FACULTY OF ENGINEERING
AND ARCHITECTURE

REACTIVE ANDROID APP ARCHITECTURE

Pieter Simoens

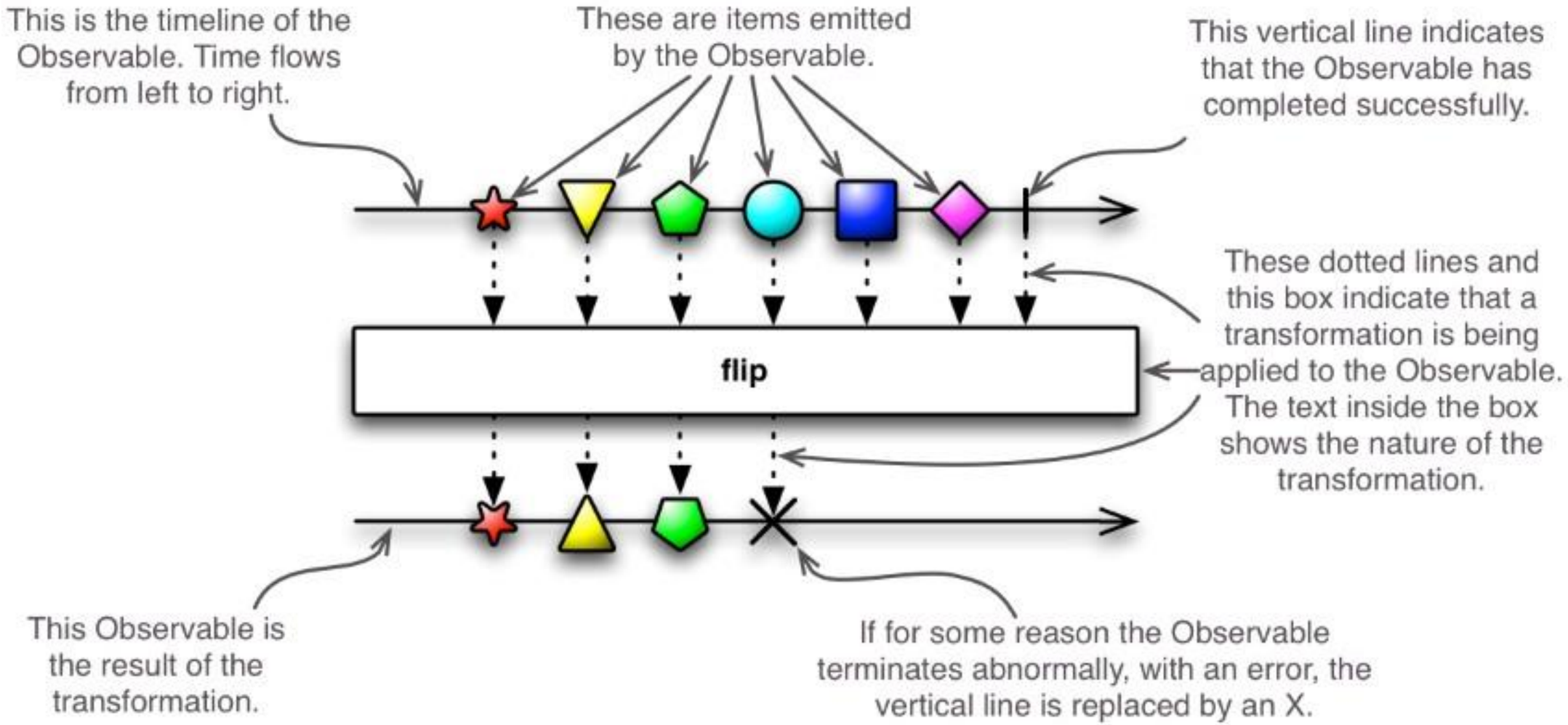
Academic Year 2017-2018

TABLE OF CONTENTS

- Reactive programming
  - concepts
  - RXJava2 constructs
- Towards a reactive Android app architecture
  - MVC → MVP → MVVM
- Hands-on exercise on RXJava2

RXJAVA2

REACTIVE PROGRAMMING



- Data is processed as streams
- Emitted data = value, error or “completed” signal
- Reactive objects consume and act on the streams, actions depend on what stream has emitted

REACTIVE PROGRAMMING

- pretty much everything is a stream of data
  - traditional data types (array, JSON…)
  - variables, properties, caches, user clicks, user swipes…
- govern the exchange of stream data across asynchronous boundaries (e.g. move to another thread or to thread pool)
- Why? Avoid challenges of asynchronous programming
  - Handling failed async requests
  - Chaining async requests
  - The user already closed the view to be updated
  - Switching threads to update UI
  - Cancelling a pending HTTP request

WHAT IS RXJAVA?

- Library for composing asynchronous and event-based programs by using observable sequences
- Implementation of Reactive Streams initiative
  - www.reactive-streams.org
  - other products: Java 9 Flow API, Akka Stream, MongoDB, Vert.x
- In these slides: RXJava2
  - huge rewrite of RXJava1, breaking APIs
  - !beware of existing tutorials/StackOverflow

MAIN RXJAVA2 PLAYERS

- `Observable` or `Flowable` – producers of data
- `Observer` or `Subscriber` – consumers of data
- `Operator` – en-route data transformation
- `Scheduler` – multi-threading support
- `Subject` or `Processor` – implements both producer and consumer

CREATING AN OBSERVABLE

```
Observable<Integer> observable = Observable.create(new ObservableOnSubscribe<Integer>()
{
  @Override

  public void subscribe(ObservableEmitter<Integer> e) throws Exception {
      //Use onNext to emit each item in the stream//
      e.onNext(1);
      e.onNext(2);
      e.onNext(3);
      e.onNext(4);

      //Once the Observable has emitted all items in the sequence, call onComplete//
      e.onComplete();
  }
 }
);
```

More concise alternatives for creating an Observable:

```
Observable.fromArray(…), Observable.range(0,5),
Observable.just(“Hello world”),
Observable.interval(1, TimeUnit.SECONDS)
```

CREATING AN OBSERVABLE: ALTERNATIVES

With a lambda function

```
Observable<Integer> observable = Observable.create(
     subscriber -> {
           for(Beer b : beerStock):
               subscriber.onNext(b);

     subscriber.onComplete();
});
```

Specific creation functions:

```
Observable.fromArray(…), Observable.range(0,5),
Observable.just(“Hello world”),
Observable.interval(1, TimeUnit.SECONDS)
```

CREATING AN OBSERVER

```
Observer<Integer> observer = new Observer<Integer>() {
  @Override
  public void onSubscribe(Disposable d) {
      Log.e(TAG, "onSubscribe: "); //d.dispose()
  }
   @Override
  public void onNext(Integer value) {
      Log.e(TAG, "onNext: " + value);
  }
   @Override
  public void onError(Throwable e) {
      Log.e(TAG, "onError: ");
  }
   @Override
  public void onComplete() {
      Log.e(TAG, "onComplete: All Done!");
  }
};

//Create our subscription//
observable.subscribe(observer);
```

Disposable object created by the observable that can be used to unsubscribe

Streaming starts here!

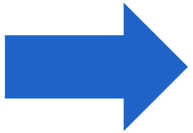
CREATING AN OBSERVER: WITH LAMBDA FUNCTIONS

```
Observable<Beer> observableBeer = Observable.create(…);

observableBeer
     .skip(1)
     .take(3)
     .filter(beer -> “USA”.equals(beer.country))
     .map(beer -> beer.name + “: $” + beer.price)
     .subscribe(
          beer –> System.out.println(beer),
          err -> System.out.println(err),
          () -> System.out.println(“Streaming is complete”),
          disposable -> System.out.println(“Someone just subscribed!”)
     );
);
```

observer

RXBINDING: CREATE OBSERVABLES FROM ANY VIEW

```
Button button =
(Button)findViewById(R.id.button);
button.setOnClickListener(new
View.OnClickListener() {
    @Override
    public void onClick(View v) {
        //Perform some work//
    }
});
```



```
Button button = (Button)
findViewById(R.id.button);
RxView.clicks(button)
      .subscribe(aVoid -> {
        //Perform some work here//
      });
```

```
final EditText name = (EditText) v.findViewById(R.id.name);
//Create a TextWatcher and specify that this TextWatcher should be called whenever the
EditText's content changes//
name.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after) {
    }
    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count) {
        //Perform some work//
    }

    @Override
    public void afterTextChanged(Editable s) {
    }
});
```

```
RxTextView.textChanges(editText)
    .subscribe(charSequence -> {
        //Perform some work here//
    });
```

HOT AND COLD OBSERVABLES

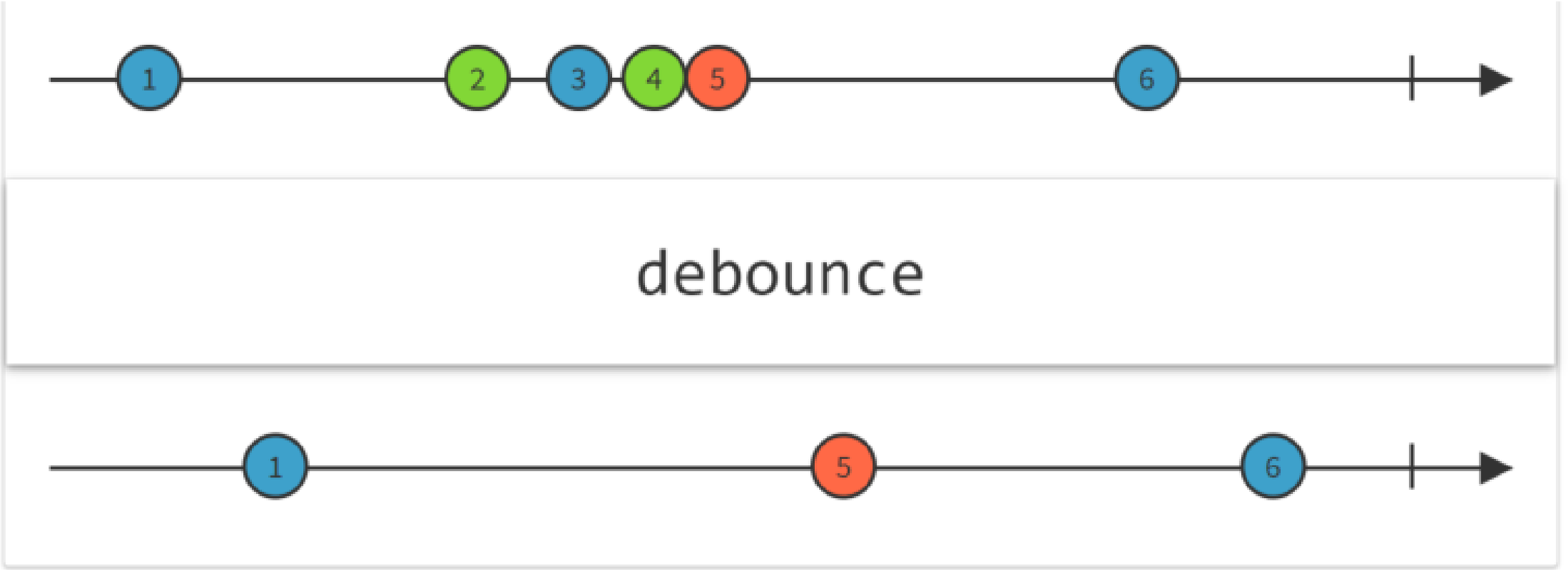
Hot

- Produces a stream even if no-one cares
- Mouse events, stock prices, accelerometer

Cold

- Produces a stream when someone asks for it

DEBOUNCING



- throttle number of events emitted
- only emit an item from an Observable if a timespan has passed without emitting another item
- e.g. accelerometer

SCHEDULERS

- Default, RX is single-threaded
- `Observable` and chain of operators will notify their observers on the same thread on which its `subscribe()` method is called

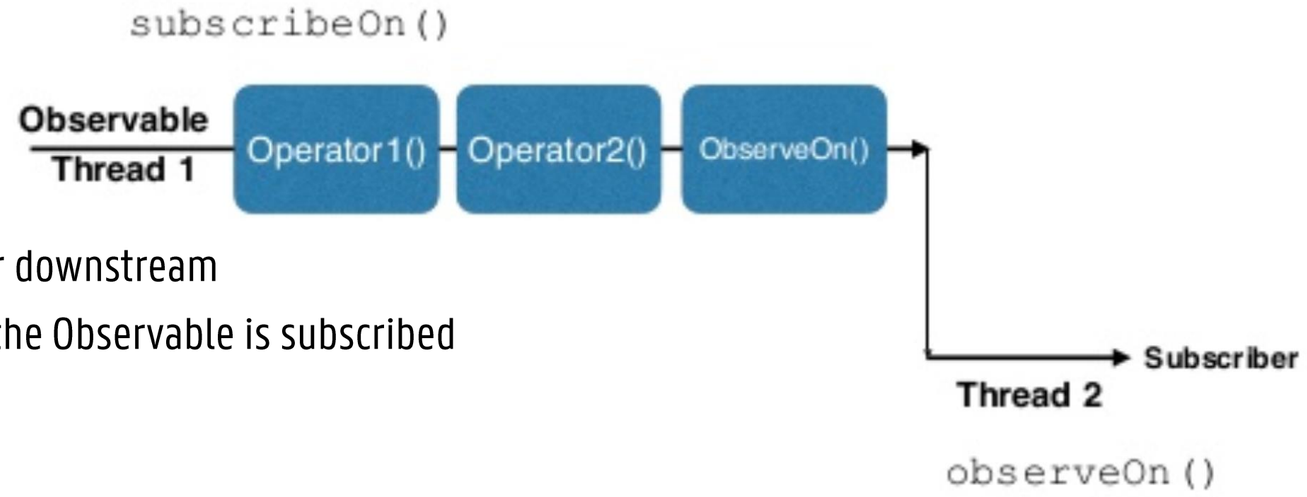
`subscribeOn(strategy)` – run **`Observable`** in a separate thread
e.g. file I/O (don't block the main UI)

`observeOn(strategy)` – run **`Observer`** in a separate thread
e.g. update Android UI

MULTI-THREADING STRATEGIES

- Schedulers.computation – for computations: # of threads <= # of CPU cores
- Schedulers.io() – for long running communications, backed by a thread pool
- Schedulers.trampoline() – queue work on the current thread
- AndroidSchedulers.mainThread() – handle data on the main thread (RxAndroid)
- …

SWITCHING BETWEEN THREADS



observeOn: changes the thread of all operators further downstream

subscribeOn: influences the thread that is used when the Observable is subscribed (position in the chain of operator does not matter)

```
onCreate(…){ //Activity lifecycle callback on the main thread
  just("Some String") // Computation
  .map(str -> str.length()) // Computation
  .map(length -> 2 * length) // Computation
  .subscribeOn(Schedulers.computation()) // -- changing the thread
  .subscribe(number -> Log.d("", "Number " + number));// Computation
}
```

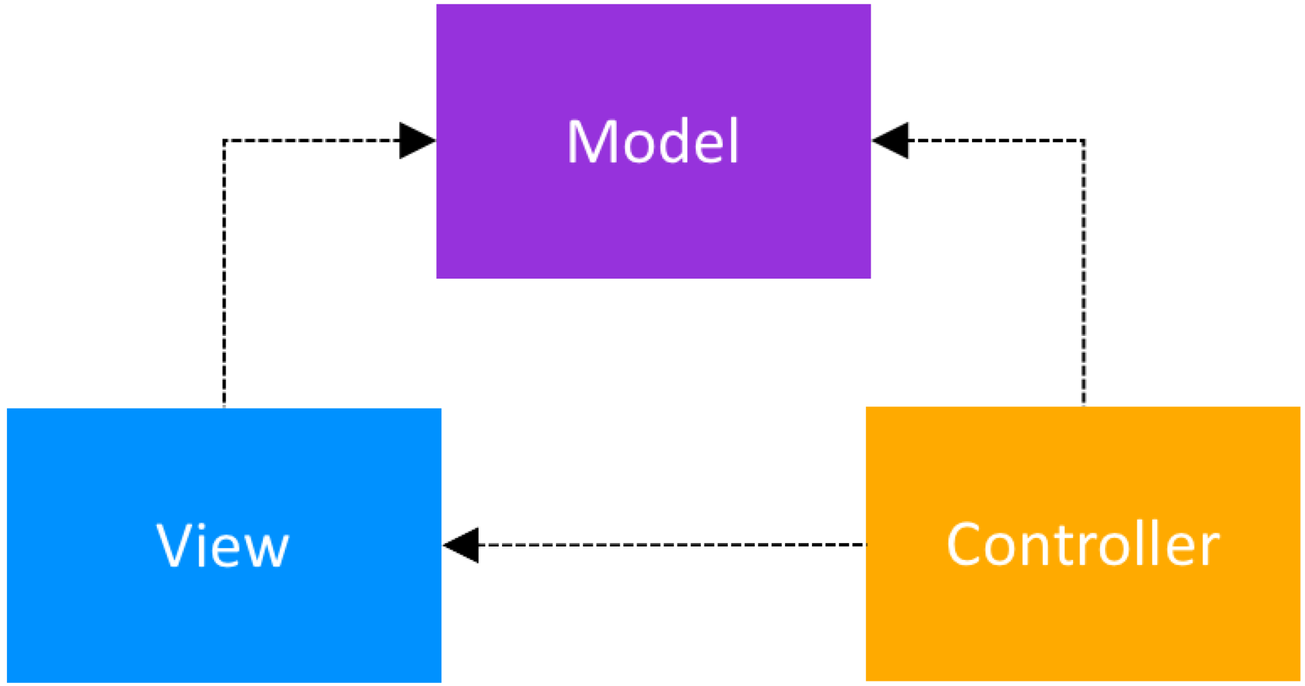
```
onCreate(…){ //Activity lifecycle callback on the main thread
  just("Some String") // UI
  .observeOn(Schedulers.computation()) //changing the threada
  .map(length -> 2 * length) // Computation
  .subscribe(number -> Log.d("", "Number " + number));// Computation
}
```

MODEL – VIEW - CONTROLLER
- PRESENTER
- VIEWMODEL

DESIGN PATTERNS FOR FRONT-END APPLICATIONS

- Several responsibilities
  - loading and persisting data
  - graphically presenting data
  - capturing user input and updating data
  - business logic
- Design pattern helps to ensure:
  - Testability
  - Modularity/Reusability
- Three patterns: MVC, MVP, MVVC

MODEL-VIEW-CONTROLLER (MVC)



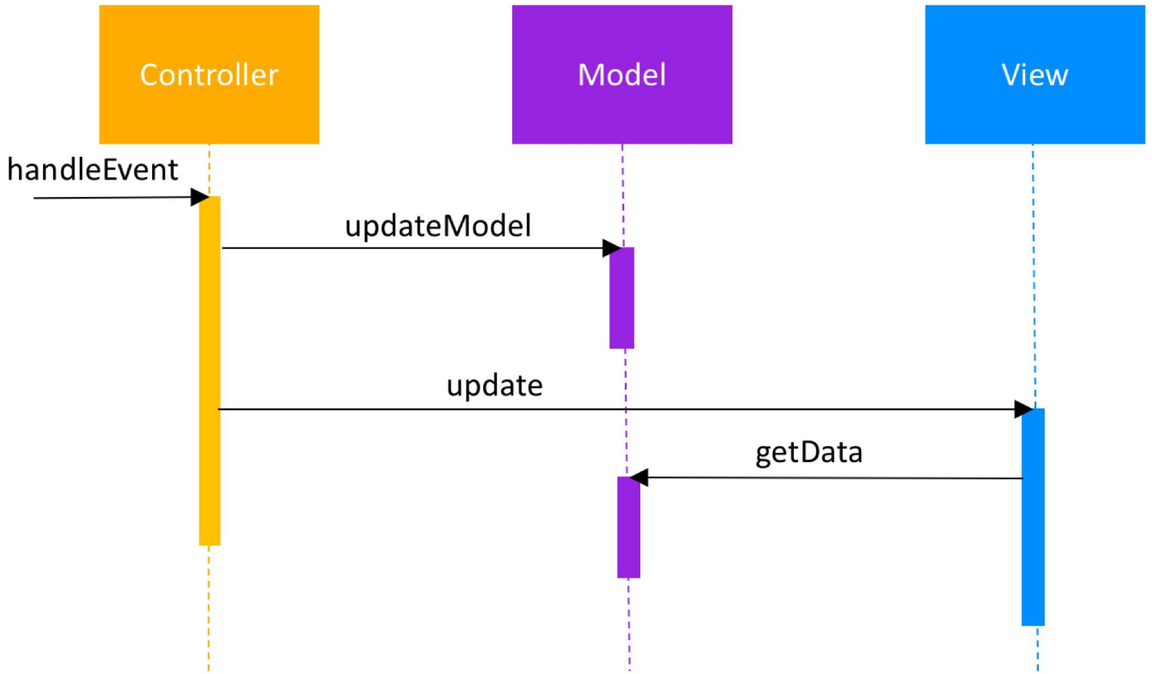
Model: data layer, business logic, network/database operations

View: render UI, communicate user interaction to controller

Controller: update model based on user interaction, update View if model data changes

MVC

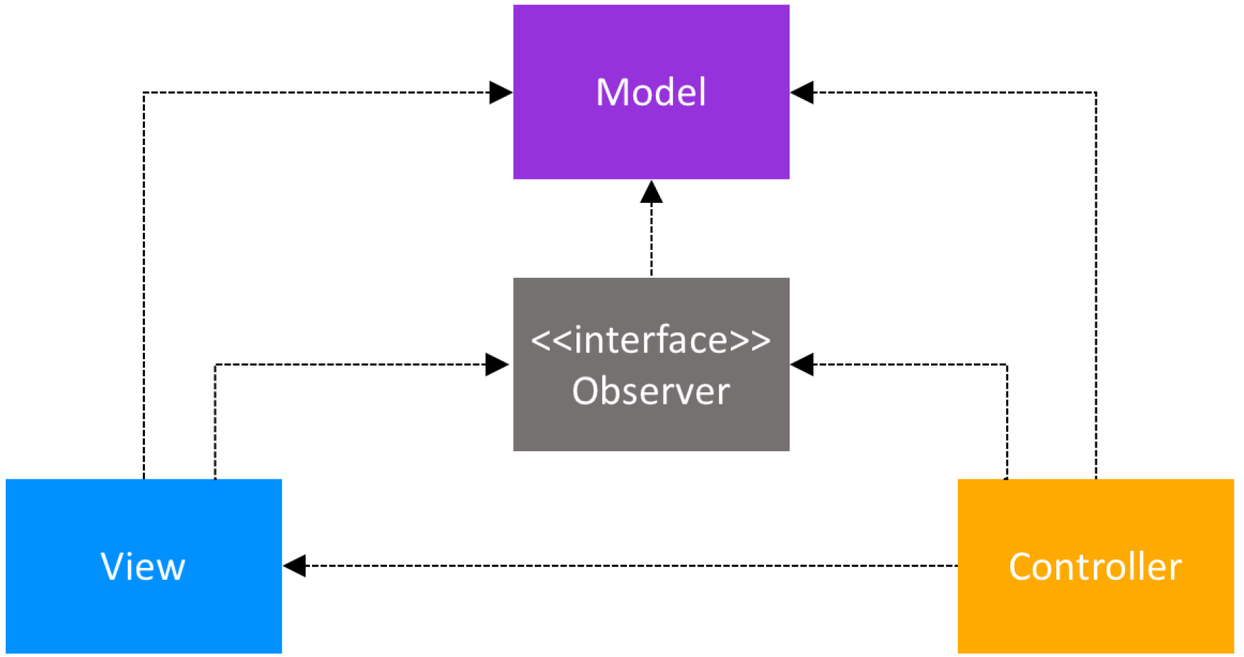
Passive model



Controller is only class that updates the model

View requests data from Model

Active model



Model notifies its observes that it was update

The view will then request data from the model

- Model has no dependencies and can be tested independently of UI
- View depends on both controller and on model → changes in UI logic might require updates in several classes
- Who decides on *how* to display the data? The model? Or the View?

MVC IN ANDROID

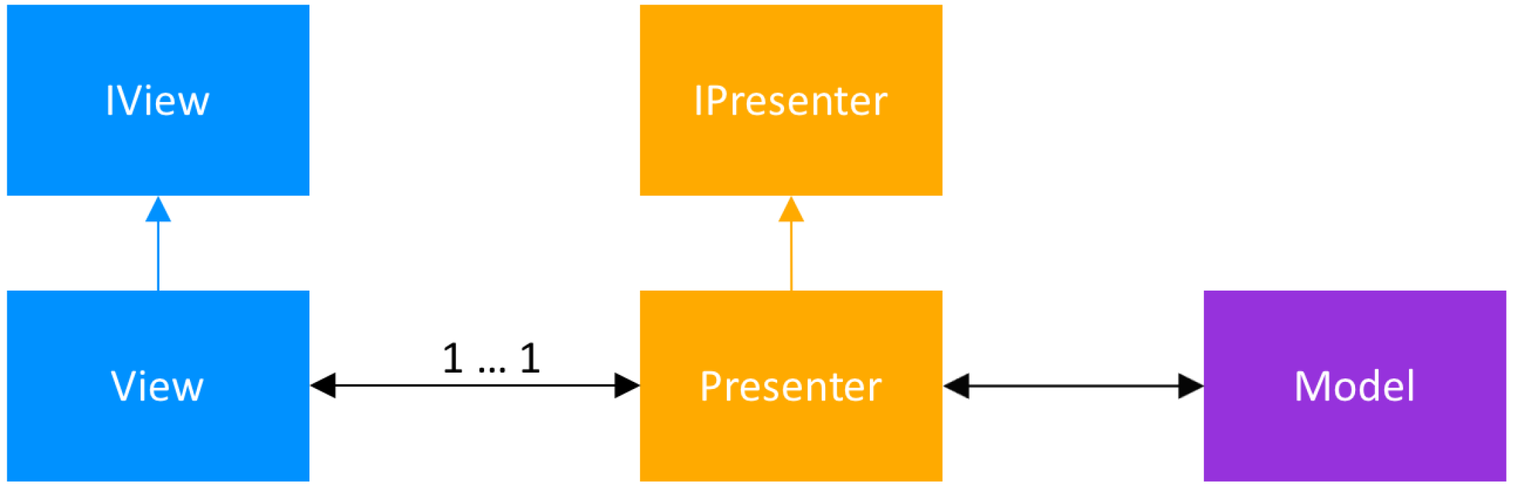
MVC IN ANDROID

```
public class TicTacToeActivity extends AppCompatActivity {
  private Board model; /* View Components referenced by the controller */
  private ViewGroup buttonGrid;
  ...

protected void onCreate(Bundle savedInstanceState) {
    ...
   setContentView(R.layout.tictactoe);
   winnerPlayerLabel = (TextView) findViewById(R.id.winnerPlayerLabel);
   …
   model = new Board(); }

 public void onCellClicked(View v) {
     Button button = (Button) v; int row = Integer.valueOf(tag.substring(0,1)); ...
     Player playerThatMoved = model.mark(row, col);
     if (playerThatMoved != null) {
          button.setText(playerThatMoved.toString());
          if (model.getWinner() != null) { winnerPlayerLabel.setText(playerThatMoved.toString());
                                           winnerPlayerViewGroup.setVisibility(View.VISIBLE);
      } } }

 private void reset() { …}
```

"View" and "Controller" functionality in same class!

MVC is not a natural fit for Android

MODEL-VIEW-PRESENTER (MVP)



Model: data layer, business logic, network/database operations
View: display data and notify the presenter about user actions
Presenter: retrieves data from the model, applies UI logic, manages the state of the view, reacts to user input notifications from the view

→ Presenter is essentially the controller, but not tied to the View, only to an interface
→ iView and iPresenter interfaces foster decoupling (and thus testability)
→ Purist vision: presenter should never have any references to any Android APIs or code

BASE INTERFACES FOR VIEW AND PRESENTER

Base view interface allows setting a Presenter

```
public interface iView<T> {
   void setPresenter(T presenter);
}
```

Base presenter interface allows the view to tell the presenter that it is ready to be updated

Some options for this interface:

```
public interface iPresenter {
   void subscribe();
   void unsubscribe();
}
```

```
public interface iPresenter {
   void onCreate();
   void onResume();
   void onPause();
   void onDestroy();
}
```

TIC-TAC-TOE, THE MVP VERSION

Activity/Fragment classes are only part of the View!


```
public interface TicTacToeView
 extends iBaseiBaseView {
  void showWinner(String winningPlayerLabel);
  void clearWinnerDisplay();
  void clearButtons();
  void setButtonText(int row, int col, String text);
}
```

```
public class TicTacToePresenter implements iPresenter {
   private TicTacToeView view;
   private Board model;
   public TicTacToePresenter(TicTacToeView view) {
      this.view = view;
   }
   public void onCreate() { model = new Board(); view.setPresenter(this) }
   public void onButtonSelected(int row, int col) {
     Player playerThatMoved = model.mark(row, col);
     if(playerThatMoved != null) {
       view.setButtonText(row, col, playerThatMoved.toString()); ...}
   public void onResetSelected() {
      view.clearWinnerDisplay(); view.clearButtons(); model.restart(); }
}
```

ADVANTAGES AND DISADVANTAGES OF MVP

+ Very good separation of concerns

- A lot of interfaces…
- Moving UI logic to the Presenter means that this becomes an all-knowing class that quickly grows

MODEL-VIEW-VIEWMODEL

(Data)Model: data layer, business logic, network/database operations

View: informs the ViewModel about the user's actions

ViewModel: expose streams of data relevant to the View

Difference with MVP?

- MVVM pattern mainly created to simply event driven programming of user interfaces
- ViewModel does not need to hold a reference to the View
- ViewModel exposes a stream of events to which the Views can bind to
    - All interfaces of the MVP pattern can be dropped

A VIEWMODEL COMPONENT

- provides the data for a specific UI component (Activity, Fragment…)
- handles communication with business part of data handling (calling other components to load data, forwarding user modifications)
- does not know about the View(s) that use its emitted data
- lifecycle aware: retained during configuration changes

EXAMPLE: FRAGMENT SHOWING USER DATA – THE VIEWMODEL

```
public class UserProfileViewModel extends ViewModel {
    private String userId;
    private LiveData<User> user;

    public void init(String userId) {
        this.userId = userId;
    }
    public LiveData<User> getUser() {
        return user;
    }
}
```

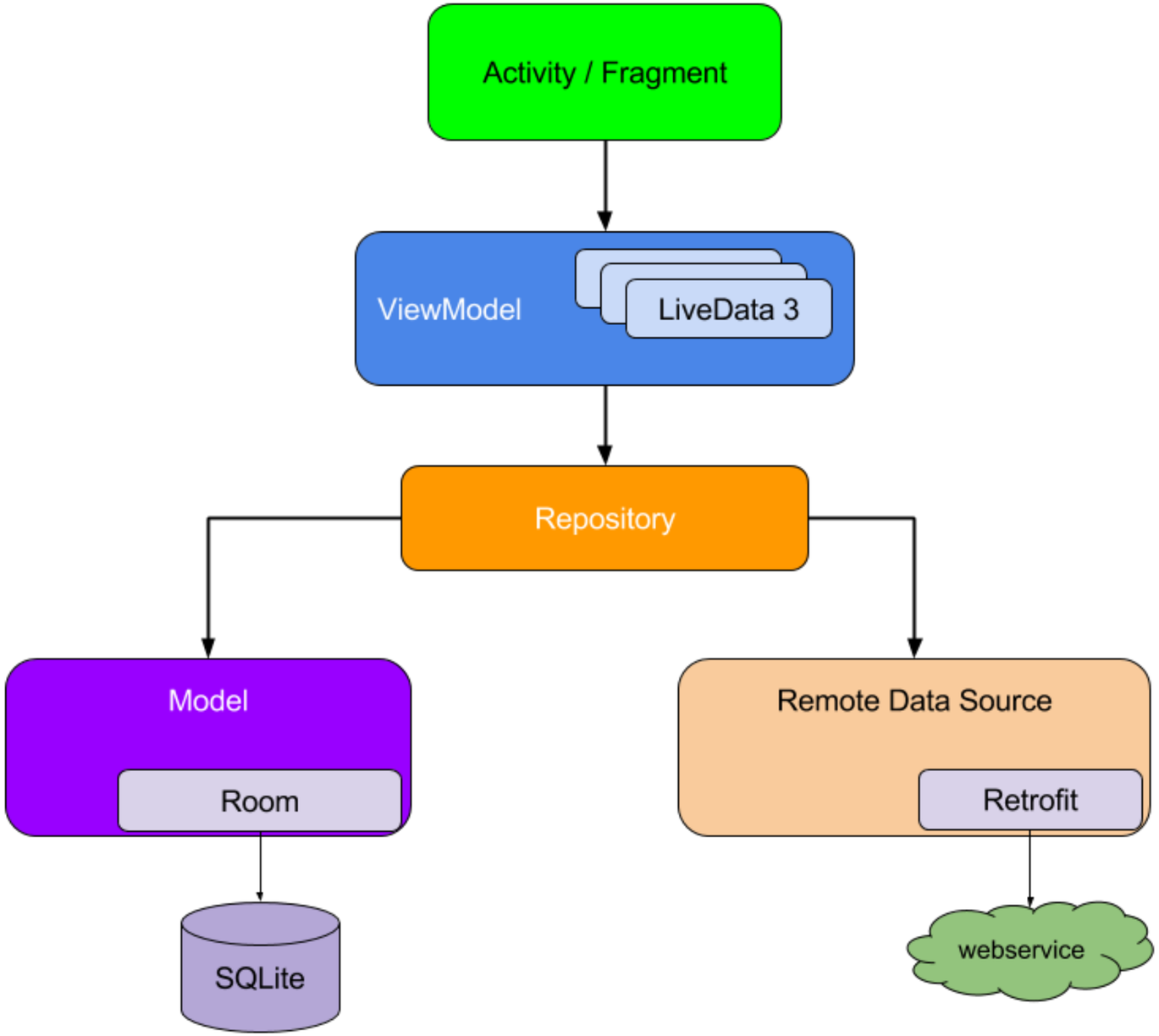
Livedata is an observable data holder that respects the lifecycle of it's observers and prevents memory leakage

EXAMPLE: FRAGMENT SHOWING USER DATA – THE VIEW

```
public class UserProfileFragment extends Fragment {
    private static final String UID_KEY = "uid";
    private UserProfileViewModel viewModel;

    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        String userId = getArguments().getString(UID_KEY);
        viewModel = ViewModelProviders.of(this).get(UserProfileViewModel.class);
        viewModel.init(userID);
        viewModel.getUser().observe(this ,user -> {
                //update UI
        }
    }

    @Override
    public View onCreateView(LayoutInflater inflater,
                 @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        return inflater.inflate(R.layout.user_profile, container, false);
    }
}
```

executed each time the user information is updated

ABSTRACTING THE MODEL DETAILS



EXERCISE

- Goal: observe user input to filter list shown in a RecyclerView
  - focus is on RXJava2 not on LiveData/ViewModel/...
  - many extensions possible, e.g. reactive network calls, etc...
- Assignment on Minerva