1 Kotlin

1.1 Data class

auto getter/setter/toString/properties/copy
`data class Artist(var id: Long, var name: String)`
destructure : `val(name,surname,age) = p`

1.2 Null safety

- `a?.name` → a.name else null / compile seulement avec `val n: String? = a?.name`
- `!!` → KotlinNullPointerException si b est null
- `?:` → `c?.name ?: "Unknown"` default if null

1.3 Extension functions

Ability to add any function to any class

1.4 Classes

- inherit of Any and are final()
- `class Person(private var name: ...)` → set private attribute
  w/o `private` → public
- multiple constructors → use ‘init‘ and ‘constructor()‘
- default getter/setter → can't be modified if created by default constructor

1.5 Inheritance

Use `open class...` → `class Student(name:String,var uni:String): Person(name,surname)` (you have to specify super constructor)

1.6 Collections

- `List, Set, Map` and `MutableList/Set/Map`
- `any, count, max, filter, map, partition, + (concatenate lists)`
- `.partition { it % 2 == 0 } = Pair(List<2,4>, List<1,3>)`

1.7 Operator overloading

- `==` is Java `equals`, `===` is Java `==`

1.8 When

Similar to switch case

```
when(view) {
  is TextView -> view.text = "Hello"
}
val res = when(x){
  0, 1 -> "binary"
  else -> "error"
}
```

1.9 Exceptions

Not mandatory to handle exceptions

1.10 Scope functions

1.10.1 let

block executed only if p is non null, no return value

```
var p : Person? = null
p?.let { it.age = 23 }
```

1.10.2 apply

block executed returning the edited value

```
Calendar.getInstance().apply{ set(Calendar.MONTH, 4) }
```

1.10.3 use

For closeable objects, automatically closes after

```
Writer("file").use { it.appendLine("stuff") }
```

1.11 Companion objects

Equivalent to static, has values, variables, methods...

2 Android Resources

2.1 Manifest File

Build tools, smartphone OS + store requirements :

- App components : activities, services, broadcast receiver, content providers
- Permissions
- Hardware/software functionalities needed

2.2 Resources

Will contain all files and static content used by app. Best practice : separate resources from code → better maintenance

2.2.1 Values

- Textual (string.xml) with Plural management possible (one/other)
- dimension.xml, use `dp`
- colors.xml
- themes.xml

2.2.2 Drawables

- Bitmap fields : each image has multiple sizes/definitions (+ optimization depending on a phone, we want to avoid a high def for a 'small' phone or the opposite)
- Vector
- Nine-patch : controlled resizing (we don't want to distor the folder image)
- State list (pressed/focused/hovered)
- Level list multi images (e.g 1-4 wifi bars)

2.2.3 Layout

ViewGroups organize the display of views (**Layout**, e.g., LinearLayout)

- Views : graphical elements (**widgets**, e.g., Button)

Main types :

- LinearLayout : horizontal/vertical
- RelativeLayout : relative to parent and Views/ViewsGroups
- ConstraintLayout : better performance and integration compared to Relative
- ScrollView : no nesting

2.2.4 Resource Contextualization

Resources adapt to device configuration at **runtime** (language, screen size/orientation, Android version, etc.) by using qualified directories :

```
res/
    values/strings.xml          # Default
    values-fr/strings.xml       # French
    values-fr-land/strings.xml  # French landscape
    layout-sw600dp/main.xml     # 7" tablets
```

Key points :

- Multiple qualifiers must follow strict ordering
- **Default** resources (without qualifiers) **must always be provided**
- System selects best match, falling back by removing qualifiers right-to-left

2.3 R Class

During build time, the Android Gradle Plugin generates a final class `R` containing static references to all application resources. The class is generated in the app's root package.

2.3.1 Package Structure

- App resources : generated in app's package (e.g., `com.example.myapp.R`)
- Library resources : each library has its own `R` class in its package
- Android framework resources : accessible via `android.R`

2.3.2 Resource References

From XML :

- App resources : `@id/my_view`
- Android resources : `@android:id/text1`

From code :

```
// App resources
setContentView(R.layout.activity_main)
findViewById<Button>(R.id.my_button)

// Android framework resources
textView.setTextColor(android.R.color.black)

// Library resources (e.g., Material components)
Snackbar.make(view, R.string.ready, LENGTH_SHORT)
```

2.3.3 Build Process

- `R` class is generated during resource compilation phase
- Each resource gets a unique integer ID
- IDs remain constant during app execution but may change between builds
- Resource references are replaced with these IDs at compile time

2.3.4 Build Scripts

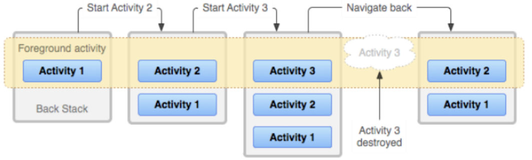
- Gradle : manage package/deps, compilation
- 2 build.gradle files : 1 for whole project, 1 for app (project module)
- Packages retrieved using maven (groupId, artifactId, version)

3 Activity, Fragments, Services

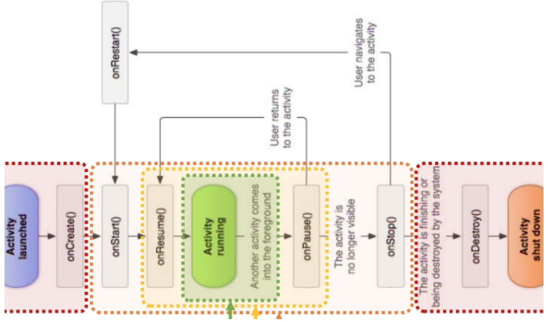
3.1 Activity

Represents a layout of an app

- Must be in Manifest file
- activity in a stack



3.1.1 Lifecycle


onRestart()
User navigates to the activity
User returns to the activity
Activity launched
onCreate()
onStart()
onResume()
Activity running
Another activity comes into the foreground
onPause()
The activity is no longer visible
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Activities have a specific lifecycle managed by the system (inversion of control). Never instantiate activities directly with constructors.

3.1.1.1 Lifecycle Methods

- `onCreate()` : Called when activity created or recreated after being killed by system. Setup UI, initialize data.
- `onStart()` : Activity becoming visible but not yet interactive.
- `onResume()` : Activity gains focus, can interact with user.
- `onPause()` : Activity losing focus but still visible (e.g., split-screen).
- `onStop()` : Activity no longer visible.
- `onDestroy()` : Activity being destroyed.

3.1.1.2 Common Triggers

- User navigates between apps : `onPause()` → `onStop()`
- Screen rotation : `onPause()` → `onStop()` → `onDestroy()` → `onCreate()`
- Split-screen activation : `onPause()` (activity remains visible)
- Back button : `onPause()` → `onStop()` → `onDestroy()`
- System kills background activity : `onDestroy()`
- Dialog opens : `onPause()` (activity partially visible)

3.1.1.3 Activity States

- **Active** : Top of stack, visible and interactive
- **Paused** : Visible/partially visible, no focus
- **Stopped** : Not visible, in memory
- **Inactive** : Temporary state when created/killed

Note : The system can kill paused or stopped activities to reclaim resources. Activities must save their state in `onSaveInstanceState()`.

3.1.2 First Steps

Override onCreate (mandatory) :

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

- Inherit from Activity/AppCompatActivity
- complexity hidden from inheritance

Interactions :

- `findViewById<Button>(R.id.my_btn)` : searches item and will return corresponding view and return the object reference. Search can be heavy. → use **lateinit var** to avoid call findViewById for each view interaction
- `btn.setOnClickListener{}`

3.1.3 Intents

Mechanism to ask the system to start an activity : By default, in Manifest the **app entry** will use an Intent

```
val i = Intent(this, MySecondActivity::class.java)
startActivity(i)
```

Intent types and behavior :

- Launch activities : added to stack. When ended, pop from stack and return to previous one
- End activity : `back` button - default behavior should be preserved
  - Overrideable using `addCallback {}`
  - Can use `finish()` to end activity
- Explicit (same app, e.g., MySecondActivity)
- Implicit (other app), e.g. Open web page, send mail/message, use camera

3.1.4 Activity Key/Value Data

Intents can carry data between activities using key-value pairs :

- Put extras using `putExtra()`
- Retrieve using appropriate getter method (`getStringExtra()`, etc.)
- Bundle for complex data structures

3.1.5 Contracts

Activity result contracts provide a type-safe way to :

- Pass data between activities
- Handle activity results
- Register for callbacks
- Manage permissions

3.1.6 Activity Save/Restore

Use `onSaveInstanceState(outState: Bundle)` and `onRestoreInstanceState(savedState: Bundle)`. Autosaves widgets with **ID**. The bundle will be passed to the new instance to `onCreate(savedSte: Bundle)`. E.g save a counter value and on rotate will destroy the activity but with onCreate will retrieve the saved counter value.

3.2 Fragments

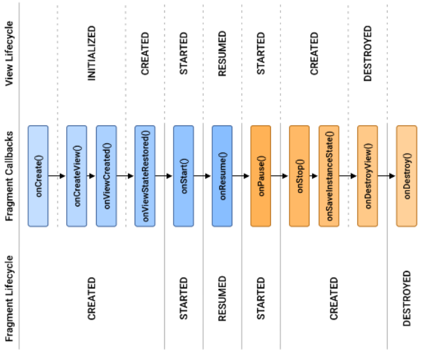
Reuse GUI, divide interface with multiple Fragments. Fragments will berun in a host Activity.

3.2.1 Fragment Lifecycle

Fragment lifecycle includes additional states and callbacks compared to activities. The fragment manager handles state transitions and manages the fragment backstack.

Key lifecycle states :

- Created
- Started
- Resumed
- Paused
- Stopped
- Destroyed



Managed by `FragmentManager` (transactions) :

- Different state transitions
- Add/pop in main activity + manage stack

3.2.2 Fragments Overview

Add fragment :

```
<androidx.fragment.app.FragmentContainerView
  android:id="@+id/framelayout"
  android:name="package.MyFragment"/>
supportFragmentManager.beginTransaction()
  .replace(R.id.framelayout, MyFragment.newInstance())
  .commit()
```

Screen rotation : recreates but preserves internal state

3.2.3 Data Exchange Activity/Fragment

- Activity → Fragment : Fragment can use Activity public methods
- Fragment → Activity : Fragment can use `getActivity`, activity can be **null**
- Fragment → Fragment : Via Activity : Frag1 → Act → Frag2

3.3 Services

For long operations in background, executed **only** in main thread (UI-Thread)

Types :

- Foreground : linked to visible notification (download, player)
- Background : no UI, time limited (server sync, save)
- Bounded : linked to app (activity), destroyed when no more links

3.3.1 Background and Foreground

- `startForegroundService()` and `startService()`
- `startFService` has 5 sec to be in front using `startF`
- `stopService` with Intent and `stopSelf()`
- Background service lives only minutes after app is in background

3.3.2 Bounded

`bindService()` + `unbindService()`, can bind to fg/bg service. Enables Service-Activity communication, otherwise use LocalBroadcast.

Example Background :

```
val i = Intent(this, MyService::class.java)
startService(i)
```

`onStartCommand()` flags :

- START_STICKY : restart service ASAP (null intent)
- START_NOT_STICKY : no restart
- START_REDELIVER_INTENT : restart with original intent

Example Bounded :

```
override fun onStart() {
    super.onStart()
    val i = Intent(this, MyService::class.java)
    bindService(i, connection, BIND_AUTO_CREATE)
}
override fun onPause() {
    super.onPause()
    unbindService(connection)
    mBound = false
}
```

When bound : `if(mBound) mService.startThread()`

4 Broadcast Receiver

Pub/sub system for messages. Register in Manifest :

```
<receiver android:name=".MyBroadcastReceiver"
↪  android:exported="true">

class MyBroadcastReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent:
    ↪  Intent) {
        when(intent.action) {
            Intent.ACTION_LOCALE_CHANGED -> {} (...)
```

5 Content Providers

- Access to centralized DB data
- Standardized communication between apps
- Data identified by URI (e.g., `content://contacts/people/1`)
- CRUD operations
- Can impose permissions

5.1 File Provider

- Access to non-structured data (files)
- Shared storage : all apps access, `READ_EXTERNAL_STORAGE`
- Private storage : app-only (sandboxing)
- Temporary access via URI to private storage files

6 Permissions

Levels :

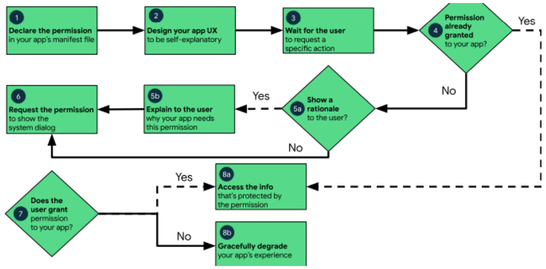
- Installation : in Manifest, auto-granted
- Execution : 'dangerous' permissions, **requires user grant**
- Special : system apps or manufacturers only

Best practices :

- Control : user choice to grant
- Transparency : clear permission purpose
- Minimal : necessary permissions only

Example :

- Download image : requires `<uses-permission android:name="android.permission.INTERNET" />`
- List mobile : `READ_PHONE_STATE` (dangerous) requires explicit grant


1 Declare the permission in your app's manifest file
2 Design your app UX to be self-explanatory
3 Wait for the user to request a specific action
4 Permission already granted to your app?
Yes
No
5a Show a rationale to the user?
Yes
No
5b Explain to the user why your app needs this permission
6 Request the permission to show the system dialog
7 Does the user grant permission to your app?
Yes
No
8a Access the info that's protected by the permission
8b Gracefully degrade your app's experience

7 Graphical Interface

7.1 View Visibility

Three possible states :

- **VISIBLE** : Default state, view is visible
- **INVISIBLE** : View not displayed but space reserved in layout
- **GONE** : View hidden and no space reserved (as if never added, size = 0)

Modification through XML or code :

```
<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:visibility="gone" />
view.visibility = View.GONE
```

7.2 Main View Types

7.2.1 TextView and EditText

7.2.1.1 TextView for displaying text

- Formatting : bold, italic, size, color
- Basic HTML support (`<b>`, `<i>`, etc.)
- `android:textIsSelectable` for text copying

7.2.1.2 EditText for user input

- `inputType` : `text`, `textPassword`, `number`, `phone`
- `hint` for input guidance
- Listeners : `TextWatcher` for input events

7.2.2 Button and ImageButton

- Click handling :

```
button.setOnClickListener { // Action on click }
button.setOnLongClickListener {
    // Action on long click
    true // return mandatory
}
```

- Icon support with `drawableLeft/Right/Top/Bottom` or Material Design

7.2.3 ImageView

- Displays images from resources or memory
- `scaleType` to control resizing :
  - `fitCenter` : Resizes to fit within bounds
  - `centerCrop` : Fills by cropping if necessary
  - `fitXY` : Stretches to fill
- Asynchronous loading required for online images

7.2.4 Selection Components

7.2.4.1 CheckBox, Switch, ToggleButton

```
switch.setOnCheckedChangeListener { _, isChecked ->
    if (isChecked) {
        // Enabled
    } else {
        // Disabled
    }
}
```

7.2.4.2 RadioGroup and RadioButtons

- Non-cancellable single choice
- Group management via `RadioGroup`
- Events via `setOnCheckedChangeListener`

7.2.4.3 Spinner

- Dropdown list
- Data source : `string-array` or `Adapter`
- Adapter enables dynamic list and updates

7.2.5 Progress Bars

7.2.5.1 ProgressBar

- Indeterminate mode : continuous animation
- Determinate mode : progress 0-100
- Horizontal or circular

7.2.5.2 SeekBar (inherits from ProgressBar)

- User interaction to set value
- `setOnSeekBarChangeListener` for events

7.2.6 WebView

- Displays web content in app
- Requires Internet permission for online content
- JavaScript configuration :

```
webView.settings.javaScriptEnabled = true
```

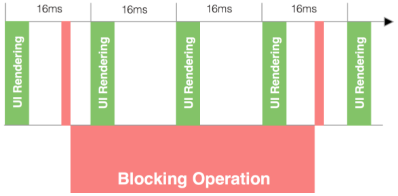
- JavaScript-Android interface possible

7.3 UI-Thread and Background Operations

- UI-Thread : main thread for user interface
- Long operations must run in background

Example :

```
val imageView = findViewById<ImageView>(R.id.image)
thread {
  val url = URL("https://example.com/image.jpg")
  val bmp = BitmapFactory.decodeStream(
    url.openConnection().getInputStream()
  )
  runOnUiThread {
    imageView.setImageBitmap(bmp)
  }
}
```



7.4 Custom Views

7.4.1 Creation

- Inherit from `View` or subclass
- `@JvmOverloads` for multiple constructors
- Implement `onDraw()` and `onTouchEvent()`

7.4.2 Custom Attributes

```
<declare-styleable name="CustomView">
    <attr name="customAttribute" format="string" />
</declare-styleable>
```

7.5 Material Components

Enhanced TextField :

- Error handling
- Start/end icons
- Hint animation

7.6 User Feedback

- Toast : simple temporary message
- Snackbar : message with possible action
- Dialog : user decision or input

7.7 Notifications

- Asynchronous, outside application
- Require channel since Android 8.0
- Actions via `PendingIntent`
- Can be expandable

Notification channel :

```
val channel = NotificationChannel(CHANNEL_ID, name,
↪  importance).apply {
    description = descriptionText
}
```

7.8 ActionBar

Main navigation with configurable icons and text

```
override fun onCreateOptionsMenu(menu: Menu): Boolean
↪  {
    menuInflater.inflate(R.menu.main_menu, menu)
    return true
}
```

7.9 ListView vs RecyclerView vs ScrollView

7.9.1 ScrollView

- Single child container allowing content to scroll
- No view recycling (all content loaded in memory)
- Cannot be nested with itself
- Use `NestedScrollView` (AndroidX) for nesting support
- Best for static content that exceeds screen size

7.9.2 ListView

Pros :

- Simpler implementation
- Built-in `OnItemClickListener`
- Easier header/footer management
- Default item animations

Cons :

- No enforced ViewHolder pattern
- Single layout type for all items
- Poor performance with large datasets
- Limited customization
- All data updates require full refresh

7.9.3 RecyclerView

Pros :

- Enforced ViewHolder pattern
- Multiple view types support
- Better memory efficiency through view recycling
- Customizable item animations
- Layout managers (`Linear`, `Grid`, `Staggered Grid`)
- `DiffUtil` for efficient updates
- Supports both vertical and horizontal scrolling
- Item decorations and spacing

Cons :

- More complex implementation
- No built-in click listeners
- Requires more boilerplate code
- Header/footer implementation more complex

When to use each :

- **ScrollView** : Static content, forms, or detail views
- **ListView** : Simple lists with single layout type
- **RecyclerView** : Complex lists, multiple view types, or large datasets

7.10 Additional Widgets

7.10.1 Floating Action Button (FAB)

- Material Design floating button
- Customizable animations
- Typical position at bottom right

7.10.2 Gesture Detection

- Via `GestureDetectorCompat`
- Detects : single/double tap, scroll, fling, long press

```
val detector = GestureDetectorCompat(this, object :
↪  GestureDetector.SimpleOnGestureListener() {
    override fun onDoubleTap(e: MotionEvent): Boolean
    ↪  {
        // Handle double tap
        return true
    }
})
```

8 LiveData and MVVM Architecture

8.1 LiveData

LiveData is a Jetpack lifecycle-aware observable data holder class.

8.1.1 Key benefits

- Automatic UI updates when data changes or observer becomes active
- No memory leaks (automatic cleanup)
- Thread-safe : Observers called on main thread
- Survives configuration changes
- LiveData created in ViewModel : replace state save of a re-created Activity and can share data/events between several components (Activity + Fragments)
- LiveData **immutable**, use MutableLiveData if changes needed
- Update value : **sync** if **UI-Thread**, asyn if any thread (data.postValue(1))

8.1.2 Observation

8.1.2.1 From activity

```
data.observe(this) { value ->
textview.text = "$value"}
```

- lifecycleOwner : activity itself
- Callback called in UI-Thread, for each value changes and when the activity is or becomes active/visible

8.1.2.2 From fragment

```
data.observe(viewLifecycleOwner) { value ->
↪  textview.text = "$value" }
```

- lifecycleOwner : viewLifecycleOwner → returns Fragment's view lifecycle.

8.1.3 Implementation

```
private val _data = MutableLiveData<Type>()
val data: LiveData<Type> = _data // Public immutable
↪  exposure
```

8.1.4 Advanced features

- **Transformations** : Map or switchMap operations
- **MediatorLiveData** : Merge multiple LiveData sources
- **List handling** : Full list decapsulation required for modifications

8.2 MVC in Android

Traditional MVC pattern doesn't directly map to Android architecture :

8.2.1 Basic Android MVC Structure

- **Model** : Data and business logic
- **View** : XML layouts and widgets
- **Controller** : Activities/Fragments

Key differences from canonical MVC :

- Views cannot directly interact with Model
- Controller (Activity/Fragment) must mediate all interactions
- Tight coupling between View and Controller

8.2.2 Controller Responsibilities

Activities/Fragments accumulate multiple responsibilities :

- View management (instantiation, updates)
- User action handling
- System API calls (sensors, Bluetooth, permissions)
- Data loading and processing
- Lifecycle management

8.2.3 Lifecycle Challenges

Major issues with Android's MVC implementation :

- **State Management** :
  - UI state lost on configuration changes
  - `savedInstanceState` inadequate for complex data
  - Temporary data destroyed without control
- **Async Operations** :
  - Ongoing operations may outlive Activity
  - Complex cleanup required
  - Resource waste from interrupted requests
  - Difficult to handle rotation during async operations

8.2.4 Architectural Problems

- Monolithic Activities
- Poor separation of concerns
- Difficult to test
- Complex state management
- No clear data ownership
- Lifecycle-dependent business logic

This leads to :

- Complex, hard to maintain code
- Difficult unit testing
- Poor reusability
- Lifecycle-related bugs

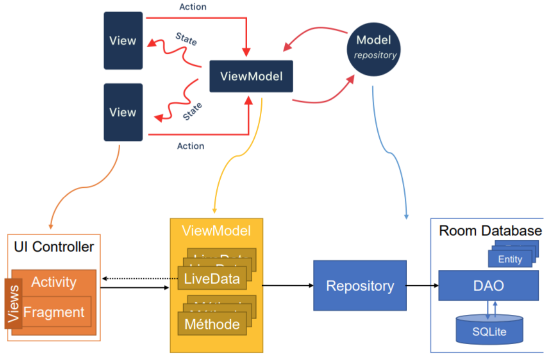
8.3 MVVM Architecture

MVVM separates concerns into :

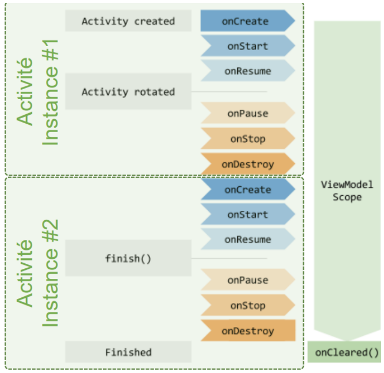
- **View** : UI layer (Activities/Fragments)
- **ViewModel** : UI logic and state holder
- **Model** : Business logic and data operations

Advantages over MVC :

- Clear separation of concerns
- Better testability (ViewModel has no Android dependencies)
- Survives configuration changes
- Handles async operations safely



8.4 ViewModel Lifecycle



Key characteristics :

- Survives Activity/Fragment recreation
- Destroyed only when Activity finished or Fragment detached
- Scope larger than Activity but smaller than Application
- ViewModel is link to one and only Activity and its fragments. Using another Activity with ViewModel will result to create a new VieModel instance
- Created lazily on first request

```
class MyViewModel : ViewModel() {
    override fun onCleared() {
        // Called when ViewModel is being destroyed
        // Clean up resources
    }
}
```

8.5 SavedState with ViewModel

While ViewModel survives configuration changes, it doesn't survive process death. Solutions :

- **SavedStateHandle** : For simple data
- **RemoteMediator** : For complex data requiring reload
- **Room** : For persistent data

```
class MyViewModel(private val savedStateHandle:
↪  SavedStateHandle) : ViewModel() {
    var state: Type
        get() = savedStateHandle.get<Type>(KEY) ?:
        ↪  defaultValue
        set(value) = savedStateHandle.set(KEY, value)
}
```

8.6 Architecture Best Practices

8.6.1 LiveData Placement

- Repository : Use Flow/Coroutines
- ViewModel : Convert to LiveData
- UI : Observe LiveData only

8.6.2 ViewModel Best Practices

- No View/Activity/Context references
- Expose immutable LiveData
- Handle process death with SavedStateHandle
- Use Coroutines for async operations
- Factory for dependency injection

8.6.3 Common pitfalls to avoid

- Storing View references
- Using Activity context
- Exposing MutableLiveData
- Heavy operations in ViewModel constructor

8.7 Jetpack Integration

Benefits of using Jetpack MVVM :

- Lifecycle awareness built-in
- SavedState handling
- Coroutines integration
- Room compatibility
- Navigation component support
- Easy testing with ViewModelScope

9 Android Data Persistence

9.1 Overview

Android offers multiple data persistence options :

- File storage (private or shared)
- Preferences
- local DB SQLite

Storage Type	Permissions	App Access	Removal
Private Files (Internal)	None	Private	On uninstall
Private Files (External)	None (API 19+)	Private	On uninstall
Media Files	RD_EXT_STRG	Shared	Persists
Shared Files	None (SAF)	Shared	Persists
Preferences	None	Private	On uninstall
Local Database	None	Private	On uninstall

9.2 File Storage

9.2.1 Private Storage

Internal storage :

- Accessed via `filesDir` and `cacheDir`
- Automatically encrypted since Android 10
- Limited space, careful management needed

External storage :

- May be emulated if no physical SD card
- Check availability with `Environment` methods
- Multiple external volumes possible
- Access via `getExternalFilesDir(null)` and `externalCacheDir`
- **Never store** absolute path since it can change

9.2.2 Shared Media File

- shared files (images, videos,...) which can be used for other app have centralized storage.
- needs `READ_EXTERNAL_STORAGE` if images are not created from app.
- API MediaStore uses query for finding content.

9.3 SharedPreferences

Key-value storage for **simple** data :

- XML-based storage
- Synchronous (`commit()`) or asynchronous (`apply()`) → preferred usage is apply
- Supports primitive types
- Accessible through high-level API

Example usage :

```
val prefs = getSharedPreferences("filename",
↪  Context.MODE_PRIVATE)
prefs.edit {
    putString("key", "value")
    putInt("counter", 42)
}
```

9.4 Room Database

Modern database solution with :

- ORM capabilities
- Compile-time verification
- LiveData/Flow integration
- Relationship support between entities

9.4.1 Components

- **Entities** : Data classes representing tables
- **DAO** : Interfaces defining data access methods (rw)
- **Database** : Abstract class defining database configuration
- **Repository** : Single source of truth for data operations. DAO encapsulation and calling IO methods has to be done in a dedicated thread or coroutine

9.4.1.1 Data Access Objects (DAO)

DAOs define the interface for database operations. Methods are annotated to specify SQL operations :

```
@Dao
interface UserDao {
    @Transaction
    @Query("SELECT * FROM user")
    fun getAll(): LiveData<List<User>>
    @Insert
    fun insert(user: User): Long
    @Update
    fun update(user: User)
    @Delete
    fun delete(user: User)
    @Query("SELECT * FROM user WHERE age > :minAge")
    fun getOlderThan(minAge: Int): List<User>
}
```

Room generates all necessary code at compile time using KSP (Kotlin Symbol Processing).

9.4.1.2 Type Converters

Converters handle complex types that Room can't store directly :

```
class DateConverter {
    @TypeConverter
    fun fromTimestamp(value: Long?): Date? {
        return value?.let { Date(it) }
    }
    @TypeConverter
    fun dateToTimestamp(date: Date?): Long? {
        return date?.time
    }
}
```

Register converters at database level with `@TypeConverters` annotation.

9.4.1.3 Entity Relationships

Room supports various relationship types :

9.4.1.3.1 Embedded Objects

```
data class Address(
    val street: String,
    val city: String
)
@Entity
data class User(
    @PrimaryKey val id: Int,
    val name: String,
    @Embedded val address: Address
)
```

9.4.1.3.2 One-to-One

```
data class UserAndLibrary(
    @Embedded val user: User,
    @Relation(
        parentColumn = "id",
        entityColumn = "userId"
    )
    val library: Library
)
```

9.4.1.3.3 One-to-Many

```
data class UserWithPets(
    @Embedded val user: User,
    @Relation(
        parentColumn = "id",
        entityColumn = "ownerId"
    )
    val pets: List<Pet>
)
```

9.4.1.3.4 Many-to-Many

```
@Entity
data class PlaylistSongCrossRef(
    @PrimaryKey val playlistId: Long,
    val songId: Long
)

data class PlaylistWithSongs(
    @Embedded val playlist: Playlist,
    @Relation(
        parentColumn = "playlistId",
        entityColumn = "songId",
        associateBy =
        ↪  Junction(PlaylistSongCrossRef::class)
    )
    val songs: List<Song>
)
```

9.4.1.4 Database Migration

Room handles schema changes through migrations :

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database:
    ↪  SupportSQLiteDatabase) {
        database.execSQL(
            "ALTER TABLE User ADD COLUMN last_update
            ↪  INTEGER"
        )
    }
}
Room.databaseBuilder(context, MyDb::class.java,
↪  "database")
    .addMigrations(MIGRATION_1_2)
    .build()
```

Migrations are crucial for preserving user data across app updates.

9.4.1.5 Database Creation

Database instance typically follows singleton pattern : → DB creation is a heavy operation, so we want to create only one instance and keep a reference. Singleton will be stored in app level.

```
@Database(
    entities = [User::class, Pet::class],
    version = 1,
    exportSchema = true
)
@TypeConverters(DateConverter::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    abstract fun petDao(): PetDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

        fun getDatabase(context: Context): AppDatabase
        ↪  {
            return INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(
                    context.applicationContext,
                    AppDatabase::class.java,
                    "app_database"
                ).build().also { INSTANCE = it }
            }
        }
    }
}
```

9.4.1.6 Performance Considerations

Key points for optimal Room usage :

- Use Suspend functions or LiveData for async operations
- Implement paging for large datasets
- Use transactions for multiple operations
- Cache complex query results
- Consider indices for frequently queried columns

9.4.2 Relationships

Supports :

- One-to-One
- One-to-Many
- Many-to-Many (with cross-reference table)
- Embedded objects

9.4.3 Best Practices

- Use Kotlin coroutines for async operations
- Implement Repository pattern
- Handle migrations properly
- Consider pagination for large datasets
- Use `distinctUntilChanged()` for LiveData queries
- Consider encryption needs (SQLCipher)

9.5 Architecture Overview

Recommended MVVM structure with Room :

- UI Controllers (Activities/Fragments) (e.g button to create a person)
- ViewModel with LiveData (e.g create entity person)
- Repository mediating data operations (**async** insert → prevent UI-Thread/coroutine lock)
- Room Database with DAOs
- Entities representing data structure

9.6 Alternative Solutions

Other database options :

- Couchbase Mobile (NoSQL)
- Firebase Realtime DB
- Nitrite-Java
- SQLCipher for encryption

10 Android Threading and Background Tasks

10.1 Threading Fundamentals

- UI-Thread handles all GUI interactions
- Network operations, I/O, and resource-intensive tasks must run on separate threads
- Results must return to UI-Thread for GUI updates

10.2 Implementation Methods

Basic Thread Creation :

```
val handler = Handler(Looper.getMainLooper()!!)
thread {
  val inputStream =
↪   url.openConnection().getInputStream()
  val bmp = BitmapFactory.decodeStream(inputStream)
  inputStream.close()
  handler.post { myImage.setImageBitmap(bmp) }
}
```

Handler Usage :

- Enables return to UI-Thread context
- Associates with specific thread
- Activity provides `runOnUiThread()` method

In an class extending Activity, the above code can be refactored :

```
thread {
  val bmp = BitmapFactory.decodeStream(
    url.openConnection().getInputStream()
  )
  runOnUiThread { myImage.setImageBitmap(bmp)
}
```

10.3 Thread Limitations

10.3.1 Memory Management Issues

- Anonymous subclasses retain Activity reference
- Activity cannot be garbage collected while threads are active
- Memory leaks occur during configuration changes
- WeakReferences required to prevent memory leaks

10.3.2 Concurrency Challenges

- Multiple parallel threads lack control and prioritization
- Resource sharing and execution order management required
- Thread interruption complex, especially with I/O operations
- No automatic thread cleanup on Activity destruction

10.3.3 Performance Concerns

- Uncontrolled thread creation impacts UI responsiveness
- Large thread pools consume excessive CPU resources
- ExecutorService available but unaware of Android lifecycle
- Thread persistence after Activity destruction

10.4 Best Practices

- Use WeakReferences for Activity references
- Implement proper thread interruption handling
- Consider thread pools for multiple operations
- Verify thread cleanup on Activity destruction

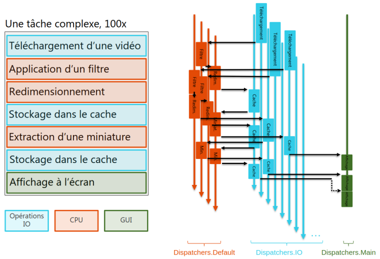
11 Kotlin Coroutines

11.1 Advantages Over Threads

- Sequential code style eliminates callback hell
- Multiple coroutines can run on single thread
- Built-in lifecycle awareness
- Automatic context switching optimization
- Memory efficient compared to thread creation

11.2 Dispatchers and Execution Context

Dispatcher	Thread Pool	Usage
Dispatchers.Main	UI Thread	GUI updates
Dispatchers.Default	CPU cores count	CPU-intensive tasks
Dispatchers.IO	Dynamic (max 64)	Blocking I/O operations



11.2.1 Dispatchers.IO Specifics

- Optimized for blocking I/O operations
- Dynamic thread pool allocation
- Prevents thread exhaustion
- Automatically scales up to 64 threads
- Ideal for network calls, file operations, database access

11.2.2 Dispatchers.Main Specifics

- Single-threaded dispatcher confined to UI thread
- Handles all UI updates and user interactions
- Required for modifying View properties
- Blocks UI if long operations run here
- Used automatically by Activity.runOnUiThread()

11.2.2.1 Usage Patterns

```
withContext(Dispatchers.Main) {
// Quick UI updates only
imageView.setImageBitmap(bitmap)
textView.text = result
}
```

11.2.3 Dispatchers.Default Specifics

- Thread pool size equals CPU cores
- Optimized for CPU-intensive tasks
- Shared between all coroutines using Default
- Used for computation, sorting, parsing
- Default dispatcher when none specified

11.2.3.1 Example Applications

- Complex calculations
- JSON parsing
- Image processing
- Data structure operations

```
withContext(Dispatchers.Default) {
val sorted = list.sorted() // CPU-intensive sorting
val filtered = data.filter { /* complex condition */ }
val processed = bmp.applyFilter() // image processing
}
```

11.3 Suspension vs Blocking

- **Blocking (Thread.sleep)** :
  - Blocks entire thread
  - Prevents other coroutines from using thread
  - Wastes system resources
- **Suspension (delay)** :
  - Releases thread during wait
  - Allows other coroutines to execute
  - Cooperative scheduling
  - Recommended approach

11.4 Coroutine Scopes

- **lifecycleScope** :
  - Bound to Activity/Fragment lifecycle
  - Auto-cancels on lifecycle end
  - Prevents memory leaks
- **viewModelScope** : For ViewModels
- **GlobalScope** : Application-wide (avoid use)

11.5 Suspending Functions

```
suspend fun downloadImage(url: URL): Bitmap? =
↪   withContext(Dispatchers.IO) {
  try {
    BitmapFactory.decodeStream(
      url.openConnection().getInputStream())
  } catch (e: IOException) {
    null
  }
}
```

11.6 suspendCoroutine Usage

- Converts callback-based APIs to suspend functions
- Bridges traditional callbacks with coroutines
- Example converting Volley :

```
suspend fun downloadHTML(url: String) =
↪   suspendCoroutine { cont ->
  val request = StringRequest(
    Request.Method.GET, url,
      { response -> cont.resume(response) },
      { error -> cont.resumeWithException(error) }
  )
  queue.add(request)
}
```

11.7 Best Practices

- Use suspending functions for I/O operations
- Prefer delay() over Thread.sleep()
- Choose appropriate dispatcher for task type
- Use lifecycleScope to prevent memory leaks
- Apply withContext() for context switching
- Convert callback APIs using suspendCoroutine

12 WorkManager

12.1 Purpose and Scope

- Handles persistent tasks surviving app restarts
- Manages tasks running in app background state
- Three task types : immediate, long-running (>10min), deferrable

12.2 System Integration

- Handles Android Doze mode (API 23+)
- Manages App Standby Buckets (API 28+)
- Adapts to App hibernation (API 30+)
- Uses SQLite for task persistence

12.3 Implementation

```
class MyWork(
  appContext: Context,
  params: WorkerParameters)
: Worker(appContext, params) {
  override fun doWork(): Result {
    return Result.success()
  }
}
// Periodic task setup
val constraints = Constraints.Builder()
  .setRequiresBatteryNotLow(true)
  .setRequiredNetworkType(NetworkType.UNMETERED)
  .setRequiresDeviceIdle(true)
  .build()
val workRequest =
↪   PeriodicWorkRequestBuilder<MyWork>(15,
↪   TimeUnit.MINUTES)
  .setConstraints(constraints)
  .setBackoffCriteria(
    BackoffPolicy.EXPONENTIAL,
    PeriodicWorkRequest.MIN_BACKOFF_MILLIS,
    TimeUnit.MILLISECONDS)
  .build()
WorkManager.getInstance(context).enqueue(workRequest)
```

12.4 Task Constraints

- Minimum periodic interval : 15 minutes
- Network conditions (metered/unmetered)
- Battery level requirements
- Device idle state
- Storage space requirements
- Charging state

13 Android Connectivity Management

13.1 Network Types

- WiFi : Local network access, prioritized
- Mobile networks (2-5G) : Cellular-based connectivity
- Simultaneous connections possible

13.2 Android API Integration

```
// Requires ACCESS_NETWORK_STATE permission
val connectivityManager = getSystemService(
  Context.CONNECTIVITY_SERVICE
) as ConnectivityManager

val networkCapabilities = connectivityManager
  .getNetworkCapabilities(
    connectivityManager.activeNetwork)
// Network state queries
val hasInternet = networkCapabilities
  ?.hasCapability(
    NetworkCapabilities.NET_CAPABILITY_INTERNET)
val isFreeToUse = networkCapabilities
  ?.hasCapability(
    NetworkCapabilities.NET_CAPABILITY_NOT_METERED)
val notRoaming = networkCapabilities
  ?.hasCapability(
    NetworkCapabilities.NET_CAPABILITY_NOT_ROAMING)
```

13.3 Security Configuration

```
<!- Manifest configuration for network security ->
<application android:networkSecurityConfig
  ="@xml/network_security_config"/>
<!- network_security_config.xml ->
<network-security-config>
  <domain-config cleartextTrafficPermitted="false">
    <domain includeSubdomains="true">
      domain.com
    </domain>
  </domain-config>
</network-security-config>
```

14 HTTP Communication Methods

14.1 java.net.URL vs Volley Comparison

Feature	java.net.URL	Volley
Threading	Manual management	Automatic
Caching	No built-in	Automatic
Request queueing	No	Yes
Memory management	Manual	Automatic
Image loading	Manual decode	Built-in
Error handling	Manual try-catch	Callback based
Kotlin coroutines	Direct support	Requires wrapper

14.2 Implementation Examples

14.2.1 java.net.URL

```
// PUT Request
val connection = url.openConnection()
  as HttpURLConnection
connection.apply {
    requestMethod = "PUT"
    doOutput = true
    setRequestProperty(
        "Content-Type",
        "application/json")
}
connection.outputStream
  .bufferedWriter(Charsets.UTF_8).use {
    it.append(jsonData) }
```

14.2.2 Volley

```
// GET Request
val queue = Volley.newRequestQueue(context)
val request = StringRequest(
    Request.Method.GET, url,
    { response -> handleSuccess(response) },
    { error -> handleError(error) }
)
queue.add(request)
```

14.3 Key Considerations

- **java.net.URL** :
  - Simple for basic operations
  - Better coroutine integration
  - Requires manual thread management
  - Synchronous nature requires explicit Dispatchers.IO
- **Volley** :
  - Better for complex applications
  - Built-in request queuing and caching
  - Automatic thread management
  - Requires callback-to-coroutine conversion

15 Data Synchronization

15.1 Local-Remote Synchronization Pattern

- Local database as source of truth
- Remote synchronization when network available
- Status tracking for local modifications

15.2 Database Structure

```
data class LocalEntity(
    val id: Long? = null, // Local primary key
    val remote_id: Long? = null, // Server reference
    val status: String, // "ok", "new", "mod", "del"
    // Entity fields
)
```

15.3 Synchronization States

- `ok` : Synchronized with server
- `new` : Local creation pending upload
- `mod` : Local modification pending upload
- `del` : Local deletion pending server notification

15.4 Implementation Pattern

```kotlin
class Repository(private val dao: LocalDAO) {
  suspend fun delete(item: LocalEntity) {
    withContext(Dispatchers.IO) {
      // Soft delete
      item.status = "del"
      dao.update(item)

      // Server sync attempt
      try {
        deleteFromServer(item.remote_id)
        dao.delete(item) // Hard delete after sync
      } catch(e: IOException) {
        // Will retry on next sync
      }
    }
  }

  suspend fun sync() {
    withContext(Dispatchers.IO) {
      // Upload new items
      dao.getNewItems().forEach { item ->
        try {
          val remoteId = uploadToServer(item)
          item.remote_id = remoteId
          item.status = "ok"
          dao.update(item)
        } catch(e: IOException) {}
      }

      // Process modifications
      dao.getModifiedItems().forEach { /* similar */ }

      // Process deletions
      dao.getDeletedItems().forEach { /* similar */ }
    }
  }
}
```

15.5 Conflict Resolution

- Server timestamp-based resolution
- Version tracking
- Merge strategies for concurrent modifications
- User intervention for unresolvable conflicts

16 Jetpack Compose

16.1 Core Concepts

- Declarative UI API for Kotlin
- Component-based architecture
- Direct state-UI relationship
- Simplified UI testing

16.2 Composable Functions

16.2.1 Basic Structure

```kotlin
@Composable
fun Greeting(name: String) {
  Text("Hello $name!")
}

class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MyTheme {
        Greeting("Android")
      }
    }
  }
}
```

16.2.2 Function Properties

- Must be fast to execute
- Avoid side effects
- Idempotent behavior
- Can be recomposed frequently (60fps)

16.3 Layouts

16.3.1 Basic Layouts

- `Column` : Vertical arrangement
- `Row` : Horizontal arrangement
- `Box` : Overlay arrangement

```kotlin
@Composable
fun LayoutExample() {
  Column(
    modifier = Modifier.fillMaxWidth(),
    horizontalAlignment = Alignment.CenterHorizontally
  ) {
    Text("Item 1")
    Text("Item 2")
  }
}
```

16.4 Preview

```kotlin
@Preview(showBackground = true)
@Composable
fun DefaultPreview() {
  MyTheme {
    Greeting("Android")
  }
}
```

16.5 Modifiers

- Chain of transformations
- Size, padding, appearance modifications
- Event handling
- Layout behavior

```kotlin
modifier = Modifier
  .fillMaxWidth()
  .height(56.dp)
  .padding(horizontal = 16.dp)
  .clickable { /* handler */ }
```

16.6 Lazy Composables

- Replace RecyclerView/ListView functionality
- Only render visible items
- No view recycling needed
- Support for item-level updates

16.6.1 Available Components

- `LazyColumn` : Vertical scrolling list
- `LazyRow` : Horizontal scrolling list
- `LazyVerticalGrid` : Grid layout

```kotlin
@Composable
fun LazyListExample() {
  val list = (1..10000).map { it.toString() }
  LazyColumn(modifier = Modifier.fillMaxSize()) {
    items(list) { item ->
      ListItem(item)
    }
  }
}

@Composable
fun ListItem(value: String) {
  Row(
    modifier = Modifier
      .fillMaxWidth()
      .height(48.dp)
      .padding(2.dp)
      .clickable { },
    horizontalArrangement = Arrangement.SpaceBetween,
    verticalAlignment = Alignment.CenterVertically
  ) {
    Text(text = value)
    Icon(/*...*/)
  }
}
```

16.6.2 Performance Considerations

- No manual ViewHolder pattern needed
- Automatic composition optimization
- State management per item
- Efficient item updates without full list recomposition

16.7 State Management

16.7.1 State Types

- `remember` : Preserves state during recomposition
- `rememberSaveable` : Survives activity recreation
- `LiveData` : Observable data holder
- `Flow` : Asynchronous data stream

16.7.2 State Declaration

```kotlin
@Composable
fun Counter() {
  var count by remember { mutableStateOf(0) }
  Button(onClick = { count++ }) {
    Text("Clicked $count times")
  }
}
```

16.7.3 State Hoisting

- Separates state management from UI logic
- Creates single source of truth
- Enables better testing and reusability
- Parent controls state modification
- Prevents state duplication

```kotlin
// Stateful
@Composable
fun StatefulCounter() {
  var count by rememberSaveable { mutableStateOf(0) }
  StatelessCounter(count, { count++ })
}
// Stateless
@Composable
fun StatelessCounter(
  count: Int,
  onIncrement: () -> Unit) {
  Button(onClick = onIncrement) {
    Text("Count: $count")
  }
}
```

16.7.3.1 Key Benefits

- **Interceptable Events** : Parent can filter/modify events
- **State Sharing** : Multiple components can share state
- **Testing** : State can be injected for testing
- **Reusability** : Components become context-independent
- **Encapsulation** : State modification controlled by parent

16.7.3.2 When to Hoist

- Multiple components need same state
- Component needs to be reused
- State changes affect multiple components
- Testing requires state control
- Event handling needs interception

16.7.4 ViewModel Integration

```kotlin
class MyViewModel : ViewModel() {
    private val _name = MutableLiveData("")
    val name: LiveData<String> get() = _name
}

@Composable
fun Editor(viewModel: MyViewModel = viewModel()) {
    val name by viewModel.name.observeAsState("")
    TextField(
        value = name,
        onValueChange = { viewModel.updateName(it) }
    )
}
```

16.7.5 State Management APIs Comparison

Feature	remember	rememberSaveable	LiveData/Flow
Persistence	Recomposition	Activity recreation	Process lifecycle
Scope	Composition	Bundle	ViewModel
Configuration changes	Lost	Preserved	Preserved
Architecture support	Local	Local	MVVM
Threading	UI Thread	UI Thread	Any thread

16.7.5.1 remember API

- Stored in Composition
- Lost on activity recreation
- Fastest performance
- Local state management

```kotlin
var count by remember {mutableStateOf(0)}
var list by remember {mutableStateListOf<String>()}
var state by remember {mutableStateOf(CustomObject())}
```

16.7.5.2 rememberSaveable API

- Stored in Bundle
- Survives configuration changes
- Automatic Parcelable handling
- Size limitations of Bundle

```kotlin
var count by rememberSaveable { mutableStateOf(0) }
var custom by rememberSaveable(stateSaver =
↪  CustomSaver()) {
    mutableStateOf(CustomObject())
}
```

16.7.5.3 LiveData/Flow Integration

- ViewModel integration
- Architecture component support
- Lifecycle awareness
- Background thread support

```kotlin
// LiveData
val name: String by viewModel.name.observeAsState("")
// Flow
val name by viewModel.name$.collectAsState(initial="")
```

16.7.5.4 LiveData/Flow Integration with Mutable Objects

- Recomposition only triggered on reference change
- Mutating object properties doesn't trigger updates
- Copy objects to force state update

```kotlin
class PersonViewModel : ViewModel() {
    private val _p = MutableLiveData(Person("", ""))
    val person: LiveData<Person> get() = _p
    // Wrong - won't trigger recomposition
    fun updateWrong(name: String) {
        _p.value!!.name = name
        _p.value = _p.value
    }
    // Correct - creates new instance
    fun updateCorrect(name: String) {
        val current = _p.value!!
        _p.value = current.copy(name = name)
    }
}
```

16.7.5.4.1 Common Pitfalls

- Modifying lists/maps in-place
- Direct property mutations
- Nested mutable objects
- Using postValue() with TextField

16.7.5.4.2 Best Practices

- Use immutable data classes
- Create new instances for updates
- Use copy() for modifications
- Prefer value over postValue for UI updates

16.7.5.5 State Selection Guidelines

- Use `remember` for :
  - UI-only state
  - Temporary values
  - Performance-critical updates
- Use `rememberSaveable` for :
  - User input
  - UI state needing persistence
  - Configuration change survival
- Use `LiveData/Flow` for :
  - Business logic state
  - Data layer integration
  - Complex state management
  - Background operations

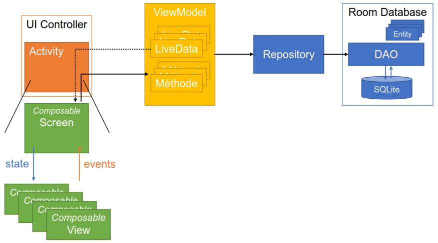
16.7.6 State Management Best Practices

- Single source of truth
- State hoisting to common ancestor
- Use appropriate state type for use case
- Avoid nested state management
- Consider side effects with LaunchedEffect

16.7.7 TextField State Considerations

- Internal state for cursor position
- Known synchronization issues with LiveData
- Use direct value updates in UI thread
- Avoid postValue() for TextField updates

```kotlin
// Preferred approach
_textState.value = newText  // Direct update
// Instead of
_textState.postValue(newText)  // Async update
```



16.8 Adaptive Layouts

16.8.1 Window Size Classes

```
enum class WindowSizeClass {COMPACT, MEDIUM, EXPANDED}
val widthWindowSizeClass = when {
    windowDpSize.width < 600.dp ->
    ↪   WindowSizeClass.COMPACT
    windowDpSize.width < 840.dp ->
    ↪   WindowSizeClass.MEDIUM
    else -> WindowSizeClass.EXPANDED
}
```

16.8.2 Root Composables

- Adapt to form factor
- Use window metrics library
- Handle different screen configurations

```
@Composable
fun RootAdaptive(widthSizeClass: WindowSizeClass) {
    when(widthSizeClass) {
        WindowSizeClass.EXPANDED -> TwoPane()
        else -> OnePane()
    }
}

@Composable
fun TwoPane() {
    Row(modifier = Modifier.fillMaxSize()) {
        Box(modifier = Modifier
            .fillMaxHeight()
            .weight(3f)
            .defaultMinSize(minWidth = 250.dp))
        Box(modifier = Modifier
            .fillMaxHeight()
            .weight(7f))
    }
}
```

16.8.3 Reusable Components

- Use BoxWithConstraints for space-aware layouts
- Base layouts on available space, not global metrics
- Adapt content based on constraints

```
@Composable
fun AdaptiveCard() {
    BoxWithConstraints {
        if (maxWidth < 400.dp) {
            CompactLayout()
        } else {
            ExpandedLayout()
        }
    }
}
```

17 Testing in Android

17.1 Testing Philosophy

- Testing finds bugs but cannot prove their absence (Dijkstra, 1970)
- Automated testing enables faster development and regression detection
- Test coverage must be repeatable and comprehensive

17.2 Test Categories

Type	Scope	Purpose
Unit	Individual classes/functions	Component validation
Integration	Module interactions	System cohesion
End-to-End	Complete workflows	User story validation

17.3 Android-Specific Testing

17.3.1 Platform Constraints

- Development occurs off-target
- Incomplete SDK implementations
- Heavy asynchronous operations
- UI animations and transitions

17.3.2 Jetpack Compose Testing

```
class ComposeTest {
    @get:Rule
    val composeRule = createComposeRule()

    @Test
    fun componentTest() {
        composeRule.setContent {
            MyComponent()
        }

        composeRule
            .onNodeWithTag("test-tag")
            .assertExists()
            .assertTextEquals("Expected Text")
    }
}
```

17.4 Instrumented Testing

17.4.1 Database Testing

```
@RunWith(AndroidJUnit4::class)
@LargeTest
class DBInstrumentedTest {
    private lateinit var db: Database
    private lateinit var dao: Dao

    @get:Rule
    val instantTaskExecutorRule =
    ↪   InstantTaskExecutorRule()

    @Before
    fun setup() {
        val context = ApplicationProvider
          .getApplicationContext<Context>()
        db = Room.inMemoryDatabaseBuilder(
          context, Database::class.java
        ).build()
        dao = db.dao()
    }

    @Test
    fun testDatabaseOperations() {
        // Test async database operations
        val liveData = dao.getAll()
        val value = liveData.waitingValue()  // Custom
        ↪   extension
        assertNotNull(value)
    }
}
```

17.4.2 UI Testing with Espresso

```
@RunWith(AndroidJUnit4::class)
class UITest {
    @get:Rule
    var activityRule =
    ↪   ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun testUIInteraction() {
        onView(withId(R.id.button))
            .perform(click())

        onView(withId(R.id.result))
            .check(matches(withText("Expected")))
    }
}
```

17.4.3 Testing Considerations

- Disable animations for reliability
- Handle asynchronous operations :
  - LiveData testing
  - Coroutine testing
  - Thread synchronization
- Manage ANR dialogs
- Account for emulator startup time

```
companion object {
  @BeforeClass
  fun setupClass() {
    // Handle ANR dialog
    UiDevice
      .getInstance(getInstrumentation())
      .findObject(UiSelector().textContains("wait"))
      ?.click()

    // Wait for startup
    Thread.sleep(5000)
  }
}
```

17.5 CI/CD Integration

- Android SDK setup required
- Docker container support
- Emulator challenges :
  - Hardware acceleration needs
  - Boot time management
  - Animation disabling
- Firebase Test Lab integration

17.6 Play Store Testing

- Automatic Monkey testing on submission
- Multi-device compatibility testing
- Accessibility validation
- Security compliance checks

17.7 Firebase Robo Tests

- Automated UI exploration
- Screen capture and logging
- Performance profiling
- API level verification
- Free tier limitations :
  - Daily test quotas
  - Device pool restrictions

```
// Example CI/CD workflow
tasks.register("ciCheck") {
    dependsOn("test")           // Unit tests
    dependsOn("connectedCheck") // Instrumented tests
    dependsOn("lintDebug")      // Static analysis
}
```