CURSO DESARROLLO DE APLICACIONES MÓVILES ANDROID TRAINEE

Módulo 4.

Desarrollo de aplicaciones móviles Android Kotlin

AIEP OTEC EDUCACIÓN CONTINUA

Partner Laboratoria

TALENTO DIGITAL INTELIGENCIA HUMANA

Temario

Unidad 2

a) Threads

b) Kotlin Coroutines

c) Persistencia de Datos

d) Qué es Android Jetpack y Room

Unidad 2

Threads

Sincrónico y Asincrónico

- Conocer los conceptos de tareas y procesos síncronos
- Conocer los conceptos de tareas y procesos asíncronos
- Comprender modalidad background de un proceso
- Comprender modalidad main thread de un proceso

Sincrónico y Asincrónico

En este capítulo estudiaremos los conceptos de desarrollo de software asociados a la creación de tareas, procesos y acciones síncronas y asíncronas. También revisaremos los conceptos de un proceso ejecutado en background y de un proceso ejecutado en el main thread.

Sincrónico y Asincrónico

Los conceptos síncrono y asíncrono son muy simples de entender a nivel general como procesos de comunicación, y están absolutamente presentes en todos los sistemas informáticos de la actualidad, por lo cual, es importante estar claro en su significado y aplicación.

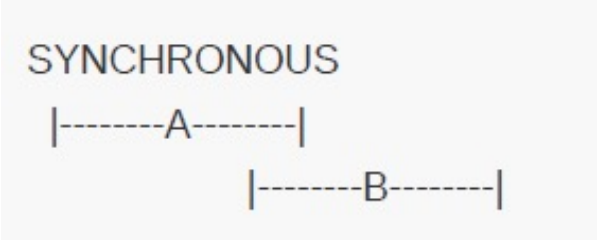
Mover numerosas o largas tareas desde el hilo principal, para que no interfieran con el renderizado suave y la rápida respuesta a la entrada del usuario, es la razón más importante para adoptar subprocesos en una aplicación.

Tareas, procesos sincronicos

En primer lugar, aclaremos que un proceso puede ser un sinónimo de una tarea, la diferencia estaría en que un proceso ocasionalmente requiere de más pasos para su cumplimiento que el de una tarea, entonces podemos entender ambos conceptos como similares, solo los separaría su dimensión en el trabajo que ejecutan, por lo tanto, de aquí en adelante, usaremos ambos términos para referirnos a la ejecución de un trabajo.

Tareas, procesos sincronicos

Un proceso síncrono en la programación de sistemas, se refiere a la conexión o dependencia sobre otra tarea o proceso, es decir, cuando se ejecuta un proceso síncrono se espera a que termine dicha ejecución antes de continuar con otro proceso.



La representación gráfica anterior, muestra un proceso A que se ejecuta y sólo cuando este finaliza, comienza la ejecución de un proceso B.

Tareas, procesos sincronicos

Si declaramos constantes numéricas que indique el tiempo de una tarea, por ejemplo:

```
const val TIME_TO_COOK_IN_MINUTES: Int = 3;
const val TIME_TO_EAT_IN_MINUTES: Int = 2;
```

Un método que muestre los segundos que se toman dichas tareas expresadas en minutos:

```
fun getSecondsSpendByTask(TIME_OF_TASK: Int, tarea: String){
  println(tarea)
  for(minutes in 0..TIME_OF_TASK) {
     var segundosTranscurridos = minutes * 60;
     println("$segundosTranscurridos segundos")
     threadRealTime()
  }
}
```

Tareas, procesos sincronicos

Otro método que haga la simulación del tiempo real con un thread, y que utilizamos en nuestra función anterior “getSecondsSpendByTask” de la siguiente forma:

```
fun threadRealTime(){
  try{
     Thread.sleep(60000)//60 segundos equivalentes a 1 minuto
  }catch (ie: InterruptedException){
     ie.printStackTrace()
  }
}
```

Tareas, procesos sincronicos

Y por último, un método que represente el ejemplo real, de una persona que necesita cocinar para comer, destacando el proceso como síncrono, ya que en esta situación necesitamos que se cocine, y una vez lista la comida, se coma, contando con nuestras limitaciones, en este caso el que una sola persona hace ambas labores.

```
fun totalSynchronousTasksTimes(){
  getSecondsSpendByTask(TIME_TO_COOK_IN_MINUTES, "COCINANDO")
  getSecondsSpendByTask(TIME_TO_EAT_IN_MINUTES, "COMIENDO")
}
```

Tareas, procesos sincronicos

Al invocar este último método iniciaremos un proceso síncrono que ejecuta dos tareas, la primera cocinar y la segunda comer, demostrando que solo si terminamos de cocinar se podrá comenzar a comer. Para probar los métodos anteriores podemos crear el método main, dentro de una clase Utils que los contenga de la siguiente forma:

```
fun main(){
  val utils = Utils()
  try {
     utils.totalSynchronousTasksTimes()
  }catch (e: Exception){e.printStackTrace()}
}
```

Tareas, procesos sincronicos

Nuestra salida por consola en el terminal sería la siguiente:

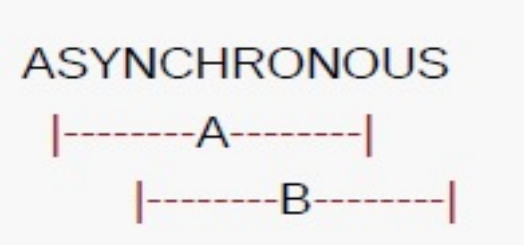
```
COCINANDO
0 segundos
60 segundos
120 segundos
COMIENDO
0 segundos
60 segundos
120 segundos

Process finished with exit code 0
```

Tareas, procesos sincronicos

Los procesos síncronos no deben ser considerados siempre como un problema, ya que las computadoras son super rápidas en su ejecución: sin embargo, si tu proceso es robusto, por ejemplo, recorre listas o mapas de datos grandes, hacer tu proceso síncrono no es la opción ya que tu sistema quedará en suspenso (bloqueado) hasta completar la tarea.

Tareas, procesos asincrónicos

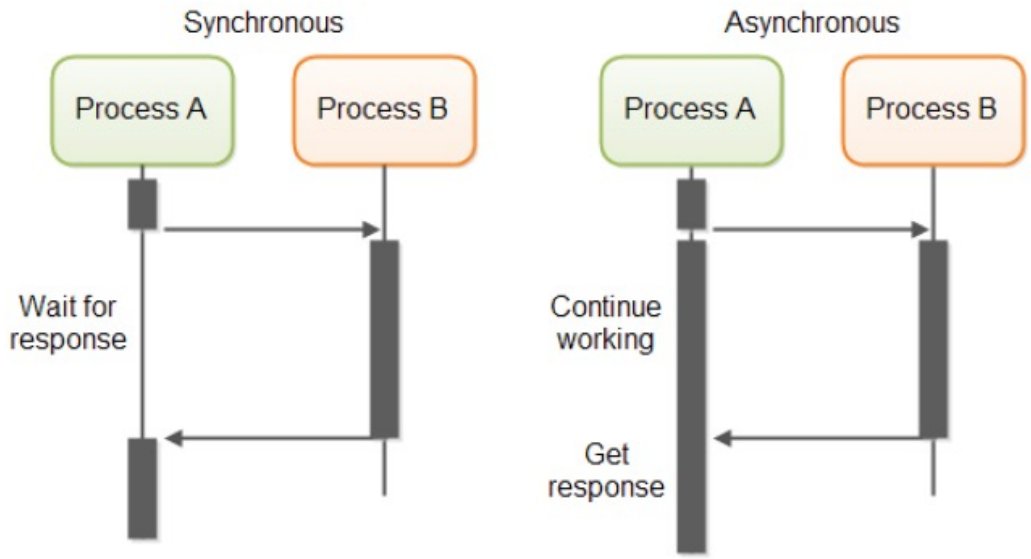
Una tarea o proceso asíncrono, en el modelo de programación de sistemas es aquel que se inicia, permitiendo continuar con otros procesos, para posteriormente recibir una respuesta; en otras palabras, cuando se ejecuta un proceso asíncrono, podemos iniciar otro proceso sin esperar a que el primero sea completado.



Tareas, procesos asincrónicos

Imaginemos que estamos en nuestra cocina, tenemos un plato en la mesa con un pan relleno de palta y deseamos iniciar un proceso asíncrono de comer, que nos permita hacer otras tareas, como por ejemplo, poner a cocer unas papas para el almuerzo, y echar a lavar nuestros pantalones en la lavadora. Estos tres procesos se pueden ejecutar sin esperar a que alguno de ellos termine, es decir, tomamos el pan, le aplicamos un mordisco, tomamos 4 papas y las colocamos en una olla de agua, aplicamos otro mordisco al pan, vamos por la ropa y la colocamos dentro de la lavadora, aplicamos otro mordisco al pan, ponemos la olla con las papas sobre el fuego, aplicamos otro mordisco al pan, echamos jabón en la lavadora… y así sucesivamente.

Tareas, procesos asincrónicos



Tareas, procesos asincrónicos

Para lograr constituir procesos asíncronos en android, disponemos de algunas técnicas para su correcto funcionamiento, las principales serían las siguientes:

- Threading
- Callbacks
- Promises
- Reactive Extensions (observable)
- Coroutines

Procesos en Background

Cada aplicación de android tiene un hilo principal (Thread UI) que se encarga de manejar la interfaz de usuario, coordinar las interacciones del usuario y recibir eventos del ciclo de vida de la aplicación. Si hay demasiado trabajo en este hilo principal, la aplicación tiende a bloquearse o ponerse lenta, lo que conlleva a una mala experiencia de usuario, que terminará en el abandono de tu aplicación por medio de las críticas.

Procesos en Background

Todas las operaciones de larga duración deben realizarse en un subproceso de fondo separado, lo que llamamos “background process”. En general, cualquier cosa que tome más de unos pocos milisegundos debe delegarse en un subproceso de fondo. Es posible que deba realizar algunas de estas tareas mientras el usuario interactúa activamente con la aplicación.

Procesos en Background

Las aplicaciones también pueden requerir que se ejecuten algunas tareas, incluso cuando el usuario no está utilizando la aplicación de forma activa, como sincronizar periódicamente con un servidor de fondo o buscar periódicamente contenido nuevo dentro de una aplicación.

Las aplicaciones también pueden requerir que los servicios se ejecuten inmediatamente hasta su finalización, incluso después de que el usuario haya terminado de interactuar con la aplicación.

Desafíos de los Procesos en Background

Las tareas en segundo plano consumen los recursos limitados de un dispositivo, como la RAM y la batería. Esto puede resultar en una mala experiencia para el usuario si no se maneja correctamente.

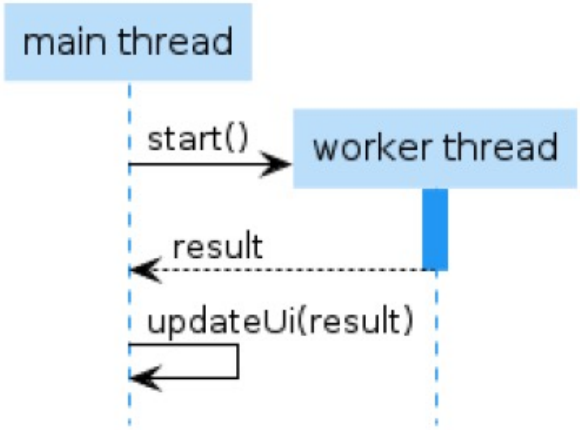
Para maximizar la batería y aplicar un buen comportamiento de la aplicación, Android restringe el trabajo en segundo plano cuando la aplicación (o una notificación de servicio en primer plano) no es visible para el usuario.

Desafíos de los Procesos en Background

Las tareas en segundo plano consumen los recursos limitados de un dispositivo, como la RAM y la batería. Esto puede resultar en una mala experiencia para el usuario si no se maneja correctamente.

Para maximizar la batería y aplicar un buen comportamiento de la aplicación, Android restringe el trabajo en segundo plano cuando la aplicación (o una notificación de servicio en primer plano) no es visible para el usuario.

Desafíos de los Procesos en Background



Desafíos de los Procesos en Background

Cuando el usuario inicia su aplicación, Android crea un nuevo proceso de Linux junto con un hilo de ejecución. Este hilo principal, también conocido como el hilo UI, es responsable de todo lo que sucede en la pantalla. Comprender cómo funciona, nos ayuda a diseñar nuestras aplicaciones de tal manera que el rendimiento sea el mejor posible.

Desafíos de los Procesos en Background

Mientras visualizamos una animación o una actualización de la pantalla en nuestra aplicación, el sistema intenta ejecutar un bloque de trabajo (responsable de dibujar la pantalla), con el fin de renderizar sin problemas a 60 fps (cuadros por segundo). Para que el sistema alcance este objetivo, la jerarquía de UI/Vista debe actualizarse en el hilo principal.

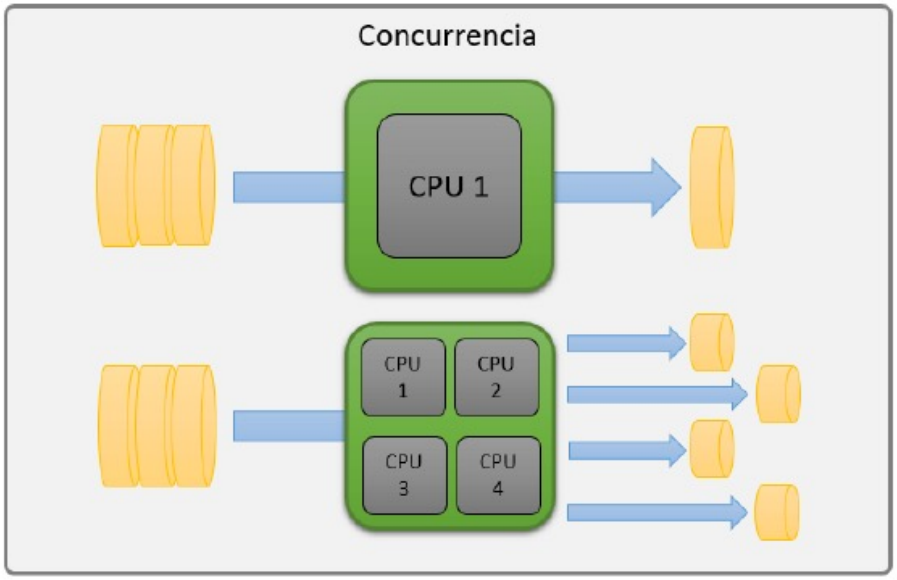
Concurrencia y paralelismo

- Conocer los conceptos de la concurrencia en la programación de sistemas.
- Conocer los conceptos de paralelismo en la concurrencia.

Concurrencia

La concurrencia es básicamente la capacidad de ejecutar múltiples tareas o procesos al mismo tiempo, pero no necesariamente simultáneos. En una aplicación concurrente tres procesos pueden iniciarse y completarse en períodos de tiempo distintos.

Concurrencia



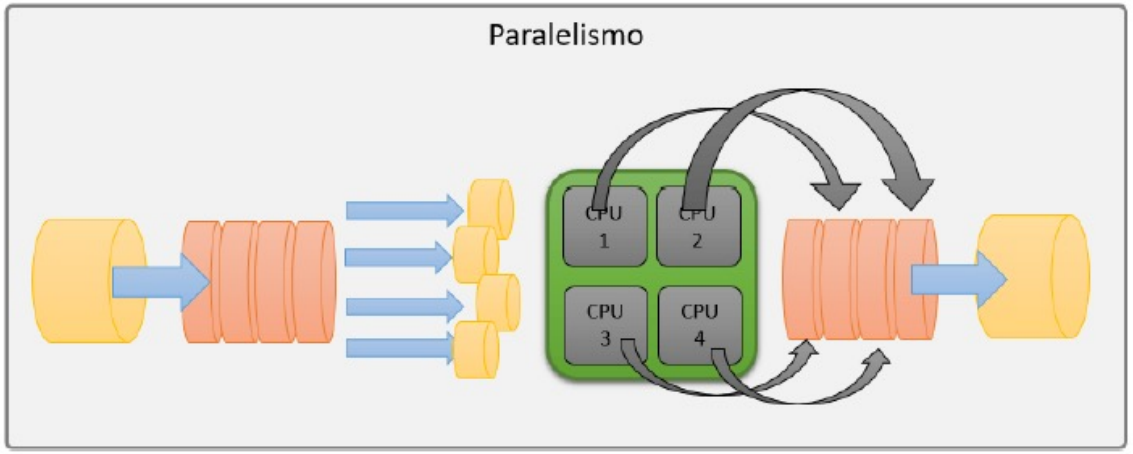
Context Switching

Un context-switching consiste en la ejecución de una rutina perteneciente al núcleo del sistema operativo multitarea de una computadora, cuyo propósito es parar la ejecución de un hilo o proceso para dar paso a la ejecución de otro distinto.

Paralelismo

El paralelismo es un tipo de concurrencia que se distingue principalmente por la capacidad que tiene el CPU para trabajar más de un proceso al mismo tiempo. Un procesador puede trabajar con una cantidad de procesos igual a su cantidad de cores, es decir, si un procesador tiene un solo core, entonces solo podrá ejecutar un proceso a la vez, por otro parte, si tenemos un procesador QUAD-CORE (4), entonces podremos ejecutar hasta cuatro procesos al mismo tiempo.

Paralelismo



Procesos

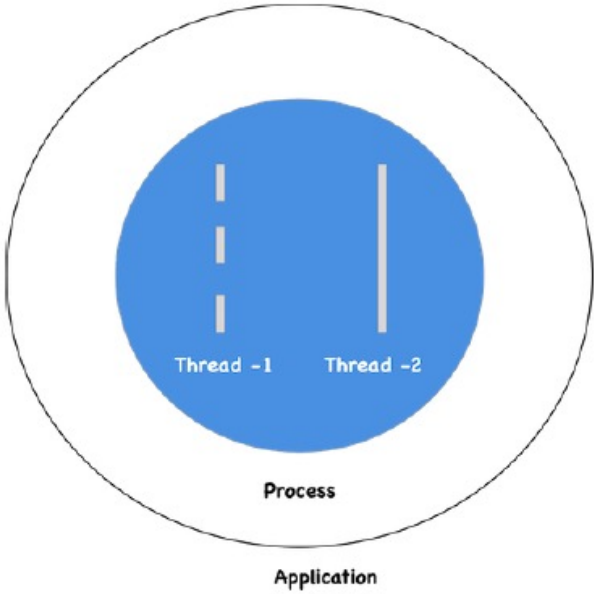
Un proceso es una instancia de un programa en ejecución. Un programa puede tener múltiples procesos. Un proceso generalmente comienza con un solo subproceso, en android lo conocemos como main thread UI, es decir, un subproceso primario, sin embargo, en el camino del proceso principal se pueden crear múltiples subprocesos.

Threads

Los hilos (threads o subproceso) son una secuencia de ejecución de código que puede ejecutarse independientemente uno del otro. **Es la unidad de tareas más pequeña que puede ejecutar un sistema operativo**. Un programa puede ser de un solo subproceso o de subprocesos múltiples.

Los threads o hilos son la manera más tradicional dentro del mundo de la programación para evitar el bloqueo de aplicaciones.

Threads



Main Threads Android

Al desarrollar aplicaciones de Android, siempre debemos ser conscientes de nuestro Main Thread.

El hilo principal está ocupado lidiando con cosas cotidianas, como dibujar nuestra interfaz de usuario, responder a las interacciones del usuario y, en general, de forma predeterminada, ejecutar en su mayoría el código que escribimos.

Threads sincrónicos y asincrónicos

Repasemos los conceptos de una tarea síncrona y asíncrona con el enfoque de los hilos para comprender su funcionamiento:

- Synchronous single thread:

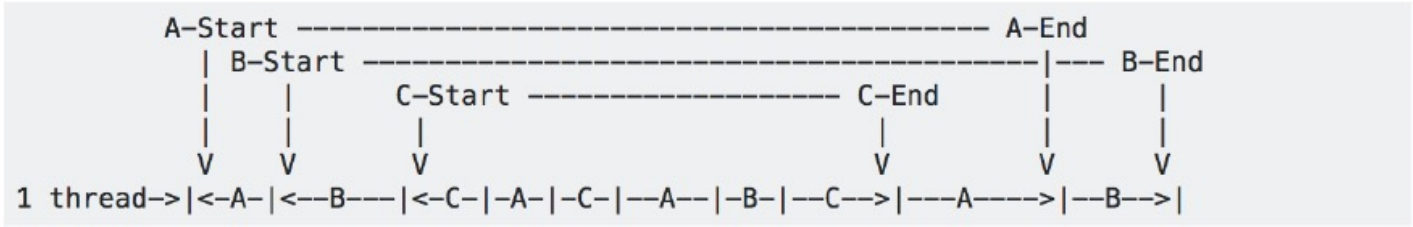
```
1 thread ->   |<---A---->||<----B---------->||<------C----->|
```

- Synchronous multi thread:

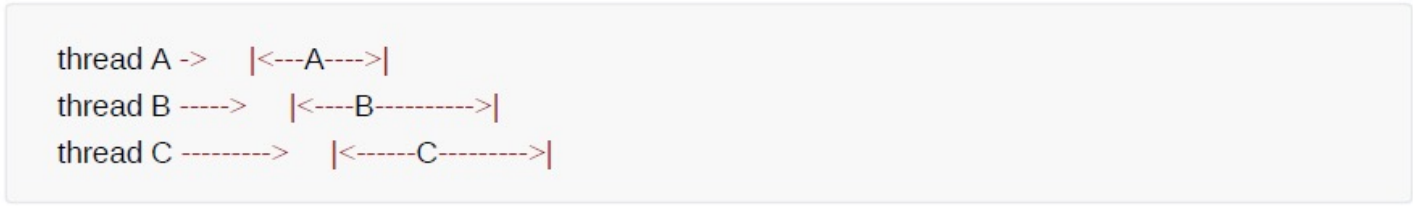
```
thread A -> |<---A---->|
                \
thread B ------------>   ->|<----B---------->|
                                \
thread C ---------------------------------->   ->|<------C----->|
```

Threads sincrónicos y asincrónicos

- Asynchronous single thread:



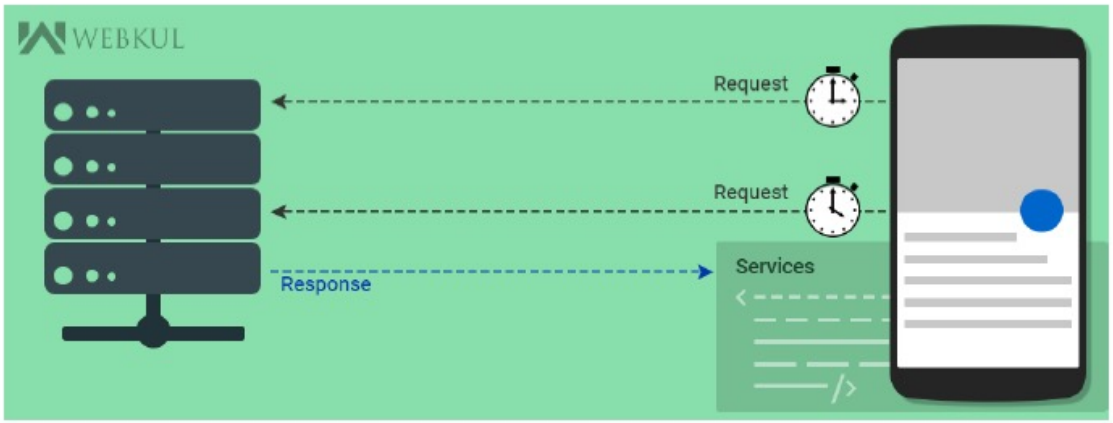
- Asynchronous multi thread:



Service

Un servicio es un componente de aplicación que puede realizar operaciones de larga ejecución en segundo plano y que no proporciona una interfaz de usuario, es decir, se ejecuta detrás de todo proceso visual de nuestra pantalla. Un componente de la aplicación, como por ejemplo una Actividad, puede iniciar un servicio y continuará ejecutándose en segundo plano aunque el usuario cambie a otra aplicación.

Service



Service

El Download Manager es un servicio del sistema que maneja descargas HTTP de larga duración.

Los clientes pueden solicitar que se descargue un URI a un archivo de destino particular. El Download Manager realizará la descarga en segundo plano, se ocupará de las interacciones HTTP y volverá a intentar las descargas después de fallas o en los cambios de conectividad y reinicios del sistema.

Service

Las aplicaciones que solicitan descargas a través de esta API deben registrar un receptor de transmisión para ACTION_NOTIFICATION_CLICKED y manejar adecuadamente cuando el usuario hace clic en una descarga en ejecución en una notificación o desde la IU de descargas.

Iniciar un Service

Un servicio está "iniciado" cuando un componente de aplicación (como una actividad) lo inicia llamando a startService(). Una vez iniciado, un servicio puede ejecutarse en segundo plano de manera indefinida, incluso si se destruye el componente que lo inició. Por lo general, un servicio iniciado realiza una sola operación y no devuelve un resultado al emisor. Por ejemplo, puede descargar o cargar un archivo a través de la red. Cuando la operación está terminada, el servicio debe detenerse por sí mismo.

Enlazar un Service

Un servicio es de “de enlace” cuando un componente de la aplicación se vincula a él llamando al método bindService(). Un servicio de enlace ofrece una interfaz cliente-servidor que permite que los componentes interactúen con el servicio, envien solicitudes, obtengan resultados e incluso lo hagan en distintos procesos con la comunicación entre procesos (IPC). Un servicio de enlace se ejecuta solamente mientras otro componente de aplicación está enlazado con él. Se pueden enlazar varios componentes con el servicio a la vez, pero cuando todos ellos se desenlazan, el servicio se destruye.

Utilizar un Service o subproceso

Un servicio es simplemente un componente que puede ejecutarse en segundo plano, incluso cuando el usuario no está interactuando con tu aplicación. Por lo tanto, debes crear un servicio solo si eso es lo que necesitas.

Si requerimos realizar un trabajo fuera del subproceso principal, pero solo mientras el usuario interactúa con la aplicación, es oportuno crear un subproceso, en lugar de un servicio, por ejemplo, si se desea reproducir música, pero sólo mientras se ejecute la actividad, podemos crear un subproceso en el método onCreate(), comenzar a ejecutarlo en onStart() y luego detenerlo en onStop().

Declaración de un Service en el manifiesto

Al igual que las actividades (y otros componentes), debes declarar todos los servicios en el archivo de manifiesto de tu aplicación.

Para declarar tu servicio, agrega un elemento como campo secundario del elemento de

```
<manifest ... >
  ...
  <application ... >
      <service android:name=".ExampleService" />
      ...
  </application>
</manifest>
```

Extencion de la clase Service

Dado que la mayoría de los servicios iniciados no necesitan manejar múltiples solicitudes simultáneamente (lo que, en realidad, puede ser un escenario peligroso de subprocesos múltiples), es mejor implementar un servicio utilizando la clase IntentService.

Extencion de la clase Service

La clase IntentService realiza las siguientes tareas:

- Crea un subproceso de trabajo predeterminado que ejecute todas las intents enviadas a onStartCommand(), independientemente del subproceso principal de tu aplicación.
- Crea una cola de trabajo que pase una intent a la vez a tu implementación onHandleIntent(), para no preocuparnos por varios subprocesos.
- Detiene el servicio una vez que se han manejado todas las solicitudes de inicio, para no tener que llamar al método stopSelf().

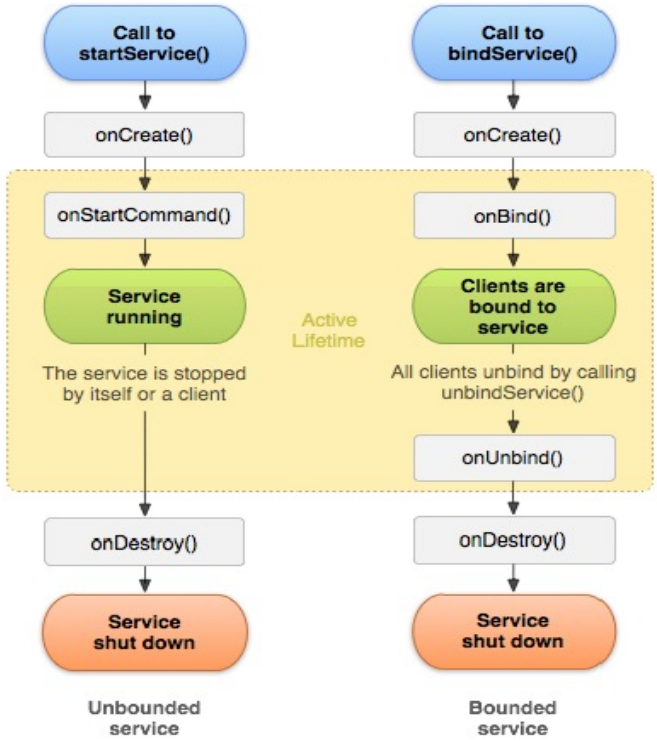
Extencion de la clase Service

- Proporciona la implementación predeterminada de onBind() que devuelve NULL.
- Proporciona una implementación predeterminada de onStartCommand() que envía el intent a la cola de trabajo y luego a la implementación del método onHandleIntent().

Implementación de Callbacks del ciclo de vida

Al igual que una actividad, un servicio tiene métodos callback del ciclo de vida que tú puedes implementar para controlar los cambios en el estado del servicio y realizar trabajos en los momentos adecuados.

Implementación de Callbacks del ciclo de vida



Alarm Manager

Esta clase proporciona acceso a los servicios de alarma del sistema operativo android. Estos servicios permiten programar tareas para que se ejecuten en algún momento del futuro. Cuando una alarma se desactiva, el Intent que se registró en su momento por un broadcast receiver, inicia automáticamente la aplicación contenedora de la tarea, si está no se está ejecutando.

Características del Alarm Manager

- Te permite disparar Intents en tiempos programados o intervalos de tiempo programados.
- Puedes usar AlarmManager junto con el componente BroadCastReceiver para iniciar servicios o ejecutar otras operaciones.
- Funcionan fuera de la aplicación, lo que te permite activar acciones o eventos fuera de tu aplicación sin necesidad de que esté funcionando.
- AlarmManager ayuda a minimizar los requerimientos de recursos en nuestras aplicaciones, sin necesidad de tener procesos background contínuamente escuchando o corriendo.

Características del Alarm Manager

Las alarmas repetidas son una buena opción para programar eventos regulares o búsquedas de datos. Una alarma repetitiva tiene las siguientes características:

- Un AlarmType.
- Un tiempo de activación, si el tiempo de activación que especifica está en el pasado, la alarma se activa de inmediato.
- El intervalo de la alarma, por ejemplo, una vez al día, cada hora, cada 5 minutos.
- Un Pending Intent que se dispara cuando se activa la alarma. Cuando configuramos una segunda alarma que usa el mismo PendingIntent, esta reemplaza la alarma original o inicial.

WorkManager

WorkManager es uno de los componentes de arquitectura de Android y parte de Android Jetpack, una nueva y obstinada forma de construir aplicaciones modernas de Android.

WorkManager es una biblioteca de Android que ejecuta trabajos en segundo plano diferibles cuando se cumplen las restricciones del trabajo .Está destinado a tareas que requieren garantizar que el sistema los ejecutará incluso si la aplicación sales.

Beneficios de utilizar WorkManager

- Maneja la compatibilidad con diferentes versiones del sistema operativo
- Sigue las mejores prácticas de salud del sistema
- Admite tareas asincrónicas únicas y periódicas.
- Admite tareas encadenadas con entrada / salida
- Le permite establecer restricciones cuando se ejecuta la tarea
- Garantiza la ejecución de tareas, incluso si la aplicación o el dispositivo se reinicia.

AsyncTask

La clase AsyncTask es ampliamente utilizada en muchas aplicaciones android desarrolladas en la actualidad, además de ser una clase con un continuo soporte y desarrollo por parte del equipo de android.

AsyncTask realiza operaciones en el subproceso en segundo plano (background thread) y se actualizará en el subproceso principal (thread UI). Asynctask nos ayuda a establecer comunicación entre hilos secundarios y principales (subThreads y mainThread/threadUI).

AsyncTask

AsyncTask permite el uso correcto y fácil del main thread UI de android. Esta clase nos ayuda a realizar operaciones en segundo plano y publicar resultados en el subproceso de la UI (interfaz de usuario) sin tener que manipular subprocesos, controladores o servicios.

AsyncTask está diseñado para ser una clase de ayuda en todo Thread y Handler, y no constituye un framework genérico para threads. Las AsyncTasks deben usarse idealmente para operaciones cortas, de unos pocos segundos.

AsyncTask

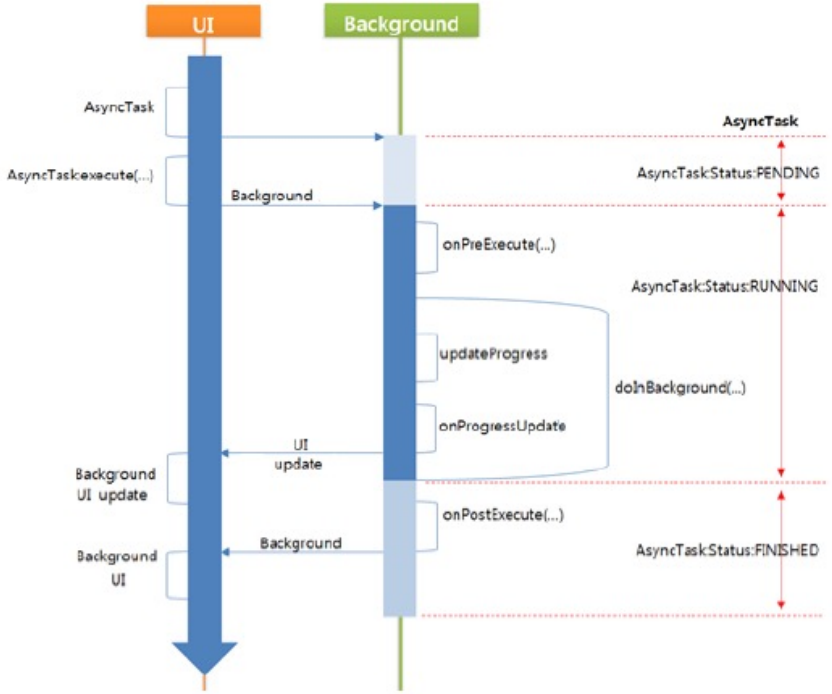


Diagrama proceso AsyncTask UI Thread y Background Thread.

AsyncTask

Una AsyncTask se define mediante tareas que se ejecuta en un subproceso en segundo plano, y cuyo resultado se publica en el subproceso de la interfaz de usuario. Un AsyncTask define tres métodos genéricos:

- doInBackground(), para todo aquel código que se ejecutará en segundo plano.
- onProgressUpdate(), que recibe toda actualización del progreso del método doInBackground.
- onPostExecute(), que lo utilizamos para actualizar la interfaz de usuario (UI) una vez el proceso en segundo plano se haya completado.

Iniciar un AsyncTask

Una clase AsyncTask se ejecutaría desde el mainThread a través de una actividad o fragmento de la siguiente forma:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Cancelar un AsyncTask

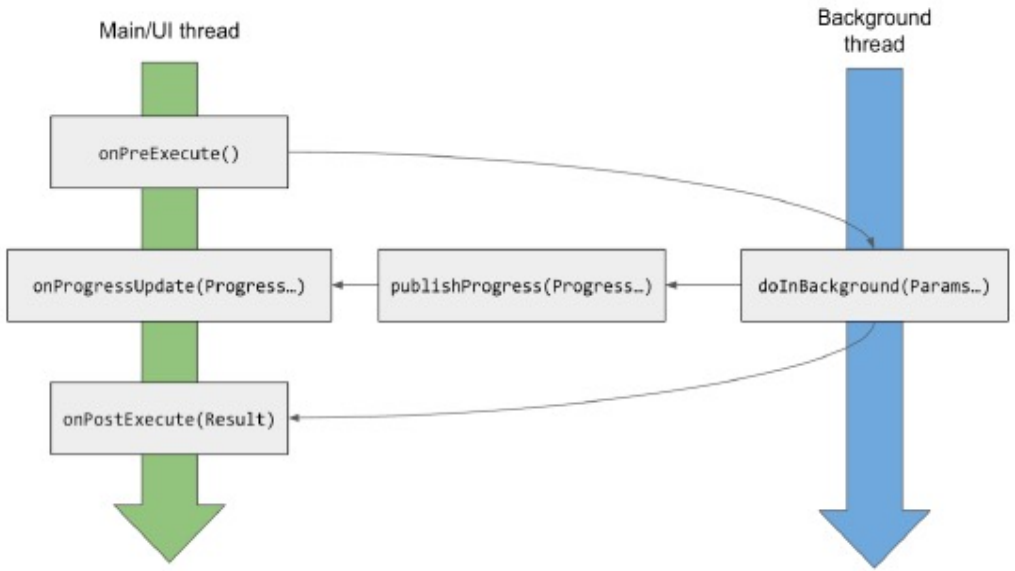
Una tarea puede cancelarse en cualquier momento invocando el método cancel(). Invocar este método hará que las siguientes llamadas (calls) invoquen al método isCancelled(), y este retornará el valor booleano true.

Una vez invocado el método cancel() la aplicación en vez de ejecutar el método onPostExecute() de AsyncTask ejecutará este método isCancelled(). Para asegurarnos de cancelar la tarea lo más rápido posible es oportuno validar el retorno de este método isCancelled() durante la ejecución del método doInBackGround().

El método onPreExecute de AsyncTask

Este método se puede invocar en el subproceso de la interfaz de usuario antes de ejecutar la tarea (UI Thread). Este paso se usa normalmente para configurar la tarea, por ejemplo, iniciando una barra de progreso en la interfaz de usuario.

El método onPreExecute de AsyncTask



Unidad 2

Kotlin Coroutines

Kotlin Coroutines

- Conocer el patrón moderno de desarrollo de promesas para tareas asíncronas.
- Aprender sobre las coroutines y sus conceptos modernos en android.
- Entender la asociación entre promesas y coroutines
- Implementar promesas y coroutines en un proyecto android.

Promises

Pensemos en una promesa como un objeto que espera a que finalice una acción asincrónica, luego llama a una segunda función. Podemos programar esa segunda función llamando al método .then() y pasando la nueva función. Cuando finaliza esta última función asincrónica, da su resultado a la promesa y la promesa lo da a la siguiente función (como parámetro).

Promises.All

A menudo queremos ejecutar acciones sólo después de que una colección de operaciones asincrónicas se haya completado con éxito. El objeto Promise.all devuelve una promesa que se resuelve si todas las promesas pasadas se resuelven. Si alguna de las promesas aprobadas rechaza, Promise.all rechaza con la razón de la primera promesa que rechazó. Esto es muy útil para garantizar que se complete un grupo de acciones asincrónicas antes de continuar con otro paso.

Coroutines

Una corrutina es un patrón de diseño de concurrencia que puede usarse en Android para simplificar el código que se ejecuta de forma asincrónica. Se agregaron corutinas a Kotlin en la versión 1.3 y se basan en conceptos establecidos de otros idiomas.

Las corrutinas podrían interpretarse como la última versión, la más avanzada y moderna de los callbacks y promesas para android, ya que en realidad las corrutinas, son en mucho el patrón async/await de javascript que lidera este avance en operaciones asíncronas.

Coroutines

En Android, las rutinas ayudan a resolver dos problemas principales:

- Administrar tareas de larga duración que de otro modo podrían bloquear el hilo principal y hacer que su aplicación se congele.
- Proporcionar seguridad principal, o llamadas con seguridad a las operaciones de red o disco desde el hilo principal.

Coroutines

Si una aplicación está asignando demasiado trabajo al hilo principal, la aplicación puede congelarse o detenerse. Las solicitudes de datos a través de grandes apis, el análisis de objetos JSON extensos, la lectura o la escritura desde una base de datos, o incluso la iteración de listas grandes, pueden hacer que la aplicación se ejecute lo suficientemente lento como para provocar que nuestra pantalla responda lentamente a eventos táctiles. Estas operaciones de larga duración deben ejecutarse fuera del hilo principal.

Coroutines

El siguiente ejemplo muestra la implementación de rutinas simples para una tarea hipotética de larga duración:

```
suspend fun fetchDocs() {                    // Dispatchers.Main
    val result = get("https://developer.android.com") // Dispatchers.IO for `get`
    show(result)                             // Dispatchers.Main
}

suspend fun get(url: String) = withContext(Dispatchers.IO) { /* ... */ }
```

Coroutines

Las corrutinas se basan en funciones regulares agregando dos operaciones para manejar tareas de larga duración. Además de invoke(o call) y return, las corrutinas agregan suspend y resume:

- **suspend**: hace pausa en la ejecución de la rutina actual, guardando todas las variables locales.
- **resume:** continúa la ejecución de una corrutina suspendida desde el lugar donde fue suspendida.

Seguridad y Coroutines

Las corrutinas de Kotlin usan despachadores para determinar qué hilos se usan para la ejecución de la rutina. Para ejecutar el código fuera del hilo principal, puede indicarle a las corrutinas de Kotlin que realicen el trabajo en un despachador predeterminado. En Kotlin, todas las corrutinas deben ejecutarse en un despachador, incluso cuando se ejecutan en el hilo principal. Las corrutinas pueden suspenderse, y el despachador es responsable de reanudarlas.

Seguridad y Coroutines

Para especificar dónde deben ejecutarse las corrutinas, Kotlin proporciona tres despachadores que puede usar:

- **Dispatchers.Main**: este despachador se usa para ejecutar una rutina en el hilo principal de Android. Este debe usarse solo para interactuar con la interfaz de usuario y realizar un trabajo rápido.

Seguridad y Coroutines

- **Dispatchers.IO**: este despachador está optimizado para realizar operaciones de disco o de red fuera del hilo principal.
- **Dispatchers.Default**: este despachador está optimizado para realizar un trabajo intensivo de CPU fuera del hilo principal. Un ejemplo podría ser ordenar una lista y analizar un objeto JSON.

Iniciando Coroutines

Podemos comenzar las corrutinas de una de dos maneras:

- **launch**: comienza una nueva corrutina y no devuelve el resultado al objeto que llama. Cualquier trabajo que se considere iniciarse y no esperar por su respuesta es ideal indicar la palabra launch.
- **async**: inicia una nueva rutina y le permite devolver un resultado con una función de suspensión llamada await.

Iniciando Coroutines

Por lo general, ejecutamos launch para iniciar una nueva corrutina de una función regular, ya que una función regular no puede llamar await. Para usar async debemos hacerlo sólo cuando se esté dentro de otra corrutina o cuando se esté dentro de una función de suspensión.

```
fun onDocsNeeded() {
    viewModelScope.launch {    // Dispatchers.Main
        fetchDocs()            // Dispatchers.Main (suspend function call)
    }
}
```

Uso de Coroutines en Android

Para utilizar las corrutinas en nuestros proyectos android es necesario implementar en el gradle de la app las librerías que nos permiten tener acceso a estas. Por ejemplo el siguiente código de versión 1.0.1:

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.1'
```

Uso de Coroutines en Android

Para utilizar las corrutinas en nuestros proyectos android es necesario implementar en el gradle de la app las librerías que nos permiten tener acceso a estas. Por ejemplo el siguiente código de versión 1.0.1:

Unidad 2

Persistencia de Datos

- Entender las diferentes opciones para almacenar datos en Android.
- Decidir cuál es la mejor opción para otorgar persistencia de datos en las Aplicaciones.
- Saber qué opciones ofrece Android para compartir información entre aplicaciones.

Persistencia de Datos

El sistema operativo Android nos ofrece diferentes opciones para almacenar los datos que utiliza nuestra aplicación. Las opciones disponibles se utilizan en diferentes casos de uso, normalmente las opciones por defecto de una aplicación es más que suficiente para la mayoría de los casos. Las opciones por defecto de Android para almacenar información o datos en una aplicación son SharedPreferences o el Sandbox de la aplicación.

Persistencia de Datos

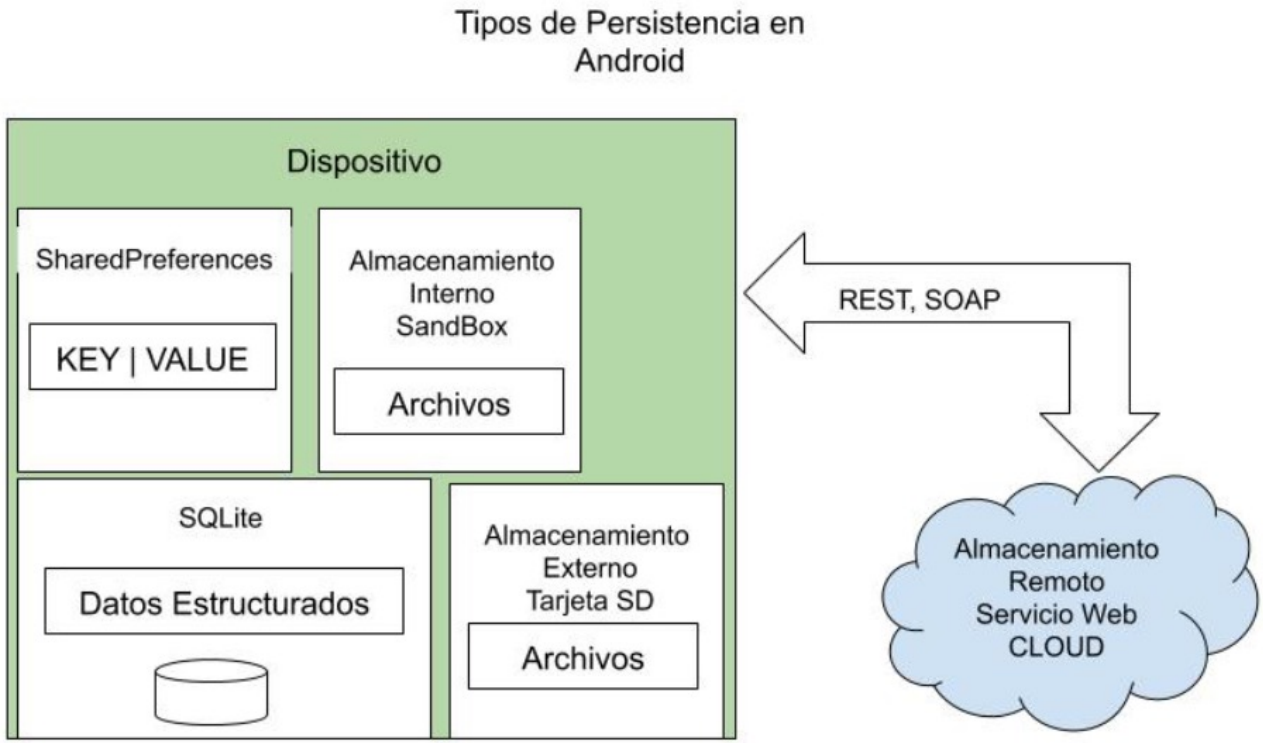
En Android, una aplicación cuenta con un ecosistema que le permite almacenar datos que son utilizados por nuestra aplicación. Cada aplicación cuenta con un Sandbox al cual sólo tiene acceso la aplicación, y el sistema operativo en modo root. Este Sandbox es el contenedor de ejecución seguro de cada aplicación, donde se almacena el código fuente, los assets, archivos de configuración, imágenes, etc.

Persistencia de Datos

No confundir contenedor de ejecución de la aplicación con ART, que es Android Run Time. ART es donde se ejecuta el código en Android, pero el contenedor de cada aplicación es el Sandbox. Nadie tiene acceso a este contenedor o sandbox, el cual se ejecuta con el mínimo de privilegios necesarios en el dispositivo.

Este concepto de Sandbox es gracias a linux, que crea un id para cada proceso que corre en el sistema operativo, con el mínimo de privilegios y sin intervención externa.

Persistencia de Datos



Persistencia de Datos

Las diferentes opciones de persistencia de datos son las siguientes:

1. SharedPreferences: Nos permite almacenar datos primitivos en formato clave-valor. Este se profundizará más adelante.

2. Almacenamiento Interno: Nos permite almacenar información en la memoria interna del dispositivo. La información se puede almacenar en archivos dentro del Sandbox de la aplicación. Sólo la aplicación tiene acceso a esos archivos, a menos que el dispositivo se encuentre rooteado.

Persistencia de Datos

3. Almacenamiento externo: Los dispositivos Android permiten conectar memoria externa, normalmente en formato de memoria SD. Esta memoria es extraíble o se puede montar en un computador, por lo cual no se asegura su disponibilidad, y se recomienda almacenar información que no sea crítica para el funcionamiento de la aplicación. Se pueden almacenar archivos, los cuales son de acceso público para todos los usuarios. Cualquiera de estos usuarios puede eliminar o modificar dichos archivos. Este tipo de almacenamiento no tiene seguridad.

Persistencia de Datos

4. Base de Datos SQLite: Almacenamiento de datos, estructurado, privado. Esta implementación de SQL está optimizada para dispositivos móviles, es la opción por defecto de base de datos en Android. Desde los inicios se ha podido crear bases de dato en SQLite. La librería Room de Google nos permite interactuar con SQLite de manera más simple, con menos errores y menos boilerplate code, facilitando el uso y el desarrollo. Es decir, Room crea una capa de abstracción que nos permite interactuar directamente con la base de datos en SQLite, pero con la familiaridad de trabajar con objetos en vez de tablas y queries SQL. Room será visto en profundidad más adelante.

Persistencia de Datos

4. Base de Datos SQLite: Almacenamiento de datos, estructurado, privado. Esta implementación de SQL está optimizada para dispositivos móviles, es la opción por defecto de base de datos en Android. Desde los inicios se ha podido crear bases de dato en SQLite. La librería Room de Google nos permite interactuar con SQLite de manera más simple, con menos errores y menos boilerplate code, facilitando el uso y el desarrollo. Es decir, Room crea una capa de abstracción que nos permite interactuar directamente con la base de datos en SQLite, pero con la familiaridad de trabajar con objetos en vez de tablas y queries SQL. Room será visto en profundidad más adelante.

Persistencia de Datos

5. Almacenamiento remoto: Podemos almacenar información de modo remoto, ya sea a través de un servicio web, normalmente REST, o a través de un servicio tipo Cloud.

Datos y Seguridad

¿Cómo podemos decidir qué tipo de almacenamiento nos conviene utilizar?. Para decidir esto debemos tener claro:

¿Qué tipo de información o datos estamos almacenando?.

¿Los datos son públicos o privados?

¿Qué disponibilidad debe tener esa información o datos?

¿Quién tiene acceso a esa información o datos ?

Almacenamiento de fotografías de la camara

Por ejemplo la aplicación de la cámara necesita almacenar archivos de imagen de gran tamaño, en este caso podemos utilizar el almacenamiento interno o externo. Las imágenes que se toman con la cámara del dispositivo, con la aplicación por defecto de la cámara, son de acceso público y se encuentran disponibles en la galería del dispositivo, o los archivos multimedia del mismo.

Almacenamiento de fotografías de la camara

Estos mismos archivos son utilizados por aplicaciones para ser publicadas en internet, estas aplicaciones deben pedir permiso para acceder a las imágenes, pero nada impide que esos mismos archivos sean extraídos en un computador personal, incluso eliminados de la memoria del dispositivo, sea interna o externa.

Preferencias de una aplicación

Consideremos el caso donde nuestra aplicación permite a un usuario configurar a su gusto algunas cosas, por ejemplo el tamaño de la letra o los colores de ciertos componentes. Como esta configuración es local, no representa problemas de privacidad, y no tiene mucha complejidad, podemos utilizar SharedPreferences para almacenar un conjunto de clave-valor y mantener los datos de la configuración entre instancias de nuestra aplicación. Si estos datos se llegan a perder, nuestra aplicación no sufre ninguna consecuencia severa, salvo pequeños inconvenientes para el usuario.

Aplicación Bancaria

Imaginemos una aplicación bancaria, en la cual tenemos acceso a diferentes servicios, por ejemplo revisar nuestra cartola mensual en formato pdf. Esta cartola contiene información sensible para el usuario, y debe ser almacenada con la mayor seguridad posible, por lo que en este caso es necesario utilizar el almacenamiento interno, en el sandbox de la aplicación, para que nadie, salvo el usuario en nuestra aplicación, pueda ver el archivo descargado. Medidas adicionales son bienvenidas en casos de extrema sensibilidad de información, algunos son: claves de acceso, encriptación o cifrado de los archivos.

Aplicación con modo online/offline

Lo más común actualmente es contar con aplicaciones que utilizan Internet para obtener información, pero muchas veces se ven afectadas por la intermitencia de las conexiones de los teléfonos celulares. En estos casos podemos desarrollar una versión Offline, normalmente con Caché o data almacenada localmente, que podemos mostrar al usuario mientras no tenemos conexión.

Aplicación con modo online/offline

Este caché, normalmente, se almacena en una base de datos local, y se va renovando con la última información, a medida que vamos cargando desde el servicio Online. Hay aplicaciones que para mejorar la experiencia del usuario, almacenan grandes cantidades de caché, lo cual termina resintiendo el desempeño del dispositivo por falta de espacio. Tener cuidado al implementar este tipo de almacenamiento en una aplicación.

Aplicación con modo online/offline

Este caché, normalmente, se almacena en una base de datos local, y se va renovando con la última información, a medida que vamos cargando desde el servicio Online. Hay aplicaciones que para mejorar la experiencia del usuario, almacenan grandes cantidades de caché, lo cual termina resintiendo el desempeño del dispositivo por falta de espacio. Tener cuidado al implementar este tipo de almacenamiento en una aplicación.

Compartiendo datos en Android

Como ya sabemos, los datos o información, en Android, sólo están disponibles para la aplicación. En el caso que necesitemos compartir información entre aplicaciones contamos con las siguientes opciones:

1. Compartir a través de SharedPreferences: Podemos compartir nuestro archivo de SharedPreferences con otras aplicaciones a través del modo global.

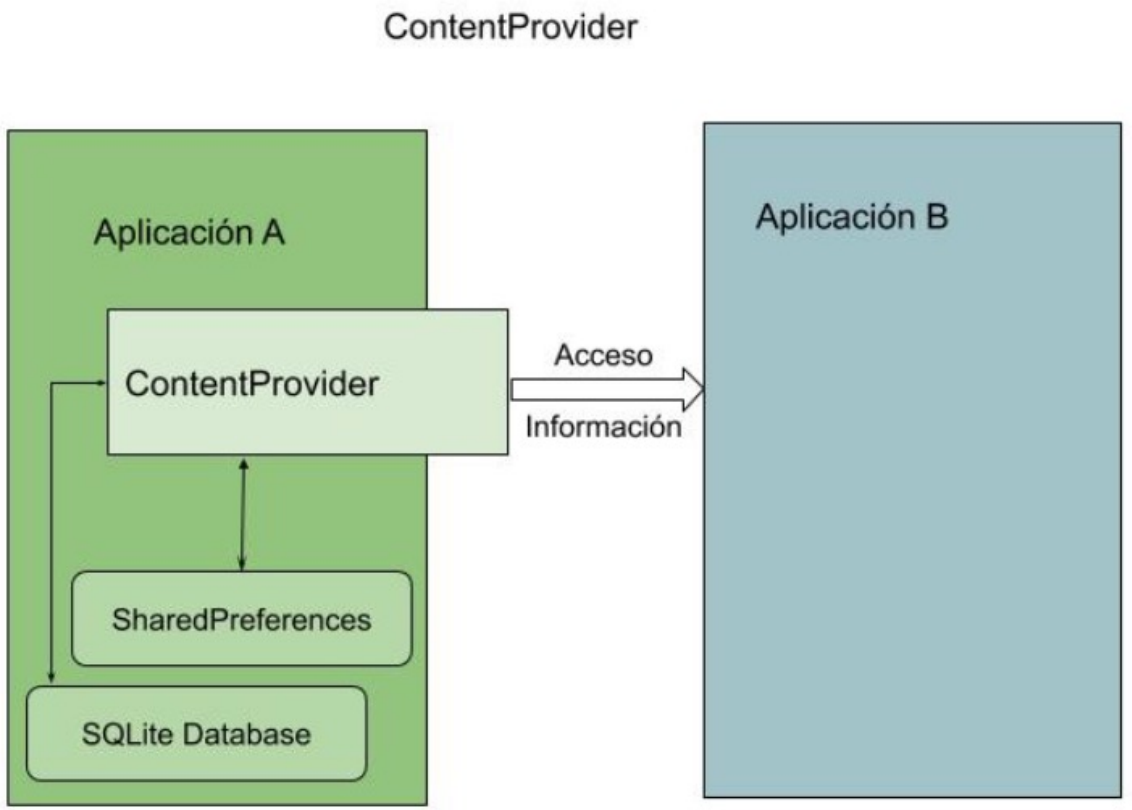
Compartiendo datos en Android

2. ContentProvider: los proveedores de contenido nos permiten acceder a datos de otra aplicación. Estos datos son compartidos para que sean usados por otras aplicaciones, incluso pueden ser modificados si el ContentProvider lo permite. Se pueden compartir imágenes, archivos, o datos de una base de datos. Si queremos compartir información desde una aplicación A a una aplicación B, la aplicación A debe implementar el ContentProvider.

Compartiendo datos en Android

La aplicación B debe utilizar el ContentProvider implementado por A y pedir los permisos necesarios. Por ejemplo WhatsApp debe utilizar el ContentProvider de la aplicación de Contactos de el sistema, así obtiene los contactos necesarios para empezar a enviar mensajes o llamar por teléfono.

Compartiendo datos en Android



SharedPreferences o archivo de preferencias

SharedPreferences es la forma más simple de almacenar información en Android. Nos permite almacenar pequeñas cantidades de información, en un formato de clave-valor, en un archivo en nuestro dispositivo.

Este archivo se puede crear con diferentes configuraciones que limitan el acceso de terceros a la información almacenada en este archivo.

SharedPreferences o archivo de preferencias

El archivo es directamente manejado por Android, todos los componentes de la aplicación tienen acceso al archivo, pero no está disponible para otras aplicaciones. El archivo de sharedpreferences se inicia de manera recomendada en modo privado, indicando que sólo los componentes de la aplicación dueña del archivo tienen acceso al mismo.

SharedPreferences o archivo de preferencias

- Existe un modo público para este archivo, pero no está recomendado, y se han deprecado sus opciones desde hace varias versiones de Android, más información sobre esto en la sección donde explicamos como crear un archivo de preferencias.
- SharedPreferences está recomendado sólo para pequeñas cantidades de información, si la información que se debe almacenar es más compleja que unas cuantas primitivas, por ejemplo un objeto, se recomienda utilizar una base de datos SQLite.

SharedPreferences vs Preference

- SharedPreferences es un conjunto de APIs que nos permite almacenar datos primitivos (String, Int, Boolean ,etc), en formato clave-valor, estos datos persisten entre instancias de la aplicación.
- Preference es un conjunto de APIs que nos permiten crear una interfaz de usuario para una o varias pantallas de preferencias, por ejemplo la configuración de idioma o notificaciones push de la aplicación. Preference ocupa SharedPreferences para almacenar y persistir estos datos que son seleccionados en la pantalla de preferencias.

Métodos de SharedPreferences

La interfaz nos permite no sólo editar, con la interfaz Editor, también nos permite leer y realizar otras operaciones sobre el archivo de preferencias. Las otras operaciones que podemos realizar son las siguientes:

- contains: método que recibe una clave y nos devuelve un boolean indicando si esta clave está presente o no en el archivo de preferencias. Sólo nos dice si el archivo contiene la clave, no el valor.
- edit: es el método que nos devuelve el Editor que describimos en el apartado anterior.

Métodos de SharedPreferences

- getAll: este método nos entrega un Map con todas las claves y valores presentes en el archivo de preferencias. Podemos iterar sobre este Map y recorrer todas la claves del archivo, pero debemos saber específicamente qué tipo de valor está relacionado con cada clave, para poder operar correctamente con ese valor.

Métodos de SharedPreferences

- getBoolean: nos permite leer un boolean desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor boolean de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo boolean, el método entrega una ClassCastException.

Métodos de SharedPreferences

- getFloat: nos permite leer un Float desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor float de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Float, el método entrega una ClassCastException.

Métodos de SharedPreferences

- getInt: nos permite leer un Int desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor int de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Int, el método entrega una ClassCastException.

Métodos de SharedPreferences

- getLong: nos permite leer un Long desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor long de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Long, el método entrega una ClassCastException.

Métodos de SharedPreferences

- getString: nos permite leer un String desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace una búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el valor String de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Int, el método entrega una ClassCastException.

Métodos de SharedPreferences

- getStringSet: nos permite leer un Set de Strings desde el archivo de preferencias. Recibe dos parámetros una clave y un valor por defecto. Con la clave se hace un búsqueda en el archivo de preferencias, si no existe la clave se crea y se devuelve el valor por defecto que le pasamos como segundo parámetro. Si la clave existe, se devuelve el Set con los valores de esa clave y se ignora el valor por defecto que entregamos como parámetro. Si la clave existe, pero no es de tipo Set, el método entrega una ClassCastException.

Unidad 2

Android Jetpack y Room

AIEP OTEC EDUCACIÓN CONTINUA

Partner Laboratoria

TALENTO DIGITAL INTELIGENCIA HUMANA

¿Por qué Room?

Room nos permite crear una capa de abstracción sobre SQLite para facilitar el acceso a los datos, y la interacción con dicha Base de Datos. Una aplicación que maneja grandes cantidades de data estructurada se benefician directamente con la implementación de room.

Room como ORM

Un Object Relational Mapper, ORM, es una herramienta que genera todo este código intermedio, boilerplate code, que une o pega nuestra aplicación con su base de datos.

Los ORM no son invención de Android, son ocupados hace mucho tiempo, y tenemos a nuestra disposición muchas opciones. La cantidad de opciones generó mucha fragmentación y limitaciones.

Room como ORM

Además, las aplicaciones tienen cada vez ciclos de vida más complejos, lo que hace difícil tomar una decisión de cuál elegir. Si elegimos de mala manera, y la arquitectura de nuestra aplicación no es la adecuada, cambiar de ORM puede ser un dolor de cabeza mayor.

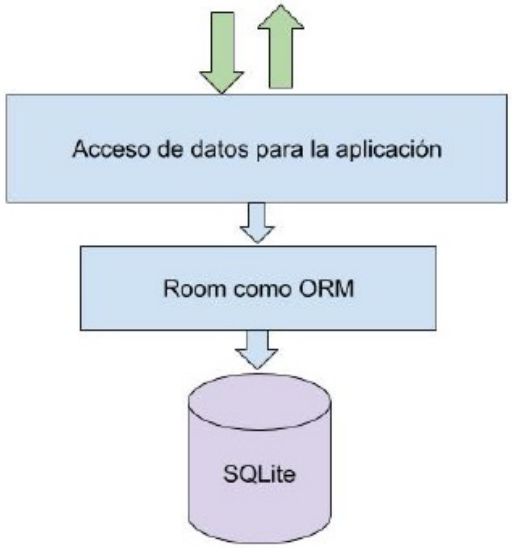
Room como ORM

Como respuesta a la fragmentación, errores comunes, y como parte de su cruzada por mejores aplicaciones con mejor arquitectura, Google crea Room. Room es parte de Jetpack, y se recomienda como uno de los componentes de una arquitectura por capas tipo MVVM o MVP. Room es parte de la capa de datos, que se recomienda implementar como un repositorio.

Room como ORM

Conceptualmente podemos entender Room como el mediador entre la Base de datos SQLite y la capa de acceso de datos de la aplicación, pudiendo esta ser implementada como repositorio u otro tipo de patrón arquitectónico.

Room como ORM



Componentes de Room

Para utilizar Room debemos generar sus tres componentes principales:

- Entity: Representa un data model, o modelo de datos, que se mapea directamente a las tablas de la base de datos.
- DAO: Data Access Object, este objeto nos permite interactuar con la base de datos. Normalmente la interacción es del tipo CRUD (Create, Read, Update and Delete)
- Database: Un contenedor que mantiene una referencia y actúa como único punto de acceso a la base de datos.

Entity

Es el primer componente que debemos definir para nuestra aplicación. Un Entity, es un objeto que nos ayuda a mantener y modelar datos dentro de nuestra aplicación. Estos datos pueden estar almacenados o ser manejados en memoria. Un Entity modela con un objeto el tipo de datos que luego almacenaremos en la base de datos. Además, a este "modelo" le agregamos información para describir cómo construir la tabla en la base de datos.

Entity

Para nuestro ejemplo definimos un set de datos y una tabla con columnas, considerando lo siguiente:

- Almacenaremos un texto con la recomendación
- Almacenaremos un id para mantener ordenados nuestros datos, este id será numérico y auto incrementado por la base de datos.
- Nuestra base de datos sólo tendrá una tabla, con dos columnas, por lo que hemos definido anteriormente.

Entity

Nuestro Entity está definido en una data class que contiene la información que nuestra aplicación necesita y almacenará en la base de datos. Entrando en el detalle de esta clase, tenemos los siguientes componentes:

- Entity: es una anotación de Room. Nos permite definir que la data class anotada es un Entity. Dentro de la anotación definimos que Entity almacenará la información en una tabla llamada “recommendations_list”. Está tabla tiene dos columnas. Para señalar que es una columna ocupamos la anotación que sigue.

Entity

- ColumnInfo.ColumnInfo: Esta anotación nos permite definir que la variable anotada es unacolumna de la tabla antes nombrada. Dentro de la anotación definimos el nombre de la columna en la tabla. En este caso tenemos dos variables anotadas como columnas:

I. id: es el identificador de la fila en la tabla, definimos un valor inicial de 0 para el id.

II. recommendation_text: la columna que contiene texto de la recomendación.

LiveData

El patrón de diseño observador

El Patrón observador está calificado como un patrón de comportamiento, este tipo de patrón se encarga principalmente en la comunicación entre clases y objetos en un programa. El patrón observer está recomendado para cuando necesitamos reaccionar a los cambios que ocurren a un objeto, pero sin la necesidad de que estemos constantemente preguntando por el estado de ese objeto. El objeto se encarga de avisar cuando ocurre un cambio en su estado, notificando a todos los que están observando esos cambios.

El patrón de diseño observador

El comportamiento ideal que debería tener el programa en este tipo de casos es:

1. La interfaz reacciona a los cambios, no pregunta por ellos
2. El objeto que cambia debe avisar a los interesados cada vez que cambia
3. Cada vez que los interesados son notificados de los cambios, se procede a actualizar la interfaz que esperaba esos cambios.
4. La relación entre un Sujeto y sus observadores es 1 - N (uno a muchos).

El patrón de diseño observador

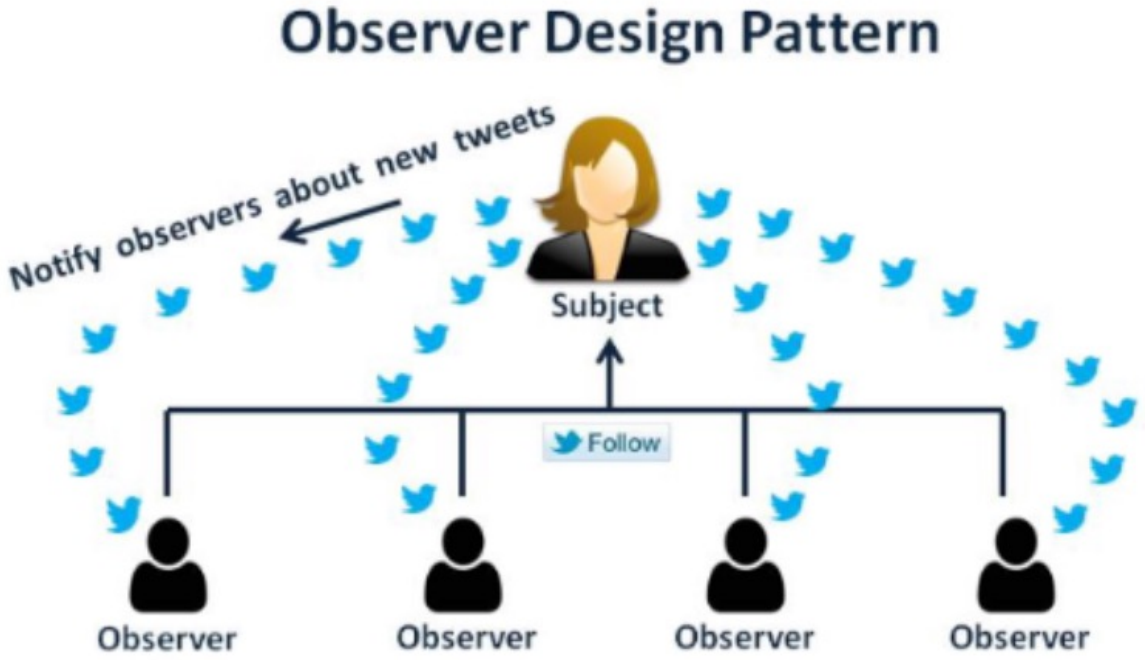
Para obtener este comportamiento se define en el patrón observador los siguientes elementos:

- Subject: Es considerado el que mantiene la información, los datos o la lógica de negocios del programa. Es la fuente de la “verdad”.
- Observer: Es un interesado en los datos, y quiere ser notificado cuando estos cambien. Es el que reacciona ante el evento gatillado.
- Register/Attach: Los observers se registran en la lista de observadores del Subject, para ser notificados cuando ocurra un cambio.

El patrón de diseño observador

- Event: Los eventos gatillan cambios en el Subject, todos los observers serán notificados de esos cambios.
- Notify: Dependiendo de qué tipo de implementación se haya realizado del patrón el Subject puede empujar la nueva información al observer, o el observer puede rescatar la nueva información desde el Subject, este proceso dependiendo del tipo de implementación será transparente para nosotros.
- Update: Los observers actualizan sus estados de manera independiente y sin verse afectado por otros observers.

El patrón de diseño observador



Ciclo de vida de una actividad y fragmento

Este ciclo de vida está definido en el framework de Android desde sus inicios. Este ciclo cuenta con 6 métodos principales, que son llamados a medida que la Actividad avanza en su ciclo de vida, los métodos son:

- onCreate: Este método se llama cuando la actividad es creada por primera vez. En este método debemos unir la interfaz, el layout xml a la actividad. Debemos configurar todas las variables iniciales, valores, y otras cosas que queremos que el usuario vea cuando la aplicación es desplegada. Debemos evitar realizar tareas grandes en este método, ya que entre más tiempo lleve esa tarea, más se demora en aparecer en pantalla la interfaz de la actividad.

Ciclo de vida de una actividad y fragmento

- conStart: Este método se llama cuando la actividad se encuentra en estado started, ya es visible para el usuario, pero este no puede interactuar con ella. Normalmente en este método podemos gatillar tareas asíncronas, que no interfieren con el ciclo de la aplicación, y pueden cargar información cuando terminen sin bloquear la interfaz de usuario.
- onResume: Este método nos señala que la aplicación se encuentra en estado resumed, lo que básicamente significa que la actividad es visible, está ocupando la pantalla y el usuario puede interactuar con ella. En este estado es cuando normalmente se ejecutan las diferentes funciones interactivas de la aplicación, por ejemplo poder hacer like a fotos o escribir un correo.

Ciclo de vida de una actividad y fragmento

- onPaused: este método se llama cuando la aplicación pierde el foco principal y entra en estado paused. Básicamente el usuario está dejando la actividad actual. Esto ocurre cuando gatillamos otra actividad, aparece un dialog o recibimos una llamada de teléfono.
- onStop: este método es llamado cuando la actividad entró en estado stopped. Esto significa que ya no es visible para el usuario, y que podría ser destruida por el sistema. La actividad ya no se encuentra en la pantalla del dispositivo.

Ciclo de vida de una actividad y fragmento

- onDestroy: este método es el último que se llama en el ciclo de vida de una actividad. Este método se llama justo antes que la actividad sea terminada por el sistema operativo.

Ciclo de vida de una actividad y fragmento

- onDestroy: este método es el último que se llama en el ciclo de vida de una actividad. Este método se llama justo antes que la actividad sea terminada por el sistema operativo.

Este es a grandes rasgos el ciclo de vida de una Actividad, si bien estos métodos son los principales, debe ser claro que se cuenta con muchos más métodos y otra consideraciones cuando trabajamos con actividades en Android.

Ciclo de vida de una actividad y fragmento

onDestrTambién es importante entender que los Fragments también poseen un ciclo de vida que está altamente ligado al ciclo de vida de la actividad que lo soporta.

Los estados y métodos más relevantes en el ciclo de vida de los fragmentos son los siguientes:

- onAttach():
- onCreate():
- onCreateView():
- onPause():

Manejo manual de los ciclos de vida

Como desarrolladores de aplicaciones Android debemos estar preocupados de que nuestras interfaces gráficas o UI mantengan sus datos lo más actualizado posibles o que se adapten a los cambios del ciclo de vida de la actividad o fragmento que se está mostrando, de forma que no pierdan información cuando ocurre un cambio en la configuración, el cual puede ocurrir simplemente rotando el dispositivo.

Manejo manual de los ciclos de vida

Existen varias alternativas para solventar estos problemas como por ejemplo:

- Manejar los datos en los componentes de UI (Objeto Dios)
- Usar Listeners
- Usar alguna alternativa de “eventBus”.

LifeCicle-Aware components

Gracias a los nuevos componentes de arquitectura ya no tenemos que preocuparnos del manejo de los ciclos de vida como hace algunos años, debido a que elementos como LifeCycle classes se crearon para que los componentes puedan manejar el ciclo de vida en el cual están inmersos y sobrevivir a los cambios de configuración sin tener que realizar tareas extras por nuestra parte.

LifeCycle y LifeCycleOwner

Como mencionamos, Android cuenta con componentes que son Lifecycle aware, lo cual significa que estos son capaces de responder o reaccionar a cambios en el ciclo de vida de otros componentes como Actividades o Fragmentos.

Ahora explicaremos los elementos que están presentes en estos componentes y que permiten que estos tengan esa clase de comportamiento.

LifeCycle

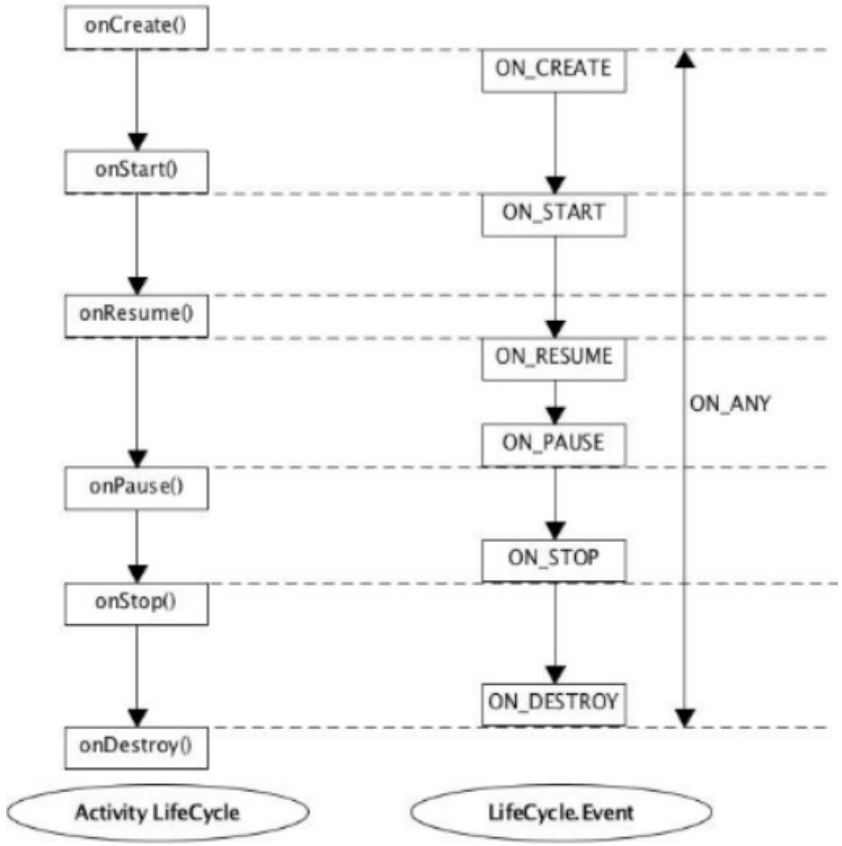
Es una clase que mantiene información acerca del estado del ciclo de vida de un componente, una actividad o un fragmento. Esta clase ocupa dos enumeraciones principales para mantener un registro de estado del ciclo de vida del componente asociado a la clase.

- Event: Los eventos del ciclo de vida que son despachados desde el framework de Android y la clase Lifecycle. Estos eventos mapean los callbacks del ciclo de vida de las actividades y los fragments.
- State: El estado actual del componente que es trackeado por el objeto Lifecycle.

LifeCycle

En el siguiente diagrama podemos ver una representación gráfica del ciclo de vida completo de un componente y los diferentes estados del mismo. Observa como los estados en el lifecycle se producen inmediatamente después de los métodos de la Actividad hasta onResume, pero luego se ejecutan de forma anterior a los métodos desde onPause en adelante hasta el final del ciclo de vida de la actividad.

LifeCycle



LifecycleOwner y LifecycleObserver

Un Lifecycle owner es cualquier componente que implementa la interfaz LifecycleOwner, esta interfaz indica que tiene un ciclo de vida en Android. Los Fragmentos y la Actividades implementan esta interfaz desde la biblioteca de soporte 26.1.0.

Se pueden crear LifecycleOwners custom, implementando la interfaz LifecycleRegistry que le permite a el componente custom manejar múltiples observers.

Un LifecycleObserver es un componente que observa los diferentes estados asociados a un LifecycleOwner, reaccionando a los diferentes cambios.

Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistre

- LifeCycle:

1. Lifecycle.Event: Enum que enumera los distintos estados del ciclo de vida ON_CREATE, ON_START, ON_RESUME, ON_PAUSED, ON_STOP, ON_DESTROY y un valor especial que sirve para comparar con cualquier evento ON_ANY.
2. Lifecycle.State: enum que enumera los diferentes estados que tiene el LifecycleOwner asociado. En este caso tenemos lo siguiente

Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistre

- CREATED: estado después de onCreate y justo antes de onStop
- DESTROYED: Estado final, justo antes de onDestroy, no se despachan más eventos después de este estado.
- INITIALIZED: Es el estado cuando la actividad ha sido construida, pero no se ha llamado a su método onCreate.

Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistre

- RESUMED: estado después del llamado a onResume: estado después de onStart y justo antes de onPause
- Tiene 3 métodos asociados: isAtLeast, valueOf y values. El primero nos dice si el estado actual es al menos el valor que le pasamos como parámetro.

Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistre

- addObsLifecycleOwner: es una interfaz que tiene un sólo método, este método, getLifecycle(), nos entrega el Lifecycle del LifecycleOwner. De esta forma clases lifecycle aware pueden observar los cambios en ese componente.
- LifecycleObserver: interfaz que marca una clase como Observador del ciclo de vida de otro componente.
- OnLifecycleEvent: esta anotación nos dice que evento está observando el método anotado. Por ejemplo @OnLifecycleEvent(Lifecycle.Event.ON_CREATE) nos señala que el método anotado reaccionará al evento ON_CREATE.

Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistre

- LifecycleRegistry: Esta clase es una implementación de Lifecycle que permite manejar múltiples observadores. Se utiliza para implementar los custom LifecycleOwners, cuenta con los siguientes componentes:

  - Constructor que recibe un LifecycleOwner que provee el ciclo de vida
  - addObserver: para agregar observers, recibe un LifecycleObserver.
  - getCurrentState: retorna el estado actual del ciclo de vida
  - getObserverCount: retorna el conteo de observadores de este registro.

Componentes Lifecycle, LifecycleOwner, LifecycleObserver, LifecycleRegistre

- handleLifecycleEvent: establece el estado actual del ciclo de vida, recibe un Lifecycle.Event. Notifica a los observers.
- markState: recibe un Lifecycle.State, se mueve a ese estado y despacha todos los eventos necesarios a los observadores.
- removeObserver: recibe un LifecycleObserver, lo remueve de la lista de observadores de este componente.

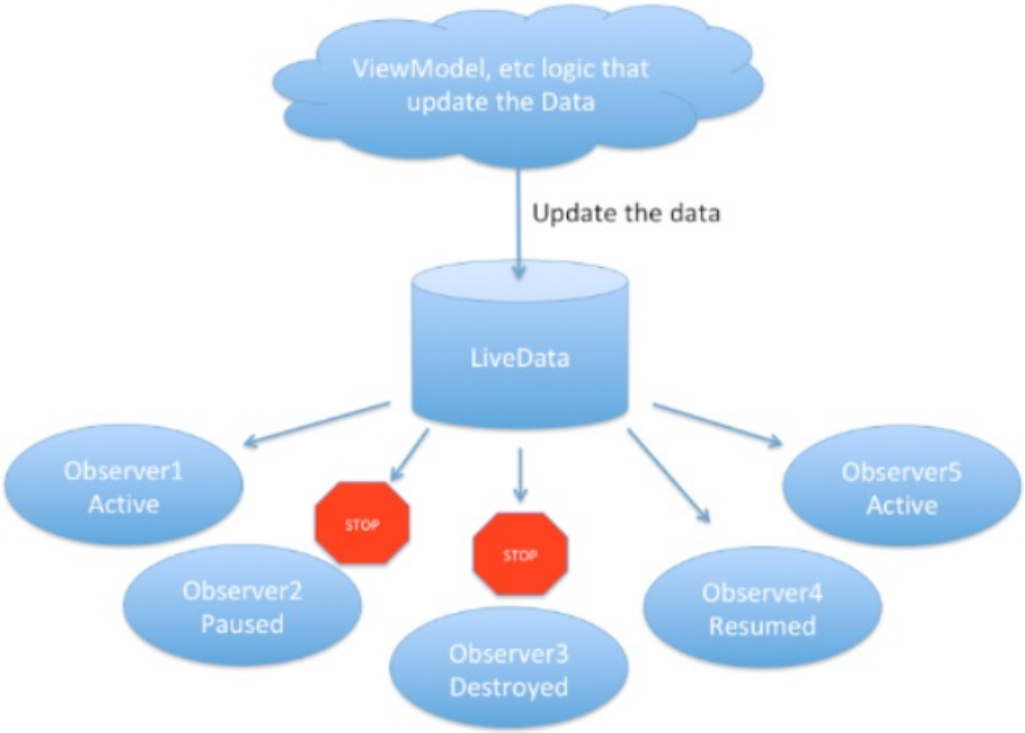
LiveData

Según la documentación oficial LiveData es una clase contenedora de datos observables. Además indica que LiveData al contrario de otros observables es consciente de los ciclos de vida, lo que quiere decir que respeta los ciclos de vida de otros componentes como por ejemplo Actividades, Fragmentos o servicios.

LiveData

Según la documentación oficial LiveData es una clase contenedora de datos observables. Además indica que LiveData al contrario de otros observables es consciente de los ciclos de vida, lo que quiere decir que respeta los ciclos de vida de otros componentes como por ejemplo Actividades, Fragmentos o servicios.

LiveData



Ventajas de usar LiveData

- Asegura que la Interfaz de Usuario, UI, esté siempre actualizada: Al seguir el patrón observador LiveData notifica los cambios a sus observadores, en este caso la interfaz gráfica. El observador será el encargado de actualizar la interfaz gráfica cuando ocurran cambios.
- No más memory leaks: Esto es una de las mejores ventajas, ya no debemos preocuparnos de limpiar los observadores para evitar memory leaks cuando están referenciados, pero no se usan más. Recordando que un objeto referenciado, pero que no se usa, no puede ser reciclado por el garbage collector.

Ventajas de usar LiveData

- No más caídas debido a actividades que son detenidas: Si el ciclo de vida del observador está inactivo, los eventos no son notificados a ese observador, el cual puede ser null y generar un NullPointerException, muy común en otras implementaciones del patrón observer, cómo RxJava, si no se limpian los observadores.
- No hay necesidad de manejar los ciclos de vida a mano: Los componentes gráficos sólo deben observar los datos relevantes, no es necesario inscribir y reinscribir los observadores al pausar y resumir los ciclos de vida.

Ventajas de usar LiveData

- Datos siempre actualizados: si un ciclo de vida pasa a estar inactivo, recibirá el último valor de los datos cuando vuelva a estar activo. Por ejemplo si tenemos una actividad en segundo plano, cuando vuelva a ser mostrada en primer plano, será actualizada con el último valor de los datos disponibles.
- Manejo de cambios de configuración: si ocurre algún cambio de configuración en el dispositivo, se rota la pantalla por ejemplo, la actividad o fragment afectado por ese cambio, recibirá inmediatamente el valor más actual de los datos.

Utilizar LiveData Objects

Existen tres pasos para utilizar objetos de LiveData:

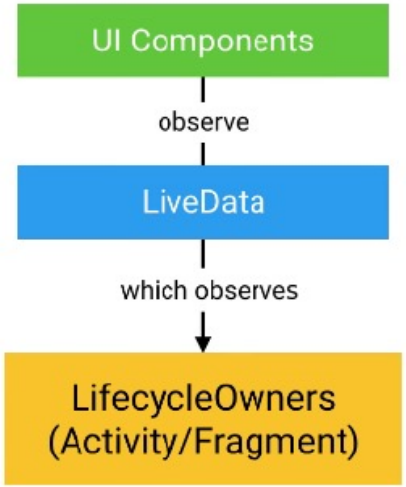
1. Crea una instancia de LiveData para mantener algún tipo de dato, regularmente esto se realizará al interior de una clase ViewModel.

2. Crea un objeto Observador que defina cuando se ejecuta un método onChange(), el cual controla que ocurre con el objeto LiveData cuando se produce un cambio. Regularmente el objeto observador lo crearemos en algún elemento de UI, ya sea actividad o fragmento.

Utilizar LiveData Objects

3. Luego debes unir al objeto observador con el de LiveData usando el método observe(), Esto realiza la subscripción del objeto observador al objeto Livedata para que los cambios puedan ser notificados. Regularmente el objeto observador será atado a una actividad o fragmento.

Utilizar LiveData Objects



Crear objetos LiveData

Como LiveData es un envoltorio de datos, lo podemos utilizar con cualquier tipo de objeto por ejemplo colecciones, listas etc. Los objetos LiveData regularmente son almacenados al interior de un objeto ViewModel y se puede acceder a ellos a través de métodos de getter.

```
class NameViewModel : ViewModel(){
  //Create a LiveData with a String
  val currentName: MutableLiveData<String> by lazy{
    MutableLiveData<String>
  }
  // Rest of the ViewModel...
}
```

Observar objetos LiveData

Según la documentación oficial el método más idóneo para comenzar a observar un objeto es en el método onCreate(). Los motivos son los siguientes.

1. Para asegurarse que el sistema no realice llamadas redundantes desde un actividad o fragmento al momento de ejecutarse el método onResume()

2. Para asegurar que la actividad fragmento obtenga la data y la pueda mostrar tan pronto como esté activa. Es decir tan pronto como los componentes de la app están en STARTED, puedan recibir la actualización de los datos.

Actualizando objetos LiveData (update)

Usualmente MutableLiveData es usado en el ViewModel y el ViewModel solo expone objetos inmutables LiveData a los observadores. Para esto se utiliza la buena práctica de exponer estos objetos con un getter hacia la vista, esto ayudará a limitar la modificación de ese objeto. Esto ocurre de esta forma porque LiveData no tiene implementado los métodos setValue y postValue.

Actualizando objetos LiveData (update)

Entonces si necesitáramos que se modifiquen, es mejor añadir una función que lo actualice. por ejemplo:

```
public LiveData<User> getUser(){
    return mUserInfoLiveData;
}
public void updateUser(User userUpdated){
    mUserInfoLiveData.postValue(userUpdated);
}
```

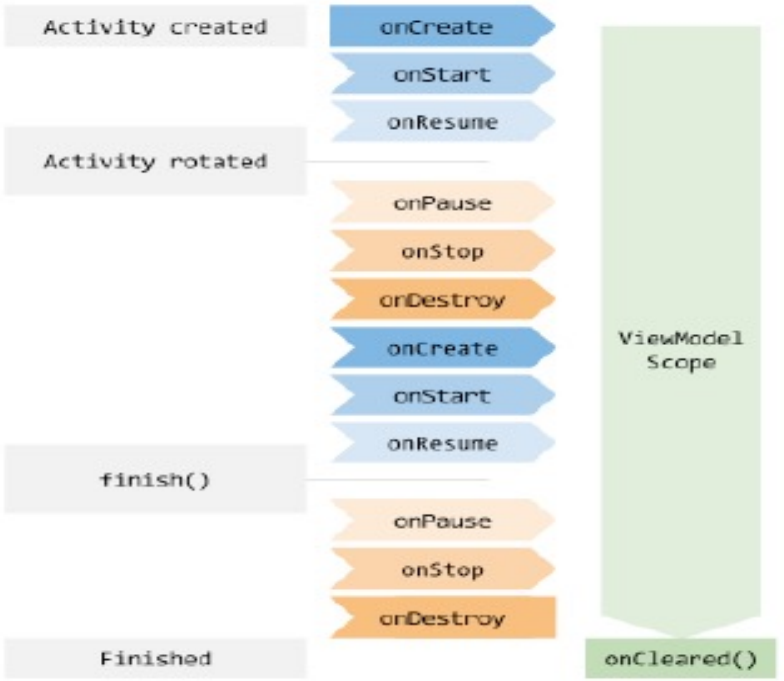
LiveData y Room

Afortunadamente la librería de persistencia de datos Room soporta consultas observables, las cuales pueden retornar objetos de LiveData. Estas consultas se escriben directamente en los DAO(Database access object). Room generará todo el código necesario para actualizar el objeto LiveData cuando la base de datos sea actualizada. Estás consultas funcionan de forma asíncrona en un background thread cuando sean necesarias, esto permite que nuestra UI se mantenga sincronizado con los cambios en la base de datos.

ViewModel

ViewModel también es un nuevo elemento presente en los componentes de arquitectura, se define como una clase llamada ViewModel, la cual es la responsable de preparar la data para ser mostrada en la UI o vista, esta clase está optimizada para respetar los ciclos de vida. Esto quiere decir que no importa que se produzca alguna rotación del dispositivo o cambio de configuración, los datos estarán disponibles para la actividad o fragmento objetivo.

ViewModel


Activity created
onCreate
onStart
onResume
Activity rotated
onPause
onStop
onDestroy
onCreate
onStart
onResume
finish()
onPause
onStop
onDestroy
Finished
ViewModel Scope
onCleared()