ANDROID DEVELOPMENT

Fundamentals

Android Studio & Setup · Activities & Lifecycle

XML Layouts & Views · User Interaction

RecyclerView · Fragments

Navigation Component · Resources & Styling

Philip Osztromok

Generated with Claude

Table of Contents

Android Development Fundamentals — 8 Chapters

📱 Android Studio Setup & First App

Both Kotlin courses covered the language; this course covers building actual Android apps with it. This chapter is pure tooling and orientation — installing Android Studio, touring the IDE, understanding what a generated project actually contains, and running a first app on an emulator. Nothing here is Kotlin-specific yet; it's the ground everything else in this course stands on.

⬇️ Installing Android Studio

Android Studio is Google's official IDE for Android development, built on JetBrains' IntelliJ IDEA — the same underlying IDE platform used for pure Kotlin/Java development, with an entire layer of Android-specific tooling added on top (layout editor, emulator management, APK analysis, and more).

- Download from `developer.android.com/studio` — it bundles a compatible JDK, so a separate Java installation usually isn't required.
- The installer offers a "Standard" setup type on first run — this downloads the Android SDK, a default emulator system image, and standard build tools. Accept the defaults unless there's a specific reason not to.
- First launch takes a few minutes while the SDK and build tools finish downloading — this only happens once.

🆕 Creating Your First Project

From the Android Studio welcome screen, choose **New Project**, then pick a template. For this course, choose **Empty Views Activity** — not "Empty Activity," which defaults to Jetpack Compose (covered later, in Course 2). Starting with the traditional View/XML system first, before Compose, matches how this course builds up: Chapter 3 covers XML layouts directly.

Name & Package Name

The project name is just a label. The **package name** (e.g. `com.philip.myfirstapp`) is a reverse-domain identifier that must be globally unique if the app is ever published — it can't be easily changed later, so it's worth getting right from the start.

Minimum SDK

The oldest Android version the app will still run on, chosen from a dropdown showing the percentage of active devices each option covers. A higher minimum means fewer devices supported but access to newer APIs without extra compatibility checks.

🧭 A Tour of the IDE

Project View (left)

Browses the project's files — defaults to "Android" view, which groups files by purpose (manifests, java, res) rather than mirroring the raw folder structure exactly. Switch to "Project" view to see the real file layout on disk.

Editor (center)

Standard IntelliJ-style code editing — the same autocomplete, refactoring tools, and inspections as any JetBrains IDE, plus a live XML layout preview when editing a layout file.

Logcat (bottom)

The running app's log output in real time — every `println`-equivalent (`Log.d(...)`, covered next chapter), crashes, and system messages, filterable by app process and log level.

Device Manager

Where virtual devices (emulators) are created and managed — covered in its own section below.

📂 What a Generated Project Actually Contains

```
MyFirstApp/
├── app/
│   ├── manifests/
│   │   └── AndroidManifest.xml   # app metadata: package, permissions, activities, launcher icon
│   ├── java/
│   │   └── com/philip/myfirstapp/
│   │       └── MainActivity.kt   # your first Kotlin file — the app's entry-point screen
│   ├── res/
│   │   ├── layout/
│   │   │   └── activity_main.xml # the XML that defines what MainActivity's screen looks like
│   │   ├── values/
│   │   │   ├── strings.xml       # text content, kept separate from layout/code
│   │   │   ├── colors.xml        # named color values
│   │   │   └── themes.xml        # app-wide styling
│   │   └── drawable/             # images and vector icons
│   └── build.gradle.kts          # APP-level build config — dependencies, SDK versions
└── build.gradle.kts              # PROJECT-level build config — shared across all modules
```

`AndroidManifest.xml` is worth singling out: it's the file the Android OS itself reads to know what your app is — which screens (`Activity`s) exist, which one launches first, what permissions it needs, what its icon and name are. Nothing runs on a real device or emulator without a correct manifest.

⚠ res/ Files Are Referenced by Name, Not Imported

A string in `strings.xml` named `app_name` is referenced from Kotlin as `R.string.app_name` and from XML as `@string/app_name` — `R` is an auto-generated class mapping every resource to a stable ID. This indirection (separate text/color/dimension files instead of hardcoded values in code) is deliberate: it's what makes translations, dark mode, and different screen sizes possible without touching application logic.

🐘 Gradle Basics

Gradle is Android's build system — it compiles Kotlin/Java code, packages resources, resolves dependencies, and produces the final installable app package (an APK or AAB). The two `build.gradle.kts` files serve different scopes:

```
// app/build.gradle.kts — this module's specific configuration
android {
    namespace = "com.philip.myfirstapp"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.philip.myfirstapp"
        minSdk = 24
        targetSdk = 34
    }
}

dependencies {
    implementation("androidx.core:core-ktx:1.12.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
}
```

Gradle vs npm — A Familiar Shape

	npm (package.json)	Gradle (build.gradle.kts)
Declares dependencies	"dependencies": { "express": "^4.18.0" }	implementation("androidx.core:core-ktx:1.12.0")
Install/sync command	npm install	"Sync Now" (automatic on file changes)
Where deps come from	npm registry	Maven repositories (Google's, Maven Central)
Build/run command	npm run build	Run ▶ in the IDE, or ./gradlew build

The dependency-declaration idea from Node.js Fundamentals (Course 1, Chapter 2) transfers directly — `implementation(...)` lines are Gradle's equivalent of entries in `package.json`'s `dependencies`, just referencing Maven coordinates (`group:artifact:version`) instead of npm package names.

📲 The Emulator: Running Your First App

The **Device Manager** creates and manages Android Virtual Devices (AVDs) — emulated phones/tablets that run on the development machine:

1. Open **Device Manager** (usually a phone-shaped icon in the toolbar or right-hand panel).
2. Click **Create Device**, pick a phone definition (e.g. Pixel 7), and select a system image (an Android version) — Android Studio may need to download the image first.
3. Once created, click the green **Run** ▶ button with the new AVD selected as the target — Android Studio builds the app, boots the emulator (first boot takes a minute or two), installs the app, and launches it automatically.

The default **Empty Views Activity** template shows a single screen with "Hello World!" text — that text lives in `activity_main.xml`, not hardcoded in `MainActivity.kt`, following the resource-separation pattern from the warning box above.

> 💡 A Physical Device Works Too, and Is Often Faster
>
> Enabling Developer Options and USB debugging on an Android phone, then connecting it by USB, lets it appear as a run target alongside emulators — often noticeably faster than emulation, and useful for testing on real hardware. Not required for this course, but worth knowing exists.

💻 Coding Challenges

Challenge 1: Create and Run a Project

Create a new project using the **Empty Views Activity** template, name it anything you like, set the minimum SDK to API 24, create an AVD, and run the app successfully — confirming "Hello World!" appears on the emulator screen.

Goal: Confirm the full toolchain (Android Studio → Gradle → emulator) works end to end.

→ Solution

Challenge 2: Edit a String Resource

Open `res/values/strings.xml`, find the string used by the "Hello World!" text (check `activity_main.xml` for the `@string/...` reference to identify its name), change its value, and re-run the app to confirm the new text appears — without touching `MainActivity.kt` or the layout XML at all.

Goal: Get comfortable with the res/ resource-reference pattern by using it, not just reading about it.

→ Solution

Challenge 3: Add a Dependency

Add the `com.google.android.material:material` dependency to `app/build.gradle.kts` (a specific, recent version), sync Gradle, and confirm the sync completes without errors. Add a short comment noting what "Sync Now" actually does, based on the Gradle vs npm comparison in this chapter.

Goal: Practice the dependency-declaration + sync workflow that every later chapter's new library will use.

→ Solution

💡 Slow Builds the First Time Are Normal

The very first Gradle sync and build on a new machine can take several minutes as it downloads dependencies and build tools — subsequent builds are dramatically faster thanks to caching. If a first sync seems stuck, it's very likely still downloading, not actually frozen.

🎯 What's Next

Next chapter: **Activities & Lifecycle** — the Activity class, lifecycle callbacks, savedInstanceState, and reading Logcat output.

🔄 Activities & Lifecycle

An Activity is a single screen in an Android app — and unlike a browser tab, which the OS mostly leaves alone, an Activity can be paused, backgrounded, and even destroyed and recreated by the system at almost any time, for reasons entirely outside your app's control. This chapter covers the Activity class, its lifecycle callbacks, why the lifecycle matters in practice, and Logcat — the tool used to actually watch it happen.

🖼️ What an Activity Is

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

An `Activity` represents one screen with a UI — `MainActivity` is the one launched when the app icon is tapped, declared as such in `AndroidManifest.xml` (Chapter 1) via an intent filter. `setContentView(R.layout.activity_main)` is the line that actually connects this Kotlin class to the XML layout file it displays — without it, the Activity exists but shows nothing.

⚠ super.onCreate() Must Come First

`super.onCreate(savedInstanceState)` has to be the first line of an overridden `onCreate` — it runs the base Android framework setup this Activity depends on. Skipping it, or calling it late, causes a crash. This applies to every lifecycle callback covered in this chapter, not just `onCreate`.

🔁 The Lifecycle Callbacks

An Activity moves through a defined sequence of states as the user (or the system) interacts with it, and Android calls a specific method at each transition:

```
class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    Log.d("MainActivity", "onCreate called")
  }

  override fun onStart() {
    super.onStart()
    Log.d("MainActivity", "onStart called")
  }

  override fun onResume() {
    super.onResume()
    Log.d("MainActivity", "onResume called")
  }

  override fun onPause() {
    super.onPause()
    Log.d("MainActivity", "onPause called")
  }

  override fun onStop() {
    super.onStop()
    Log.d("MainActivity", "onStop called")
  }

  override fun onDestroy() {
    super.onDestroy()
    Log.d("MainActivity", "onDestroy called")
  }
}
```

Each Callback, Plainly

Callback	Fires when…
onCreate	The Activity is first created — one-time setup (setContentView, initializing views)
onStart	The Activity becomes visible (but not yet interactive)
onResume	The Activity is now in the foreground and interactive — the user can tap things
onPause	Losing focus, but still (partially) visible — e.g. a dialog appears over it
onStop	No longer visible at all — e.g. user pressed Home, or navigated to another Activity
onDestroy	Being permanently removed — user pressed Back, or the system reclaimed memory

Rotating the device, pressing Home, receiving a phone call mid-app, or the system simply needing memory back can all trigger these transitions — often `onPause`/`onStop` followed later by `onDestroy` with no user action that looks like "closing the app" at all from their perspective.

vs the Web: No "Onload/Onunload" Equivalent Guarantee

	Browser Page	Android Activity
Typical lifetime	Loads once, stays alive until tab closes	Created/destroyed repeatedly during normal use
Rotation/resize	CSS reflow only — JS state untouched	Can fully recreate the Activity from scratch by default
Backgrounding	Tab keeps running (usually)	onStop then likely onDestroy if memory is needed

This is the single biggest mental shift coming from web development: a browser tab's JavaScript state is a comparatively stable, long-lived thing. An Activity's Kotlin state is **not** — by default, rotating the screen destroys and recreates the entire Activity, which is exactly why the next section matters.

💾 savedInstanceState — Surviving Recreation

Before destroying an Activity for a recoverable reason (like rotation, not like the user pressing Back), Android calls `onSaveInstanceState`, giving a chance to stash small bits of UI state into a `Bundle` — which then comes back as the `savedInstanceState` parameter in the next `onCreate`:

```
class MainActivity : AppCompatActivity() {
   private var counter = 0
override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_main)

      // Restore, if a previous instance saved something
      counter = savedInstanceState?.getInt("counter") ?: 0
   }

   override fun onSaveInstanceState(outState: Bundle) {
      super.onSaveInstanceState(outState)
      outState.putInt("counter", counter)
   }
}
```

This is deliberately for small, transient UI state (a scroll position, a form field's current text, a counter like above) — not a substitute for real persistence. Anything that needs to survive the app being fully closed and relaunched later belongs in a database or file (covered in a later course), not a `Bundle`.

⚠ savedInstanceState Is Null on a Genuinely Fresh Start

The `savedInstanceState?.getInt(...)` null-safe call above matters: `savedInstanceState` is `null` when the Activity is starting completely fresh (first launch, or after the user explicitly closed it) — the safe-call plus Elvis operator pattern from Kotlin Fundamentals Chapter 3 is exactly what handles both cases correctly in one line.

📋 Logcat — Watching It All Happen

`Log.d(...)` (and its siblings `Log.i`, `Log.w`, `Log.e`) writes to Logcat — Android's equivalent of `console.log`, viewable in Android Studio's Logcat panel (Chapter 1) while the app runs:

```
Log.d("MainActivity", "onCreate called")   // Debug — general development info
Log.i("MainActivity", "User logged in")    // Info — notable events
Log.w("MainActivity", "Cache miss")        // Warn — recoverable problems
Log.e("MainActivity", "Failed to load data") // Error — something went wrong
```

The first argument is a **tag** — conventionally the class name — which Logcat's search bar can filter by, letting a specific Activity's output be isolated out of the constant stream of system-level log noise every Android app produces. Running the lifecycle example above and rotating the emulator is genuinely the best way to internalize the callback order — watching `onPause` → `onStop` → `onDestroy` → `onCreate` → `onStart` → `onResume` scroll past in Logcat makes the abstract sequence concrete.

💻 Coding Challenges

Challenge 1: Logging Every Lifecycle Callback

Add the six lifecycle overrides from this chapter (onCreate through onDestroy) to MainActivity, each with a `Log.d` call using a consistent tag. Run the app, watch Logcat, then rotate the emulator (Ctrl+F11/F12, or the rotate button in the emulator toolbar) and observe which callbacks fire in what order.

Goal: See the lifecycle sequence happen for real, not just read about it.

→ Solution

Challenge 2: Surviving Rotation with savedInstanceState

Add a private `var clickCount: Int = 0` to MainActivity, a Button in the layout that increments and logs it on click, and the onSaveInstanceState/onCreate pair needed to restore `clickCount` after rotation. Click the button several times, rotate the device, and confirm (via Logcat) the count survived.

Goal: Practice the actual save/restore round trip, not just writing the two methods in isolation.

→ Solution

Challenge 3: Reading a Real Logcat Filter

With the app from Challenge 1 running, use Logcat's search/filter bar to show only logs with your chosen tag, then separately filter to show only "Warn" level and above. Write a short comment describing what each filter changed about the visible output.

Goal: Get comfortable filtering Logcat's noisy default output down to what's actually relevant.

→ Solution

💡 Getting Comfortable Here Prevents Real Bugs

"Why did my data disappear when I rotated the phone?" is one of the most common early Android bugs, and it's a direct consequence of not internalizing this chapter. The lifecycle isn't an edge case to occasionally worry about — it's the normal, constant background rhythm every Activity lives inside.

🎯 What's Next

Next chapter: **XML Layouts & Views** — LinearLayout/ConstraintLayout, common Views, and view binding.

📐 XML Layouts & Views

Every screen's visual structure is described in XML, not code — conceptually close to HTML describing a page's structure while CSS handles positioning. This chapter covers the two layout containers used constantly (LinearLayout and ConstraintLayout), the common building-block Views, and view binding — the type-safe way to reach a View from Kotlin, replacing the raw findViewById calls used last chapter.

XML Layout Basics

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">

  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/hello_world" />

</LinearLayout>
```

Every View needs `layout_width` and `layout_height` — usually `match_parent` (fill the available space, like CSS `width: 100%`) or `wrap_content` (only as big as the content needs, like CSS's default inline sizing). There's no XML equivalent of a raw pixel value being the norm the way it might be in a quick HTML mockup — Android strongly prefers these two content-relative modes, or explicit `dp` (density-independent pixel) values from a dimens resource for anything else.

📏 LinearLayout — Stack in One Direction

`LinearLayout` arranges children in a single row or column, controlled by `orientation` — directly analogous to a CSS flexbox container locked to one direction:

```
<LinearLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="horizontal">

  <Button
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="Cancel" />

  <Button
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="Confirm" />

</LinearLayout>
```

`layout_weight` distributes remaining space proportionally among children — the pattern `layout_width="0dp"` + a weight is the standard idiom for "split the available width," directly equivalent to giving several CSS flex items `flex: 1` and `width: 0`.

🔗 ConstraintLayout — Relative Positioning at Scale

`ConstraintLayout` positions each child relative to the parent or to *other children*, using explicit constraints on each edge — the default root layout for new activities, because it avoids deeply nested `LinearLayout`s inside `LinearLayout`s for anything beyond a simple stack:

```
<androidx.constraintlayout.widget.ConstraintLayout
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Welcome"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent" />

  <Button
    android:id="@+id/continueButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Continue"
    app:layout_constraintTop_toBottomOf="@id/title"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

`app:layout_constraintTop_toBottomOf="@id/title"` reads naturally: this View's top edge attaches below `title`'s bottom edge. Every child needs both a horizontal constraint (start/end) and a vertical constraint (top/bottom) or its position is undefined — the layout editor's visual designer (Chapter 1's Editor panel) makes dragging these connections considerably faster than hand-writing them.

Android Layouts vs CSS

CSS Concept	Android Equivalent
display: flex; flex-direction: column;	LinearLayout, orientation="vertical"
flex: 1	layout_weight="1" (with layout_width="0dp")
width: 100%	layout_width="match_parent"
width: auto (content-sized)	layout_width="wrap_content"
CSS Grid / relative anchoring	ConstraintLayout constraints

🧱 Common Views — A Quick Reference

TextView

Displays text — Android's `<p>`/`<span>`. Set via `android:text`, styled via `textSize`, `textColor`.

Button

A tappable button. Click handling is covered fully in Chapter 4, but the View itself is declared the same way as any other.

EditText

A text input field — Android's `<input type="text">`. `inputType` controls the keyboard shown (e.g. `"number"`, `"textEmailAddress"`).

ImageView

Displays an image, from a drawable resource (`android:src="@drawable/logo"`) or set at runtime from code.

🔒 View Binding — Type-Safe View References

Chapter 2's `findViewById<Button>(R.id.clickButton)` works, but has two real problems: it's unchecked at compile time (a typo'd ID or wrong type only fails at runtime), and it's verbose to repeat for every View. **View binding** fixes both by generating a typed binding class from each layout file automatically.

First, enable it in `app/build.gradle.kts`:

```
android {
    // ...existing config...
    buildFeatures {
        viewBinding = true
    }
}
```

For a layout file `activity_main.xml`, Android generates a class called `ActivityMainBinding` with a property for every View that has an `android:id`:

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)  // binding.root replaces the old R.layout.activity_main reference

        binding.clickButton.setOnClickListener {  // typed, autocompleted, and compile-checked
            binding.titleText.text = "Clicked!"
        }
    }
}
```

findViewById vs View Binding

	findViewById	View Binding
Type safety	Manual cast, wrong-type crashes at runtime	Compiler-checked, correct type generated
Typo'd ID	Compiles, crashes at runtime (returns null / ClassCastException)	Won't compile — the property simply doesn't exist
Repetition	One call per View, every time it's needed	One binding.inflate() call, then binding.viewName everywhere

⚠ lateinit var — A Kotlin Reminder

`lateinit var binding: ActivityMainBinding` is necessary because `binding` genuinely can't be initialized where it's declared (it needs `layoutInflater`, only available once the Activity exists) — but it's guaranteed to be set before any code that uses it actually runs, in `onCreate`. This is exactly the kind of case Kotlin's `lateinit` exists for: a non-nullable `var` that's set once, shortly after construction, not immediately at declaration.

💻 Coding Challenges

Challenge 1: A LinearLayout Form

Build a vertical LinearLayout containing a TextView (label), an EditText (input), and a Button, each with sensible layout_width/layout_height. Give the EditText an appropriate inputType for a name field.

Goal: Practice basic LinearLayout structure and common View attributes.

→ Solution

Challenge 2: A ConstraintLayout Screen

Rebuild the same three Views (label, input, button) from Challenge 1 using ConstraintLayout instead — the label at the top, the input below it, and the button below that, all with correct top/start/end constraints so the layout works regardless of screen width.

Goal: Practice writing explicit ConstraintLayout constraints by hand.

→ Solution

Challenge 3: Migrate to View Binding

Enable viewBinding in build.gradle.kts, then rewrite MainActivity from Challenge 2 (or Chapter 2) to use view binding instead of any findViewById calls — set the button's click listener to update the TextView's text using binding.viewName syntax throughout.

Goal: Practice the full view binding setup and usage, end to end.

→ Solution

💡 ConstraintLayout Feels Unfamiliar Before It Feels Natural

Coming from CSS, writing explicit constraints for every single edge feels tedious at first compared to flexbox's implicit flow. The payoff shows up once layouts get more complex — a ConstraintLayout with 15 Views stays a single flat hierarchy, where the equivalent nested-LinearLayout version would be five or six layers deep and considerably harder to reason about or restyle.

🎯 What's Next

Next chapter: **User Interaction** — onClick, event listeners, Intents (explicit/implicit), and passing data between screens.

👆 User Interaction

Chapter 3 built static screens; this chapter makes them respond to taps and navigate between each other. It covers click and text-change listeners, then Intents — the object that both launches a new screen and carries data along with it, in one mechanism.

🖱️ Click Listeners

Chapter 3 already used `setOnClickListener` briefly — here's the fuller picture, including the pattern for reading input at click time:

```
binding.submitButton.setOnClickListener {
    val name = binding.nameInput.text.toString()
    binding.nameLabel.text = "Hello, $name!"
}
```

`setOnClickListener` takes a lambda — this is exactly the same lambda-as-callback pattern from Kotlin Fundamentals Chapter 2, just receiving a View click event instead of, say, a collection element. There's also an older XML-attribute style (`android:onClick="onSubmitClicked"`, referencing a method on the Activity) still seen in legacy code, but the Kotlin lambda form is the modern default and what this course uses throughout.

setOnLongClickListener

Fires on a press-and-hold, not a tap — the lambda must return a `Boolean` (whether the long click was "consumed," suppressing any regular click that might otherwise also fire).

addTextChangedListener

Fires on every keystroke in an EditText — the live-validation equivalent of a JavaScript `input` event listener, useful for things like a real-time character counter or inline form validation.

🚀 Explicit Intents — Launching Your Own Screens

An `Intent` is an object describing "an operation to perform" — most commonly, "start this specific Activity." An **explicit** Intent names the target Activity directly:

```
val intent = Intent(this, DetailActivity::class.java)
startActivity(intent)
```

`this` is the current Activity (acting as the required `Context` parameter), and `DetailActivity::class.java` identifies the destination — both required arguments. `startActivity(intent)` then hands the Intent to the Android system, which creates and launches the new Activity, pushing the current one to `onPause`/`onStop` (Chapter 2) in the process.

📦 Passing Data Along With an Intent

An Intent doubles as a small data carrier — `putExtra` attaches key-value pairs, retrieved on the receiving end with the matching typed getter:

```
// Sending Activity
val intent = Intent(this, DetailActivity::class.java)
intent.putExtra("USER_NAME", "Philip")
intent.putExtra("USER_AGE", 33)
startActivity(intent)


// Receiving Activity — DetailActivity
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  setContentView(R.layout.activity_detail)

  val name = intent.getStringExtra("USER_NAME") ?: "Unknown"
val age = intent.getIntExtra("USER_AGE", 0)
  // getStringExtra returns String? — Elvis fallback handles the missing-key case
// getIntExtra takes a default value directly as its second parameter instead
}
```

⚠ Extras Are Untyped Until You Retrieve Them

An Intent's extras are stored in a `Bundle` (the same type from Chapter 2's savedInstanceState) — retrieving the wrong type for a given key, or a mistyped key entirely, doesn't fail at compile time. Keeping extra keys as `const val` constants (`const val EXTRA_USER_NAME = "USER_NAME"`), shared between sender and receiver, avoids typo-based bugs — a small habit worth adopting early.

🌐 Implicit Intents — Asking the System for Help

An **implicit** Intent doesn't name a specific Activity — it describes an *action*, and lets Android find whatever app on the device can handle it:

```
// Open a URL in whatever browser is installed
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("https://osztromok.com"))
startActivity(intent)

// Share text via whatever apps offer sharing (Messages, Email, etc.)
val shareIntent = Intent(Intent.ACTION_SEND).apply {
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, "Check out this app!")
}
startActivity(Intent.createChooser(shareIntent, "Share via"))
```

`ACTION_VIEW` with a URI, `ACTION_SEND` for sharing, `ACTION_DIAL` for a phone number — the action constant plus relevant data tells the system what's needed, and any installed app that's registered to handle that action becomes a candidate. `Intent.createChooser(...)` explicitly shows the picker UI even if only one app could handle it, which is usually the friendlier choice for a "Share" action specifically.

Intents vs Web Navigation

	Web (window.location / React Router)	Android
Navigate to a new "page"	window.location.href = "/detail" or <Link to="/detail">	startActivity(Intent(this, DetailActivity::class.java))
Pass data along	URL params, query strings, router state	Intent.putExtra(key, value)
Delegate to another app entirely	Not really applicable — the browser IS the app	An implicit Intent (ACTION_VIEW, ACTION_SEND, ...)

The explicit/implicit split has no real web equivalent — a web app effectively can't "hand off" to another completely separate application the way `ACTION_SEND` can open the user's choice of messaging app. It's a genuinely different, OS-level idea: apps on Android cooperate through Intents rather than staying fully siloed.

💻 Coding Challenges

Challenge 1: Navigate Between Two Activities

Create a second Activity called `DetailActivity` with a simple layout (one TextView). In MainActivity, add a button that launches `DetailActivity` using an explicit Intent when clicked.

Goal: Practice the basic explicit-Intent navigation pattern.

→ Solution

Challenge 2: Pass Data to the Second Activity

Extend Challenge 1: add an EditText to MainActivity, and when the button is clicked, pass its current text to DetailActivity via putExtra. In DetailActivity's onCreate, retrieve the extra (with an appropriate fallback if missing) and display it in the TextView.

Goal: Practice the full putExtra/getXExtra round trip.

→ Solution

Challenge 3: An Implicit Share Intent

Add a "Share" button to DetailActivity that uses an implicit ACTION_SEND Intent to share the text it's displaying (from Challenge 2), wrapped in Intent.createChooser so a picker always appears.

Goal: Practice building and launching an implicit Intent.

→ Solution

💡 A New Activity Needs Registering — Don't Forget the Manifest

Any new Activity created (like DetailActivity in the challenges) must be declared with an `<activity>` entry in AndroidManifest.xml (Chapter 1) before it can be launched — Android Studio adds this automatically when using its "New → Activity" wizard, but a manually-created Activity class needs it added by hand, or startActivity() crashes with an ActivityNotFoundException.

🎯 What's Next

Next chapter: **RecyclerView** — the adapter pattern, ViewHolder, DiffUtil, and click handling in scrollable lists.

📜 RecyclerView

RecyclerView is Android's tool for scrollable lists — a contact list, a feed, search results. Its defining trait is right in the name: instead of creating a View for every single item up front, it recycles a small, fixed number of item Views as the user scrolls, reusing off-screen ones instead of endlessly allocating new ones. This chapter covers the adapter pattern that makes that possible, ViewHolder, efficient updates with DiffUtil, and handling clicks on individual items.

🧩 The Adapter Pattern

RecyclerView itself knows nothing about your data — an `Adapter` is the bridge between a data source (a `List<T>`, most often) and the actual item Views on screen, implemented with three required overrides:

```
data class Contact(val name: String, val phone: String)

class ContactAdapter(private val contacts: List<Contact>) :
    RecyclerView.Adapter<ContactAdapter.ContactViewHolder>() {

    class ContactViewHolder(val binding: ItemContactBinding) :
        RecyclerView.ViewHolder(binding.root)

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ContactViewHolder {
        val binding = ItemContactBinding.inflate(
            LayoutInflater.from(parent.context), parent, false
        )
        return ContactViewHolder(binding)
    }

    override fun onBindViewHolder(holder: ContactViewHolder, position: Int) {
        val contact = contacts[position]
        holder.binding.nameText.text = contact.name
        holder.binding.phoneText.text = contact.phone
    }

    override fun getItemCount() = contacts.size
}
```

onCreateViewHolder — Called Rarely

Inflates a brand-new item layout and wraps it in a ViewHolder. Only called enough times to fill the visible screen plus a small buffer — **not** once per data item, which is the entire performance idea behind RecyclerView.

onBindViewHolder — Called Constantly

Takes an already-created ViewHolder and fills it with data for a specific `position`. This runs every time a recycled View scrolls back on screen with new data — it should stay cheap, since it fires very frequently during scrolling.

🗂️ ViewHolder — Caching View Lookups

`ViewHolder` exists purely to hold references to one item's Views (via view binding, per Chapter 3) so they're looked up **once**, at creation, rather than re-searched on every bind. Without it, every scroll frame would repeat the equivalent of a fresh `findViewById` for every visible item, which used to be a genuine performance problem in Android's early list-view APIs.

```
// item_contact.xml — one row's layout, referenced by ItemContactBinding above
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical"
  android:padding="12dp">

  <TextView android:id="@+id/nameText" android:layout_width="match_parent" android:layout_height="wrap_content" />
  <TextView android:id="@+id/phoneText" android:layout_width="match_parent" android:layout_height="wrap_content" />

</LinearLayout>
```

🔌 Wiring It Up in an Activity

```
val contacts = listOf(
    Contact("Alice", "555-0101"),
    Contact("Bob", "555-0102")
)

binding.recyclerView.layoutManager = LinearLayoutManager(this)
binding.recyclerView.adapter = ContactAdapter(contacts)
```

A `layoutManager` is a separate, required piece — it decides *how* items are arranged (`LinearLayoutManager` for a simple vertical or horizontal list, `GridLayoutManager` for a grid). RecyclerView deliberately splits "how data becomes Views" (the Adapter) from "how those Views are positioned" (the LayoutManager) — two independent, swappable concerns.

⚡ DiffUtil — Efficient List Updates

Calling `notifyDataSetChanged()` after a data change tells the Adapter "something changed, redraw everything" — correct, but wasteful, and it kills any scroll-position-preserving item animations. `DiffUtil` instead computes the minimal set of actual changes between an old list and a new one:

```
class ContactDiffCallback(
    private val oldList: List<Contact>,
    private val newList: List<Contact>
) : DiffUtil.Callback() {
    override fun getOldListSize() = oldList.size
    override fun getNewListSize() = newList.size

    override fun areItemsTheSame(oldPos: Int, newPos: Int) =
        oldList[oldPos].phone == newList[newPos].phone   // same underlying entity? (an ID, typically)
override fun areContentsTheSame(oldPos: Int, newPos: Int) =
        oldList[oldPos] == newList[newPos]   // same VALUES? (data class equals(), from Kotlin Fundamentals Ch 4)
}
```

`areItemsTheSame` answers "is this the same real-world contact?" (usually by a stable ID) — `areContentsTheSame` answers "has anything about it actually changed?" DiffUtil then tells the Adapter to animate exactly the items that were added, removed, moved, or changed, leaving everything else alone.

DiffUtil vs React's Key-Based Reconciliation

	React	RecyclerView + DiffUtil
Identity check	key prop on each list item	areItemsTheSame()
Change check	Shallow prop comparison (or memo)	areContentsTheSame()
Result	Minimal virtual DOM diff applied	Minimal RecyclerView item animations applied

⚠ ListAdapter — DiffUtil Without the Boilerplate

In practice, most real code extends `ListAdapter<Contact, ContactViewHolder>` instead of plain `RecyclerView.Adapter` — it wraps DiffUtil internally (via a smaller `DiffUtil.ItemCallback`) and exposes a single `submitList(newList)` call that handles the diffing and updating automatically. The manual `DiffUtil.Callback` above is worth understanding once, precisely so `ListAdapter`'s automatic version doesn't feel like unexplained magic.

👆 Handling Clicks on List Items

Click handling is set up per-item inside `onBindViewHolder`, most cleanly by passing a lambda into the Adapter's constructor — a direct callback-parameter pattern, same shape as passing a lambda to `map`/`filter` back in Kotlin Fundamentals Chapter 6:

```
class ContactAdapter(
    private val contacts: List<Contact>,
    private val onContactClick: (Contact) -> Unit
) : RecyclerView.Adapter<ContactAdapter.ContactViewHolder>() {

    override fun onBindViewHolder(holder: ContactViewHolder, position: Int) {
        val contact = contacts[position]
        holder.binding.nameText.text = contact.name
        holder.binding.root.setOnClickListener { onContactClick(contact) }
    }
    // ...onCreateViewHolder, getItemCount unchanged...
}

// In the Activity:
binding.recyclerView.adapter = ContactAdapter(contacts) { clickedContact ->
    Log.d("MainActivity", "Clicked: ${clickedContact.name}")
}
```

💻 Coding Challenges

Challenge 1: A Basic RecyclerView List

Build a RecyclerView showing a list of at least 5 `data class Product(val name: String, val price: Double)` items, with an item layout showing both fields. Write the Adapter and ViewHolder, and wire it up in the Activity with a LinearLayoutManager.

Goal: Practice the full Adapter/ViewHolder/getItemCount setup from scratch.

→ Solution

Challenge 2: Click Handling

Extend Challenge 1's Adapter to accept an `onProductClick: (Product) -> Unit` lambda in its constructor, set on each item's root view in onBindViewHolder. In the Activity, log the clicked product's name via the passed-in lambda.

Goal: Practice the constructor-lambda click-handling pattern.

→ Solution

Challenge 3: A DiffUtil.Callback

Write a `ProductDiffCallback` implementing DiffUtil.Callback for two `List<Product>`, using the product name as the stable identity for areItemsTheSame, and full data class equality for areContentsTheSame. In a comment, describe what would happen (in terms of item animations) if two lists differed by one product's price changing.

Goal: Practice writing DiffUtil.Callback correctly, understanding the distinct roles of its two comparison methods.

→ Solution

💡 The Same Recycling Idea Applies to Every Scrollable List

Every long list in a real Android app — a chat history, an e-commerce catalog, a social feed — uses this exact same adapter/ViewHolder shape underneath, no matter how visually different the items look. Once this pattern is genuinely comfortable, most list-related work becomes "design this one item's layout," not "figure out list infrastructure again."

🎯 What's Next

Next chapter: **Fragments** — fragment lifecycle, the fragment manager, and communicating with the hosting Activity.

🧩 Fragments

A Fragment is a reusable, self-contained piece of UI and behavior that lives inside an Activity — closer to a component in web frameworks than anything covered so far in this course. This chapter covers the Fragment lifecycle (which nests inside the Activity lifecycle from Chapter 2), the FragmentManager that adds and swaps Fragments, and the two standard ways a Fragment talks back to its hosting Activity.

🖼️ What a Fragment Is, and Why It Exists

An Activity represents a full screen; a Fragment represents a **portion** of one, with its own layout and its own Kotlin class — designed to be combined, swapped, and reused across different Activities or screen configurations (a tablet showing two Fragments side by side where a phone shows them one at a time, for instance).

Fragments vs Web Components

	React Component	Android Fragment
Represents	A reusable piece of UI	A reusable piece of UI
Has its own lifecycle?	Yes (mount/update/unmount)	Yes, nested inside the host Activity's
Hosted by	A parent component / the DOM tree	An Activity (or another Fragment)
Swapped at runtime?	Via conditional rendering / router	Via FragmentManager transactions

🔁 The Fragment Lifecycle

A Fragment has its own set of lifecycle callbacks, layered on top of (and driven by) its host Activity's lifecycle from Chapter 2:

```
class ProfileFragment : Fragment() {

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?
    ): View {
        return inflater.inflate(R.layout.fragment_profile, container, false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        // Safe to touch the Fragment's own Views from here onward
        Log.d("ProfileFragment", "View is ready")
    }

    override fun onDestroyView() {
        super.onDestroyView()
        // The Fragment's View is being torn down — clear any View references here
    }
}
```

onCreateView — Inflate, Don't Configure

Inflates and returns the Fragment's layout — the Fragment equivalent of an Activity's `setContentView`, except the View is *returned* rather than passed to a framework method.

onViewCreated — Configure Here Instead

Called right after `onCreateView`, with the inflated View available as a parameter — the right place to set up click listeners, bind data, and generally do what Chapter 2's Activity `onCreate` did, but for a Fragment's Views specifically.

⚠ A Fragment's View Can Be Destroyed Without the Fragment Itself Being Destroyed

This is the trap that catches most beginners: a Fragment instance can survive (e.g. kept on a back stack) while its View is torn down and later recreated — `onDestroyView` fires without `onDestroy`. Holding a View reference in a regular property beyond `onDestroyView` is a classic memory-leak/crash source; view binding references (Chapter 3) should be nulled out in `onDestroyView` in real production code — simplified away here to keep the examples focused, but worth knowing exists.

🗂️ FragmentManager — Adding & Swapping Fragments

An Activity's layout reserves a container (a `FrameLayout`, typically), and the `FragmentManager` adds, replaces, or removes Fragments inside it at runtime:

```
// activity_main.xml — a container for Fragments to live in
<FrameLayout
    android:id="@+id/fragmentContainer"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />


// MainActivity.kt — inside onCreate, showing a Fragment for the first time
if (savedInstanceState == null) {
    supportFragmentManager.beginTransaction()
        .replace(R.id.fragmentContainer, ProfileFragment())
        .commit()
}
```

The `if (savedInstanceState == null)` check matters: without it, rotating the device (Chapter 2) would re-add a duplicate Fragment on top of the one the FragmentManager already automatically restored. `replace(...)` swaps out whatever Fragment currently occupies that container for a new one — `add(...)` stacks a new one on top instead, without removing what's there.

> ⚠ This Chapter Shows FragmentManager Directly — Real Apps Usually Don't
>
> Manually calling `supportFragmentManager.beginTransaction()...` is genuinely how Fragment swapping works underneath, but next chapter's Navigation Component provides a higher-level, declarative way to manage Fragment navigation (a nav graph, back stack handling, and passing arguments all handled for you) that most real apps use instead. Understanding the manual mechanism first is what makes the Navigation Component feel like a natural upgrade rather than unexplained magic.

📢 Talking Back to the Host Activity

A Fragment shouldn't hold a hard reference to a specific Activity class (that would defeat its whole reusability point) — the standard pattern is an interface the Fragment defines and the hosting Activity implements:

```
class ProfileFragment : Fragment() {

    interface ProfileListener {
        fun onProfileSaved(name: String)
    }

    private var listener: ProfileListener? = null
override fun onAttach(context: Context) {
        super.onAttach(context)
        listener = context as? ProfileListener   // safe cast — null if the Activity doesn't implement it
    }

    fun saveProfile(name: String) {
        listener?.onProfileSaved(name)   // safe call — no-op if listener is null
    }
}

// MainActivity.kt
class MainActivity : AppCompatActivity(), ProfileFragment.ProfileListener {
    override fun onProfileSaved(name: String) {
        Log.d("MainActivity", "Fragment reported: $name")
    }
}
```

`context as? ProfileListener` is a safe cast (Kotlin Fundamentals Chapter 3's `?.`/`?:` family, applied to a cast rather than a call) — it yields `null` instead of crashing if the hosting Activity doesn't implement the interface, which the following `listener?.onProfileSaved(...)` then handles gracefully by simply doing nothing.

> 💡 Shared ViewModel Is the Modern Alternative
>
> The interface-callback pattern above is the traditional approach and worth knowing, but modern Android code increasingly favors a **shared ViewModel** (scoped to the Activity, accessible from every hosted Fragment) for cross-Fragment communication instead — covered fully once ViewModels are introduced in Course 2. Both solve the same underlying problem; the interface pattern is the more foundational one to understand first.

💻 Coding Challenges

Challenge 1: A Basic Fragment

Create a `WelcomeFragment` with a simple layout (one TextView), implementing onCreateView. Add a FrameLayout container to MainActivity's layout, and load WelcomeFragment into it via FragmentManager in onCreate, guarded by the savedInstanceState null check.

Goal: Practice the basic Fragment creation and FragmentManager loading pattern.

→ Solution

Challenge 2: Swapping Fragments

Create a second Fragment, `DetailsFragment`, and add a Button to WelcomeFragment that, when clicked, uses FragmentManager to replace WelcomeFragment with DetailsFragment in the same container.

Goal: Practice replace() for swapping Fragments at runtime, not just loading one initially.

→ Solution

Challenge 3: Fragment-to-Activity Communication

Add an interface `OnDetailsClosedListener` with a function `onDetailsClosed()` to DetailsFragment, implement it via onAttach with a safe cast, and add a "Close" button that calls it. Have MainActivity implement the interface and log a message when it's called.

Goal: Practice the full interface-callback communication pattern end to end.

→ Solution

💡 Fragments Feel Heavier at First — That's Normal

Compared to an Activity, a Fragment involves more moving pieces (its own lifecycle, a hosting container, a manager to add/remove it) for what's conceptually "just a reusable piece of UI." That overhead pays for itself once an app has several screens sharing common pieces, or needs different screen arrangements on different device sizes — exactly the kind of reuse a single-Activity design can't offer on its own.

🎯 What's Next

Next chapter: **Navigation Component** — the nav graph, SafeArgs, deep links, and back stack management, building directly on this chapter's Fragments.

🧭 Navigation Component

Last chapter's FragmentManager transactions work, but every "go here," back-stack entry, and passed argument was written by hand. The Navigation Component replaces that with a declarative nav graph: destinations and the paths between them are described in one XML resource, and the library handles transactions, the back stack, and (via SafeArgs) type-safe argument passing automatically.

🗺️ The Nav Graph

A nav graph is an XML resource listing every destination (usually a Fragment) and the actions connecting them:

```
// res/navigation/nav_graph.xml
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  android:id="@+id/nav_graph"
  app:startDestination="@id/welcomeFragment">

  <fragment
    android:id="@+id/welcomeFragment"
    android:name=".WelcomeFragment"
    android:label="Welcome">
    <action
      android:id="@+id/action_welcome_to_details"
      app:destination="@id/detailsFragment" />
  </fragment>

  <fragment
    android:id="@+id/detailsFragment"
    android:name=".DetailsFragment"
    android:label="Details" />

</navigation>
```

Android Studio's **Navigation Editor** shows this graph visually, as boxes connected by arrows — dragging a connection between two Fragments writes the corresponding `<action>` XML automatically, which is normally faster than hand-editing this file directly.

🏠 NavHostFragment — Where the Graph Lives

An Activity hosts the whole graph through a single special Fragment, replacing the plain `FrameLayout` container from Chapter 6:

```
// activity_main.xml
<androidx.fragment.app.FragmentContainerView
  android:id="@+id/navHostFragment"
  android:name="androidx.navigation.fragment.NavHostFragment"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  app:navGraph="@navigation/nav_graph"
  app:defaultNavHost="true" />
```

`app:navGraph` wires this container to the nav graph resource; `app:defaultNavHost="true"` makes the system Back button correctly step backward through the nav graph's own back stack, rather than falling through to the Activity's default Back behavior.

➡️ Navigating Between Destinations

```
// Inside WelcomeFragment, e.g. a button's click listener
binding.viewDetailsButton.setOnClickListener {
    findNavController().navigate(R.id.action_welcome_to_details)
}
```

`findNavController()` locates the `NavController` managing the graph this Fragment lives in, and `navigate(...)` follows the specified action — no manual `FragmentManager.beginTransaction()` call needed anymore. The library also automatically pushes this navigation onto a back stack, so the system Back button (and an app's own explicit "up" button) work correctly without any extra code.

🔒 SafeArgs — Type-Safe Argument Passing

Chapter 4's `Intent.putExtra` / `getStringExtra` pattern is stringly-typed and easy to typo. SafeArgs (a Gradle plugin, added to `build.gradle.kts`) generates typed argument classes directly from arguments declared in the nav graph:

```
// nav_graph.xml — declaring an argument on a destination
<fragment
  android:id="@+id/detailsFragment"
  android:name=".DetailsFragment">
  <argument
    android:name="userName"
    app:argType="string" />
</fragment>


// Sending — a generated Directions class, not a raw Bundle
val action = WelcomeFragmentDirections.actionWelcomeToDetails(userName = "Philip")
findNavController().navigate(action)


// Receiving — a generated Args class, in DetailsFragment
private val args: DetailsFragmentArgs by navArgs()  // property delegation, from Kotlin Intermediate Ch4
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
  super.onViewCreated(view, savedInstanceState)
  binding.detailText.text = args.userName  // typed String, no getStringExtra, no key string, no null check
}
```

Intent Extras vs SafeArgs

	Intent.putExtra (Chapter 4)	SafeArgs
Key	Raw string, must match exactly	Named parameter — checked by the compiler
Type	Chosen getter (getStringExtra, etc.) — can mismatch	Declared once in the nav graph, generated correctly everywhere
Missing value	Runtime null / crash	Compile error if a required argument isn't provided

`args: DetailsFragmentArgs by navArgs()` reuses Kotlin Intermediate Chapter 4's property delegation directly — `navArgs()` is itself a delegate provided by the Navigation library, following the exact same `by` mechanism as `lazy` or a custom delegate.

BACK Back Stack Management

Every `navigate()` call pushes the new destination onto the graph's back stack automatically. An action can also specify `popUpTo` to remove destinations from the stack as part of navigating — useful for flows like "after login, don't let Back return to the login screen":

```
<action
  android:id="@+id/action_login_to_home"
  app:destination="@id/homeFragment"
  app:popUpTo="@id/loginFragment"
  app:popUpToInclusive="true" />
```

`popUpTo="@id/loginFragment"` with `popUpToInclusive="true"` removes `loginFragment` itself (and anything above it) from the back stack while navigating to `homeFragment` — pressing Back from the home screen then skips login entirely, exiting the app instead, exactly the behavior a real login flow needs.

🔗 Deep Links

A deep link lets an external source (a notification, a web link, another app) open the app directly at a specific destination, rather than always starting from the launcher Activity:

```
<fragment
  android:id="@+id/detailsFragment"
  android:name=".DetailsFragment">
  <deepLink app:uri="myapp://details/{userName}" />
</fragment>
```

Tapping a `myapp://details/Philip` link anywhere on the device (a notification, a browser link with the app's scheme registered) launches the app directly into `DetailsFragment`, with `userName` automatically populated as a SafeArgs argument — no manual Intent parsing required.

Navigation Component vs Web Routing

	React Router / Next.js	Navigation Component
Route definitions	<Route path="/details/:id"> or file-based routes	nav_graph.xml destinations + actions
Typed params	useParams() (often untyped without extra work)	SafeArgs-generated Args classes
Programmatic navigation	navigate("/details/5")	findNavController().navigate(action)
Deep linking	Just… a URL, inherently	Explicit <deepLink> declarations needed

💻 Coding Challenges

Challenge 1: Convert Chapter 6's Fragments to a Nav Graph

Take WelcomeFragment and DetailsFragment from Chapter 6, create a nav_graph.xml with both as destinations and an action between them, replace the FrameLayout container with a NavHostFragment, and replace the manual FragmentManager.replace() call with findNavController().navigate().

Goal: Directly experience the FragmentManager-to-NavGraph migration.

→ Solution

Challenge 2: Pass an Argument with SafeArgs

Add the SafeArgs Gradle plugin, declare a String argument named "message" on detailsFragment in the nav graph, send it from WelcomeFragment using the generated Directions class, and display it in DetailsFragment using navArgs().

Goal: Practice the full SafeArgs round trip, replacing the Intent-extras pattern from Chapter 4 with its type-safe nav-graph equivalent.

→ Solution

Challenge 3: popUpTo Behavior

Add a third destination, homeFragment, and an action from detailsFragment to homeFragment that uses popUpTo/popUpToInclusive to clear both welcomeFragment and detailsFragment off the back stack. In a comment, describe what pressing Back from homeFragment would do as a result.

Goal: Practice configuring back stack behavior declaratively via an action's attributes.

→ Solution

💡 Course 1 Nearly Complete — One Chapter Left

Navigation Component is genuinely how most real, multi-screen Android apps handle moving between screens today — the manual FragmentManager approach from Chapter 6 is worth understanding as the foundation, but this declarative graph-based approach is what actual production code reaches for. The final chapter covers resources and styling in more depth, rounding out Course 1's foundation.

🎯 What's Next

Next chapter — the final chapter of Course 1: **Resources & Styling** — string/color/dimen resources in depth, themes, and dark mode basics.

🎨 Resources & Styling

Earlier chapters used string, color, and dimension resources in passing; this final chapter of Course 1 goes deeper — themes, the style/theme distinction, and dark mode, which turns out to be less about writing conditional code and more about how Android's resource system already works.

📋 Resources, Revisited

```
// res/values/strings.xml
<resources>
  <string name="app_name">My First App</string>
  <plurals name="item_count">
    <item quantity="one">%d item</item>
    <item quantity="other">%d items</item>
  </plurals>
</resources>

// res/values/colors.xml
<resources>
  <color name="brand_primary">#3DDC84</color>
  <color name="text_dark">#1A1A1A</color>
</resources>

// res/values/dimens.xml
<resources>
  <dimen name="spacing_medium">16dp</dimen>
  <dimen name="text_size_body">16sp</dimen>
</resources>
```

`<plurals>` is worth calling out specifically — Android handles pluralization rules per-language automatically (some languages have more than two plural forms), which a hand-rolled `if (count == 1) "item" else "items"` in Kotlin can't do correctly across languages. `dp` (density-independent pixels) scales consistently across different screen densities; `sp` (scale-independent pixels) additionally respects the user's font-size accessibility setting — text sizes should use `sp`, everything else (padding, margins, icon sizes) should use `dp`.

🎭 Styles vs Themes

A **style** is a named bundle of attributes applied to one View; a **theme** is a style applied to an entire Activity or app, cascading down to every View inside it:

```
// res/values/styles.xml — a STYLE, for one specific kind of View
<style name="PrimaryButton">
  <item name="android:backgroundTint">@color/brand_primary</item>
  <item name="android:textColor">@color/text_dark</item>
</style>


<Button
  style="@style/PrimaryButton"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Submit" />


// res/values/themes.xml — a THEME, applied app-wide via AndroidManifest.xml
<style name="Theme.MyFirstApp" parent="Theme.MaterialComponents.DayNight">
  <item name="colorPrimary">@color/brand_primary</item>
  <item name="colorOnPrimary">@color/text_dark</item>
</style>
```

The theme is referenced once, in `AndroidManifest.xml`'s `<application android:theme="@style/Theme.MyFirstApp">` — every screen and every View in the app then inherits `colorPrimary` and friends automatically, which is why Material components (buttons, switches, etc.) already look "on brand" without individually styling each one.

Style/Theme vs CSS

CSS Concept	Android Equivalent
A CSS class applied to one element	An Android style applied to one View
CSS custom properties (--primary-color) set on :root	Theme attributes (colorPrimary) inherited app-wide
Cascading inheritance down the DOM tree	Theme attributes cascading down the View tree

🌓 Dark Mode — Automatic, Not Conditional

Dark mode isn't implemented with an `if (isDarkMode)` check scattered through code — it's a **resource qualifier**. A folder named `values-night` holds alternate resource definitions that Android automatically swaps in when the system is in dark mode:

```
res/
├── values/
│   └── colors.xml       # light mode — the default
│       #   #FFFFFF
│       #   #1A1A1A
└── values-night/
    └── colors.xml       # dark mode — same NAMES, different VALUES
#   #121212
#   #E6E6E6
```

Both files define a color named `background` — code and layouts reference `@color/background` exactly once, and Android resolves it to whichever file matches the current system setting, entirely automatically. This requires the app's theme to have a `DayNight` parent (like `Theme.MaterialComponents.DayNight`, used above) — a non-DayNight parent theme won't participate in this automatic switching at all.

⚠ values-night Is One of Many Qualifiers

The same "same resource name, different folder = different value per condition" idea extends to screen orientation (`layout-land/`), language (`values-fr/`, used for the translated-strings example back in Chapter 1), API level (`values-v24/`), and more — `values-night` is just the dark-mode instance of a much more general Android mechanism, conceptually parallel to a CSS media query, but resolved once by the OS rather than continuously re-evaluated by the browser.

Don't Hardcode Colors in Layouts

A hardcoded `android:textColor="#1A1A1A"` directly in a layout XML file bypasses the whole day/night mechanism — it stays that exact color regardless of theme. Always reference `@color/...`, never a raw hex value, to get automatic dark mode support for free.

Testing Dark Mode

The emulator's system settings (or the notification shade's quick-settings toggle) can switch dark mode on the fly — the running app updates its colors immediately without a restart, which is the fastest way to verify a `values-night` setup actually works.

💻 Coding Challenges

Challenge 1: A Reusable Button Style

Define a style named `SecondaryButton` in styles.xml with a distinct background tint and text color (referencing color resources, not hardcoded hex values), and apply it to two different Buttons in a layout to confirm both pick up the shared styling.

Goal: Practice defining and reusing a style across multiple Views.

→ Solution

Challenge 2: A Custom App Theme

Define a theme in themes.xml with a DayNight parent, setting colorPrimary and colorOnPrimary to two color resources of your choice, and apply it via AndroidManifest.xml. Confirm (by running the app) that a default Material Button now picks up colorPrimary without being individually styled.

Goal: Practice the app-wide theme mechanism, distinct from Challenge 1's per-View style.

→ Solution

Challenge 3: Dark Mode Colors

Create a values-night/colors.xml alongside your existing values/colors.xml, redefining background and text_primary (or similar names) with appropriately inverted values for dark mode. Reference both from a layout via @color/, run the app, and toggle system dark mode to confirm the colors switch automatically with no code changes.

Goal: Practice the values-night qualifier folder pattern end to end.

→ Solution

💡 Course 1 Complete — What This Sets Up

Android Development Fundamentals closes here having covered the full traditional-View toolkit: setup and tooling, the Activity lifecycle, XML layouts, user interaction and Intents, RecyclerView, Fragments, Navigation Component, and now resources/theming. Course 2 (Architecture & Data) shifts to Jetpack Compose, ViewModels, Room, and networking — building on this foundation rather than replacing it, since Compose still runs inside Activities with the same lifecycle from Chapter 2.

🎯 What's Next

Android Development Fundamentals (Course 1) is complete. Course 2 (Architecture & Data) begins with **Jetpack Compose Intro** — composables, recomposition, state, and an XML-vs-Compose comparison.