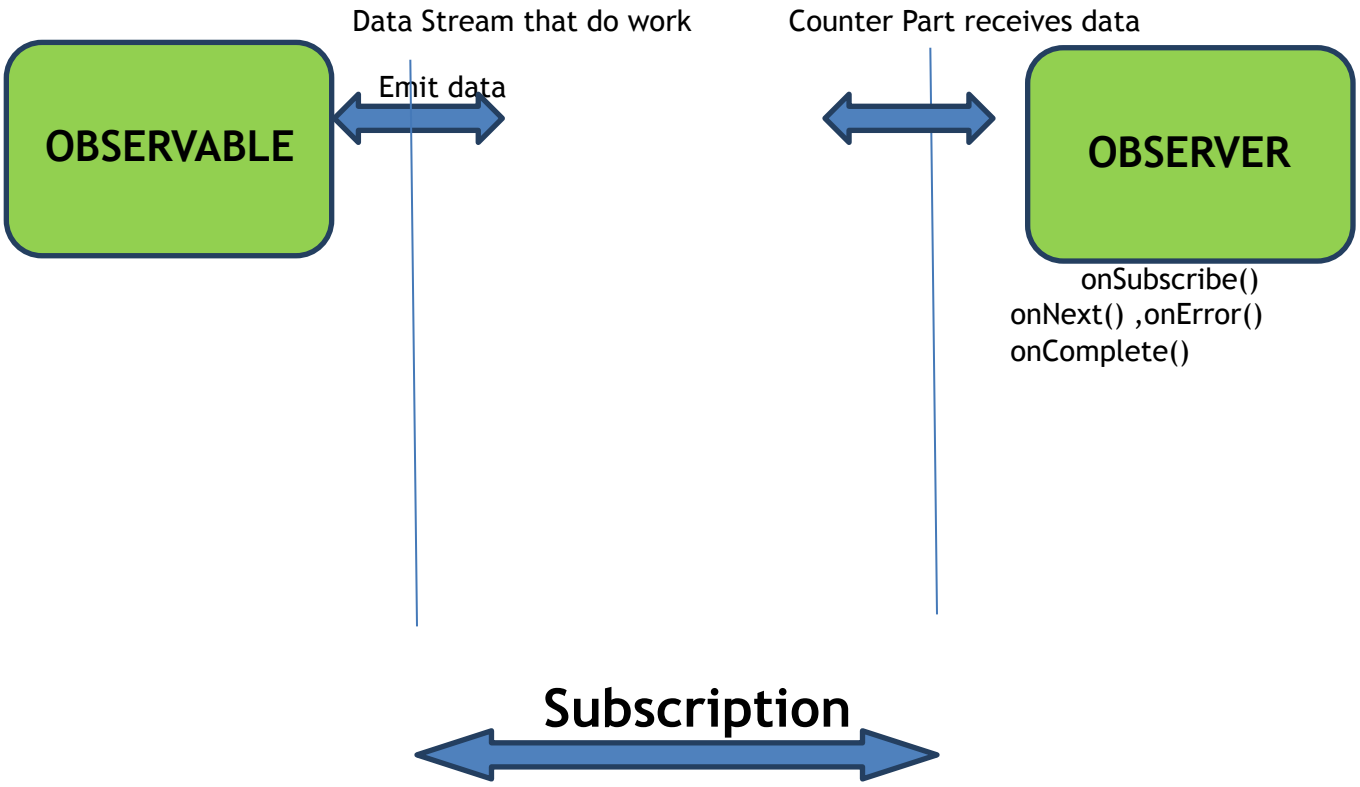
RXJAVA

REACTIVE PROGRAMMING :

As We discussed before in my Rxjava basic tutorial what is reactive programming and why we use that it interesting part of learning as first level so if give in one line definition that it is event based async programming , data stream as async which can be observed and we take action on it . and use it as background process .. here every task run on its thread , if we discuss the tasks every task run it in own thread and can start simultaneously and complete time is based on longer task.

And as we discuss before Rxjava is java implementation of Reactive Extension and RxAndroid is related to Android platform with few added classes of Rxjava .

If We go for more detail in Rxjava there are two key component of Rxjava Observable and Observer



Bonding between Observable and Observer

Single or Multiple

In Simple Way

The build blocks for RxJava code are the following:

observables representing sources of data

subscribers (or observers) listening to the observables

a set of methods for modifying and composing the data

An observable emits items; a subscriber consumes those items.

SCHEDULERS

Schedulers : They Decide the thread on which Observable emit the data and on which on which observer we receives it .

Two basics are Schedulers.io() and AndroidSchedulers.mainThread() beside this we have more schedulers that I will show you how to use they are

Schedulers.newThread(),Schedulers.computation(),Schedulers.single(),Schedulers.immediate(),

Schedulers.trampoline(),Schedulers.from()

The main is Schedulers.io() -> CPU intensive operations – network calls reading disc files database operation as I show you in room with rxjava tutorial

And AndroidSchedulers.mainThread() – called mainThread or UI Thread -> updating UI

And Schedulers.newThread() is for whenever we create a newThread each time task schedule

Schedulers.computation() use to perform CPU intensive operation like processing huge data or bitmap process

And Schedulers.single is used for execute all tasks in sequential order

And if you want to do some task in immediate responding by blocking the main thread .

And Schedulers.trampoline is used FIFO manners first in first out

And Schedulers.from() -> by limiting number of threads

implementation ‘io.reactivex.rxjava2:rxjava:2.0.1’

// RxAndroid

implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'

Flowable<T>

Emits 0 or n items and terminates with an success or an error event. Supports backpressure, which allows to control how fast a source emits items.

Observable<T>

Emits 0 or n items and terminates with an success or an error event.

Single<T>

Emits either a single item or an error event. The reactive version of a method call.

Maybe<T>

Succeeds with an item, or no item, or errors. The reactive version of an Optional.

Completable

Either completes with an success or with an error event. It never emits items. The reactive version of a Runnable.

Again will describe you one by one by examples

```
Observable<String> namesobservable = Observable.just("Vishal", "John", "Mark", "Sanjay", "Rita");
Observer<String> namesobserver = getnamesObserver();

private Observer<String> getnamesObserver() {
        return new Observer<String>() {
            @Override
            public void onSubscribe(Disposable d) {
                Log.d(TAG, "onSubscribe");
            }

            @Override
            public void onNext(String s) {
                Log.d(TAG, "Name: " + s);
            }
```

```
        @Override
        public void onError(Throwable e) {
            Log.e(TAG, "onError: " + e.getMessage());
        }

        @Override
        public void onComplete() {
            Log.d(TAG, “Successfully get Items!");
        }
    };
}
```

```
namesobservable
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(namesobserver);
```

Disposable: Disposable is used to dispose the subscription when an Observer no longer wants to listen to Observable. In android disposable are very useful in avoiding memory leaks.

To avoid memory leaks

```
 private Disposable disposable;
public void onSubscribe(Disposable d) {
            Log.d(TAG, "onSubscribe");
            disposable = d;
        }
protected void onDestroy() {
    super.onDestroy();

    // don't send events once the activity is destroyed
    disposable.dispose();
  }
```

Now will Explore more how we can use other observable

Different Types of Observables

Single

Single is an Observable that always emit only one value or throws an error. A typical use case of Single observable would be when we make a network call in Android and receive a response.

The Single Observer always emits only once so there is no onNext() .Single emits only one value and applying some of the operator makes no sense. Like we don’t want to take value and collect it to a list.

Maybe

Maybe is an Observable that may or may not emit a value. For example, we would like to know if a particular user exists in our db. The user may or may not exist.

Flowable :

Flowable can be regarded as a special type of Observable (but internally it isn't). It has almost the same method signature such as the Observable as well.

Flowable<T>

Emits 0 or n items and terminates with an success or an error event. Supports backpressure, which allows to control how fast a source emits items.

As We discussed Observable emit the items .

The difference is that Flowable allows you to process items that emitted faster from the source than some of the following steps can handle.

Assume that you have a source that can emit a million items per second. However, the next step uses those items to do a network request.

When to use Observable

- **You have a flow of no more than 1000 elements at its longest: i.e.**
- **You deal with GUI events such as mouse moves or touch events: these can rarely be backpressured reasonably and aren't that frequent. You may be able to handle an element frequency of 1000 Hz or less with Observable.**

When to use Flowable

- **Dealing with 10k+ of elements that are generated in some fashion somewhere and thus the chain can tell the source to limit the amount it generates.**
- **Reading (parsing) files from disk is inherently blocking and pull-based which works well with back pressure as you control, for example, how many lines you read from this for a specified request amount).**
- **Big Files where back pressure needed.**

RxJava 2 introduced a clear distinction between these two kinds of sources – back pressure-aware sources are now represented using a dedicated class – *Flowable*.

***Observable* sources don't support back pressure.**

Also, if we're dealing with a big number of elements, two possible problems connected with back pressure can occur depending on the type of the *Observable*.

Dealing with possibly infinite streams is very challenging, as we need to face a problem of a backpressure.

it's not difficult to get into a situation in which an *Observable* is emitting items more rapidly than a subscriber can consume them.

Flowable comes to picture when there is a case that the Observable is emitting huge numbers of values which can't be consumed by the Observer.

In this case, the Observable needs to skip some values on the basis of some strategy else it will throw an exception.

The Flowable Observable handles the exception with a strategy.The strategy is called **BackPressureStrategy** and the exception is called **MissingBackPressureException**.

In JAVA

Cold *Observables*

A cold *Observable* emits a particular sequence of items but can begin emitting this sequence when its *Observer* finds it to be convenient, and at whatever rate the *Observer* desires, without disrupting the integrity of the sequence. **Cold *Observable* is providing items in a lazy way**

The *Observer* is taking elements only when it is ready to process that item, and items do not need to be buffered in an *Observable* because they are requested in a pull fashion.

For example, if you create an *Observable* based on a static range of elements from one to one million, that *Observable* would emit the same sequence of items no matter how frequently those items are observed:

Hot Observables

A hot *Observable* begins generating items and emits them immediately when they are created. It is contrary to a Cold *Observables* pull model of processing. **Hot *Observable* emits items at its own pace, and it is up to its observers to keep up.**

When the *Observer* is not able to consume items as quickly as they are produced by an *Observable*they need to be buffered or handled in some other way, as they will fill up the memory, finally causing *OutOfMemoryException*.

```
Observable<Integer> observable=
PublishSubject.create(new
ObservableOnSubscribe<Integer>() {
    @Override
    public void
subscribe(ObservableEmitter<Integer>
emitter) throws Exception {

        for(int i = 1 ; i <= 10000;i++)
        {
            emitter.onNext(i);
        }


    }
});

//observable.publish();

observable.subscribe(new
Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Integer integer) {
        Log.d("integer 1 ",""+integer);
```

```
    }

    @Override
    public void onError(Throwable e) {
        Log.d("Error:",""+e.getMessage());
    }

    @Override
    public void onComplete() {

    }
});
```

we need backuppressuring , which in simple words is just a way to handle the items that can’t be processed

Buffer

periodically gather items emitted by an Observable into bundles and emit these bundles rather than emitting the items one at a time

The Buffer operator transforms an Observable that emits items into an Observable that emits buffered collections of those items. There are a number of variants in the various language-specific implementations of Buffer that differ in how they choose which items go in which buffers.

//default which basically buffers upto 128 items in the queue.

Since the default queue size on Android is only 16 items (versus 128 on the JVM), this can become a major issue.

onBackpressureBuffer

maintains a buffer of all emissions from the source Observable and emits them to downstream Subscribers according to the requests they generate

onBackpressureDrop

drops emissions from the source Observable unless there is a pending request from a downstream Subscriber, in which case it will emit enough items to fulfill the request

observable.toFlowable(BackpressureStrategy.LATEST)

BackpressureStrategy.*MISSING*

BackpressureStrategy.*ERROR*

BackpressureStrategy.***DROP***
BackpressureStrategy.***LATEST***
BackpressureStrategy.***BUFFER***

BUFFER
Buffers all onNext values until the downstream consumes it.

DROP
Drops the most recent onNext value if the downstream can't keep up.

ERROR
Signals a MissingBackpressureException in case the downstream can't keep up.

LATEST
Keeps only the latest onNext value, overwriting any previous value if the downstream can't keep up.

MISSING
OnNext events are written without any buffering or dropping.

4 types of actions that can be executed when the buffer fills up:

ON_OVERFLOW_ERROR – this is the default behaviour signalling a BufferOverflowException when the buffer is full

ON_OVERFLOW_DEFAULT – currently it is the same as ON_OVERFLOW_ERROR

ON_OVERFLOW_DROP_LATEST – if an overflow would happen, the current value will be simply ignored and only the old values will be delivered once the downstream Observer requests

ON_OVERFLOW_DROP_OLDEST – drops the oldest element in the buffer and adds the current value to it

Example of flowable like

locationUpdates() , onImageCapture(),

……..

Completable :

Emits no item means it done the job .

Completable class represents a deferred computation without any value but only indication for completion or exception.

Completable behaves similarly to Observable except that it can only emit either a completion or error signal

The Completable class implements the CompletableSource base interface and the default consumer type it interacts with is the CompletableObserver via the subscribe(CompletableObserver) method. The Completable operates with the following sequential protocol:

onSubscribe (onError | onComplete)?

Either completes with an success or with an error event. It never emits items. The reactive version of a Runnable.

```
public interface CompletableSubscriber {

    void onCompleted();

    void onError(Throwable e);

    void onSubscribe(Subscription d);
}
```

//———- RETROFIT CALL — ASYNC TASK — //

AsyncTask enables proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask is designed to be a helper class around Thread and Handler and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.)

AsyncTask enables proper and easy use of the UI thread. This class allows you to perform background operations and publish results on the UI thread without having to manipulate threads and/or handlers.

AsyncTask is designed to be a helper class around Thread and Handler and does not constitute a generic threading framework. AsyncTasks should ideally be used for short operations (a few seconds at the most.) If you need to keep threads running for long periods of time, it is highly recommended you use the various APIs provided by the java.util.concurrent package such as Executor, ThreadPoolExecutor and FutureTask.

An asynchronous task is defined by a computation that runs on a background thread and whose result is published on the UI thread. An asynchronous task is defined by 3 generic types, called Params, Progress and Result, and 4 steps, called onPreExecute, doInBackground, onProgressUpdate and onPostExecute.

Usage

AsyncTask must be subclassed to be used. The subclass will override at least one method (doInBackground(Params...)), and most often will override a second one (onPostExecute(Result).)

Here is an example of subclassing:

Params, the type of the parameters sent to the task upon execution.

Progress, the type of the progress units published during the background computation.

Result, the type of the result of the background computation.

Not all types are always used by an asynchronous task. To mark a type as unused, simply use the type Void:

```
private class MyTask extends AsyncTask<Void, Void, Void> { ... }
```

The 4 steps

When an asynchronous task is executed, the task goes through 4 steps:

onPreExecute(), invoked on the UI thread before the task is executed. This step is normally used to setup the task, for instance by showing a progress bar in the user interface.

doInBackground(Params...), invoked on the background thread immediately after onPreExecute() finishes executing. This step is used to perform background computation that can take a long time. The parameters of the asynchronous task are passed to this step. The result of the computation must be returned by this step and will be passed back to the last step. This step can also use publishProgress(Progress...) to publish one or more units of progress. These values are published on the UI thread, in the onProgressUpdate(Progress...) step.

onProgressUpdate(Progress...), invoked on the UI thread after a call to publishProgress(Progress...). The timing of the execution is undefined. This method is used to display any form of progress in the user interface while the background computation is still executing. For instance, it can be used to animate a progress bar or show logs in a text field.

onPostExecute(Result), invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Drawbacks :

Memory leaks

Because an AsyncTask has methods that run on the worker thread (doInBackground()) as well as methods that run on the UI (e.g. onPostExecute()), it has took keep a reference to it's Activity as long as it's running. But if the Activity has already been destroyed, it will still keep this reference in memory. This is completely useless because the task has been cancelled anyway.

Losing your results

Another problem is that we lose our results of the AsyncTask if our Activity has been recreated. For example when an orientation change occurs. The Activity will be destroyed and recreated, but our AsyncTask will now have an invalid reference to its Activity, so onPostExecute() will have no effect. There is a solution for this. You can hold onto a reference to AsyncTask that lasts between configuration changes (for example using a global holder in the Application object).

Operators

Where we use the main thread or where we work with background process it depends on schedulers

what is async or sync behaviour or what is subscribeOn what is observeOn also Schedulers and lot of other things

but first basic thing we discuss first difference between Async and Sync tasks and how android support it ..

When you execute something synchronously, we have to wait for it to finish before moving on to another task. When we execute something asynchronously, we can move on to another task before it finishes.

This sequential behaviour can be termed as Synchronous …

The execution which happens independently, without interrupting the Normal flow of execution is called Asynchronous call.

Android Behaviour : Main UI ::::Everything that We run on main/UI thread runs in synchronous manner. is it ?

If We start doing long running task on main thread (like really long calculation and load lot of data- more than 5 seconds), then till that task is done, we can't do anything in that app. Our long running task is blocking the UI thread.some times it create this will create a Application Not Responding (ANR dialog).

so its good we shift the long running task somewhere in new thread from Main UI Thread and once it done we show that there …so we worked with Async task as We have done in just previous uploaded video ….

so that was old approach we discussing the new component Rxjava or Reactive Programming …

Reactive programming is actually easier and immensely powerful if you know how to make proper use of it. If it seems difficult to you, it simply means that we have to work on right concepts …

many authors have putted the Rxjava views or useful area …

Reactive programming is useful in scenarios such as the following:

- **Processing user events such as mouse movement and clicks, keyboard typing, GPS signals changing over time as users move with their device, device gyro- scope signals, touch events, and so on.**

- **Responding to and processing any and all latency-bound IO events from disk or network, given that IO is inherently asynchronous (a request is made, time passes, a response might or might not be received, which then triggers further work).**

- **Handling events or data pushed at an application by a producer it cannot control (system events from a server, the aforementioned user events, signals from hard- ware, events triggered by the analog world from sensors, and so on).**

Today Netflix,Twitter is good example of Rxjava .. will discuss some other session

BUT

First important point is when we work with Rxjava it means it Async behaviour of Observable …so need to understand

Whenever we are making a network call to retrieve some data from some remote server. The network call is not synchronous.

May be it will take some time, from a some milliseconds to a few seconds, depending on whatever be quality of network , server load, etc. to deliver the results back to you. many advantages using Rxjava

avoid memory leaks , achieve multiple web service calls and many others..will discuss multiple calls and how make a group or join in next video …

Today we have to discuss

the difference between subscribeOn and observeOn and Schedulers type …

so as we discussed that Observable emits data and we subscribeOn and observeOn and observer receive it ..

SubscribeOn specify the Scheduler on which an Observable will operate. ObserveOn specify the Scheduler on which an observer will observe this Observable.

subscribeOn, we specify the background thread where the data flow will be generated and processed.(depends on scheduler)

observeOn is supposed to redirect the chain of operations to another thread.

So basically SubscribeOn is mostly subscribed (executed) on a background thread (you do not want to block the UI thread while waiting for the observable) and also in ObserveOn you want to observe the result on a main thread…

so in observeOn main thread we use as we are working AndroidSchedulers.mainthread ….

SubscribeOn is similar to doInBackground method and ObserveOn to onPostExecute..

An asynchronous Observable emits on a thread that is separate from the Observer's thread while synchronous Observable emits on the same thread to that of the Observer will discuss later ..

so we can say the Observable to send notifications to Observers on a specified Scheduler.

ObserveOn affects the thread that the Observable will use below where that operator appears.

SubscribeOn operator designates which thread the Observable will begin operating on, no matter at what point in the chain of operators that operator is called.

observeOn works only downstream.

subscribeOn works downstream and upstream.

By default, Rx is single-threaded which implies that an Observable and the chain of operators that we can apply to it will notify its observers on the same thread on which its subscribe() method is called.

*****

The SubscribeOn operator changes this behaviour by specifying a different Scheduler on which the Observable

should operate. The ObserveOn operator specifies a different Scheduler that the Observable will use to send notifications to its observers. *

The observeOn and subscribeOn methods take as an argument a Scheduler, that, as the name suggests, is a tool that we can use for scheduling individual actions.

Schedulers are one of the main components in RxJava. They are responsible for performing operations of Observable on different threads. They help to offload the time-consuming onto different threads.

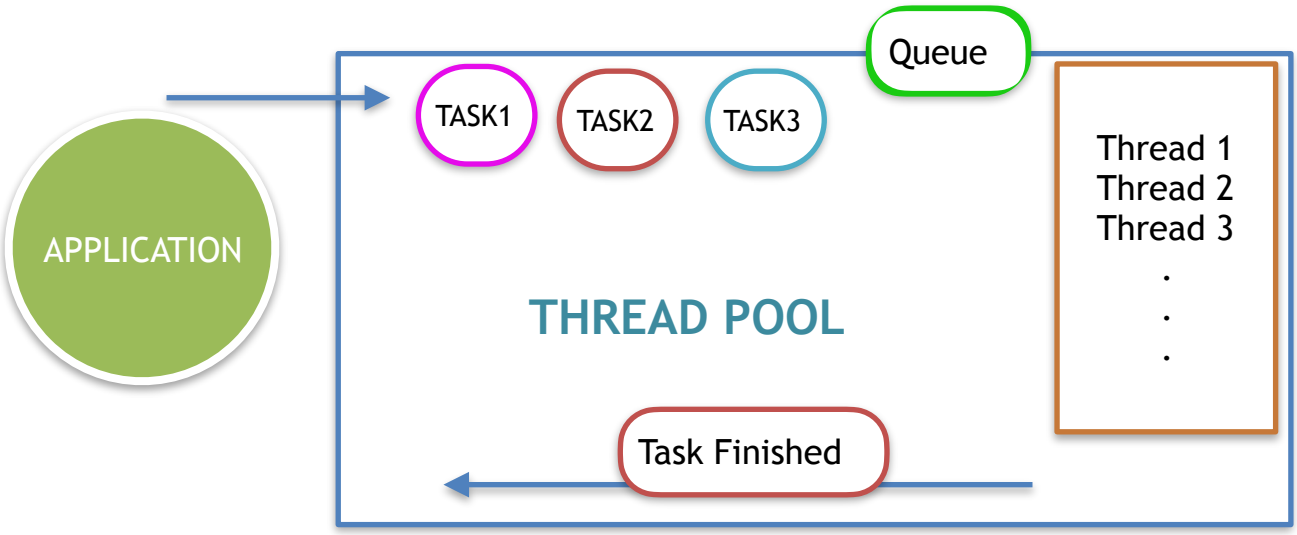
ObserveOn, on the other hand, affects the thread that the Observable will use below where that operator appears. For this reason, you may call ObserveOn multiple times at various points during the chain of Observable operators in order to change on which threads certain of those operators operate.

Thread Pool :

A thread pool is the collection of pre-instantiated standby threads which are ready to execute application level tasks.

Thread pools improve performance by running multiple tasks simultaneously, and at the same time they prevent the time and memory overhead incur during thread creation.

This is very simple and everyone know how Thread pool works but here important thing that we need to know



bounded and unbounded Queues .. that is working in Rxjava schedulers ..

Queues can be bounded (having a predefined capacity) or unbounded (without a predefined capacity).

if we go with the java thread pooling and if number of queued tasks exceeds beyond the capacity of a bounded queue (means size)

There might be two logical approaches there: either extra tasks should be rejected at this point or new threads should be allowed to be created again. Now if new threads are allowed again then unlimited task submission may again bring about the resource crisis and eventually application might fail. so here it is important which scheduler be used so doing perfect response with our app…

so need to understand the upstream and downstream ….

Observable

So from top to the operator, we call it upstream.

From operator to the bottom, we call it downstream.

subscribeOn and observeOn are only operators.

subscribeOn is affecting its upstream and downstream.

observeOn only affecting downstream.

An operator is a function that takes and alters the behavior of an upstream Observable and returns a downstream Observable or Subscriber,

The ObserveOn operator specifies a different Scheduler that the Observable will use for sending notifications to Observers:We see that elements were produced in the main thread and were pushed all the way to the first map call.

One problem that may arise with observeOn is the bottom stream can produce emissions faster than the top stream can process them. This can cause issues with backpressure that we may have to consider.

To specify on which Scheduler the Observable should operate, we can use the subscribeOn operator:

SCHEDULERS

First Lets discuss about Thread behind the scene

The UIThread or Main Thread execute our application. means where the most of our application code is run. and All of our application components (Activities, Services, ContentProviders, BroadcastReceivers) are created in this thread, and any system calls to those components are performed in this thread.

Every Android app has a main thread which is in charge of handling UI (including measuring and drawing views), coordinating user interactions, and receiving lifecycle events. If there is too much work happening on this thread, the app appears to hang or slow down, leading to an undesirable user experience. Any long-running computations and operations such as decoding a bitmap, accessing the disk, or performing network requests should be done on a separate background thread.

ThreadPools provide a group of background threads that accept and enqueue submitted work. If you need to monitor system triggers during this time.

A thread pool can run multiple parallel instances of a task, so you should ensure that your code is thread-safe. Enclose variables that can be accessed by more than one thread in a synchronized block. This approach will prevent one thread from reading the variable while another is writing to it.

so what was our old approach of doing background process …Asynctask

If we want to show something in the UI Main while the AsyncTask is computing the background task we should implement the onProgressUpdate(Progress... values) method which is called from publishProgress(Progress... values) and implements the logic under the background task progress. we just show the final result in the onPostExecute()„„

so that we do with Rxjava and as we discussed we have operators observeOn and subscribeOn that do work for observers

Observable and the chain of operators that you apply to it will do its work, and will notify its observers, on the same thread on which its Subscribe method is called. The SubscribeOn operator changes this behavior by specifying a different Scheduler on which the Observable should operate. The ObserveOn operator specifies a different Scheduler that the Observable will use to send notifications to its observers.

and in most cases observeOn operator work with AndroidScheduler.mainThread() … once process complete then show in main thread observers

Java Thread pool represents a group of worker threads that are waiting for the job and reuse many times.

Scheduler IO - This is one of the most common types of Schedulers that are used. They are generally used for IO related stuff. Such as network requests, file system operations. IO scheduler is backed by thread-pool.

observable.subscribeOn(Schedulers.io())

```
Observable.just("one","two")
  .subscribeOn(Schedulers.io())
  .subscribe(i -> result +=
Thread.currentThread().getName());
```

***** we can say this …**

io() is backed by an unbounded thread-pool and is the sort of thing you'd use for non-computationally intensive tasks, that is stuff that doesn't put much load on the CPU.*

****

Schedulers.io( ) by default is a CachedThreadScheduler, which is something like a new thread scheduler with thread caching…

I have made a retrofit call on my last video of rxjava using scheduler.io()

Creates and returns a Scheduler intended for IO-bound work.

The implementation is backed by an Executor thread-pool that will grow as needed.

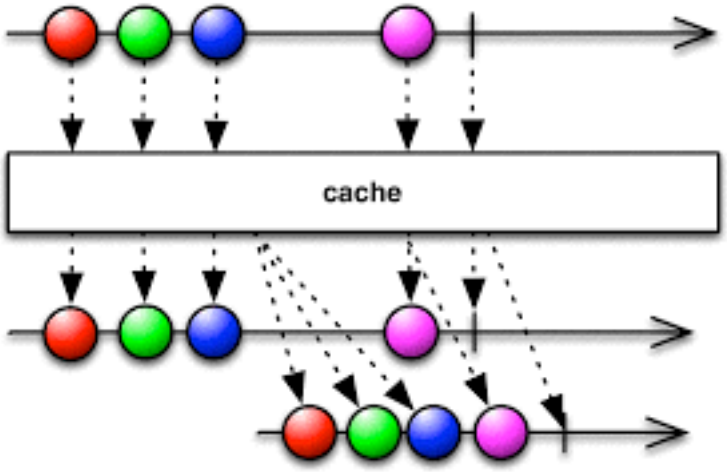
This can be used for asynchronously performing blocking IO.

Do not perform computational work on this scheduler. Use computation() instead.

Un handled errors will be delivered to the scheduler Thread's Thread.UncaughtExceptionHandler.

we shall discuss smoother time why it is cache scheduler

Scheduler io cache



Computation

A computational task is defined as any type of calculation that includes both arithmetical and non-arithmetical steps and follows a well-defined model e.g. an algorithm." So basically when it comes to computation, think about algorithms and math. We all know CPUs are brilliant at solving mathematical/structured problems that involve complex logic and multiple tasks …

Computation - This scheduler is quite similar to IO Schedulers as this is backed by thread pool too. However, the number of threads that can be used is fixed to the number of cores present in the system. it is good for performing small calculations and are generally quick to perform operation. It is available as:observable.subscribeOn(Schedulers.computation())

Computation Scheduler by default limits the number of threads running in parallel to the value of availableProcessors(), as found in the Runtime.getRuntime() utility class.

The Computation scheduler uses a limited number of threads. This limit is tightly associated with the number of CPU cores you have.

CPU bound work is any type of work whose efficient completion is bound by the hardware characteristics of your CPU.

where we have to work with delays , intervals like that ..

RxJava Operators that default to the computation scheduler

Static: interval(), timer(), timeout()

Instance: buffer(), delay(), take(), skip(), takeWhile(), skipWhile(), timeout() and window().

So we should use a computation scheduler when tasks are entirely CPU-bound; that is, they require computational power and have no blocking code.

```
Observable.just(“one”,”two")
  .subscribeOn(Schedulers.computation())
  .subscribe(i -> result +=
Thread.currentThread().getName());
```

public static Scheduler computation()

Creates and returns a Scheduler intended for computational work.

This can be used for event-loops, processing callbacks and other computational work.

Do not perform IO-bound work on this scheduler. Use io() instead.

Unhandled errors will be delivered to the scheduler Thread's Thread.UncaughtExceptionHandler.

Returns:

a Scheduler meant for computation-bound work

NewThread - As the name suggests, it spawns a new thread for each active observable. This can be used to offload time consuming operation from main thread onto other thread.

need to take care of this because thread creation is a costly operation and can have a drastic effect in mobile environment if the number of observables are high enough.

observable.subscribeOn(Schedulers.newThread())

This scheduler simply starts a new thread every time it is requested via subscribeOn() or observeOn().

It's hardly ever a good choice, not only because of the latency involved when starting a thread but also because this thread is not reused …

so its important to be note that we have to use very rare singleThread for this we have to go in little depth

newThread() creates a new thread for each unit of work. io() will use a thread pool

A thread pool is a group of pre-instantiated, idle threads which stand ready to be given work. These are preferred over instantiating new threads for each task when there is a large number of short tasks to be done rather than a small number of long ones. This prevents having to incur the overhead of creating a thread a large number of times.

generally we need to call in our Retrofit calling schedulers.io()

Creating a new thread object every time you need something to be executed asynchronously is expensive. In a thread pool you would just add the tasks you wish to be executed asynchronously to the task queue and the thread pool takes care of assigning an available thread, if any, for the corresponding task.

means use in very rare case for async call schedulers.newThread() and dispose it ….

Single - This scheduler is quite simple as it is backed just by one single thread. So no matter how many observables are there, it will run only on that one thread. It can be thought as a replacement to your main thread. It is available as:

observable.subscribeOn(Schedulers.single())

Trampoline - This scheduler runs the code on current thread. So if you have a code running on the main thread, this scheduler will add the code block on the queue of main thread. It is quite similar to Immediate Scheduler as it also blocks the thread, however, it waits for the current task to execute completely(while Immediate Scheduler invokes the task right away). Trampoline schedulers come in handy when we have more than one observable and we want them to execute in order.

```
observable.subscribeOn(Schedulers.trampoline())
Observable.just(1,2,3,4)
    .subscribeOn(Schedulers.trampoline())
    .subscribe(onNext);
 Observable.just( 5,6, 7,8, 9)
    .subscribeOn(Schedulers.trampoline())
    .subscribe(onNext);
```

The trampoline Scheduler is very similar to immediate because it also schedules tasks in the same thread, effectively blocking.

However, the upcoming task is executed when all previously scheduled tasks complete:

1

2

3

4

5

6

7

8

```
Observable.just(2, 4, 6, 8)
  .subscribeOn(Schedulers.trampoline())
  .subscribe(i -> result += "" + i);
Observable.just(1, 3, 5, 7, 9)
  .subscribeOn(Schedulers.trampoline())
  .subscribe(i -> result += "" + i);
Thread.sleep(500);
```

Android Scheduler - This Scheduler is provided by rxAndroid library. This is used to bring back the execution to the main thread so that UI modification can be made. This is usually used in observeOn method. It can be used by:

```
AndroidSchedulers.mainThread()
```

More Schedulers

Schedulers.test

This Scheduler is used only for testing purposes, and we'll never see it in production code. Its main advantage is the ability to advance the clock

Default Schedulers

Some Observable operators in RxJava have alternate forms that allow us to set which Scheduler the operator will use for its operation. Others don't operate on any particular Scheduler or operate on a particular default Scheduler.

For example, the delay operator takes upstream events and pushes them downstream after a given time. Obviously, it cannot hold the original thread during that period, so it must use a different Scheduler:

Non-Blocking – asynchronous execution is supported and is allowed to unsubscribe at any point in the event stream. On this article, we'll focus mostly on this kind of type

Blocking – all onNext observer calls will be synchronous, and it is not possible to unsubscribe in the middle of an event

stream. We can always convert an Observable into a Blocking Observable, using the method toBlocking:

background process depends on thread so can use it there new Thread .

```
@GET("/user/{id}/photo")
Observable<Photo> getUserPhoto(@Path("id") int id);
@GET("/photo/{id}/metadata")
Observable<Metadata> getPhotoMetadata(@Path("id") int id);
Observable.zip(
service.getUserPhoto(id),
service.getPhotoMetadata(id),
(photo, metadata) -> createPhotoWithData(photo, metadata))
.subscribeOn(Schedulers.io()) // executed in the background thread
.observeOn(AndroidSchedulers.mainThread()) // shows the result in the UI thread
.subscribe(photoWithData -> showPhoto(photoWithData));
```

subscribeOn works downstream and upstream. All the tasks above and below it would use the same thread.

observeOn works downstream only.

consecutive subscribeOn methods won’t change the thread. Only the first subscribeOn thread would be used.

consecutive observeOn methods will change the thread.

Combining Two Observables

sometimes we need to consume data from more than just a single source. Some examples We have multiple inputs include:

multi touch surfaces

news feeds

price feeds

social media aggregators

multiple file watchers and many more

as we can say stimulation

We want to consume it and integrated data, or one sequence at a time as sequential data. Sometimes we need to get it in an orderly fashion, pairing data values from two sources to be processed together, or perhaps just consume the data from the first source that responds to the request.

so in language of Rxjava if we emit two or more Observable values then need to give single sequence via observer ..

so can say that…

Instead of waiting for the previous stream to complete before requesting the next stream, we can call both at the same time and subscribe to the combined streams.

There are two kind of combination

Sequential Combination

Concurrent Combination

Sequential Combination :

Working with sequence like as we did in java concat same here it works

```
Observable first= Observable.range(1,10);
Observable second= Observable.range(2,15);

first.concatWith(second).subscribeOn(Schedu
lers.computation()).observeOn(AndroidSchedu
lers.mainThread()).subscribe(new Observer()
{
    @Override
```

```
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Object o) {

        Log.d("Value",""+o.toString());

    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});



\
```

```
using model …
Observable.concat(getProgrammingBooks(),get
Autobiography()).observeOn(AndroidScheduler
s.mainThread()).subscribe(new
Observer<Book>() {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Book book) {



Log.d("bookname",""+book.getBookname());

Log.d("booktype",""+book.getBooktype());
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {
```

```
    }
});
private Observable<Book> getAutobiography()
{
    String [] booknames= new String[]
{"book1","book2"};
    final ArrayList<Book> arrayList = new
ArrayList<>();
    for(String name:booknames){
        Book book = new Book();
        book.setBooktype("Autobiograph");
        book.setBookname(name);
        arrayList.add(book);
    }

    return Observable.create(new
ObservableOnSubscribe<Book>() {
        @Override
        public void
subscribe(ObservableEmitter<Book> emitter)
throws Exception {

            for(Book book:arrayList)
            {
                emitter.onNext(book);

            }
            if (!emitter.isDisposed()) {
                emitter.onComplete();
```

```
            }
        }
    }).subscribeOn(Schedulers.io());
}

private Observable<Book>
getProgrammingBooks()
{
    String [] booknames= new String[]
{"book3","book4"};
    final ArrayList<Book> arrayList = new
ArrayList<>();
    for(String name:booknames){
        Book book = new Book();
        book.setBooktype("Programming");
        book.setBookname(name);
        arrayList.add(book);
    }

    return Observable.create(new
ObservableOnSubscribe<Book>() {
        @Override
        public void
subscribe(ObservableEmitter<Book> emitter)
throws Exception {

            for(Book book:arrayList)
            {
                emitter.onNext(book);
```

```
            }
            if (!emitter.isDisposed()) {
                emitter.onComplete();
            }
        }
    }).subscribeOn(Schedulers.io());
}
```

we may need to get two sets of asynchronous data streams that are independent of each other.

Instead of waiting for the previous stream to complete before requesting the next stream, we can call both at the same time and subscribe to the combined streams.

Amb , Merge , zip , join ,combine latest

MERGE

This operator combines multiple Observables into one by merging their emissions i.e. merges multiple Observables into a single Observable but it not maintain the sequential execution

it emit data from both the observable simultaneously as soon as the data become available to emit .

Merge also merges multiple Observables into a single Observable but it won't maintain the sequential execution.

Observable.merge(observer1,observer2)..

ZIP

This operator combines the emissions of multiple Observables together via a specified function and emit single items for each combination based on the results of this function.

Zip operator strictly pairs emitted items from observables. It waits for both (or more) items to arrive then merges them.

Observable.zip(observer1,observer2, function …..

```
 Observable.zip(observable1, observable2, new
BiFunction<String, String, Object>() {
    @Override
    public Object apply(String s, String s2) throws Exception {

        String svalue = s + "---" +s2;
        return svalue;
    }
 }).observeOn(AndroidSchedulers.mainThread()).subscribeOn(
Schedulers.io()).subscribe(new Observer<Object>() {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Object o) {

        Log.d("under next",""+o.toString());

    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {
```

```
    }
});

/*
Observable.merge(getAutobiographyBooks(),getProgrammingB
ooks()).observeOn(AndroidSchedulers.mainThread()).subscribe
On(Schedulers.io()).subscribe(new Observer<Book>() {

@Override
```

Join

The Join operator combines the items emitted by two Observables, and selects which items to combine based on duration-windows that you define on a per-item basis. You implement these windows as Observables whose lifespans begin with each item emitted by either Observable. When such a window-defining Observable either emits an item or

completes, the window for the item it is associated with closes.

Combine items emitted by two Observables whenever an item from one Observable is emitted during a time window defined according to an item emitted by the other Observable

The Join operator combines the items emitted by two Observables, and selects which items to combine based on duration-windows that you define on a per-item basis..

Whenever two items (each one for one source) are overlapped, they will be paired and sent to the resultSelector which computes and returns them. The join() operator takes the following items:

right—the second Observable to join items from.

leftDurationSelector—a function to select a duration for each item emitted by the source Observable, used to determine overlap.

rightDurationSelector—a function to select a duration for each item emitted by the right Observable, used to determine overlap.

resultSelector—a function that computes an item to be emitted by the resulting Observable for any two overlapping items emitted by the two Observables.