ASYNCHRONOUS PROGRAMMING

Roberto Beraldi

SYNC vs ASYNC: an analogy

- Suppose you call an help desk service to receive assistance
- In the sync model you keep waiting until the operator replies and then talk with she/he
- In the async model, you leave a message with your number and the operator will call you back

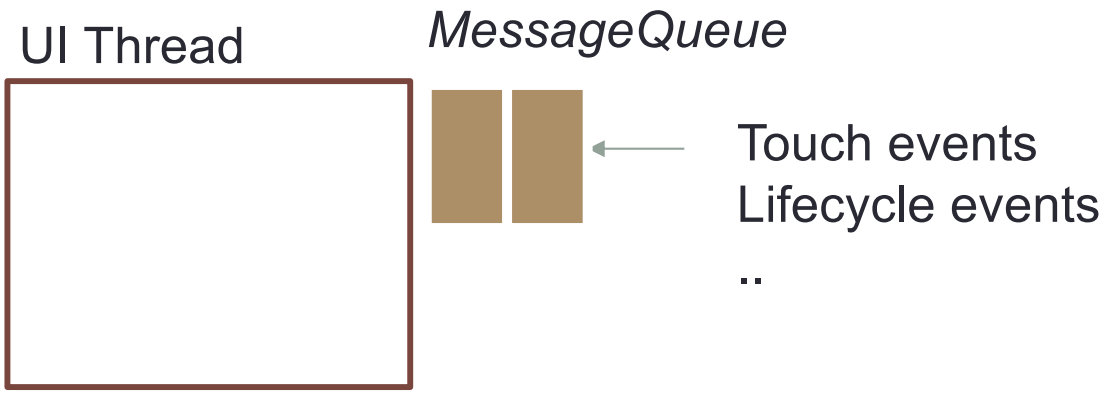
The need of async programming

- Asynchronous programming in android is required to achieve responsive UI
- Android provides different ‘frameworks’ (i.e., classes) to write concurrent code, according to typical problems
- All the frameworks are eventually built on top of standard java Thread facility, but the frameworks simplify programming compared to direct usage of threads.
- Nevertheless, sometimes Thread can be the right choice

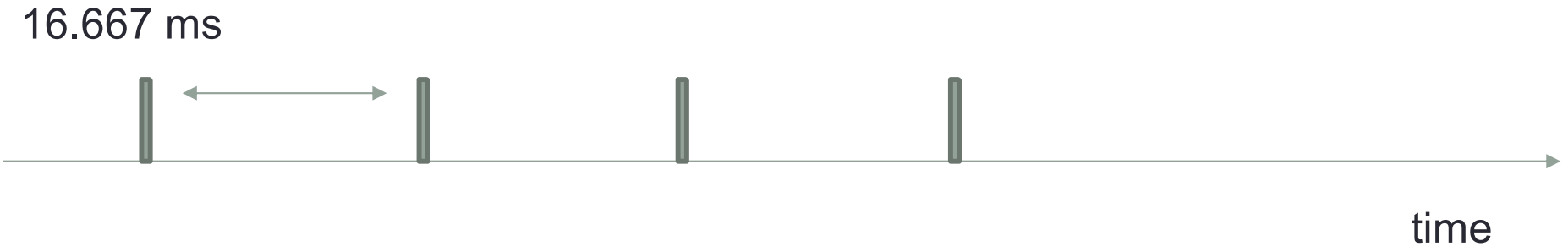
Typical time consuming activities

- File up/downlaod
- Web-API calls
- Async notifications
- Math Calculation
  - e.g., pattern recognition, image processing,..
- File I/O
- But also, playing music, etc.

Main thread model

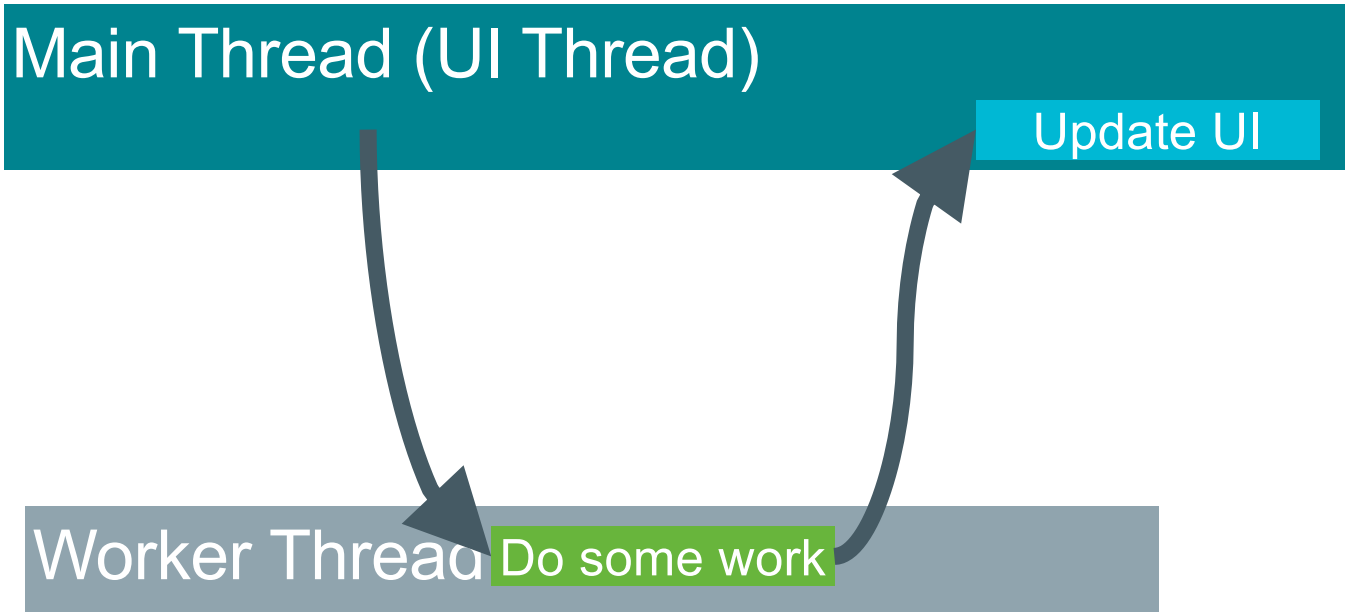


- The UI thread is designed as a reactive software, this means it continuously reads from a queue that contains information about which code to execute



time distance among two *onDraw* approx 16 ms

Async programming model



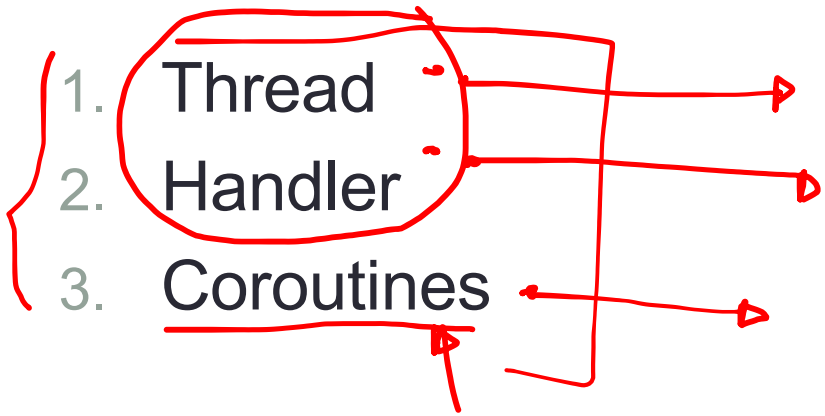
The UI thread is not **thread-safe**

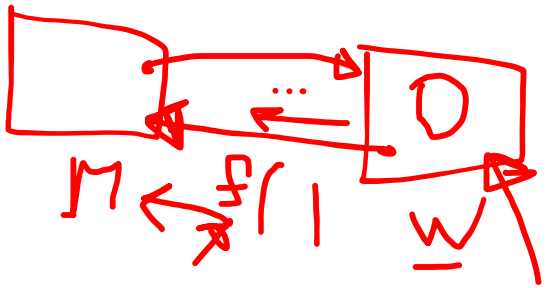
- Variables in the UI thread cannot be touched from a worker thread

- A worker thread cannot manipulate element of the UI
- All manipulation to the user interface must be done from the UI thread
- For example, if a secondary thread tries to update a TextView, the application crashes

How to write async programs

1. Thread
2. Handler
3. Coroutines

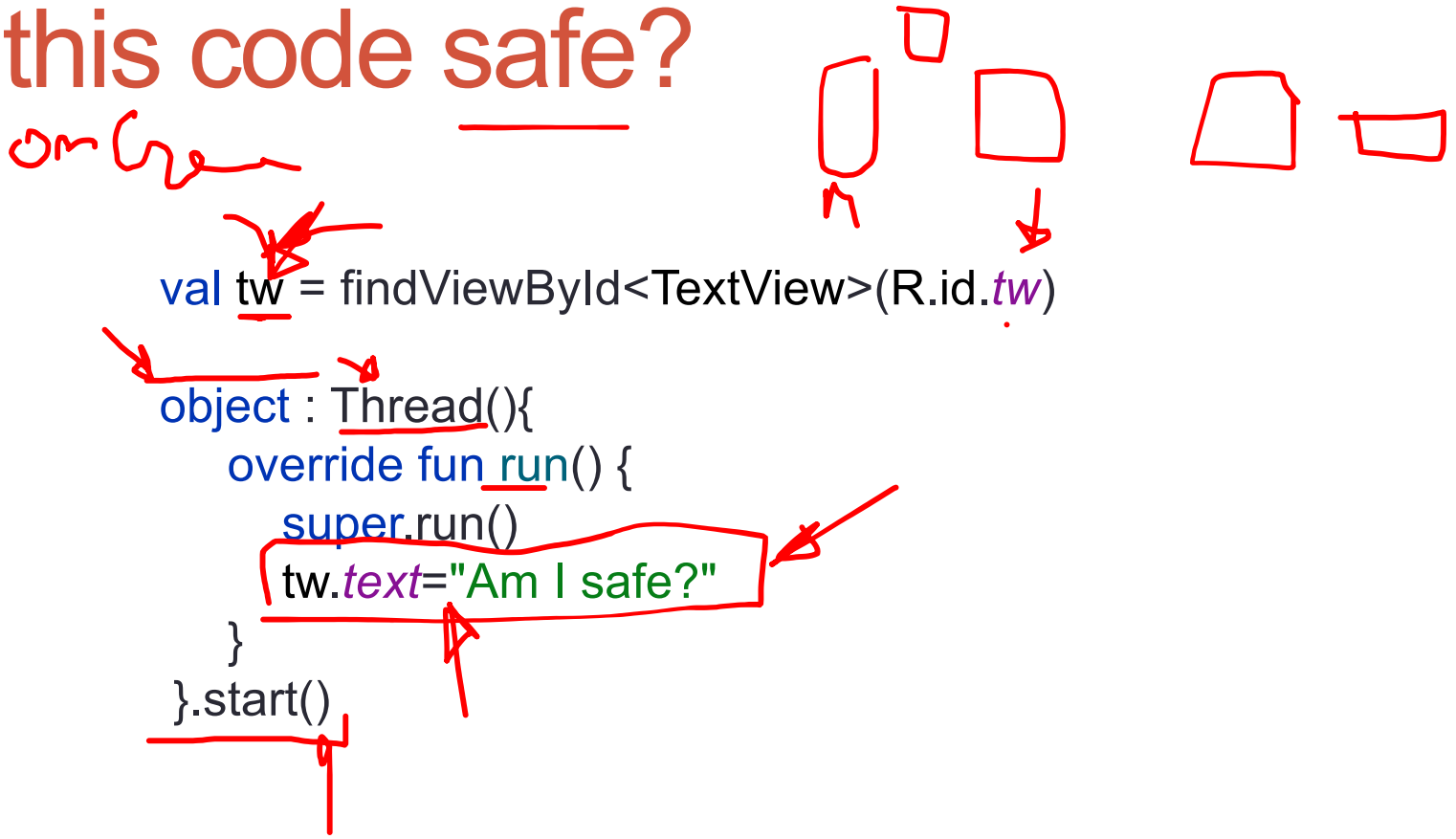




Classic thread usage

- There are basically two main ways to run code concurrently
- One is providing a new class that extends **Thread** and overriding its *run()* method
- The other is providing a new Thread instance with an object that implements the **Runnable** interface as argument during its creation
  - Again, the object must override the run method
- In both cases, the *start()* method must be called to actually execute the code

Is this code safe?

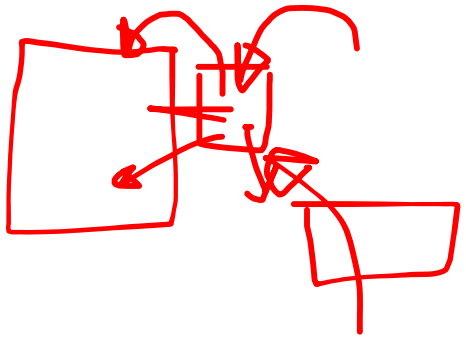


```
val tw = findViewById<TextView>(R.id.tw)

object : Thread(){
    override fun run() {
        super.run()
        tw.text="Am I safe?"
    }
}.start()
```

- No, because the thread updates the textview

How to make the code main-safe



```
object : Thread(){
    override fun run() {
        super.run()
        Handler(Looper.getMainLooper()).post {tw.text="Safe!"}
    }
}.start()
```

Correct code

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val tw = findViewById<TextView>(R.id.tw)

    Log.i("info", "onCreate: "+Thread.currentThread().name)

   object : Thread(){
       override fun run() {
           super.run()
           Log.i("info", "Thread: "+currentThread().name)
           Handler(Looper.getMainLooper()).post { Log.i("info", "run: "+currentThread().name) }
       }
    }.start()
```

Log

```
onCreate: main
I/info: Thread: Thread-2
I/info: run: main
```

UI upate

- How a worker thread can update the UI?
- As UI thread is not thread-safe, the solution is either
- Allow the worker tread to run the update code in main thread
- Communicate the new value to the main thread, which then update the UI

Scheduling primitives on UI messageQueue

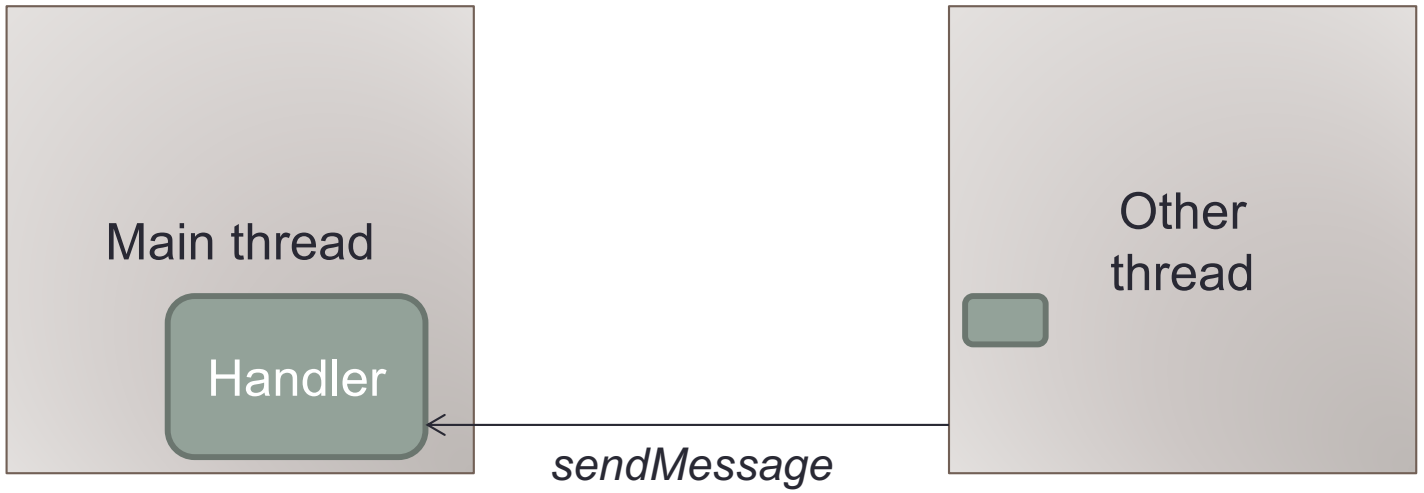
- Scheduling a runnable is accomplished with the methods (of the Handler class):
  - *post(Runnable),*
  - *postAtTime(Runnable, long),*
  - *postDelayed(Runnable, long)*
  - *postAtFrontQueue(Runnable)*

- Sending a message is accomplished with methods:
  - *sendEmptyMessage(int),*
  - *sendMessage(Message),*
  - *sendMessageAtTime(Message, long)*
  - *sendMessageDelayed(Message, long)*

- a Message object can contain a bundle of data

Periodic polling with bcast receiver

Using messages



- The UI thread creates a Handler internal object
- The working thread uses this object to send a message to the UI thread

Example code

```
val h = object : Handler(Looper.getMainLooper()){
    override fun handleMessage(msg: Message) {
        super.handleMessage(msg)
        tw.text="msg received"
    }
}

object : Thread(){
    override fun run() {
        super.run()
        h.sendMessageDelayed (Message(),10000L)
    }
}.start()
```

Reaching the message queue

- *Activity.runOnUIThread(Runnable)* method

Coroutine

- Kotlin simplifies asynchronous programming by coroutines
- A coroutine runs inside a scope
- A coroutine is hosted inside a thread
- A thread can host many coroutines
- Coroutines are implemented in user space (it's a library)

```
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.9'
```

Coroutine

- The key features of a coroutine are that
- A coroutine can be suspended without the hosting thread goes to sleep
- A coroutine can be resumed
- A coroutine can be migrated from a scope to other
- A coroutine is cancelled as soon as its scope is cancelled

Suspend vs sleep

- A coroutine can be suspended for a given amount of time or until an event happens
- This doesn’t block the hosting thread, i.e. other coroutines will continue to run

Why coroutines ?

- It simplifies programming as one can avoid to use callbacks
  - Less errors
- There are scopes that are lifecycle-aware

Coroutine by example: Scope and dispatcher

Launches the code into a thread
Global scope means the
coroutine ends as soon as the app ends

```
GlobalScope.launch { }
```

```
GlobalScope.launch(Dispatchers.Main) {
    Log.i(TAG, "GlobalScope: "+Thread.currentThread().name)
}
```

Dispatcher allows to specify which thread
will host the coroutine

I/info: GlobalScope: main

```
GlobalScope.launch(newSingleThreadContext("my")) {
    Log.i(TAG, "GlobalScope: "+Thread.currentThread().name)
}
```

A new thread can be created

Change dispatcher

- Coroutines can be moved from one thread to another
- This is useful, for example to update the UI

```
GlobalScope.launch(Dispatchers.IO) {
    //Make a long network call
    Log.i(TAG, "GlobalScopeIO: "+Thread.currentThread().name)
    withContext(Dispatchers.Main){
        Log.i(TAG, "GlobalScopeMAIN: "+Thread.currentThread().name)
    }
```

```
I/info: GlobalScopeIO: DefaultDispatcher-worker-3
I/info: GlobalScopeMAIN: main
```

Example

- As an example, of coroutine usage consider a network call

- Using coroutines is possible to run the code related to the network off the main thread and then update the UI after the reply is received

Example

```
GlobalScope.launch(Dispatchers.Main) {
        val result = get("http://.....")
        tw.text=result

    }

suspend fun get(url:String):String{
    delay(2000L) //Simulate a network call
    return "result.."
}
```

In this example, get() still runs on the main thread, but it suspends the coroutine before it starts the network request.
When the network request completes, get resumes the suspended coroutine **instead of using a callback to notify the main thread**.

An example with retrofit