1. What is Reactive Programming

Reactive Programming is basically **event:based asynchronous programming**.
Everything you see is an **asynchronous data stream**, which can be observed and an action will be taken place when it emits values.
You can create **data stream** out of anything**; variable changes, click events, http calls, data storage, errors and what not**.
When it says asynchronous, that means every code module runs on its own thread thus **executing multiple code blocks simultaneously**.

2. Reactive Extensions

Reactive Extensions (ReactiveX or RX) is a library that follows Reactive Programming principles i.e compose asynchronous and event based programs by using observable sequence.
These libraries provides set of interfaces and methods which helps developers write clean and simpler code.
Reactive Extensions are available in multiple languages C++ (RxCpp), C# (Rx.NET), Java (RxJava), Kotlin (RxKotlin), Swift (RxSwift) and lot more. We specifically interested in RxJava and RxAndroid as android is our focused area.

3. What is RxJava

RxJava is Java implementation of **Reactive Extension** (from Netflix).
Basically it's a library that composes **asynchronous events** by following **Observer Pattern**.
You can create **asynchronous data stream on any thread, transform the data and consumed it by an Observer on any thread**.
The library offers wide range of amazing operators like **map, combine, merge, filter** and lot more that can be applied onto data stream.

4. What is RxAndroid

RxAndroid is specific to Android Platform with few added classes **on top of RxJava**.
More specifically, **Schedulers** are introduced in RxAndroid (**AndroidSchedulers.mainThread()**) which plays major role in supporting multithreading concept in android applications.
Schedulers basically decides the thread on which a particular code runs whether on **background thread** or **main thread**.
Apart from it **everything** we use is from RxJava library **only.**

5. Schedulers

- **Schedulers.io()**: This is used to perform non CPU: intensive operations like making network calls, reading disc / files, database operations etc., This maintains pool of threads.

- **AndroidSchedulers.mainThread()**: This provides access to android Main Thread / UI Thread. Usually operations like updating UI, user interactions happen on this thread. We shouldn't perform any intensive operations on this thread as it makes the app glitchy or ANR dialog can be thrown.
- **Schedulers.newThread()**: Using this, a new thread will be created each time a task is scheduled. It's usually suggested not to use scheduler unless there is a very long running operation. The threads created via newThread() won't be reused.
- **Schedulers.computation()**: This scheduler can be used to perform CPU:intensive operations like processing huge data, bitmap processing etc., The number of threads created using this scheduler completely depends on number CPU cores available.
- **Schedulers.single()**: This scheduler will execute all the tasks in sequential order they are added. This can be used when there is necessity of sequential execution is required.
- **Schedulers.immediate()**: This scheduler executes the task immediately in synchronous way by blocking the main thread.
- **Schedulers.trampoline()**: It executes the tasks in First In – First Out manner. All the scheduled tasks will be executed one by one by limiting the number of background threads to one.
- **Schedulers.from()**: This allows us to create a scheduler from an executor by limiting number of threads to be created. When thread pool is occupied, tasks will be queued.

6. RxJava Basics: Observable, Observer

RxJava is all about two key components: **Observable** and **Observer**. In addition to these, there are other things like **Schedulers, Operators and Subscription**.

- **Observable**: Observable is a **data stream that do some work** and **emits data**.
- **Observer**: Observer is the **counter part** of Observable. It **receives the data** emitted by Observable.
- **Subscription**: The **bonding between Observable and Observer** is called as Subscription.
  There can be **multiple Observers subscribed to a single Observable**.
- **Operator** / **Transformation**: Operators **modifies the data emitted** by Observable before an observer receives them.
- **Schedulers**: Schedulers **decides the thread** on which Observable should emit the data and on **which Observer should receive the data** i.e. background thread, main thread etc.

7. 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**.

Let's say you are making a long running network call and updating the UI.
By the time network call completes its work, if the activity / fragment is already destroyed,
as the Observer subscription is still alive, it tries to update already destroyed activity.
In this case it can throw a memory leak. So using the Disposables, the un-subscription can be when the activity is destroyed.

8. Multiple Observers and CompositeDisposable

Consider a case where you have **multiple Observables and Observers**. Disposing them in Destroy **one by one** is a tedious task and it can be **error prone** as you might forget to dispose. In this case we can use **CompositeDisposable**.

CompositeDisposable: Can maintain **list of subscriptions in a pool and can dispose them all at once**.

Usually we call **compositeDisposable.clear() in onDestroy()** method, but you can **call anywhere in the code**.

Example 1: Basic Observable, Observer

Source code:
https://github.com/MohamedAssemAli/Rx:Java:Recap/blob/master/app/src/main/java/com/orchtech/assem/rxrecap/BasicExamplesActivity.java

9. Operators

RxJava Operators allows you **manipulate the data emitted by Observables**.
Basically, operators tell Observable, **how to modify the data** and **when to emit the data**.
Using the operators, you can **modify, merge, filter or group the data streams**.

Operators by category:

Creating Observables

- **Operators that originate new Observables.**
    - **Create:** create an Observable from scratch by calling observer methods programmatically
    - **Just:** convert an object or a set of objects into an Observable that emits that or those objects
    - **Range:** create an Observable that emits a range of sequential integers

Transforming Observables

- Operators that transform items that are emitted by an Observable.
    - **Buffer:** periodically gather items from an Observable into bundles and emit these bundles rather than emitting the items one at a time
    - **FlatMap:** transform the items emitted by an Observable into Observables, then flatten the emissions from those into a single Observable
    - **GroupBy:** divide an Observable into a set of Observables that each emit a different group of items from the original Observable, organized by key
    - **Map:** transform the items emitted by an Observable by applying a function to each item

Filtering Observables

- Operators that selectively emit items from a source Observable.
  - **Debounce:** only emit an item from an Observable if a particular timespan has passed without it emitting another item
  - **Distinct:** suppress duplicate items emitted by an Observable
  - **ElementAt:** emit only item n emitted by an Observable
  - **Filter:** emit only those items from an Observable that pass a predicate test
  - **First:** emit only the first item, or the first item that meets a condition, from an Observable
  - **Last:** emit only the last item emitted by an Observable
  - **Skip:** suppress the first n items emitted by an Observable
  - **SkipLast:** suppress the last n items emitted by an Observable
  - **Take:** emit only the first n items emitted by an Observable
  - **TakeLast:** emit only the last n items emitted by an Observable

Choosing between Map operators:

Consider using Map operator where there is an offline operation needs to be done on emitted data.
As explained in the article, we got something from server but that doesn't fulfil our requirement.
In that case, Map can be used to alter the emitted data.

Choose FlatMap when the order is not important.
Let's say you are building an Airline Ticket Fair app that fetches the prices of each airline separately and display on the screen.
For this both **FlatMap** and **ConcatMap** can be used.

But **if the order is not important** and want to send all the network calls simultaneously, I would consider **FlatMap over ConcatMap**.
If you **consider ConcatMap** in this scenario, the time takes to fetch the prices takes very longer time as the **ConcatMap won't make simultaneous calls** in order to **maintain item order**.

SwitchMap is best suited when you want to **discard the response and consider the latest one**.
Let's say you are writing an **Instant Search app** which s**ends search query to server each time user types** something.
In this case **multiple requests** will be sent to server with multiple queries, but we want to show the **result of latest typed query only**. For this case, **SwitchMap** is best operator to use.

Another use case of **SwitchMap** is, you **have a feed screen** in which feed is refreshed each time user perform **pulldown to refresh**. In this scenario, SwitchMap is best suited as it can **ignore the older feed response** and consider only the latest request.