Programació multimèdia
i dispositius mòbils

UD 02. Programació d'aplicacions per a dispositius mòbils. Android.

Disseny d'interfícies

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2021-22

Continguts

Disseny de Layouts

Com hem vist, la interfície d'usuari d'una activitat de l'aplicació s'especifica com un recurs de tipus *layout*, en format XML. Aquest document XML podríem escriure'l manualment, però Android Studio disposa d'un potent editor visual basat en arrossegar i soltar elements.

Quan creem una interfície d'usuari, aquesta es compon d'una jerarquia de *layouts* i *widgets*. Aquests *layouts* seran objectes de tipus *ViewGroup*, controladors que controlaran el posicionament i les vistes secundàries de la pantalla. Els *widgets* seran objectes del tipus *View* que ja coneixem, com botons o quadres de text.

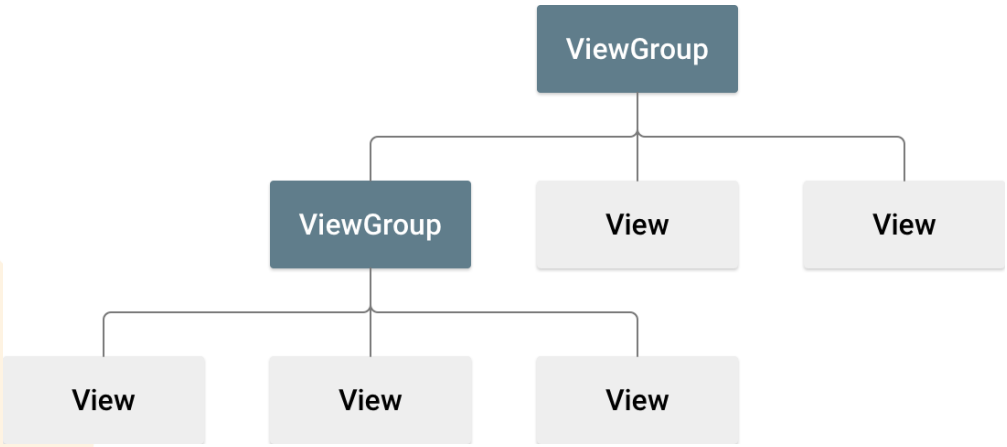


Figura 1: Construcció de la interfície

En general, existeixen principalment tres formes d'organitzar el disseny d'una aplicació:

- **Aplicar un disseny mitjançant *LinearLayouts***: Es tracta de la forma més primitiva per realitzar intefícies. Es tracta de contenidors on es poden afegir elements que s'ajusten bé ven vertical o en horitzontal, segons indique el paràmetre `android:orientation`. A més, disposa de l'atribut `android:gravity` per posicionar els elements i centrar-los horitzontalment, verticalment o d'ambdues maneres. També tenim la possibilitat d'afegir un *pes* (`android:layout_weight`) en cadascun dels elements de dins el layout per especificar en quantes parts es dividirà la vista general i quantes parts ocupa l'element.
- **Realitzar un disseny mitjançant *RelativeLayouts***, un disseny on les posicions de les vistes filles es poden descriure respecte a altres filles o el seu pare, fent referència a ells a través del seu ID, al qual ens referirem amb `android:id="@id/xxx"`. Una vegada referenciat l'element, podem indicar el seu posicionament relatiu (si està dalt, baix, a un costat…)
- **Realitzar el disseny mitjançant *ConstraintLayout***, l'últim en aparéixer, i que ens permet gestionar vistes complexes de forma senzilla i linial. Per tal d'utilitzar-les, ens haurem d'assegurar

que al fitxer `build.gradle` de l'aplicació tenim la dependència:

```
dependencies {
...
implementation 'androidx.constraintlayout:constraintlayout:2.0.2'
...
}
```

Amb els *ConstraintLayouts* podem ajustr les vistes respecte a altres vistes o respecta el pare, a l'igual que fariem amb els *RelativeLayouts*, però amb molts més paràmetrs i possibilitats. El fet d'utilitzar aquests dissenys, fa que es genere un codi XML més linial, sense tants elements anidats, ja que tots estan al mateix nivell, i el posicionament es realitza en base a la relació amb els altres elements. El dissenyador d'Android, ens ajuda bastant en açò.

Articles relacionats

Podeu trobar més informació sobre els diferents tipus de disseny als següents articles:

- *Constraint, Relative y Linear Layout*: https://medium.com/knowing-android/constraint-relative-y-linear-layout-e287eb2b2db1
- *Android Relative vs Constraint Layouts, Cuál es mejor y como usarlo?*: https://codearmy.co/android-relative-vs-constraint-layouts-cual-es-mejor-y-como-usarlo-95c08582ab2e

Documentació oficial

- *Ús de l'editor de disseny*: https://developer.android.com/studio/write/layout-editor?hl=es

Creació d'una interfície senzilla: https://developer.android.com/training/basics/firstapp/building-ui?hl=es
Codelab de Google sobre el Constraint Layout: https://developer.android.com/codelabs/constraint-layout#1

Per altra banda, per vore de manera més visual la creació d'interfícies, podem consultar el següent canal de Youtube:
Canal *La Cueva del Programador*
https://www.youtube.com/channel/UCmV-TWMANQFKZx4iC07agqg

Elements de la interfície

Tots els elements de la interfície d'usuari d'una app d'Android són objectes de tipus *View* o *ViewGroup*. En general, les vistes o objectes de tipus View són objectes que *pinten* en la pantalla i amb les que l'usuari pot interactuar. Per la seua banda, els *ViewGroups* són objectes que contenen altres objectes de tipus *View/ViewGroup*.

Android ens ofereix un conjunt de classes d'aquest tipus per als controls més comuns: botons, quadres dee text, així com diferents dissenys (*layouts*) per organitzar-los de forma jeràrquica. Recordem que com més plana siga aquesta jerarquia, el rendiment serà millor.

Aquests elements d'interfície i disseny es poden crear en temps d'execució, o bé declarar-los en un fitxer d'interfície en format XML, que serà el més habitual.

Documentació oficial
Al lloc de desenvolupadors d'Android disposeu de molta més informació sobre els elements de la interfície, els tipus de disseny, i les seues propietats:

- https://developer.android.com/guide/topics/ui/overview?hl=es-419
- https://developer.android.com/guide/topics/ui/declaring-layout

En aquest apartat, anem a ampliar el nostre coneixement sobre els elements View de tipus botó, els checkboxes i radioButtons i la visualització de llistes.

Tot i que el més habitual serà utilitzar el dissenyador d'interfícies, veurem els diferents elements sobre el fitxer XML, per tal de veure més clar com es plasmen les diferents propietats que utilitzem. No obstant això, la manipulació d'aquestes propietats es pot fer igualment des del dissenyador d'interfícies.

Botons

Els botons són elements d'intefície que consisteixen en un text i/o una icona, que realitzen certa acció quan l'usuari fa clic en ells. Aquests s'implementen amb les classes `Button` , que és una subclasse de `TextView` i `ImageButton` que és una subclasse d'`ImageView`.

En general, l'XML per definir un botó, quedaria:

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    ... />
```

Les propietats que s'han inclòs són:

- L'identificador amb `id`, i valor `@+id/button`, per fer referència a ell amb `button`,
- Les propietats `layout_width` i `layout_height`, per indicar la grandària del botó, amb valor `wrap_content` per a que s'ajuste al contingut, i
- La propietat `text`, amb el text del botó. Cal tindre en compte que aquest text és aconsellable definir-lo com a un recurs de tipus *String*, dins el fitxer `res/values/strings.xml`, de la forma: `<string name="button_text">Text del botó</string>`.

A més, si volem que incloga una imatge, afegirem la propietat `drawableLeft`:

- `android:drawableLeft="@drawable/button_icon`: On estariem afegint el recurs d'imatge *button_icon*, ubicat a la carpeta de recursos `res/drawable`.

Per altra banda, si voleu un botó d'imatge directament, fariem:

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/button_icon"
    android:contentDescription="@string/button_icon_desc"
    ... />
```

Copm veiem, la imatge s'inclou amb l'atribut `src`, i amb `contentDescription` afegim una descrició. En cas que la imatge siga un recurs vectorial, s'utilitzarà, en lloc d'`android:src` la propietat `app:srcCompat`. Doneu una ullada a la documentació d'ampliació per veure com utilitzar-ho!

Responent a events

Si volem que un botó responga a una acció de l'usuari, directament podem utilitzar la propietat `android:onClick` i especificar-li un mètode de la nostra classe associada:

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    ...
    android:onClick="metodeEnClasse" />
```

I en la nostra classe, definim el mètode, que haurà de ser públic, no retornar res i rebre la vista (botó) com a paràmetre.

```
fun metodeEnClasse(view: View){
    ...
}
```

Per altra banda, l'opció més habitual per associar esdeveniments a botons, com ja sabem és fent ús d'un *receptor d'esdeveniments* `OnClicListener`:

```
boto.setOnClickListener {
    // Callback com a resposta al clic
}
```

Sent botó la referència obtinguda amb `FindViewById`, o bé si hem utilitzat el Data Binding:

```
binding.id.setOnClickListener {
    // Callback com a resposta al clic
}
```

Personalització de botons

En Android podem aplicar un tema concret a tots els elements d'una aplicació mitjançant l'atribut `android:theme` a l'etiqueta `<appliccation>`, del fitxer de Manifest. Per defecte, aquest valor

apareix com a: `android:theme="@style/AppTheme"`, però per exemple, a partir d'Android 4.0 podem utilitzar el tema Holo indicant `android:theme="@android:style/Theme.Holo"`.

Si volem utilitzar un estil personalitzat per als diferents botons de la nostra aplicació, podem fer les següents modificacions:

- Afegir un color de fons, mitjançant l'atribut `android:background`, i fent referència a un recurs de tipus color o *drawable*.
- Modificar la font, mitjançant `android:fontFamily`, el seu color amb `android:textColor`, habilitar o deshabilitar les lletres capitals amb `android:textAllCaps` o canviar la grandària amb `android:textSize`, entre d'altres.

Per altra banda, també podem aplicar un estil directament al botó, mitjançant la propietat `style` en l'etiqueta `<Button>`. Per exemple per fer ús d'un estil de botó sense vores (*borderless*), podem utilitzar:

```
<Button
    android:id="@+id/button"
    ...
    style="?android:attr/borderlessButtonStyle" />
```

Podeu trobar més informació sobre botons i com personalitzar-los a la doccumentació que se us facilita més avall.

Documentació oficial

- Sobre l'element Button: https://developer.android.com/guide/topics/ui/controls/button
- Sobre les imatges de tipus vectorial: https://developer.android.com/studio/write/vector-asset-studio?hl=es-419
- Especificació de la classe Button: https://developer.android.com/reference/android/widget/Button

Checkboxes, RadioButtons, Toggle Buttons i Spinners

Les caselles de verificació (*checkboxes*) i els botons de selecció (*radiobuttons*) són elements que ens permeten seleccionar opcions entre un conjunt. Mentre que les caselles de verificació permeten se-

leccionar vàries opcions ente una o més opcions d'un conjunt, els botons de selecció ens serviran per tal que l'usuari seleccione una única opció d'entre un conjunt.

En aquest apartat anem a veure aquests dos tipus d'elements de la interfície més els Spinners i els Toggle Buttons.

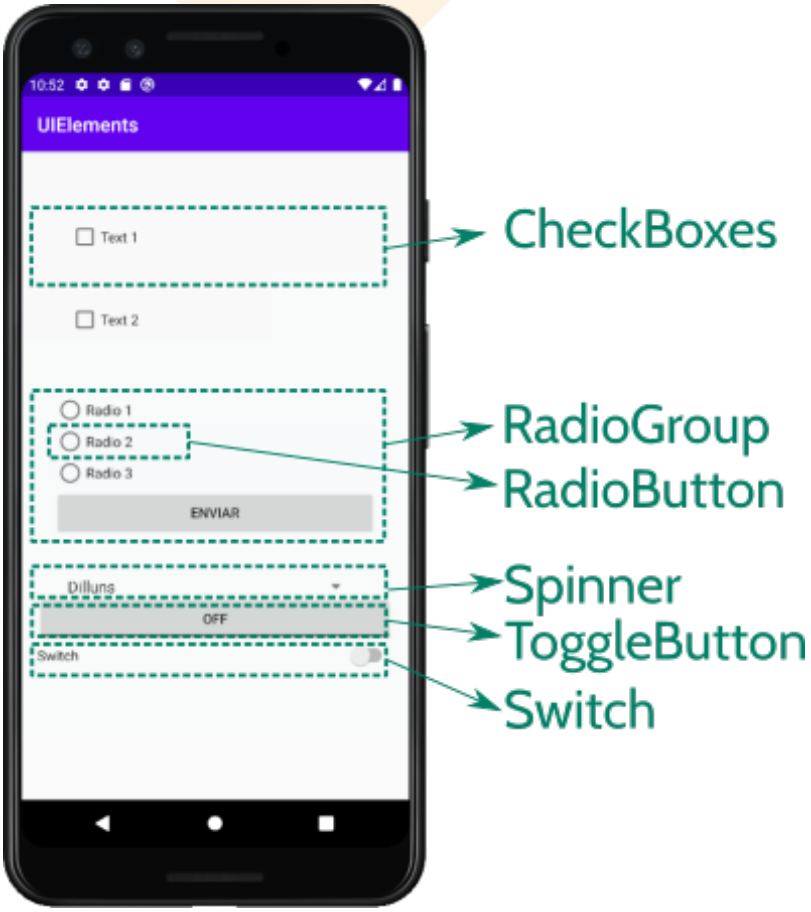


Figura 2: Elements de la interfície

Caselles de verificació o ChecBoxes

Per tal de crear una casella de verificació, crearem un objecte de tipus `checkBox` al nostre disseny. Com que es poden triar vàries opcions, cada checkbox es gestiona de forma separada, i caldrà registrar un receptor d'esdeveniments per a cadascun.

L'event que es dispararà quan fem clic en un `CheckBox` és *onClick*, i pot ser capturat bé directament a l'etiqueta `CheckBox` mitjançan l'atribut `android:onClick` o bé mitjançant un receptor d'esdeveniments (eventListener) capturant l'event `onClick`. El funcionament és el mateix que hem

vist per als botons comuns. Una vegada dins el mètode que gestione l'event, podem accedir al seu valor amb la propietat `isChecked`.

Veiem un exemple. Temim els següents botons *checkbox1* i *checkbox2* dins un Layout:

```
<CheckBox android:id="@+id/checkbox1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/text1"
    android:onClick="MetodeEnFerClic"/>

<CheckBox android:id="@+id/checkbox2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/text2" />
```

Si ens fixem, el primer *checkbox* especifica el mètode a executar quan es fa clic en ell, mentre que el segon serà gestionat per un receptor d'esdeveniments.

Un possible codi per gestionar ambdós events seria:

```
class MainActivity : AppCompatActivity() {

    // Mètode que s'invoca directament quan fem clic
    // en el primer checkbox.
    fun MetodeEnFerClic(view: View){
        // Comprovem que és un checkbox
        if (view is CheckBox) {
            val activat:String=if (view.isChecked) "activat" else
             ↪  "desactivat"
            val text = "Has "+activat+" el checkbox "+view.text
            // Congigurem el toast
            val duration = Toast.LENGTH_SHORT
            // Creem el toast i el mostrem
            Toast.makeText(applicationContext, text, duration).show()
        }
    }

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

```
        // EventListener per al checkbox2, que invocarà
        // el mateix mètode que invocava directament
        // el primer checkbox.
        // Fixeu-vos que per simplificar, no hem fet el
        // data binding, sinò que utilitzem directament
        // la referència a l'objecte obtinguda amb findViewById:
        findViewById<CheckBox>(R.id.checkbox2).setOnClickListener { view:
↪   View ->
            MetodeEnFerClic(view)
        }
    }

}
```

Per altra banda, si volem canviar des del propi codi el valor del checkbox, ho podem fer amb els mpetodes `setChecked(boolean)` per indicar un valor concret, o bé el mètode `toggle()` per canviar el seu estat.

Documentació oficial
Documentació i guía sobre el Checkbox: https://developer.android.com/guide/topics/ui/controls/checkbox
Referència del widget Checkbox: https://developer.android.com/reference/android/widget/CheckBox

Botons de selecció o RadioButtons

Els botons de selecció ens permeten escollir una entre un conjunt d'opcions mútuament excloents. Aquests botons s'implementen amb la vista `RadioButton`. Per tal d'indicar que diversos `RadioButtons` són excloents, els posem a tots dins un mateix `RadioGroup`.

La gestió d'esdeveniments és exactament igual que als *CheckBoxes* i als botons. Quan fem clic al botó es dispara l'esdeveniment *onClick*, que podem gestionar bé directament mitjançant el corresponent atribut en l'etiqueta `RadioButton` o bé amb un receptor d'esdeveniments des del codi. Una vegada dins el mètode que gestione l'esdeveniment, també tenim accés a la propietat `isChecked`, per comprovar si l'element està seleccionat o no.

Veiem un exemple amb tres radiobuttons:

```
<RadioGroup
        android:id="@+id/rg1"
        ...
        >

        <RadioButton
            android:id="@+id/rb1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/radio1"/>

        <RadioButton
            android:id="@+id/rb2"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/radio2"/>

        <RadioButton
            android:id="@+id/rb3"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/radio3"/>

        <Button
            android:id="@+id/btEnviar"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/btEnviar" />

    </RadioGroup>
```

Cal tindr en compte, que el contenidor `RadioGroup` és una subclasse de`LinearLayout` amb orientació vertical, de manera que els `RadioButtons` que anem afegint a dins s'aniran ordenant amb aquesta orientació.

Com podem veure, a l'exemple, hem afegit un botó a dins el *RadioGroup*. Aquest botó podría anar tant fora com dins, però s'ha afegit dins per tal que entre dins aquest *LinearLayout*.

Veiem com quedaria un possible codi per gestionar el clic al botó i mostrar un toast en funció del *RadioButton* seleccionat:

```
lateinit var
 ↪  binding:com.ieseljust.pmdm.menus3.databinding.ActivityMain2Binding;

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

        var binding = ActivityMain2Binding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.btEnviar.setOnClickListener { view: View ->
            // Busquem dins del RadioGroup,
            // quin radioButton està seleccionat
            // i ens quedem amb el seu ID
            val idRadioButtonSeleccionat: Int =
             ↪  binding.rg1!!.checkedRadioButtonId

            // Declarem un objecte de tipus radioButton
            val radioButton: RadioButton
            // I l'assignem al radioButton que té aquest ID
            radioButton = findViewById(idRadioButtonSeleccionat)

            // Ara creem el text per al toast
            var text = "Ha seleccionat el RadioButton "
            // I li afegim el text del radioButton
            text+=radioButton.text

            // Finalment, mostrem el toast
            Toast.makeText(baseContext, text, Toast.LENGTH_SHORT).show()
        }
    }
```

Per altra banda, recordeu que si voleu inicialitzar un *RadioButton* o modificar el seu estat des del codi, podeu utilitzar els mètodes `setChecked(boolean)` o `toggle()`.

onCheckedChanged

També podeu capturar cada vegada que es canvia el contingut d'algun *RadioButton* dins un *RadioGroup* amb el mètode `onCheckedChanged`.
Podeu trobar com fer-ho als següents enllaços:

- Article *How to bind method on RadioGroup on checkChanged event with data-binding*
- Article *Android Radio Button Using Kotlin With Example*

Documentació oficial

- Documentació i guía sobre el RadioButton: https://developer.android.com/guide/topics/ui/controls/radiobutton
- Referència del widget RadioGroup: https://developer.android.com/reference/android/widget/RadioButton
- Referència del widget RadioGroup: https://developer.android.com/reference/android/widget/RadioGroup

Spinners

Els spinners ofereixen una forma ràpida de triar un valor d'un conjunt. A diferència dels *RadioButtons*, les diferents opcions disponibles no estan visibles, sinò que es *despleguen* en fer clic sobre l'spinner, que en tot moment mostra el valor seleccionat. Per tal d'afegir un spinner des del dissenyador d'interfícies, cal buscar-lo en la paleta de *Containers* > *Spinner*

L'XML corresponent a un Spinner seria:

```
<Spinner
      android:id="@+id/dies_setmana"
      ..
      />
```

A diferència dels RadioButtons, en un Spinner no indiquem la llista d'opcions en l'XML, sinò que ho fem des del propi codi font, en una activitat o fragment. Les classes implicades en aquest procés seran `Spinner`, `SpinnerAdapter` i `AdapterView.OnItemSelectedListener`.

Adapter és un patró de disseny estructural, que ens permet que objectes amb diferents interfícies puguen col·laborar entre ells fent ús d'aquest adaptador, que *adaptarà* la interfície d'un objecte de manera que altre puga entendre-la.
Podeu trobar més informació sobre aquest patró a https://refactoring.guru/es/design-patterns/adapter

Per tal d'incorporar aquesta llista d'opcions necessitarem d'un `SpinnerAdapter`, com puguen ser un `ArrayAdapter` si el contingut es troba en un vector o bé un `CursorAdapter` si està en una consulta a la base de dades.

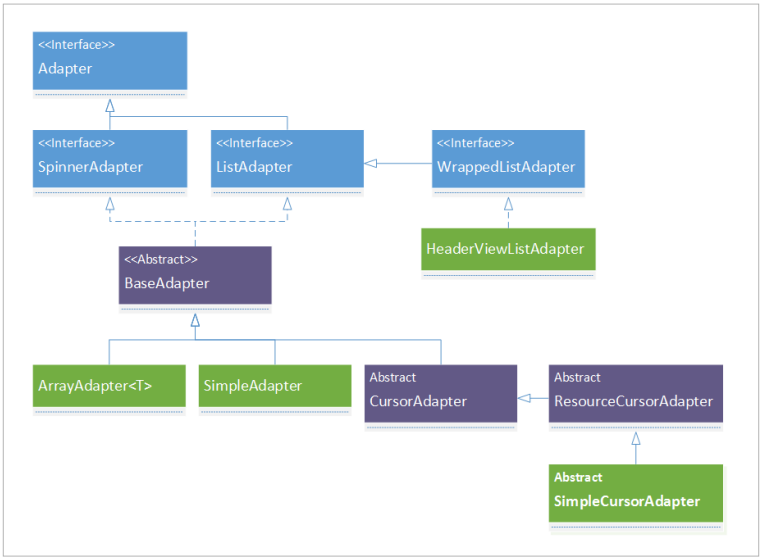
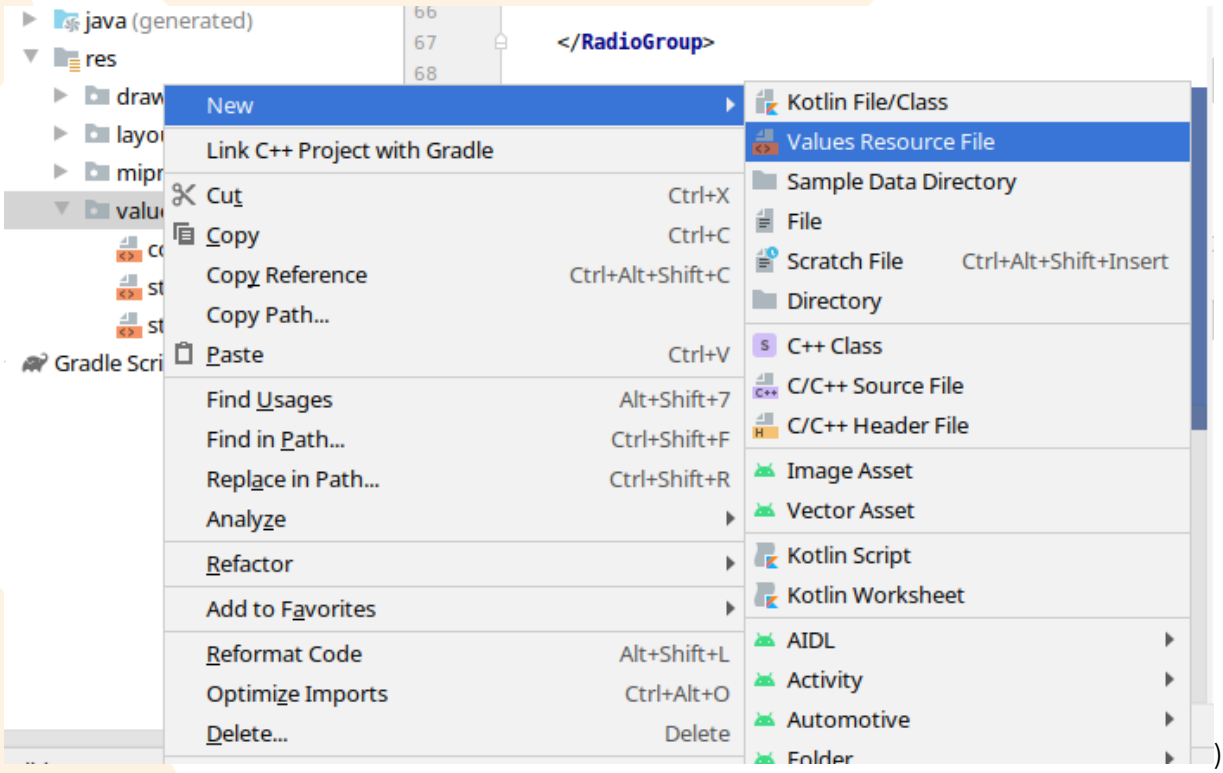


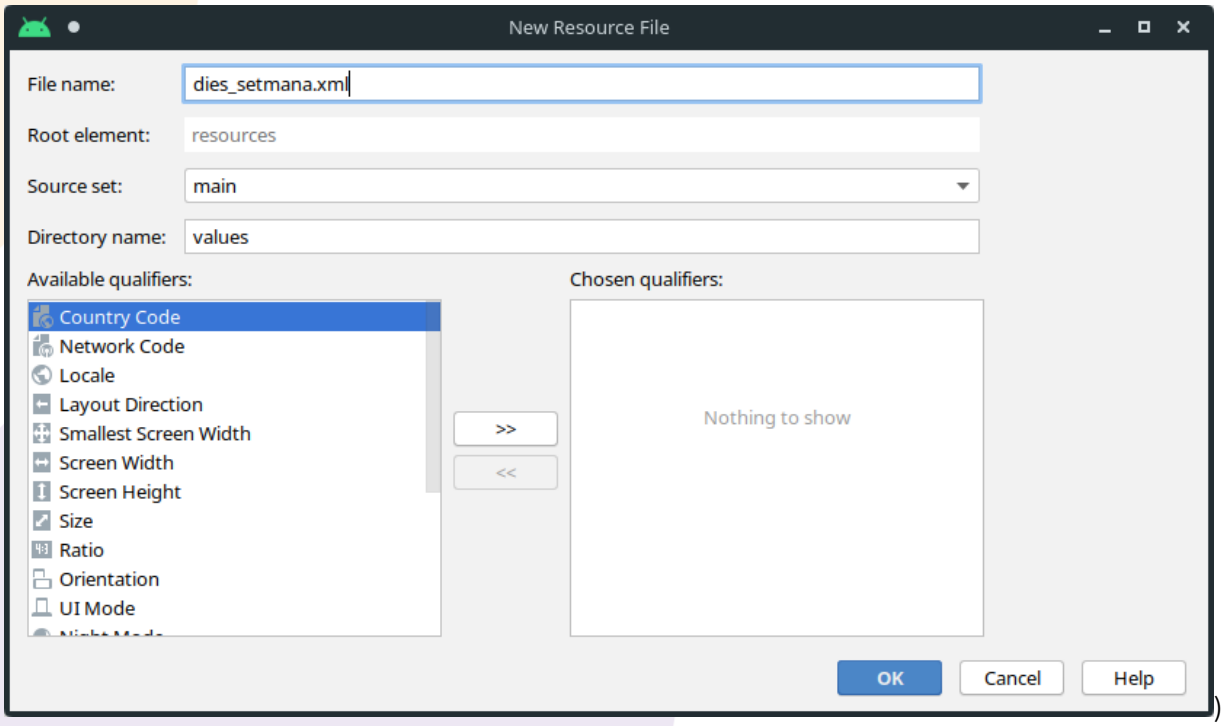
Figura 3: Jerarquía de la classe Adapter en Android. Font: https://www.intertech.com/android-adapters-adapterviews/

Per exemple, anem a definir-nos un nou fitxer de recursos de tipus *values* amb un vector. Per a això, ens situem al directori *res/values* i fem clic al botó dret, per triar l'opció *New* > *Values Resource File*:



){ width=500px }

I creem un nou recurs anomenat *dies_setmana.xml*:



){ width=500px }

El contingut d'aquest fitxer serà el següent:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="dies_setmana">
        <item>Dilluns</item>
        <item>Dimarts</item>
        <item>Dimecres</item>
        <item>Dijous</item>
        <item>Divendres</item>
        <item>Dissabte</item>
        <item>Diumenge</item>
    </string-array>
</resources>
```

Per tal de *poblar* el nostre *Spinner* amb aquest contingut, faríem ús d'una instància d'*ArrayAdapter*, de la següent forma:

```
// Creem un mètode específic per poblar l'spinner
private fun populateSpinner(){
    // Creem un ArrayAdapter a partir d'un recurs de tipus array
    // Requereix tres paràmetres: El context, el recurs, i el
    // disseny de l'entrada (utilitzarem el proporcionat per la
    // pròpia plataforma.
    ArrayAdapter.createFromResource(
        this,
        R.array.dies_setmana,
        android.R.layout.simple_spinner_item
    ).also { adapter ->
        // Si tenim l'adaptador preparat, seleccionem el disseny
        // per a la llista d'opcions (el proporcionat per la plataforma)
        adap-
 ↪  ter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
        // I finalment, afegim l'adaptador a l'spinner
        // A l'id del qual accedim a través del binding
        binding.diesSetmana.adapter = adapter
    }
}

// I l'invoquem des del mètode onCreate
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
```

```
    ...
    populateSpinner()
}
```

Fixeu-vos que hem fet us de la següent construcció en Kotlin:

```
    ArrayAdapter.createFromResource(...)
    .also { adapter ->

    }
```

El mètode/funció d'àmbit .also
El mètode o funció d'àmbit `.also` és molt semblant al `.let`, i serveix per a una execució condicional de tot un bloc en cas que el que li passem no siga nul, estalviant-nos així fer aquesta comprovació.
La principal diferència és que amb `let` es retorna el resultat d'una funció, mentre que `also` retorna l'objecte en sí (te *efectes col·laterals*).
Principalment, aquesta construcció s'utilitza quan volem fer operacions addicionals amb un objecte. En aquest cas, hem creat un adaptador, i *a més* de crear-lo, l'hem incorporat a l'spinner.

Com sabem la selecció de l'usuari?

Cada vegada que l'usuari canvia la selecció es dispara un event de tipus *ItemSelected*. Per tal de capturar aquest event, cal implementar la interfície *AdapterView.OnItemSelectedListener* i el callback *onItemSelected()* corresponent, definit en aquesta interfície.

El codi que caldría modificar/afegir a la classe vindría a ser:

```
// Implementem la interfície AdapterView.OnItemSelectedListener
class MainActivity2 : ActivitatAmbMenus(),
 ↪  AdapterView.OnItemSelectedListener {

    // Sobreescrivim els mètodes de la interfície onItemSelected i on
     ↪  NothingSelected
    override fun onItemSelected(parent: AdapterView<*>, view: View, pos:
     ↪  Int, id: Long) {
```

```
        // Es dispara quan es selecciona un item en l'spinner
        // per tal d'obtenir l'element seleccionat, ferm ús de
        // parent.getItemAtPosition(pos)
        var text = "Selecció de l'spinner: "+ parent.getItemAtPosition(pos)

        Toast.makeText(this@MainActivity2,
            text, Toast.LENGTH_SHORT).show()
    }


    override fun onNothingSelected(parent: AdapterView<*>) {
        Toast.makeText(this@MainActivity2,
            "No s'ha seleccionat res", Toast.LENGTH_SHORT).show()
    }


    override fun onCreate(savedInstanceState: Bundle?) {
        .
        .
        .
        // Invocacó al mètode populateSpinner
        populateSpinner()
        // Associem l'event onItemSelected per a que
        // el gestione la nostra classe
        binding.diesSetmana.onItemSelectedListener = this
        ...
    }
}
```

Algunes alternatives a aquest codi qur també ens podem trobar són:

- Per accedir al conringut del vector, en lloc de fer ús del `parent`, podem recuperar el recurs del vector i accedir a la posició seleccionada.

```
override fun onItemSelected(parent: AdapterView<*>, view: View, pos:
 ↪  Int, id: Long) {
        val dies_setmana =
         ↪  resources.getStringArray(R.array.dies_setmana)
        var text = "Selecció de l'spinner "+ dies_setmana[pos]

        Toast.makeText(this@MainActivity,
            text, Toast.LENGTH_SHORT).show()
```

```
}
```

- Fer un d'un objecte d'una classe anònima que implemente la interfície `AdapterView.OnItemSelectedListener`, de manera que no l'hajam d'implementar a la nostra activitat principal:

```
class MainActivity : AppCompatActivity() {

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

        // Creem un objecte anònim que implemente
        ↪ AdapterView.OnItemSelectedListener
        // i sobreescrivim aci els seus mètodes.
        binding.diesSetmana.onItemSelectedListener = object :
            AdapterView.OnItemSelectedListener {
            override fun onItemSelected(parent: AdapterView<*>,
                                        view: View, pos: Int, id: Long) {
                var text = "Selecció de l'spinner: "+
                ↪ parent.getItemAtPosition(pos)
                Toast.makeText(this@MainActivity,
                    text, Toast.LENGTH_SHORT).show()
            }

            override fun onNothingSelected(parent: AdapterView<*>) {
                Toast.makeText(this@MainActivity,
                    "No s'ha seleccionat res", Toast.LENGTH_SHORT).show()
            }
        }
    }
    ...
}
```

Documentació oficial
Guía sobre els spinners: https://developer.android.com/guide/topics/ui/controls/spinner
Articles
Spinners en Kotlin, de *Geeks for Geeks*: https://www.geeksforgeeks.org/spinner-in-kotlin/

ToggleButtons

Els Toggle Buttons són botons que ens permeten alternar entre dos estats. L'objecte que gestiona aquests controls és el `ToggleButton`. A partir d'Android 4.0 (API level 14), s'introdueix altre tipus de Toggle Button anomenat switch, controlats per l'objecte `Switch` i `SwitchCompat` per fer-los compatibles amb versions anteriors.

Per tal de modificar-ne l'estat, podem fer ús dels mètodes `setChecked()` o `toggle()`.

Veiem com expressariem aquests elements en l'XML:

- Per al toggle:

```
<ToggleButton
    android:id="@+id/toggleButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="ToggleButton" />
```

- Per al Switch:

```
<Switch
    android:id="@+id/switch1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Switch" />
```

Per tal de detectar l'activació de qualsevol d'aquests botons, farem ús del mètode `setOnCheckedChangeListener()` per assignar el callback:

```
    toggleButton.setOnCheckedChangeListener { _,  isChecked ->
        var text = if (isChecked) "El toggle està activat"
                   else "El toggle està desactivat"
        Toast.makeText(this@MainActivity,
```

```
            text, Toast.LENGTH_SHORT).show()
        }

        switch1.setOnCheckedChangeListener { _,  isChecked ->
            var text = if (isChecked) "El switch està activat"
            else "El switch està desactivat"
            Toast.makeText(this@MainActivity,
                text, Toast.LENGTH_SHORT).show()
        }
```

Documentació oficial
Guía dels toggle Buttons: https://developer.android.com/guide/topics/ui/controls/togglebutton

Components Webviews

Els Webviews són components que proporcionen una vista de *Navegador web*, capaç d'interpretar codi HTML i javascript dins les nostres aplicacions.

La classe webview descendeix de la classe View d'Android, i ens permet mostrar pàgines web com a part del disseny de la nostra activitat. Aquesta vista no inclou funcions com controls de navegació o barra d'adreces.

Per afegit un webview als nostres dissenys, podem arrossegar des de la paleta a la secció de *Widgets* el component *WebView*, o generar el següent codi XML:

```
<WebView
    android:id="@+id/webview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>
```

Des del nostre codi, per tal de carregar una pàgina al webView farem ús del mètode `loadUrl` des del mètode `onCreate` de la vista:

```
val myWebView: WebView = findViewById(R.id.webview)
myWebView.loadUrl("http://www.ieseljust.com")
```

Per defecte, javascript està inhabilitat al webview. Si volem habilitar-lo caldrà indicar-ho amb:

```
myWebView.settings.javaScriptEnabled = true
```

Per tal que aquest codi funcione, necessitarem accés a Internet, i caldrà demanar-li permís a l'usuari. Per a això, al nostre fitxer de Manifest haurem d'indicar-ho explícitament:

```
<manifest ... >
   <uses-permission android:name="android.permission.INTERNET" />
    ...
</manifest>
```

> **Documentació oficial**
>
> Sobre webview: https://developer.android.com/guide/webapps/webview
>
> Sobre WebViewAssetLoader: https://developer.android.com/reference/androidx/webkit/WebViewAssetLoader