{desafío}
latam_

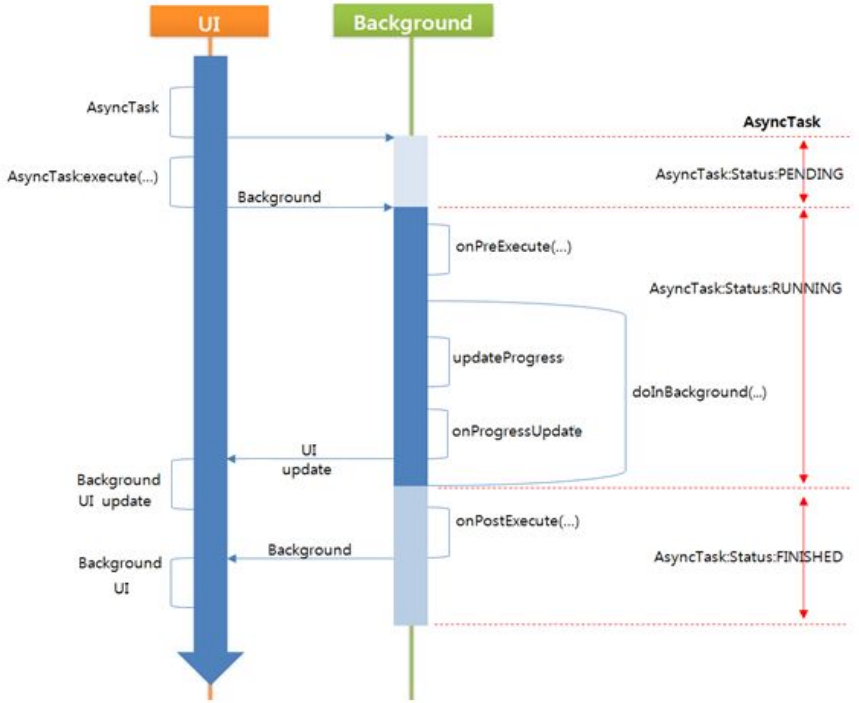
Threads y Coroutines _

Parte II

AsyncTask

AsyncTask

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.



Métodos de clase genéricos - AsyncTask

- **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.

Clase auxiliar ejemplo - AsyncTask

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
protected Long doInBackground(URL... urls) {
// code that will run in the background
return ;
}

protected void onProgressUpdate(Integer... progress) {
// receive progress updates from doInBackground
}

protected void onPostExecute(Long result) {
// update the UI after background processes completes
}
}
```

Iniciar un AsyncTask

Una clase AsyncTask se ejecutaría desde el mainThread a través de una actividad o fragmento de la siguiente forma (parámetros son de ejemplo):

```
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().

Indicaciones para el uso correcto - AsyncTask

- La clase AsyncTask debe cargarse en el hilo de la interfaz de usuario (UI Thread). Esto se hace automáticamente a partir de la versión android JELLY_BEAN.

- La instancia de la tarea debe crearse en el subproceso de la interfaz de usuario (UI Thread).

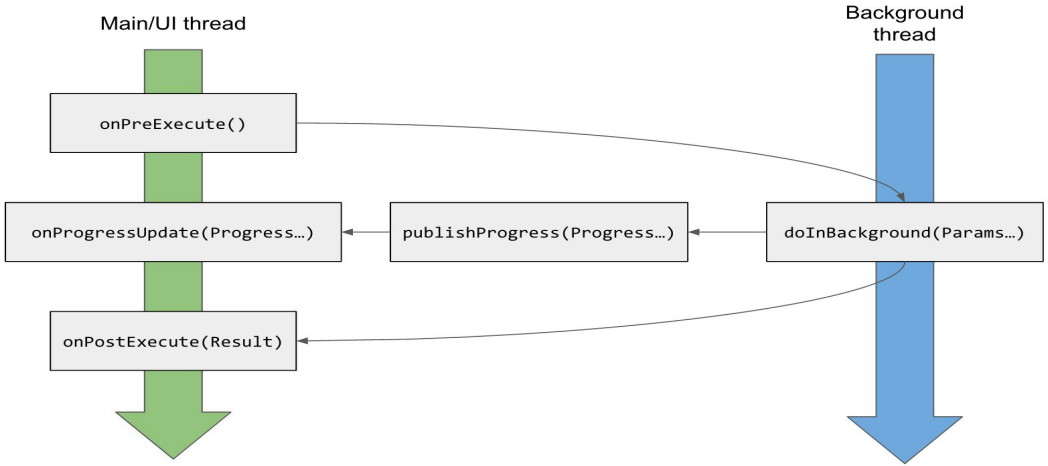
- El método execute(), debe invocarse en el hilo de la interfaz de usuario (UI Thread).

- No se debe invocar los métodos de la clase AsyncTask por separado o manualmente doInBackGround, PostExecute, etc.).

- La tarea se puede ejecutar solo una vez. Si se intentará ejecutar por una segunda vez una excepción debe ser enviada.

preExecute() - AsyncTask

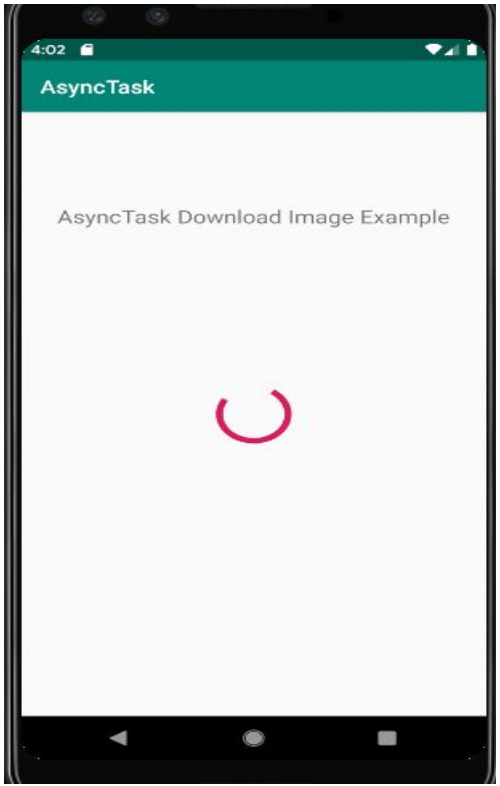
Este método se puede invocar en el subproceso de la interfaz de usuario antes de ejecutar la tarea (UI Thread). Este paso se usa normalmente para configurar la tarea, por ejemplo, iniciando una barra de progreso en la interfaz de usuario.



Ejercicio 5

Crear un proyecto llamado “AsyncTask” donde se descargue al momento de iniciarse la aplicación, la imagen del día de la nasa en formato HD y sea instanciada la imagen en una vista tipo ImageView. También se requiere el uso de una progressbar que se muestre en el lugar que ocupará la imagen una vez se haya descargado.

Ejercicio 5 - Solución



Promises y Coroutines

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).

Cada llamada al método .then() espera la promesa anterior y ejecuta la siguiente función, luego convierte el resultado en una promesa si es necesario. Esto nos permite encadenar sin problemas llamadas síncronas y asíncronas. Simplifica tanto el código, que la mayoría de las nuevas especificaciones devuelven promesas de sus métodos asincrónicos.

CallBack vs Promises

```
//CallBack Function example
fun isUserTooYoung(id, callback) {
  openDatabase(fun(db) {
    getCollection(db, 'users', fun(col) {
      find(col, {'id': id}, fun(result) {
        result.filter(fun(user) {
          callback(user.age < cutoffAge);
        });
      });
    });
  });
}
```

VS

```
fun isUserTooYoung(id): Db { //Promise function example
  return openDatabase() // returns a promise
  .then(fun(db: Database) {return getCollection(db, 'users');})
  .then(fun(col: Collection) {return find(col, {'id': id});})
  .then(fun(user: User) {return user.age < cutoffAge;});
}
```

{desafío} latam_

Promises.All

```
//Promises.All Example

var promise1 = getJSON('/users.json');
var promise2 = getJSON('/articles.json');

Promise.all([promise1, promise2]) // Array of promises to complete
.then(fun(results) {
  println('all data has loaded');
})
.catch(fun(error) {
  println('one or more requests have failed: ' + error);
});
```

Coroutines

Una Coroutine 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 lenguajes.

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

Tareas principales - Coroutines

- Administrar tareas de larga duración que de otro modo podrían bloquear el hilo principal y hacer que su aplicación se congele.
- Proporcionar seguridad en el hilo principal de la aplicación.
- Ejecutar llamadas (calls) con seguridad, ya sean hacia un api o internamente a la base de datos del dispositivo u otro contenedor de datos desde el hilo principal.

Procesos de larga duración - Coroutines

```
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) { /* ... */ }
```

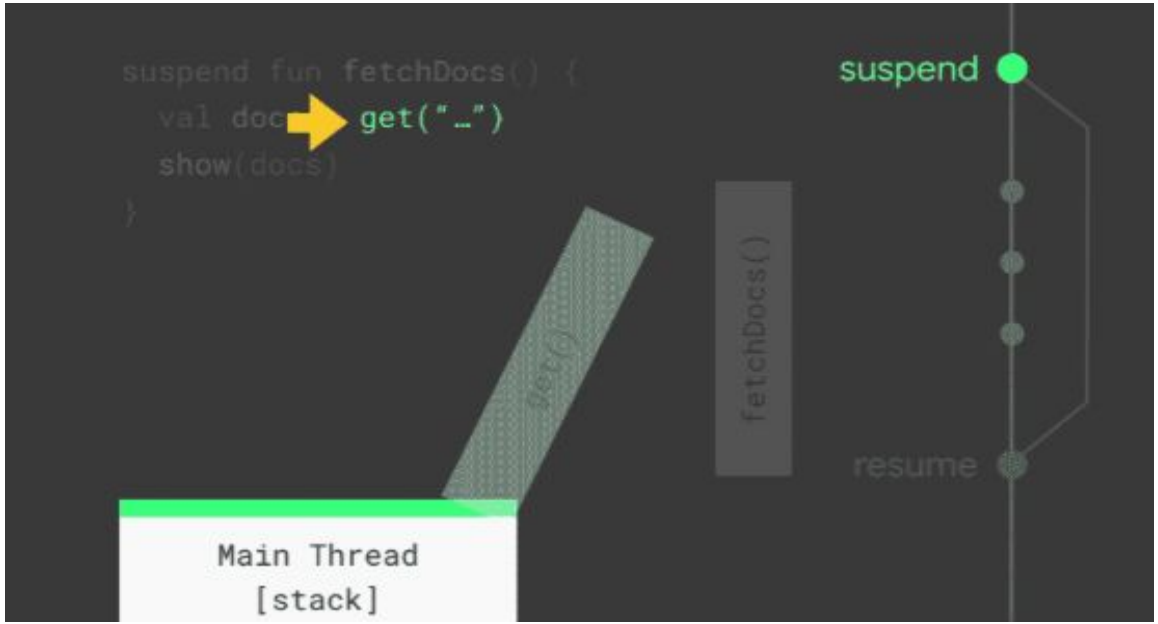
El método get() todavía se ejecuta en el hilo principal, pero suspende la rutina antes de que comience la solicitud de red. Cuando finaliza la solicitud de red, get() reanuda la rutina suspendida en lugar de utilizar una devolución de llamada para notificar al hilo principal.

Suspend / Resume - Coroutines

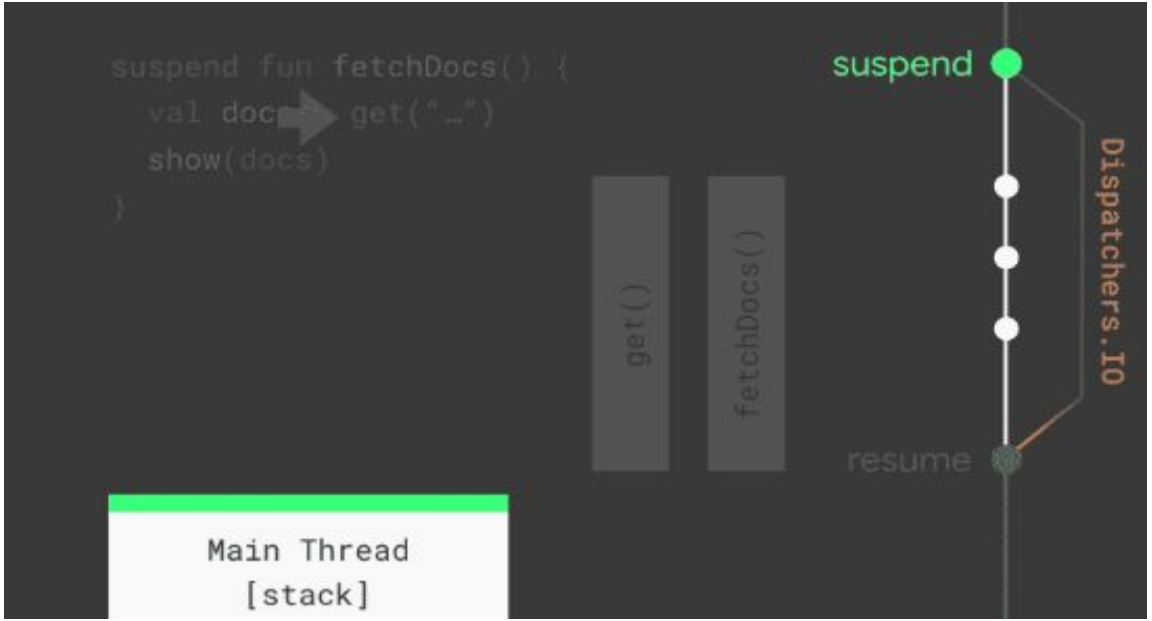
Las Coroutines se basan en funciones regulares agregando dos operaciones para manejar tareas de larga duración. Además de invoke(o call) y return, las Coroutines agregan suspend y resume:

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

Secuencia de funcionamiento - Coroutines



Secuencia de funcionamiento - Coroutines



Secuencia de funcionamiento - Coroutines

Seguridad - Coroutines

Las Coroutines 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 Coroutines de Kotlin que realicen el trabajo en un despachador predeterminado.

- **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.

- **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.

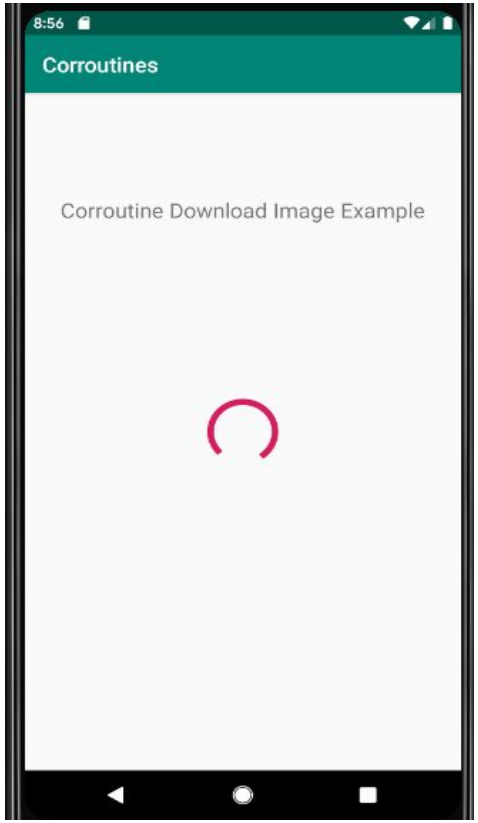
Iniciando Coroutines

- **launch**: comienza una nueva Coroutine 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.

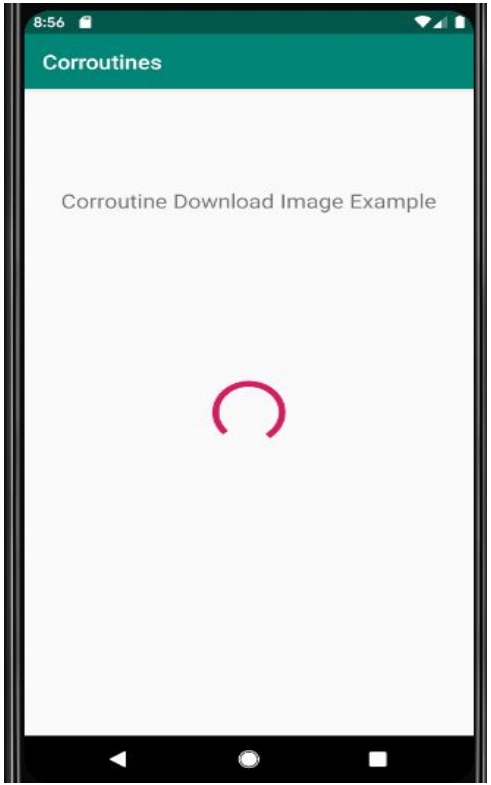
- **async**: inicia una nueva rutina y le permite devolver un resultado con una función de suspensión llamada await.

Ejercicio 6

Crear un proyecto llamado “Coroutines” que realice la descarga de la imagen del día de la nasa y la muestre en un imageview, utilizando java.net.URL. La actividad debe implementar CoroutineScope y tener un método que aplique launch{}, que a su vez llame otro método que ejecuta la descarga utilizando una referencia a Dispatchers.IO.



Ejercicio 6 - Solución



{desafío} latam_

Academia de talentos digitales

www.desafiolatam.com