Services & Background

Tasks
Introduction

• Last week we discussed various storage-related tasks
  • Reading/Writing Files, Databases
• Tasks like network requests or disk write/reading can potentially take some time to execute
• Let’s see how and why to handle suchs tasks in the background
Process Management Recap

• By default, components of a single app run in the same Linux process

• When the system wants to run a new component:
  • If app doesn’t yet have any running components, a new process with a single execution thread is started
  • Otherwise, component is started within the running process
  • By default, all components are run on the same thread!
    • “Main thread”

Note: It’s possible to control which process certain components run in
  • Manifest android:process XML attribute
  • Most apps don’t need to change this

Threads

• Main thread – handles UI
  • Dispatches events to UI widgets
    • Draw events
    • User interaction

• In most cases, your app interacts with the Android UI toolkit on the main thread

• Also other system calls will run on the main thread
  • Lifecycle events, system callbacks (onTouch)

Blocking calls on a single thread

• Synchronous code
  • Freezes UI / gives us exceptions (Networkonmainthread)

```kotlin
fun onDataNeeded() {
    val data = networkRequest()
    show(data)
}
```

• Common pattern is callback (e.g. firebase)

```kotlin
fun onDataNeeded() {
    networkRequest() { result ->
        show(result)
    }
}
```
Threads

• As there is 1 main execution thread, both application components and UI interactions are done **synchronously**

• Long computations, I/O background tasks on the main thread will block the UI

• If the UI thread hangs for more than a few seconds (5s), the “Application Not Responding (ANR)” dialog is presented
  • Poor user experience

• All long-running tasks should be run on a separate background thread (worker thread)
Basic thread creation in Kotlin

• Java-style:

```kotlin
Thread(object: Runnable {
    override fun run() {
        // some time-consuming task
    }
}).start()
```

• Kotlin `thread()` helper method:

```kotlin
thread(start = true) {
    // some time-consuming task:
    val bmp: Bitmap = processBitmap(s: "image.png")

    // TODO: do something with the result - show in UI, ..
}
```

https://developer.android.com/guide/components/processes-and-threads#WorkerThreads
https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.concurrent/thread.html
Threads continued

- However, we can update UI only from the main thread!

So..

- On one hand - should do work on worker threads
- On the other hand – can’t update UI from those worker threads

- What to do? Use messages and/or helper methods:
  - `Activity.runOnUiThread(Runnable)`
  - `View.post(Runnable)`
  - `View.postDelayed(Runnable, long)`

https://developer.android.com/guide/components/processes-and-threads#WorkerThreads
Example – sending results from Worker thread to UI thread

```java
thread(start = true) {
    // some time-consuming task:
    val bmp: Bitmap = processBitmap(s: "image.png")

    // show in UI:
    my_imageview.post {
        // code run on UI thread:
        my_imageview.setImageBitmap(bmp)
    }
}
```
Callbacks

• What if we want to define a function which
  • handles background work
  • but leaves the result handling to the function caller?

The below example is tied to my_imageview:

```kotlin
// we need to change this function to allow fun. caller use the bitmap where they want..
fun loadBitmap() {

    thread(start = true) {
        val bmp: Bitmap = processBitmap(s: "image.png")

        my_imageview.post {
            my_imageview.setImageBitmap(bmp)
        }
    }
}
```
Callbacks - defining

fun loadBitmap(callback: (Bitmap) -> Unit) {
    thread(start = true) {
        val bmp: Bitmap = processBitmap(s: "image.png")
        runOnUiThread {
            callback(bmp)
        }
    }
}

- Add a higher order function as a parameter
- Invoke that function with your result data

Callbacks - usage

```kotlin
fun loadBitmap( callback: (Bitmap) -> Unit ) {

    thread(start = true) {
        val bmp: Bitmap = processBitmap(s: "image.png")
        runOnUiThread { callback(bmp) }
    }

} // use it like so:
loadBitmap( callback = fun(bmp: Bitmap) {
    my_imageview.setImageBitmap(bmp)
})

// same as above, shortened:
loadBitmap { bmp : Bitmap -> my_imageview.setImageBitmap(bmp) }
```
Callbacks

```kotlin
fun loadBitmap( callback: (Bitmap) -> Unit ) {
    thread(start = true) {
        val bmp: Bitmap = processBitmap(s: "image.png")
        runOnUiThread { callback(bmp) }
    }
}

// use it like so:
loadBitmap( callback = fun(bmp: Bitmap){
    my_imageview.setImageBitmap(bmp)
})

// same as above, shortened:
loadBitmap { bmp: Bitmap -> my_imageview.setImageBitmap(bmp) }
```

Don’t forget to run on UI thread if you intend to manipulate UI! (Not all callbacks need this)
Main-safety

• A function is **main-safe** when it doesn't block UI updates on the main thread

```kotlin
// this function is not main-safe
fun onDataNeeded() {
    val data = networkRequest()
    show(data)
}

fun networkRequest() {
    // blocking, time-consuming task:
    return makeHttp_Request("https://www.ut.ee")
}
```
Main-safety

• This function is main-safe:

```kotlin
fun loadBitmap( callback: (Bitmap) -> Unit ) {
    thread(start = true) {
        val bmp: Bitmap = processBitmap(s: "image.png")
        runOnUiThread { callback(bmp) }
    }
}
```
Categories of background tasks

1) Immediate tasks
   • Needs to complete while user is interacting with the app

2) Exact tasks
   • Doesn’t need to run while user is interacting with the app, but should run at an exact time

3) Deferred tasks
   • Task can run later, at an opportune time, exact time doesn’t matter (e.g. will run when network availability is good and battery isn’t low)

Image: https://developer.android.com/guide/background
Solutions

1) Immediate
   • Threads (Java/Kotlin)
     • Use the helper methods from prev. slides
     • Use callbacks, do messaging with Handlers
     • ThreadPool
   • Kotlin coroutines
   • If tasks needs to start immediately, but continue running if user puts app to background - WorkManager

2) Exact
   • AlarmManager

3) Deferred
   • WorkManager - schedule tasks to be run later while balancing system resources

https://developer.android.com/guide/background#recommended-solutions
ThreadPool and ExecutorService

• If you are running lots of threads concurrently, you need to start managing them.

• Commonly, you may want to limit the max. no of threads running

• Thread pool is a FIFO task queue

Example:
  • Need to read in 100 text-files and use concurrent worker-threads
  • To limit resource consumption, you run no more than 10 worker threads at a time

https://developer.android.com/guide/background
ExecutorService example

```kotlin
val pool: ExecutorService = Executors.newFixedThreadPool(nThreads: 10)

• Add tasks to queue with execute() method, passing a Runnable:
  pool.execute { Runnable {
    // blocking call
  }

• If you need results, use .submit(), passing callable, returns Future:
  val future = pool.submit(object : Callable<String> {
    override fun call(): String {
      val result = blockingFunction()
      return result
    }
  })

  future.get()
```

Kotlin coroutines
Introduction

• Coroutine is a concurrency design pattern of Kotlin
• Recommended for asynchronous situations on Android
• Can be seen as a light-weight thread
  • Can run in parallel, wait for each other, communicate
  • Very lightweight – can easily create thousands of coroutines, while running thousands of threads is not straight-forward
• Convenient way of coding:
  • Write blocking and non-blocking code together sequentially, without call-backs

https://developer.android.com/kotlin/coroutines
How does it look like?

```kotlin
val scope = CoroutineScope(Dispatchers.Default)

scope.launch { this: CoroutineScope
    loadData()
}
```

- Coroutines can be run run with `launch`, (and `async`)
- They are run within some `CoroutineScope`
- Scope manages lifetime of one or more related coroutines. Scopes can be cancelled, which cancels all running coroutines in it
  - E.g when user closes fragment/activity
Running a coroutine

```kotlin
val scope = CoroutineScope(Dispatchers.Default)

scope.launch {
    this: CoroutineScope
    loadData()
}
```

• The coroutine is run on the thread determined by the dispatcher.

• There are handy built-in scopes:
  • `ViewModelScope`, `LifeCycleScope`

*Note: to use co-routines, you need the dependency:*

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9'
```
What dispatchers are available

• **Dispatcher.IO** - network and disk
• **Dispatcher.Default** - CPU intensive task
• **Dispatcher.Main** - UI code or non-blocking code (something which executes fast)
  - E.g. calling suspend functions, updating LiveData objects.
• OK – we can create a coroutine and set the dispatcher to control which thread is used.
• This doesn’t help with main safety..
Main-safety with coroutines

• In the previous example, the caller of loadData had to make sure its not running on UI thread

```
// loadData is not main-safe
fun loadData(){
    blockingCall()
}

val scope = CoroutineScope(Dispatchers.Default)

scope.launch { this: CoroutineScope
    loadData()
}
```
withContext

- By using `withContext()`, a coroutine can move the execution to a different thread
- Then, the caller of `loadData` can be running on main thread – the coroutine will take care of the switch

```kotlin
// loadData is now main-safe
suspend fun loadData()
{
    withContext(Dispatchers.Default){
        this: CoroutineScope
        blockingCall()
    }
}
```

- The execution Calling `withContext` switches to the other dispatcher just for the lambda then comes back to the dispatcher that called it with the result
withContext

• By using it you can make calling functions main-safe
• However, to use `withContext`, you have to mark your function as a `suspend` function
Suspend modifier

```kotlin
suspend fun loadData()
{
    withContext(Dispatchers.Default)
    {
        blockingCall()
    }
}
```

- Specifies that a function should be called from within a coroutine
- Doesn’t block the Main thread!
  - Execution gets *suspended* without blocking the thread
  - While waiting for a result, the thread used by the coroutine is unblocked so that other work (functions, coroutines) can continue running
- All suspend functions must be executed in a coroutine. *withContext is also a suspend function*
Suspend

• Thanks to this, we get sequential code, which actually runs asynchronously

```kotlin
suspend fun updateUi() {
    val data = loadData() // loadData will cause
    // execution to be suspended here

    display(data) // display() won't run until loadData is done
}
```

• Remember that suspend functions have to be called via coroutines:

```kotlin
CoroutinesScope(Dispatchers.Main).launch {
    updateUi()
}
```
Callbacks

```java
doNetworkRequest { result ->
    saveToDb(result) { dbResult ->
        // handle dbResult
    }
}
```

Coroutines

```java
val result = doNetworkRequest
val dbResult = saveToDb(result)
// handle dbResult
```
Starting coroutines

- Launch – “fire and forget”
  - starts a new coroutine
  - doesn't return the result to the caller
- Async
  - starts a new coroutine
  - allows you to return a result with a suspend function called `await`

Use async-await if you have to run multiple concurrent jobs and then return some result which depends on all of them

https://developer.android.com/kotlin/coroutines-adv#start
Services
Services

• Faceless components that run in the background
  • E.g. music player, network download, chat service notifies user even if they the app is not in the foreground

• Have higher priority than background activities, thus suitable for long-running tasks
  • Recall runtime memory management

• Services are started by app components using Intent
  • `startService( ..)`

https://developer.android.com/guide/components/services
Kinds of services

Service can take 3 forms:

• **Background service**
  • Called with `startService()`
  • Runs indefinitely
    • Until manually stopped

• **Foreground service**
  • A notification is shown while the service keeps running

• **Bound service**
  • App components bind to the service with `bindService()`
  • Runs as long as there are components bound to it
  • Provides a *binding* – a client-server interface

The type is determined by which callback methods you implement

• `onStartCommand()` – starts a background service
• `onBind()`

https://developer.android.com/guide/components/services#Types-of-services
Services - continued

Implementing your own service

• Extend Service class, declare component in Manifest

• Override callback methods
  • `onStartCommand()`, `onBind()`

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="ee.ut.cs.lecture6">
  ...
  <service android:name=".DownloadsService"
    android:enabled="true">
  </service>
</manifest>
```

https://developer.android.com/guide/components/services#Declaring
Kotlin example – *started* background service

```kotlin
class MyService: Service() {
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        val foo = intent?.getStringExtra("key1")
        // do some work
        return START_STICKY
    }

    override fun onBind(intent: Intent?): IBinder? {
        // should return a communication channel to the service
        return null // if not using bound service
    }
}
```

- Bound service example:
  - [https://developer.android.com/guide/components/bound-services.html#Binder](https://developer.android.com/guide/components/bound-services.html#Binder)
Starting, Stopping a service

- Start via Intent object
  - Attach extras, set action
    
    ```
    val intent = Intent(applicationContext, MyService::class.java)
    startService(intent)
    ```

- Stop service:
  - stopSelf() – from the service itself
  - stopService() – from other components

- Bound services start/stop based on component binding to them

- Android system may stop a service due to low memory
  - Generally you should be able to handle restarts of your service

https://developer.android.com/guide/components/services#StartingAService
Results of services with broadcasts

- Using broadcasts, your service can notify interested apps about some task finishing

```java
// In service:
var done = Intent()
done.setAction("ee.ut.cs.dl_file")
done.putExtra("downloadedFilePath", filePath)
sendBroadcast(done)

// In activity
val filter = IntentFilter()
filter.addAction("ee.ut.cs.dl_file")
registerReceiver(bmpReceiver, filter)
```
Services & Threads

• Services still run in the main thread of host process by default!

• Generally, you want to create a new thread inside the Service

• There is an extension of Service – JobIntentService, which creates a thread for you
  • Includes a sequential work queue
Notifications

• Messages displayed to user outside of app’s main UI in the top “navigation drawer”
  • System events, updates, new messages, tasks finishing

• Notification have:
  • Icons (1,4), titles (5), text (6), timestamps(3)
  • Interactive behaviour (actions upon click)

• May also appear in lock screen, wearables

Images: https://developer.android.com/guide/topics/ui/notifiers/notifications
https://developer.android.com/guide/topics/ui/notifiers/notifications
Building a basic notification

• Notification.Builder – to build the object
• NotificationManager – to send out the notification

```kotlin
val builder = NotificationCompat.Builder(this, CHANNEL_ID)
    .setSmallIcon(android.R.drawable.ic_dialog_info)
    .setContentTitle("Done")
    .setContentText("My Service finished")
    .setPriority(NotificationCompat.PRIORITY_DEFAULT)

val notification = builder.build()
with(NotificationManagerCompat.from(this)) {
    // notificationId is a unique int for
    // each notification that you define!!
    notify(notificationId, builder.build())
}
```
### Configuring notifications

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setAutoCancel( bool: Boolean)</td>
<td>Whether clicking hides the notification</td>
</tr>
<tr>
<td>setLargeIcon(resourceld)</td>
<td></td>
</tr>
<tr>
<td>setContentTitle(string)</td>
<td></td>
</tr>
<tr>
<td>setContentText(string)</td>
<td></td>
</tr>
<tr>
<td>setPriority()</td>
<td>Used on &lt;= 7.1, for 8.0+ use channel</td>
</tr>
<tr>
<td>setStyle</td>
<td>Various extra options: multiple texts, images, media payback buttons</td>
</tr>
<tr>
<td>setContentIntent()</td>
<td>Intent action which is run when clicked</td>
</tr>
<tr>
<td>setSound()</td>
<td></td>
</tr>
<tr>
<td>setTicker()</td>
<td>Scrolling text</td>
</tr>
<tr>
<td>addAction()</td>
<td>Extra button with action attached</td>
</tr>
<tr>
<td>setGroup()</td>
<td>Which group to appear in</td>
</tr>
</tbody>
</table>

For more information, visit:
- [https://developer.android.com/training/notify-user/expanded.html](https://developer.android.com/training/notify-user/expanded.html)
Notification Channel

• Since Android API 26 (Android 8.0), all notifications must be assigned a channel
  • Allows users to opt out selectively from certain notifications

```kotlin
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
    // Create the NotificationChannel
    val name = getString(R.string.channel_name)
    val descriptionText = getString(R.string.channel_description)
    val importance = NotificationManager.IMPORTANCE_DEFAULT
    val mChannel = NotificationChannel(CHANNEL_ID, name, importance)
    mChannel.description = descriptionText
    // Register the channel with the system; you can't change the importance
    // or other notification behaviors after this
    val notificationManager = getSystemService(NOTIFICATION_SERVICE) as NotificationManager
    notificationManager.createNotificationChannel(mChannel)
}
```

https://developer.android.com/guide/topics/ui/notifiers/notifications#ManageChannels
https://developer.android.com/training/notify-user/channels
Adding actions to notification

```java
createNotificationChannel()

val builder = ...

// Create an explicit intent for an Activity in your app
val intent = Intent(packageContext: this, MainActivity::class.java)
intent.putExtra(name: "processedImage", image)

// Wrap it in a PendingIntent
val pending: PendingIntent =
    PendingIntent.getActivity(context: this, requestCode: 0, intent, PendingIntent.FLAG_UPDATE_CURRENT)

// Set the intent fires when user taps the notification
builder.setContentIntent(pending)

val notification = builder.build()
```
Foreground service

• A service which the user is aware of – via notification
  • `startForeground(id: Int, notification: Notification)`
  • `stopForeground()`

• Has higher priority from system perspective

• Need permission
  `android.permission.FOREGROUND_SERVICE`

• Must provide a notification for the status bar
  • Cannot be dismissed unless the service is either stopped or removed from the foreground.

https://developer.android.com/guide/components/services#Foreground
To conclude

• UI can be updated only from UI thread
• Use coroutines or threads
• Services
  • Started, Foreground, Bound
  • Still need to manage threading!
• Notifications – another entry point into your application!
Next week

• Automated Android testing
  • Unit tests
  • UI tests

• Home Assignment 1 will be announced!