Kotlin Recipes

for Android Developers

Ted Hagos

55 Kotlin Recipes for Android Programming

Practical Solutions to Common Android Development Problems

Ted Hagos

This book is available at
https://leanpub.com/55kotlinrecipesforandroidprogramming

This version was published on 2025-10-01

Contents

Kotlin Basics for Android

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 1: val vs. var keyword. Which one to use?

`val` and `var` are two keywords that let you work with *variables*. They look almost identical, and many people use them interchangeably, but the distinction is important. It may affect the reliability of your code.

Meet `val` – Your "Constant" Buddy.

Think of `val` as a commitment. Once you assign a value, it's locked in—you can't point it to something else later. For example:

```
1  val framework = "Kotlin"
2  framework = "Java" // Not Allowed
```

If you're coming from Java, `val` is equivalent to the `final` keyword.

But here's a subtle detail. While the reference ("Kotlin") can't change, remember that the object it points to might still be mutable. For example.

```
1  val groceries = mutableListOf("Milk", "Eggs")
2  groceries.add("Bread") // Still fine
```

So val is like saying: this name will always refer to the same box, but what's inside that box might change.

"But String are objects" you might say. Yes, String is an object in Kotlin (actually, a wrapper around Java's String, which is immutable). What's important to remember are;

- `val` controls the **reference** (the variable binding)
- But it does **not** make the object itself immutable.

- Since String is already immutable, you can't change its contents anyway, unlike in our MutableList example above.

Enter `var` – The Flexible One.

`var` is the opposite. It's a variable you can reassign as often as you like:

var mood = "Happy" mood = "Not Happy" // This is fine

This is useful for things that naturally change—like counters, user inputs, or the state of a game.

Which Should You Use?

A good Kotlin habit is to default to val. Why? Because immutability makes your code easier to understand and less prone to bugs. Use var only when you truly need to reassign values.

Final Thoughts

If val is your safe, predictable friend, var is your free-spirited one. Both have their place, but leaning on val as much as possible will make your Kotlin code cleaner and more maintainable.

Recipe 2: Null safety with ?, !!, and safe calls.

For nearly three decades, Java developers wrestled with the very pesky `NullPointerException` (NPE); so much that in the classic book *Effective Java*, Joshua Bloch strongly recommended avoiding *null* whenever possible because of bugs it invites and the subsequent runtime crashes. The designers of Kotlin took that advice seriously and baked null safety right into the language.

In Kotlin, all variables are **non-nullable by default**. This is a deliberate design choice to make your code safer and reduce the chances of NPEs.

```
1  var name: String = "Ted"
2  // name = null // Compilation error
```

But Kotlin offers a way for a variable to hold `null`, if you really must. To do this, you need to explicitly tell Kotlin that a variable is *nullable*. You do this by postfixing the variable with a question mark (`?`), as shown in the snippet below.

```
var thename: String? = null
```

This distinction makes you really think about the possibility of `null`. Kotlin then gives you several ways (operators, actually) to work with nullable types.

Safe Call Operator `?.`

The safe call operator lets you safely access a nullable object. If the object is `null`, the expression simply evaluates to `null` instead of crashing; like this.

```
println(thename?.length) // Prints null safely
```

This pattern is useful when chaining multiple calls, as it avoids repetitive null checks.

Non-Null Assertion Operator `!!`

The `!!` operator is Kotlin's escape hatch. It asserts that the value is not null and throws an NPE if you're wrong.

```
println(thename!!.length) // This may crash
```

Use it sparingly; relying on it defeats the purpose of Kotlin's null safety.

Elvis Operator `?:`

When you want a fallback, the Elvis operator is concise and expressive:

```
val displayName = thename ?: "Anonymous"
println(displayName)
```

If `thename` is `null`, `"Anonymous"` is used instead; if it isn't, then the actual value `thename` will be printed.

Final Thoughts

Kotlin puts into practice the advice from *Effective Java*: minimize the use of `null`, and deal with it explicitly when unavoidable. With the operators `?`, `!!`, and `?:`, you can write code that is both safer and more concise.

Reach for safe calls and Elvis operators first. Only use `!!` only as a last resort.

Recipe 3: Default parameters & named arguments (reduce overloads)

In the early days, before Java popularized method overloading, developers had to give functions **distinct names** just to handle slightly different parameter lists. You might see functions like `printUser()`, `printUserWithAge()`, or `printUserWithDetails()`. It worked, but the naming was clumsy and inconsistent. Java improved this by introducing overloads, letting developers reuse the same method name with different signatures. While it was a step forward, it also led to a proliferation of overloads that bloated APIs and made maintenance harder.

Kotlin provides a cleaner and more expressive alternative: **default parameters**. Instead of writing multiple overloads, you can assign default values directly in the function signature. Callers then provide only the arguments they care about, while Kotlin fills in the rest.

```
1  fun greet(name: String, greeting: String = "Hello") {
2      println("$greeting, $name!")
3  }
4
5  // Usage
6  greet("Lyra")                  // Hello, Lyra!
7  greet("Soren", "Welcome")      // Welcome, Soren!
```

Here, a single function replaces what would have been two or more overloads in Java. Less boilerplate, more clarity.

Another powerful feature is **named arguments**. This lets you specify parameters by name, which makes calls self-documenting, especially when parameters are of the same type or appear in long lists:

```
fun displayProfile(name: String, age: Int, isActive: Boolean = true) {
    println("Name: $name, Age: $age, Active: $isActive")
}

// Clearer with named arguments
displayProfile(name = "Elowen", age = 30)
displayProfile(age = 40, name = "Caius", isActive = false)
```

Named arguments improve readability and reduce reliance on remembering parameter order. Combined with defaults, they eliminate the need for most overloads, making APIs far friendlier.

It's worth noting that Joshua Bloch's *Effective Java* also warned against excessive overloads, recommending alternative design patterns for cleaner APIs. Kotlin achieves this by providing developers with language-level support for defaults and named arguments—features that keep code lean, expressive, and maintainable.

Recipe 4: Using string templates instead of concatenation.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 5: Extension functions for cleaner utilities.

One of the coolest things about **Kotlin** is **extension functions**.

Think of them like little superpowers you can instantly grant to any class, even ones you didn't write—and especially handy when dealing with Java's default behavior, where many classes are **final** (meaning you can't subclass them).

Instead of creating a whole new subclass to add one tiny method, or constantly using messy `Utility.doSomething(object)` calls everywhere, you can write an extension function that makes that new ability look and feel like it was *always* a part of the original class. You can simply call `object.doSomething()`.

Since you're not actually modifying the original source code or relying on inheritance, you can effectively extend **final** classes (closed for inheritance), which is a huge advantage over trying to do something similar in standard Java. It keeps your code super clean and intuitive.

This is especially handy for Android development, where you often find yourself writing repetitive utility functions for `View`, `Context`, `String`, or `Date`. By turning them into extensions, you keep your code readable and discoverable through IntelliJ/Android Studio's autocomplete.

* * *

Example: Converting a Utility Method into an Extension

In Java, we often wrote "utility classes" like this:

```
public class StringUtils {
    public static boolean isValidEmail(String input) {
        return input != null && input.contains("@");
    }
}
```

In Kotlin, we can *extend* `String` directly:

```
fun String.isValidEmail(): Boolean {
    return this.contains("@")
}
```

Now you can call it naturally:

```
val email = "dev@example.com"
if (email.isValidEmail()) {
    println("Looks good!")
}
```

Notice how `isValidEmail()` feels like a *native* method of `String`.

More Android-Flavored Example: Hiding the Keyboard

Instead of writing a global helper function:

```
fun hideKeyboard(activity: Activity) {
    val imm = activity.getSystemService(Context.INPUT_METHOD_SERVICE) as
    ↪  InputMethodManager
    imm.hideSoftInputFromWindow(activity.currentFocus?.windowToken, 0)
}
```

We can turn it into an extension on `Activity`:

```
fun Activity.hideKeyboard() {
    val imm = getSystemService(Context.INPUT_METHOD_SERVICE) as
    ↪  InputMethodManager
    currentFocus?.let {
        imm.hideSoftInputFromWindow(it.windowToken, 0)
    }
}
```

Now inside an `Activity`, calling it looks natural:

```
class MainActivity : AppCompatActivity() {

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

        val button = findViewById<Button>(R.id.myButton)
        button.setOnClickListener {
            // Directly available because this is an Activity
            hideKeyboard()
        }
    }
}
```

Here, `hideKeyboard()` is called just like any other method of `Activity`. IntelliJ/Android Studio will even suggest it in autocomplete, as though it were a built-in API.

Working with Android Views & Lifecycle

Recipe 6: Safe Fragment View Binding (avoid leaks)

ViewBinding is an Android feature that generates a binding class for each XML layout file. Each binding class contains **direct references to all the views** in that layout.

For example, if you have a `fragment_safe.xml` with a `TextView` whose ID is `textView`, the generated `FragmentSafeBinding` class will expose it as `binding.textView`.

This gives you:

- **Type safety** – no need to cast views.
- **Null safety** – missing views in layout variants are caught at compile time.
- **Less boilerplate** – no more `findViewById()` scattered across your code.

This is why most modern Android apps prefer **ViewBinding** over the older `findViewById`.

Why care about "safe binding" in fragments?

Fragments have **two overlapping lifecycles**: the fragment itself and the fragment's view. The fragment object can stay alive (for example, when it's placed on the back stack or attached to an activity that's still running), even though its **view hierarchy** is destroyed and later recreated.

If you keep a strong reference to the `binding` outside the view's lifecycle, you're essentially holding onto a view tree that no longer exists – which leads to wasted memory and potential leaks.

That's why we need a pattern to clear the binding safely.

Don't do this - Incorrect binding usage.

```
class UnsafeFragment : Fragment(R.layout.fragment_unsafe) {

    private lateinit var binding: FragmentUnsafeBinding

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding = FragmentUnsafeBinding.bind(view)
        binding.textView.text = "Hello, World!"
    }

    // ⚠️ Problem: binding is never cleared.
    // This may leak memory when the fragment's view is destroyed.
}
```

Do this instead - Manual safe binding pattern

```
class SafeFragment : Fragment(R.layout.fragment_safe) {

    private var _binding: FragmentSafeBinding? = null
    private val binding get() = _binding!!

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        _binding = FragmentSafeBinding.bind(view)

        binding.textView.text = "Hello, safely bound view!"
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null   // clear reference to avoid leaks
    }
}
```

In this way:

- `_binding` is nullable and private.
- The `binding` getter is non-nullable, but only safe to access between `onViewCreated` and `onDestroyView`.
- The binding is explicitly cleared to prevent leaks.

Cleaner utility: Reusable delegate

Manually repeating `_binding` boilerplate in every fragment can get tedious. A **property delegate** can automate safe lifecycle handling:

```
import kotlin.properties.ReadOnlyProperty
import kotlin.reflect.KProperty
import androidx.fragment.app.Fragment
import androidx.lifecycle.DefaultLifecycleObserver
import androidx.lifecycle.LifecycleOwner
import android.view.View
import androidx.viewbinding.ViewBinding

class FragmentViewBindingDelegate<T : ViewBinding>(
    val fragment: Fragment,
    val bind: (View) -> T
) : ReadOnlyProperty<Fragment, T> {

    private var binding: T? = null

    override fun getValue(thisRef: Fragment, property: KProperty<*>): T {
        val currentBinding = binding
        if (currentBinding != null) return currentBinding

        val view = thisRef.view ?: throw IllegalStateException(
            "Cannot access binding. View is null and might be destroyed."
        )

        return bind(view).also { createdBinding ->
            binding = createdBinding
            thisRef.viewLifecycleOwner.lifecycle.addObserver(object :
            ↪  DefaultLifecycleObserver {
                override fun onDestroy(owner: LifecycleOwner) {
                    binding = null
                }
            })
        }
    }
}

fun <T : ViewBinding> Fragment.viewBinding(bind: (View) -> T) =
    FragmentViewBindingDelegate(this, bind)
```

Using the delegate

```
class CleanerFragment : Fragment(R.layout.fragment_cleaner) {

    private val binding by viewBinding(FragmentCleanerBinding::bind)

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.textView.text = "Hello, cleaner world!"
    }
}
```

Just Remember:

- `findViewById` was error-prone and verbose.
- **ViewBinding** provides type-safe, null-safe, and boilerplate-free access to views.
- In fragments, always clear the binding in `onDestroyView`, or use a **delegate** to automate the cleanup.

Recipe 7: Using by lazy for view and resource initialization.

In Android apps, certain resources—such as views, database clients, or configuration values—are often not needed immediately at object creation. Initializing them too early can waste memory or processing time. Kotlin's `by lazy` delegate gives you a clean way to **delay initialization until the first time the property is accessed**.

* * *

What is by `lazy`?

In Kotlin, `by lazy` is a **property delegate**. Instead of immediately assigning a value to a property, you hand over the responsibility of initialization to the `lazy { ... }` block. This block runs **only once**—the first time you access the property. After that, the computed value is cached and reused.

Think of it as saying:

"Don't do the work now. Do it later, the first time I actually need it."

By default, `by lazy` is **thread-safe**, which means multiple threads can safely access the property without creating duplicate instances.

Lazy-loading a view in an Activity

```
class MainActivity : AppCompatActivity() {

    // View binding only happens when buttonView is first accessed
    private val buttonView: Button by lazy {
        findViewById(R.id.myButton)
    }

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

        // First access — initialization happens here
        buttonView.setOnClickListener {
            Toast.makeText(this, "Clicked!", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Here, `buttonView` isn't created when the activity object is constructed. Instead, the first time you use `buttonView`, Kotlin runs the initializer (`findViewById(...)`) and caches the result. Any future calls just return the same object.

Lazy initialization for resources

```
class ConfigManager(context: Context) {

    val prefs: SharedPreferences by lazy {
        context.getSharedPreferences("app_prefs", Context.MODE_PRIVATE)
    }

    val apiBaseUrl: String by lazy {
        // Expensive or conditional loading (could be from a remote source)
        "https://api.example.com"
    }
}
```

Here, `prefs` and `apiBaseUrl` values are only created when first accessed. This helps keep your app lightweight at startup.

* * *

Why it's important

Without `by lazy`, you'd typically write something like:

```
private var buttonView: Button? = null

override fun onCreate(savedInstanceState: Bundle?) {
    buttonView = findViewById(R.id.myButton)
}
```

This introduces potential `null` checks and extra boilerplate; whereas using lazy initialization, you get a **non-nullable property** with concise and safe syntax.

Recipe 8: Lifecycle-aware coroutines with lifecycleScope.

When working with coroutines in Android, one common challenge is managing their lifecycle. If you launch a coroutine tied directly to an `Activity` or `Fragment` without proper scoping, it may keep running even after the UI component is destroyed, leading to crashes, memory leaks, or wasted work.

To solve this, AndroidX provides **`lifecycleScope`**, an extension property available in `ComponentActivity` and `Fragment`. Coroutines launched in `lifecycleScope` are automatically canceled when the corresponding lifecycle is destroyed.

A quick word about coroutines

Coroutines are Kotlin's modern approach to **asynchronous programming**, replacing older tools like `AsyncTask`, callbacks, or manually managed threads. They let you write asynchronous code in a **sequential style**, are **lightweight**, and support **structured concurrency**, so work can be neatly tied to a scope (like `lifecycleScope`).

> 💡 **Tip: What is AndroidX?** AndroidX is the modern replacement for the old Android Support Libraries. It provides backwards-compatible libraries with new features, bug fixes, and consistent package naming (`androidx.*`). The `lifecycleScope` API comes from `androidx.lifecycle:lifecycle-runtime-ktx`, which adds coroutine support to lifecycle-aware components like Activities and Fragments.

Example: Launching a Coroutine in a Fragment

```
class ProfileFragment : Fragment(R.layout.fragment_profile) {

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        // Launch coroutine tied to this Fragment's view lifecycle
        viewLifecycleOwner.lifecycleScope.launch {
            val userData = fetchUserData()
            renderUserData(userData)
        }
    }

    private suspend fun fetchUserData(): String {
        delay(2000) // simulate network/database delay
        return "Hello from lifecycleScope!"
    }

    private fun renderUserData(data: String) {
        Toast.makeText(requireContext(), data, Toast.LENGTH_SHORT).show()
    }
}
```

Example: Launching Coroutine in an Activity

```
class MainActivity : AppCompatActivity() {

    override fun onStart() {
        super.onStart()

        lifecycleScope.launch {
            repeat(5) { i ->
                delay(1000)
                Log.d("MainActivity", "Tick $i")
            }
        }
    }
}
```

* * *

Why Use `lifecycleScope`?

- **Lifecycle-aware**: Coroutines are automatically canceled when the lifecycle is destroyed.
- **Cleaner code**: No need to manually track and cancel `Job` objects.
- **Safer UI updates**: Prevents updating views after they've been destroyed.

> Tip: Use `viewLifecycleOwner.lifecycleScope` in Fragments when working with UI, since the Fragment object can survive longer than its views.

Recipe 9: Passing data between Activities/Fragments safely with Bundles.

Most Android apps you'll build will likely have more than one screen, so, you'll often need

to send data between screens – for example, from one `Activity` to another, or from a parent `Fragment` to a child.

The traditional way is to use **Intents** and **Bundles**:

- **Extras** □ Key-value pairs you attach to an `Intent` when launching an Activity.
- **Bundle** □ A container for storing structured data (key-value pairs) that can be passed around inside the Android framework. Both Activities and Fragments can receive a `Bundle` as part of their lifecycle.

This system works, but it has two main drawbacks:

1. Keys are just strings (easy to misspell).
2. Values must be manually cast to the right type (easy to mismatch).

Traditional Approach (Risky)

Here's how you might normally pass data:

```
// Activity A
val intent = Intent(this, DetailActivity::class.java)
intent.putExtra("USER_ID", 42)   // Extra added to Intent
startActivity(intent)

// Activity B
val userId = intent.getIntExtra("USER_ID", -1) // Extract from extras
```

For fragments, you usually pack arguments into a `Bundle`:

```
// Creating Fragment with arguments
val fragment = DetailFragment().apply {
    arguments = Bundle().apply {
        putString("USERNAME", "alice")
    }
}

// Inside DetailFragment
val username = arguments?.getString("USERNAME")
```

This works, but notice the problems:

- Keys like `"USER_ID"` and `"USERNAME"` are just raw strings.
- You have to remember the correct type (`Int` vs `String`).
- Mistakes only show up at runtime.

Safe Args with Navigation

Safe Args (part of Android Jetpack Navigation) solves this problem by generating type-safe classes for your arguments.

Define your arguments in `nav_graph.xml`:

```
<fragment
    android:id="@+id/homeFragment"
    android:name="com.example.HomeFragment" >
    <action
        android:id="@+id/action_home_to_detail"
        app:destination="@id/detailFragment" >
        <argument
            android:name="userId"
            app:argType="integer" />
    </action>
</fragment>
```

Then use them like this:

```
// From HomeFragment
val action = HomeFragmentDirections.actionHomeToDetail(userId = 42)
findNavController().navigate(action)

// In DetailFragment
val args: DetailFragmentArgs by navArgs()
val userId = args.userId
```

No more raw keys, no casting. The compiler enforces correctness.

* * *

Tip: Safe Args works for both Activities and Fragments as long as they're defined in your Navigation Graph.

* * *

Why Use Safe Args?

- Eliminates fragile string keys.

- Type-safe — compiler catches mistakes early.
- Cleaner, easier-to-read code.

If you're starting a new Android project, consider setting up Navigation and Safe Args right away. It will save you time and prevent subtle bugs when passing data.

Recipe 10: Toasts & Snackbars as one-shot events (SharedFlow)

In Android apps, you often need to show **temporary messages** — things like **toasts** or **snackbars**. These are **one-time messages**: the user should see them once, and then they disappear.

For example, after the user saves a form, you might want to show:

> "Profile updated successfully!"

The problem is that Android UI works with lifecycles. If you use something like `LiveData` or a state variable to store this message, it can get delivered again when the screen is rotated or recreated. The result? The same toast or snackbar shows up multiple times. Not good.

This is where **Kotlin's `SharedFlow`** comes in. It's a good tool for one-time messages because it only delivers values to collectors that are actively listening, and (if you set it up with `replay = 0`) it won't repeat past messages.

Using `SharedFlow` for UI Messages

In your **ViewModel**:

```
class ProfileViewModel : ViewModel() {

    private val _events = MutableSharedFlow<UiEvent>()
    val events: SharedFlow<UiEvent> = _events

    fun saveProfile() {
        // Do your save logic
        viewModelScope.launch {
            // After successful save, emit a one-time message
            _events.emit(UiEvent.ShowMessage("Profile updated successfully!"))
        }
    }
}

sealed class UiEvent {
    data class ShowMessage(val message: String) : UiEvent()
}
```

Collecting the Messages in the UI Layer

In your **Fragment** (or Activity), collect the events inside a lifecycle-aware scope:

```
class ProfileFragment : Fragment(R.layout.fragment_profile) {

    private val viewModel: ProfileViewModel by viewModels()

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        viewLifecycleOwner.lifecycleScope.launch {
            viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {
                viewModel.events.collect { event ->
                    when (event) {
                        is UiEvent.ShowMessage -> {
                            Snackbar.make(requireView(), event.message,
                            ↪  Snackbar.LENGTH_SHORT).show()
                        }
                    }
                }
            }
        }
    }
}
```

Because the `SharedFlow` is set with `replay = 0` by default, the message is **shown only once** and won't be repeated on screen rotation.

Why not just use a `String` or `LiveData`?

You might wonder: *"Why not just expose a `String` from the ViewModel and read it in the Fragment?"*

That works for the simplest cases, but it has problems:

- A plain `String` is just **data**. The ViewModel would have no way to tell the Fragment *when* the message is ready. You'd have to check or poll constantly.
- It also has no **lifecycle awareness**. If the Fragment is destroyed and recreated, you either lose the message or risk showing it again.

What about `LiveData`?

- `LiveData` is better because the ViewModel can push updates to the UI.
- But `LiveData` **replays the last value** to every new observer. That means after a screen rotation, the message gets delivered again – causing duplicate snackbars or toasts.

This is where `SharedFlow` fits in:

- It behaves like an event stream, not a stored state.
- With `replay = 0`, it only delivers new events to active collectors.
- That means your one-time messages are delivered exactly once, when they happen – and nowhere else.

What's key to remember:

- Use a **String** only for fixed text, not for events.
- Use **LiveData** for *state that should persist* (e.g. the current username).
- Use **SharedFlow** for *events that should happen once* (e.g. a toast or snackbar).

* * *

Tip: If you really need to "remember" the last event (like showing an error after a network failure), you can configure `SharedFlow(replay = 1)`. For simple toasts and snackbars, stick with the default (`replay = 0`) so messages only appear once.

* * *

Data & State Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 11: Data classes with copy() for immutability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Example: Updating a Profile

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Why immutability matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Copying with no changes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Copying selectively

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Android Example: Updating UI State in a ViewModel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Why not just mutable POJOs?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 12: Sealed classes for UI states (Loading, Error, Success)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Problem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Solution: Use a sealed class

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Using it in the ViewModel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Consuming the state in your UI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Why is this better?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 13: Safe Args in Navigation (type-safe arguments)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Setting up Safe Args

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Defining Arguments in the Navigation Graph

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Passing Data with Safe Args

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Receiving Data in the Destination Fragment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Why This Works

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 14: Persisting small UI state with SavedStateHandle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Using SavedStateHandle in ViewModels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Consuming in a Fragment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

When to Use SavedStateHandle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Tip

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 15: SharedPreferences → DataStore migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

1. Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

2. Automatic Migration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

3. Manual Migration

4. Reading/Writing DataStore (Common)

Takeaways:

Networking & Persistence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 16: Retrofit + Coroutines integration.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 17: Safe retry with exponential backoff for API calls.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 18: Repository pattern: API + Room cache.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 19: Room migrations (avoid crashes on schema changes).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 20: Mapping API models → domain models with extensions.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Coroutines & Flow in Android

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 21: Using viewModelScope.launch correctly.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 22: Switching threads with withContext(Dispatchers.IO).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 23: Handling cancellation in coroutines.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 24: Flow + debounce for search bars.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 25: Testing coroutines with runTest + TestDispatcher.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Jetpack Compose Essentials

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 26: State hoisting in Compose (lifting state up).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 27: Using rememberSaveable to persist across rotations.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 28: LazyColumn best practices (keys, avoiding recomposition).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 29: Preview parameter providers for test data.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 30: Theming with Material 3 in Kotlin.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Dependency Injection & Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 31: ViewModel Assisted Injection with Hilt.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 32: Modularizing Android projects (domain/data/ui).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 33: Using inline value classes for stronger typing.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 34: Creating a sealed error hierarchy (AppException).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 35: Clean MVVM pattern in Kotlin (with Repository + UseCases).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Advanced Kotlin in Android

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 36: Reducing method count (@JvmField, const val).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 37: Avoiding performance pitfalls (lambdas, allocations).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 38: Interop with Java: @JvmStatic, @JvmOverloads.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 39: SupervisorScope for independent coroutines.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Recipe 40: WorkManager + CoroutineWorker for resilient background tasks.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Appendix A: Beginner to Intermediate Recipes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Using apply, let, also, run scope functions (when to use each).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Creating companion objects as factories instead of static helpers.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Using when expression instead of multiple if/else.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Default values in data classes for cleaner constructors.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Smart casts with is operator (safe type checks).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Null coalescing with ?: operator.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Using lateinit safely (when it's OK, when it's not).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

String resource formatting with placeholders (getString(R.string.hello, name)).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Simple sealed class navigation (before using Jetpack Navigation).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Handling RecyclerView clicks with lambdas instead of interfaces.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Using const val for compile-time constants.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Testing simple functions with JUnit & Kotlin test DSL.

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Creating inline helper extensions (fun View.show() / fun View.hide()).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Using Pair and Triple effectively (vs. creating a new class).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.

Filtering & mapping lists with Kotlin collection operators (filter, map, firstOrNull).

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/55kotlinrecipesforandroidprogramming.