# Ciclo de vida de componentes

## Android (Parte I)

{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.

# ¿Qué aprenderás en esta sesión?

- *Reconocer los aspectos fundamentales del ciclo de vida de Activities y Fragments en Android para el desarrollo de aplicaciones*

{desafío}
latam_

# ¿Conoces el ciclo de vida de un software?

{desafío} latam_

# /* Android Life Cycle */

# Android Life Cycle

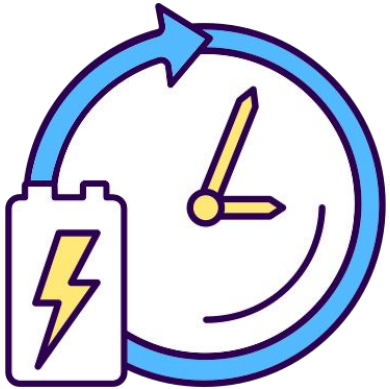
Cuando se habla de ciclo de vida en Android es probable que se refiera a uno de los siguientes casos:


{desafío} latam_

En esta unidad nos enfocaremos en los ciclos de vida de los **activities y fragments**

# /* El ciclo de vida de las activities */

# El ciclo de vida de las activities



Para navegar por las transiciones entre las etapas del ciclo de vida de una actividad, la clase Activity proporciona un conjunto básico de seis devoluciones callbacks: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()` y `onDestroy()`.

# El ciclo de vida de las activities

Con el correcto uso del ciclo de vida, se garantiza que la app:

1. No falle si el usuario recibe una llamada telefónica o cambia a otra app mientras usa la tuya.
2. No consuma recursos valiosos del sistema cuando el usuario no la utilice de forma activa.
3. No pierda el progreso del usuario si este abandona tu app y regresa a ella posteriormente.
4. No falle ni pierda el progreso del usuario cuando gire la pantalla entre la orientación horizontal y la vertical.

# Devoluciones Callbacks

1. **`onCreate()`**: Es llamado cuando el sistema crea el activity por primera vez, en este método, ejecutas la lógica de arranque básica de la aplicación que debe ocurrir una sola vez en toda la vida de la actividad. Por ejemplo, aquí definimos nuestro view binding

2. **`onStart()`**: Hace que el usuario pueda ver el activity mientras la app se prepara para que esta entre en primer plano y se convierta en interactiva. Por ejemplo, este método es donde la app inicializa el código que mantiene la IU.

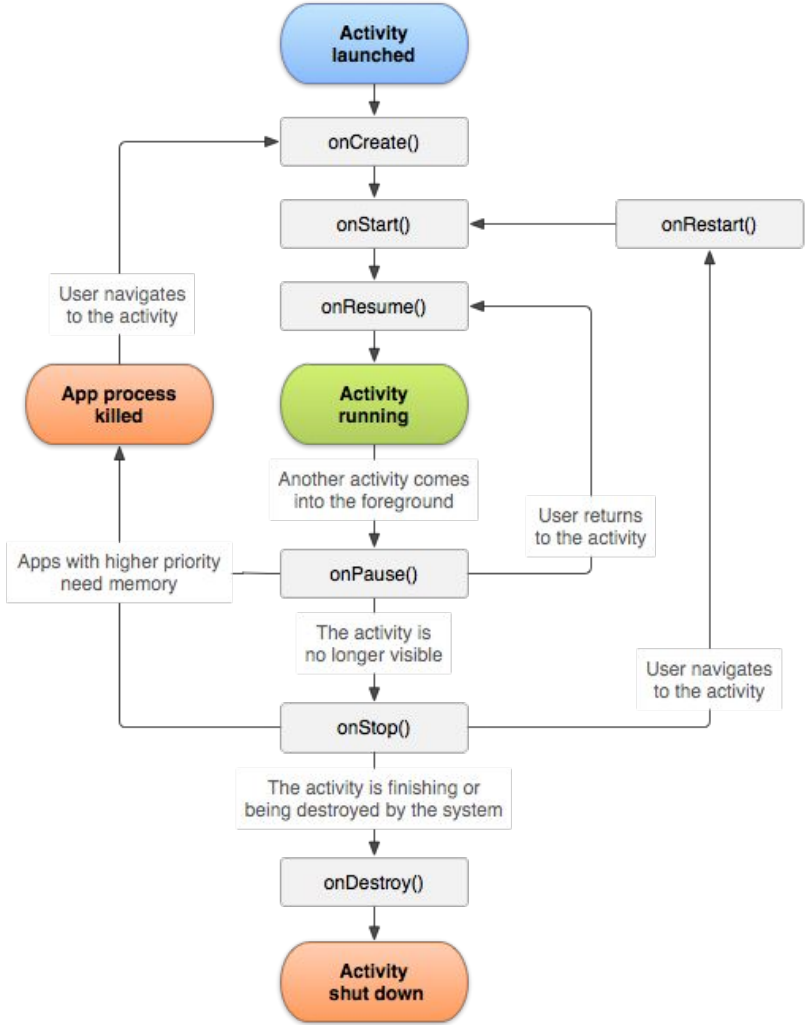
3. **`onResume()`**: Este es el estado en el que la app interactúa con el usuario. La app permanece en este estado hasta que ocurre algún evento que la quita de foco. Por ejemplo, que el usuario navegue a otra Activity.

# Devoluciones Callbacks

1. **`onPause()`**: El sistema llama a este método como la primera indicación de que el usuario está abandonando su actividad (aunque no siempre significa que la actividad se está destruyendo); indica que la actividad ya no está en primer plano.

2. **`onStop():`** Cuando el usuario ya no puede ver tu actividad, significa que ha entrado en el estado Stopped, y el sistema invoca la devolución de llamada onStop(). El sistema también puede llamar a onStop() cuando haya terminado la actividad y esté a punto de finalizar.

3. **`onDestroy()`**Se llama a onDestroy() antes de que finalice la actividad. El sistema invoca esta devolución de llamada por los siguientes motivos:
   a. La actividad está terminando (debido a que el usuario la descarta por completo o a que se llama a finish()).
   b. El sistema está finalizando temporalmente la actividad debido a un cambio de configuración (como la rotación del dispositivo o el modo multiventana).



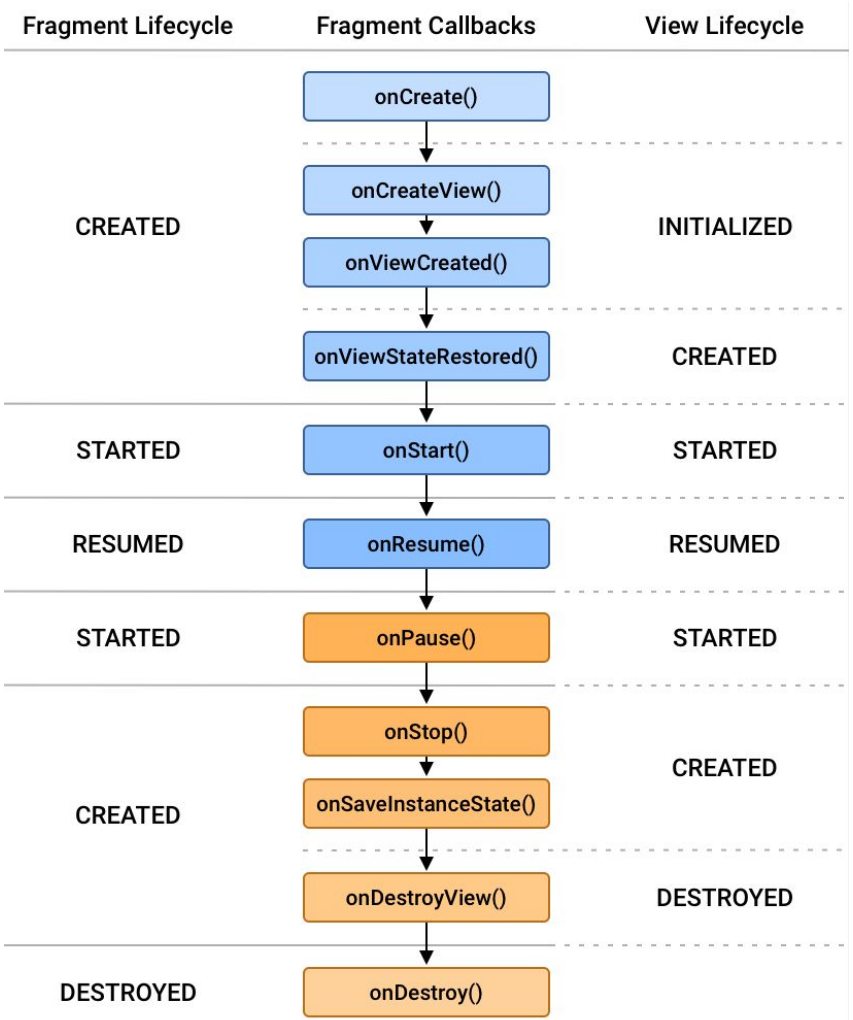
# /* El ciclo de vida de los fragments */

# El ciclo de vida de los fragments

Para administrar el ciclo de vida, Fragment implementa `LifecycleOwner` y expone un objeto `Lifecycle` al que puede acceder a través del método `getLifecycle()`.

Cada posible estado del ciclo de vida se representa con un `enum` `Lifecycle.State`.

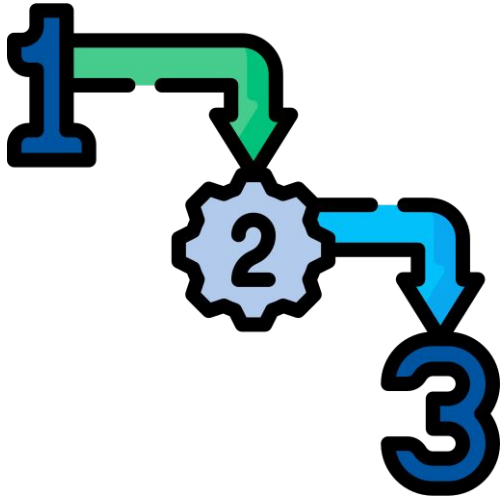
1. `INITIALIZED`
2. `CREATED`
3. `STARTED`
4. `RESUMED`
5. `DESTROYED`



# El ciclo de vida de los fragments

A medida que un fragmento avanza a través de su ciclo de vida, se mueve hacia arriba y hacia abajo a través de sus estados. Por ejemplo, un fragment que se agrega a la parte superior del back stack, se mueve hacia arriba desde **CREATED** hasta **STARTED** y luego **RESUMED**.

Por el contrario, cuando un fragment se extrae del back stack, se mueve hacia abajo a través de esos estados, pasando de **RESUMED** a **STARTED** a **CREATED** y finalmente **DESTROYED**.



# /* Los fragments y el método de factoría */

# Los Fragments y el método de factoría o FragmentFactory

La definición corta de **FragmentFactory** es: ***"una clase utilizada para controlar la creación de instancias de fragmentos."***

Antiguamente, solo se podía crear una instancia de Fragment utilizando su constructor vacío predeterminado. Esto se debe a que el sistema necesitaría reiniciarlo bajo ciertas circunstancias, como cambios de configuración y recreación del proceso de la aplicación. Si no fuera por la restricción del constructor predeterminado, el sistema no sabría cómo reiniciar la instancia de Fragment.

**FragmentFactory** se creó para evitar esta limitación, pues ayuda al sistema a crear una instancia de Fragment al proporcionar los argumentos/dependencias necesarios para instanciar el Fragment.

# ¿Cómo se usa FragmentFactory?

Extendiendo `FragmentFactory` y anulando `FragmentFactory#instantiate()`, luego asignándole a un FragmentManager.

Si un fragment tiene un constructor **no vacío**, deberá crear un `FragmentFactory` que se encargará de inicializarlo. Esto se hace extendiendo `FragmentFactory` y anulando su método `FragmentFactory#instantiate()`.

A continuación veremos un ejemplo real.

# Ejemplo de FragmentFactory

```
class CustomFragmentFactory(private val dependency: Dependency) : FragmentFactory() {
    override fun instantiate(classLoader: ClassLoader, className: String): Fragment {
        if (className == CustomFragment::class.java.name) {
            return CustomFragment(dependency)
        }
        return super.instantiate(classLoader, className)
    }
}

Luego si lo queremos usar en un Activity:
class HostActivity : AppCompatActivity() {
    private val customFragmentFactory = CustomFragmentFactory(Dependency())
    override fun onCreate(savedInstanceState: Bundle?) {
        supportFragmentManager.fragmentFactory = customFragmentFactory
        super.onCreate(savedInstanceState)
    }
}
```

# ¿Necesitamos usar FragmentFactory siempre?

La respuesta corta es NO.

Sin embargo, puede ser una opción de diseño.

Hasta ahora, probablemente hayas estado creando tus Fragments usando sus constructores predeterminados, o simplemente inicializándolos dentro del Fragment en algún momento antes de que se usen.

# ¿Necesitamos usar FragmentFactory siempre?

Si tu Fragment tiene un constructor **vacío** predeterminado, no es necesario usar FragmentFactory.

Sin embargo, si su Fragment toma argumentos en su constructor, se debe usar FragmentFactory; de lo contrario, se lanzará una Fragment.InstantiationException, ya que el FragmentFactory predeterminado que se usará no sabrá cómo instanciar una instancia de su Fragment.

# Ideas Fuerza

- Ten siempre presente el ciclo de vida de la app
- Recuerda, el usuario nota cuando haces uso correcto del ciclo de vida
- FragmentFactory puede ser un buen aliado, si embargo, recuerda siempre que es opcional usarlo

# ¿Qué contenido aprendido hoy crees que será relevante en tu práctica profesional futura?

# Próxima sesión...

- *Se continuará con los aspectos fundamentales del ciclo de vida de Activities y Fragments en Android para el desarrollo de aplicaciones*

{desafío}
latam_

Academia de
talentos digitales