**[1. Concat()](https://www.androidhive.info/RxJava/rxjava-operators-concat-merge/" \l "concat)**

[Concat](http://reactivex.io/documentation/operators/concat.html) operator combines output of two or more Observables into a single Observable. Concat operator always maintains the sequential execution without interleaving the emissions. So the first Observables completes its emission before the second starts and so forth if there are more observables.

Let’s say we have two separate Observables that emits **Male** and **Female** users. When Concat operator is used, the both Observables will be combined and act as single Observable.

|  |
| --- |
| Observable                  .concat(getMaleObservable(), getFemaleObservable())                  .observeOn(AndroidSchedulers.mainThread())                  .subscribe(new Observer<User>() {                      @Override                      public void onSubscribe(Disposable d) {                      }                        @Override                      public void onNext(User user) {                          Log.e(TAG, user.getName() + ", " + user.getGender());                      }                        @Override                      public void onError(Throwable e) {                        }                        @Override                      public void onComplete() {                        }                  });    private Observable<User> getFemaleObservable() {          String[] names = new String[]{"Lucy", "Scarlett", "April"};            final List<User> users = new ArrayList<>();          for (String name : names) {              User user = new User();              user.setName(name);              user.setGender("female");                users.add(user);          }          return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  Thread.sleep(1000);                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      }        private Observable<User> getMaleObservable() {          String[] names = new String[]{"Mark", "John", "Trump", "Obama"};            final List<User> users = new ArrayList<>();            for (String name : names) {              User user = new User();              user.setName(name);              user.setGender("male");                users.add(user);          }          return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  Thread.sleep(500);                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      } |

You can see the sequential order is maintained while emitting the items. Male users emitted first then Female users.

**Output:**

|  |
| --- |
| Mark, male  John, male  Trump, male  Obama, male  Lucy, female  Scarlett, female  April, female |

**[2. Merge()](https://www.androidhive.info/RxJava/rxjava-operators-concat-merge/" \l "merge)**

[Merge](http://reactivex.io/documentation/operators/merge.html) also merges multiple Observables into a single Observable but it won’t maintain the sequential execution.

Let’s consider the same above example but with **Merge** operator.

|  |
| --- |
| Observable                  .merge(getMaleObservable(), getFemaleObservable())                  .observeOn(AndroidSchedulers.mainThread())                  .subscribeWith(new Observer<User>() {                      @Override                      public void onSubscribe(Disposable d) {                      }                        @Override                      public void onNext(User user) {                          Log.e(TAG, user.getName() + ", " + user.getGender());                      }                        @Override                      public void onError(Throwable e) {                        }                        @Override                      public void onComplete() {                        }                  });    private Observable<User> getFemaleObservable() {          String[] names = new String[]{"Lucy", "Scarlett", "April"};            final List<User> users = new ArrayList<>();          for (String name : names) {              User user = new User();              user.setName(name);              user.setGender("female");                users.add(user);          }          return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  Thread.sleep(1000);                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      }        private Observable<User> getMaleObservable() {          String[] names = new String[]{"Mark", "John", "Trump", "Obama"};            final List<User> users = new ArrayList<>();            for (String name : names) {              User user = new User();              user.setName(name);              user.setGender("male");                users.add(user);          }          return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  Thread.sleep(500);                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      } |

If you run this example, you can see the emissions are random. The Male and Female emissions doesn’t followed the sequential order.

**Output:**

|  |
| --- |
| Mark, male  Lucy, female  John, male  Trump, male  Scarlett, female  Obama, male  April, female |

###### Output:

|  |
| --- |
| Mark, male  John, male  Trump, male  Obama, male  Lucy, female  Scarlett, female  April, female |

## [2. Merge()](https://www.androidhive.info/RxJava/rxjava-operators-concat-merge/" \l "merge)

[Merge](http://reactivex.io/documentation/operators/merge.html) also merges multiple Observables into a single Observable but it won’t maintain the sequential execution.

Let’s consider the same above example but with **Merge** operator.

|  |
| --- |
| Observable                  .merge(getMaleObservable(), getFemaleObservable())                  .observeOn(AndroidSchedulers.mainThread())                  .subscribeWith(new Observer<User>() {                      @Override                      public void onSubscribe(Disposable d) {                      }                        @Override                      public void onNext(User user) {                          Log.e(TAG, user.getName() + ", " + user.getGender());                      }                        @Override                      public void onError(Throwable e) {                        }                        @Override                      public void onComplete() {                        }                  });    private Observable<User> getFemaleObservable() {          String[] names = new String[]{"Lucy", "Scarlett", "April"};            final List<User> users = new ArrayList<>();          for (String name : names) {              User user = new User();              user.setName(name);              user.setGender("female");                users.add(user);          }          return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  Thread.sleep(1000);                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      }        private Observable<User> getMaleObservable() {          String[] names = new String[]{"Mark", "John", "Trump", "Obama"};            final List<User> users = new ArrayList<>();            for (String name : names) {              User user = new User();              user.setName(name);              user.setGender("male");                users.add(user);          }          return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  Thread.sleep(500);                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      } |

**[1. Map()](https://www.androidhive.info/RxJava/map-flatmap-switchmap-concatmap/" \l "map)**

**Map** operator transform each item emitted by an Observable and emits the modified item.

Let’s say we have an Observable that makes a network call (assume the network call is made) and emits the **User** objects with **name** and **gender**. But in our requirement we need an email address to be present for each user, which is missing in the network response. Then we can alter each User object by applying **Map()** operation.

* **getUsersObservable()** : assume this method is making a network call and fetching user objects. This returns an Observable that emits User objects with **name** and **gender**.
* **map()** operator applies **Function<User, User>** on each User object and **adds email address** and returns the modified User object.

|  |
| --- |
| import io.reactivex.Observable;  import io.reactivex.ObservableEmitter;  import io.reactivex.ObservableOnSubscribe;  import io.reactivex.Observer;  import io.reactivex.android.schedulers.AndroidSchedulers;  import io.reactivex.disposables.Disposable;  import io.reactivex.functions.Function;  import io.reactivex.schedulers.Schedulers;    public class MapOperatorActivity extends AppCompatActivity {        private static final String TAG = MapOperatorActivity.class.getSimpleName();      private Disposable disposable;        @Override      protected void onCreate(Bundle savedInstanceState) {          super.onCreate(savedInstanceState);          setContentView(R.layout.activity\_map\_operator);            getUsersObservable()                  .subscribeOn(Schedulers.io())                  .observeOn(AndroidSchedulers.mainThread())                  .map(new Function<User, User>() {                      @Override                      public User apply(User user) throws Exception {                          // modifying user object by adding email address                          // turning user name to uppercase                          user.setEmail(String.format("%s@rxjava.wtf", user.getName()));                          user.setName(user.getName().toUpperCase());                          return user;                      }                  })                  .subscribe(new Observer<User>() {                      @Override                      public void onSubscribe(Disposable d) {                          disposable = d;                      }                        @Override                      public void onNext(User user) {                          Log.e(TAG, "onNext: " + user.getName() + ", " + user.getGender() + ", " + user.getAddress().getAddress());                      }                        @Override                      public void onError(Throwable e) {                          Log.e(TAG, "onError: " + e.getMessage());                      }                        @Override                      public void onComplete() {                          Log.e(TAG, "All users emitted!");                      }                  });      }        /\*\*       \* Assume this method is making a network call and fetching Users       \* an Observable that emits list of users       \* each User has name and email, but missing email id       \*/      private Observable<User> getUsersObservable() {          String[] names = new String[]{"mark", "john", "trump", "obama"};            final List<User> users = new ArrayList<>();          for (String name : names) {              User user = new User();              user.setName(name);              user.setGender("male");                users.add(user);          }          return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      }        @Override      protected void onDestroy() {          super.onDestroy();          disposable.dispose();      }  } |
| public class User {      String name;      String email;      String gender;      Address address;        // getters and setters  } |

If you run the example, you can notice email address added to each User. You can also notice that the name is modified to uppercase.

**Output**

|  |
| --- |
| onSubscribe  onNext: MARK, male, mark@rxjava.wtf  onNext: JOHN, male, john@rxjava.wtf  onNext: TRUMP, male, trump@rxjava.wtf  onNext: OBAMA, male, obama@rxjava.wtf  All users emitted! |

**[2. FlatMap()](https://www.androidhive.info/RxJava/map-flatmap-switchmap-concatmap/" \l "flatmap)**

To better understand FlatMap, consider a scenario where you have a **network call to fetch Users** with name and gender. Then you have **another network that gives you address** of each user. Now the requirement is to create an **Observable that emits Users with name, gender and address** properties. To achieve this, you need to get the users first, then make separate network call for each user to fetch his address. This can be done easily using FlatMap operator.

* **getUsersObservable()** : assume it makes a network call and returns an Observable that emits User (name and gender) objects.
* **getAddressObservable()** : assume it makes another network call just to fetch user address. This also returns an Observable that emits User by adding address node to existing name and gender.
* **flatMap()** operator makes **getAddressObservable()** call each time a User is emitted and returns an Observable that emits User including the address filed.
* Finally flatMap() returns an Observable by **merging two Observables** together.
* **Thread.sleep(sleepTime)**; added here to simulate network latency.

|  |
| --- |
| import android.os.Bundle;  import android.support.v7.app.AppCompatActivity;  import android.util.Log;    import java.util.ArrayList;  import java.util.List;  import java.util.Random;    import info.androidhive.rxandroidexamples.R;  import info.androidhive.rxandroidexamples.operators.model.Address;  import info.androidhive.rxandroidexamples.operators.model.User;  import io.reactivex.Observable;  import io.reactivex.ObservableEmitter;  import io.reactivex.ObservableOnSubscribe;  import io.reactivex.Observer;  import io.reactivex.android.schedulers.AndroidSchedulers;  import io.reactivex.disposables.Disposable;  import io.reactivex.functions.Function;  import io.reactivex.schedulers.Schedulers;    public class FlatMapActivity extends AppCompatActivity {        private static final String TAG = FlatMapActivity.class.getSimpleName();        private Disposable disposable;        @Override      protected void onCreate(Bundle savedInstanceState) {          super.onCreate(savedInstanceState);          setContentView(R.layout.activity\_flat\_map);            getUsersObservable()                  .subscribeOn(Schedulers.io())                  .observeOn(AndroidSchedulers.mainThread())                  .flatMap(new Function<User, Observable<User>>() {                        @Override                      public Observable<User> apply(User user) throws Exception {                            // getting each user address by making another network call                          return getAddressObservable(user);                      }                  })                  .subscribe(new Observer<User>() {                      @Override                      public void onSubscribe(Disposable d) {                          Log.e(TAG, "onSubscribe");                          disposable = d;                      }                        @Override                      public void onNext(User user) {                          Log.e(TAG, "onNext: " + user.getName() + ", " + user.getGender() + ", " + user.getAddress().getAddress());                      }                        @Override                      public void onError(Throwable e) {                        }                        @Override                      public void onComplete() {                          Log.e(TAG, "All users emitted!");                      }                  });      }        /\*\*       \* Assume this as a network call       \* returns Users with address filed added       \*/      private Observable<User> getAddressObservable(final User user) {            final String[] addresses = new String[]{                  "1600 Amphitheatre Parkway, Mountain View, CA 94043",                  "2300 Traverwood Dr. Ann Arbor, MI 48105",                  "500 W 2nd St Suite 2900 Austin, TX 78701",                  "355 Main Street Cambridge, MA 02142"          };            return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          Address address = new Address();                          address.setAddress(addresses[new Random().nextInt(2) + 0]);                          if (!emitter.isDisposed()) {                              user.setAddress(address);                                  // Generate network latency of random duration                              int sleepTime = new Random().nextInt(1000) + 500;                                Thread.sleep(sleepTime);                              emitter.onNext(user);                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      }        /\*\*       \* Assume this is a network call to fetch users       \* returns Users with name and gender but missing address       \*/      private Observable<User> getUsersObservable() {          String[] maleUsers = new String[]{"Mark", "John", "Trump", "Obama"};            final List<User> users = new ArrayList<>();            for (String name : maleUsers) {              User user = new User();              user.setName(name);              user.setGender("male");                users.add(user);          }            return Observable                  .create(new ObservableOnSubscribe<User>() {                      @Override                      public void subscribe(ObservableEmitter<User> emitter) throws Exception {                          for (User user : users) {                              if (!emitter.isDisposed()) {                                  emitter.onNext(user);                              }                          }                            if (!emitter.isDisposed()) {                              emitter.onComplete();                          }                      }                  }).subscribeOn(Schedulers.io());      }        @Override      protected void onDestroy() {          super.onDestroy();          disposable.dispose();      }  } |

If you run this example you can see the output like below. Here, name and gender are fetched from one observable and address is fetched from another observable. Also notice that the **order of items is not maintained** as source observable. You can see the order changed each time you run this example.

**Output**

|  |
| --- |
| onSubscribe  onNext: John, male, 2300 Traverwood Dr. Ann Arbor, MI 48105  onNext: Obama, male, 2300 Traverwood Dr. Ann Arbor, MI 48105  onNext: Mark, male, 1600 Amphitheatre Parkway, Mountain View, CA 94043  onNext: Trump, male, 2300 Traverwood Dr. Ann Arbor, MI 48105  All users emitted! |