScreenB> {
                    val args = it.toRoute<ScreenB>()
                    Column(
                        modifier = Modifier.fillMaxSize(),
                        verticalArrangement = Arrangement.Center,
                        horizontalAlignment = Alignment.CenterHorizontally
                    ) {
                        Text(text = "${args.name}, ${args.age} years old")
                    }
                }
            }
        }
```

Interactivity

CS 346: Application Development

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") }
    )
}
```