Using Flow for Everything Async

Quick Run

Async Tasks

Simple use case : I have created a new Activity and I want to load an image inside an ImageView while allowing the user to scroll up and down.

```
public class MyUIActivity extends AppCompatActivity {
    private ImageView mImageView = view.findViewById(R.id.someView);

    //Usage
    //- inside of a function of from a onClick
        new DownloadImageTask().execute("http://example.com/image.png");

    private class DownloadImageTask extends AsyncTask<String, Void, Bitmap> {
        protected  Bitmap doInBackground(String... urls) {
            return loadImageFromNetwork(urls[0]);
        }
        protected void onPostExecute(Bitmap result) {
            mImageView.setImageBitmap(result);
        }
    }
}
```

Callbacks - Some call it the Hollywood principle: “Don’t call us we call you”

```
public class User {
    private int id;
    private String email;
    private String username;
}

...

Call<User> call = userService.login();
call.enqueue(new Callback<User>() {
    @Override
    public void onResponse(Call<User> call, Response<User> response) {
        // todo deal with returned data (user)
    }

    public void onFailure(Call<User> call, Throwable t) {
        // todo deal with the failed network request
    }
});
```

Rx

- To simply pass bits of information from one activity or class to another.
- Easy to set up communication between multiple threads
- You can create asynchronous data stream on any thread, transform the data and consume it by an Observer on any thread.

```
public Single<List<Restaurant>> getRestaurants(int userId) {
  return ddApi.getUserInfo(userId).flatMap(user -> {
    return ddApi.getAvailableRestaurants(user.defaultAddress.lat, user.defaultAddress.lng);
  });
}
```

Rx

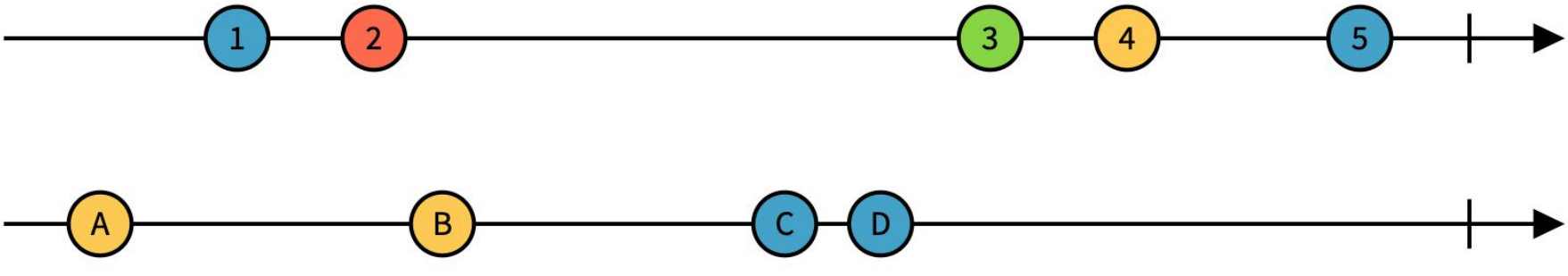
```
public class RestaurantFragment {

  @Override
  public void onResume() {
    // subscribe to the Single returned by RestaurantApi
    restaurantDataSource
      .getRestaurants(userId)
      .subscribeOn(Schedulers.io())
      .observeOn(AndroidSchedulers.mainThread())
      .subscribe(new SingleObserver<Restaurant>() {
            @Override
            public void onSuccess(List<Restaurant> Restaurants) {
                // update the adapter with restaurants
            }

            @Override
            public void onError(Throwable e) {
                // display an error message
            }
        });
  }
}
```

Rx - With Room

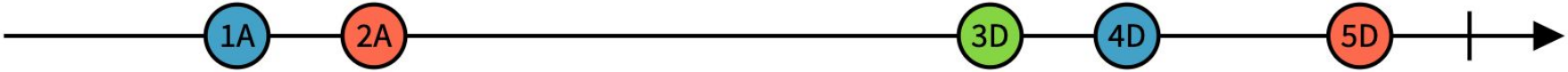
- Create entity class
- Create DAO
- Add AppDatabase.class

```
class UserRepository(val userApi: UserApi, val userDao: UserDao) {

    fun getUsers(): Observable<List<User>> {
        return Observable.concatArray(
                getUsersFromDb(),
                getUsersFromApi())
    }


    fun getUsersFromDb(): Observable<List<User>> {
        return userDao.getUsers().filter { it.isNotEmpty() }
                .toObservable()
                .doOnNext {
                    Timber.d("Dispatching ${it.size} users from DB...")
                }
    }

    fun getUsersFromApi(): Observable<List<User>> {
        return userApi.getUsers()
                .doOnNext {
                    Timber.d("Dispatching ${it.size} users from API...")
                    storeUsersInDb(it)
                }
    }

    fun storeUsersInDb(users: List<User>) {
        Observable.fromCallable { userDao.insertAll(users) }
                .subscribeOn(Schedulers.io())
                .observeOn(Schedulers.io())
                .subscribe {
                    Timber.d("Inserted ${users.size} users from API in DB...")
                }
    }

}
```

Rx




```
withLatestFrom((x, y) => "" + x + y)
```



Rx

Coroutines - A coroutine is a way to handle concurrency tasks in a thread

```
interface RetrofitService {
    @GET("/posts")
    suspend fun getPosts(): Response<List<Post>>
}
```

```
object RetrofitFactory {
    const val BASE_URL = "https://jsonplaceholder.typicode.com"

    fun makeRetrofitService(): RetrofitService {
        return Retrofit.Builder()
                .baseUrl(BASE_URL)
                .addConverterFactory(MoshiConverterFactory.create())
                .build().create(RetrofitService::class.java)
    }
}
```

```
class MainActivity : AppCompatActivity() {

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

    val service = RetrofitFactory.makeRetrofitService()
    CoroutineScope(Dispatchers.IO).launch {
        val response = service.getPosts()
        withContext(Dispatchers.Main) {
            try {
                if (response.isSuccessful) {
                    //Do something with response e.g show to the UI.
                } else {
                    toast("Error: ${response.code()}")
                }
            } catch (e: HttpException) {
                toast("Exception ${e.message}")
            } catch (e: Throwable) {
                toast("Ooops: Something else went wrong")
            }
        }
    }
}
```

Coroutines

```
@Dao
interface UsersDao {

    @Query("SELECT * FROM users")
    suspend fun getUsers(): List<User>

    @Query("UPDATE users SET age = age + 1 WHERE userId = :userId")
    suspend fun incrementUserAge(userId: String)

    @Insert
    suspend fun insertUser(user: User)

    @Update
    suspend fun updateUser(user: User)

    @Delete
    suspend fun deleteUser(user: User)

}
```

```
class MyViewModel(private val usersDao : UsersDao) : ViewModel() {
   fun insert(user : User) = launch(Common) {
        usersDao.insert(user)
   }
}
```

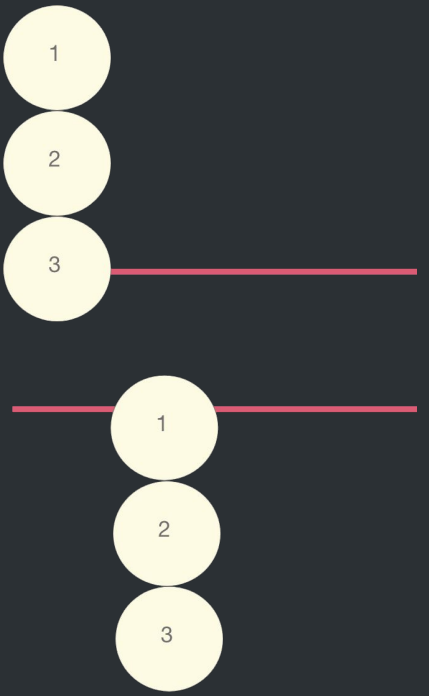
Flowing Asynchronously

Why Flow?

Problem Statement: Return a list of integers

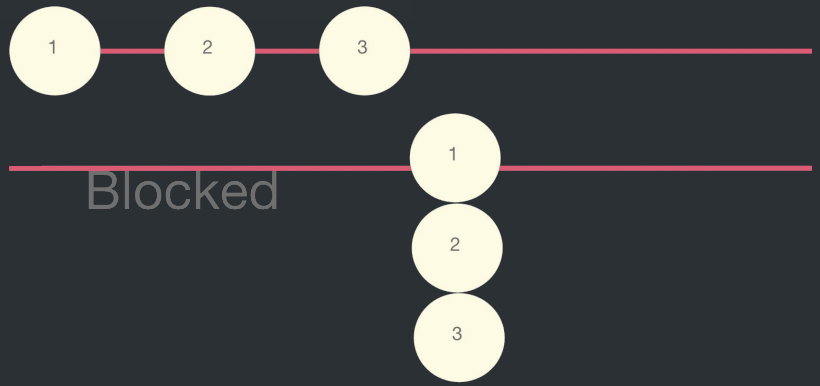
Collections

```
fun main() {
    generateIntegers().forEach { value -> println(value) }
}


fun generateIntegers(): Collection<Int> {
    return listOf(1, 2, 3)
}
```



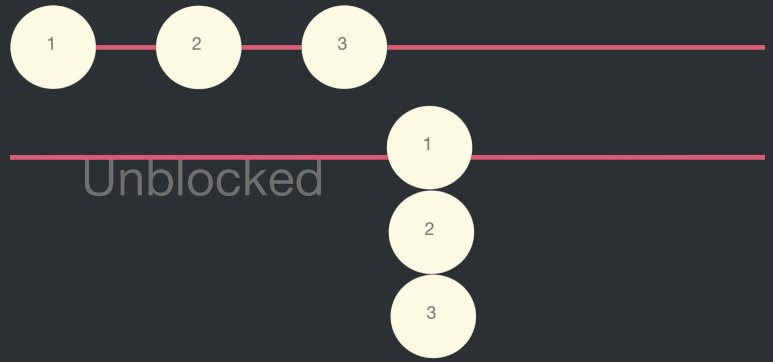
Sequence

```
fun main() {
    generateIntegers().forEach { value -> println(value) }
}

fun generateIntegers(): Sequence<Int> = sequence { this: SequenceScope<Int>
    for (i in 1..3) {
        Thread.sleep(100) // pretend we are computing it
        yield(i) // yield next value
    }
}
```



Coroutines

```
suspend fun generateIntegers(): List<Int> {
    delay( timeMillis: 1000) // pretend we are doing something asynchronous here
    return listOf(1, 2, 3)
}

fun main() = runBlocking { this: CoroutineScope
    generateIntegers().forEach { value -> println(value) }
}
```



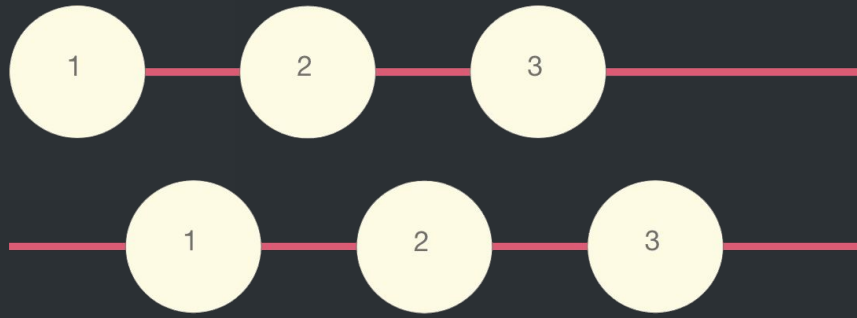
Flows

```
fun generateIntegers(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay( timeMillis: 100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> { this: CoroutineScope
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch { this: CoroutineScope
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay( timeMillis: 100)
        }
    }
    // Collect the flow
    generateIntegers().collect { value -> println(value) }
}
```

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

Flows

```
fun generateIntegers(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay( timeMillis: 100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> { this: CoroutineScope
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch { this: CoroutineScope
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay( timeMillis: 100)
        }
    }
    // Collect the flow
    generateIntegers().collect { value -> println(value) }
}
```



Flows

Flows are cold streams

Flows

Flows are cold streams

```
fun generateIntegers(): Flow<Int> = flow { this: FlowCollector<Int>
    println("Flow started")
    for (i in 1..3) {
        delay( timeMillis: 100)
        emit(i)
    }
}


fun main() = runBlocking { this: CoroutineScope
    println("Calling foo...")
    val flow = generateIntegers()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
```

```
Calling foo...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

Flows

Flow Cancellation

Flows

Flow Cancellation

```
private fun generateIntegers(): Flow<Int> = flow { this: FlowCollector<Int>
    delay( timeMillis: 100)
    emit( value: 1)
    delay( timeMillis: 100)
    emit( value: 2)
    delay( timeMillis: 100)
    emit( value: 3)
}

@ExperimentalCoroutinesApi
private fun main() {
    generateIntegers()
        .onEach { value ->
            log("$value")
        }
        .catch { cause ->
            log("exception : ${cause.message}")
        }
        .onCompletion { log("done") }
        .launchIn( scope: this)
}

override fun onStop() {
    super.onStop()
    job.cancel()
}
```

Suspending and cancellable function

Flows

Common Operators and Builders

Flows

Builders

```
fun generateIntegers(): Flow<Int>  {
    return (1..3).asFlow()
}
```

```
fun generateIntegers(): Flow<Int> = flow { this: FlowCollector<Int>
    for (i in 1..3) {
        emit(i)
    }
}
```

Flows

Operators

- Intermediate Operators
  - Map
  - Filter
  - Transform
- Size Limiting Operators
  - Take
- Terminal Operators
  - toList / toSet
  - Reduce
  - fold

```
suspend fun performRequest(request: Int): String {
    delay( timeMillis: 1000) // imitate long-running asynchronous work
    return "response $request"
}

@FlowPreview
fun main() = runBlocking<Unit> { this: CoroutineScope
    (1..3).asFlow() // a flow of requests
        .transform { request ->
            emit( value: "Making request $request")
            emit(performRequest(request))
        }
        .collect { response -> println(response) }
}
```

Flows

flowOn

```
fun foo(): Flow<Int> = flow { this: FlowCollector<Int>
    for (i in 1..3) {
        Thread.sleep( millis: 100) // pretend we are computing it in CPU-consuming way
        log( msg: "Emitting $i")
        emit(i) // emit next value
    }
}.flowOn(Dispatchers.Default) // RIGHT way to change context for CPU-consuming code in flow builder

fun main() = runBlocking<Unit> { this: CoroutineScope
    foo().collect { value ->
        log( msg: "Collected $value")
    }
}
```


BEAR WITH US.

Flows

Merging Flows

Flows

Error Handling

Flows

Everything is caught

```
fun foo(): Flow<String> =
    flow { this: FlowCollector<Int>
        for (i in 1..3) {
            println("Emitting $i")
            emit(i) // emit next value
        }
    }
        .map { value ->
            check( value: value <= 1) { "Crashed on $value" }
            "string $value" ^map
        }

fun main() = runBlocking<Unit> { this: CoroutineScope
    try {
        foo().collect { value -> println(value) }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

```
Emitting 1
string 1
Emitting 2
Caught java.lang.IllegalStateException: Crashed on 2
```

Flows

Catching Declaratively

```
suspend fun main() {
    generateIntegers()
        .onEach { value ->
            check( value: value <= 1) { "Collected $value" }
            println(value)
        }
        .catch { e -> println("Caught $e") }
        .collect()
}
```

Flows

Flow Completion

Flows

Try / catch finally block

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    try {
        generateIntegers().collect { value -> println(value) }
    } finally {
        println("Done")
    }
}
```

Flows

onCompletion

```
fun generateIntegers(): Flow<Int> = flow { this: FlowCollector<Int>
    emit( value: 1)
    throw RuntimeException()
}

@ExperimentalCoroutinesApi
fun main() = runBlocking<Unit> { this: CoroutineScope
    generateIntegers()
        .onCompletion { cause -> if (cause != null) println("Flow completed exceptionally") }
        .catch { cause -> println("Caught exception") }
        .collect { value -> println(value) }
}
```

```
1
Flow completed exceptionally
Caught exception
```

Flows

Launching Flow

Flows

Blocking Launch

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    generateIntegers()
        .onEach { event -> println("Event: $event") }
        .collect() // <--- Collecting the flow waits
    println("Done")
}
```

```
Event: 1
Event: 2
Event: 3
Done
```

Flows

launchIn

```
fun main() = runBlocking<Unit> { this: CoroutineScope
    generateIntegers()
        .onEach { event -> println("Event: $event") }
        .launchIn( scope: this) // <--- Launching the flow in a separate coroutine
    println("Done")
}
```

```
Done
Event: 1
Event: 2
Event: 3
```

Flow in Action

Flows

LocationManager.kt

```
suspend fun getLastKnownLocation(): Location? = suspendCoroutine { it: Continuation<Location?>
    fusedLocationClient.lastLocation.addOnSuccessListener { location ->
        it.resumeWith(Result.success(location))
    }.addOnFailureListener { it: Exception
        Result.failure<Exception>(it)
    }
}

fun getLocationUpdates(): Flow<Location?> = flow { this: FlowCollector<Location?>
    coroutineScope { this: CoroutineScope
        while (isActive) {
            emit(getLastKnownLocation())
            delay( timeMillis: 1000)
        }
    }
}
```

Flows

MainActivity.kt

```
private fun setupLocation() {
    locationManager = LocationManager( activity: this)
    locationManager.getLocationUpdates()
        .flowOn(Dispatchers.Default)
        .onEach { location ->
            log("${System.currentTimeMillis()} :: flow location : ${location?.latitude}, ${location?.longitude}")
        }
        .catch { cause ->
            log("exception : ${cause.message}")
        }
        .onCompletion { log("done") }
        .launchIn( scope: this)
}
```

Video


IT'S OVER
THIS MIND NUMBING POWERPOINT PRESENTATION
IS FINALLY OVER