UiThread

https://github.com/androidannotations/androidannotations/wiki/WorkingWithThreads#uithread

En Android sólo un hilo puede escribir en la interfaz de usuario. Entonces, como hemos visto en la clase anterior, la anotación `@Background` no puede escribir en la UI. Para solventar este "problema", AndroidAnnotation tiene la anotación `@UiThread` la cual permite que un método pueda ejecutarse en el hilo de la interfaz de usuario. Vamos a verlo!

Vamos a crear un nuevo proyecto, y pondremos un TextView con el ID `tv_example` y vamos a escribir este código:

```
@EActivity(R.layout.activity_main)
public class MainActivity extends AppCompatActivity {

    @ViewById
    TextView tv_example;

    @AfterViews
    void initAfterViews() {
    initForUI();
    }

    @Background
    void initForUI() {
    updateTheUI("Updated in '@UiThread' annotated method");
    }

    @UiThread
    void updateTheUI(String toUpdate) {
    tv_example.setText(toUpdate);
    }
}
```

Ahora lancemos la aplicación. El TextView habrá sido actualizado con el mensaje `Updated in '@UiThread' annotated method`.

Test @UiThread

Dijimos que sólo un hilo puede escribir en la UI. Vamos a imaginar este caso:

- *Tenemos dos métodos anotados `@UiThread`.*
- *El primero es realmente lento (realiza operaciones lógicas, actualiza la interfaz de usuario, realiza operaciones lógicas nuevamente y finalmente actualiza la interfaz de usuario).*
- *El segundo método proviene de un método anotado `@Background`. Ambos actualizan el TextView*

```
@AfterViews
void initAfterViews() {
    slowUIUpdaterUiThread();
    initForUIBackground();
}

@Background
void initForUIBackground() {
    updateTheUIUiThread("Starting process...");
}

@UiThread
void updateTheUIUiThread(String toUpdate) {
    tv_example.setText(toUpdate);
}

@UiThread
void slowUIUpdaterUiThread() {
    try {
        sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    String toUpdate = "Updated in '@UiThread' annotated method";
    tv_example.setText(toUpdate);


    try {
        sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    toUpdate = "Done!";
    tv_example.setText(toUpdate);
}
```

Se mostrará el mensaje `Updated in '@UiThread' annotated method` y el mensaje de `Done!`?

No se mostrará porque el segundo proceso largo ("sleep(2000);") está bloqueando el hilo de UI y por lo tanto la operación de "tv_example.setText" no tiene slots de tiempo entonces no será ejecutada o mejor dicho, no le dará tiempo.

Podemos ver un ejemplo un poco más claro si comentamos el método `initForUIBackground();` sucederá lo mismo que anteriormente, no mostrará `Updated in '@UiThread' annotated method` por carencia de tiempo pero sí mostrará el `Done!`

Por lo tanto y resumiendo, la UI comenzará, se bloqueará y finalmente mostrará en la UI lo que le de tiempo. La moraleja de esto es los métodos con `@UiThread` **deben ser lo más pequeños posibles** y con la mínima lógica posible. La lógica **debe** estar dentro de los métodos anotados con `@Background`.

A continuación la solución correcta:

```
@AfterViews
void initAfterViews() {
    initSlowUIUpdaterUiThread();
    initForUIBackground();
}

@Background
void initForUIBackground() {
    updateTheUIUiThread("Starting process...");
}

@UiThread
void updateTheUIUiThread(String toUpdate) {
    tv_example.setText(toUpdate);
}

@Background
void initSlowUIUpdaterUiThread() {
    try {
        sleep(3000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    slowUIUpdaterUiThread();
}

@UiThread
void slowUIUpdaterUiThread() {
    String toUpdate = "Updated in '@UiThread' annotated method";
    tv_example.setText(toUpdate);
    secondLogicFromSlowUIUpdaterUiThread();
}

@Background
void secondLogicFromSlowUIUpdaterUiThread() {
    try {
        sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    finnishUIUpdaterUiThread();
```

```
}

@UiThread
void finnishUIUpdaterUiThread() {
    String toUpdate = "Done!";
    tv_example.setText(toUpdate);
}
```

UiThread - Delay

https://github.com/androidannotations/androidannotations/wiki/WorkingWithThreads#delay-1

Al igual que la anotación de background, los métodos anotados con `@UiThread` pueden demorarse:

```
@UiThread(delay = 10000)
void finnishUIUpdaterUiThread()...
```

UiThread - Propagation

https://github.com/androidannotations/androidannotations/wiki/WorkingWithThreads#propagation

Antes de la versión `3.0`, las llamadas al método anotado `@Thread` siempre se agregaban a la cola de ejecución del controlador para garantizar que la ejecución se realizaba en el hilo Ui. AA en la versión `3.0`, se mantiene este mismo comportamiento por propósitos de compatibilidad.

Pero, las llamadas de UiThread pueden ser optimizadas, para ello debemos cambiar el valor de propagación a `REUSE`. En esta configuración, el código realizará una llamada directa al método si el hilo actual ya es un hilo de UI. Si no lo fuese, el comportamiento sería el inicial por defecto, es decir, llamará del controlador:

```
@UiThread(propagation = UiThread.Propagation.REUSE)
void reusedFinnishUIUpdaterUiThread() {
    String toUpdate = "Reusing UI, it is more optimal: Done!";
    tv_example.setText(toUpdate);
}
```

Si combinamos el `delay` con la `propagation` ésta última será ignorada and la llamada será siempre manejada por defecto, es decir llamando a la cola de ejecución del controlador.

UiThread - Id

https://github.com/androidannotations/androidannotations/wiki/

WorkingWithThreads#id-1

En caso de querer cancelar una tarea de hilo de UI, podemos usar el campo id (al igual que en background). Cada tarea podría ser cancelada por `UiThreadExecutor.cancelAll("id");`:

```
    void myMethod() {
        someCancellableUiThreadMethod("hello", 42);
        // [...]
        UiThreadExecutor.cancelAll("cancellable_task");
    }

    @UiThread(id="cancellable_task")
    void someCancellableUiThreadMethod(String aParam, long anotherParam)
        // [...]
    }
```

Si la tarea ya está lanzada, no será cancelada. Por otro lado las tareas con Las tareas con `propagation REUSE` no pueden tener un ID y no pueden cancelarse.