Ciclo de vida de componentes

Android Adaptadores

{desafío} latam_

Utilizar elementos del ciclo de vida para la implementación de un aplicativo Android que resuelve un problema.

- Unidad 1: Kotlin para el desarrollo de aplicaciones.
- Unidad 2: Ciclo de vida de componentes.
- Unidad 3: Arquitectura en Android.
- Unidad 4: Programación asíncrona en Android.

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Describir el rol de los adaptadores y del reciclador en un programa Andoid*

¡Hagamos un experimento!

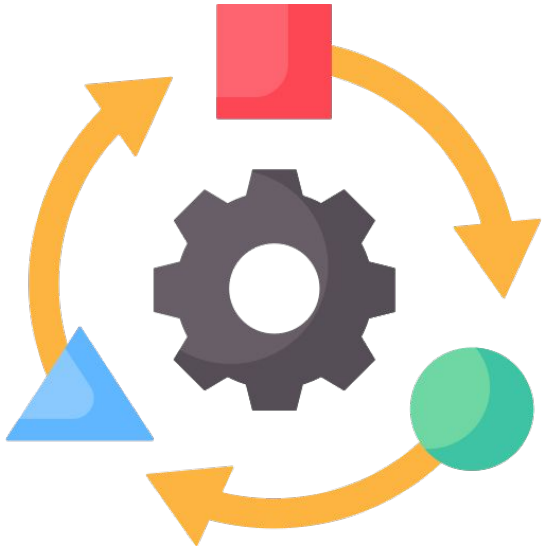
Toma tu celular, revisa las aplicaciones instaladas y responde ¿Cuentan con adaptadores? ¿Cómo lo supiste?

/* Qué son los adaptadores */

¿Qué son los adaptadores?

Un Adaptador o Adapter actúa como un puente entre un AdapterView y los datos subyacentes para esa vista. El adaptador proporciona acceso a los elementos de datos. También es responsable de crear una vista para cada elemento del conjunto de datos.

El adapter permite mostrar en un objeto tipo lista, fragmentos de vistas

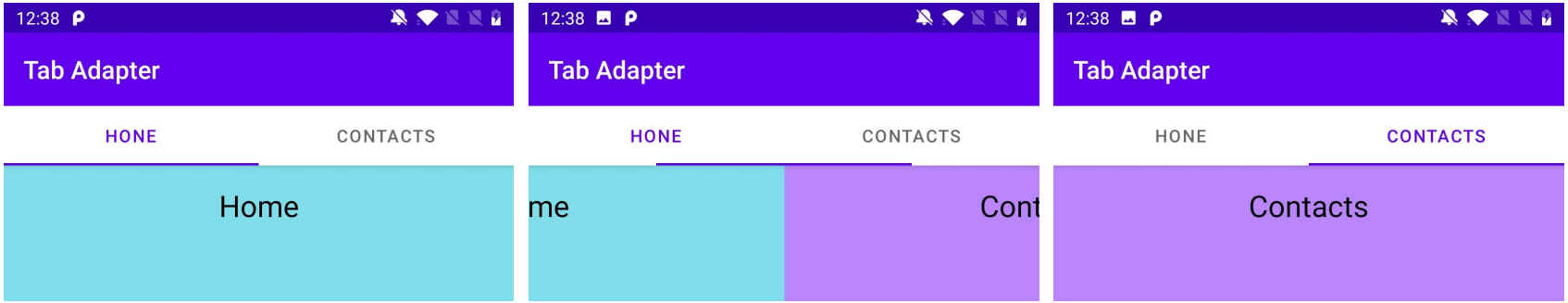


/* Adaptador para fragmentos */

ViewPager Adapter (`ViewPager2`)

ViewPager, (específicamente `ViewPager2`) permite navegar entre vistas hermanas, fragments en el mismo nivel, como pestañas, con un gesto de dedo horizontal o deslizar.

Este patrón de navegación también se conoce como paginación horizontal.



¿Cómo se usa ViewPager Adapter (ViewPager2)?

Una implementación sencilla sería la siguiente:

```
class SectionsPagerAdapter(fm: FragmentManager) :
    FragmentStatePagerAdapter(fm, BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT) {

    override fun getItem(position: Int): Fragment = when (position) {
        0 -> HomeFragment()
        1 -> ContactsFragment()
        else -> HomeFragment()
    }

    override fun getPageTitle(position: Int): CharSequence = when (position) {
        0 -> "Hone"
        1 -> "Contacts"
        else -> "Home"
    }

    override fun getCount(): Int = 2
}
```

¿Qué hace esto? El adaptador crea una instancia de un fragment y un título. Dependiendo de la posición del adaptador, se puede combinar con un TabLayout. No es obligatorio, solo sirve para mostrar el título del fragment activo

/* Historia de los adaptadores y porqué surge el reciclador */

Un poco de historia

Originalmente cuando se quería mostrar una lista de vistas, se podía usar un ListView, el problema ocurre cuando las vistas que se querían mostrar eran muy complejas o las listas muy largas.

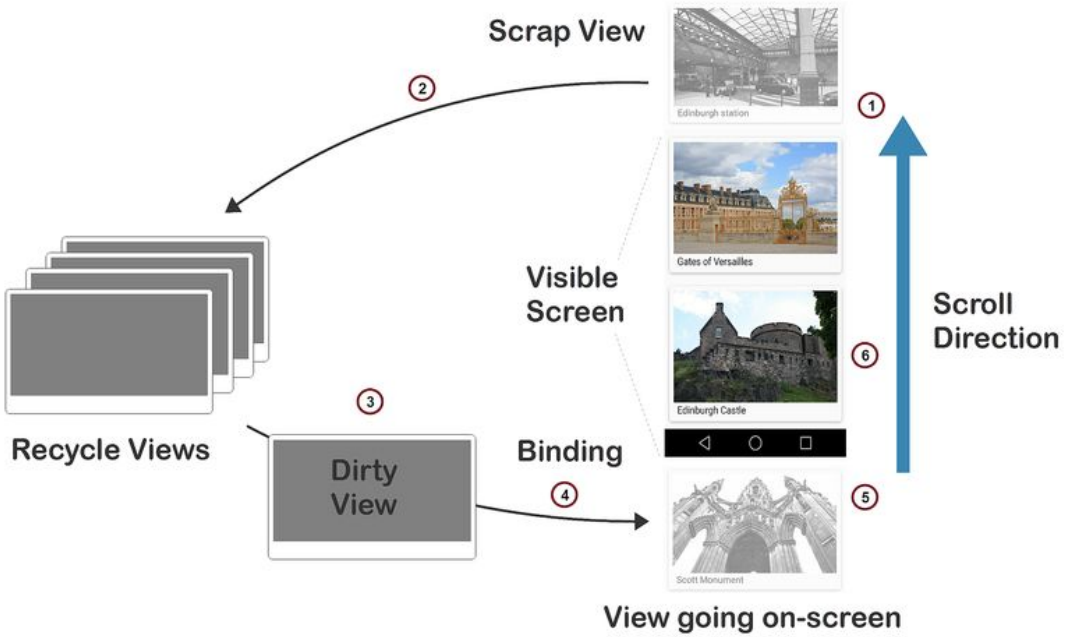
El ListView sirvió durante mucho tiempo. Pudimos cubrir la mayoría de los casos. Pero las necesidades del usuario son diferentes ahora y son mucho más desafiantes. Los diseños de listas se volvieron cada vez más complejos y ListView no estaba ayudando a manejarlos.

Afortunadamente, se introdujo RecyclerView y se resolvieron muchos problemas. Es más eficiente por defecto, sus animaciones son más simples, el diseño es más fácil de usar y la API es mucho más agradable.

/* Adaptador reciclador para listas */

¿Cómo funciona el RecyclerView?

Cuando el usuario usa el scroll para mover una lista de objetos, estos se van cargando en memoria (pre cargando) antes de ser mostrados, y los que ya no se encuentran en el área visible, se mantienen en memoria por un tiempo en caso de que el usuario decida usar el scroll en dirección contraria, todo esto se hace con la ayuda del adapter.



{desafío}
latam_

RecyclerView Adapter

Un RecyclerView Adapter debería estar acompañado siempre de su ViewHolder, el cual es el objeto que se quiere listar en el RecyclerView. Por ejemplo, el siguiente adaptador se encarga de mostrar una lista de contactos:

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

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

    override fun onBindViewHolder(holder: ContactViewHolder, position: Int) {
        holder.onBind(contacts[position])
    }

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

ViewHolder

El ViewHolder es el objeto que se quiere listar, y puede ser definido como inner class del recyclerview adapter en caso de que ese sea el único adaptador que hará del view holder.

binding nos permite acceder a la vista de diseño en XML y el objeto contact, proviene de la lista de contactos que es recibida en el RecyclerView Adapter.

```
inner class ContactViewHolder(private val binding:
ItemContactBinding) :
        RecyclerView.ViewHolder(binding.root) {
        fun onBind(contact: Contact) {
            with(binding) {
                contact.run {
                    tvContactName.text = name
                    tvContactPhoneNumber.text = phoneNumber
                }
            }
        }
    }
```

RecyclerView Adapter + ViewHolder

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

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

    override fun onBindViewHolder(holder: ContactViewHolder, position: Int) {
        holder.onBind(contacts[position])
    }

    override fun getItemCount(): Int = contacts.size

    inner class ContactViewHolder(private val binding: ItemContactBinding) :
        RecyclerView.ViewHolder(binding.root) {
        fun onBind(contact: Contact) {
            with(binding) {
                contact.run {
                    tvContactName.text = name
                    tvContactPhoneNumber.text = phoneNumber
                }
            }
        }
    }
}
```

RecyclerView Adapter + ViewHolder

En algunas ocasiones puede ser necesario mostrar más de un tipo de objeto en el RecyclerView, es decir, vamos a tener que manejar más de un ViewHolder.
En el ejemplo anterior vimos que en el método `onCreateViewHolder`, éste retorna un `ContactViewHolder`.

Ahora imagina que tenemos un chat. Si no creamos distintos viewHolder, no es posible a simple vista quién es el autor de los mensajes, para esto debemos crear al menos dos ViewHolder, una para los mensajes que envías y otro para los que recibes.

Es necesario identificar de alguna forma quien es el autor de los mensajes, para el ejemplo del Chat se puede hacer por medio de un rol de usuario, pero para otros casos dependerá de la lógica de negocio.

Ejemplo RecyclerView Adapter + 2 ViewHolder

Podemos usar `getItemViewType` para identificar el tipo de vista, ejemplo:

```
override fun getItemViewType(position: Int): Int
{
        return when (getItem(position)?.role) {
            UserRole.ADMIN -> VIEW_TYPE_ADMIN
            UserRole.USER -> VIEW_TYPE_USER
            else -> VIEW_TYPE_USER
        }
    }
```

Luego podemos crear un ViewHolder diferente, dependiendo del rol del usuario

```
override fun onCreateViewHolder(parent: ViewGroup,
viewType: Int): RecyclerView.ViewHolder =
        when (viewType) {
            VIEW_TYPE_USER ->
MyDisplayItem(inflate ...)
            VIEW_TYPE_ADMIN ->
OtherDisplayItem(inflate ...)
            else -> MyDisplayItem(inflate ...)
        }
```

ConcatAdapter

Ya que los diseños pueden ser más complejos que simplemente mostrar una lista del mismo tipo de objeto repetido varias veces, a veces es necesario combinar no solo distintos tipos de objetos (definidos por más de un ViewHolder), sino que distintos tipos de adaptadores.

Para esto existe una herramienta llamada ConcatAdapter, como el nombre lo dice “concatena adaptadores”, y forma en que lo hace es secuencial y muy sencilla.

Veamos el siguiente ejemplo::

ConcatAdapter

Supongamos que queremos mostrar una lista de objetos, pero además queremos que el primer ítem sea distinto: queremos que sea una cabecera, de igual forma, que el último ítem sea diferente a los demás. Todo esto independiente que cuantos ítems tenga el adaptador con el contenido principal.
Suena bastante complejo.

```
val headerAdapter: HeaderAdapter = ...
val contentAdapter: ContentAdapter = ...
val footerAdapter: FooterAdapter = ...

val concatAdapter = ConcatAdapter(headerAdapter,
contentAdapter, footerAdapter)

recyclerView.adapter = concatAdapter
```

Como vemos en el ejemplo, basta con pasar los distintos tipos de adaptadores al ConcatAdapter y luego usar el resultado como adaptador para el recyclerView.

Resumen

Hemos terminado de ver que son los adaptadores, sin embargo, es un tema extremadamente amplio, y como ya te habrás dado cuenta, los adaptadores están por todas partes.

Te dejo unas recomendaciones:

- Antes de implementar un adaptador, ten siempre presente lo aprendido anteriormente durante el curso: recuerda el lifecycle de los activities y fragments.
- Los ViewHolders deben mostrar solo lo necesario, no los recargues con datos que no necesitas.

Próxima sesión...

- *Revisar el material de estudio sincrónico que consiste en una guía de estudio para practicar los conceptos aprendidos en esta sesión.*

{desafío} latam_

{desafío}
latam_

Academia de
talentos digitales