

Background jobs

1TEL05 - Servicios y Aplicaciones para IoT

Hilos (thread)

Process



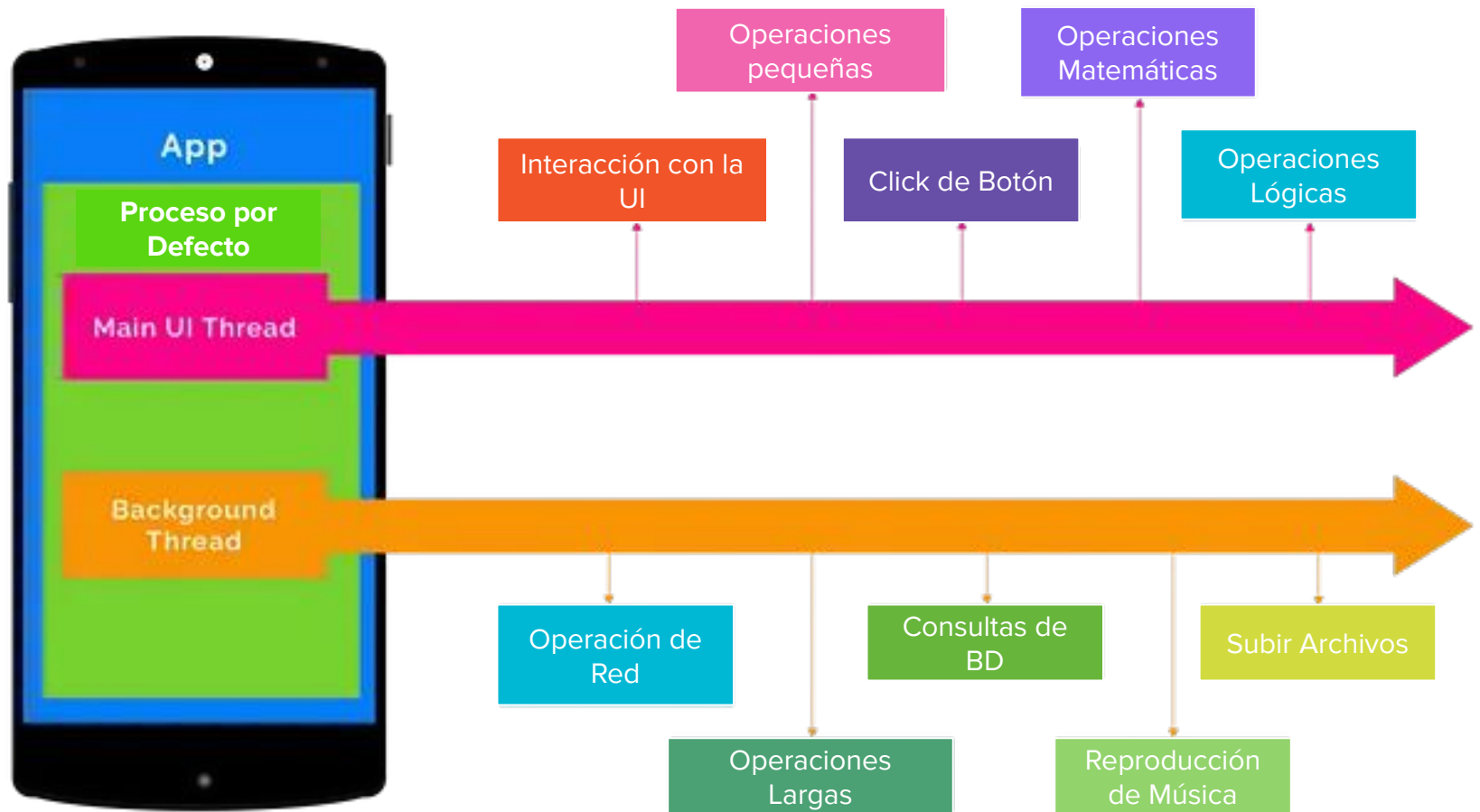
Vs

Threads



**Real life example
in Animated way**

Hilos



Hilo principal - UI thread

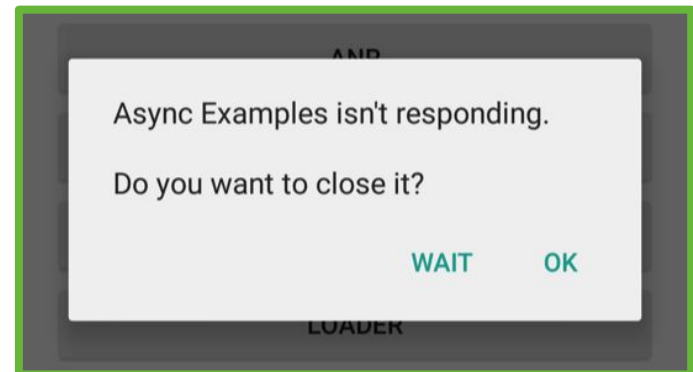
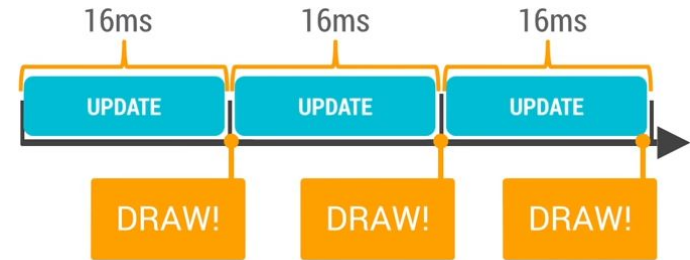
- La aplicación se ejecuta en un hilo de Java llamado "main" o "UI thread"
- Encargado de dibujar la interfaz de usuario (UI) en la pantalla
- Responde a las acciones del usuario manejando eventos de la interfaz.
- Se tienen dos reglas para el **UI thread**
 - *No bloquear el hilo con acciones que demoren mucho tiempo*
 - *Gestionar solo elementos en la **UI thread***

El hilo principal debe ser rápido $\leq 16\text{ms}$

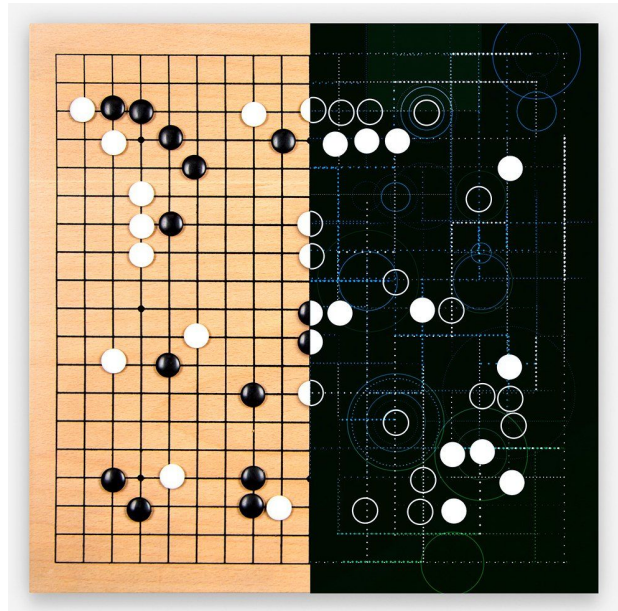
- Android refresca la pantalla cada 16ms (para dibujar 60fps [frames/second])
- Si toma más de 16ms, se considera un **frame drop**, causando “lag” o una interfaz lenta.
- Por ejemplo:

Obtener información de un servidor remoto puede **demorar 1 segundo**; si esa información se obtiene en el UI thread, son **60 frames que se pierden**.

Si la aplicación pierde 300 frames (5 segundos), aparecerá el diálogo ANR (Application Not Responding)



Procesos que toman mucho tiempo

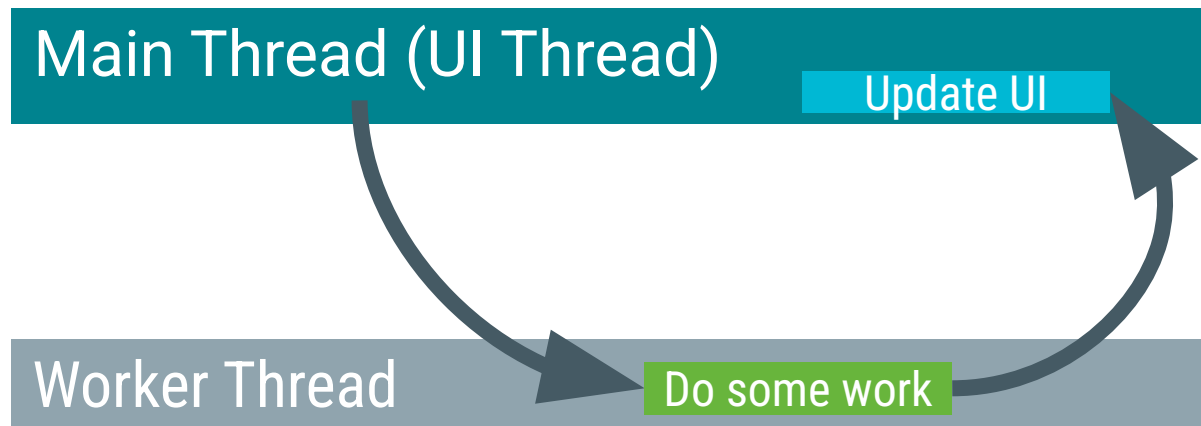


- Operaciones de red (GET, POST, etc).
- Cálculos matemáticos muy complejos.
- Descargar o subir archivos
- Procesamiento de imágenes

Hilos en background

Las operaciones que consumen mucho tiempo o recursos, deben correr fuera del hilo UI. A estos hilos se les llaman: **background thread**. Existen varias formas:

- Trabajo asíncrono:
 - AsyncTask (Obsoleto desde Android 30)
 - Loader Framework (Obsoleto desde Android 28)
 - **ViewModel + Java concurrency**
- Trabajo persistente:
 - **Workers**



Asynchronous work vs Persistent work

Ambas son tareas que se realizan en “**background**”; sin embargo:

Asynchronous work	Persistent work
Sucede en un momento de tiempo	Puede suceder en varios instantes de tiempo. Permite schedule.
No persiste al reiniciar la app o el dispositivo	Puede persistir al reiniciar la app y el dispositivo

Trabajo asíncrono

- Java concurrency

Java concurrency

- Java permite crear hilos adicionales o “background threads” para manejar tareas inmediatas, de respuesta “corta” y que deseen actualizar la UI.
- Existen librerías como Guava o RxJava que permiten manejar mejor los hilos en java; sin embargo, se recomienda primero, entender el funcionamiento low-level de los hilos en background.

Thread pool

- Es una colección de hilos que corren en background y a solicitud de la aplicación.
- Por defecto una aplicación en Android solo crea el Thread UI.
- Crear hilos es una tarea costosa computacionalmente, por tal motivo, se debe realizar una única vez al iniciar su aplicación.
- Para crear hilos y mandar un tarea a background, se utiliza `ExecutorService`
- Puede aplicar más configuraciones al ExecutorService [aquí](#).

Clase “**Application**”

Permite definir variables globales que persisten a lo largo de toda la aplicación. **Se ejecutan antes de los Activities.**

Ejemplo

- Cree una clase y herede de Application.
- Cree 4 hilos con el ExecutorService.

```
public class ApplicationThreads extends Application {  
  
    public ExecutorService executorService = Executors.newFixedThreadPool(4);  
}
```

Registrar la clase

Luego de crear su clase que hereda de Application, debe registrarla en el **Manifest**.

```
<application
    android:name=".ApplicationThreads"
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="Clase 4"
```



Ejercicio

Cree un contador desde 1 a 10 (con descansos de 1 segundo) que no se detenga aún si gira la pantalla.

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="188dp"
    android:text="INICIAR CONTADOR"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.497"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/contadorVal"
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:textSize="24sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintVertical_bias="0.398" />
```



Prueba 1

Correr todo en UI thread.

```
public class MainActivity0 extends AppCompatActivity {

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        binding.button.setOnClickListener(view -> {
            for (int i = 1; i <= 10; i++) {

                binding.contadorTextView.setText(String.valueOf(i));
                Log.d( tag: "msg-test", msg: "i: " + i);
                try {
                    Thread.sleep( millis: 1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        });
    }
}
```

Vea los logs....

```
I/Choreographer: Skipped 32 frames!  The application may be doing too much work on its main thread.
I/Choreographer: Skipped 603 frames!  The application may be doing too much work on its main thread.
```


Usando ExecutorService

Para utilizar hilos en background, debe llamar a ExecutorService (el que creó en su Application).

```
// ExecutorService
ApplicationThreads application = (ApplicationThreads) getApplication();
ExecutorService executorService = application.executorService;

binding.button.setOnClickListener(view -> {
    executorService.execute(new Runnable() {
        @Override // oscar-diaz
        public void run() {
            for (int i = 1; i <= 10; i++) {
                binding.contadorTextView.setText(String.valueOf(i));
                Log.d( tag: "msg-test-executorservice", msg: "i: " + i);
                try {
                    Thread.sleep( millis: 1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }
    });
});
```

Todo se ve bien... sin embargo:

- Está manipulando UI fuera del UI Thread
- Gire la pantalla...

View Model + Java concurrent

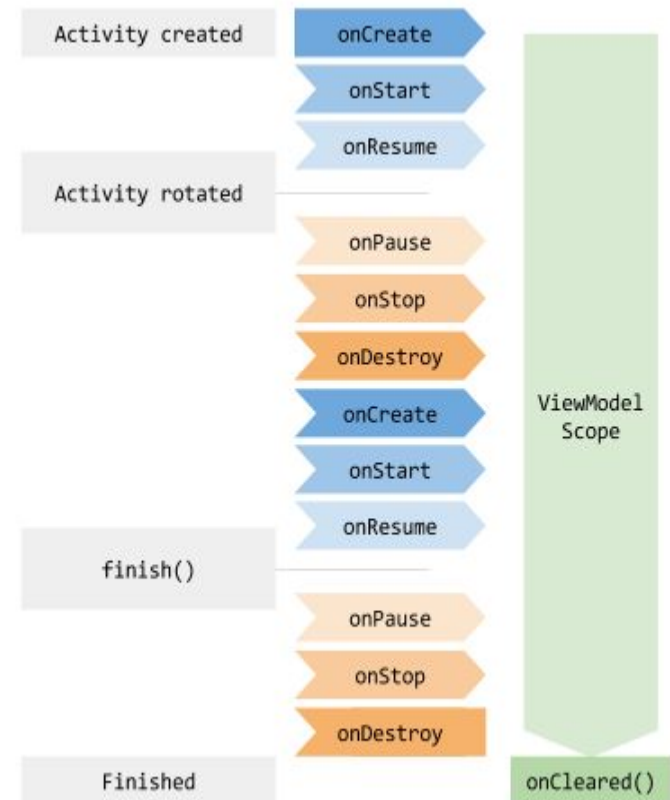
Clase View Model

Esta clase permite mantener un estado de las variables independiente del ciclo de vida de la actividad (por ejemplo, si se volvió a crear al girar la pantalla).

Clase observable y consciente del ciclo de vida, permite a la UI observar cambios en los datos.

Para usar **viewmodel** es necesario adicionar la siguiente dependencias en build.gradle:

```
implementation 'androidx.lifecycle:lifecycle-viewmodel:2.6.1'  
implementation 'androidx.lifecycle:lifecycle-viewmodel-savedstate:2.6.1'
```



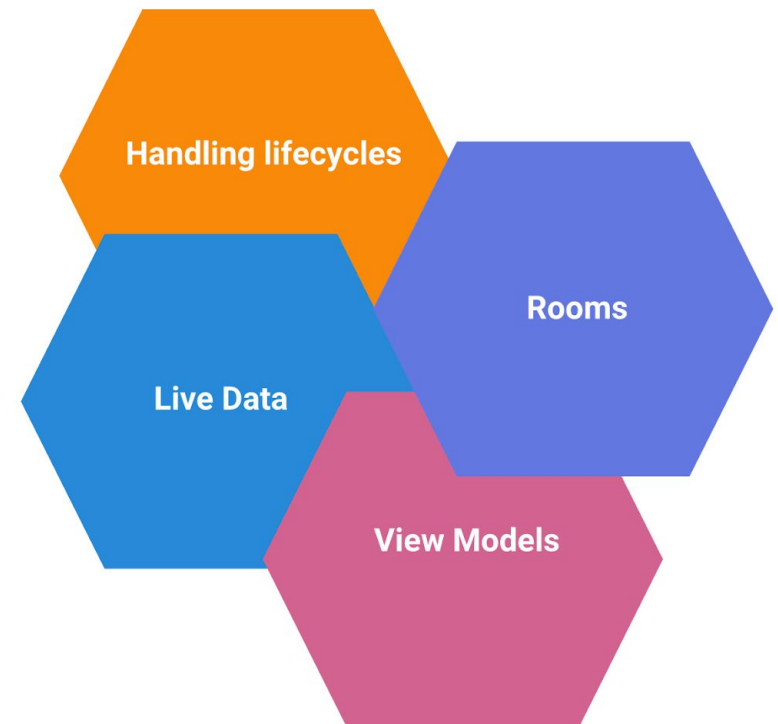
Clase View Model

Es parte de los llamados “**Componentes de arquitectura de Android**”, los cuales en conjunto, facilitan el trabajo en background entre las vistas y fragmentos.

En este curso se verán dos componentes:

- View models
- Live Data

Android Architecture Components



Clase View Model

Es necesario crear una clase que herede de **ViewModel**, *pues esta clase contendrá los objetos cuyo valor debe persistir aún si la actividad se recrea como resultado de girar la pantalla*. Así mismo, aquí se colocará la lógica del contador.

```
public class ContadorViewModel extends ViewModel {
```

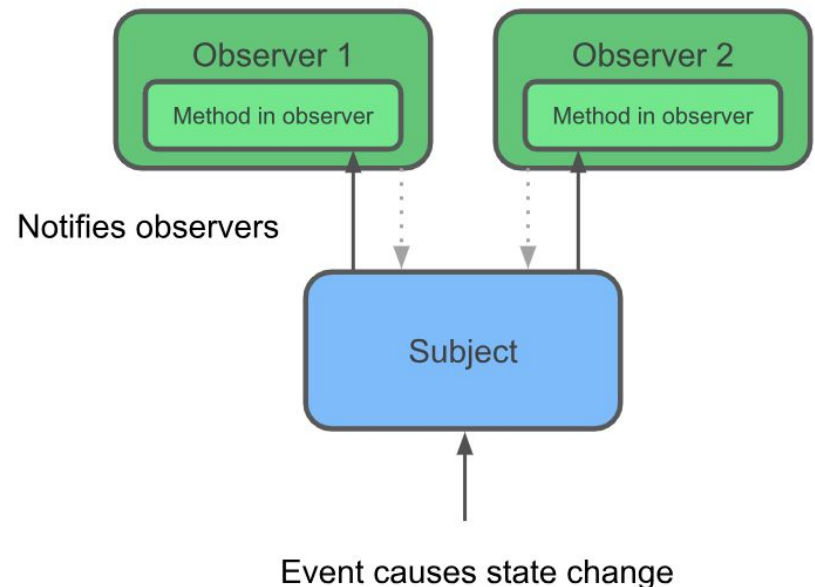
Atributos: Los atributos de la clase creada deben permitir almacenar la información que debe permanecer “viva” aún si se gira la pantalla, en este caso, sería el contador.

→ Para que un atributo sea “vivo” este debe definirse como ***MutableLiveData***.

MutableLiveData y Observers

- **MutableLiveData**: Clase que hereda de **LiveData** y *permite contener y modificar* objetos que pueden ser “observables” desde la UI Thread.
- **Observers**: Interfaz que permite *escuchar* por cambios en objetos del tipo MutableLiveData.

Observer Pattern



MutableLiveData y Observers

Concepto	¿Qué hace?	¿Dónde se usa?
LiveData	Observa datos, pero no los modifica	UI (Activity/Fragment)
MutableLiveData	Modifica y expone datos	ViewModel
Observer	Escucha cambios en el LiveData	UI (Fragment/Activity)

MutableLiveData

Entonces, se define la variable “contador” del tipo **MutableLiveData** para que pueda ser observada desde el hilo principal cuando suceda un cambio, con su correspondiente get (no existen los setter mutables).

```
public class ContadorViewModel extends ViewModel {  
  
    private final MutableLiveData<Integer> contador = new MutableLiveData<>();  
  
    public MutableLiveData<Integer> getContador() {  
        return contador;  
    }  
}
```

Lo que se define dentro de <.....> es el tipo de dato que servirá para ser enviado entre el hilo en background y la vista.

Obtener una instancia del View Model

Para obtener una instancia del ViewModel, se utiliza el **ViewModelProvider**, usando como referencia la Actividad actual y el ViewModel deseado.

```
ContadorViewModel contadorViewModel =  
    new ViewModelProvider( owner: MainActivity.this).get(ContadorViewModel.class);
```

Al usar el método **get()**, ViewModelProvider obtendrá la instancia de su ContadorViewModel bajo la siguiente condición:

- Si se había creado previamente, obtiene el estado de la clase (con todo el valor de sus atributos previos) y se la devuelve.
- Si es la primera vez que se crea, instancia una nueva clase ViewModel lista para mantener sus datos.

Setear valores → **setValue()** & **postValue()**

Para poner un valor a un **MutableLiveData** y notificar a los observadores se tiene:

- Si está *en el hilo principal* (UI Thread) → **setValue()**
- Si está *en un hilo fuera del hilo principal* (UI Thread) → **postValue()**

Considere que tanto **setValue()** como **postValue()** notificarán a todos los “observadores” de esta variable, que el valor ha cambiado.

Actualizando el código - notificando observers

```
binding.button.setOnClickListener(view -> {  
  
    //es un hilo en background  
    executorService.execute(() -> {  
        for (int i = 1; i <= 10; i++) {  
            contadorViewModel.getContador().postValue(i);  
            Log.d( tag: "msg-test", msg: "i: " + i);  
            try {  
                Thread.sleep( millis: 1000);  
            } catch (InterruptedException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    });  
});
```

Se actualiza el valor
del contador.

Capturar en valor en el UI Thread → Observers

El contador está cambiando su valor y a la vez notificando a todos con el método `postValue()`. Para que esta nueva actualización de su valor pueda ser capturada en la UI Thread, es necesario crear un observador.

Esta funcionalidad debe ser implementada en el método **`onCreate()`**.

```
ContadorViewModel contadorViewModel =  
    new ViewModelProvider( owner: MainActivity.this).get(ContadorViewModel.class);
```

Del ViewModel se obtiene la variable `MutableLiveData`, en este caso, `contador` (con **`getContador()`**)

Con el método **`observe`** se indica que se estará supervisando cualquier cambio que suceda en la variable `MutableLiveData`, en este caso: **`contador`**.

```
contadorViewModel.getContador().observe( owner: this, contador -> {  
    binding.contadorTextView.setText(String.valueOf(contador));  
});
```

El método **`observe`** recibe dos parámetros: la actividad donde se supervisará la variable, y una instancia de `Observer`, donde se obtendrá la información en caso de cambio.

OJO

Si al correr el proyecto tiene problema de clases duplicadas, añadir:

```
implementation(platform("org.jetbrains.kotlin:kotlin-bom:1.8.0"))
```

Trabajo persistente

- Work Manager

Trabajo persistente

- Es cuando permanece programado mediante reinicios de la aplicación o incluso, reinicios del sistema. **WorkManager** es la solución recomendada para el trabajo persistente.
- Debido a que la mayor parte del procesamiento en segundo plano se logra mejor a través del trabajo persistente, WorkManager es también la API principal recomendada para el procesamiento en segundo plano en general.

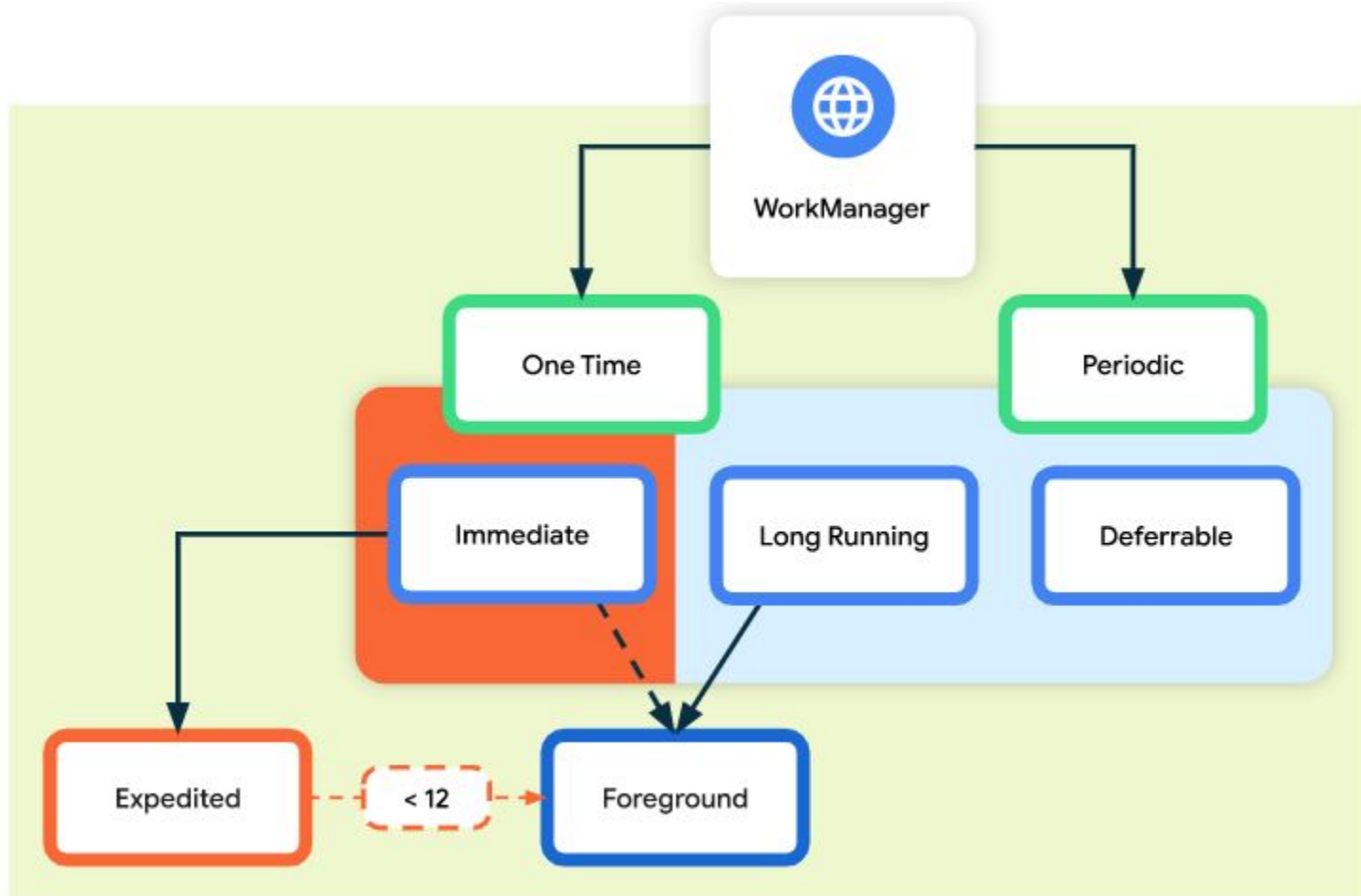
Tipos de trabajo persistente:

- Inmediato: tareas que deben comenzar de inmediato y completarse pronto.
- Larga ejecución: tareas que pueden ejecutarse durante más tiempo, potencialmente más de 10 minutos.
- Aplazable: tareas programadas que comienzan en un momento posterior y pueden ejecutarse periódicamente.

Mayor información en:

<https://developer.android.com/guide/background/persistent/getting-started>

Tipos de trabajo persistente



Dependencias

Para java en **build.gradle** (Module):

```
implementation "androidx.work:work-runtime:2.8.1"
```

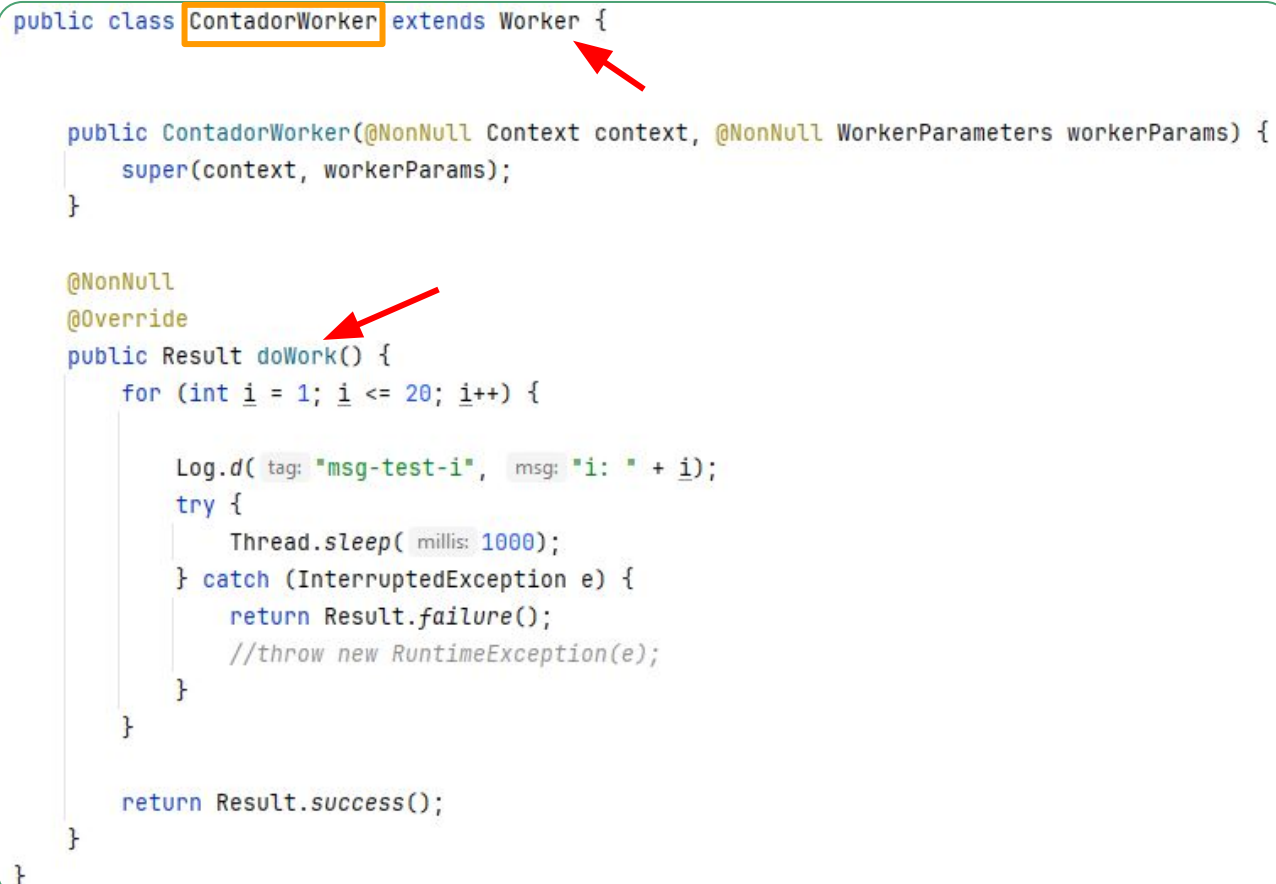
Ejercicio

Se creará un contador que imprima en el Log del 1 al 10 incluso si la app se minimiza/cierra.

Crear el Worker

Se debe crear una clase que herede de `Worker` y en el método `doWork()` se ejecuta el trabajo en background.

```
public class ContadorWorker extends Worker {  
  
    public ContadorWorker(@NonNull Context context, @NonNull WorkerParameters workerParams) {  
        super(context, workerParams);  
    }  
  
    @NonNull  
    @Override  
    public Result doWork() {  
        for (int i = 1; i <= 20; i++) {  
  
            Log.d( tag: "msg-test-i", msg: "i: " + i);  
            try {  
                Thread.sleep( millis: 1000);  
            } catch (InterruptedException e) {  
                return Result.failure();  
                //throw new RuntimeException(e);  
            }  
        }  
  
        return Result.success();  
    }  
}
```

A diagram showing a code snippet for a Java class named ContadorWorker. The class is defined as 'public class ContadorWorker extends Worker {'. A red arrow points from the word 'Worker' to the 'extends' keyword. The class has a constructor 'public ContadorWorker(@NonNull Context context, @NonNull WorkerParameters workerParams) { super(context, workerParams); }'. It also has an '@Override' method 'public Result doWork() {'. A red arrow points from the '@Override' annotation to the 'doWork()' method. The 'doWork()' method contains a 'for' loop from 1 to 20, logging each iteration, sleeping for 1000 milliseconds, and returning 'Result.failure()' if interrupted. After the loop, it returns 'Result.success()'.

Resultado de un Worker

Un worker debe siempre devolver información de lo sucedido:

- **`Result.success()`** → Si fue exitoso
- **`Result.failure()`** → Si ocurrió un error

Lanzando el worker - OneTimeWorkRequest

Según la periodicidad que se desee ejecutar, se tienen diferentes clases, en este ejemplo se lanzará un worker de 1 sola ejecución.

Luego de crear el WorkRequest, este se encola con el WorkManager.

```
binding.buttonWorkMang.setOnClickListener(view -> {  
  
    WorkRequest workRequest = new OneTimeWorkRequest.Builder(ContadorWorker.class).build();  
  
    WorkManager  
        .getInstance(binding.getRoot().getContext())  
        .enqueue(workRequest);  
});
```

Envío y recepción de datos

Se creará un contador que imprima en el Log del número X (que se le envíe) hasta $X + 10$, incluso si la app se minimiza/cierra. Si la app está abierta, debe mostrar en la interfaz. Si la app se cierra y luego se vuelve a abrir, si el hilo está corriendo, debe mostrar el resultado.

Envío de parámetros al Worker

Se utiliza la **clase Data** con su builder. Cada parámetro se envía utilizando **putX**, donde X es el tipo de dato.

Debe identificar a su Worker por ID o por TAG para luego poder observarlo.

```
UUID uuid = UUID.randomUUID();

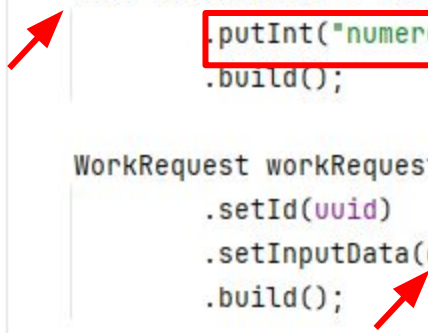
binding.buttonWorkMang.setOnClickListener(view -> {

    Data dataBuilder = new Data.Builder()
        .putInt("numero", new Random().nextInt( bound: 10))
        .build();

    WorkRequest workRequest = new OneTimeWorkRequest.Builder(ContadorWorker3.class)
        .setId(uuid)
        .setInputData(dataBuilder)
        .build();

    WorkManager
        .getInstance(MainActivity3.this.getApplicationContext())
        .enqueue(workRequest);

});
```



Recepción de parámetros en el Worker

- Utilizando `getInputData()` se tiene acceso a la data enviada al Worker

```
@NonNull
@Override
public Result doWork() {

    int contador = getInputData().getInt( key: "numero", defaultValue: 0);
    int contadorFinal = contador + 10;

    while (contador <= contadorFinal) {

        Log.d( tag: "msg-test-contador", msg: "contador: " + contador);
        setProgressAsync(new Data.Builder().putInt("contador", contador).build());
        contador++;
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            return Result.failure();
        }
    }

    return Result.success();
}
```



Envío de actualizaciones parciales desde el Worker

- Con el método `setProgressAsync()`, puede enviar actualizaciones parciales mientras el trabajo no finaliza.


```
@NonNull
@Override
public Result doWork() {

    int contador = getInputData().getInt( key: "numero", defaultValue: 0);
    int contadorFinal = contador + 10;

    while (contador <= contadorFinal) {

        Log.d( tag: "msg-test-contador", msg: "contador: " + contador);
        setProgressAsync(new Data.Builder().putInt("contador", contador).build());
        contador++;
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            return Result.failure();
        }
    }

    return Result.success();
}
```



Recepción de actualizaciones en el Activity

Siguiendo el patrón observer, obtenemos una instancia del WorkManager y nos ponemos a escuchar los cambios que suceden sobre ese Worker.

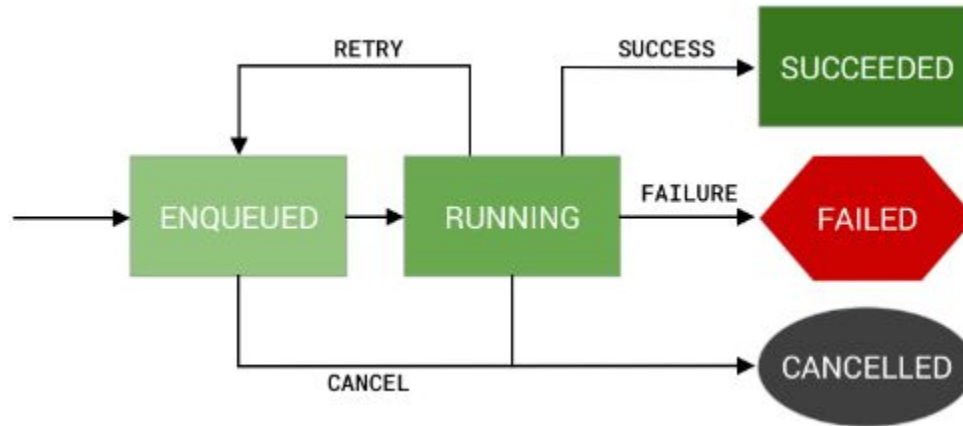
La variable **workInfo** tiene información del Worker corriendo en background.

```
WorkManager.getInstance(binding.getRoot().getContext()) WorkManager
    .getWorkInfoByIdLiveData(uuid) LiveData<WorkInfo>
    .observe( owner: MainActivity3.this, workInfo -> {
        if(workInfo != null){
            Data progress = workInfo.getProgress();
            int contador = progress.getInt( key: "contador", defaultValue: 0);
            Log.d( tag: "msg-test", msg: "progress: " + contador);
            binding.contadorVal.setText(String.valueOf(contador));
        }else{
            Log.d( tag: "msg-test", msg: "work info == null ");
        }
    });
```

Si lo prueba, todo funciona bien excepto al final, que aparece siempre 0.

Validación del estado

Si envía actualizaciones parciales, estas se realizan mientras el Worker está en estado Running. Cuando termina, este pasa al estado Succeeded o failed, en función del código.



Cuando el Worker termina, no se envía ningún parámetro, por tal motivo, el valor recibido es 0, pues este solo se envía cuando es una actualización parcial.

Envío de data al finalizar el Worker

Puede enviar data al finalizar el Worker para realizar alguna validación.

```
Data data = new Data.Builder()
    .putString("info", "Worker finalizado")
    .build();

return Result.success(data);
```

Validación de estado y captura de data

Con el método `workInfo.getState()` valida el estado del Worker y con `getOutputData()`, la información enviada cuando el estado es Succeeded.

```
WorkManager.getInstance(binding.getRoot().getContext())
    .getWorkInfoByIdLiveData(uuid)
    .observe( owner: MainActivity3.this, workInfo -> {
        if (workInfo != null) {
            if (workInfo.getState() == WorkInfo.State.RUNNING) {
                Data progress = workInfo.getProgress();
                int contador = progress.getInt( key: "contador", defaultValue: 0);
                Log.d( tag: "msg-test", msg: "progress: " + contador);
                binding.contadorVal.setText(String.valueOf(contador));
            } else if (workInfo.getState() == WorkInfo.State.SUCCEEDED) {
                Data outputData = workInfo.getOutputData();
                String texto = outputData.getString( key: "info");
                Log.d( tag: "msg-test", texto);
            }
        } else {
            Log.d( tag: "msg-test", msg: "work info == null ");
        }
    });
```

¿Preguntas?

Muchas gracias
