

Briques de base

Structure d'un projet

Manifest : fichier obligatoire décrivant l'application
 • Composants (activités, services, receivers, providers)
 • Permissions requises
 • Fonctionnalités hardware/software
 • Point d'entrée (LAUNCHER activity avec intent-filter MAIN)

Ressources (res/) : séparation code/contenu

- values/ : strings, dimensions, couleurs, thèmes
- drawable/ : images (bitmap, vector, nine-patch)
- layout/ : interfaces graphiques
- menu/ : menus d'action
- anim/ : animations
- Contextualisation : -fr, -night, -sw600dp, -land, etc.

Gradle : système de build

- build.gradle (projet et app)
- Dépendances Maven (groupId:artifactId:version)
- Configuration minSdk, targetSdk, compileSdk

Ressources essentielles

Strings : strings.xml

```
<string name="app_name">My App</string>
<string name="welcome">Hello %1$s!</string>
<plurals name="clicks">
    <item quantity="one">%d click</item>
    <item quantity="other">%d clicks</item>
</plurals>
<!-- Tableaux -->
<string-array name="countries">
    <item>@string/switzerland</item>
</string-array>
```

Dimensions : privilégier dp et sp

```
<dimen name="margin">16dp</dimen>      <!-- Layout -->
<dimen name="text_size">14sp</dimen>      <!-- Texte -->
• dp (density-independent) : 1dp = 1px à 160dpi
• sp (scale-independent) : s'adapte aux préférences utilisateur
```

Drawables

- Bitmap : PNG, WEBP, JPEG (densités : mdpi=1x, hdpi=1.5x, xhdpi=2x, xxhdpi=3x, xxxhdpi=4x)
- Vector : SVG Android, scalable, <vector> XML
- Nine-Patch : zones extensibles (bordures 1px), éviter distorsions
- State List : états (pressed, focused, hovered, default)
- Level List : niveaux numériques (0-n), ex: signal wifi

Layouts

- LinearLayout : direction unique (vertical/horizontal), orientation, layout_weight
- RelativeLayout : positionnement relatif (layout_above, layout_alignParent*)
- ConstraintLayout : contraintes flexibles (recommandé), app:layout_constraint*

Activités

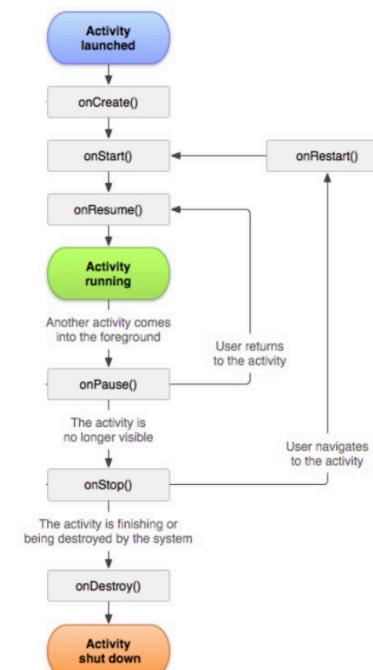
Déclaration : AndroidManifest.xml

```
<activity android:name=".MainActivity"
    android:exported="true">
    <intent-filter>
        <action
            android:name="android.intent.action.MAIN"/>
        <category
            android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Implémentation minimale

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

Cycle de vie



- onCreate(savedInstanceState) : création, initialisation, setContentView(). Appelé une seule fois.
- onStart() : activité devient visible, pas encore interactive
- onResume() : activité au premier plan, interactive. Reprendre animations, caméra, capteurs.
- onPause() : activité perd le focus (dialog, split-screen). Sauvegarder brouillons, pause animations. Exécution rapide (<1s).
- onStop() : activité invisible (autre activité). Libérer ressources lourdes. Peut être tuée par système.
- onDestroy() : destruction définitive ou rotation. Libérer toutes ressources.

États

- Active (Running) : premier plan, utilisateur interagit
- En pause (Paused) : visible mais pas focus, conserve état
- Arrêtée (Stopped) : invisible, conserve état, peut être tuée
- Inactive : détruite, doit être recréée

i Info

En multi-fenêtres, activité visible (onStart) mais pas focus (onPause). Rotation écran → onPause → onStop → onDestroy → onCreate → onStart → onResume

View Binding (recommandé)

```
// build.gradle
android { buildFeatures { viewBinding = true } }
```

```
// Activity
private lateinit var binding: ActivityMainBinding
```

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding =
        ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)
```

```
    binding.myButton.setOnClickListener { }
```

Intents

```
// Explicite (interne) - classe spécifique
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("KEY", "value")
intent.putExtra("USER_ID", 42)
intent.putExtra("IS_PREMIUM", true)
startActivity(intent)
```

```
// Implicit (externe) - action générique
val intent = Intent(Intent.ACTION_VIEW)
intent.data = Uri.parse("https://heig-vd.ch")
if (intent.resolveActivity(packageManager) != null) {
    startActivity(intent)
}
```

```
// Autres actions courantes
Intent(Intent.ACTION_DIAL,
Uri.parse("tel:0123456789")) // Téléphone
Intent(Intent.ACTION_SEND).apply { // Partage
    type = "text/plain"
    putExtra(Intent.EXTRA_TEXT, "Message à partager")
}
Intent(MediaStore.ACTION_IMAGE_CAPTURE) // Caméra

// Réception
val value = intent.getStringExtra("KEY")
val age = intent.getIntExtra("AGE_KEY", -1) // défaut si absent
val user =
    intent.getParcelableExtra<User>("USER") // Objet Parcelable
val users =
    intent.getParcelableArrayListExtra<User>("USERS") // Liste
```

Activity Result (moderne)

```
// Contrat personnalisé
class PickNameContract : ActivityResultContract<Void?, String?>() {
    override fun createIntent(context: Context, input: Void?) =
        Intent(context, PickActivity::class.java)

    override fun parseResult(resultCode: Int, result: Intent?) =
        if (resultCode == RESULT_OK)
            result?.getStringExtra("NAME") else null
    }
```

```
// Enregistrement (avant onCreate)
private val getName =
    registerForActivityResult(PickNameContract()) { name ->
    // Traiter résultat (peut être null)
}

// Lancement
getName.launch(null)

// Retour depuis PickActivity
setResult(RESULT_OK, Intent().putExtra("NAME", name))
finish()
```

Sauvegarde d'état (rotation, manque mémoire)

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

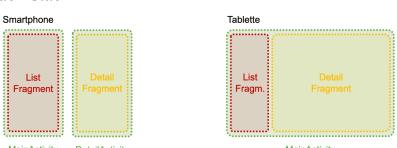
```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    counter =
        savedInstanceState.getInt("COUNTER", 0)
}
```

⚠ Warning

Le système sauvegarde automatiquement les widgets avec android:id. Pour données complexes, sauvegarder manuellement.

Fragments

Avantages : modulaires, réutilisables, cycle de vie propre, multi-écrans (tablette/smartphone), navigation back stack



Cycle de vie spécifique

- onAttach() : attaché à l'activité, accès context
- onCreate() : initialisation (pas de vue)
- onCreateView() : création vue, inflate layout, retourner View
- onViewCreated() : vue créée, initialiser UI (View Binding ici)
- onDestroyView() : vue détruite (rotation), libérer binding
- onDetach() : détaché de l'activité

Implémentation complète

```

class MyFragment : Fragment() {
    private var _binding: FragmentMyBinding? = null
    private val binding get() = _binding!!

    override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View {
        _binding =
            FragmentMyBinding.inflate(inflater, container, false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.button.setOnClickListener { /* ... */ }
    }

    override fun onDestroyView() {
        super.onDestroyView()
        _binding = null // IMPORTANT : éviter fuites mémoire
    }
}

Gestion : FragmentManager, transactions
supportFragmentManager.beginTransaction()
    .replace(R.id.fragment_container, MyFragment())
    .addToBackStack(null) // Ajouter à la pile retour
    .commit()

// Avec arguments
val fragment = MyFragment().apply {
    arguments = Bundle().apply {
        putString("KEY", "value")
       .putInt("ID", 42)
    }
}

// Dans Fragment : récupération
val value = requireArguments().getString("KEY")

```

Communication

- Activité → Fragment : arguments (Bundle), éviter méthodes directes
- Fragment → Activité : interfaces callback ou ViewModels (recommandé)
- Fragment ↔ Fragment : ViewModels partagés (activityViewModels())

ViewModel partagé

```
// Dans plusieurs fragments
private val sharedViewModel: SharedViewModel by activityViewModels()

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    sharedViewModel.data.observe(viewLifecycleOwner) { data ->
        // Mise à jour UI
    }
}
```

⚠ Warning

Toujours utiliser viewLifecycleOwner pour observer LiveData dans les fragments, pas this. Fragments peuvent être recréés mais leurs vues détruites (rotation).

Services

Types

- Foreground** : notification obligatoire, tâches visibles (lecteur audio, navigation GPS). Priorité haute.
- Background** : limités API 26+, tâches courtes. Restrictions importantes.
- Bounded** : liés à un composant, détruits quand plus de liens. Communication bidirectionnelle.

Cycle de vie

- onCreate() : création, initialisation une fois
- onStartCommand(intent, flags, startId) : chaque appel startService()
 - START_STICKY : redémarrer si tué (lecteur musique)
 - START_NOT_STICKY : ne redémarrer pas
 - START_REDELIVER_INTENT : redémarrer avec dernier intent
- onBind(intent) : retourne IBinder pour communication
- onUnbind(intent) : tous clients déconnectés
- onDestroy() : nettoyage, libérer ressources

Foreground Service

```

class MusicService : Service() {
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        val notification = createNotification()
        startForeground(NOTIFICATION_ID, notification)

        thread {
            // Tâche longue
            stopSelf() // Arrêter quand terminé
        }

        return START_STICKY
    }

    override fun onBind(intent: Intent): IBinder? = null

    // Manifest
    <service android:name=".MusicService"
        android:foregroundServiceType="mediaPlayback" />
    <uses-permission
        android:name="android.permission.FOREGROUND_SERVICE" />
    <uses-permission
        android:name="android.permission.FOREGROUND_SERVICE_MEDIA_LIBRARY" />

    // Démarrer
    if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        startForegroundService(Intent(this, MusicService::class.java))
    } else {
        startService(Intent(this, MusicService::class.java))
    }
}

WorkManager (recommandé pour tâches différées)
val workRequest =
    OneTimeWorkRequestBuilder<MyWorker>()
        .setConstraints(Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresCharging(true)
            .build())
        .setInputData(workDataOf("key" to "value"))
        .build()

WorkManager.getInstance(context).enqueue(workRequest)

```

```

// Worker
class MyWorker(context: Context, params: WorkerParameters) : Worker(context, params) {
    override fun doWork(): Result {
        val data = inputData.getString("key")
        // Tâche (thread séparé automatique)
        return Result.success(workDataOf("result" to "value"))
    }
}

```

⚠ Warning

Services = UI-Thread. Créer thread séparé pour tâches longues. API 26+ : restrictions background importantes. WorkManager préféré pour tâches différées.

Permissions

Niveaux

- Installation : automatiques (Internet, Bluetooth)
- Exécution : demande popup (localisation, caméra, contacts)
- Spéciales : système/constructeur uniquement

Principes

Contrôle, Transparence, Minimisation

Implémentation

- Déclarer dans Manifest (<uses-permission android:name="..." />)
- Vérifier : ContextCompat.checkSelfPermission()
- Demander :


```
registerForActivityResult(RequestPermission()) { granted -> }
```
- Annoter : @SuppressLint("MissingPermission") après vérification

Interfaces graphiques

Composants de base

Visibilité : VISIBLE, INVISIBLE (espace réservé), GONE (pas d'espace)

Vues

- TextView : text, textSize, textColor, textStyle
- EditText : getText().toString(), inputType (clavier adapté), TextWatcher
- Button : setOnClickListener, setOnLongClickListener
- ImageView : scaleType (fitCenter, centerCrop, fitXY, centerInside)

Sélection

- CheckBox/Switch : isChecked, setOnCheckedChangeListener
- RadioGroup : sélection unique, setOnCheckedChangeListener
- Spinner : ArrayAdapter + onItemSelectedListener

Progression

- ProgressBar : indéterminée (animation) ou déterminée (0-max)
- SeekBar : setOnSeekBarChangeListener (onProgressChanged, onStartTracking, onStopTracking)

WebView : permission INTERNET, settings.javaScriptEnabled = true, webViewClient

UI-Thread

Règle d'or : opérations longues → thread séparé, UI → UI-Thread uniquement

```
thread {
    val result = downloadData() // Thread séparé
    runOnUiThread { textView.text = result } // UI-Thread
}
```

⚠ Warning

Modifier l'UI hors UI-Thread → CalledFromWrongThreadException ou ANR (Application Not Responding).

Alternatives modernes : Coroutines, RxJava, WorkManager

Vues personnalisées

Extension

```

class MyView @JvmOverloads constructor(
    context: Context, attrs: AttributeSet? = null, defStyleAttr: Int = 0) : View(context, attrs, defStyleAttr) {
    init {
        attrs?.let {
            val ta =
                context.obtainStyledAttributes(it, R.styleable.MyView)
            // val color =
            ta.getColor(R.styleable.MyView_customColor, Color.BLACK)
            ta.recycle() // IMPORTANT : libérer ressources
        }
    }
}
```

From scratch : onDraw(canvas), onMeasure(widthMeasureSpec, heightMeasureSpec), onTouchEvent(), invalidate() (redessiner)

Material Design

TextInputLayout : label flottant, erreurs, compteur, icônes

```
inputLayout.error = "message" // Afficher erreur
inputLayout.error = null // Effacer erreur
inputLayout.isCounterEnabled = true
inputLayout.counterMaxLength = 50
```

Autres : MaterialButton, Chip, BottomNavigationView, TabLayout, CardView, FloatingActionButton

Feedback utilisateur

Toast

```
Toast.makeText(this, "msg",
    Toast.LENGTH_SHORT).show() // 2s
// Toast.LENGTH_LONG // 3.5s
```

Snackbar (Material, avec action)

```
Snackbar.make(view, "msg",
    Snackbar.LENGTH_SHORT)
    .setAction("Annuler") { /* undo */ }
    .show()
```

Dialog

```
AlertDialog.Builder(this)
    .setTitle("Titre")
    .setMessage("Message")
    .setPositiveButton("OK") { _, _ -> }
```

```
.setNegativeButton("Annuler", null)
.setNeutralButton("Plus tard", null)
.show()
```

Variante : .setItems(items) { _, which -> }, .setSingleChoiceItems(items, checked) { _, which -> }, .setMultiChoiceItems(items, checked) { _, which, isChecked -> }, .setView(customView)

DatePickerDialog / TimePickerDialog : sélection date/heure native

DialogFragment : dialogues complexes, survit rotation, réutilisable

Notifications

Canal requis (API 26+)

```
val channel = NotificationChannel(
    CHANNEL_ID,
    "Nom visible",
    NotificationManager.IMPORTANCE_DEFAULT
)
channel.description = "Description du canal"
notificationManager.createNotificationChannel(channel)
```

Création

```
val notif = NotificationCompat.Builder(this,
    CHANNEL_ID)
    .setSmallIcon(R.drawable.icon) // OBLIGATOIRE
    .setContentTitle("Titre")
    .setContentText("Message")
    .setContentIntent(pendingIntent) // Action au tap
    .setAutoCancel(true) // Disparaît au tap
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)
    .addAction(icon, "Action", pendingIntent) // Max 3
    .build()
```

NotificationManagerCompat.from(this).notify(notification, notif)

PendingIntent :

PendingIntent.getActivity(context, requestCode, intent, PendingIntent.FLAG_IMMUTABLE or PendingIntent.FLAG_UPDATE_CURRENT)

⚠ Warning

API 31+ : FLAG_IMMUTABLE obligatoire pour PendingIntent.

Styles étendus

BigTextStyle (texte long)

```
val style = NotificationCompat.BigTextStyle()
    .bigText("""Texte très long qui sera affiché en entier
lorsque l'utilisateur déploie la notification""")
    .setBigContentTitle("Titre détaillé")
    .setSummaryText("Résumé")
```

notificationBuilder.setStyle(style)

BigPictureStyle (image)

```
val bitmap =
BitmapFactory.decodeResource(resources,
R.drawable.photo)
val style = NotificationCompat.BigPictureStyle()
    .bigPicture(bitmap)
    .bigLargeIcon(null) // Cache large icon quand déployé
    .setBigContentTitle("Nouvelle photo")
```

notificationBuilder.setLargeIcon(bitmap).setStyle(style)

InboxStyle (liste)

```
val style = NotificationCompat.InboxStyle()
    .addLine("Email 1: Sujet important")
    .addLine("Email 2: Confirmation")
    .addLine("Email 3: Newsletter")
    .setBigContentTitle("3 nouveaux emails")
    .setSummaryText("inbox@example.com")
```

notificationBuilder.setStyle(style)

MessagingStyle (conversation)

```
val style =
NotificationCompat.MessagingStyle("Moi")
    .addMessage("Salut!", System.currentTimeMillis(), "Jean")
    .addMessage("ça va?", System.currentTimeMillis(), "Jean")
    .addMessage("Très bien!", System.currentTimeMillis(), null) // Moi
    .setConversationTitle("Conversation avec Jean")
```

notificationBuilder.setStyle(style)

Actions avec input

```
val remoteInput =
RemoteInput.Builder("KEY_TEXT_REPLY")
    .setLabel("Répondre")
    .build()

val replyAction =
NotificationCompat.Action.Builder(
    R.drawable.ic_reply,
    "Répondre",
    replyPendingIntent)
.addRemoteInput(remoteInput).build()

notificationBuilder.addAction(replyAction)

// Récupération dans Activity
val input =
RemoteInput.getResultsFromIntent(intent)
val reply =
input?.getCharSequence("KEY_TEXT_REPLY")?.toString()

Groupes : .setGroup(GROUP_KEY), .setGroupSummary(true) pour notification récapitulative
```

Canaux multiples

```
val channels = listOf(
    NotificationChannel("messages", "Messages",
    IMPORTANCE_HIGH),
    NotificationChannel("updates", "Mises à jour", IMPORTANCE_LOW),
    NotificationChannel("alerts", "Alertes", IMPORTANCE_DEFAULT))
notificationManager.createNotificationChannels(channels)
```

ActionBar et Menu

Configuration

```
setSupportActionBar(toolbar)
supportActionBar?.apply {
    title = "Titre"
    setDisplayHomeAsUpEnabled(true) // Bouton retour
}
```

Menu XML (res/menu/main_menu.xml)

```
<menu xmlns:android="..." xmlns:app="...">
<item android:id="@+id/action_search"
    android:title="Rechercher"
    android:icon="@drawable/ic_search"
    app:showAsAction="ifRoom|withText" />
<!-- showAsAction: never, ifRoom, always, withText, collapseActionView -->
</menu>
```

Gestion

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

override fun onOptionsItemSelected(item: MenuItem) = when(item.itemId) {
    R.id.action_search -> { /* ... */; true }
    android.R.id.home -> {
        onBackPressedDispatcher.onBackPressed()
        true
    }
    else -> super.onOptionsItemSelected(item)
}
```

SearchView :

```
app:actionViewClass="androidx.appcompat.widget.SearchView"
setOnQueryTextListener
```

Menu contextuel : clic long

```
registerForContextMenu(view)

override fun onCreateContextMenu(menu: ContextMenu, v: View, menuInfo: ContextMenuInfo?) {
    menuInflater.inflate(R.menu.context_menu, menu)
}

override fun onContextItemSelected(item: MenuItem): Boolean {
    return when(item.itemId) {
        R.id.action_edit -> { /* ... */; true }
        else ->
            super.onContextItemSelected(item)
    }
}
```

RecyclerView

Avantages : recyclage obligatoire, ViewHolder standardisé, animations, layouts flexibles, DiffUtil

Configuration

```
recyclerView.apply {
    adapter = MyAdapter()
    layoutManager =
LinearLayoutManager(context) // ou
```

GridLayoutManager(context, 2)

```
addItemDecoration(DividerItemDecoration(context,
DividerItemDecoration.VERTICAL))
```

Adapter simple

```
class MyAdapter :
RecyclerView.Adapter<MyAdapter.ViewHolder>() {
    private var items = listOf<String>()

    fun updateItems(newItems: List<String>) {
        items = newItems
        notifyDataSetChanged() // Force tout redessiner
    }

    override fun getItemCount() = items.size
```

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val view =
LayoutInflator.from(parent.context)
    .inflate(R.layout.item, parent, false)
    return ViewHolder(view)
}
```

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    holder.bind(items[position])
}

inner class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    private val title: TextView =
itemView.findViewById(R.id.title)

    fun bind(item: String) {
        title.text = item
        itemView.setOnClickListener {
            // adapterPosition donne la position actuelle
            Toast.makeText(itemView.context,
"Cliqué: $item", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Types multiples

```
class MultiTypeAdapter :
RecyclerView.Adapter<RecyclerView.ViewHolder>() {
    companion object {
        const val TYPE_HEADER = 0
        const val TYPE_ITEM = 1
    }

    override fun getItemViewType(position: Int) = when(items[position]) {
        is Header -> TYPE_HEADER
        is Item -> TYPE_ITEM
        else -> throw IllegalArgumentException()
    }
```

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
when(viewType) {
    TYPE_HEADER -> HeaderViewHolder(...)
    TYPE_ITEM -> ItemViewHolder(...)
    else -> throw IllegalArgumentException()
}
```

```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
    when(holder) {
        is HeaderViewHolder ->
            holder.bind(items[position] as Header)
        is ItemViewHolder ->
            holder.bind(items[position] as Item)
    }
}
```

Optimisations performances

```
recyclerView.apply {
    setHasFixedSize(true) // Si taille fixe
    setItemViewCacheSize(20) // Cache vues hors écran
    recycledViewPool.setMaxRecycledViews(TYPE_ITEM, 10)
}
```

```
// Dans ViewHolder : éviter findViewById répétés
class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    private val title: TextView =
itemView.findViewById(R.id.title)
    private val image: ImageView =
itemView.findViewById(R.id.image)

    fun bind(item: Item) {
        title.text = item.title
    }
}
```

```

    // Glide.with(itemView).load(item.imageUrl).into(image)
}

DiffUtil : calcul automatique des différences, animations optimales
class ItemDiffCallback(
    private val oldList: List<Item>,
    private val newList: List<Item>
) : DiffUtil.Callback() {
    override fun getOldListSize() = oldList.size
    override fun getNewListSize() = newList.size

    override fun areItemsTheSame(oldPos: Int,
        newPos: Int) =
        oldList[oldPos].id == newList[newPos].id // Même entité?
    override fun areContentsTheSame(oldPos: Int,
        newPos: Int) =
        oldList[oldPos] == newList[newPos] // Même contenu?
}

// Utilisation
fun updateItems(newItems: List<Item>) {
    val diffResult =
        DiffUtil.calculateDiff(ItemDiffCallback(items,
            newItems))
    items = newItems
    diffResult.dispatchUpdatesTo(this) // Applique changements avec animations
}

ListAdapter : DiffUtil intégré, plus simple
class MyAdapter : ListAdapter<Item>,
    MyAdapter.ViewHolder>{
    object : DiffUtil.ItemCallback<Item>() {
        override fun areItemsTheSame(old: Item,
            new: Item) = old.id == new.id
        override fun areContentsTheSame(old:
            Item, new: Item) = old == new
    }
    // ...
    override fun onBindViewHolder(holder:
        ViewHolder, position: Int) {
        holder.bind(getItem(position)) // getItem() au lieu de items[position]
    }
}

// Mise à jour ultra simple
adapter.submitList(newItems) // DiffUtil automatique

```

Animations

```

XML (res/anim/fade_in.xml)
<set android:fillAfter="true">
    <alpha android:fromAlpha="0.0"
        android:toAlpha="1.0" android:duration="300" />
    <translate android:fromYDelta="100%" 
        android:toYDelta="0%" android:duration="300" />
    <scale android:fromXScale="0.5"
        android:toXScale="1.0" android:fromYScale="0.5"
        android:toYScale="1.0" android:pivotX="50%" 
        android:pivotY="50%" />
    <rotate android:fromDegrees="0"
        android:toDegrees="360" android:pivotX="50%" 
        android:pivotY="50%" />
</set>

```

Application

```

val animation =
    AnimationUtils.loadAnimation(this,
        R.anim.fade_in)
view.startAnimation(animation)

animation.setAnimationListener(object :
    Animation.AnimationListener {
    override fun onAnimationEnd(animation:
        Animation?) { /* Fin */ }
    override fun onAnimationStart(animation:
        Animation?) {}
    override fun onAnimationRepeat(animation:
        Animation?) {}
})

```

Programmatique

```

view.animate()
    .alpha(1f)
    .translationY(100f)
    .rotation(360f)
    .scaleX(1.5f)
    .setDuration(300)
    .setInterpolator(AccelerateDecelerateInterpolator())
    .start()

    // Mise à jour
    data.value = 42 // Synchronre, UI-thread
    uniquement
    data.postValue(42) // Asynchrone, tout thread
    (background ok)

    // Observation (Activité)
    data.observe(this) { value ->
        textView.text = "$value"
    }

```

Gestures

```

GestureDetector
private val gestureDetector =
    GestureDetectorCompat(this, object :
        GestureDetector.SimpleOnGestureListener() {

        override fun onDown(e: MotionEvent) =
            true // OBLIGATOIRE

        override fun onSingleTapUp(e: MotionEvent):
            Boolean {
            // Tap simple
            return true
        }

        override fun onLongPress(e: MotionEvent) {
            // Appui long
        }

        override fun onFling(
            e1: MotionEvent?, e2: MotionEvent,
            velocityX: Float, velocityY: Float
        ): Boolean {
            if (e1 == null) return false
            val diffX = e2.x - e1.x
            val diffY = e2.y - e1.y

            if (abs(diffX) > abs(diffY)) {
                // Swipe horizontal
                if (abs(diffX) > SWIPE_THRESHOLD &&
                    abs(velocityX) > SWIPE_VELOCITY_THRESHOLD) {
                    if (diffX > 0) onSwipeRight()
                } else onSwipeLeft()
                return true
            } else {
                // Swipe vertical
                if (abs(diffY) > SWIPE_THRESHOLD &&
                    abs(velocityY) > SWIPE_VELOCITY_THRESHOLD) {
                    if (diffY > 0) onSwipeDown()
                } else onSwipeUp()
                return true
            }
        }
    })

    override fun onTouchEvent(event: MotionEvent) =
        gestureDetector.onTouchEvent(event) ||
        super.onTouchEvent(event)

    companion object {
        const val SWIPE_THRESHOLD = 100
        const val SWIPE_VELOCITY_THRESHOLD = 100
    }
}

```

Pinch-to-zoom

```

private val scaleDetector =
    ScaleGestureDetector(context, object :
        ScaleGestureDetector.SimpleOnScaleGestureListener() {

        override fun onScale(detector:
            ScaleGestureDetector): Boolean {
            scaleFactor *= detector.scaleFactor
            scaleFactor = scaleFactor.coerceIn(0.1f,
                5.0f) // Min/Max
            invalidate()
            return true
        }
    })

    override fun onTouchEvent(event: MotionEvent):
        Boolean {
        scaleDetector.onTouchEvent(event)
        return true
    }
}

```

Live Data & MVVM

LiveData

Observable respectant le cycle de vie

- Mise à jour automatique de l'UI
- Évite fuites mémoire
- Observateur appelé dans UI-thread

Utilisation

```

    // Création
    val data = MutableLiveData(0)

    // Mise à jour
    data.value = 42 // Synchronre, UI-thread
    uniquement
    data.postValue(42) // Asynchrone, tout thread
    (background ok)

    // Observation (Activité)
    data.observe(this) { value ->
        textView.text = "$value"
    }

```

```

// Observation (Fragment)
data.observe(viewLifecycleOwner) { value -> //
    IMPORTANT: viewLifecycleOwner
        textView.text = "$value"
}

```

⚠ Warning

LiveData ne sont pas modifiables. Utiliser MutableLiveData. value peut retourner null (implémentation Java).

Transformations

```

map : transformation synchrone
val userNames: LiveData<List<String>> =
    users.map { usersList ->
        usersList.map { "${it.firstname}" +
            "${it.name}" }
    }

val formattedDate: LiveData<String> =
    timestamp.map { millis ->
        SimpleDateFormat("dd/MM/yyyy",
            Locale.getDefault()).format(Date(millis))
    }

val isValid: LiveData<Boolean> =
    email.map {
        Patterns.EMAIL_ADDRESS.matcher(it).matches()
    }

```

switchMap : LiveData dépendante (changement source)

```

val userId = MutableLiveData<Long>()

// Chaque changement de userId charge nouveau
user
val userDetails: LiveData<User> =
    userId.switchMap { id ->
        repository.getUserById(id) // Retourne
        LiveData<User>
    }

// Exemple recherche dynamique
val searchQuery = MutableLiveData<String>()
val searchResults: LiveData<List<Item>> =
    searchQuery.switchMap { query ->
        if (query.isBlank())
            MutableLiveData(emptyList())
        else repository.search(query)
    }

```

distinctUntilChanged : éviter doublons

```

val filteredData: LiveData<String> =
    rawData.distinctUntilChanged()

```

Combinaisons personnalisées

```

val result: LiveData<String> =
    MediatorLiveData<String>().apply {
        var firstName: String? = null
        var lastName: String? = null

        fun update() {
            value = "$firstName $lastName"
        }

        addSource(firstNameLiveData) {
            firstName = it
            update()
        }
        addSource(lastNameLiveData) {
            lastName = it
            update()
        }
    }

```

MediatorLiveData

: fusion de sources

```

val ld = MediatorLiveData<Int>().apply {
    addSource(ld1) { v -> value = v }
    addSource(ld2) { v -> value = v.toInt() }
    // removeSource() pour arrêter l'observation
}

```

MVVM Architecture

Séparation

- View : UI (Activities, Fragments), observe ViewModel
- ViewModel : logique présentation, état UI, survit rotations
- Model : logique métier, données (Repository, Room)

Avantages : testabilité, maintenabilité, survie aux changements de configuration

ViewModel

Basique

```

class MyViewModel : ViewModel() {
    private val _counter = MutableLiveData(0)
    val counter: LiveData<Int> get() =
        _counter // Read-only

    fun increment() {

```

```

        _counter.postValue(_counter.value!! + 1)

    override fun onCleared() {
        super.onCleared()
        // Nettoyage (annuler coroutines, fermer
        connexions)
    }

    // Activité
    private val viewModel: MyViewModel by
    viewModels()

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

        viewModel.counter.observe(this) { value ->
            textView.text = "$value"
        }

        button.setOnClickListener
        { viewModel.increment() }
    }

    Avec paramètres (Factory)
    class MyViewModel(defaultValue: Int) :
    ViewModel() {
        private val _counter =
        MutableLiveData(defaultValue)
        val counter: LiveData<Int> get() = _counter
    }

    class MyViewModelFactory(private val
    defaultValue: Int) :
    ViewModelProvider.Factory {
        override fun <T : ViewModel>
        create(modelClass: Class<T>): T {
            if
            (modelClass.isAssignableFrom(MyViewModel::class.java))
                return MyViewModel(defaultValue) as
            T
            throw IllegalArgumentException("Unknown
            ViewModel class")
        }
    }

    // Utilisation
    private val viewModel: MyViewModel by viewModels
    {
        MyViewModelFactory(10)
    }
}

```

⚠ Warning

Fragments : toujours viewLifecycleOwner pour observer. Si plusieurs fragments utilisent Factory avec paramètres différents, une seule instance sera créée avec la première Factory.

```

AndroidViewModel : accès contexte Application
class MyViewModel(application: Application) :
AndroidViewModel(application) {
    private val prefs =
    application.getSharedPreferences("prefs",
    MODE_PRIVATE)

    private val _data =
    MutableLiveData<String>()
    val data: LiveData<String> get() = _data

    init {
        _data.value = prefs.getString("key",
        "default")
    }

    // Si seul paramètre = Application, pas de
    Factory nécessaire
    private val viewModel: MyViewModel by
    viewModels()
}

```

Bonnes pratiques

⚠ Warning

NE JAMAIS :

- Référence View, Activity, Fragment ou Context d'activité dans ViewModel
- Exposer MutableLiveData publiques
- Faire des opérations I/O directement dans ViewModel

Recommandations

- Exposer LiveData (lecture seule), garder MutableLiveData privées
- ViewModels ≠ persistance long terme → utiliser Room
- ViewModel détruit uniquement quand activité terminée définitivement (onCleared())
- Utiliser Coroutines pour opérations asynchrones
- Un ViewModel par écran, séparer responsabilités

Dépendances

```

implementation("androidx.lifecycle:lifecycle-
livedata-ktx:2.6.2")
implementation("androidx.lifecycle:lifecycle-
viewmodel-ktx:2.6.2")
implementation("androidx.activity:activity-
ktx:1.8.0")
implementation("androidx.fragment:fragment-
ktx:1.6.1")

```

Persistence des données

Solutions disponibles

- Fichiers : interne/externe, privé/partagé
- Préférences : clé-valeur (SharedPreferences, DataStore)
- Bases de données : SQLite via Room (recommandé)

Stockage fichiers

Interne privé

```

val filesDir = filesDir // Données persistantes
val cacheDir = cacheDir // Cache (système peut
supprimer)

// Écriture
File(filesDir,
"myfile.txt").writeText("contenu")

// Lecture
val content = File(filesDir,
"myfile.txt").readText()
• Chiffré automatiquement (API 29+)
• Supprimé à désinstallation
• Jamais accessible par autres apps

```

Externe privé

```

fun isExternalStorageWritable() =
    Environment.getExternalStorageState() ==
Environment.MEDIA_MOUNTED

val externalRoot = getExternalFilesDir(null) //
ou
getExternalFilesDir(Environment.DIRECTORY_PICTURES)
val externalCache = externalCacheDir
• Carte SD (ou émulé)
• Vérifier disponibilité
• Jamais chemins en dur
• Supprimé à désinstallation

```

Média partagés : MediaStore, non supprimés à désinstallation, accessibles par autres apps

SharedPreferences

Utilisation

```

// Obtenir
val prefs = getSharedPreferences("fichier",
Context.MODE_PRIVATE)

// Écriture (commit synchrone, apply asynchrone)
prefs.edit {
    putString("key", "value")
   .putInt("count", 42)
    putBoolean("flag", true)
    putFloat("score", 9.5f)
    putLong("timestamp",
System.currentTimeMillis())
    // putStringSet() pour Set<String>
}

// Lecture (avec valeur par défaut si clé
absente)
val value = prefs.getString("key", "default")
val count = prefs.getInt("count", 0)

// Supprimer

```

```

prefs.edit { remove("key") }

```

```

// Tout supprimer
prefs.edit { clear() }

```

⚠ Warning

Google recommande DataStore pour nouveaux projets (Flow, thread-safe, asynchrone).

Room Database

Architecture : Entity ↔ DAO ↔ Database ↔ Repository ↔ ViewModel ↔ UI

Avantages

- Vérification requêtes compilation
- Génération code (KSP)
- Gestion migrations
- Support LiveData/Flow
- Thread-safe

Entity

```

@Entity(tableName = "person") // Nom table
optionnel
data class Person(
    @PrimaryKey(autoGenerate = true) var id:
    Long?,
    @ColumnInfo(name = "person_name") var name:
    String, // Nom colonne optionnel
    var birthday: Calendar,
    @Ignore var tempValue: String = "" // Colonne ignorée
)

```

DAO

```

@Dao
interface PersonDao {
    @Query("SELECT * FROM Person ORDER BY name
ASC")
    fun getAll(): LiveData<List<Person>>

    @Query("SELECT * FROM Person WHERE name
LIKE :search")
    fun search(search: String):
    LiveData<List<Person>>

```

```

    @Query("SELECT * FROM Person WHERE birthday
> :date LIMIT :limit")
    fun getYoungerThan(date: Long, limit: Int):
    LiveData<List<Person>>

```

```

    @Query("SELECT COUNT(*) FROM Person")
    fun getCount(): LiveData<Int>

    @Query("SELECT * FROM Person WHERE id =
:id")
    fun getById(id: Long): LiveData<Person?>

```

```

    @Query("SELECT * FROM Person WHERE id IN
(:ids)")
    fun getIds(ids: List<Long>):
    LiveData<List<Person>>

```

```

    @Insert(onConflict =
OnConflictStrategy.REPLACE)
    suspend fun insert(person: Person): Long // Coroutine

```

```

    @Insert
    fun insertAll(vararg persons: Person):
    List<Long>

```

```

    @Update
    suspend fun update(person: Person)

```

```

    @Delete
    suspend fun delete(person: Person)

```

```

    @Query("DELETE FROM Person WHERE id = :id")
    suspend fun deleteById(id: Long)

```

```

    @Query("DELETE FROM Person")
    suspend fun deleteAll()

```

```

    @Transaction // Garantit atomicité
    suspend fun updatePersons(personsToDelete:
    List<Person>, personsToInsert:
    List<Person>) {
        personsToDelete.forEach { delete(it) }
        personsToInsert.forEach { insert(it) }
    }
}

```

Requêtes JOIN

```

@Query("""SELECT Person.*,
COUNT(Phone.phoneId)
as phoneCount
FROM Person
LEFT JOIN Phone ON Person.id =
Phone.ownerId
GROUP BY Person.id""")

```

```

fun getPersonsWithPhoneCount(): LiveData<List<PersonWithCount>>
data class PersonWithCount(
    @Embedded val person: Person,
    val phoneCount: Int
)

Database
@Database(entities = [Person::class], version = 1, exportSchema = true // Exporter schéma pour migrations)
@TypeConverters(CalendarConverter::class)
abstract class MyDatabase : RoomDatabase() {
    abstract fun personDao(): PersonDao

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

        fun getDatabase(context: Context): MyDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance =
                    Room.databaseBuilder(
                        context.applicationContext,
                        MyDatabase::class.java,
                        "database.db"
                    )
                    .fallbackToDestructiveMigration() // ATTENTION: supprime données si migration absente
                    // .addMigrations(MIGRATION_1_2, MIGRATION_2_3)
                    .build()
                INSTANCE = instance
                instance
            }
        }
    }
}

TypeConverter
class CalendarConverter {
    @TypeConverter
    fun toCalendar(dateLong: Long) = Calendar.getInstance().apply { time = Date(dateLong) }

    @TypeConverter
    fun fromCalendar(date: Calendar) = date.time.time
}

// Autres exemples
class Converters {
    @TypeConverter
    fun fromList(list: List<String>) = list.joinToString(",")

    @TypeConverter
    fun toList(string: String) = string.split(",")
}

```

Repository

```

class Repository(private val dao: PersonDao) {
    val allPersons = dao.getAll()

    fun insert(person: Person) {
        thread { dao.insert(person) } // Ou viewModelScope.launch
    }

    fun delete(person: Person) {
        thread { dao.delete(person) }
    }

    fun search(query: String) = dao.search("%$query%")
}

```

⚠ Warning

Opérations I/O TOUJOURS asynchrones (thread, coroutines, ou WorkManager).

ViewModel intégration

```

class MyViewModel(private val repository: Repository) : ViewModel() {
    val persons = repository.allPersons

    fun addPerson(name: String) {
        repository.insert(Person(null, name, Calendar.getInstance()))
    }

    class MyViewModelFactory(private val repository:

```

```

Repository) :
    ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(MyViewModel::class.java))
            return MyViewModel(repository) as T
        throw IllegalArgumentException("Unknown ViewModel")
    }
}

```

```

// Application
class MyApp : Application() {
    val repository by lazy {
        val database =
            MyDatabase.getDatabase(this)
        Repository(database.personDao())
    }
}

// Activité
private val viewModel: MyViewModel by viewModels {
    MyViewModelFactory((application as
        MyApp).repository)
}

```

Relations

```

One-to-Many
@Entity
data class Phone(
    @PrimaryKey val phoneId: Long,
    val number: String,
    val ownerId: Long // Foreign key
)

data class PersonWithPhones(
    @Embedded val person: Person,
    @Relation(
        parentColumn = "id",
        entityColumn = "ownerId"
    )
    val phones: List<Phone>
)

@Dao
interface PersonDao {
    @Transaction
    @Query("SELECT * FROM Person")
    fun getPersonsWithPhones(): LiveData<List<PersonWithPhones>>
}

Many-to-Many
@Entity(primaryKeys = ["playlistId", "songId"])
data class PlaylistSongCrossRef(
    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>
)

@Transaction
@Query("SELECT * FROM Playlist")
fun getPlaylistsWithSongs(): LiveData<List<PlaylistWithSongs>>

```

Migrations

```

val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(db: SupportsSQLiteDatabase) {
        db.execSQL("ALTER TABLE Person ADD
COLUMN email TEXT")
    }
}

val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(db: SupportsSQLiteDatabase) {
        db.execSQL("CREATE TABLE IF NOT EXISTS
Phone (...)")
    }
}

Room.databaseBuilder(...)
    .addMigrations(MIGRATION_1_2, MIGRATION_2_3)
    .build()

```

Schéma export

```

// build.gradle (app)
android {
    defaultConfig {
        ksp {
            arg("room.schemaLocation",

```

```

"$projectDir/schemas"
        }
    }
}

```

Bonnes pratiques

- Room > SQLite direct
- Jamais I/O sur UI-thread
- Exposer LiveData (lecture), garder MutableLiveData privées
- Singleton Database dans Application
- distinctUntilChanged() sur LiveData si nécessaire
- DataStore pour préférences
- exportSchema = true + versionner schémas
- Utiliser Coroutines avec suspend fun dans DAO

⚠ Warning

Jamais référencer View, Activity, Context dans ViewModel.

Dépendances

```

// Room
implementation("androidx.room:room-
runtime:2.6.1")
implementation("androidx.room:room-ktx:2.6.1")
ksp("androidx.room:room-compiler:2.6.1")

// KSP
plugins {
    id("com.google.devtools.ksp") version
    "1.9.20-1.0.14"
}

```