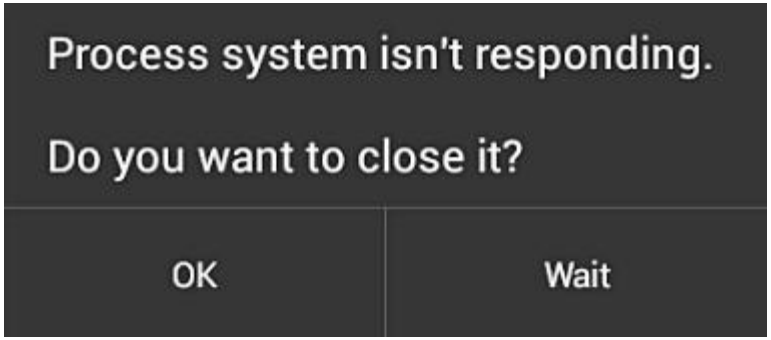
Multithreading techniques

Marcin Mrugas
Piotr Jaszkiewicz

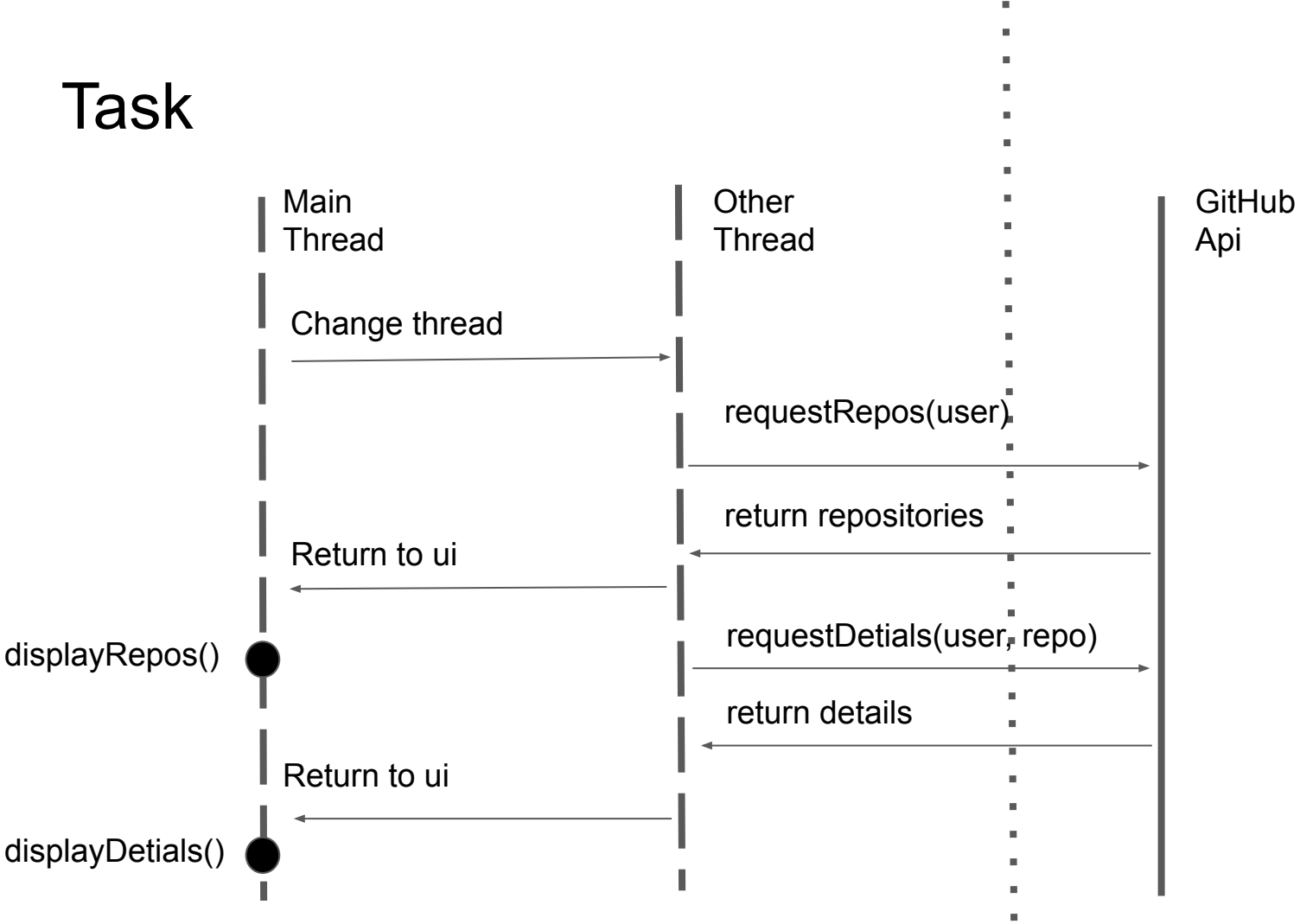
Multithreaded programming

Multithreading in Android

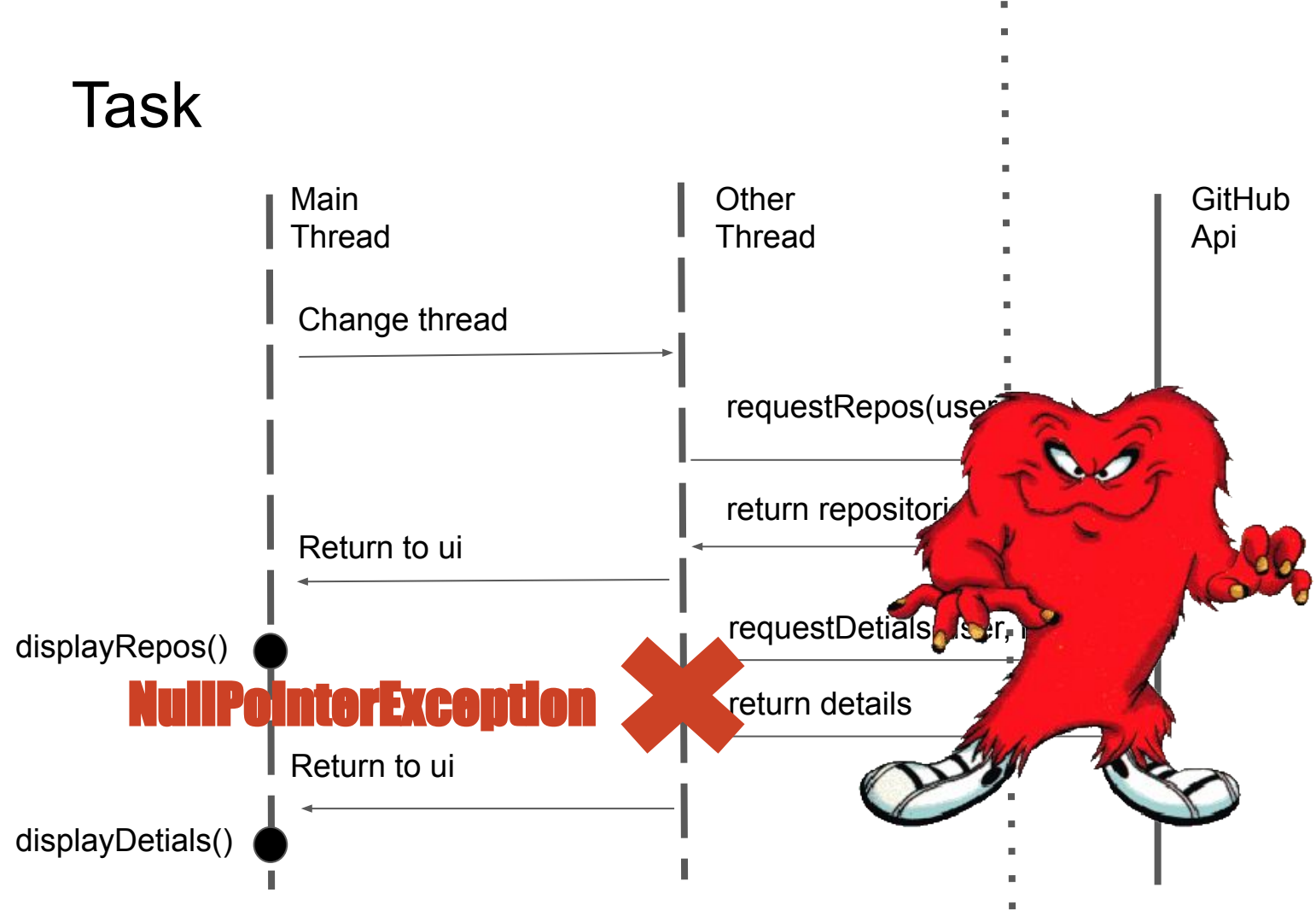
- Network - android.os.NetworkOnMainThreadException od API 3
- Database
- Computation
- IO
- ANR dialog



Task


Main
Thread
Other
Thread
GitHub
Api
Change thread
requestRepos(user)
return repositories
Return to ui
displayRepos()
requestDetials(user, repo)
return details
Return to ui
displayDetials()

Task


Main Thread
Other Thread
GitHub Api
Change thread
requestRepos(user
return repositori
Return to ui
displayRepos()
requestDetials
return details
NullPointerException
Return to ui
displayDetials()

Requests

```
const val BASE_URL = "https://api.github.com"

fun <T> httpRequest(url: String, type: Type): T? {
    val requestUrl = URL(url)
    val conn = requestUrl.openConnection() as HttpURLConnection
    conn.connect()
    if (conn.responseCode == 200) {
        val reader = BufferedReader(InputStreamReader(conn.inputStream))
        return Gson().fromJson<T>(reader, type)
    }
    return null
}

fun requestRepos(user: String): List<Repo>? {
    return httpRequest<List<Repo>>(
        "$BASE_URL/users/$user/repos?sort=updated&direction=desc",
        object : TypeToken<List<Repo>>() {}.type
    )
}

fun requestDetails(user: String, repo: String): Repo? {
    return httpRequest<Repo>(
        "$BASE_URL/repos/$user/$repo",
        Repo::class.java
    )
}
```

Threading

```
Button btThreads = findViewById(R.id.bt_threads);
btThreads.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                List<Repo> repos = requestRepos(user);
                Handler mHandler = new Handler(Looper.getMainLooper());
                mHandler.post(new Runnable() {
                    @Override
                    public void run() {
                        displayRepos(repos);
                    }
                });
                String repoName = repos.get(0).getName();
                Repo details = requestDetails(user, repoName);
                mHandler.post(new Runnable() {
                    @Override
                    public void run() {
                        displayDetails(details);
                    }
                });
            }
```

Threading

```
bt_threads.setOnClickListener {
    Thread(Runnable {
        val repos = requestRepos(user)
        val mHandler = Handler(Looper.getMainLooper())
        mHandler.post {
            displayRepos(repos)
        }
        val repoName = repos?.get(0)?.name ?: return@Runnable
        val details = requestDetails(user, repoName)
        mHandler.post {
            displayDetails(details)
        }
    }).start()
}
```

Threading

```
Thread(Runnable {
     val repos = requestRepos(user)
     runOnUiThread {
         displayRepos(repos)
     }
     val repoName = repos?.get(0)?.name ?: return@Runnable
     val details = requestDetails(user, repoName)
     runOnUiThread {
         displayDetails(details)
     }
 }).start()
```

Threading, handling exceptions

```
thread.setUncaughtExceptionHandler{ ... }
```

Threading

- Threads aren't cheap. Threads require context switches which are costly.
- Threads aren't infinite. The number of threads that can be launched is limited by the underlying operating system. In server-side applications, this could cause a major bottleneck.
- Threads aren't always available. Some platform, such as JavaScript do not even support threads
- Threads aren't easy. Debugging threads, avoiding race conditions are common problems we suffer in multi-threaded programming.

AsyncTask / Callbacks

```
class AsyncTaskWithCallbackReposRequest(
    private val user: String,
    private val callback: (List<Repo>?) -> Unit
) : AsyncTask<Void, Void, List<Repo>?>() {
    override fun doInBackground(vararg params: Void?): List<Repo>? {
        return requestRepos(user)
    }

    override fun onPostExecute(result: List<Repo>?) {
        callback.invoke(result)
    }
}
```

AsyncTask / Callbacks

```
AsyncTaskWithCallbackReposRequest(user) { repos ->
                displayRepos(repos)
                val repoName = repos?.get(0)?.name ?: return@AsyncTas
                AsyncTaskWithCallbackDetailsRequest(user, repoName)
                { details -> displayDetails(details) }.execute()
            }.execute()
```

AsyncTask / Callbacks

```
AsyncTaskWithCallbackReposRequest(user) { repos ->
                displayRepos(repos)
                val repoName = repos?.get(0)?.name ?: return@AsyncTas
                AsyncTaskWithCallbackDetailsRequest(user, repoName)
                { details -> displayDetails(details) }.execute()
            }.execute()
```

AsyncTask Exceptions?

1. `AsyncTask` property holding exception
2. `Try-catch` in `doInBackground()`
3. Store catched exception in property
4. Return exception to caller for handling

Callbacks

- Difficulty of nested callbacks. Usually a function that is used as a callback, often ends up needing its own callback. This leads to a series of nested callbacks which lead to incomprehensible code. The pattern is often referred to as the titled christmas tree (brace represent branches of the tree).
- Error handling is complicated. The nesting model makes error handling and propagation of these somewhat more complicated.

Futures / Promises, Executors

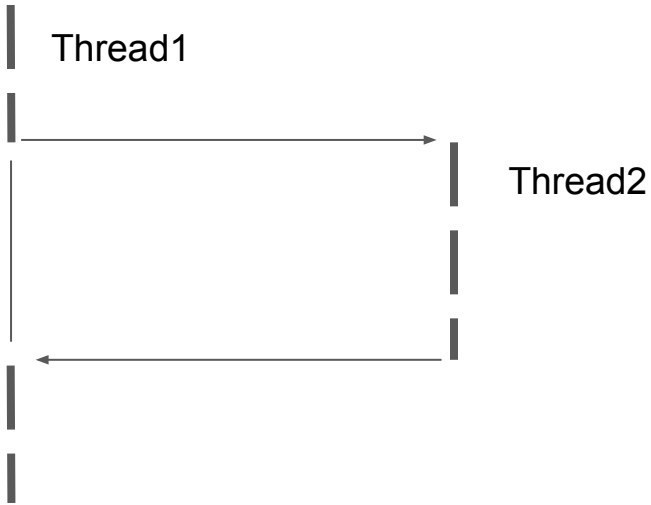
```
public interface Executor {
    void execute(Runnable command);
}
```

- ThreadPoolExecutor,
- **ExecutorService**,
- ScheduledExecutorService,
- ScheduledThreadPoolExecutor,
- ThreadPoolExecutor.

```
val cores = Runtime.getRuntime().availableProcessors()
val executor: ExecutorService = Executors.newFixedThreadPool(cores + 1)
val mainThreadExecutor = MainThreadExecutor()
```

Futures.get()

```
executor.submit(Callable { ... }).get()
```



Futures

```
fun futureRequestRepos(user: String): Future<List<Repo>?> {
    return executor.submit(Callable { requestRepos(user) })
}

fun futureRequestDetails(user: String, repo: String): Future<Repo?> {
    return executor.submit(Callable { requestDetails(user, repo) })
}
```

Futures

```
executor.submit {
    val repos = futureRequestRepos(user).get()
    mainThreadExecutor.execute {
        displayRepos(repos)
    }
    val repoName = repos?.get(0)?.name ?: return@submit
    val topics = futureRequestDetails(user, repoName).get()
    mainThreadExecutor.execute {
        displayDetails(topics)
    }
}
```

Futures

```
executor.submit {
    val repos = futureRequestRepos(user).get()
    mainThreadExecutor.execute {
        displayRepos(repos)
    }
    val repoName = repos?.get(0)?.name ?: return@submit
    val topics = futureRequestDetails(user, repoName).get()
    mainThreadExecutor.execute {
        displayDetails(topics)
    }
}
```

Futures

```
class ExtendedExecutor : ThreadPoolExecutor {
    override fun afterExecute(r: Runnable?, t: Throwable?) {
        super.afterExecute(r, t)
        ...
    }
}
```

CompletableFuture / Promise (js)

```
CompletableFuture
        .supplyAsync(action: (() -> T), executor)
        .thenApply(function: ((T) → U))
        .exceptionally(function: ((Throwable) → Void))
        .thenAccept(function: ((U) → Unit))
```

completableFuture**Async**

```
completableFuture
        .thenApplyAsync(function, executor)
        .exceptionallyAsync(function, executor)
        .thenAcceptAsync(function, executor)
```

CompletableFuture

```
fun promiseRequestRepos(user: String): CompletableFuture<List<Repo>?> {
    return CompletableFuture.supplyAsync { requestRepos(user) }
}

fun promiseRequestDetails(user: String, repo: String): CompletableFuture<Repo?>? {
    return CompletableFuture.supplyAsync { requestDetails(user, repo) }
}
```

CompletableFuture

```
val repositories = promiseRequestRepos(user)
repositories
    .thenAcceptAsync(
        Consumer { t -> displayRepos(t) },
        mainThreadExecutor
    )
repositories
    .thenApplyAsync { repos ->
        repos?.get(0)?.name ?: throw NullPointerException()
    }
    .thenApply { repoName -> promiseRequestDetails(user, repoName) }
    .thenAcceptAsync(
        Consumer { repo -> displayDetails(repo?.get()) },
        mainThreadExecutor
    )
```

CompletableFuture

```
completableFuture: CompletableFuture<U>

completableFuture.orTimeout(timeout: Long, unit: TimeUnit);

completabeFuture.completeOnTimeout(value: V, timeout: Long, unit: TimeUnit)

completableFuture.thenCombine(other: CompletableFuture<T>, function: ((U, T) -> V))

completableFuture.thenCompose(function: ((U) -> CompletableFuture<V>))

completableFuture.exceptionally(function: ((t : Throwable) -> U?))
```

CompletableFuture

- ≥ API 24: `import java.util.concurrent.CompletableFuture`
- < API 24: `import java9.util.concurrent.CompletableFuture`

```
dependencies {
    compile 'net.sourceforge.streamsupport:android-retrofuture:1.7.0'
}
```

Futures / Promises

- Different programming model. Similar to callbacks, the programming model moves away from a top-down imperative approach to a compositional model with chained calls. Traditional program structures such as loops, exception handling, etc. usually are no longer valid in this model.
- Different APIs. Usually there's a need to learn a completely new API such as thenCompose or thenAccept, which can also vary across platforms.
- Specific return type. The return type moves away from the actual data that we need and instead returns a new type Promise which has to be introspected.
- Error handling can be complicated. The propagation and chaining of errors isn't always straightforward.

Reactive Programming

"everything is a stream, and it's observable"

Observables

Non-observable example

```
a = 1;
b = a + 1;

a = 2;

print a; //---> 2
print b; //---> 2
```

Observable example

```
a = 1;
b = a + 1;

a = 2;

print a; //---> 2
print b; //---> 3
```

Streams

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

Schedulers

```
Schedulers.io()
Schedulers.computation()
Schedulers.newThread()
Schedulers.single()
Schedulers.trampoline()
Schedulers.from(executor)
AndroidSchedulers.mainThread()
```

Reactive Programming

Observable + Streams + Schedulers =

Operators

Creating

Create
Defer
Empty/Never/Throw
From
Interval
Just
Range
Repeat
Start
Timer

Transforming

Buffer
FlatMap
GroupBy
Map
Scan
Window

Filtering

Debounce
Distinct
ElementAt
Filter
First
IgnoreElements
Last
Sample
Skip
SkipLast
Take
TakeLast

Combining

And/Then/When
CombineLatest
Join
Merge
StartWith
Switch
Zip

Error Handling

Utility

Conditional and Boolean

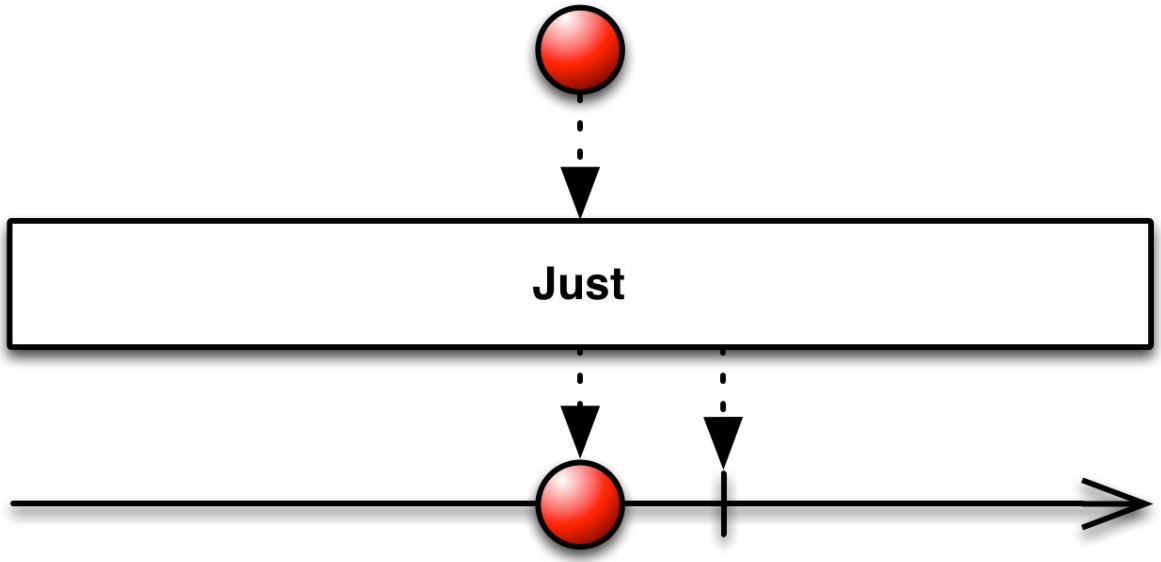
Mathematical

Backpressure

Connectable

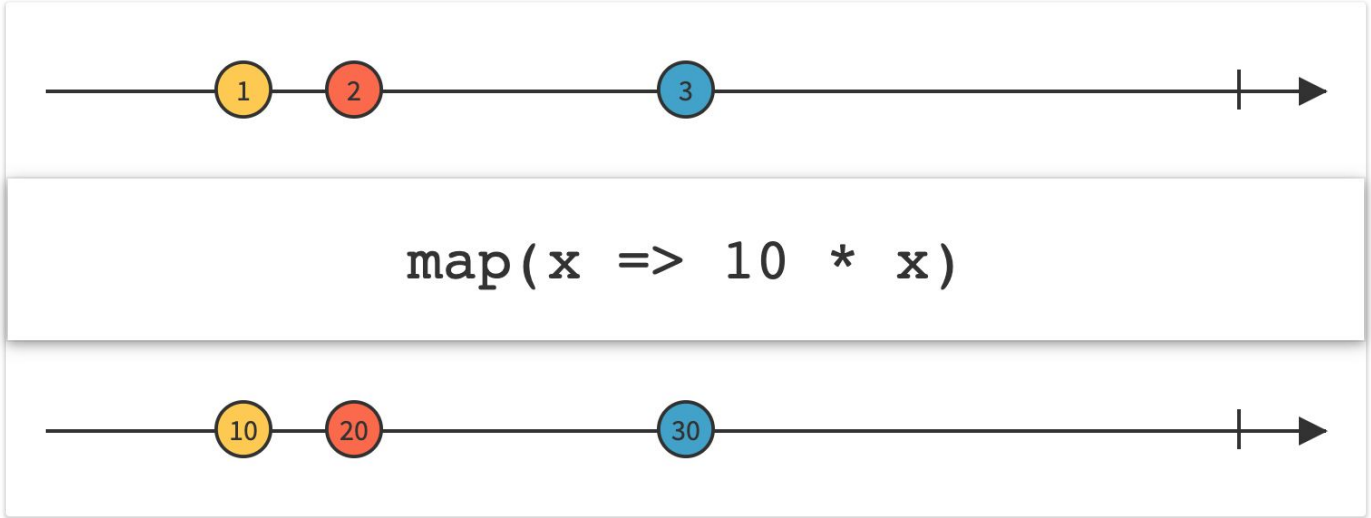
Converting

Just



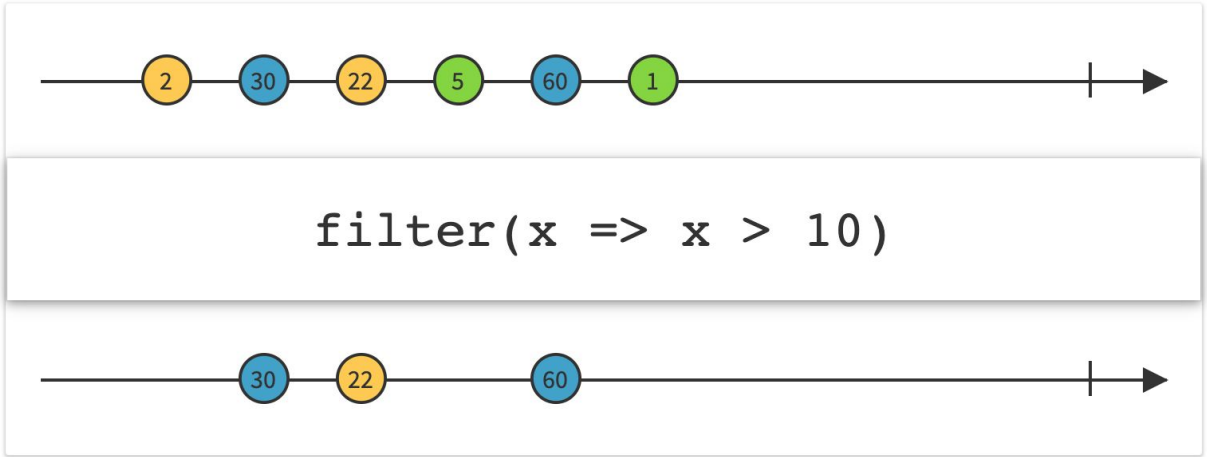

```
Observable
    .just(item)
```

Map




```
Observable
    .fromArray(1, 2, 3)
    .map {item -> item * 10 }
```

Filter




```
Observable
    .fromArray(2, 30, 22, 5, 60, 1)
    .filter { it > 10 }
```

Schedulers once again

```
Observable
    .fromArray(2, 30, 22, 5, 60, 1)
    .filter { it > 10 }
    .subscribe(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe{ Log.d(TAG, it.toString()) }
```

Subscribe

```
.subscribe(onNext: ((T) -> Unit),
           onError: ((Throwable) -> Unit)),
           onComplete: Action)


.subscribe(
              { item ->
                  items.add(item);
                  adapter.notifyDataSetChanged()
              },
              { e -> Toast.makeText(this@MyActivity, e.message, Toast.LENGTH_LONG).show() },
              { progressBar.visibility = View.INVISIBLE }
          )
```

Reactive Programming

```
val repositories = Observable.just(user)
                .map { user -> requestRepos(user) }
                .subscribeOn(Schedulers.io())
```

Reactive Programming

```
    repositories
       .observeOn(AndroidSchedulers.mainThread())
       .subscribe(
            { repos ->
                displayRepos(repos)
            },
            { e -> Toast.makeText(this@MainActivity, e.message,
Toast.LENGTH_LONG).show() }
        )
```

Reactive Programming

```
        repositories
            .map { repos ->
                val repoName = repos[0].name
                requestDetails(user, repoName)
            }
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
                { repo ->
                    displayDetails(repo)
                },
                { e -> Toast.makeText(this@MainActivity, e.message,
Toast.LENGTH_LONG).show() }
            )
```

Reactive Programming

- In approach it's quite similar to Futures, but one can think of a Future as returning a discrete element, whereby Rx returns a stream. However, similar to the previous, it also introduces a complete new way of thinking about our programming model.
- This implies a different way to approach problems and quite a significant shift from what we're using to when writing synchronous code. One benefit as opposed to Futures is that given its ported to so many platforms, generally we can find a consistent API experience no matter what we use it, be it C#, Java, JavaScript, or any other language where Rx is available.
- In addition, Rx does introduce a somewhat nicer approach to error handling.

Coroutines

- Easy to read (no callbacks)
- Light-weight

Coroutines are light-weight

Threads

```
for (i in 1..10000) {
    val a = thread(start = true) {
        Thread.sleep(1000)
        println(i)
    }
    threads.add(a)
}
```

Coroutines

```
for (i in 1..10000) {
    val job = GlobalScope.launch {
        delay(1000)
        println(i)
    }
    jobs.add(job)
}
```

Blocking

```
fun loadRepos() {
    val repos = requestRepos()
    displayRepos(repos)
}
```

Callback

```
fun loadRepos() {
    api.requestRepos() { repos ->
        displayRepos(repos)
    }
}
```

Coroutines

```
suspend fun loadRepos() {
    val repos = requestRepos()
    displayRepos(repos)
}
```

Coroutines

```
suspend fun loadRepos() {
    val repos = requestRepos()
    displayRepos(repos)
}
```

```
fun loadRepos() {
    val repos = requestRepos()
    displayRepos(repos)
}
```

```
suspend fun loadRepos() {
    val repos = requestRepos()
    displayRepos(repos)
}
```

```
suspend fun requestRepos(): List<Repo> {
    return withContext(Disptachers.IO) {
        return classicHttpRequest()
    }
}
```

Dispatchers

- `Dispatchers.Main` – the UI thread
- `Dispatchers.IO` – for blocking tasks, like database access, file writing, etc.
- `Dispatchers.Default` – for CPU intensive work

```
suspend fun loadRepos() {
    ...
}
```

```
class MainActivity : AppCompatActivity(), CoroutineScope by MainScope() {

    private fun coroutinesRequest() {
        launch {
            loadRepos()
        }
    }

}
```

```
class MainActivity : ... {

    override fun onDestroy() {
        super.onDestroy()
        cancel()
    }

}
```

CoroutineScope

```
val request = launch {
    GlobalScope.launch {
        delay(1000)
        println("job1")
    }
    launch {
        delay(1000)
        println("job2")
    }
}
delay(500)
request.cancel()
delay(1000)
```

Coroutine Builders

- launch
- withContext

Async

```
val a = async {
    delay(1000)
    return@async 6
}
val b = async {
    delay(500)
    return@async 7
}
println("The answer is
${a.await() * b.await()}")
```

Exception handling

```
val deferred = GlobalScope.async {
    throw ArithmeticException()
}
try {
    deferred.await()
} catch (e: ArithmeticException) {
    println("Caught ArithmeticException")
}

GlobalScope.launch {
    throw IndexOutOfBoundsException()
}
```

Exception handling

```
val deferred = async {
    throw ArithmeticException()
}
try {
    deferred.await()
} catch (e: ArithmeticException) {
    println("Caught ArithmeticException")
}

launch {
    throw IndexOutOfBoundsException()
}
```

Exception handling

```
val handler = CoroutineExceptionHandler { _, exception ->
   println("Caught $exception")
}

val deferred = GlobalScope.async(handler) {
   throw ArithmeticException()
}

GlobalScope.launch(handler) {
   throw IndexOutOfBoundsException()
}
```

Integrations

- CompletableFuture
- RxJava2
- Retrofit
- LiveData
- ...

Integrations

```
class LibraryClass {
    fun libraryCall(onSuccess: (Int) -> Unit,
                    onError: (Exception) -> Unit) {
        ...
    }
}

suspend fun LibraryClass.await(): Int {
    return suspendCoroutine { cont: Continuation<Int> ->
        libraryCall(
            onSuccess = { cont.resume(it) },
            onError = { cont.resumeWithException(it) }
        )
    }
}
```

Channels (experimental)

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x * x)
    channel.close()
}
for (y in channel) println(y)
```

Coroutines

- The function signature remains exactly the same. The only difference is suspend being added to it. The return type however is the type we want returned.
- The code is still written as if we were writing synchronous code, top-down, without the need of any special syntax, beyond the use of a function called launch which essentially kicks-off the coroutine.
- The programming model and APIs remain the same. We can continue to use loops, exception handling, etc. and there's no need to learn a complete set of new APIs
- It is platform independent. Whether we targeting JVM, JavaScript or any other platform, the code we write is the same. Under the covers the compiler takes care of adapting it to each platform.

DEMO

Sample Project

https://github.com/mrugacz95/coroutines

Bibliografia

- https://kotlinlang.org/docs/tutorials/coroutines/async-programming.html
- https://developer.android.com/reference/android/os/AsyncTask
- https://developer.android.com/reference/java/util/concurrent/Future
- https://www.deadcoderising.com/java8-writing-asynchronous-code-with-completablefuture/
- https://branch-blog.qlik.com/what-is-reactive-programming-a1e82cf28575
- https://kotlinlang.org/docs/reference/coroutines/coroutines-guide.html
- https://www.youtube.com/watch?v=BOHK_w09pVA
- https://stackoverflow.com/questions/38221673/completablefuture-in-the-android-support-library
- https://gist.github.com/staltz/868e7e9bc2a7b8c1f754
- https://medium.com/upday-devs/rxjava-subscribeon-vs-observeon-9af518ded53a