* **RxAndroid-Pending**
* **Flow**
* **RxJava**
* **RxKotlin**

**AndroidX**

The androidx namespace comprises the [Android Jetpack](https://developer.android.com/jetpack) libraries. Like the Support Library, libraries in the androidx namespace ship separately from the Android platform and provide backward compatibility across Android releases.

AndroidX is a major improvement to the original Android [Support Library](https://developer.android.com/topic/libraries/support-library), which is no longer maintained. androidx packages fully replace the Support Library by providing feature parity and new libraries.

**Note**: AndroidX is collection of library(LiveData, ViewModel, Room, Navigation) with backward support with replacement of support library.It is the fully replacement of support library

**AndroidX** is the open-source project that the Android team uses to develop, test, package, version and release libraries within [Jetpack](https://developer.android.com/jetpack).

**AndroidX** is a major improvement to the original [Android Support Library](https://developer.android.com/topic/libraries/support-library/index). Like the Support Library, AndroidX ships separately from the Android OS and provides backward-compatibility across Android releases. AndroidX fully replaces the Support Library by providing feature parity and new libraries.

AndroidX includes the following features:

* All packages in AndroidX live in a consistent namespace starting with the string androidx. The Support Library packages have been mapped into the corresponding androidx.\* packages. For a full mapping of all the old classes and build artifacts to the new ones, see the Package Refactoring page.
* Unlike the Support Library, AndroidX packages are separately maintained and updated. The androidx packages use strict Semantic Versioning starting with version 1.0.0. You can update AndroidX libraries in your project independently.
* All new Support Library development will occur in the AndroidX library. This includes maintenance of the original Support Library artifacts and introduction of new Jetpack components

com.android.support.\*\* : **androidx.**

com.android.support:appcompat-v7 : **androidx.appcompat:appcompat**

com.android.support:recyclerviewv7 : **androidx.recyclerview:recyclerview**

com.android.support:design : **com.google.android.material:material**

**Flow**

**Flow:** Kotlin Flow is one of the latest addition to the Kotlin Coroutines. With Kotlin Flow we can handle streams of data asynchronously which is being executed sequentially.

Flow is the Kotlin type that can be used to model streams of data. Just like LiveData and RxJava streams, Flow lets you implement the observer pattern: a software design pattern that consists of an object (a.k.a. observable, producer, source, emitter) that maintains a list of its dependents, called observers (subscribers, collectors, receivers, etc.), and notifies them automatically of any state changes. Observables can be either hot or cold. Hot observables send or emit data even if no one is watching them and cold observables emit data only if they have an active observer. By default, Kotlin flows are cold.

In coroutines, a *flow* is a type that can emit multiple values sequentially, as opposed to *suspend functions* that return only a single value. For example, you can use a flow to receive live updates from a database.

Flows are built on top of coroutines and can provide multiple values. A flow is conceptually a *stream of data* that can be computed asynchronously.

**RxJava**

**RxJava** is a Java library that enables Functional Reactive Programming in Android development. It is a library that’s used for writing asynchronous event-based reactive code /program by using observables pattern.

**Rx(**Reactive Extensions**)**: Reactive Programming refers to the scenario where program reacts as and when data appears. It is a event based programming concept and events can propagate to registers observers. As per the **Reactive**, they have combined the best of Observer pattern, Iterator pattern and functional pattern.

**Reactive programming** is a general programming term that is focused on reacting to changes, such as data values or events. A callback is an approach to reactive programming done imperatively.

It’s one of the most discussed libraries for enabling Reactive Programming in Android development.

For **example**, if you have one source of data (producer) and one target for data (consumer); then after connecting the consumer to subscriber – reactive programming framework is responsible for pushing the data, generated by the producer, to the consumer. Please note that an observable can have any number of subscribers.

**RxJava**

* **Rx** stands for Reactive Extensions.
* RxJava is a JVM implementation of **Reactive Extensions**.
* Reactive Extension is a library that’s used for writing asynchronous event-based reactive code by using observables. We’ll see what are observables shortly.
* RxJava is useful and very powerful in the sense that it takes care of [multithreading](https://www.journaldev.com/1079/multithreading-in-java) very well.
* If you’re a Java developer, you’ll be well aware of the fact that multithreading can get tricky. RxJava takes care of multi-threading by doing complex thread operations, keeping everything synchronized and returning the relevant things to the main thread.
* In Android, the main thread is the UI thread. RxJava handles multithreading with a level of abstraction. We need to write less code and the underlying methods do the rest for us.

**The basic building blocks of RxJava are:**

* **Observables(Publisher)**: That emits data streams. It can asynchronously push the items/data. It represents an object similar to Stream which can emit zero or more data, can send error message, whose speed can be controlled while emitting a set of data, can send finite as well as infinite data.
* **Observers and Subscribers(Consumer)**: That consume the data stream. The only difference between an Observer and a Subscriber is that a Subscriber class has the methods to unsubscribe/resubscribe independently without the need of the observerable methods. It subscribes to Observable's data of sequence and reacts per item of the observables. Observers are notified whenever Observable emits a data. An Observer handles data one by one.
* **Operators**: That transform the data stream

1. **Create() =>** The create() method is used to create a new Observable that can emit items.
2. **Observable.just() =>** Observable.just() emits whatever is present inside the just function. It can take between 2 to 9 parameters. If you pass a List/Array in just() it’ll emit the List/Array only.
3. **onSubscribe() =>** Method will be called when an Observer subscribes to Observable.
4. **Call =>** The call method is where the items are pushed on an instance of the subscriber.
5. **onNext() =>** The onNext() method gets the current value. Each of the items are emitted in onNext() one by one.
6. **onComplete() =>** The onComplete() method gets triggered when there is no more data left to be sent by the observable.
7. **onError() =>**  The onError() method gets triggered in case an exception occurs.
8. **Subscribe() =>**  For the Observer and Subscriber to listen to the data stream emitted by the Observable they need to be subscribed using the subscribe() method as
9. **subscribeOn(**Schedulers.io() or Schedulers.Main())  **=>**
10. **observeOn(Your thread)) =>**
11. **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.

import rx.Observable;

import rx.Observer;

import rx.Subscriber;

**public class MyRxClass {**

public static void main(String[] args) {

//Create Observable

**Observable<String> createObserver = Observable.create(new Observable.OnSubscribe<String>() {**

List<String> numbers = new ArrayList<>();

numbers.add(“Ok”);

numbers.add(“Not Ok”);

Observable<String> fromObservable = Observable.from(numbers);

Observable<String> justObservable = Observable.just(“Abhishek”, “Run”, “Rai”);

justObservable.subscribe(intSubscriber);

//publish the data to subscriper

@Override

**public void call(Subscriber<? super String> subscriber) {**

subscriber.onNext("Hello World”); // Emit the data once changes

subscriber.onCompleted(); // call once data imitation has been done

}

});

// Create subscriber

Subscriber<String> mySubscriber = new Subscriber<String>() {

@Override

public void onNext(String s) { //Call when data set has been chgnage

System.out.println("MySubscriber onNext(): "+ s);

}

@Override

public void onCompleted() { // call when dataset change/emit has been done/compelted

System.out.println("Subscriber completed");

}

@Override

public void onError(Throwable e) { //if any error will accrue at time of data emit

System.out.println("OnError");

}

};

// Create subscriber

**Observer<String> myObserver = new Observer<String>() {**

@Override

**public void onNext(String s) {**

System.out.println("MyObserver onNext(): "+ s);

}

@Override

**public void onCompleted() {**

System.out.println("Observer completed");

}

@Override

**public void onError(Throwable e) {**

}

};

**createObserver.subscribe(mySubscriber);** // Register subscriber with observable

**createObserver.subscribe(myObserver); //** Register subscriber with observable

}

}

**Example: 1**

import androidx.appcompat.app.AppCompatActivity

import android.os.Bundle

import android.util.Log

import io.reactivex.Observable

import io.reactivex.Observer

import io.reactivex.disposables.Disposable

import kotlinx.android.synthetic.main.activity\_main.\*

class MainActivity : AppCompatActivity() {

  private var TAG = "MainActivity"

  override fun onCreate(savedInstanceState: Bundle?) {

      super.onCreate(savedInstanceState)

      setContentView(R.layout.activity\_main)

//Start the stream when the button is clicked//

       button.setOnClickListener { startRStream()

}

  }

  private fun startRStream() {

//Create an Observable//

  val myObservable = getObservable()

//Create an Observer//

  val myObserver = getObserver()

//Subscribe myObserver to myObservable//

  myObservable.subscribe(myObserver)

  }

  private fun getObserver(): Observer<String> {

      return object : Observer<String> {

          override fun onSubscribe(d: Disposable) {

          }

//Every time onNext is called, print the value to Android Studio’s Logcat//

          override fun onNext(s: String) {

              Log.d(TAG, "onNext: $s")

          }

//Called if an exception is thrown//

          override fun onError(e: Throwable) {

              Log.e(TAG, "onError: " + e.message)

          }

//When onComplete is called, print the following to Logcat//

          override fun onComplete() {

              Log.d(TAG, "onComplete")

          }

      }

  }

//Give myObservable some data to emit

  private fun getObservable(): Observable<String> {

      return Observable.just("1", "2", "3", "4", "5")

  }

}

**Example: 2**

import android.os.Bundle

import androidx.appcompat.app.AppCompatActivity

import io.reactivex.rxkotlin.subscribeBy

import io.reactivex.rxkotlin.toObservable

import kotlinx.android.synthetic.main.activity\_main.\*

class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {

      super.onCreate(savedInstanceState)

      setContentView(R.layout.activity\_main)

//Start the stream when the button is clicked//

      button.setOnClickListener { startRStream() }

  }

  private fun startRStream() {

      val list = listOf("1", "2", "3", "4", "5")

//Apply the toObservable() extension function//

      list.toObservable()

//Construct your Observer using the subscribeBy() extension function//

              .subscribeBy(

                      onNext = { println(it) },

                      onError = { it.printStackTrace() },

                      onComplete = { println("onComplete!") }

              )

  }

}

**Example: 3**

import android.support.v7.app.AppCompatActivity;

import android.os.Bundle;

import android.util.Log;

import info.androidhive.rxandroidexamples.R;

import io.reactivex.Observable;

import io.reactivex.Observer;

import io.reactivex.android.schedulers.AndroidSchedulers;

import io.reactivex.disposables.Disposable;

import io.reactivex.functions.Predicate;

import io.reactivex.schedulers.Schedulers;

public class Example3Activity extends AppCompatActivity {

    /\*\*

     \* Basic Observable, Observer, Subscriber example

     \* Observable emits list of animal names

     \* You can see Disposable introduced in this example

     \*/

    private static final String TAG = Example3Activity.class.getSimpleName();

    private Disposable disposable;

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity\_example3);

        // observable

        Observable<String> animalsObservable = getAnimalsObservable();

        // observer

        Observer<String> animalsObserver = getAnimalsObserver();

        // observer subscribing to observable

        animalsObservable

                .subscribeOn(Schedulers.io())

                .observeOn(AndroidSchedulers.mainThread())

                .filter(new Predicate<String>() {

                    @Override

                    public boolean test(String s) throws Exception {

                        return s.toLowerCase().startsWith("b");

                    }

                })

                .subscribeWith(animalsObserver);

    }

    private Observer<String> getAnimalsObserver() {

        return new Observer<String>() {

            @Override

            public void onSubscribe(Disposable d) {

                Log.d(TAG, "onSubscribe");

                disposable = d;

            }

            @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, "All items are emitted!");

            }

        };

    }

    private Observable<String> getAnimalsObservable() {

        return Observable.fromArray(

                "Ant", "Ape",

                "Bat", "Bee", "Bear", "Butterfly",

                "Cat", "Crab", "Cod",

                "Dog", "Dove",

                "Fox", "Frog");

    }

    @Override

    protected void onDestroy() {

        super.onDestroy();

        // don't send events once the activity is destroyed

        disposable.dispose();

    }

}