Android Application Development

CUSTOMIZED

LiveData

Android Jetpack (AndroidX)

- Jetpack encompasses a collection of Android libraries that incorporate best practices and provide backwards compatibility in your Android apps.
- Jetpack essentially defines a set of recommendations describing how an Android app project should be structured while providing a set of libraries and components that make it easier to conform with these guidelines with the goal of developing reliable apps with less coding and fewer errors.
  - LiveData
  - WorkManager
  - ViewModel
  - Room
  - CameraX

Mobile app user experiences

- In most cases, desktop apps have a single-entry point from a desktop or program launcher, then run as a single, monolithic process.
- Android apps, on the other hand, have a much more complex structure. A typical Android app contains multiple app components, including activities, fragments, services, content providers, and broadcast receivers.
- You declare most of these app components in your app manifest. The Android OS then uses this file to decide how to integrate your app into the device's overall user experience.
- Given that a properly-written Android app contains multiple components and that users often interact with multiple apps in a short period of time, apps need to adapt to different kinds of user-driven workflows and tasks.

Example

- Consider what happens when you share a photo in your favorite social networking app:
  - The app triggers a camera intent. The Android OS then launches a camera app to handle the request. At this point, the user has left the social networking app, but their experience is still seamless.
  - The camera app might trigger other intents, like launching the file chooser, which may launch yet another app.
  - Eventually, the user returns to the social networking app and shares the photo.

Example (cont)

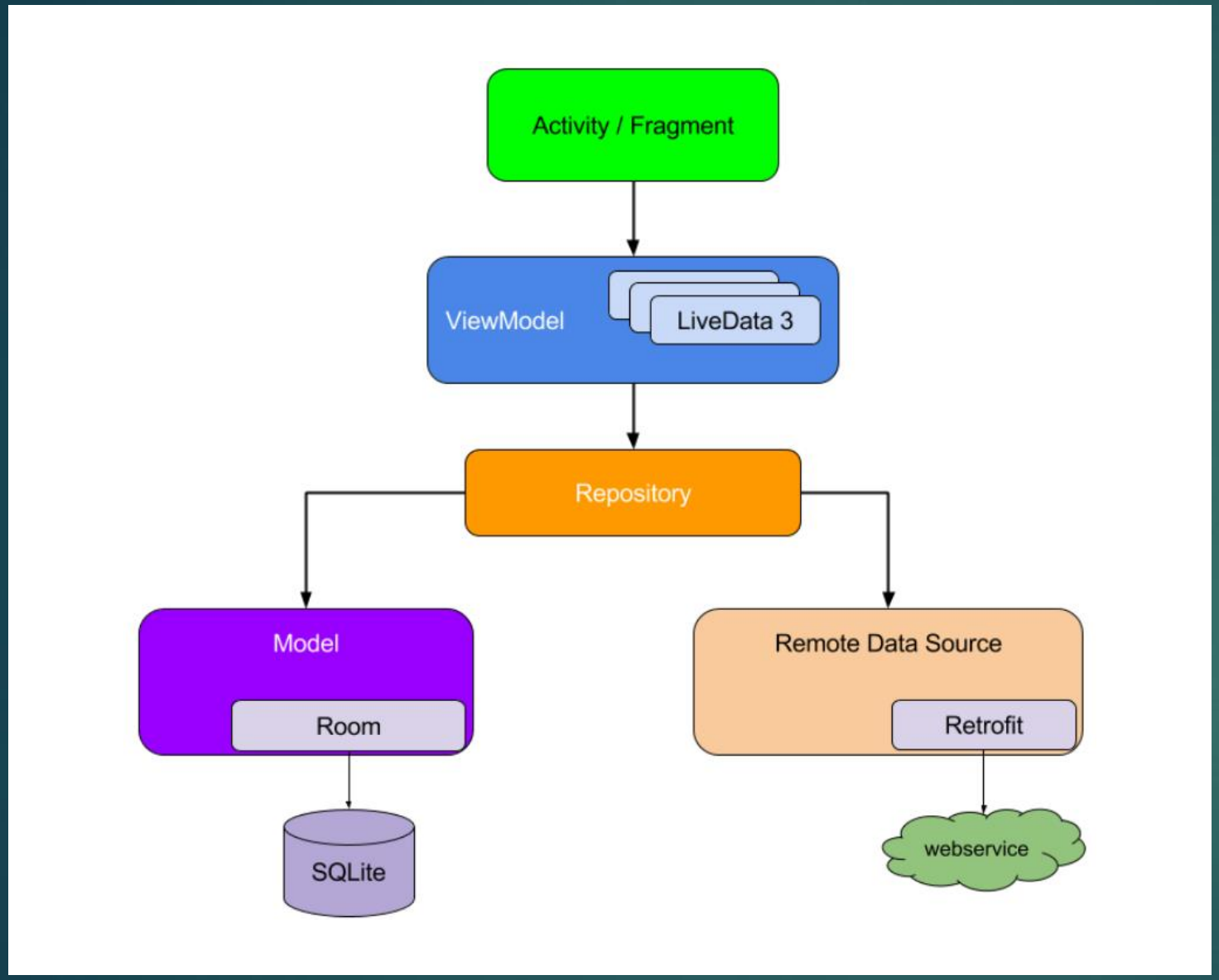
- At any point during the process, the user could be interrupted by a phone call or notification.
- After acting upon this interruption, the user expects to be able to return to, and resume, this photo-sharing process. This app-hopping behavior is common on mobile devices, so your app must handle these flows correctly.
- Keep in mind that mobile devices are also resource-constrained, so at any time, the operating system might kill some app processes to make room for new ones.
- Given the conditions of this environment, it's possible for your app components to be launched individually and out-of-order, and the operating system or user can destroy them at any time. Because these events aren't under your control, you shouldn't store any app data or state in your app components, and your app components shouldn't depend on each other.

Common architectural principles

- **Separation of concerns**
  - It's a common mistake to write all your code in an Activity or a Fragment.
  - These UI-based classes should only contain logic that handles UI and operating system interactions.
  - Keep in mind that you don't own implementations of Activity and Fragment; rather, these are just glue classes that represent the contract between the Android OS and your app.
  - The OS can destroy them at any time based on user interactions or because of system conditions like low memory.

Common architectural principles

- **Drive UI from a model**
  - Another important principle is that you should drive your UI from a model, preferably a persistent model.
  - Models are components that are responsible for handling the data for an app.
  - They're independent from the View objects and app components in your app, so they're unaffected by the app's lifecycle and the associated concerns.
  - Persistence is ideal for the following reasons:
    - Your users don't lose data if the Android OS destroys your app to free up resources.
    - Your app continues to work in cases when a network connection is flaky or not available.



Recommended app architecture

How it works

- Notice that each component depends only on the component one level below it.
- For example, activities and fragments depend only on a view model.
- The repository is the only class that depends on multiple other classes; in this example, the repository depends on a persistent data model and a remote backend data source.
- This design creates a consistent and pleasant user experience.
- Regardless of whether the user comes back to the app several minutes after they've last closed it or several days later, they instantly see a user's information that the app persists locally.
- If this data is stale, the app's repository module starts updating the data in the background.

ViewModel

- The ViewModel class is designed to store and manage UI-related data in a lifecycle conscious way.
- The ViewModel class allows data to survive configuration changes such as screen rotations.
- Architecture Components provides ViewModel helper class for the UI controller that is responsible for preparing data for the UI.
- ViewModel objects are automatically retained during configuration changes so that data they hold is immediately available to the next activity or fragment instance.
- For example, if you need to display a list of users in your app, make sure to assign responsibility to acquire and keep the list of users to a ViewModel, instead of an activity or fragment.

LiveData

- LiveData is an observable data holder class.
- LiveData is lifecycle-aware, meaning it respects the lifecycle of other app components.
- LiveData considers an observer, which is represented by the Observer class, to be in an active state it its lifecycle is in the STARTED or RESUMED state.
- Inactive observers registered to watch LiveData objects aren't notified about changes.

Advantages of LiveData

- Ensures your UI matches your data state
- No memory leaks
- No crashes due to stopped activities
- No more manual lifecycle handling
- Always up to date data
- Proper configuration changes
- Sharing resources

Observer Pattern

- The Observer Pattern defines a one-to-many dependencies between objects so that one object changes state, all its dependents are notified and updated automatically.
- If this is not simple enough then let's think about a real-world example :
  - You have subscribed to a website(Subject) and it notifies you(Observer) via email about a new post that is published on their website.

Networking with Observables

RxJava and event streams

- Previously, you saw how to use RxJava and RxBindings utility to handle events originating in the UI.

RxJava and networking

- RxJava and callback methods.

- Libraries for network requests on Android - Retrofit

```
dependencies {
    ...
    compile 'com.squareup.retrofit2:retrofit:2.0.0-beta4'
    ...
}
```

Subscribers

```
numberOfOrangesObservable
   .subscribe(
      numberOfOranges ->
        Log.d("Number of oranges: " + numberOfOranges)
   );
```

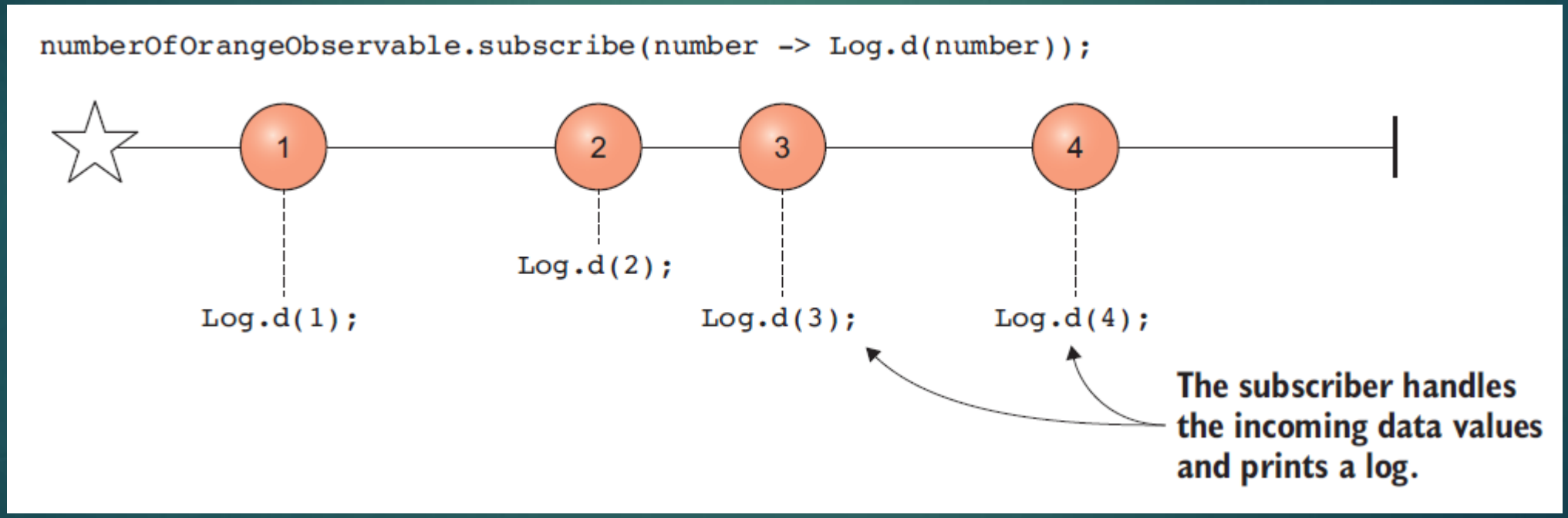
The observable from which you want to get updates.

This is the subscriber function. It'll be called on each value the observable emits.

Subscribers are also sometimes called *observers*, though we don't use the term because it's too easy to confuse with the word *observable*.

Subscribers and marbles

- You can create a marble diagram in which each marble represents the number of oranges that are in a basket at a given moment.



Decoupling of data sources

- The subscribers itself doesn't know where values come from; it's only a simple function.
- The data/values could come from the UI, network, or even a triggered timer.
- The subscriber is concerned only with **what to do with the value once it arrives**.

RxJava 2 observable types

- Observable
  - The most used observable.
  - Can emit any number of values and then complete or emit an error.
- Single
  - When you expect only one value to be emitted.
  - Single either emits a value and completes or emits an error.
  - Single can't complete without emitting a value.
    - Network responses
    - Results from complex calculations
    - toList operator (converts observable into a single list)

RxJava 2 observable types

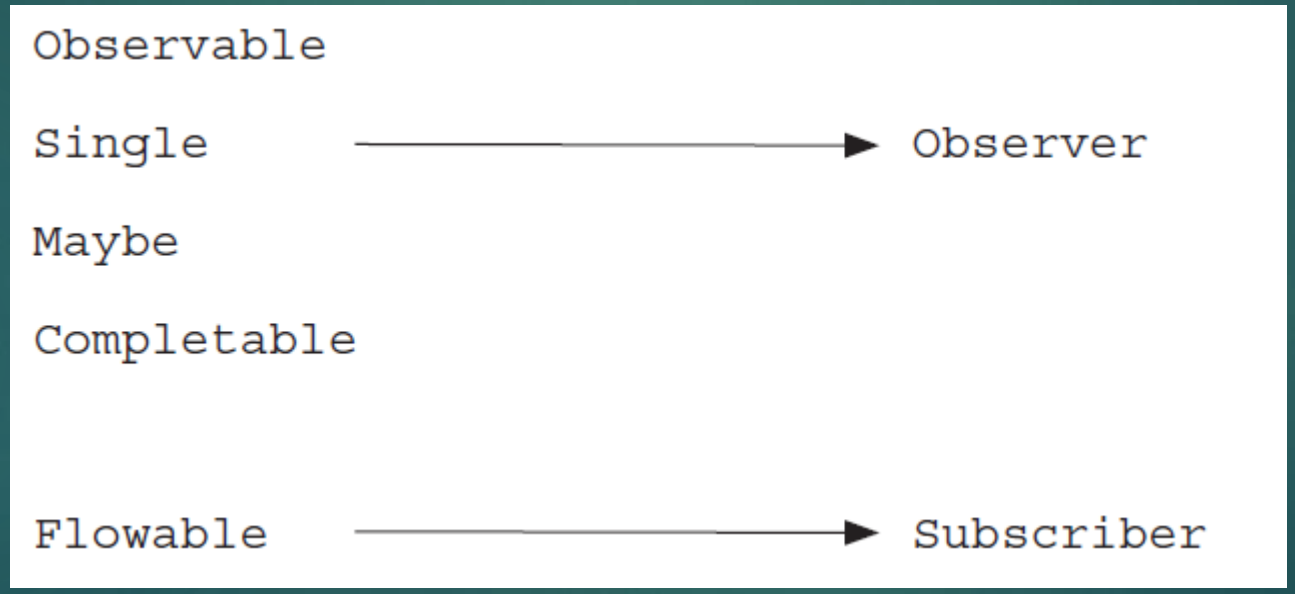
- Maybe
  - Similar to Single, but no guarantee of getting that single value.
  - Either emit an item and complete, or just complete.
  - Can also emit an error.
- Completable
  - Doesn't emit anything, but just completes or not.
  - May emit an error.
  - This is basically an "event" that indicates something happened or finished.
  - Could be used to indicate state change, such as when a fragment is destroyed.

RxJava 2 observable types

- Flowable
  - In later versions of RxJava 1, a concept called back pressure was introduced to manage situations in which an observable produces too many items for the subscriber to handle.
  - In RxJava 2 the special back-pressure techniques were moved into a Flowable type.
  - In Flowable, you're forced to define what happens if the source produces too many items.
  - Overkill and seldom used.

Subscribing to and converting different observables

- In terms of the terminology used, however, we talk about only observables and subscribers, regardless of the specific class.



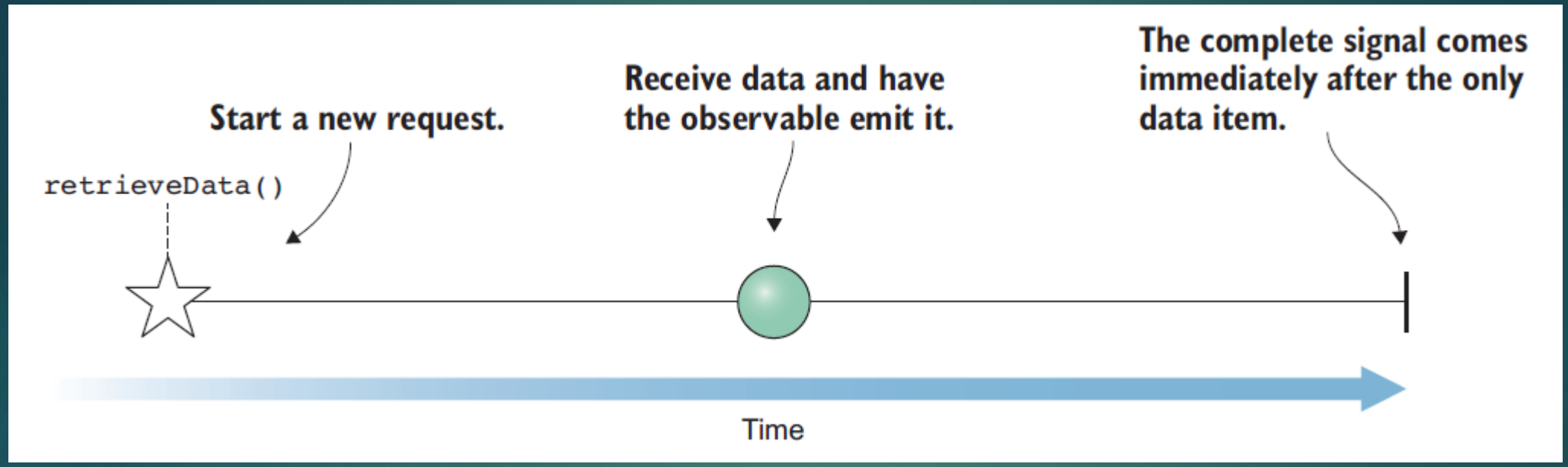
What happens when you make a normal network request?

1. Initiate the network request.
   - Call retrieveData() to start the request.
2. Pass a callback to the function.
   - Define a callback function (point of re-entry).
3. Wait for a response.
   - No need to wait, just continue with other codes.
4. Receive data in the callback and do stuff.
   - Callback will be triggered with the fresh data from the network.
   - Display results in UI.

What happens when you make a network request with an observable?

1. Create a network observable.
   - Define retrieveData() to return an observable instead.
2. Subscribe to the network observable.
3. Wait for a response.
4. Receive data in the subscriber and do stuff.
   - When everything is ready, the callback will be triggered with the fresh data from the network.
   - Display results.

Network request as an observable

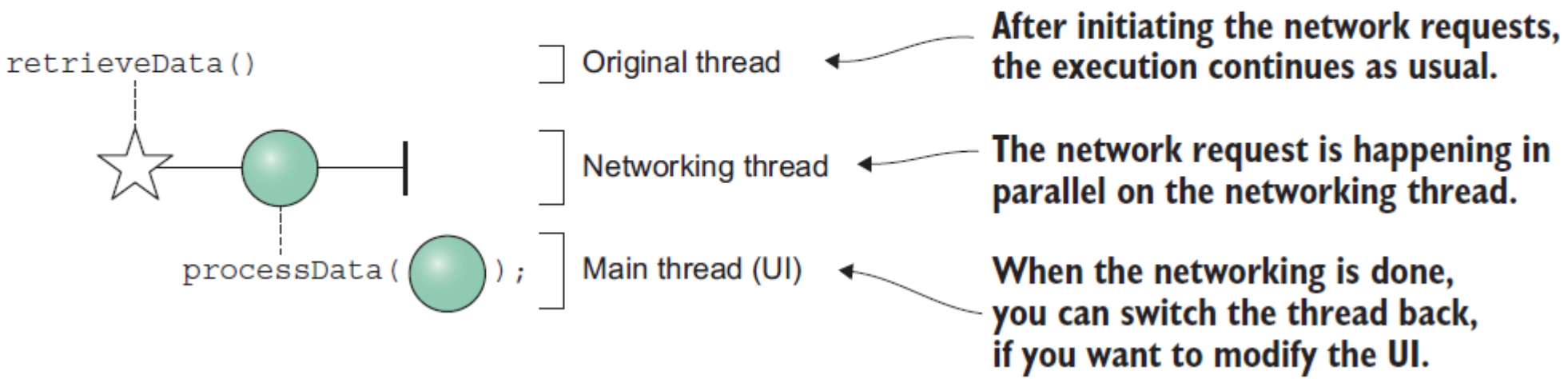


- The function you'll use for processing the data is triggered at the point when the data arrives (the marble).
- The subscriber function is executed then and there.

Network request as an observable

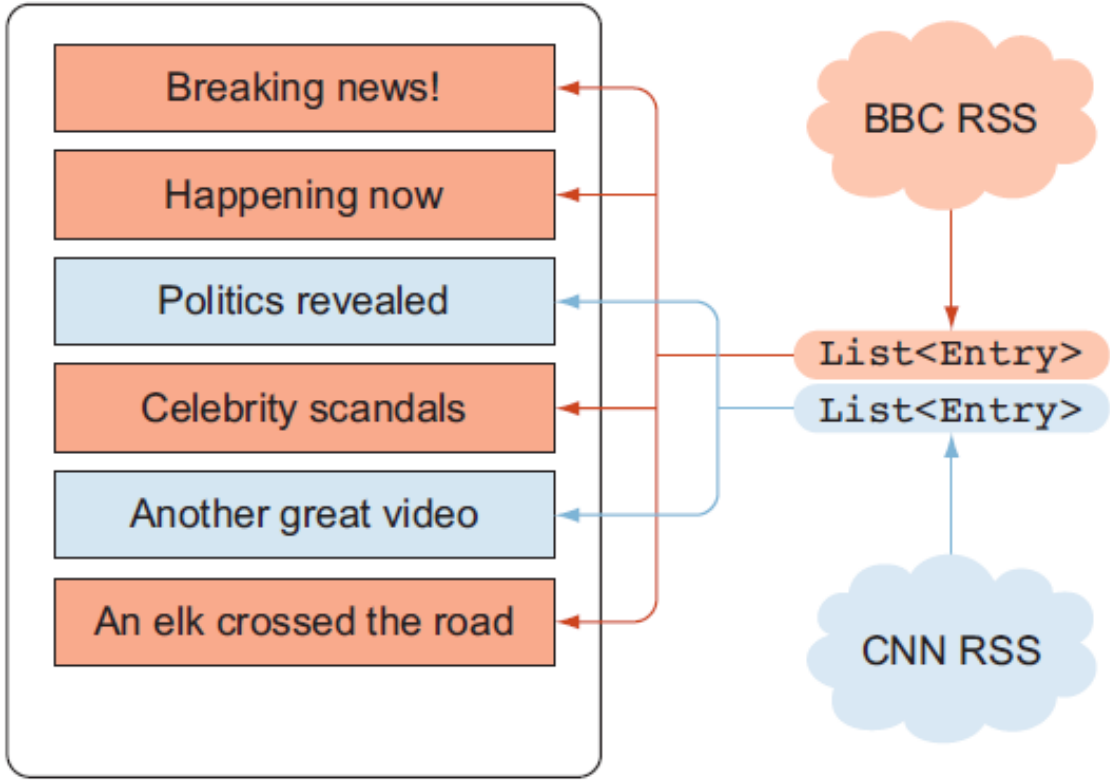
- Use the observeOn operator to switch all following operations and subscribers onto the thread specified

```
retrieveData()
   .observeOn(AndroidSchedulers.mainThread())
   .subscribe(this::processData);
```



Example: An RSS feed aggregator

- To make things more concrete, you'll make a little app that can load multiple RSS feeds and show them in one list, ordered by date.
- You have two API endpoints but only one list of mixed content. You need to make two calls and combine the results.



The feed structure

The parser produces a list of items of type `Entry`, which it defines itself. It looks like this:

```
public static class Entry {
    public final String id;
    public final String title;
    public final String link;
    public final long updated;

   Entry(String id,
       String title,
       String link,
       long updated) {
        this.id = id;
        this.title = title;
        this.link = link;
        this.updated = updated;
    }

    public String toString () {
        return new Date(updated).toString() + "\n" + title;
    }
}
```

The title of the feed entry. We will show this on the list.

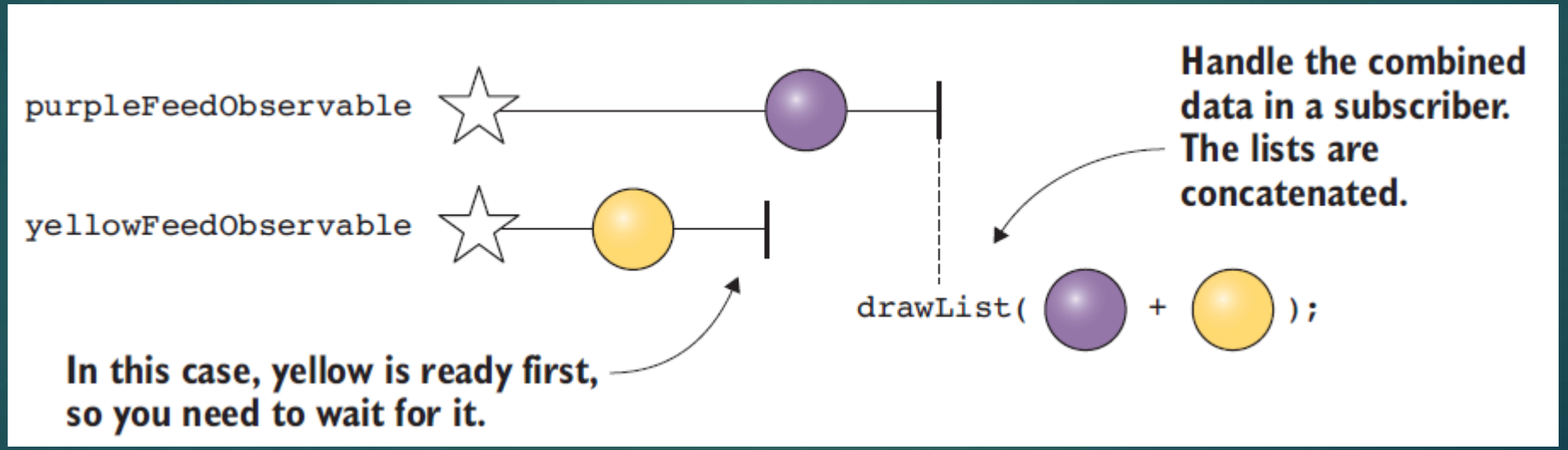
Link to the actual article that the feed contains. This is a URL.

Timestamp that you'll use for sorting.

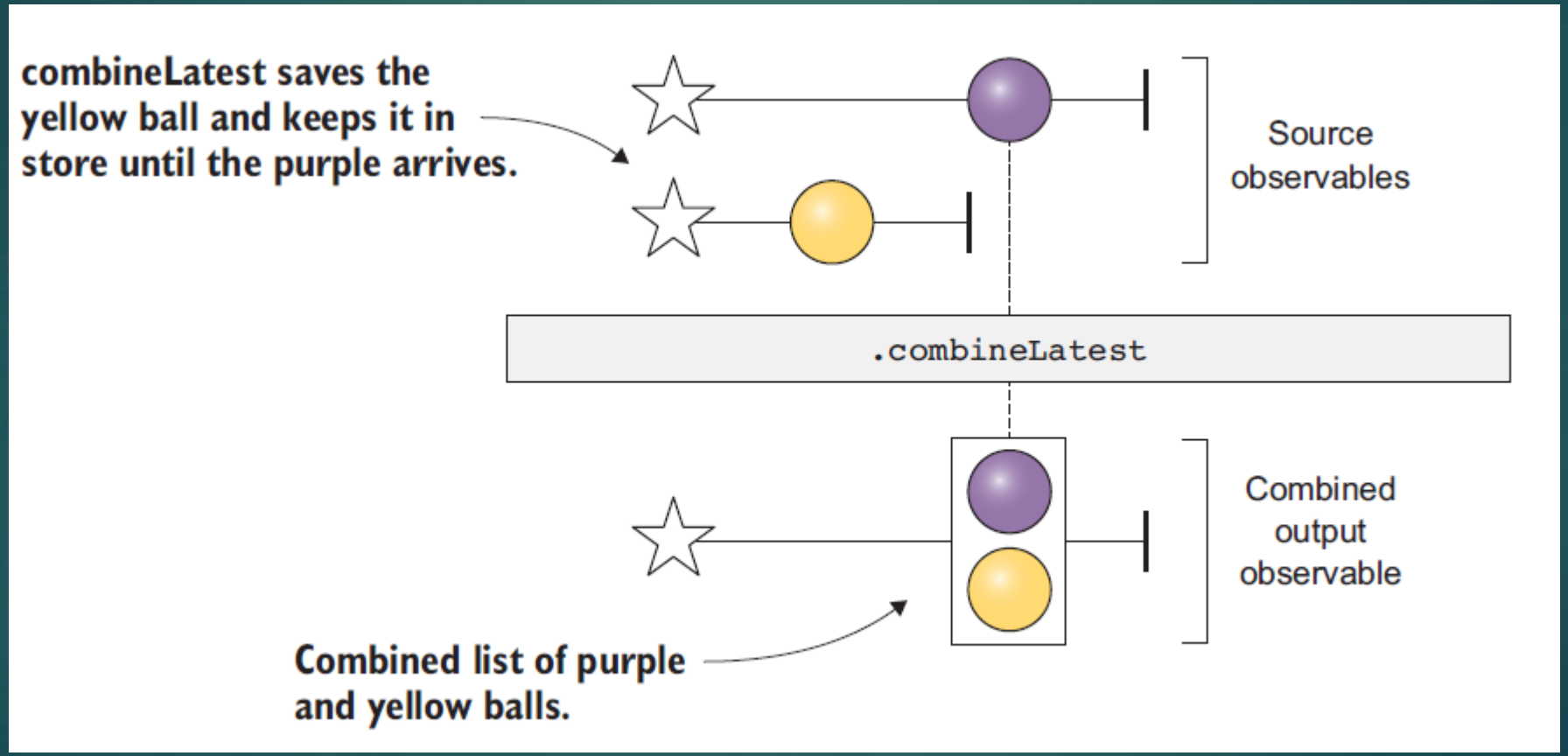
For convenience, you define a toString method. It'll be used on the lists at first.

Getting the data

1. Start requests for both feeds.
2. Wait until the requests are completed.
3. Call drawList with the combined results.



The combineLatest operator



The Rx code so far

```
public class MainActivity extends Activity {
    private static final String TAG =
      MainActivity.class.getSimpleName();

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Observable<List<Entry>> purpleFeedObservable =
                FeedObservable.getFeed(
                        "https://news.google.com/?output=atom");

        Observable<List<Entry>> yellowFeedObservable =
                FeedObservable.getFeed(
                        "http://www.theregister.co.uk/software/
                           headlines.atom");

        Observable<List<Entry>> combinedObservable =
                Observable.combineLatest(
                        purpleFeedObservable, yellowFeedObservable,
                        (purpleList, yellowList) -> {
                            final List<Entry> list = new ArrayList<>();
                            list.addAll(purpleList);
                            list.addAll(yellowList);
                            return list;
                        }
                );

        combinedObservable
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(this::drawList);
    }

    private void drawList(List<Entry> listItems) {
        final ListView list = (ListView) findViewById(R.id.list);
        final ArrayAdapter<Entry> itemsAdapter =
                new ArrayAdapter<>(this,
                 android.R.layout.simple_list_item_1,
                   listItems);
        list.setAdapter(itemsAdapter);
    }
}
```

The beginning of the Rx logic you already saw, but here it's in the Activity context.

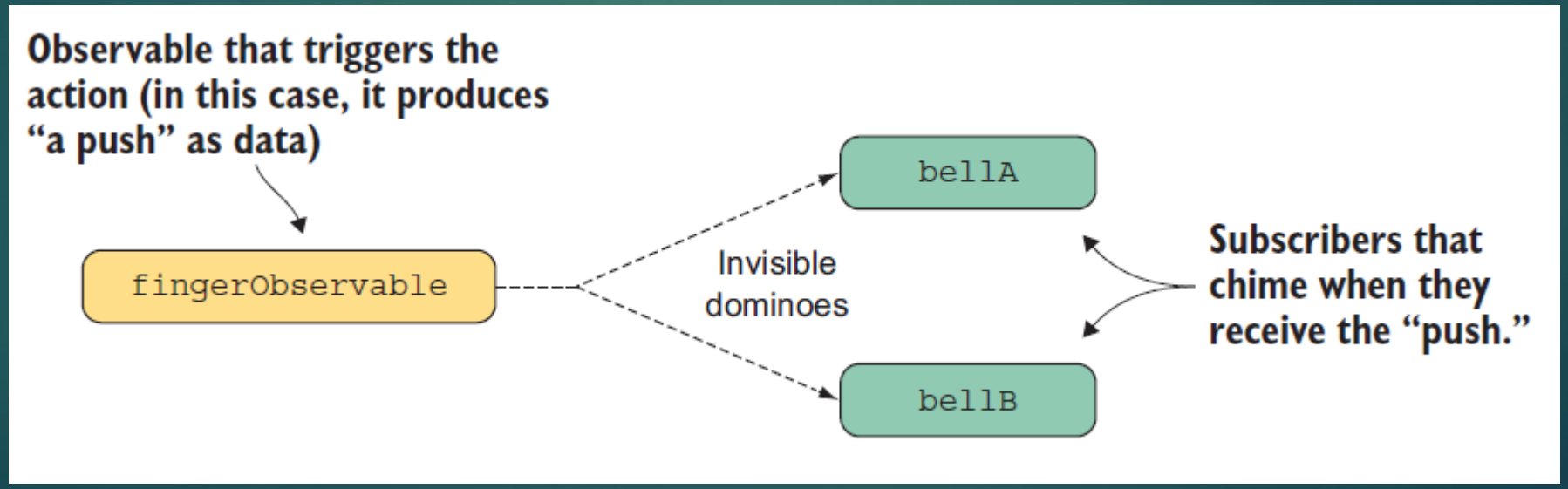
In this version, the feeds are retrieved only once and there's no update. You might add one later!

Switch the thread to the main thread and pass the aggregated list to the drawing function.

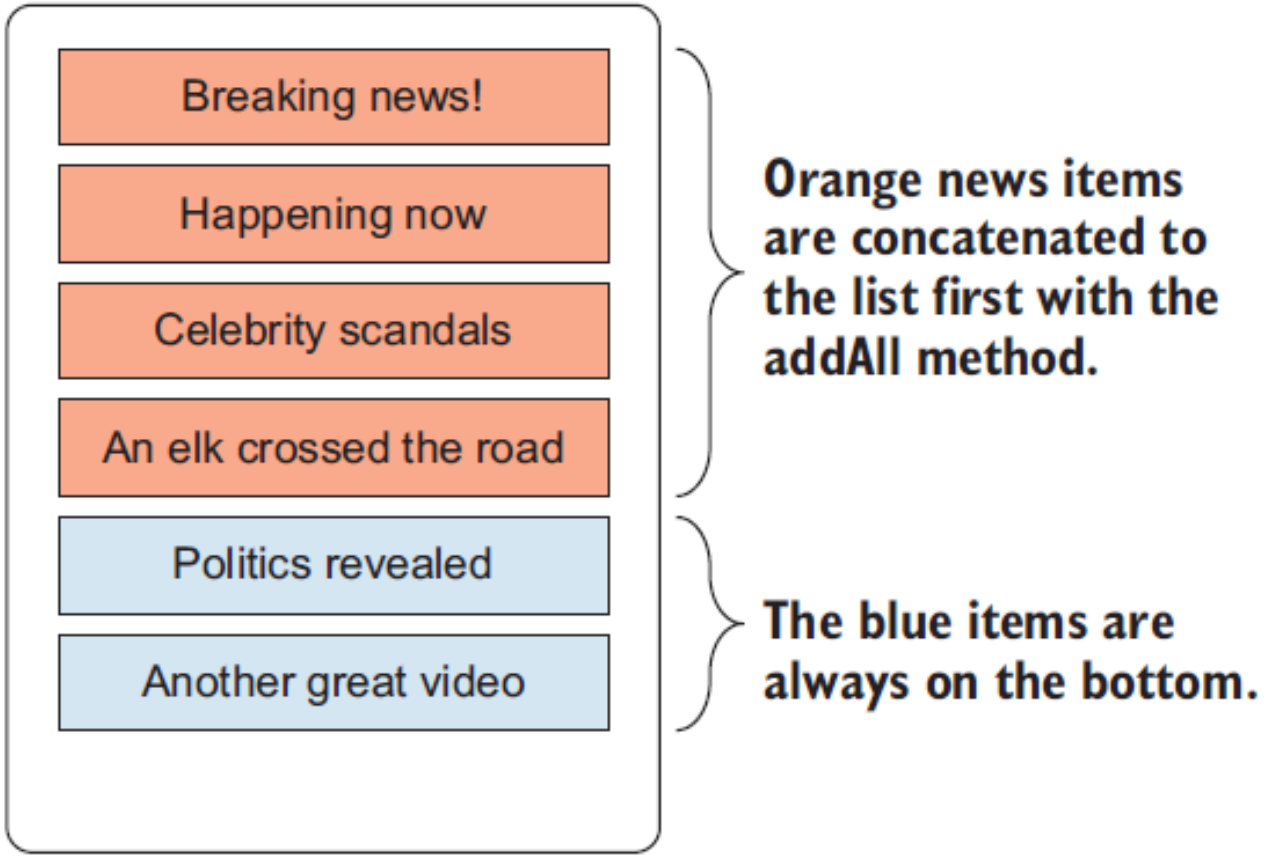
Create a new ArrayAdapter and populate the list with your retrieved items.

Asynchronous data processing chains

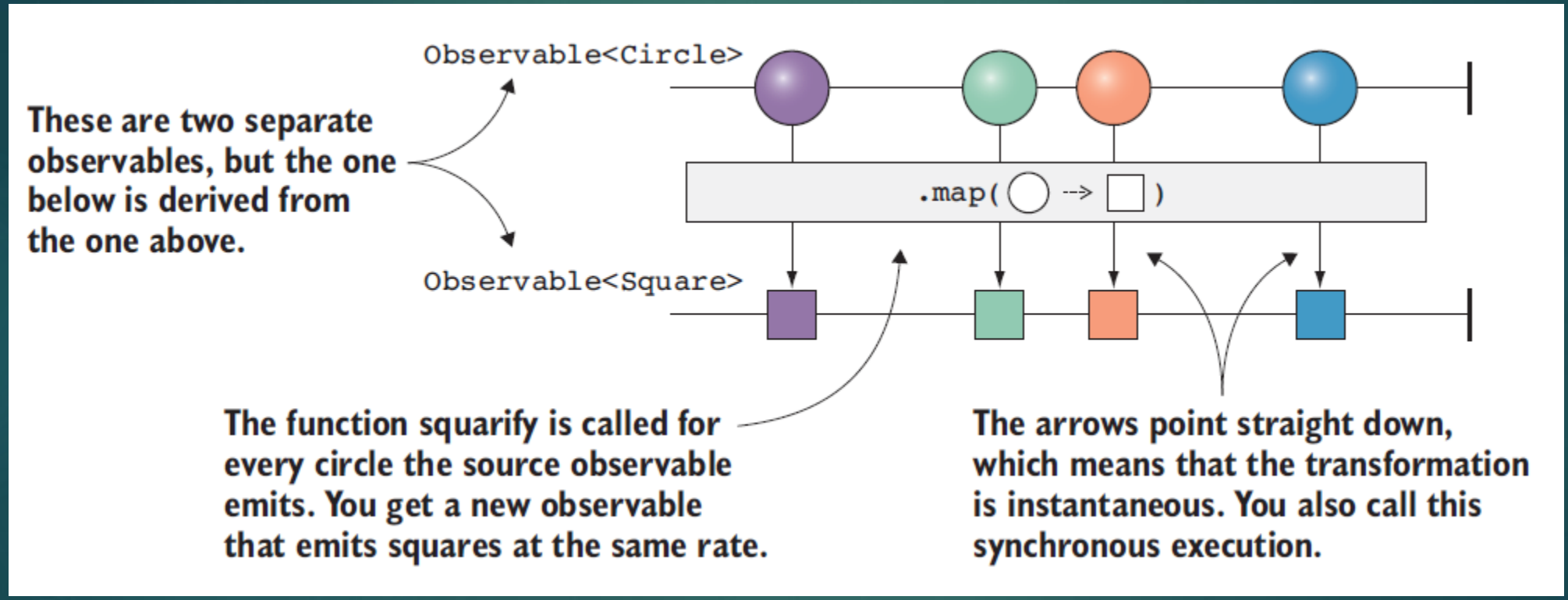
- A processing chain is a bit like a string of dominoes where, at each step, the incoming piece of data triggers the next operation.
- This scenario can be expressed with one event observable that gives its output to two subscribers.

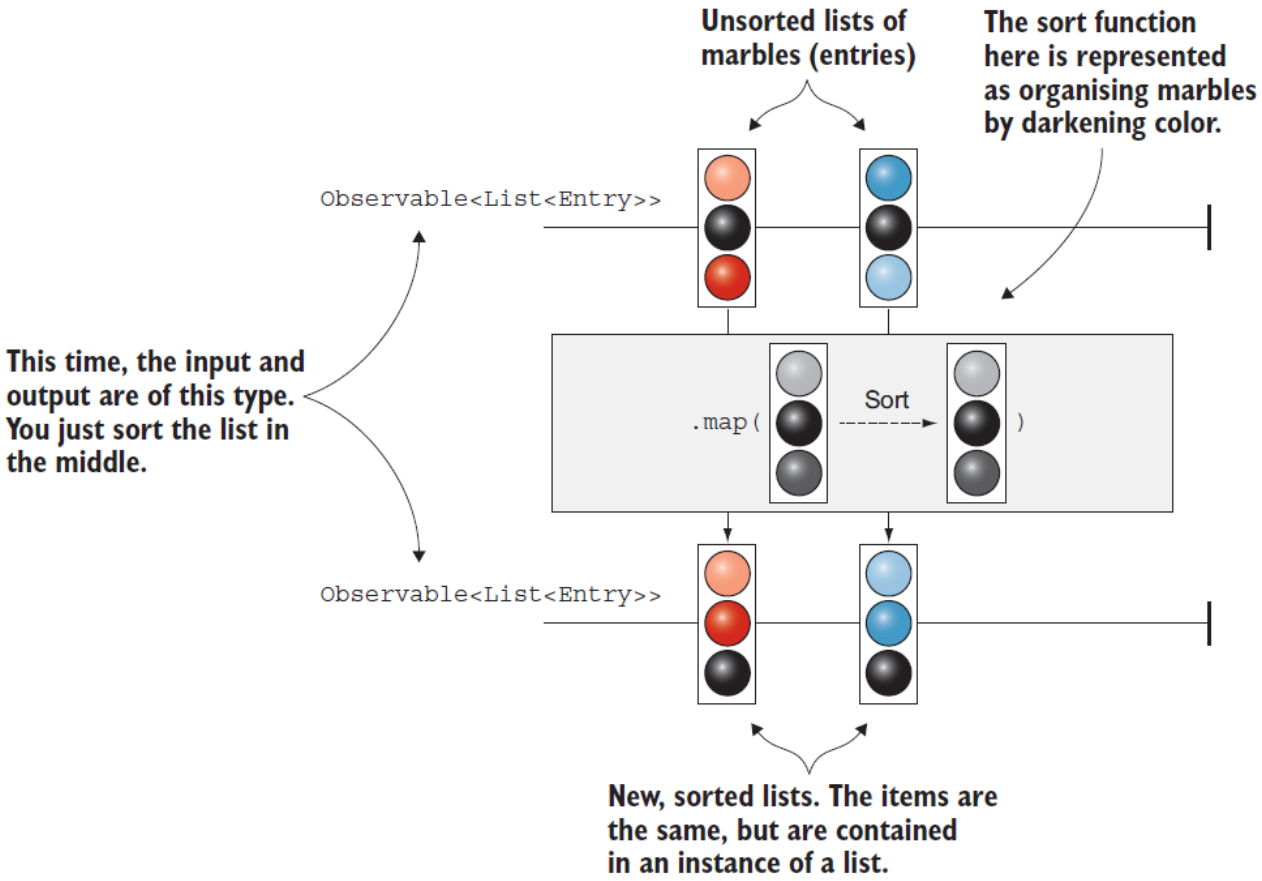


Putting the list in order

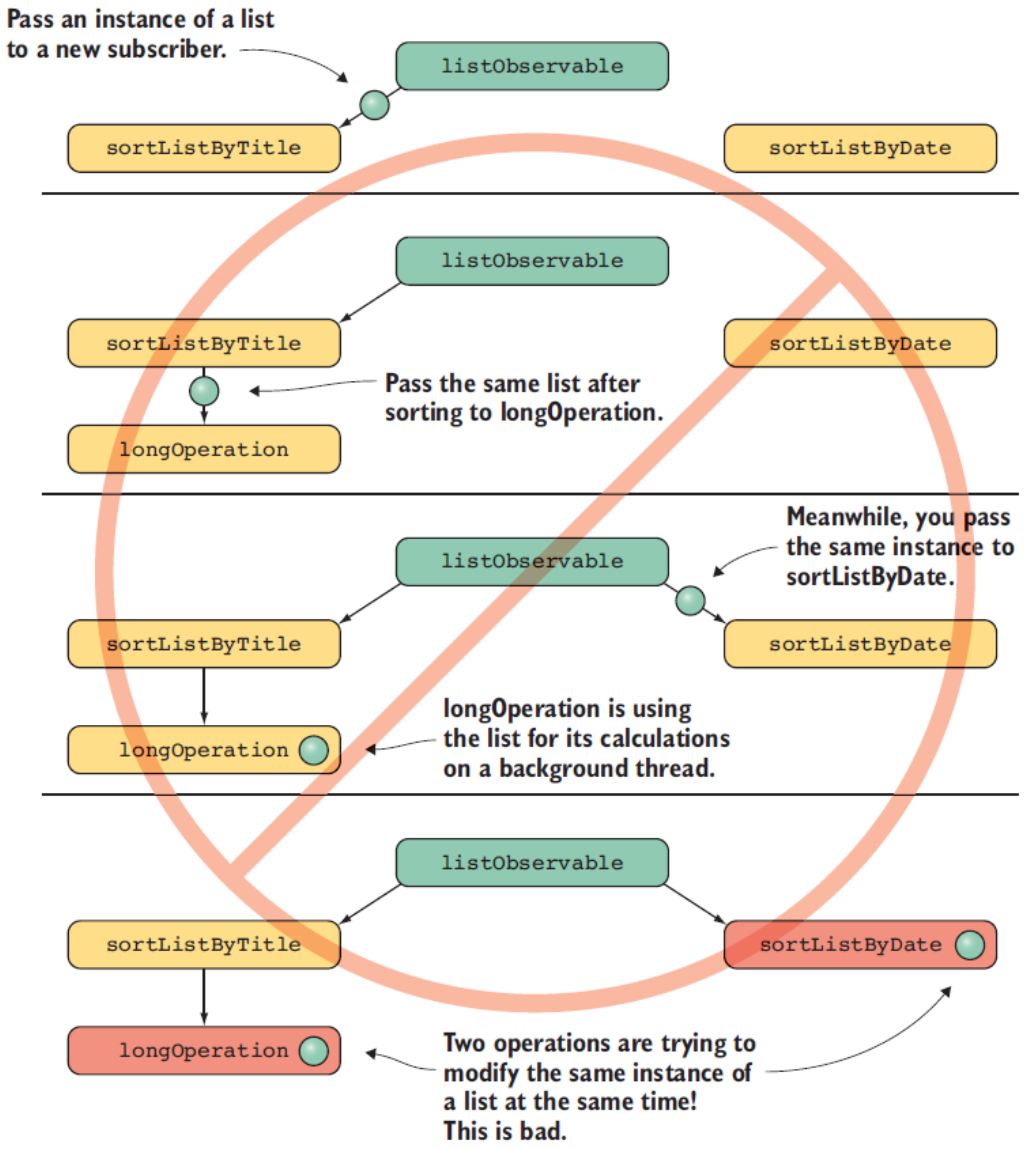


The map operator

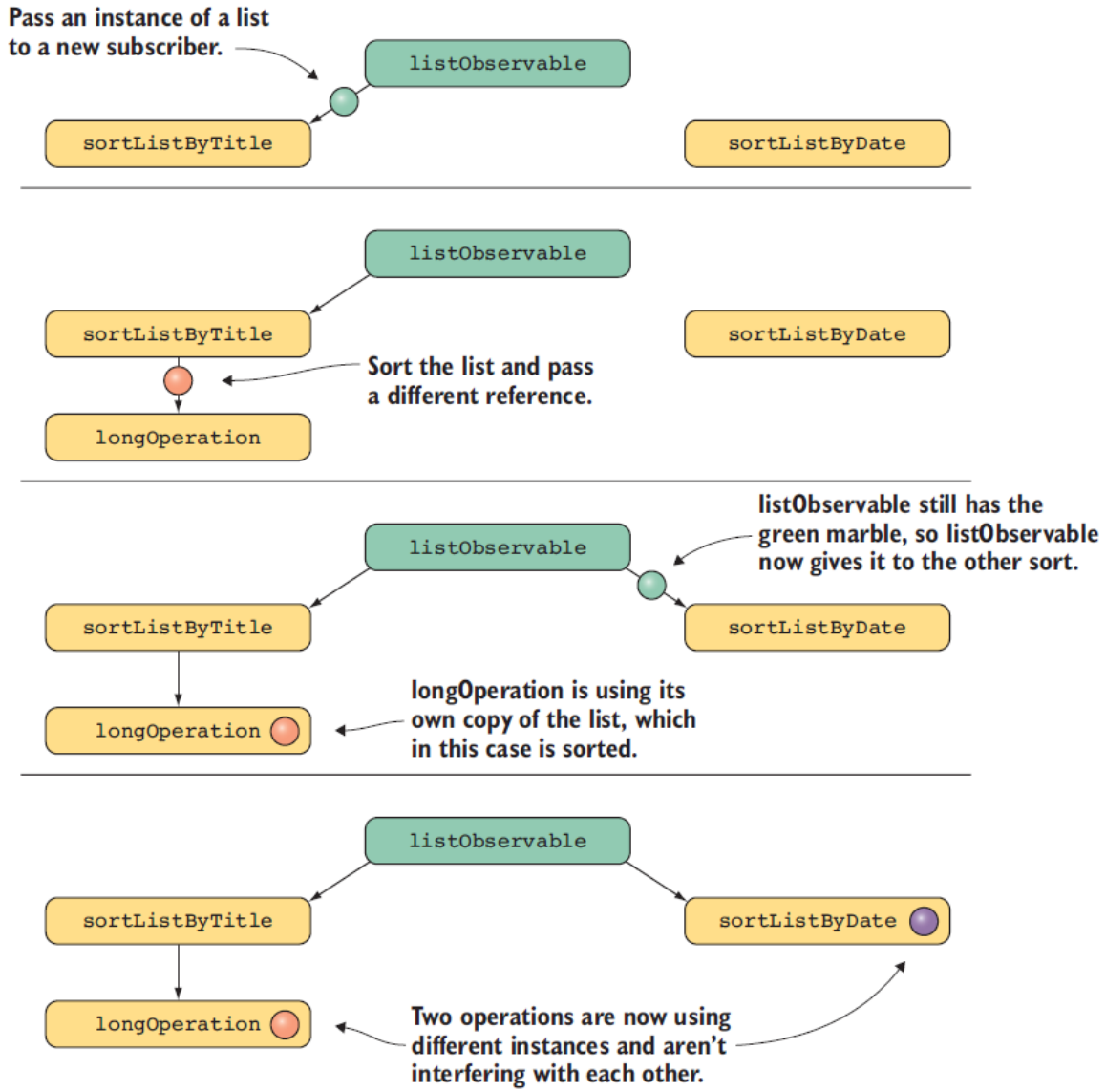




Using map to sort a list



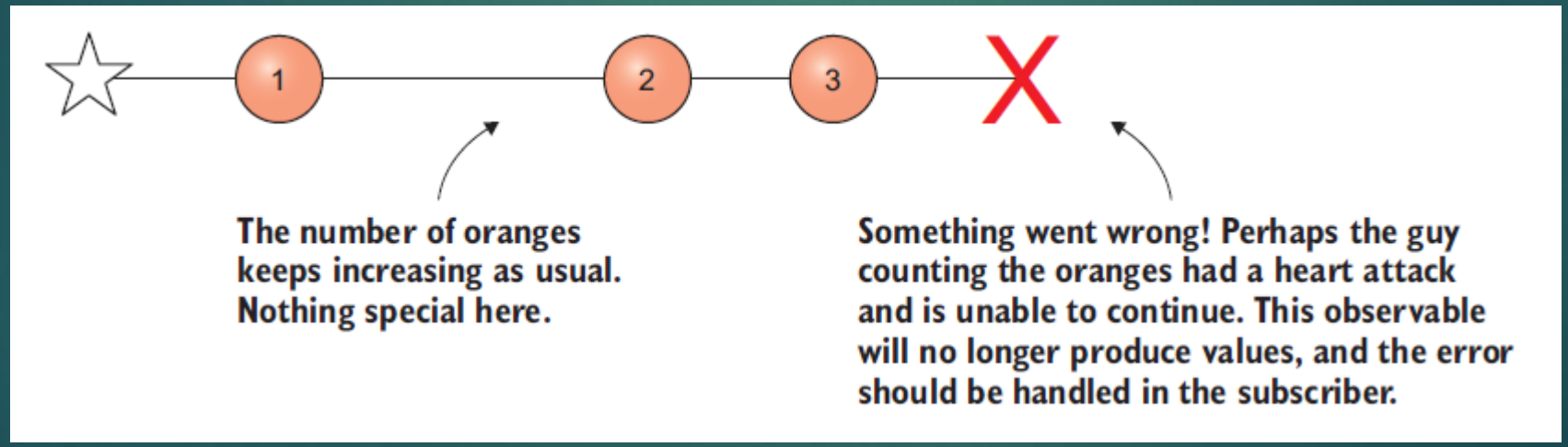
Chain without immutability



Chain with immutable data

Error handling

- An error in RxJava won't stop the execution of the program itself but will produce a notification that's of type error.
- You can then deal with the error similarly to the way you would with normal values.



Network errors and handling them

```
combinedObservable
    .subscribe(
        pair -> drawList(pair.first, pair.second),
        error -> Log.e("Error occurred", error)
    );
```

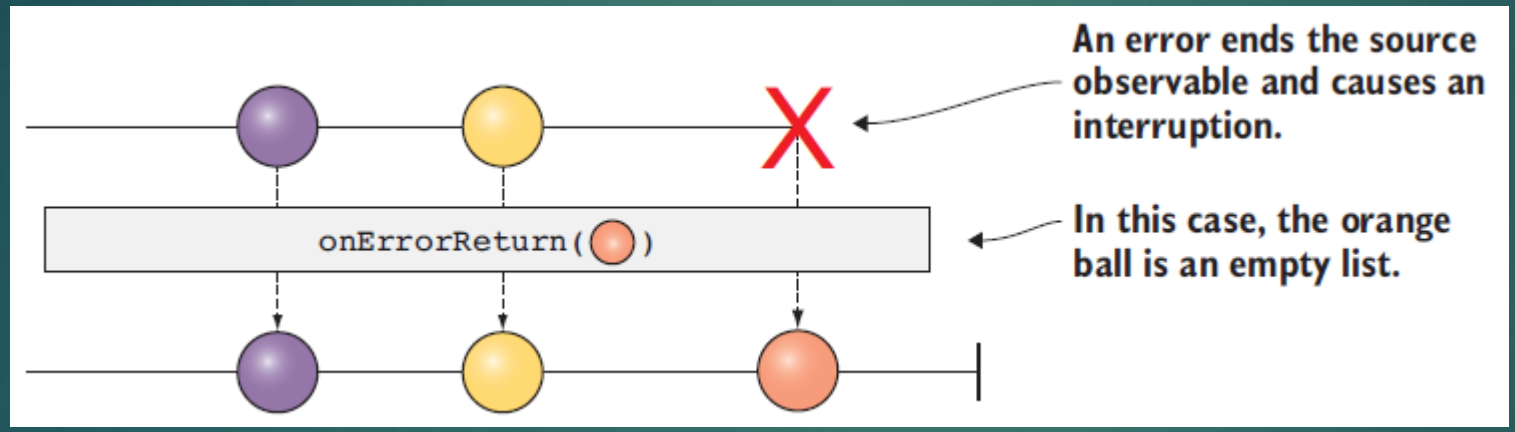
The second parameter of subscribe is a function that's called when an error is emitted

```
Observable<List<Entry>> combinedObservable =
  Observable.combineLatest(
    purpleFeedObservable.retry(3),
    yellowFeedObservable.retry(3),
    ...
  );
```

You define both of these operations to be retried three times before letting the error fall though.

What to do when a real error comes?

- RxJava provides a simple way to accomplish that: you can declare a policy for returning an empty list in case the feed network observable emits an error.

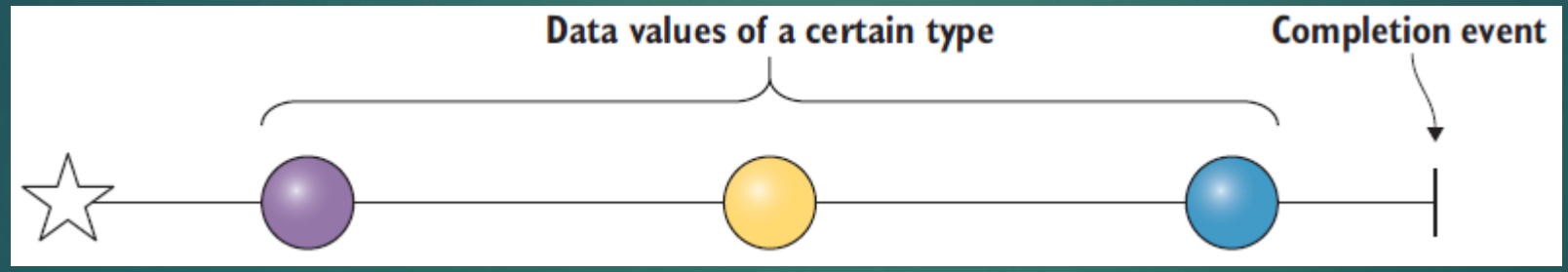



```
purpleFeedObservable
   .retry(3)
   .onErrorReturn(e -> new ArrayList<>())
```

Building Data Processing Chains

Different roles of observables

- An observable emits a value whenever it has a new one.
- It can also complete or throw an error.
- Essentially, there are two uses for the Observable class: event observables and reactive variables.



Event observables

- This is what's typical of an observable that's a plain event source:
  - Emitted events are time-based and can be filtered based on the time.
  - Events contain little or even no data.
  - The clicks observable is a good example of an event observable.

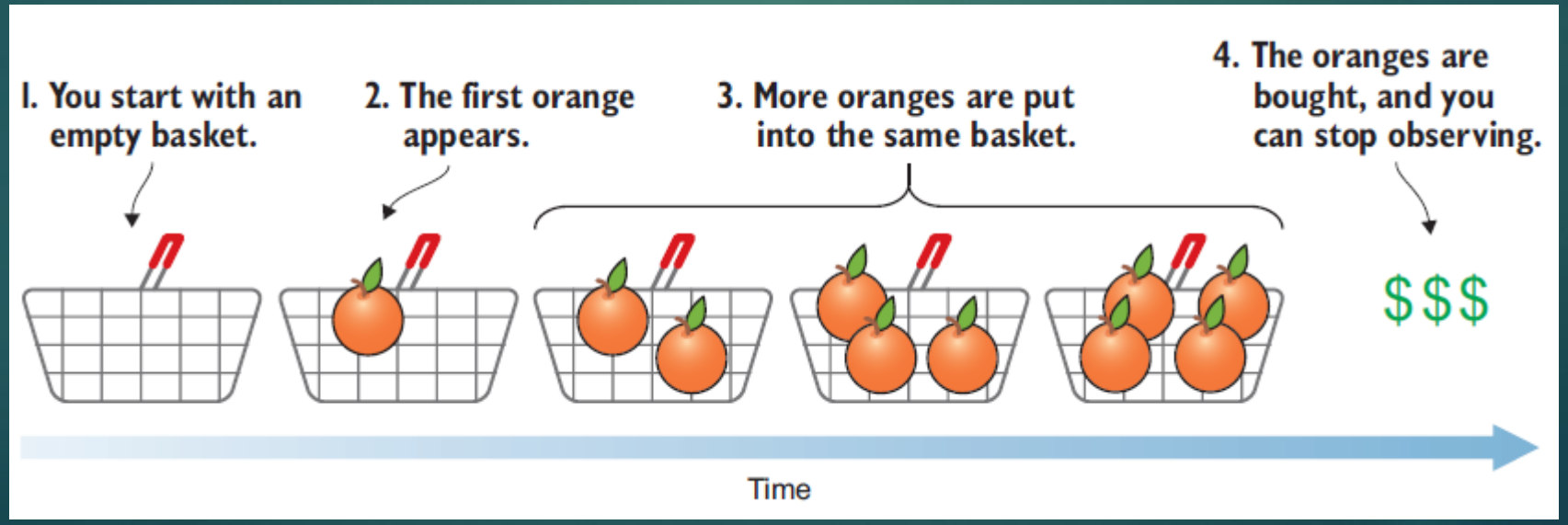
Reactive variables

- An observable can be used as a reactive variable that tells everyone whenever it changes, as follows:
  - Emits its possible previous state immediately to new subscribers
  - When _updated, always emits its full state to all subscribers

Number of oranges in a basket

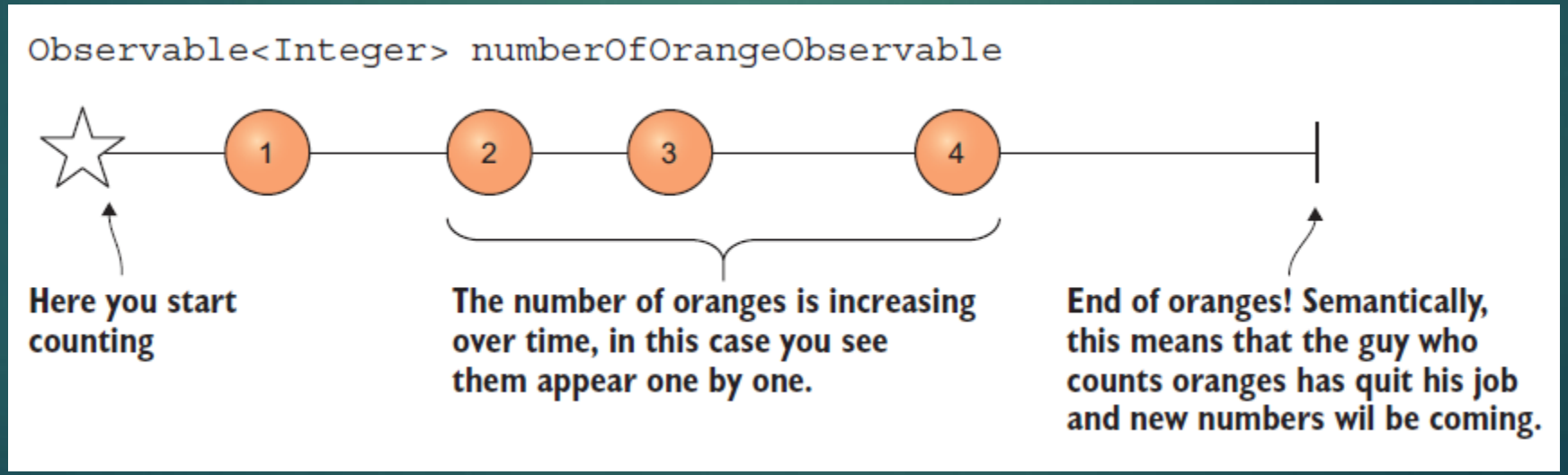
```
// An observable for how many oranges are in a basket
Observable<Integer> numberOfOrangeObservable = ...;
```

- As an example of a reactive variable, imagine a basket of oranges. You'll use an observable to keep track of changes in the number of them.
- Whenever someone puts more in or takes them out, the observable emits the new value immediately.



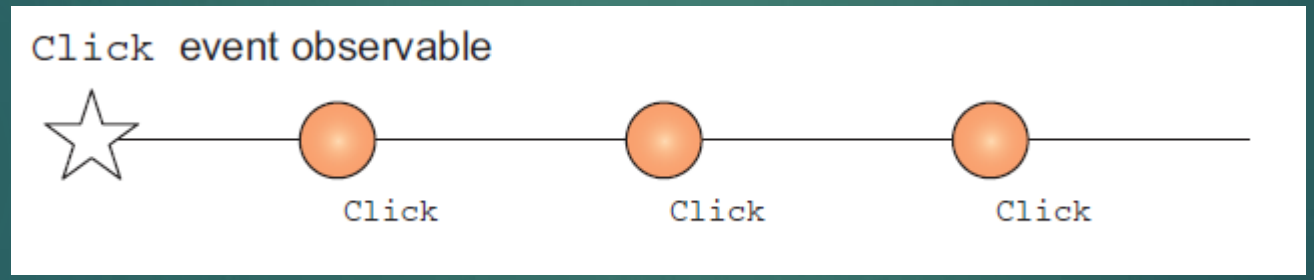
Number of oranges in a basket

- You can make this picture more concise by using an integer number to indicate the number of oranges you have at a given moment.
- In the marble diagrams that represent it, you can see a new marble every time the number of oranges changes.



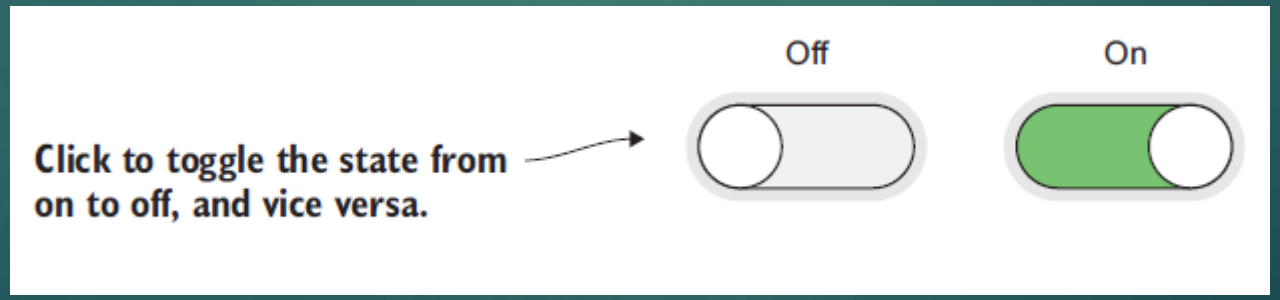
Events vs. reactive state

- Click event observable
  - Let's start with an event observable that sends clicks on a particular part of the UI.
  - Notice that the clicks are just events in time; you don't even have the pixel coordinates.



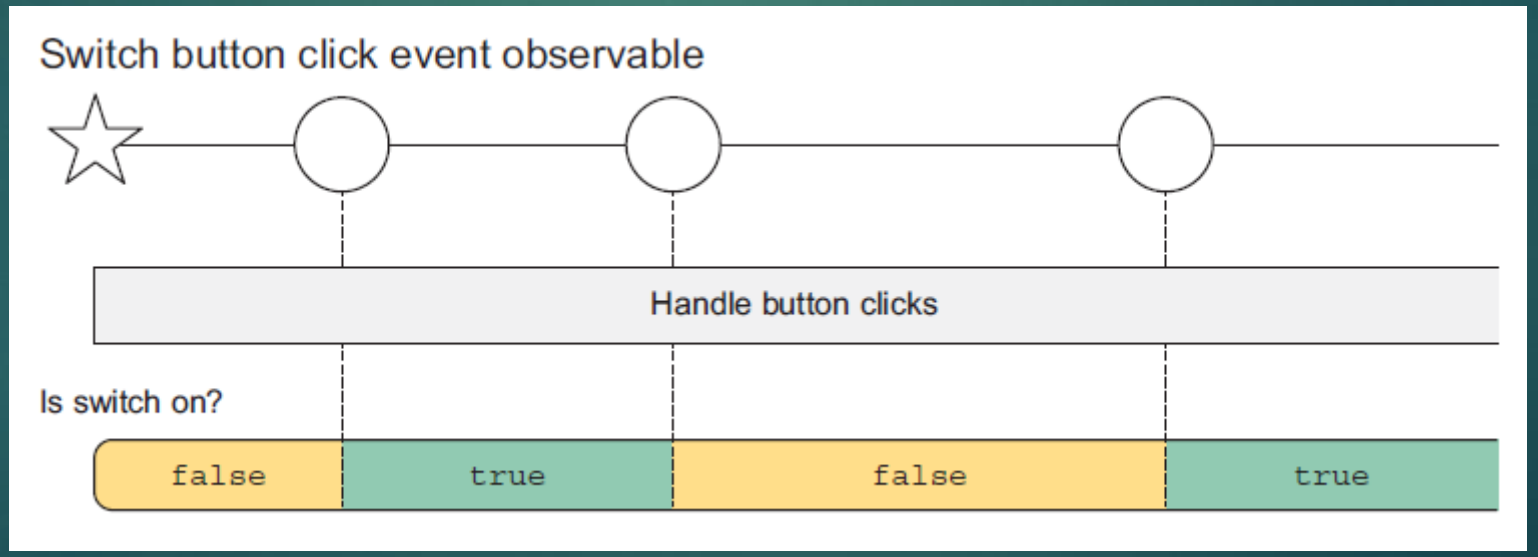
Events vs. reactive state

- The switch button
  - Events don't have much information except for the time.
  - But you can add logic that does something with the events.
  - You interpret the events into state.
  - In this case, you'll take the switch button as an example: clicking it turns it on and off.



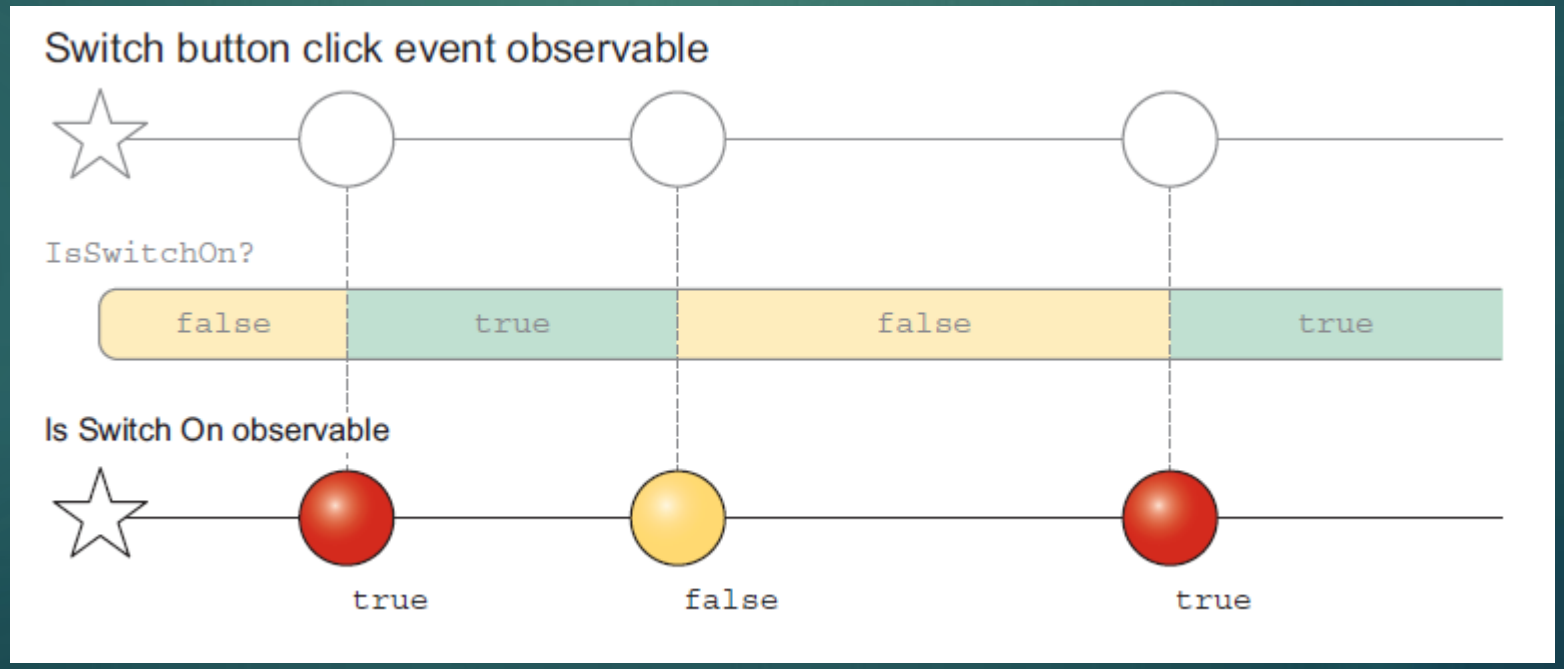
Events vs. reactive state

- Converting events into state
  - If you consider the clicks you saw before as happening on the switch button, you can build a layer that interprets the clicks and changes the button state accordingly.

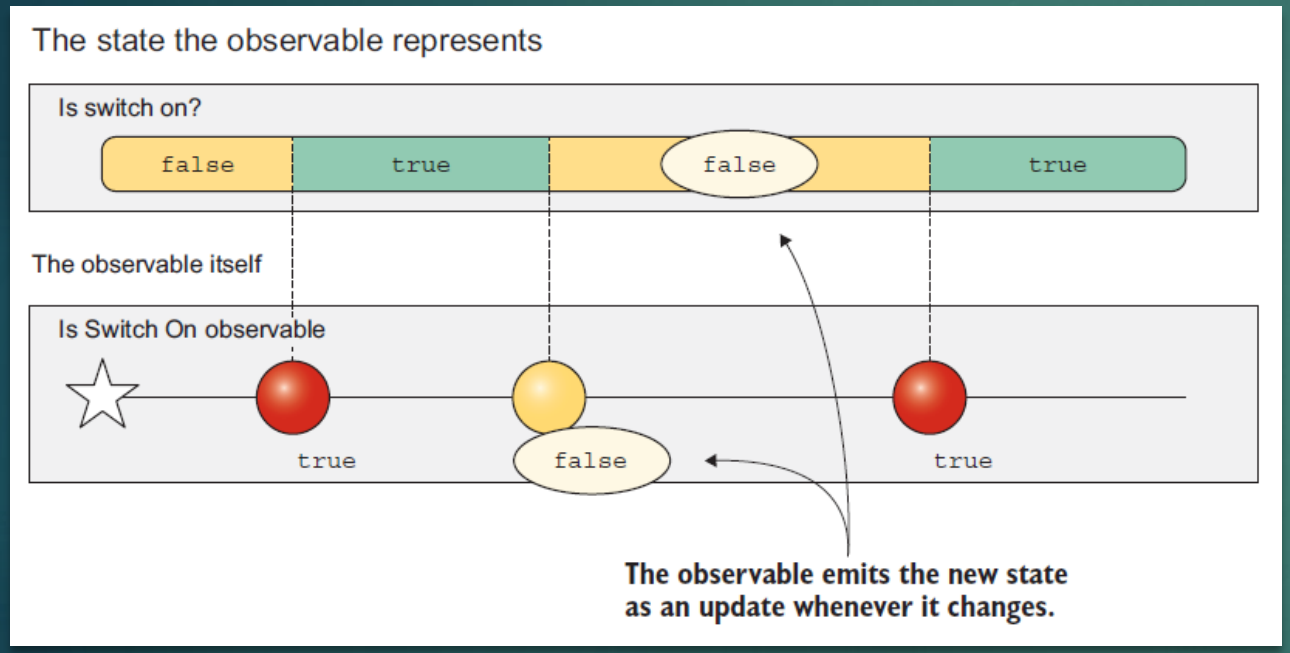


Events vs. reactive state

- Observable as a reactive variable
  - We'll now take the last step and create an observable of type Observable<Boolean> to represent the state indicating whether the switch button is turned on.



Internal state of an observable



- What you saw is a true observable: one that emits the full state whenever it changes.
- What changes is the variable the observable represents, and sometimes it's kept as an internal state of the observable.
- The biggest difference as compared to the click's observable is that this one emits the full state (a Boolean value) every time it changes.
- In case of an event, you didn't even have any data.

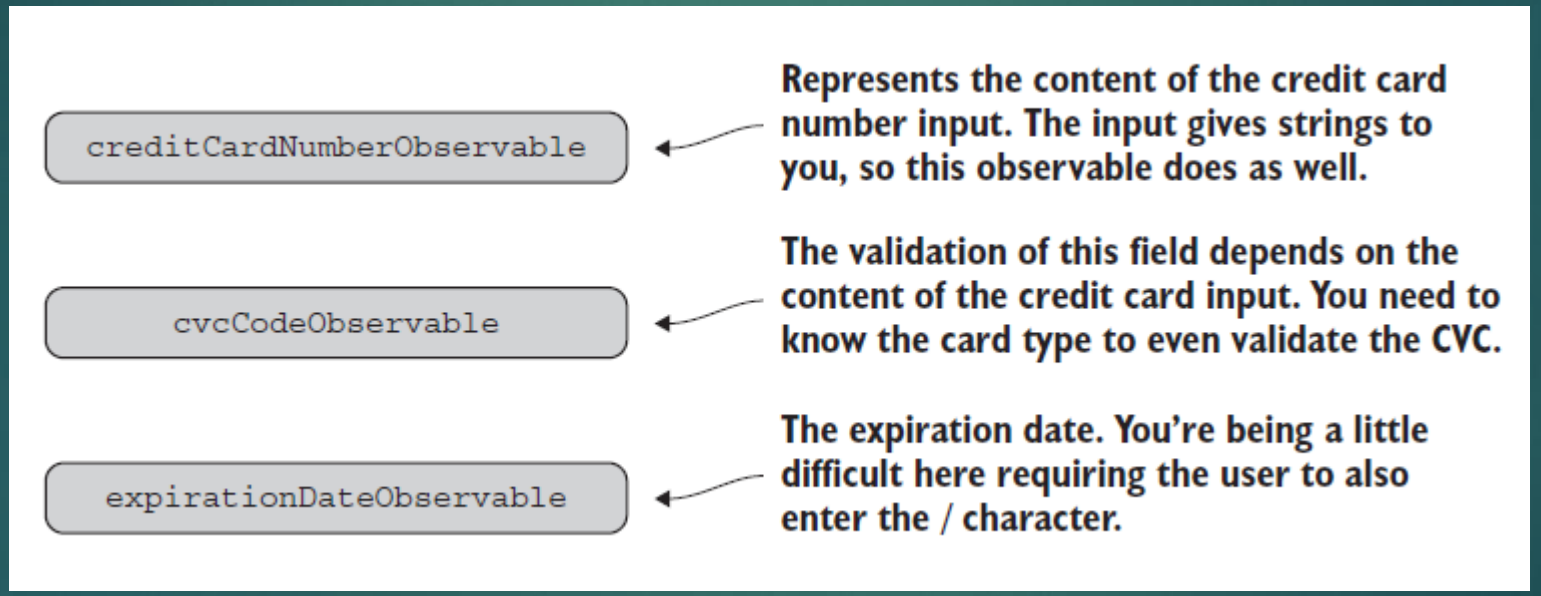
Example: Credit card validation form

- https://en.wikipedia.org/wiki/Payment_card_number

Validating the numbers in steps

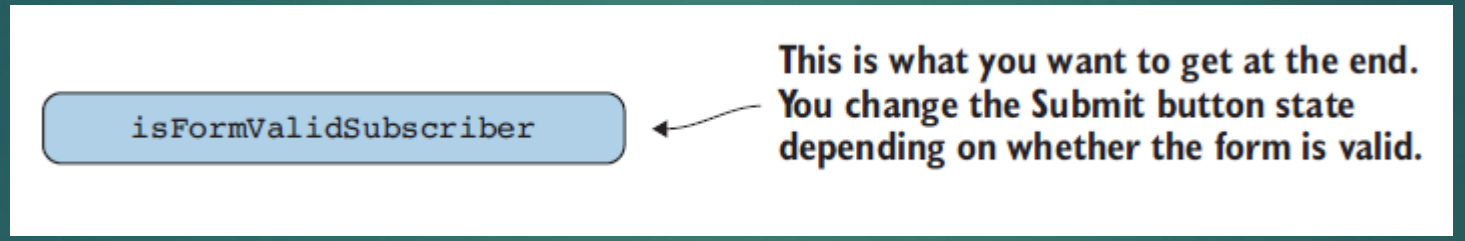
1. creditCardNumber conforms to one of the card types.
2. creditCardNumber passes the check sum function.
3. cvcCode is of the right length (depending on the card type).
4. expirationDate is properly formatted (MM/YY).

Inputs

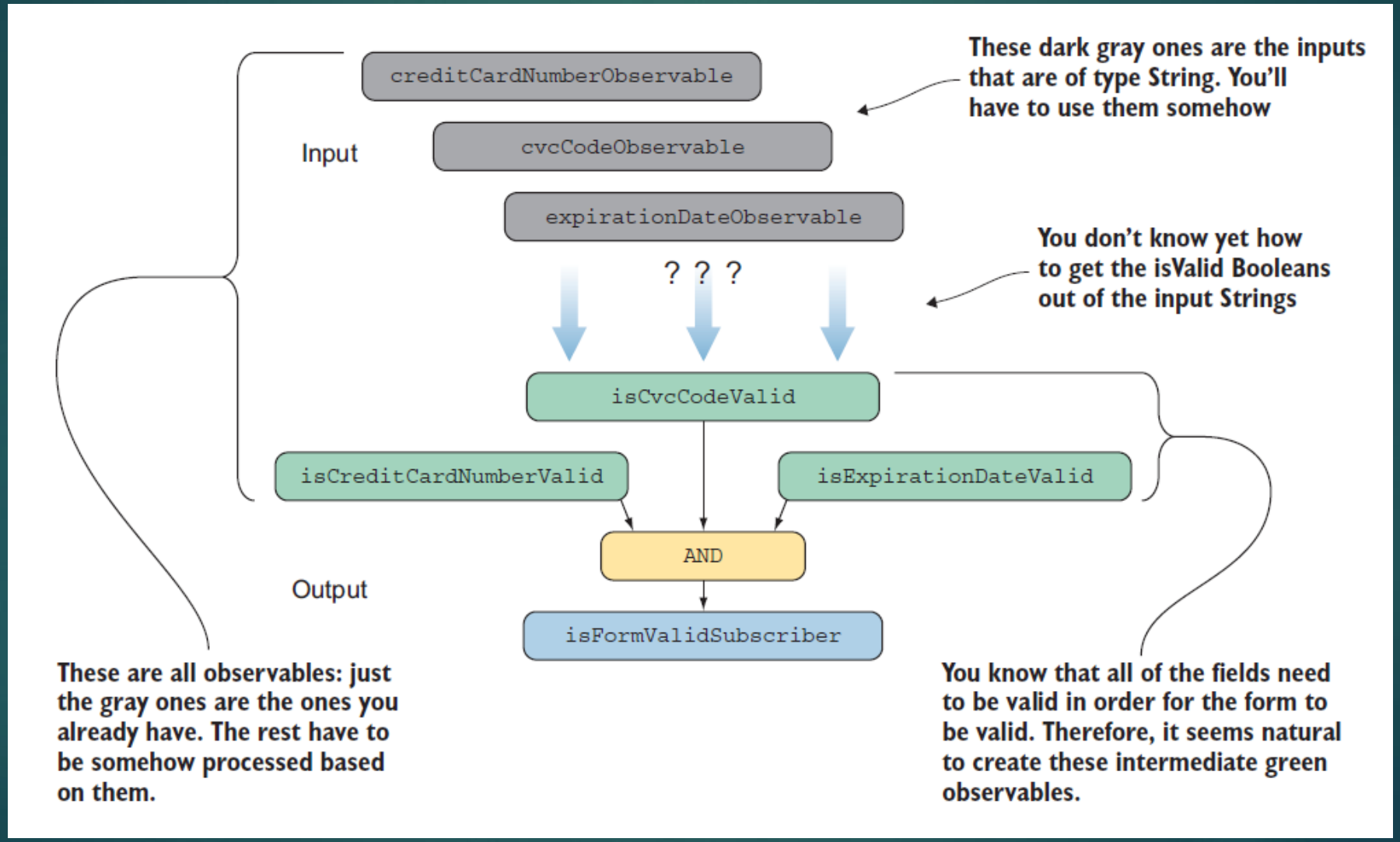

creditCardNumberObservable
Represents the content of the credit card number input. The input gives strings to you, so this observable does as well.
cvcCodeObservable
The validation of this field depends on the content of the credit card input. You need to know the card type to even validate the CVC.
expirationDateObservable
The expiration date. You're being a little difficult here requiring the user to also enter the / character.

Outputs

- Ultimately, you have only one goal - to know whether the entire form is valid and you're ready to submit.
- You define this goal as **isFormValidObservable**.



Solving the equation



Reactive View Models

The view layer

- On a practical level, you usually put the code that determines what the program looks like visually in one file, and the information to display in another file.

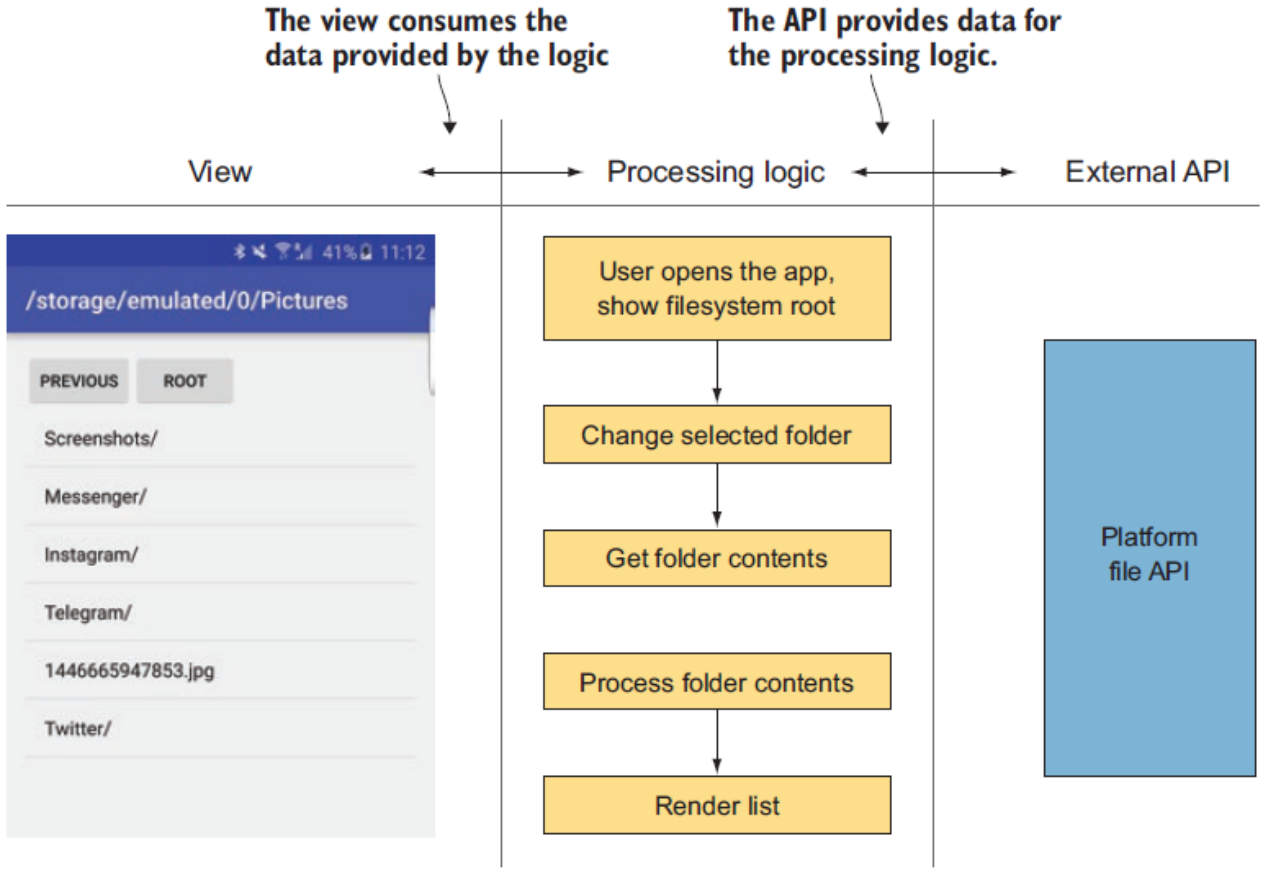
Rendering	Logic
• The size and color of a spinning loader	• The visibility of a loading spinner
• The position of a thumbnail of a list item	• The logic to get the necessary data to show on a list
• The positions of circles and crosses on a grid of tic-tac-toe	• Game rules that determine what a player is allowed to do

View

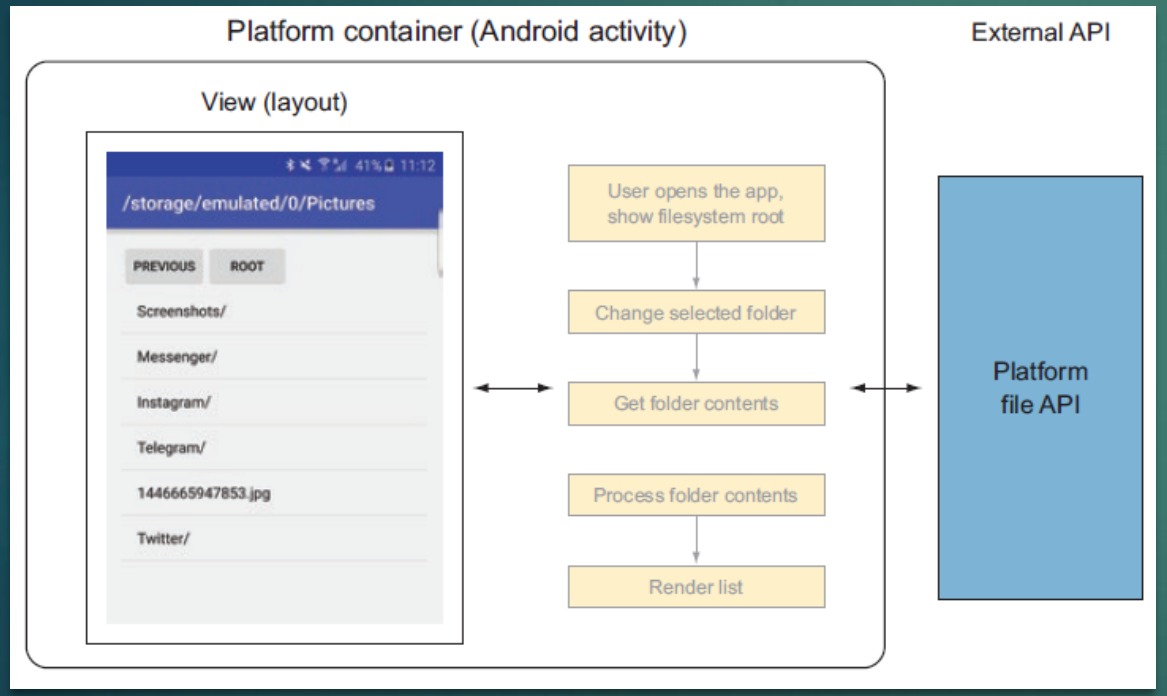
- Here are the key characteristics of a view:
  - Occupies a part of the visible screen
  - Decides how the pixels it contains are rendered
  - Represents the endpoint of data processing
  - Can contain light logic, such as drop-down states
- It's well worth mentioning that view is a generic name for a layer that represents all the components of the app that match the preceding description.

The view and the file browser

Structure of the file browser example



Platform containers



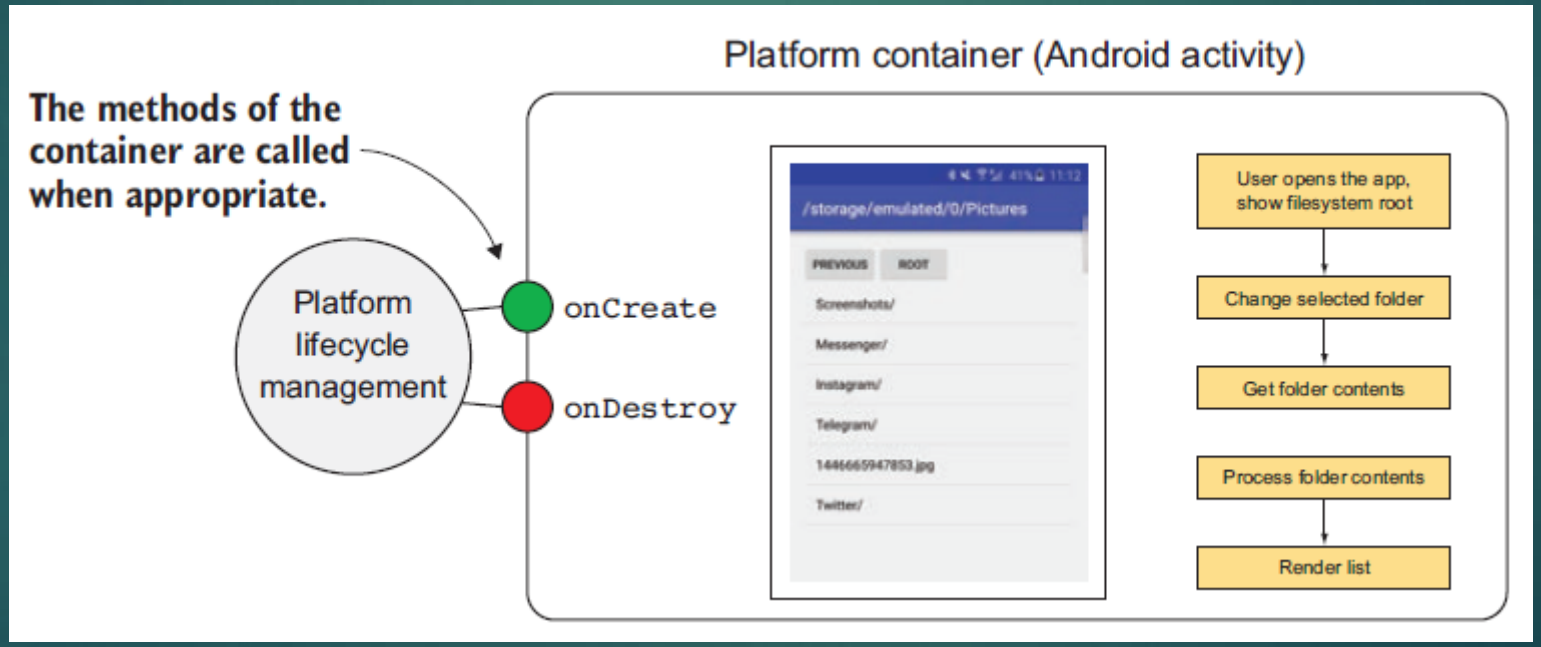
- In terms of classes, most of our examples have consisted of a master class, which contains pretty much everything, and a declaration for the view layout. On Android, this is typically a static XML file.
- We'll call this container class an **owner**. It's a container for your code that the platform (in this case, Android) provides.

Characteristics of platform container

- A platform container is created and managed by the platform operating system.
- The owner could be the application itself, a single screen inside of the application, or an independent component on a screen.
- On Android, the owner is usually Application, Activity, or Fragment.

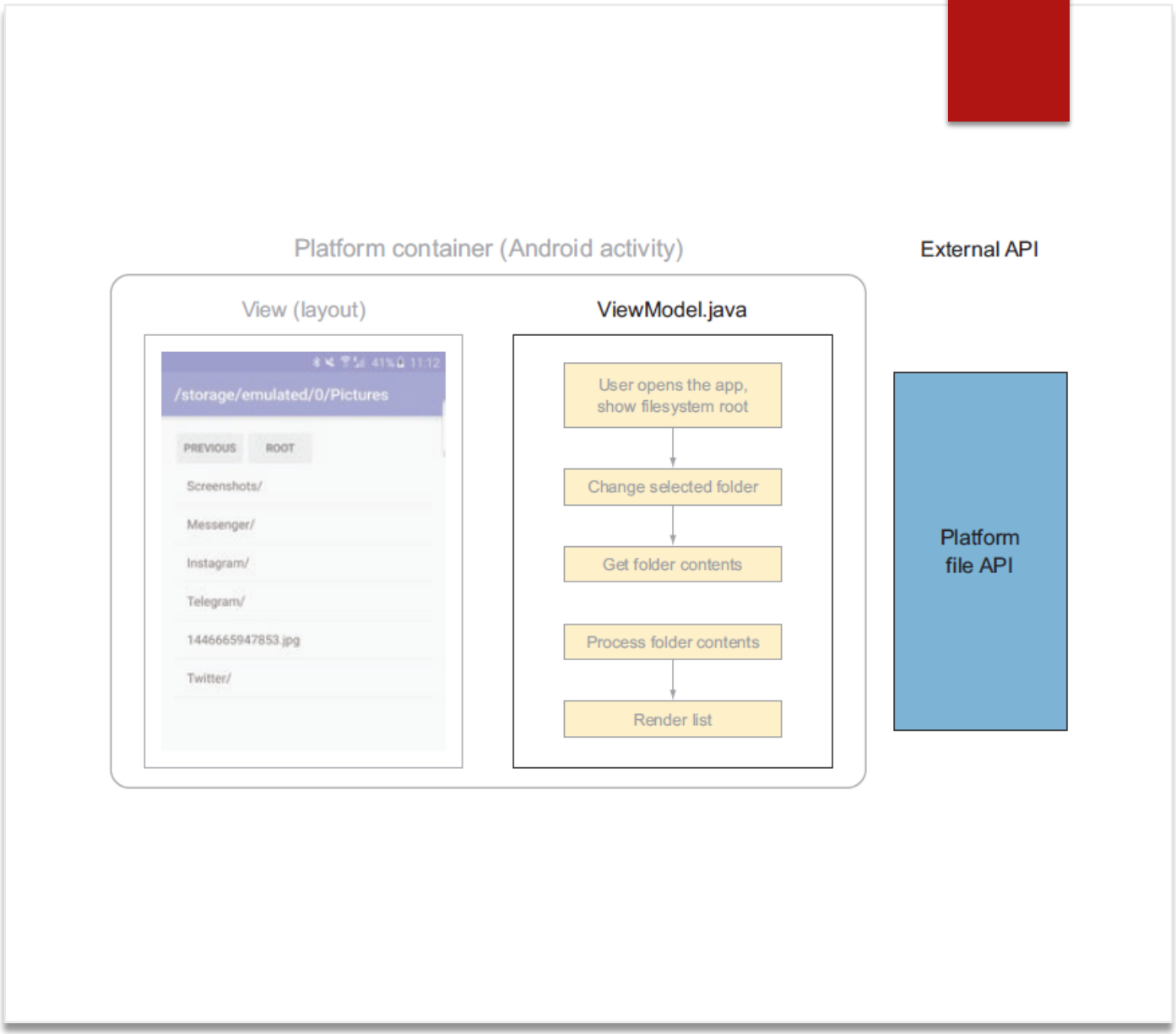
Platform container lifecycle

- It should be noted that you can usually recognize them by their overloaded methods. E.g. onCreate() and onDestroy()



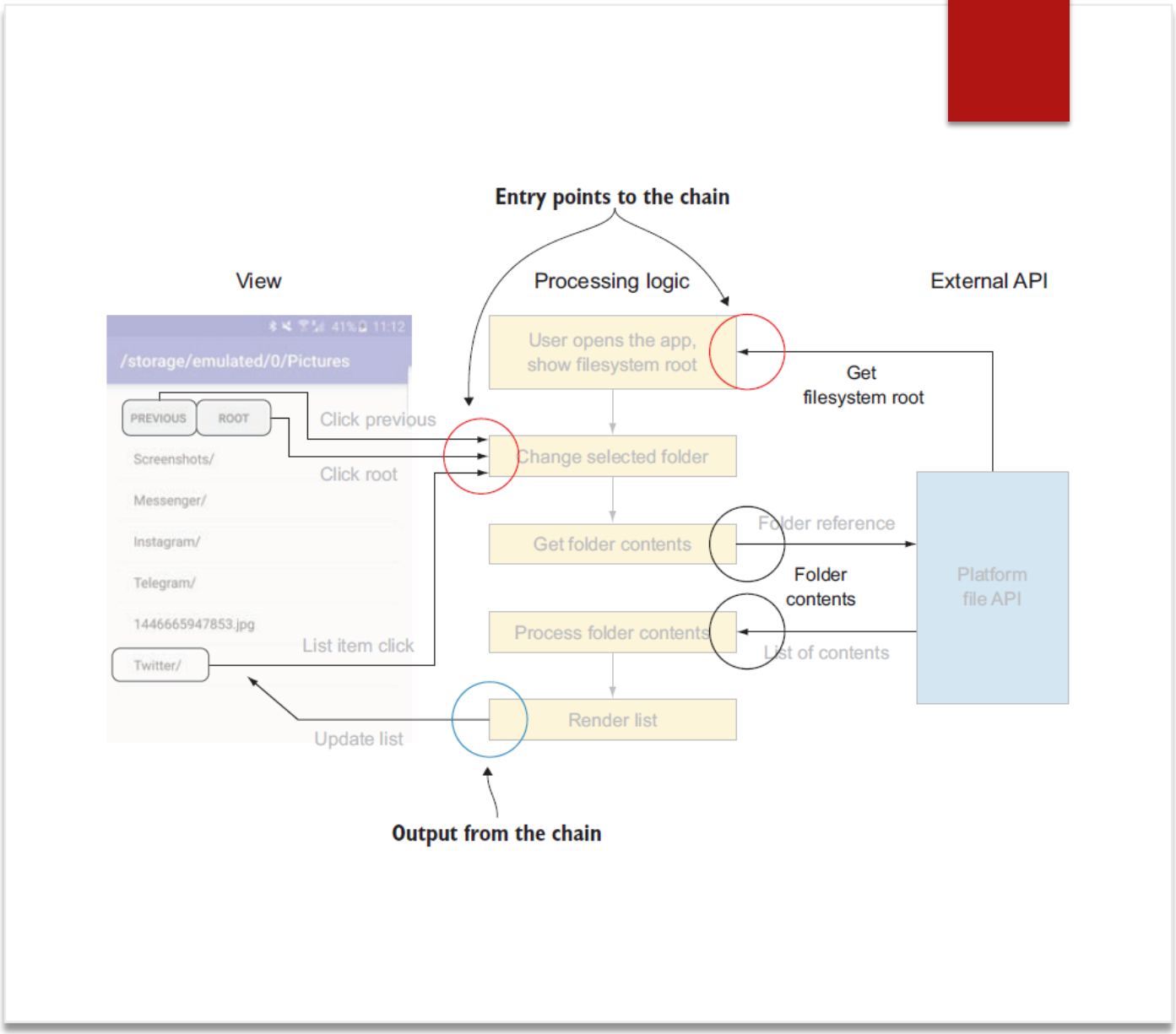
View models

- As a program grows, you want to isolate the logic in its own module (basically a class).
- We'll call this extracted logic the **view model**.



What are the dependencies of the reactive logic?

- A view model needs to encapsulate all the logic needed to run the UI.



The characteristics of a view model

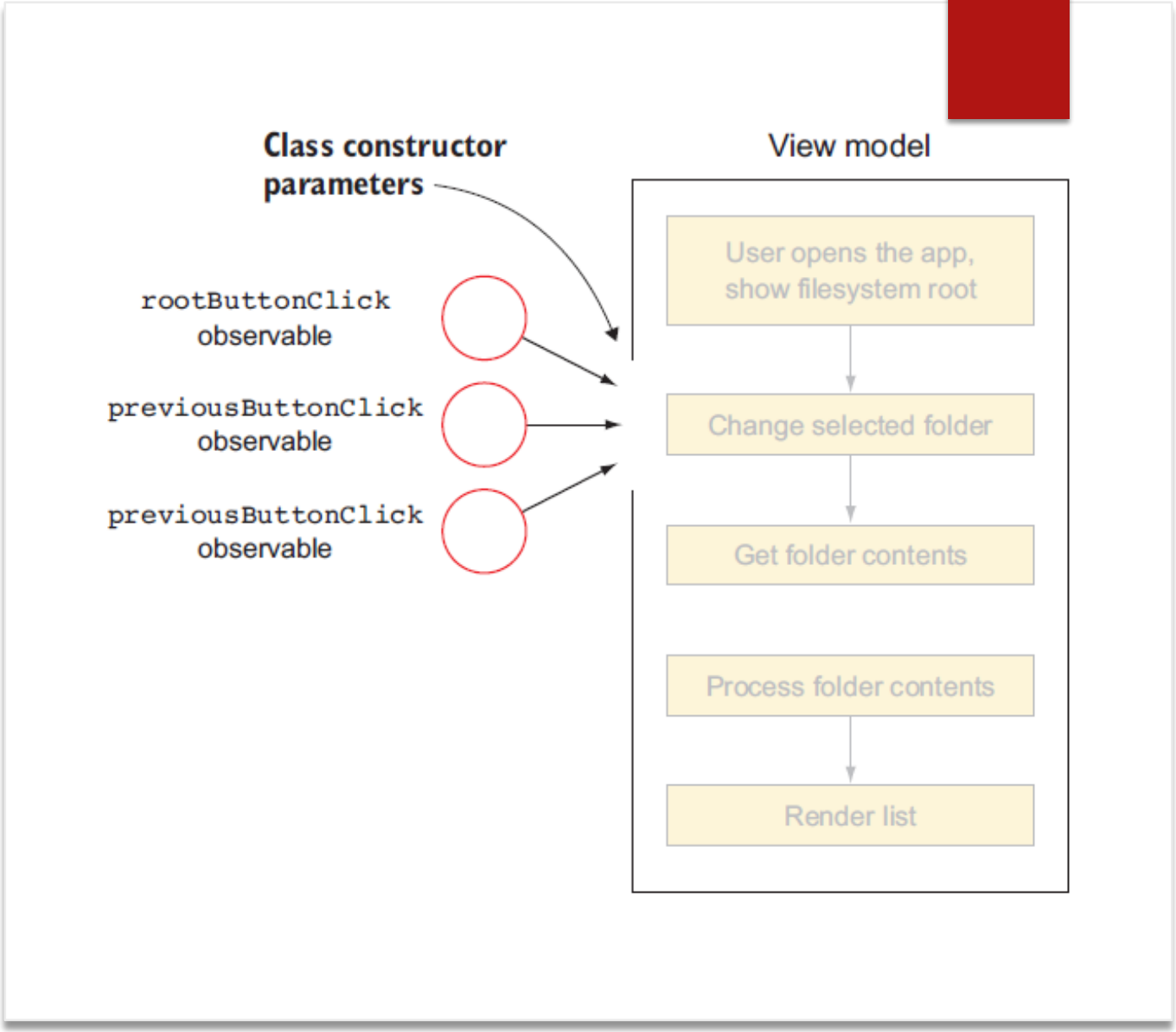
- Doesn't contain references to any platform-specific components
- Exposes as few inputs and outputs as possible
- Doesn't directly use external parts of the code

Migrating existing code into a view model

- The view model can take dependencies in constructor parameters. These include input sources and handles to external APIs.
- The outputs of the view model are typically getter functions that return an observable of the data they wish to expose. But it's important that the observables emit the last value immediately, much like BehaviorSubjects covered before.
- Thread changing is usually done outside of the view model, because it can make testing more difficult. But you'll learn more about that later.

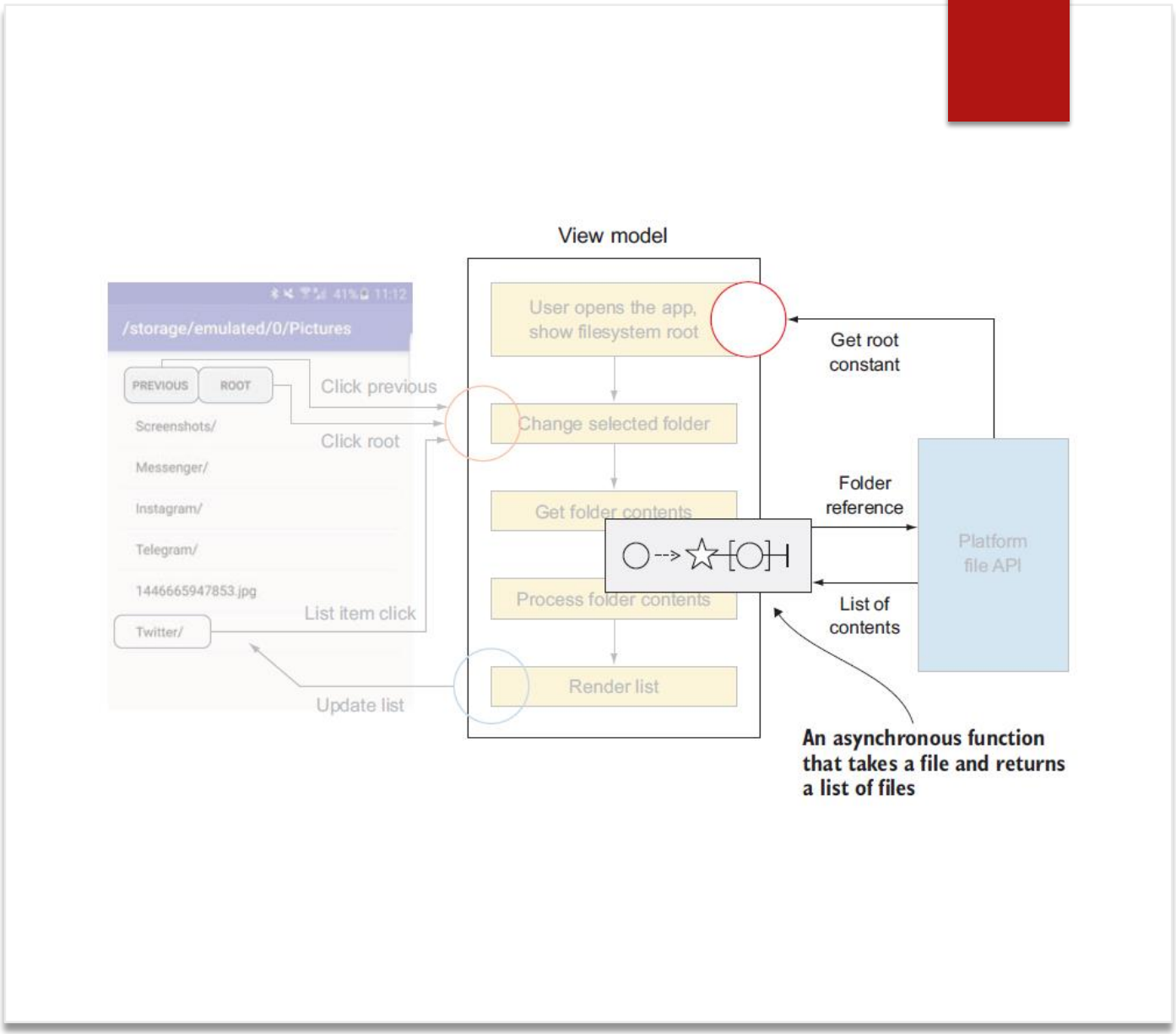
Constructor arguments

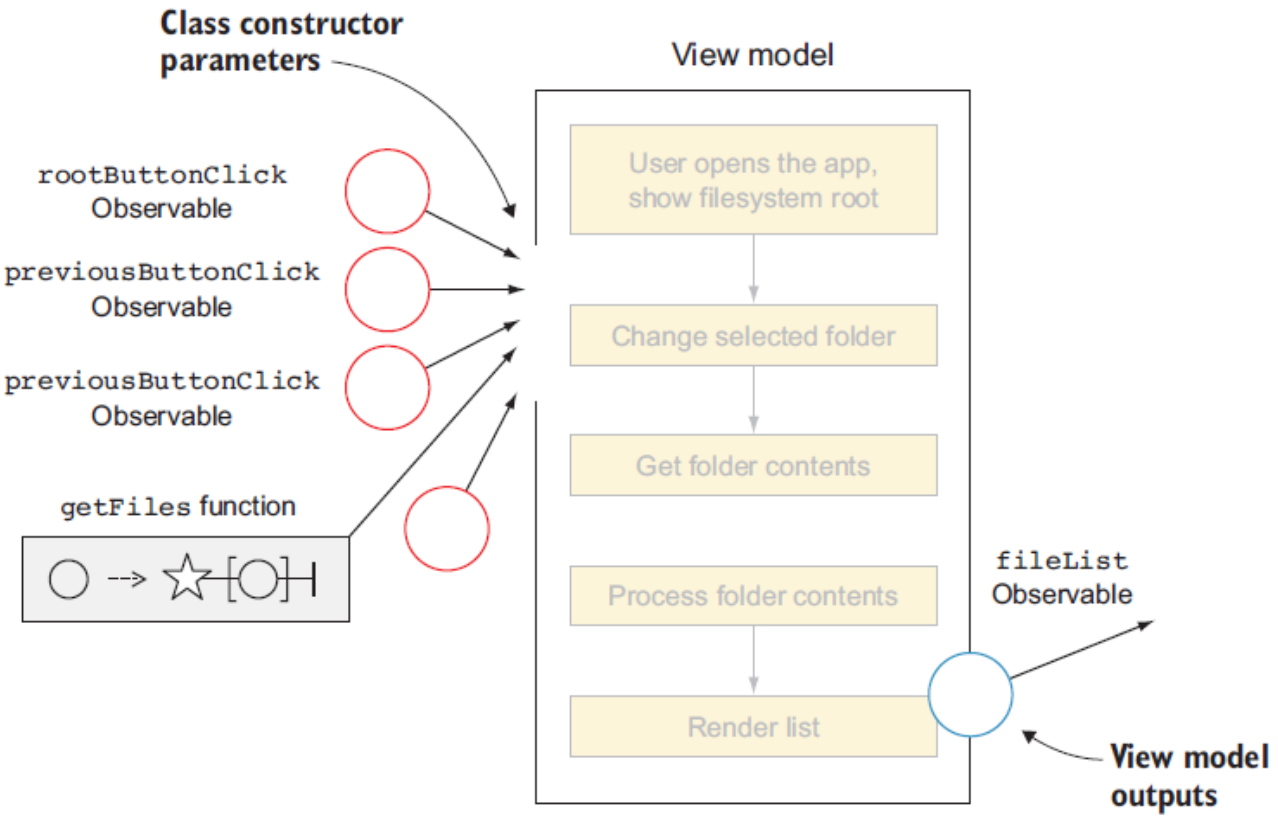
- A simple way to pass inputs to the view model is to use constructor arguments.
- They're a good way too, because they establish a clear relationship between the view model and its creator.



Getting external data in the view model

- To encapsulate the filesystem API without calling it directly from the view model, you can pass it a single asynchronous function.
- The function takes a directory (of type File) and returns an observable that emits the contents of that directory when the operation is ready.





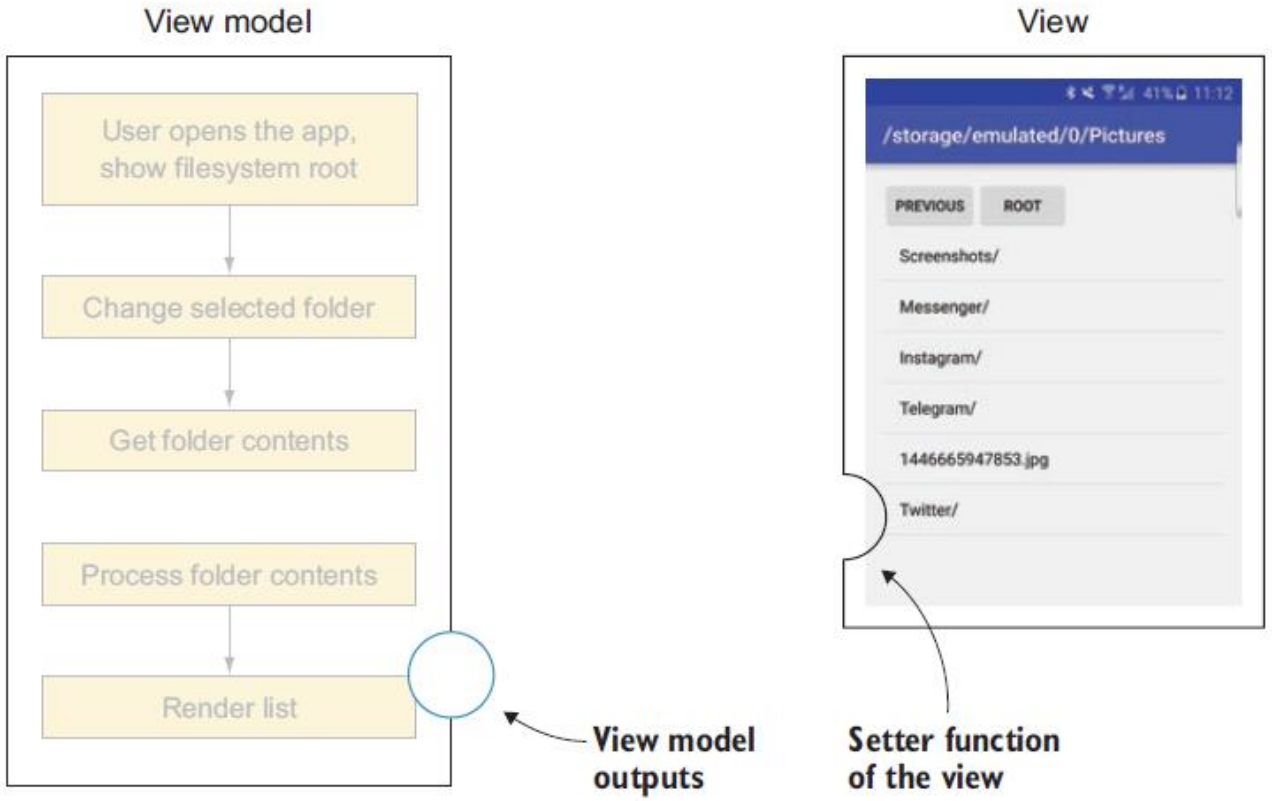
The full constructor of the view model

FileBrowserViewModel.java

```
...

public FileBrowserViewModel(
        Observable<File> listItemClickObservable,
        Observable<Object> previousClickObservable,
        Observable<Object> rootClickObservable,
        File fileSystemRoot,
        Func1<File, Observable<List<File>>> getFiles) {
  this.listItemClickObservable = listItemClickObservable;
  this.previousClickObservable = previousClickObservable;
  this.rootClickObservable = rootClickObservable;
  this.fileSystemRoot = fileSystemRoot;
  this.getFiles = getFiles;
}
```

Connecting views and view models

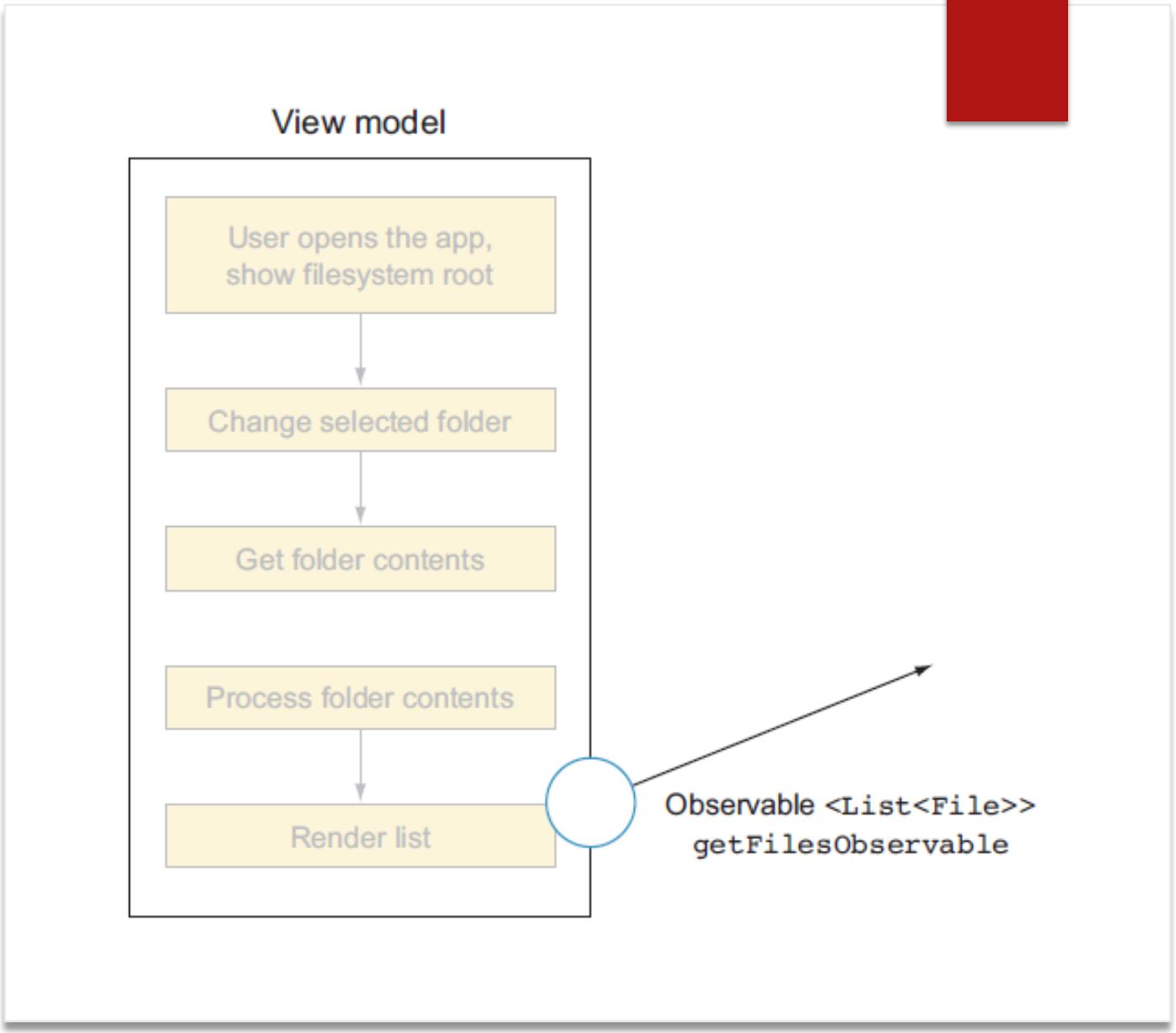
- The name view model comes from the idea that it provides the data, or the model for the view.
- The view in this case isn't necessarily an instance of the Android View class, but rather any part of the application that's able to present data.



Setting up the view and the view model

Exposing outputs from view models

- What you want to expose is a BehaviorObservable that gives the last value immediately and subsequent ones as they're updated.



FileBrowserViewModel.java

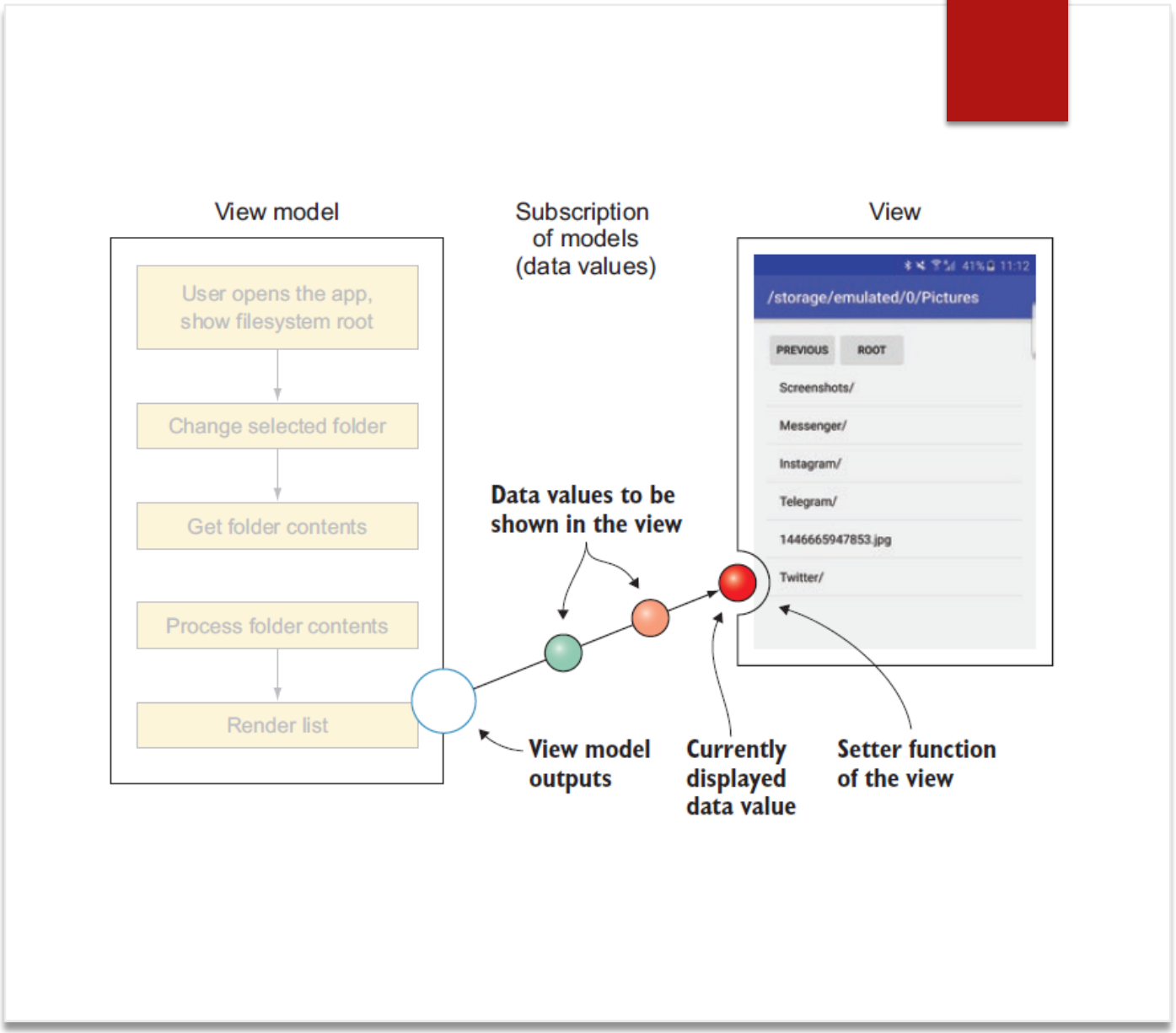
```
private final BehaviorSubject<List<File>> filesOutput
  = BehaviorSubject.create();

...

public Observable<List<File>> getFilesObservable() {
    return filesOutput.hide();
}
```

Subject.hide() makes sure the receiver can't push more events in the subject. It used to be called .asObservable().

The subject is final and has to be initialized only one time. This ensures that whoever subscribes to that subject can be sure it will stay the same as long as the view model exists.

Binding view models to views

- Next, you'll create a subscription between the output of the view model and that setter function of the view.



MainActivity.java initWithPermissions

```
FileBrowserViewModel viewModel =
  new FileBrowserViewModel(
    listItemClickObservable,
    backEventObservable,
    homeEventObservable,
    root, this::createFilesObservable
);
```

Create the view model with its dependencies

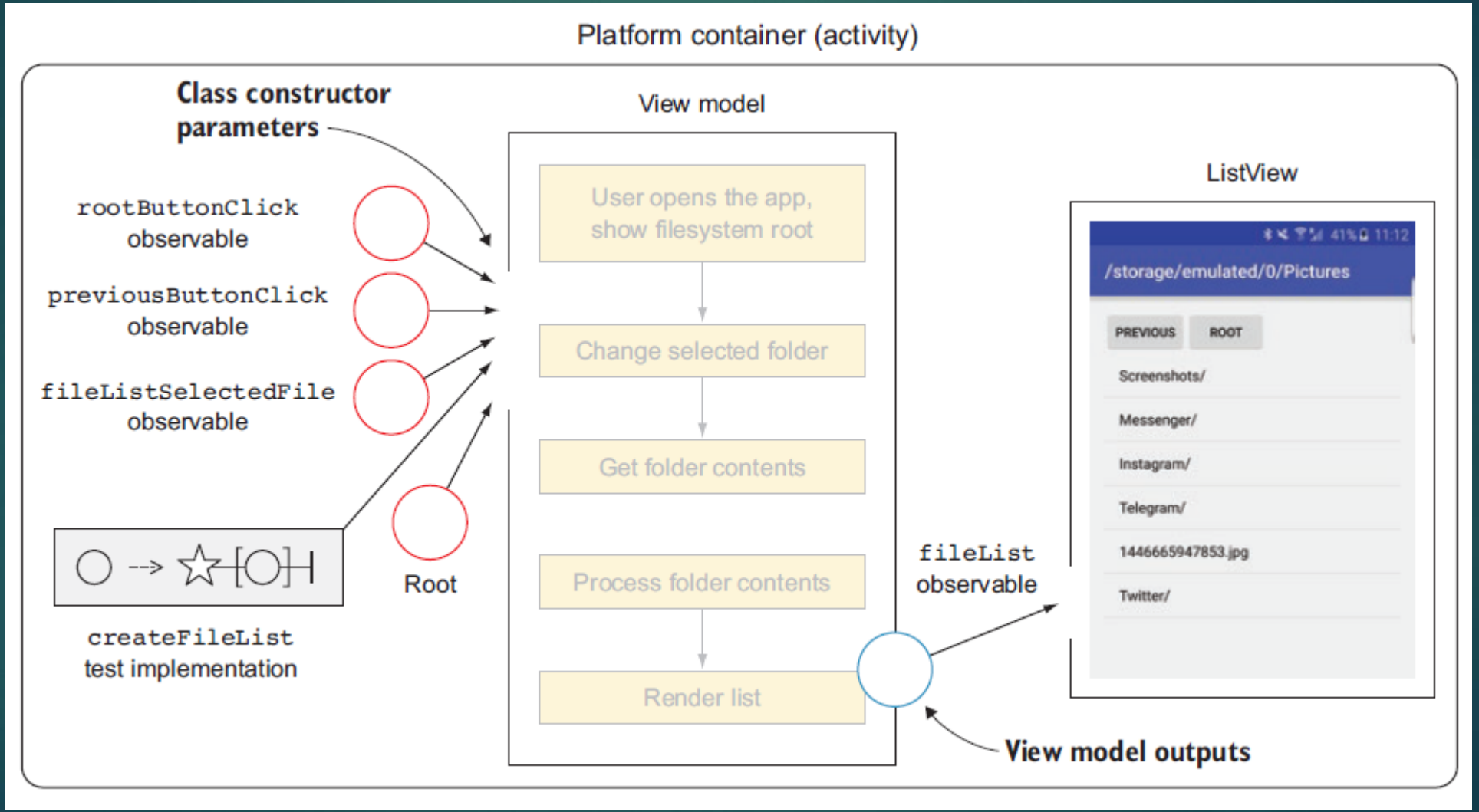
MainActivity.java initWithPermissions

```
viewSubscriptions.add(
  viewModel.getFileListObservable()
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(this::setFileList)
);
```

The list of subscriptions represented as Disposable instances

Binding with a subscription

The whole picture



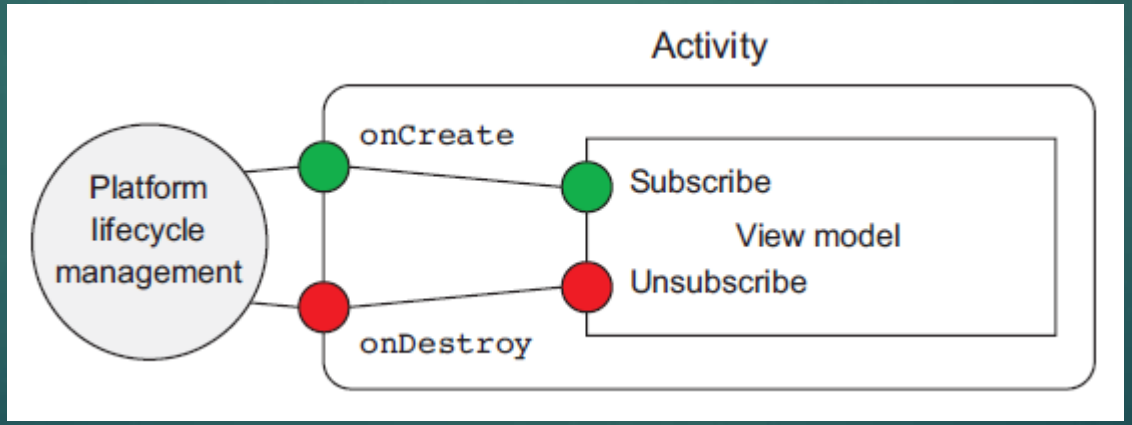
View model lifecycle

- The problem is that a part of it creates subscriptions that we were saving in a CompositeDisposable in the Activity.
- Now if you move the code in the view model, what do you do with the subscriptions?

```
subscriptions.add(selectedFile
        .flatMap(createFilesObservable)
        .subscribe(filesObservable::onNext));
```

Saving subscriptions in a view model

- To manage our subscriptions, add another CompositeDisposable to the view model instance that will keep track of all subscriptions that it has created.
- You'll add functions for the view model to create the subscriptions as well as to release them.



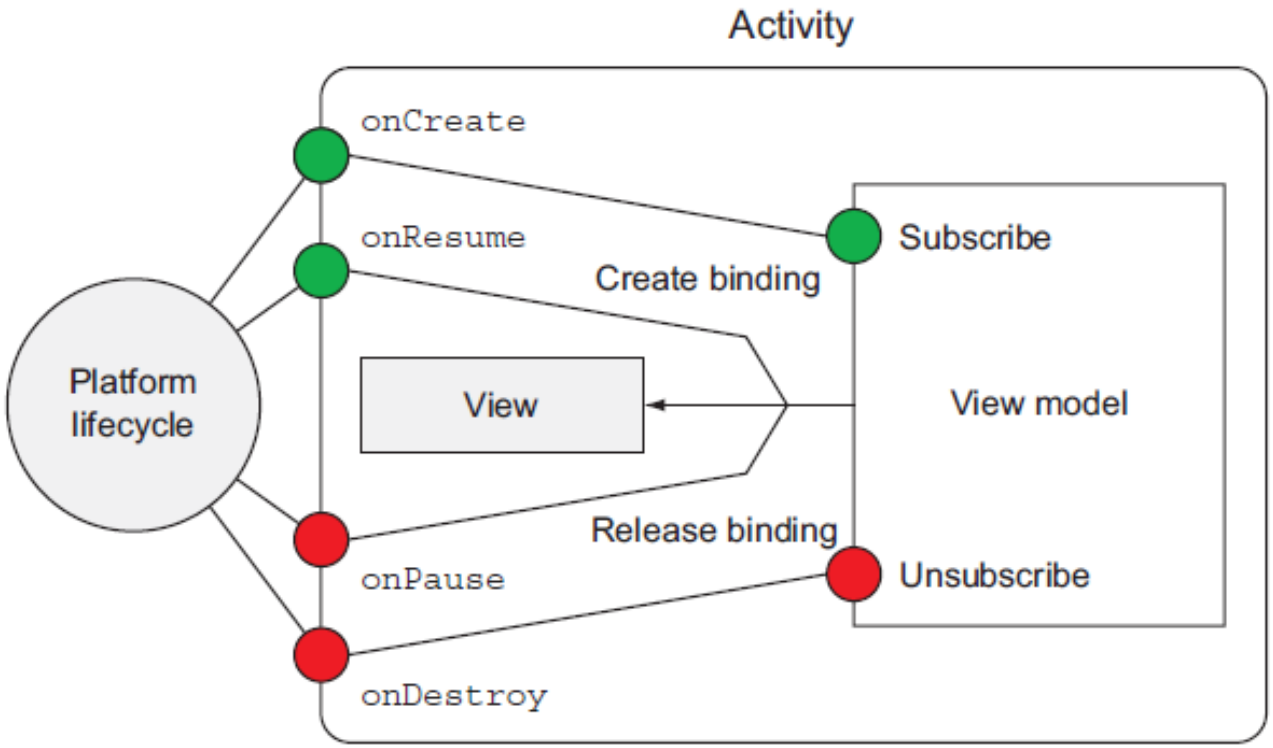
The code of the view model

FileBrowserViewModel.java subscribe function

```
public void subscribe() {
  final BehaviorSubject<File> selectedFile =
          BehaviorSubject.createDefault(fileSystemRoot);

  Observable<File> previousFileObservable =
          previousClickObservable
                  .map(event ->
                          selectedFile.getValue()
                                  .getParentFile());

  Observable<File> rootClickObservable =
          rootButtonObservable
                  .map(event -> fileSystemRoot);

  subscriptions.add(Observable.merge(
          listItemClickObservable,
          previousFileObservable,
          rootFileObservable)
          .subscribe(selectedFile));

  subscriptions.add(selectedFile
          .switchMap(getFiles)
          .subscribe(filesObservable::onNext));
}
```

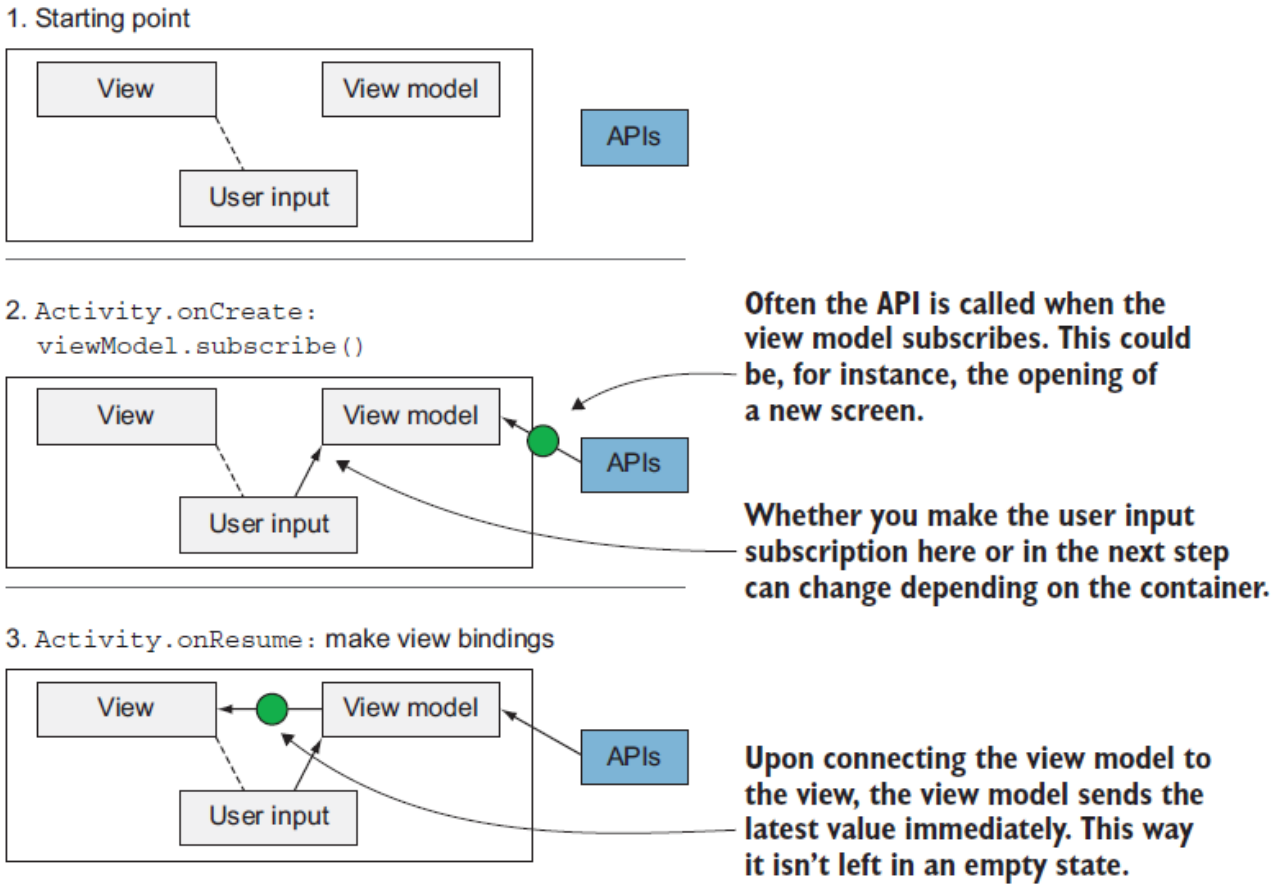
This might look a bit strange, but it means that you want to emit the fileSystemRoot every time the user clicks the Home button.

Here you use the function that was given to you. It takes a file (folder) and returns its contents asynchronously.



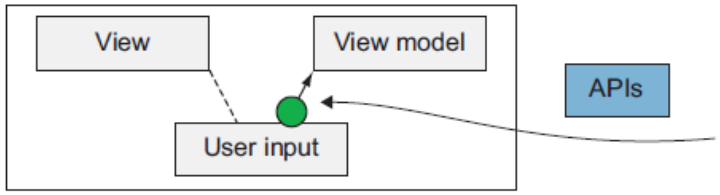
View models and the Android lifecycles

View model phases on Android



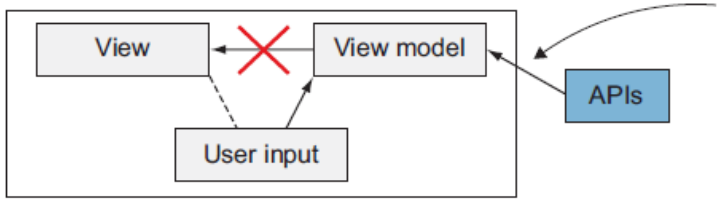
View model phases on Android

4. User uses the app



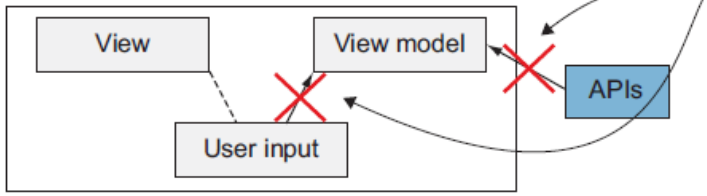
The user starts interacting, thus producing input to the view model. This might trigger new API operations as well.

5. `Activity.onPause`: release view bindings



After disconnecting the view binding you can still receive updates from the API. They just wouldn't be shown in the view (UI) until possibly rebinding the view.

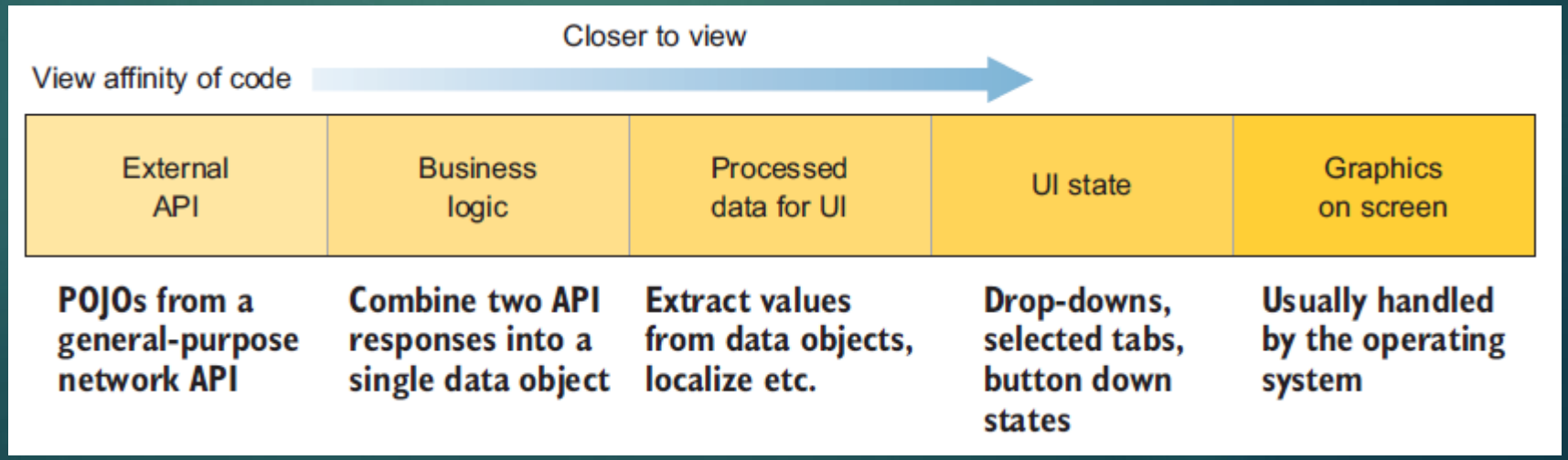
6. `Activity.onDestroy`: `viewModel.unsubscribe()`



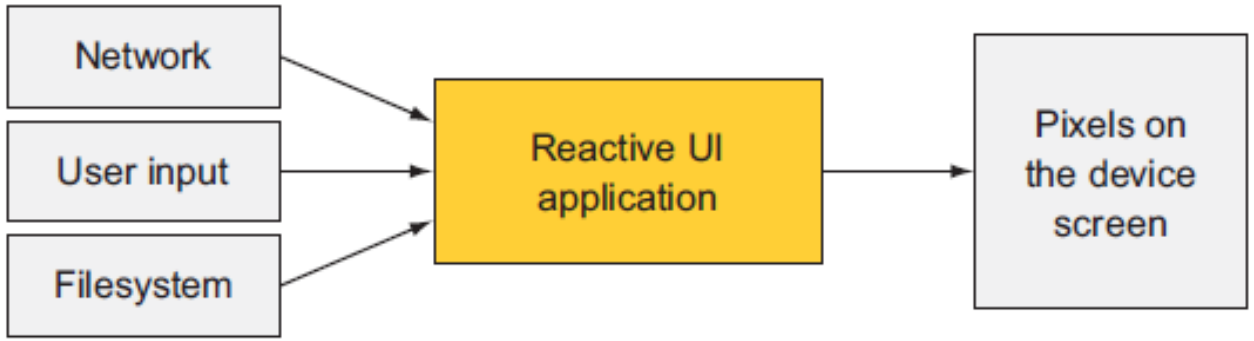
All remaining subscriptions are released. This is important to prevent possible memory leaks.

The view affinity of the code

- These pieces of code, sometimes functions, all have different, distinct, responsibilities.
- In terms of a reactive application, you can roughly identify sections that are more about just data and ones that center around showing something to the user.
- The more our code is affected by the view, the stronger is its view affinity.

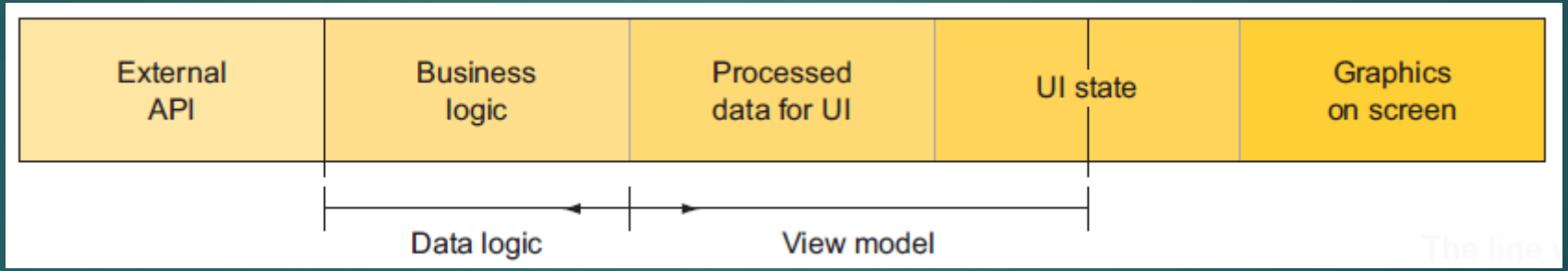


The Reactive UI Application with a View



Where do view models fit on our spectrum?

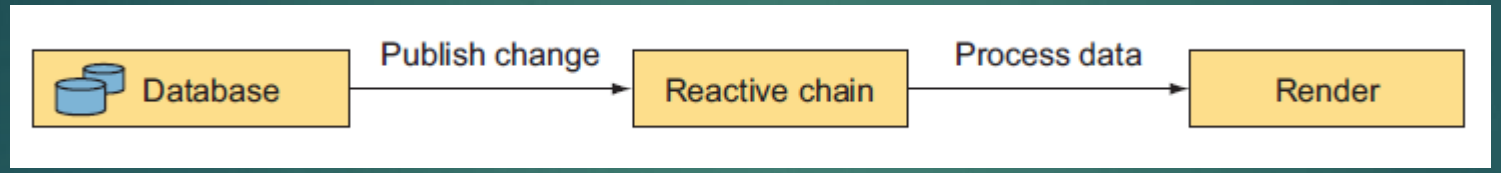
- As mentioned, view models are containers for the reactive logic, which can be everything from external APIs to the final view itself.
- Typically, a view model, however, covers a big chunk right before the view, and the rest is considered generic data/business logic.



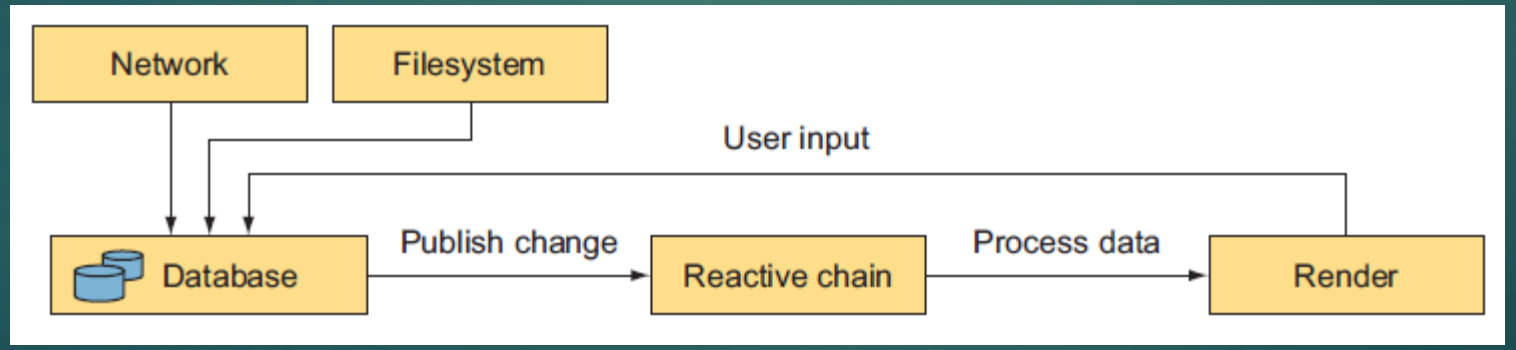
Reactive Architectures

Fundamentals of reactive architectures

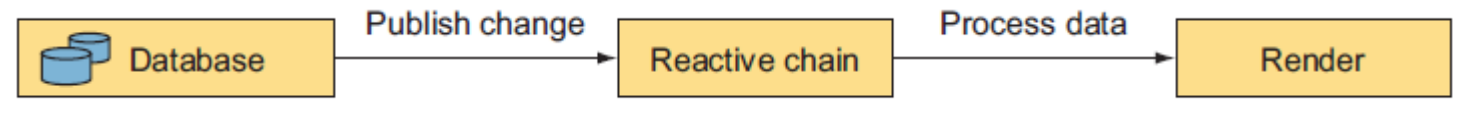
- Data change, process data, render.



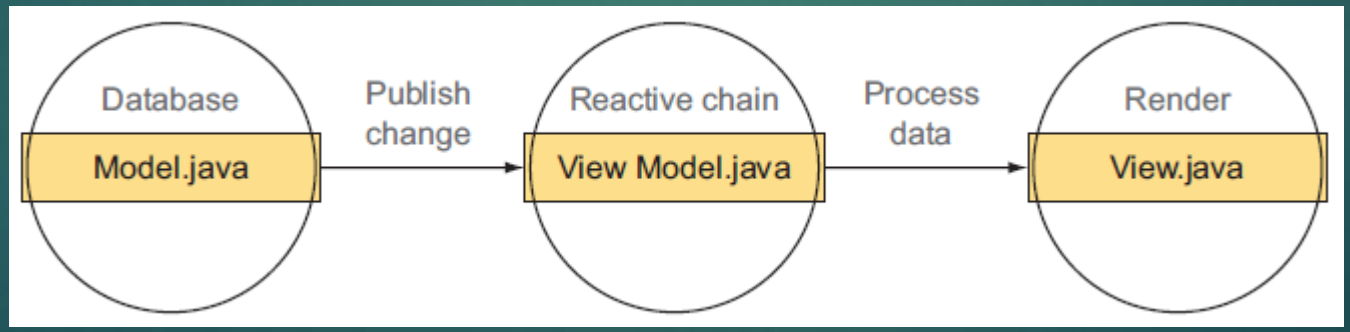
- On the other hand, you have mechanisms for updating the database. These include incoming data from the network, user input, or something read from the disk.

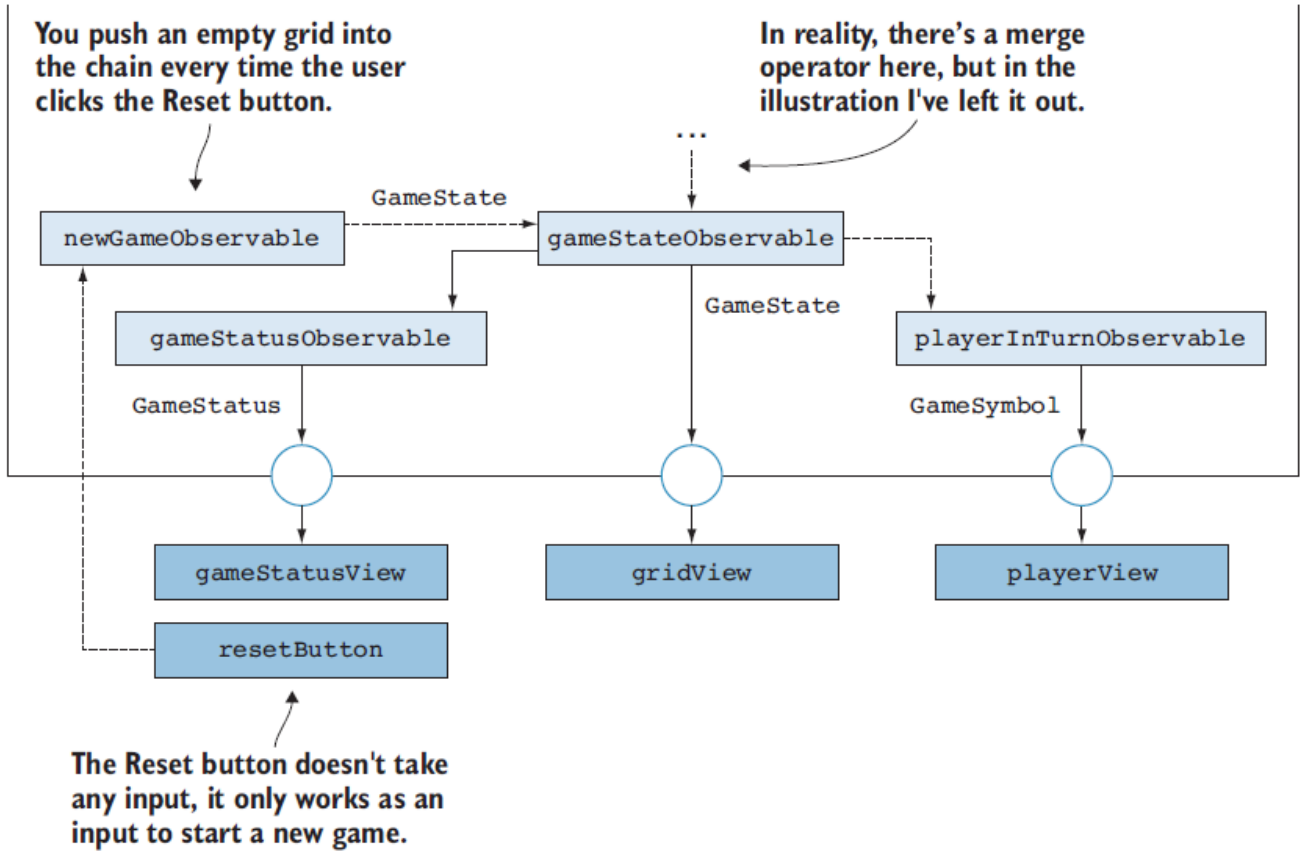


Model-View-View model



- What you have in the middle, then, is the reactive code. So far, you've just put that code inside a container, such as Activity, but now you'll name it and see how to define it as a separate part of the architecture.
- The idea is to separate the reactive chain from the store as well as the view. You'll call the rendering part the **view** from here on, and the reactive chain becomes the **view model**.

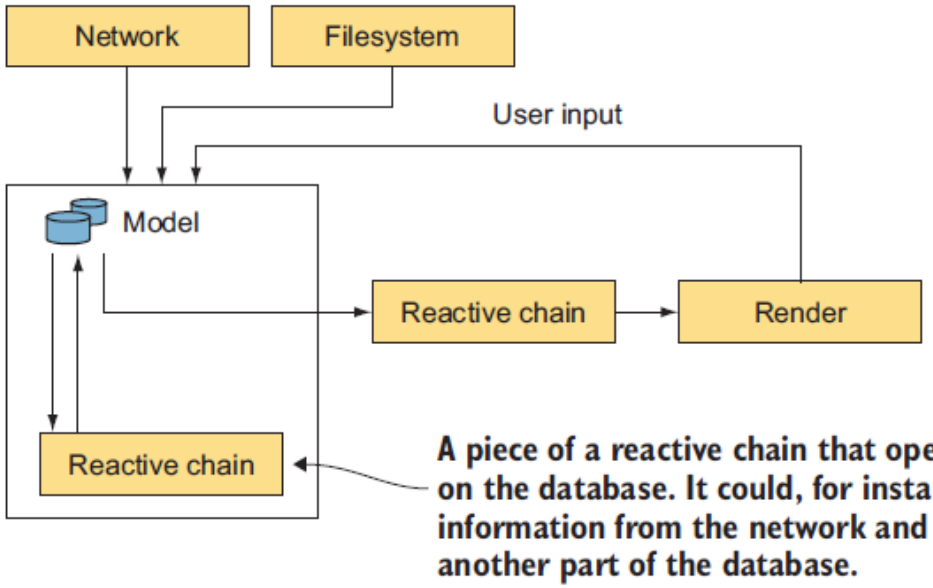




Splitting classes - decouple

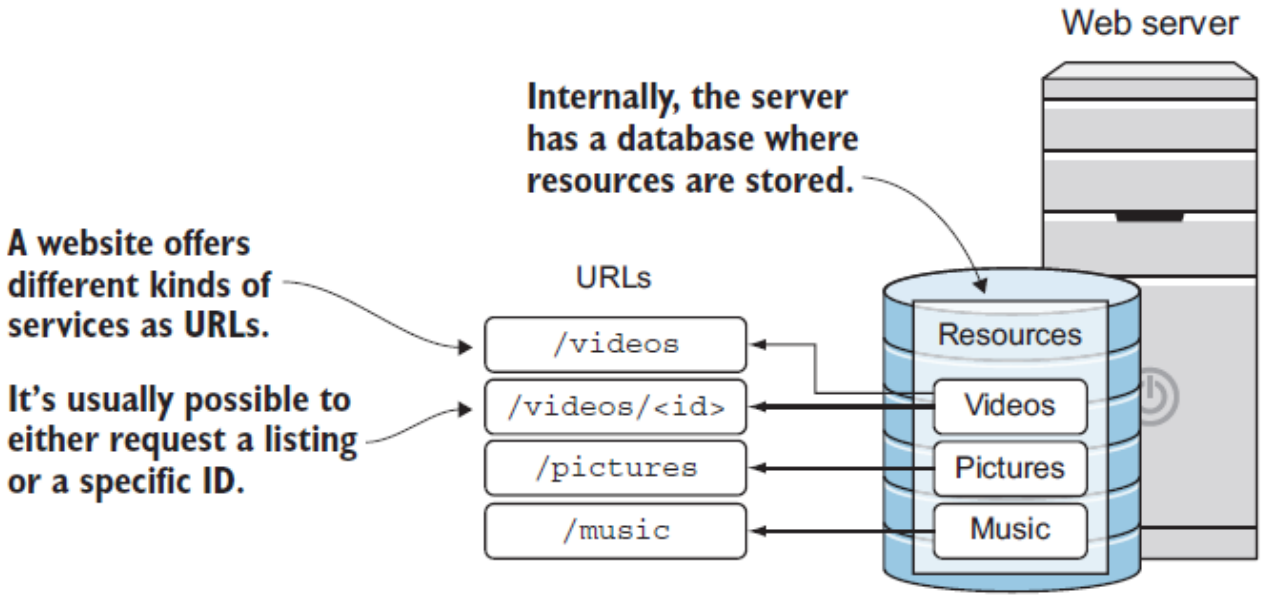
Internal relationships of the model

- You can also put in some processing logic that's invisible to the consumer of the database.
- This could be considered to represent relationships between different parts of the database.



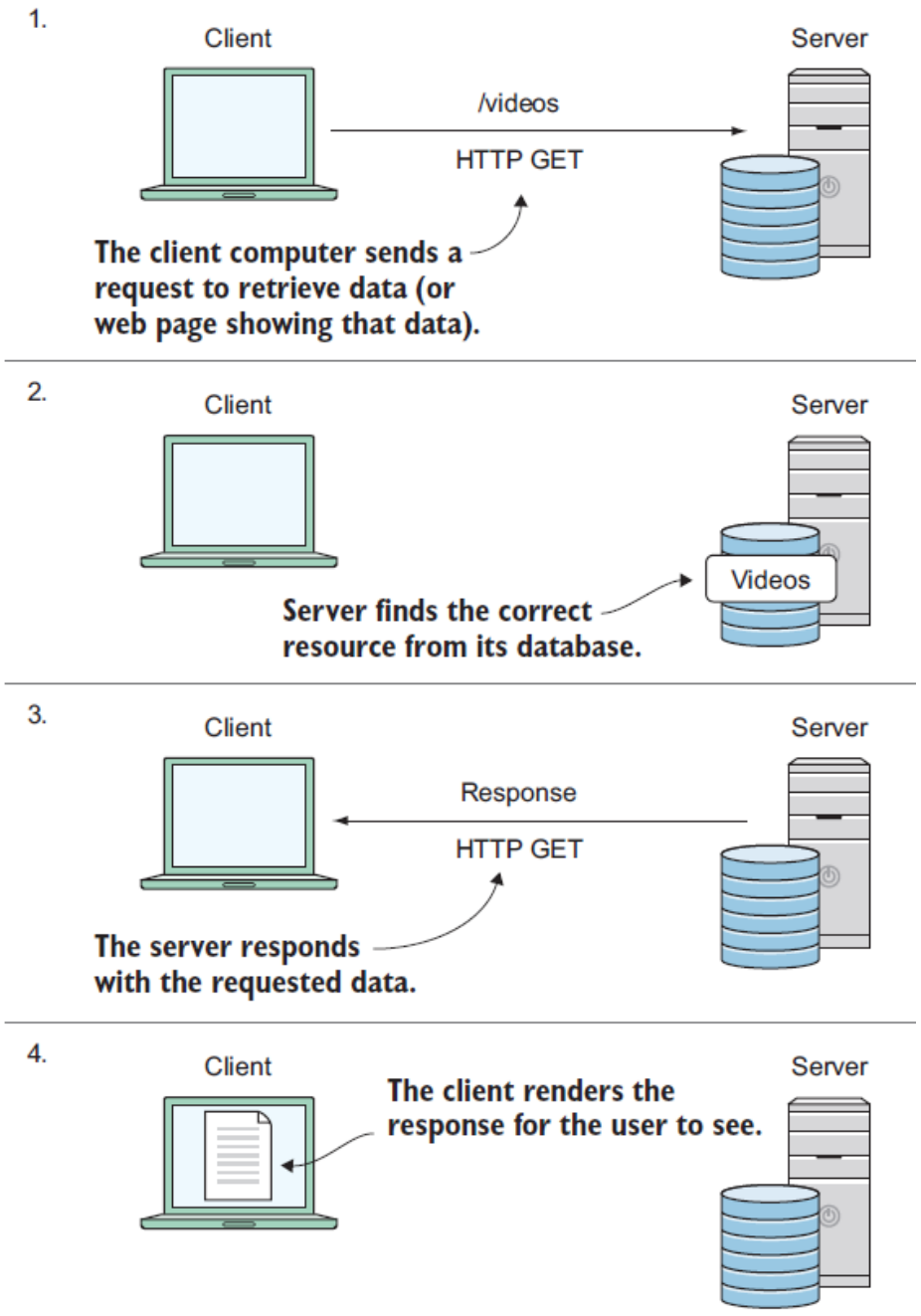
Reactive model

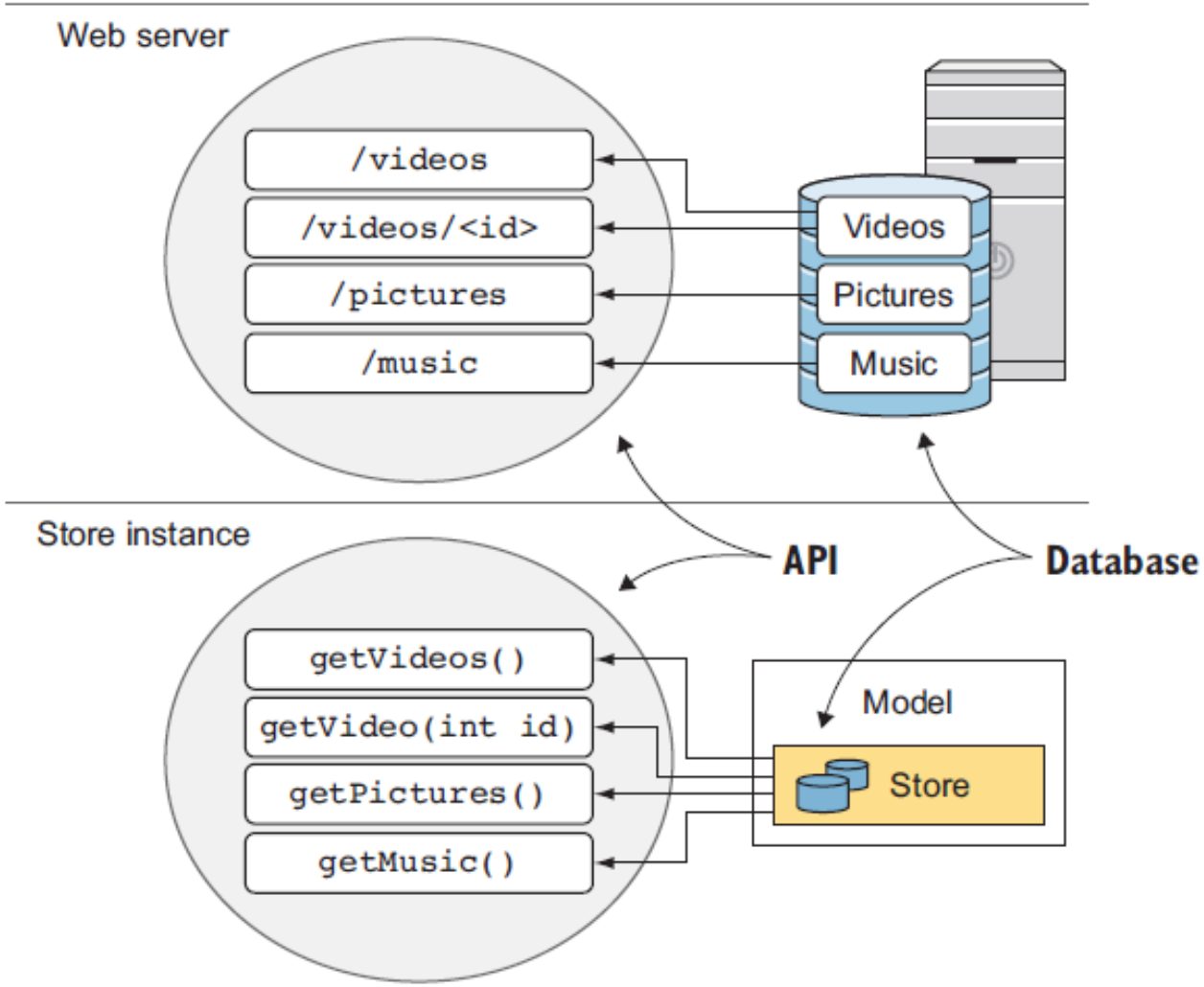
- On Android, there are currently no strongly opinionated popular frameworks to start doing reactive programming.
- Instead, the approach is to take bits and pieces from libraries that fit what you want to do.
- You already saw RxJava, which is the cornerstone of what you're planning to build.
- It provides the glue between different parts of the architecture, as well to conveniently create the processing chains.



Web server as a repository of entities

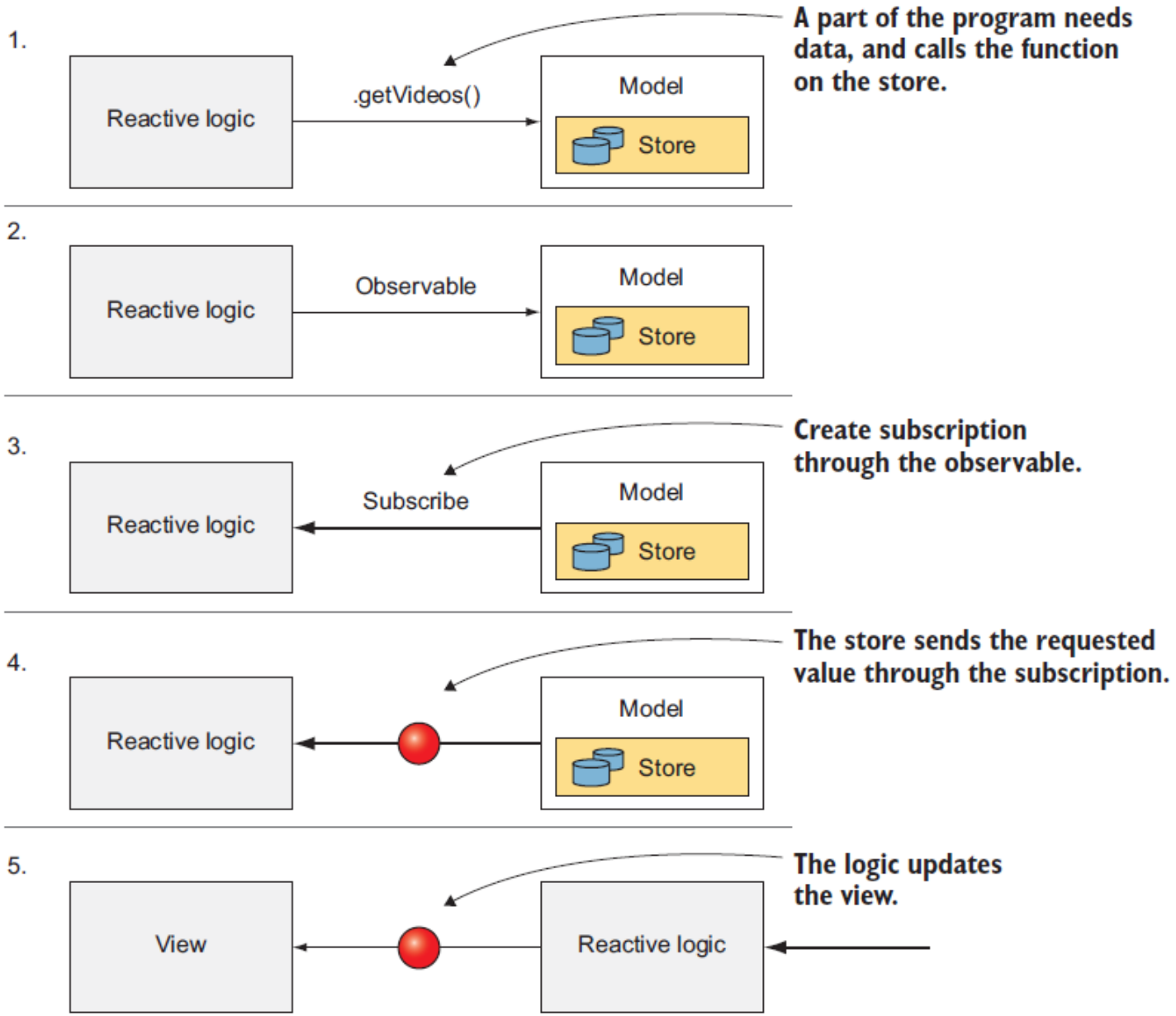
Web request flow



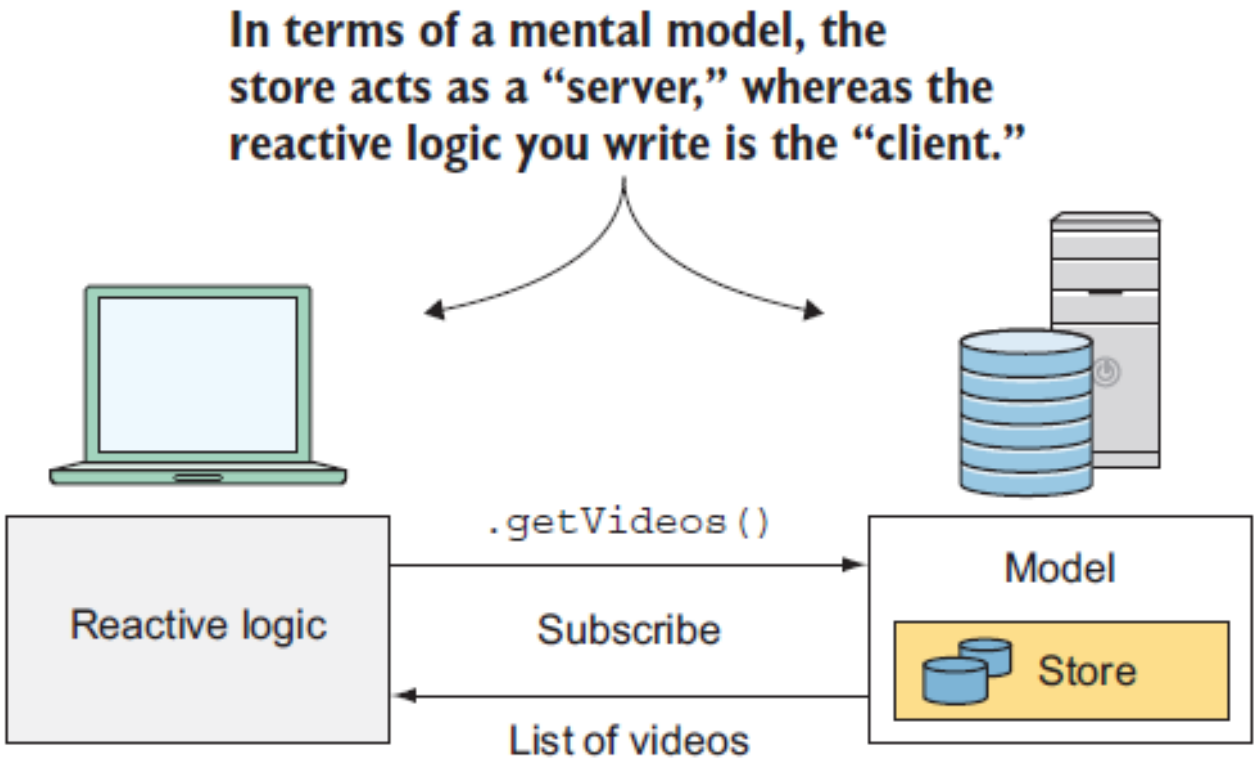


Model as a repository of entities

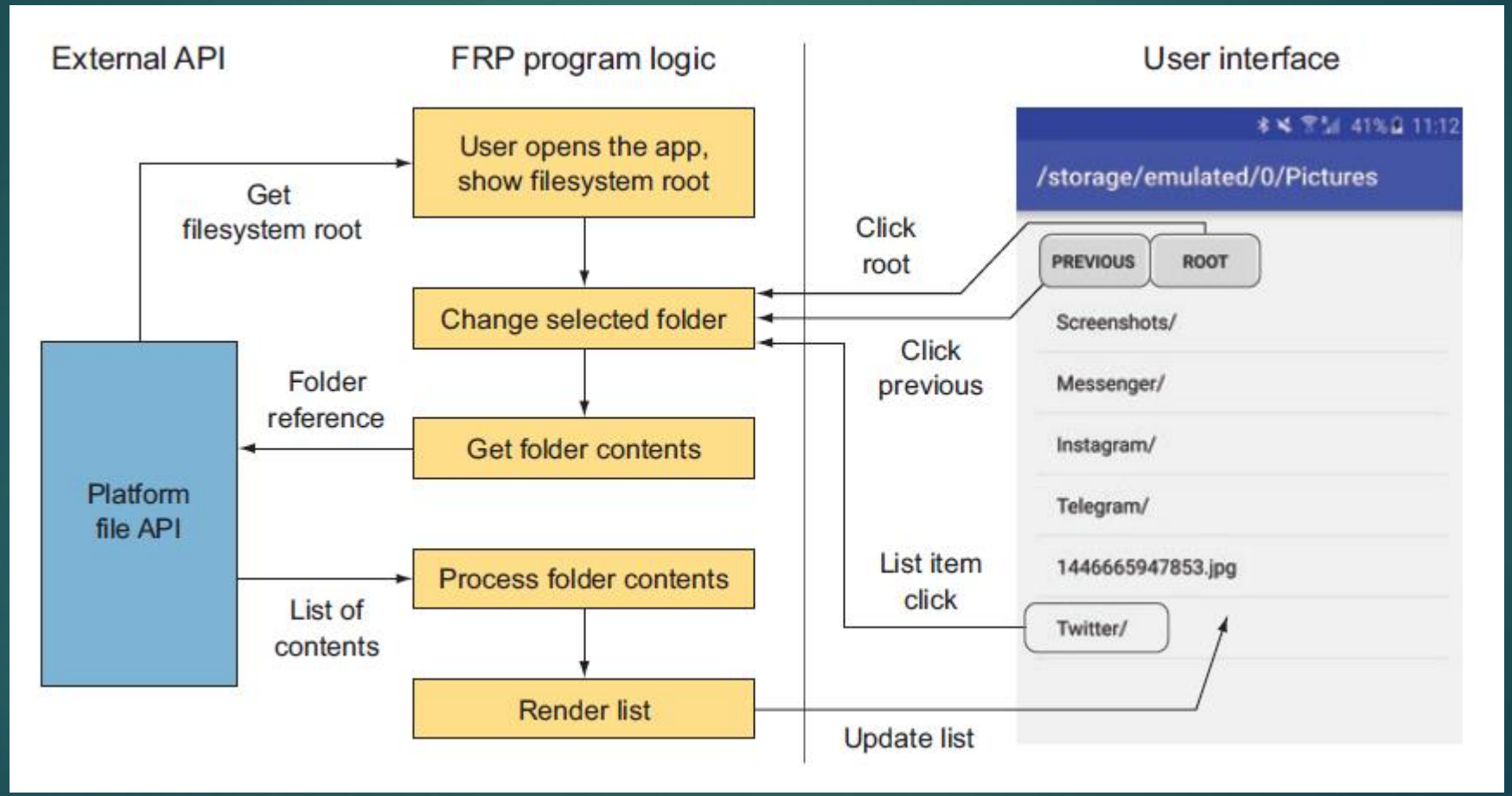
Retrieving data from the model



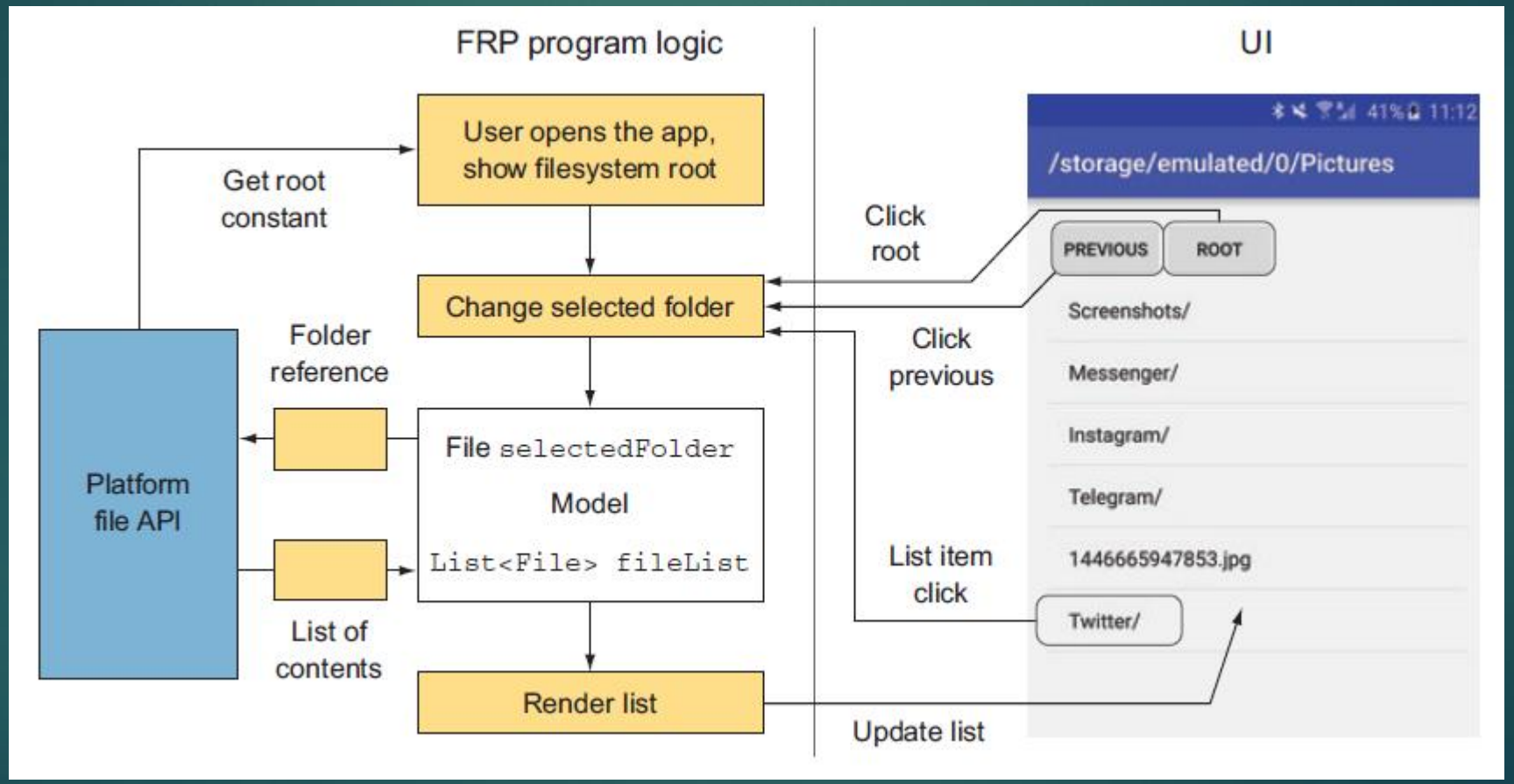
A piece of code as a "client"



Revising the file browser

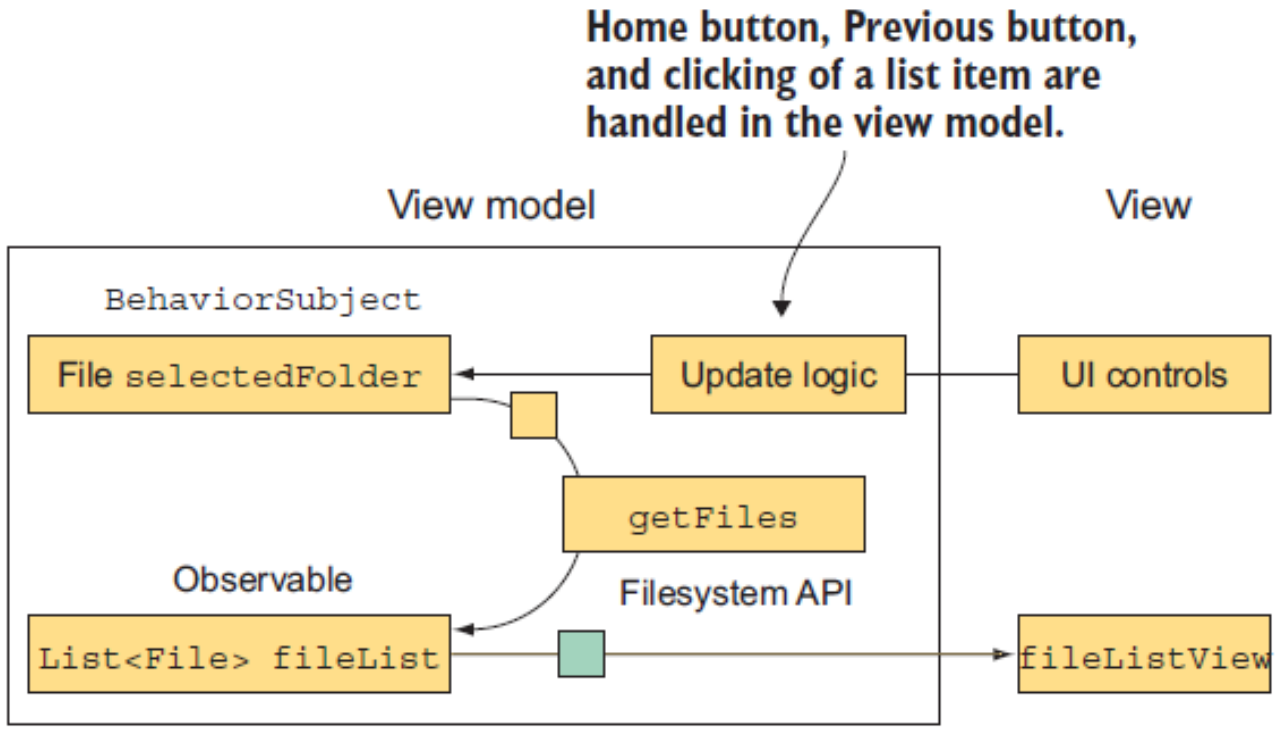


File browser graph with a model



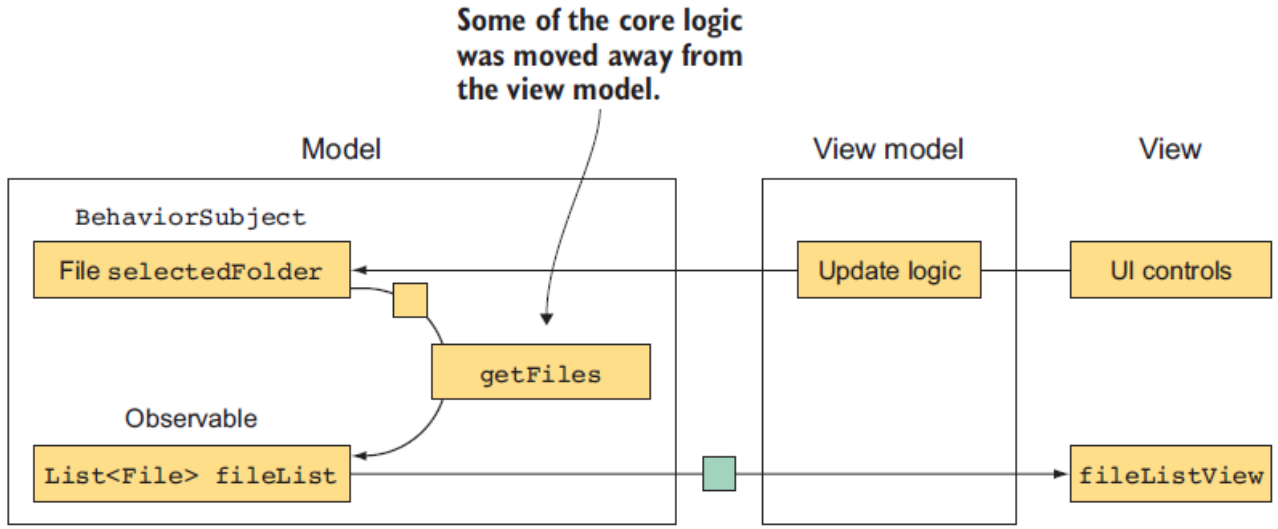
Constructing the model for the file browser – existing

- In the file browser you have two main observables for state: the selected file (folder), and the list of files within that folder.
- These two observables were linked together in the view model in a way that calculates the fileList based on the selectedFolder.



Moving state from view model into the model

- You'll move all the state from the view model to the model.
- This includes the selected folder and the resulting file list.
- This means you can take this part out of the view model as well and move it into the model instead.
- Our proposed architecture will thus be a Model-View Model-View with divided responsibilities.



FileBrowser model implementation

```
public class FileBrowserModel {
  private final BehaviorSubject<File> selectedFile
    = BehaviorSubject.createDefault();
  private final Observable<List<File>>
    filesListObservable;

  public FileBrowserModel(
      File fileSystemRoot,
      Function<File, Observable<List<File>>>
        getFiles) {
    filesListObservable = selectedFile
      .switchMap(file ->
        getFiles.apply(file)
          .subscribeOn(Schedulers.io())
      );
  }

  public Observable<File> getSelectedFile() {
    return selectedFile.hide();
  }

  public void putSelectedFile(File file) {
    selectedFile.onNext(file);
  }

  public Observable<List<File>> getFilesList() {
    return filesListObservable;
  }
}
```

The model contains selectedFile as a mini "store." The BehaviorSubject from the view model was moved here.

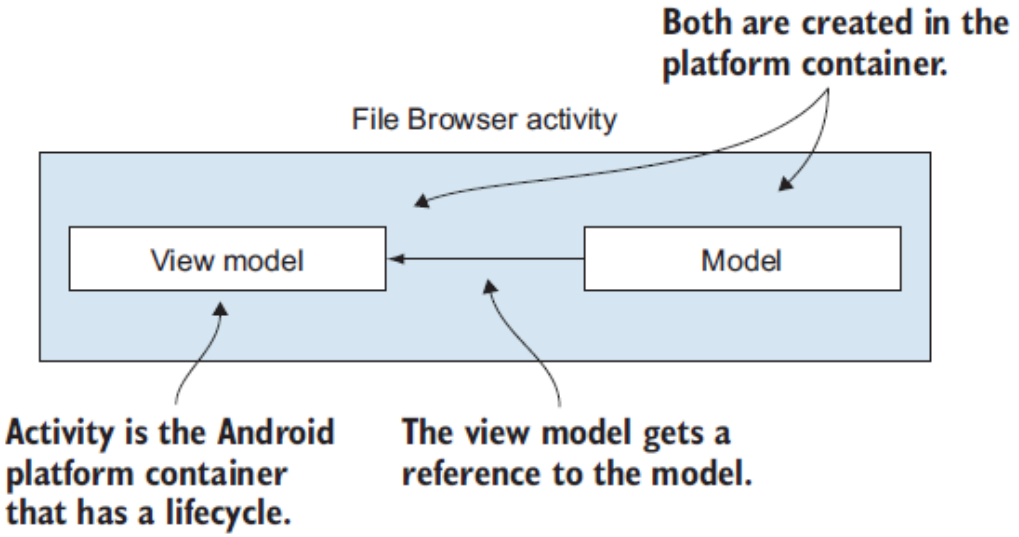
I omitted error catching here; you can see the full code in the online examples.

You don't expose the subjects directly from the model. It has full control over its data.

The reason you say "put" instead of "set" is semantics: you don't "set" the value of the selected folder but "push" another value that will replace it.

Creating the model

- Where you create the model depends on who needs to use it and how long it needs to exist.
- In our case, you have only one Android activity, so you can create your model there.
- It just shouldn't be created in a view model - that would be strange, considering the view model is its consumer, not the owner. The activity (owner) makes the connection.



Creating the model - codes

- In terms of code, you can connect the view model to the model in the onCreate function of the MainActivity. (The view model will be activated later in initWithPermissions because you first need to ask for filesystem privileges.)

MainActivity.java onCreate

```
fileBrowserModel =
  new FileBrowserModel(this::createFilesObservable);

viewModel = new FileBrowserViewModel(
  fileBrowserModel, listItemClickObservable,
  backEventObservable, homeEventObservable,
  root
);
```

The getFiles function was moved from the view model to the model. The view model no longer has a direct reference.

```
// Somewhere in the initialization code
BehaviorSubject<File> selectedFolderSubject = ...;
...
Observable.merge(
        listItemClickObservable,
        fileChangeBackEventObservable,
        fileChangeHomeEventObservable)
        .subscribe(
          selectedFolderSubject::onNext
        );
```

Updating the model from the view model – old codes

```
// Somewhere in the initialization code
Observable<File> selectedFolder
  = fileBrowserModel.getSelectedFolder();
...
Observable.merge(
        listItemClickObservable,
        fileChangeBackEventObservable,
        fileChangeHomeEventObservable)
        .subscribe(
          fileBrowserModel::putSelectedFolder
        );
```

Updating the model from the view model - updated

Removing the logic from the view model – old codes

FileBrowserViewModel.java subscribe method

```
subscriptions.add(selectedFile
   .switchMap(file ->
     getFiles.apply(file)
       .subscribeOn(Schedulers.io()))
```

FileBrowserViewModel.java subscribe method

```
subscriptions.add(fileBrowserModel.getFilesList()
   .subscribe(filesSubject::onNext));
```

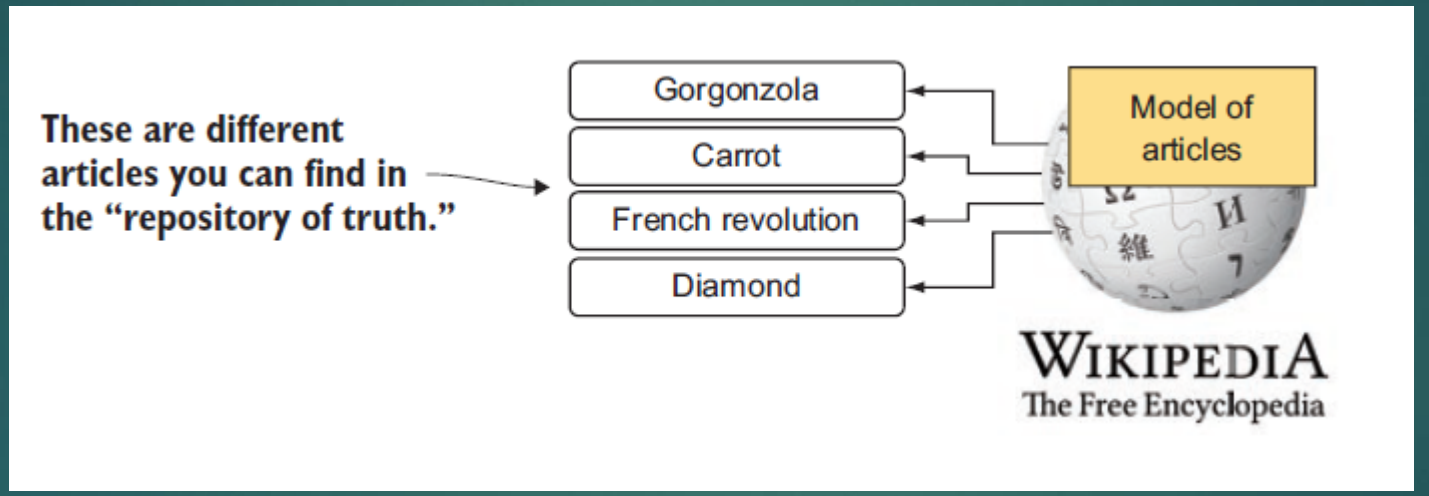
Removing the logic from the view model - updated

Rules of the model and its consumers

- The model is the only source of truth.
- The model gives the latest value first.
- All consumers of the model have to be ready to receive updates.

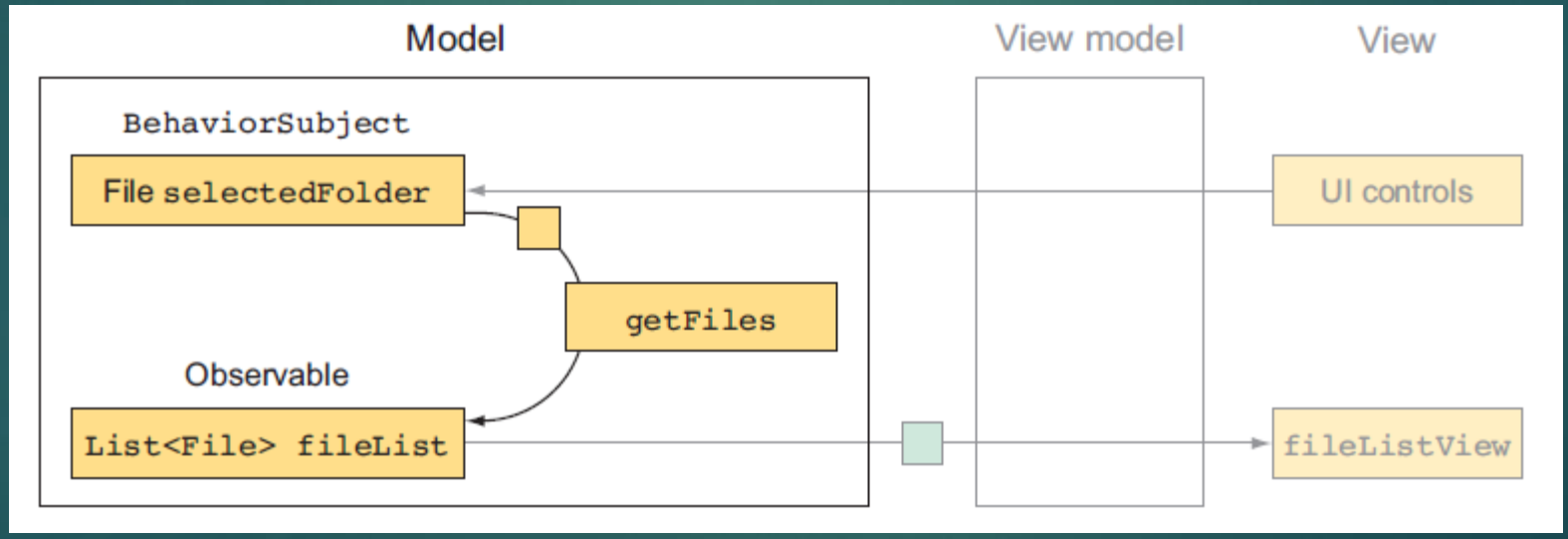
Single source of truth

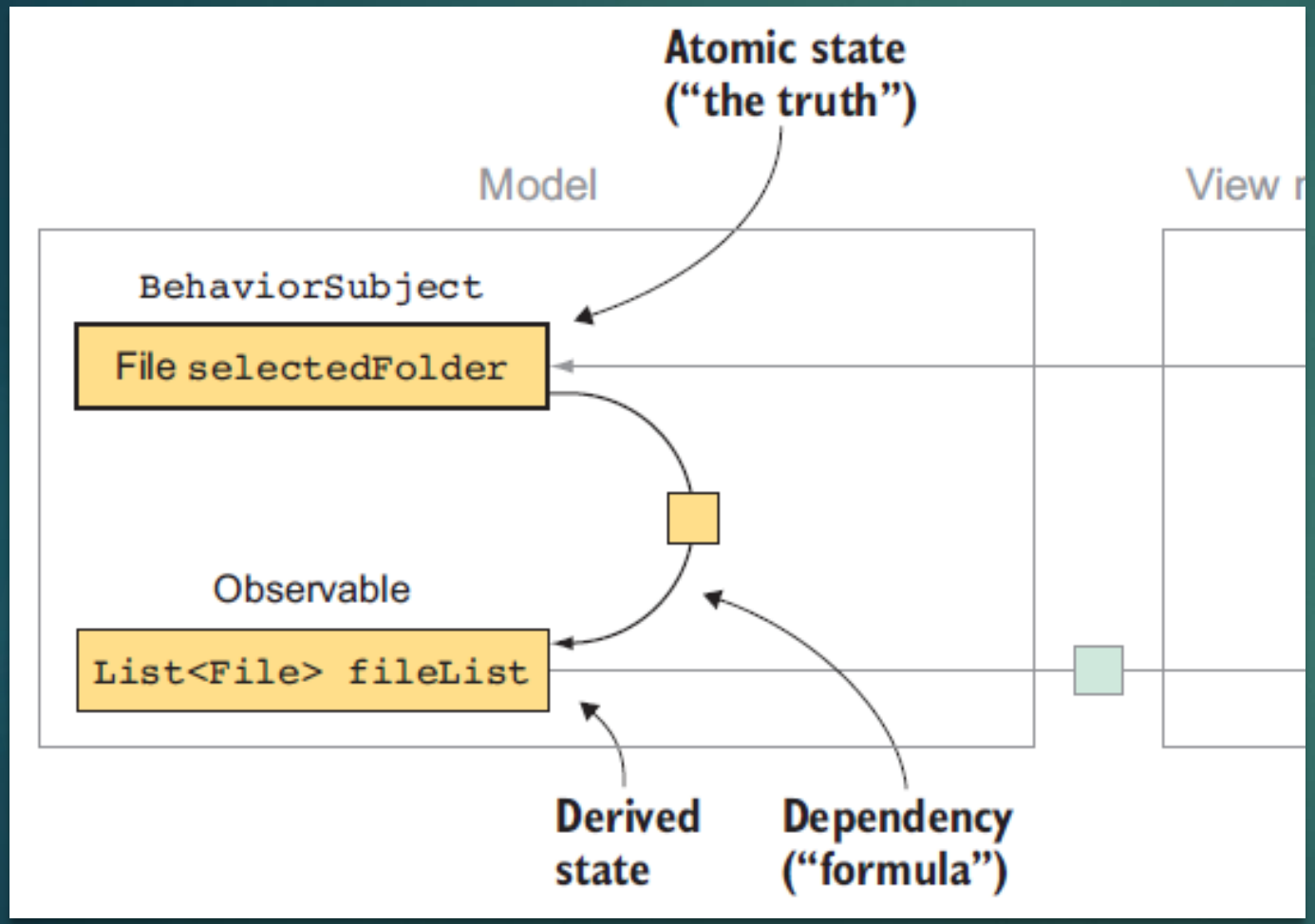
- The model is the Wikipedia of your app.



Persisting app state

- The question you need to ask is, what information is required to initialize the app? This will be the state you'll want to persist to reload the app.

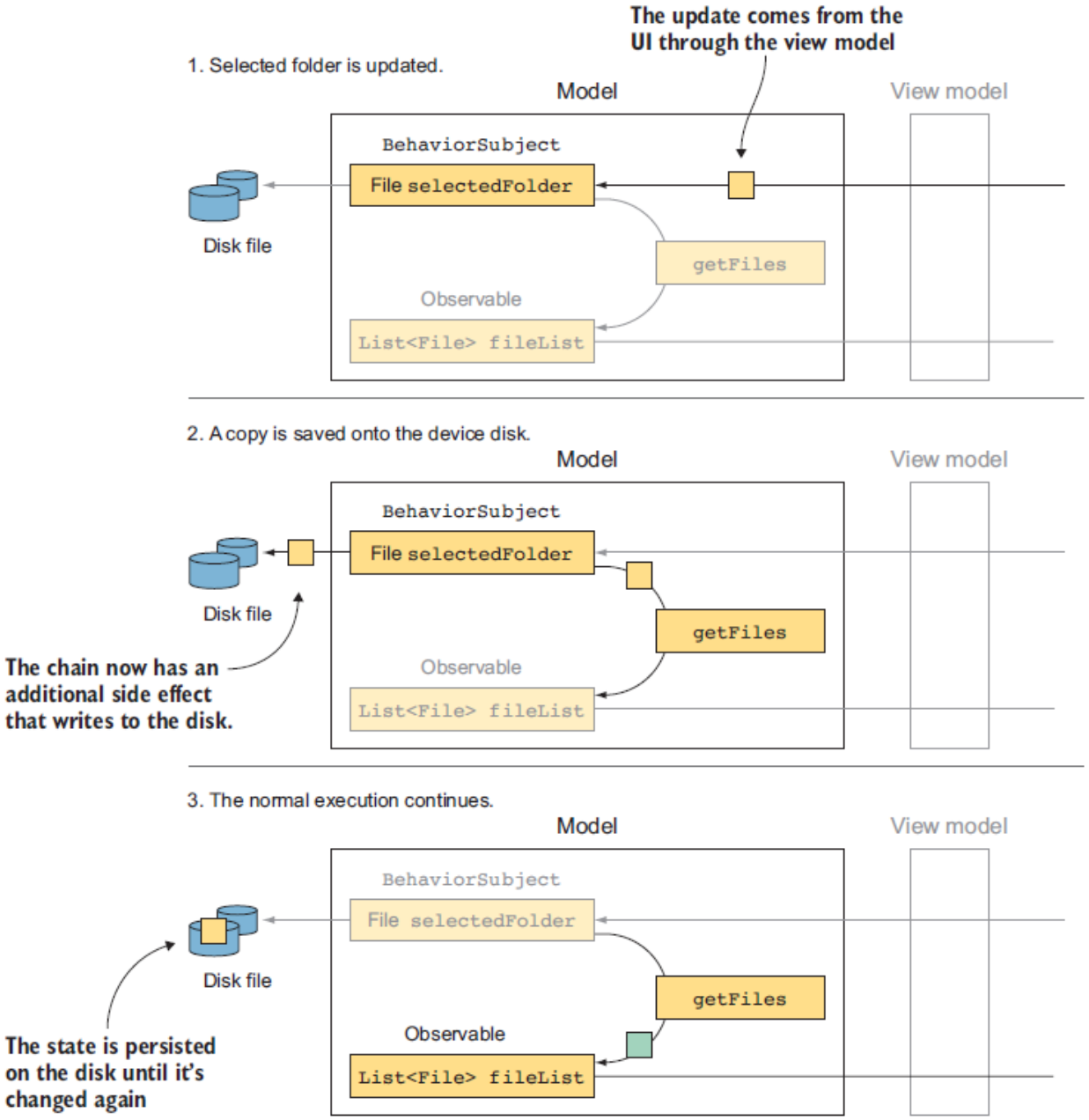




Atomic state

- You'll name this information that can't be calculated from anything else atomic state.
- The list of files in the selected folder isn't atomic state because you can always retrieve it again through the API.

Saving the model state



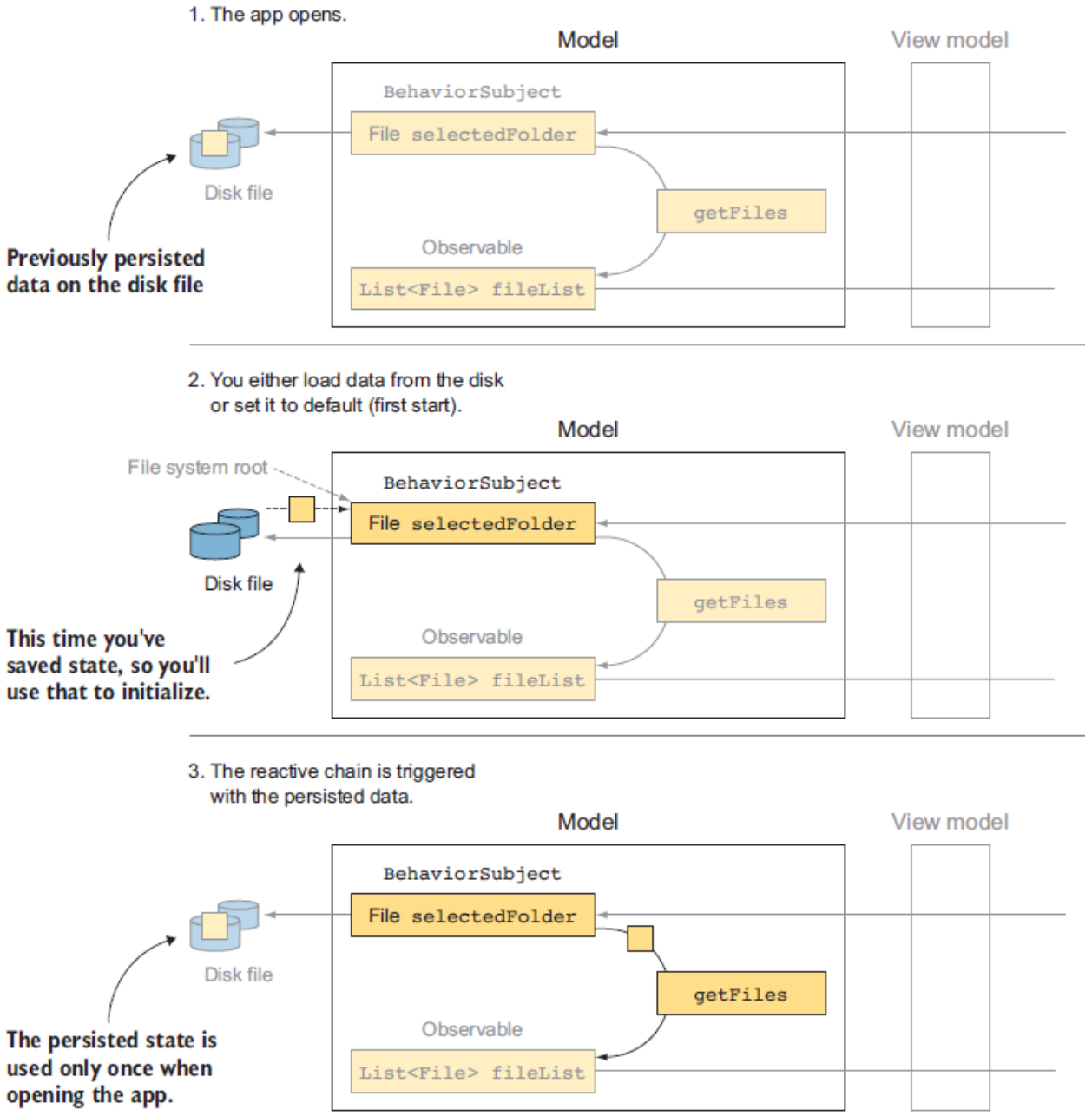
Code for saving the model state

FileBrowserViewModel.java

```
public FileBrowserModel(
    Function<File, Observable<List<File>>>
      getFiles,
    SharedPreferences sharedPreferences) {
  selectedFolder
    .observeOn(Schedulers.io())
    .subscribe(folder ->
      sharedPreferences.edit()
        .putString(
          SELECTED_FOLDER_KEY,
          folder.getAbsolutePath()
        )
        .commit());
...
```

SharedPreferences is given as a reference from outside. This could be changed into a function that writes to disk.

You use the edit command of SharedPreferences to open a writer. In the end, you commit the changes to affect the operation.



Loading the model state on startup

Code for loading the model state

With the added code, the beginning of the constructor first loads the state and then proceeds with the rest of the initialization.

This particular code will result in one extra write operation as the `BehaviorSubject` emits its value immediately, but because it doesn't cause a loop, you'll leave it as a further improvement.

FileBrowserModel.java

```
public FileBrowserModel(
    Function<File, Observable<List<File>>> getFiles,
    String defaultPath,
    SharedPreferences sharedPreferences) {

  // Load previously persisted value or use the default
  String persistedSelectedFolder Path = sharedPreferences
    .getString(SELECTED_FOLDER_KEY, defaultPath);

  File initialSelectedFolder =
    new File(persistedSelectedFolderPath);

  selectedFolder = BehaviorSubject
    .createDefault(initialSelectedFolder);
  ...
```

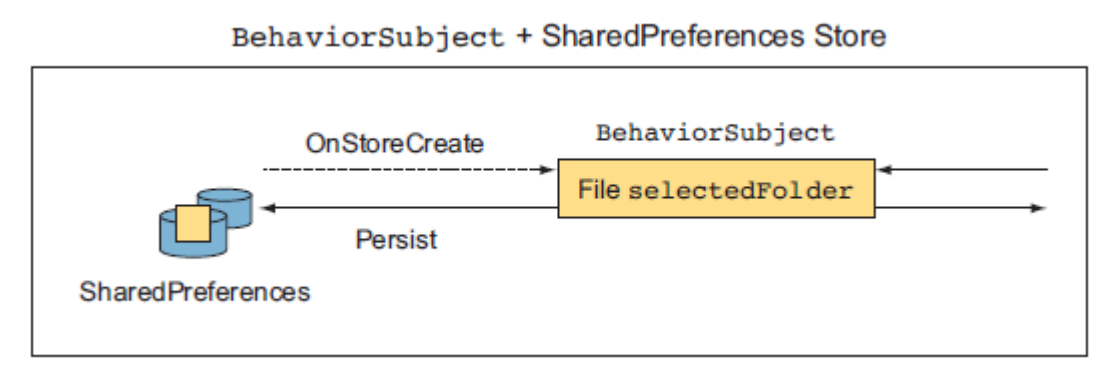
Provided as a dependency from the owner. In our case, it's the activity, but you could start using a dependency injection framework too.

A string for the default folder to use in case no persisted one is available

This is the part that loads the value or uses the default folder path given.

BehaviorSubjects and stores

- The simple one-value store
  - Perhaps the most reduced persisted store on Android can be constructed from a BehaviorSubject and a disk backing.



Simple SharedPreferencesStore

SharedPreferencesStore.Java

```
public class SharedPreferencesStore<T> {
  private final BehaviorSubject<T> subject;

  public SharedPreferencesStore(final String key,
      final String defaultValue,
      final SharedPreferences sharedPreferences,
      final Function<T, String> serialize,
      final Function<String, T> deserialize) {
    T initialValue = deserialize.apply(
      sharedPreferences.getString(key, defaultValue)
    );
    subject = BehaviorSubject
      .createDefault(initialValue);
    subject.subscribe(value ->
      sharedPreferences.edit()
        .putString(key, serialize.apply(value))
        .commit()
      );
  }

  public void put(T value) {
    subject.onNext(value);
  }

  public Observable<T> getStream() {
    return subject.hide();
  }
}
```

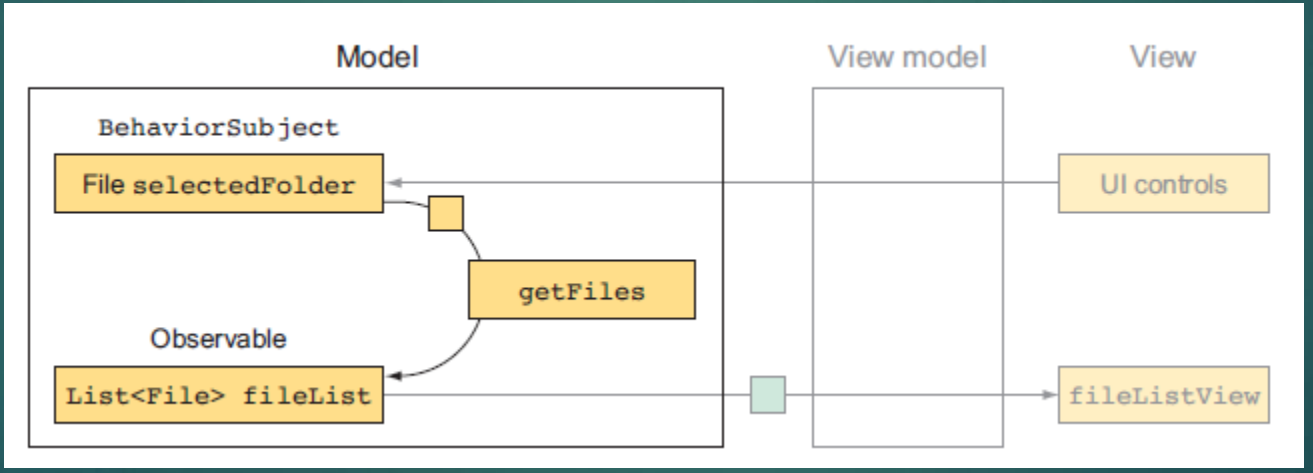
This is the subject you previously had directly in the model code. Now you'll replace it with an instance of a store.

Because you use strings as a format for persisting, you need to add functions to tell the store how to change from file to string, and vice versa.

You don't save the subscription at this time. Usually stores live so long that their internal subscriptions are released as the app is completely shut down.

Using SharedPreferencesStore

- The model you have will now internally use the store as a way to store the selectedFolder.
- Notice that to the outside, it looks exactly the same as before.



Using SharedPreferencesStore - codes

FileBrowserModel.java constructor

```
selectedFolderStore = new SharedPreferencesStore<>(
  SELECTED_FOLDER_KEY,
  defaultPath,
  sharedPreferences,
  file -> file.getAbsolutePath(),
  path -> new File(path)
);
```

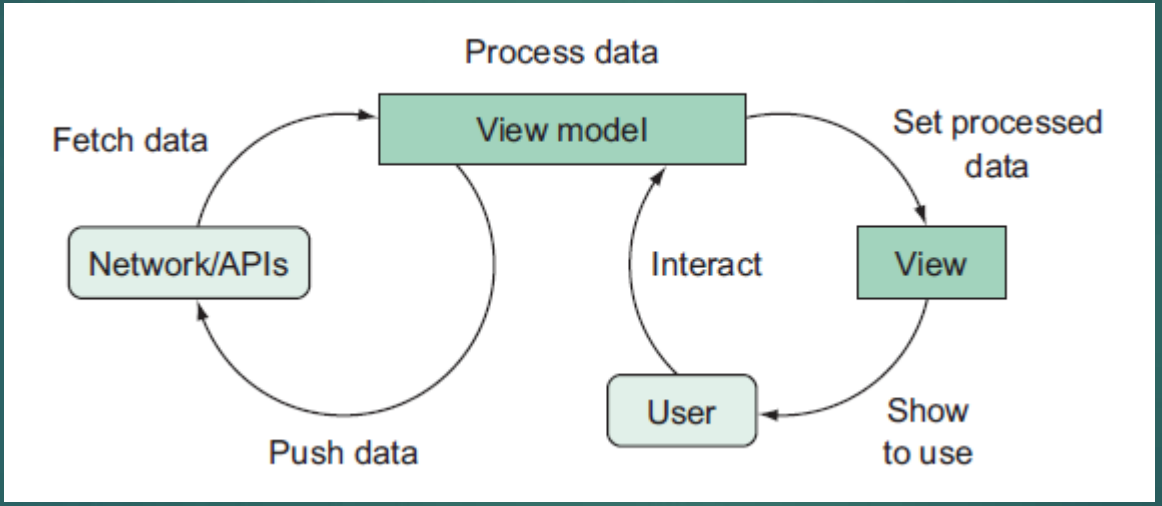
Arbitrary string that will be used as the filename for SharedPreferences

These two are the serializing and deserializing functions between strings and files.

Developing with View Models

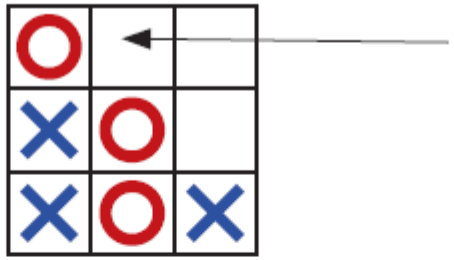
View models and the view

- The view layer is everything and anything that displays the processed data to the user.



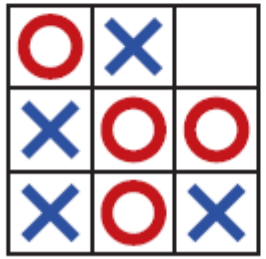
Example: Tic-tac-toe

Here's a game that has been played for three rounds. The players can place their icons in only the empty squares.



Because the circle always starts, in this particular game the circle is about to win.

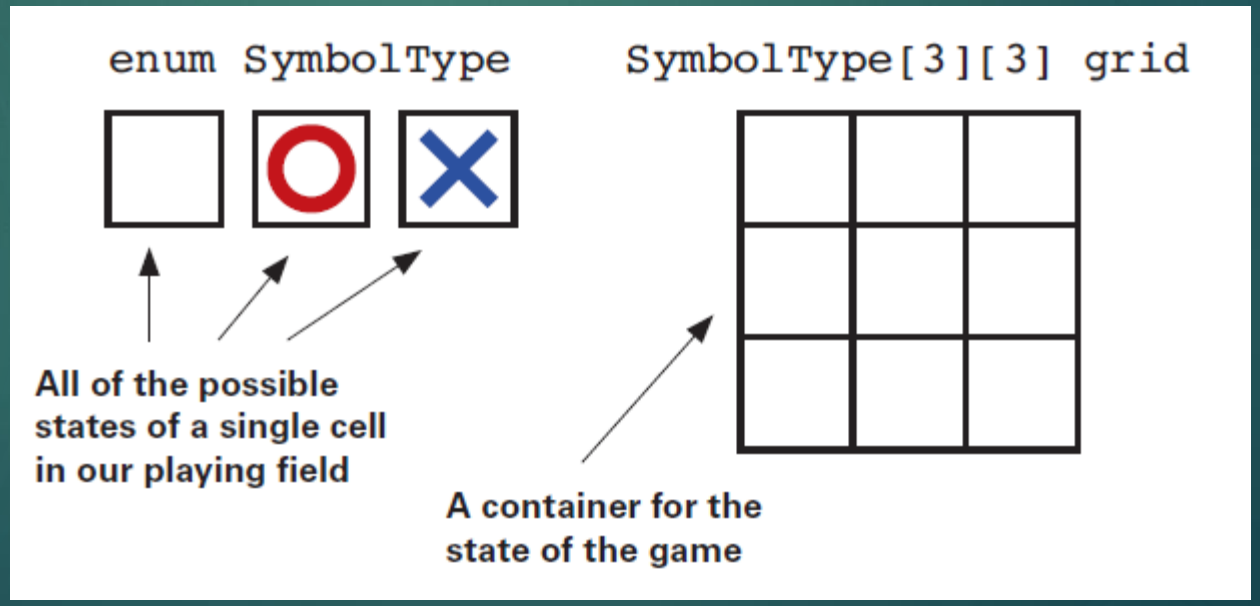
A tie



Circle wins

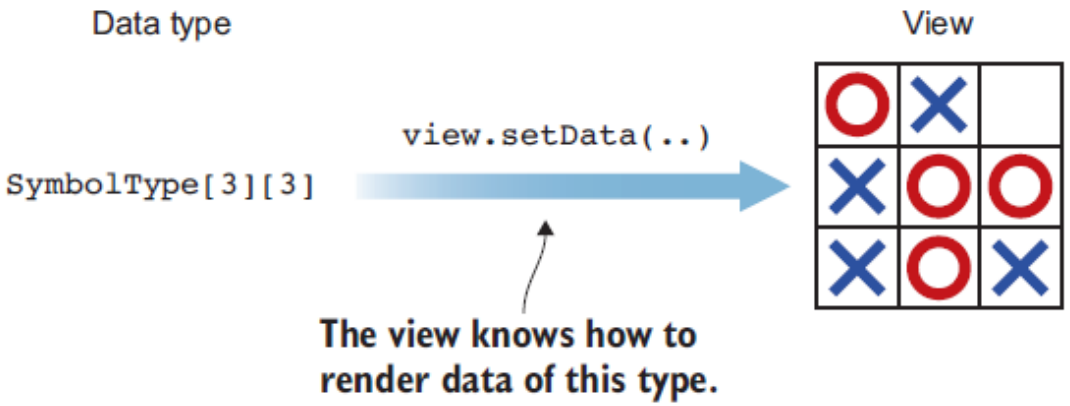
The different moves of tic-tac-toe

- In the game of tic-tac-toe we have a two-dimensional grid of three kinds of states: empty, a circle, or a cross.
- In the simplest form, this can be done with an enumerated type and a 3 × 3 array.



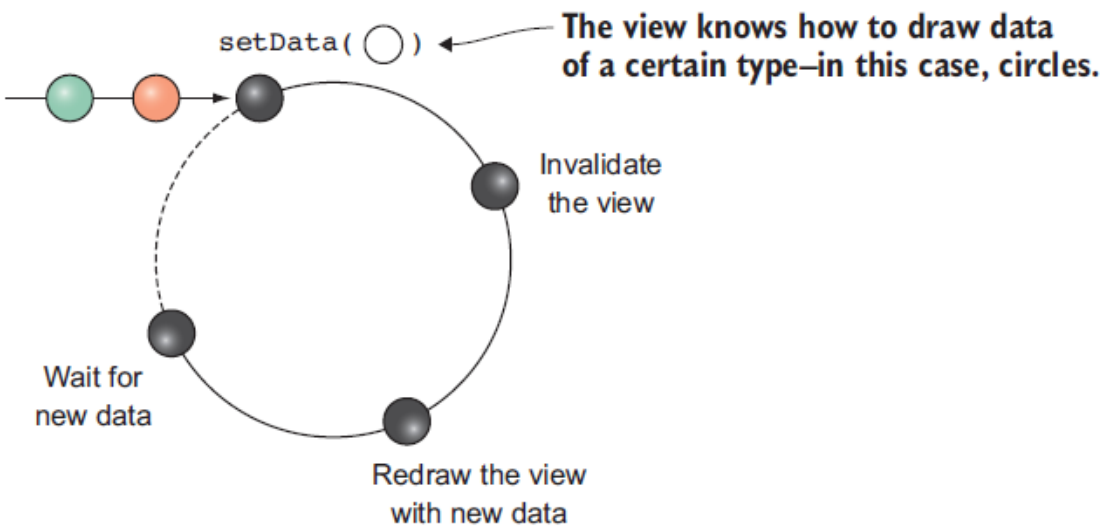
The different moves of tic-tac-toe

- All data at this point is of type SymbolType[3][3].
- With this you can draw the grid and the symbols.
- You'll create a view that knows how to draw data of this type. It'll have a setter setData(SymbolType[][] data) that accepts the data.
- As soon as new data is given, the view is redrawn to represent the new information.



Drawing the game grid

- **Create**
  - The owner creates the interface component.
  - The interface component is initialized with an empty grid.
- **Call setData() to trigger update**
  - setData invalidates the state of the view.
  - The redraw is scheduled.
  - The old graphics are cleared and updated with ones based on the new data.

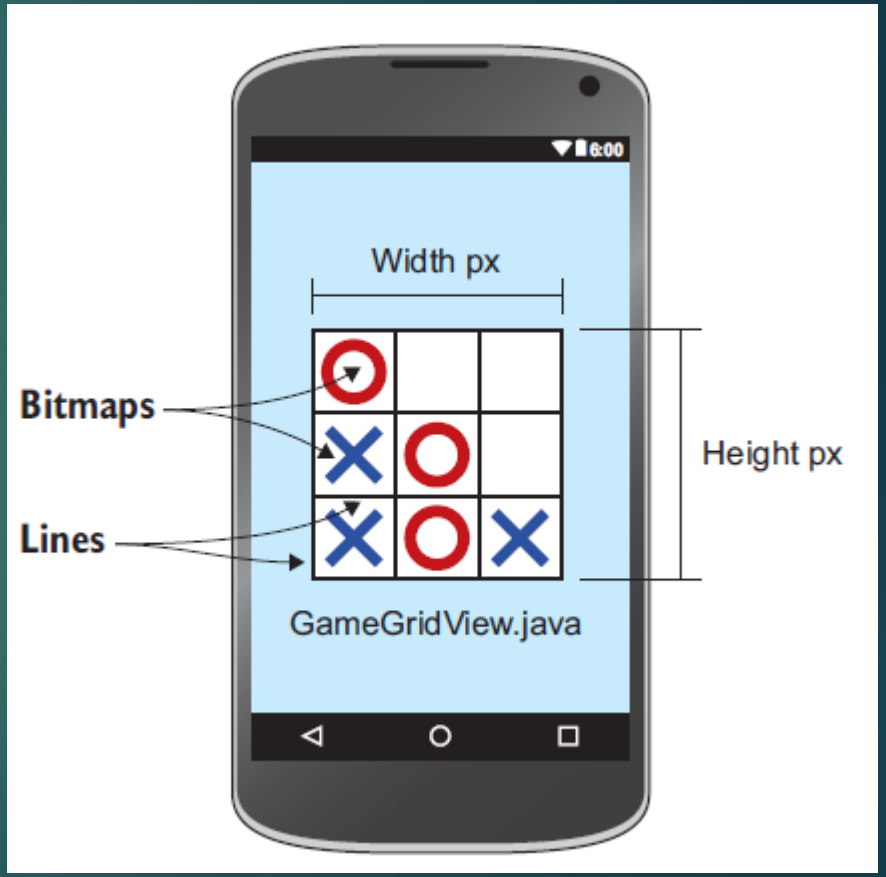


The draw function

- The custom view class
  - The first thing you'll do is create the new class.
  - Everything related only to drawing will be kept there.

GameGridView.java

```
public class GameGridView extends View {
  private GameSymbol[][] gameState;
  ...
```



The draw function

- Updating the data that the view presents
  - You need to add a way to tell your custom view that it has something new to show.

GameGridView.java

```
public void setData(SymbolType[][] gameState) {
  // Save data for drawing
  this.gameState = gameState;

  // Schedule redraw
  this.invalidate();
}
```

This is a member variable you created for the View class. Notice that it's just a holder for the data and shouldn't be exposed outside this class!

```
@Override
protected void onDraw(Canvas canvas) {
  // Clear the old drawings
  clearCanvas(canvas);

  // Draw background
  drawGridLines(canvas);

  // Draw symbols by looping through them
  for (int i = 0; i < 3; i++) {
    for (int n = 0; n < 3; n++) {
      Symbol symbol = this.data[i][n]
      if (symbol == Symbol.CIRCLE) {
        drawCircle(canvas, i, n);
      } else if (symbol == Symbol.CROSS) {
        drawCross(canvas, i, n);
      }
    }
  }
}
```

You're skipping some more straightforward code, but you can see it in the online code example.

This is a double loop that goes through your entire 3 × 3 array. Using magic numbers in the code instead of constants is bad, but you'll change that later.

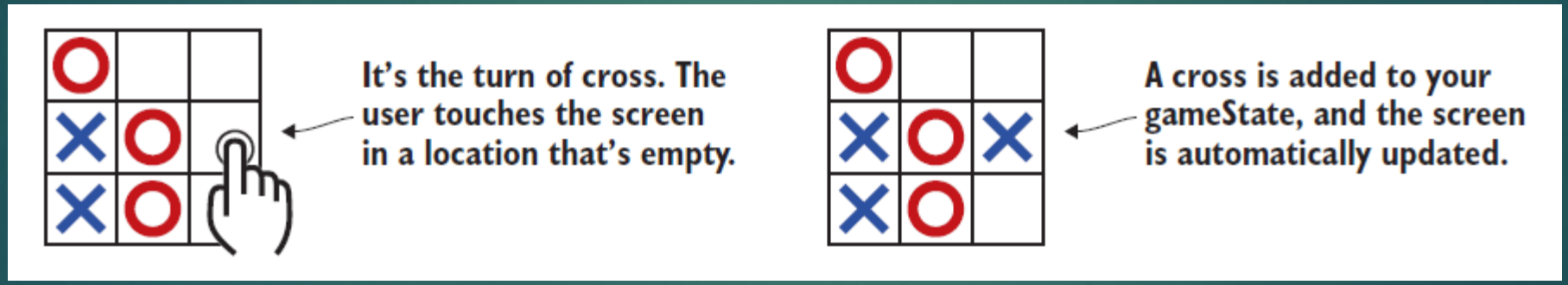
Trying out the view with hardcoded values

MainActivity onCreate

```
GameGridView gameGridView =
   (GameGridView) findViewById(R.id.grid_view);

gameGridView.setData(
   new GameSymbol[][] {
      new GameSymbol[] {
   GameSymbol.CIRCLE, GameSymbol.EMPTY, GameSymbol.EMPTY
      },
      new GameSymbol[] {
   GameSymbol.CIRCLE, GameSymbol.CROSS, GameSymbol.EMPTY
      },
      new GameSymbol[] {
   GameSymbol.CROSS, GameSymbol.EMPTY, GameSymbol.EMPTY
      }
   }
);
```

Making it interactive

- You're getting closer to the reactive part of the app.
- The interaction will be done with an Rx chain, but let's see what you plan to do.



Getting the touch events

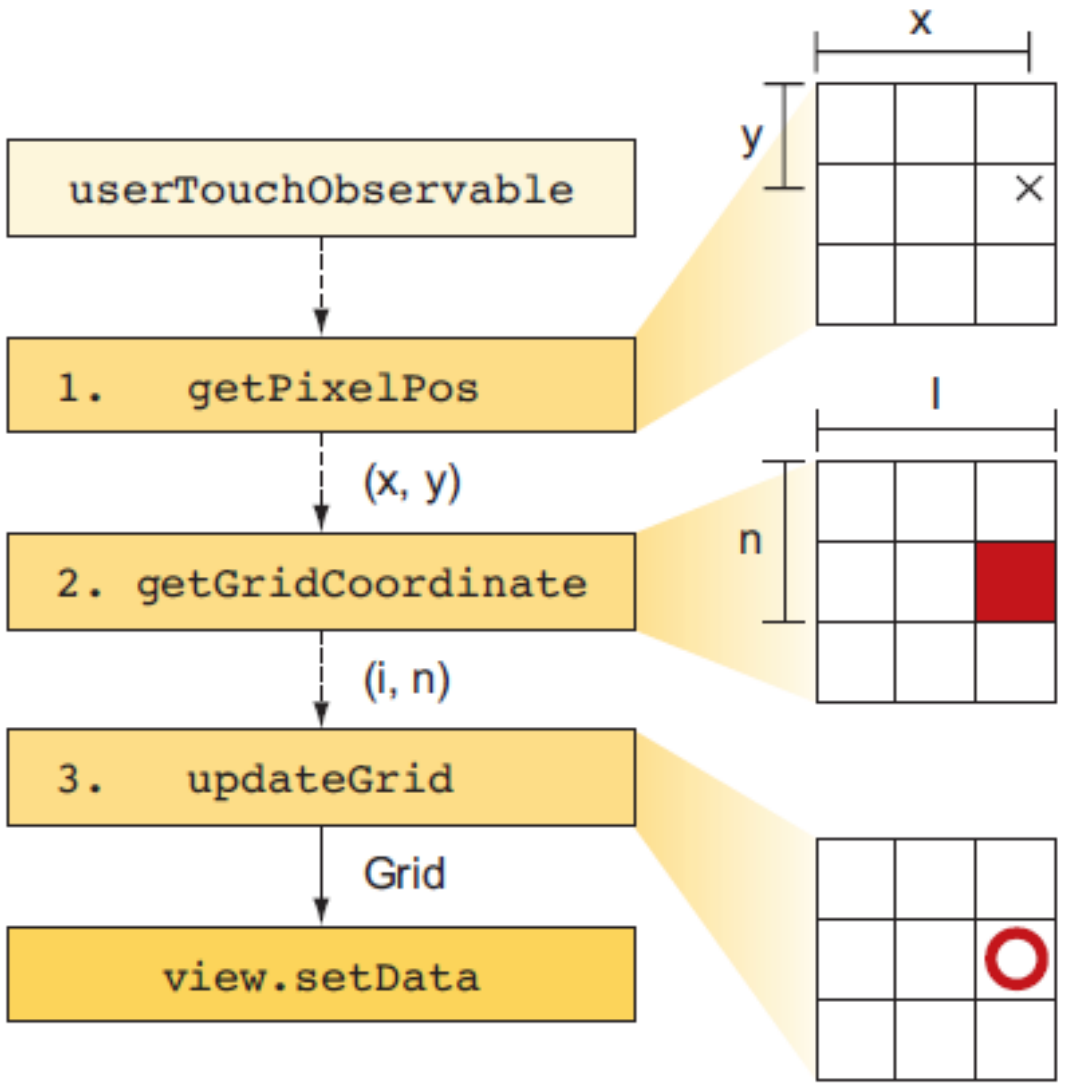
- You'll again use the RxBinding library to get a wrapper that gives you the touches on the view as an event observable.
- Normally, you'd register a listener for touch events, but this way, you can use the event processing capabilities of RxJava.

MainActivity.java onCreate

```
// Retrieve a reference to the created view
GameGridView gameGridView =
  (GameGridView) findViewById(R.id.grid_view);

// Get an observable with the RxBinding wrapper
Observable<MotionEvent> userTouchObservable =
  RxView.touches(gameGridView);
```

The reactive processing chain

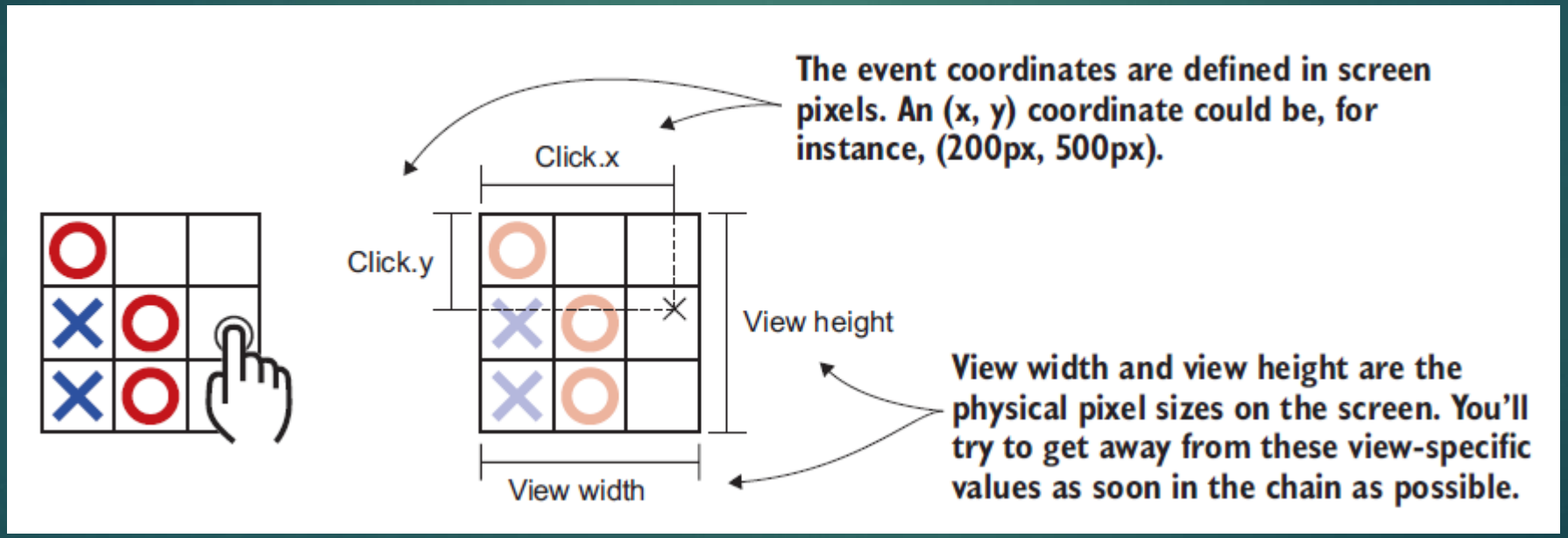
- When the user clicks on the playing grid, you want to insert the correct symbol into that location and then change the turn.
- To know which grid, tile the user clicked, you run the click event through a simple processing chain. The steps are roughly as follows:
  1. Start with the relative (x, y) coordinate of where the user clicked inside the grid view.
  2. Determine into which tile the click landed and emit this simplified coordinate. Instead of screen coordinates, these are indexes for the grid array you have.
  3. Update the grid by placing a new symbol in the location you determined.



The reactive processing chain

Grid coordinate resolving code

- To get back to the touch processing, in step 2 you have to identify which grid tile the user touched.



The algorithm

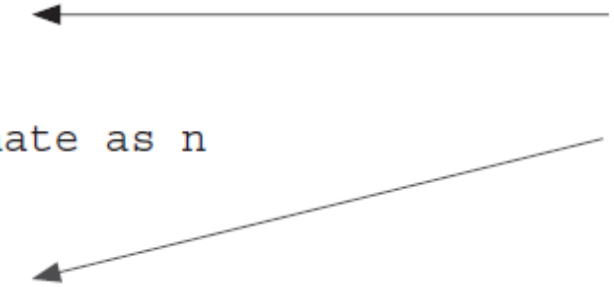
- For the horizontal grid position, you first divide the x coordinate of the click by the width of the whole view on the screen.
- You'll get a number between 0 and 1, 0.5 being in the middle.

1 You multiply the number from the first step by the number of tiles in the grid on the horizontal axis, which in this case is 3. Rounding down the number, you get the horizontal grid position.

2 What you'll have is a function that takes pixel coordinates and produces `GridPositions`. To distinguish these two you create a new data type that holds these values. In this case they range from (0, 0) to (2, 2). Indexing here starts from 0.

MainActivity getGridPosition

```
private static GridPosition getGridPosition(
        float touchX, float touchY,
        int viewWidthPixels, int viewHeightPixels,
        int gridWidth, int gridHeight) {

  // Horizontal GridPosition coordinate as i
  float rx = touchX /
    (float)(viewWidthPixels+1);
  int i = (int)(rx * gridWidth);

  // Vertical GridPosition coordinate as n
  float ry = touchY /
    (float)(viewHeightPixels+1);
  int n = (int)(ry * gridHeight);

  return new GridPosition(i, n);
}
```



You sometimes use i and n to make a distinction that these aren't pixel coordinates. But it's easier to remember that x is horizontal and y is vertical, so we won't stick to them all the time.

MainActivity onCreate

```
// Get the touches
Observable<GridPosition> userTouchEventObservable =
  RxView.touches(gridView, motionEvent -> true)
    .filter(ev ->
      ev.getAction() == MotionEvent.ACTION_UP);
```

Listen to the touch event

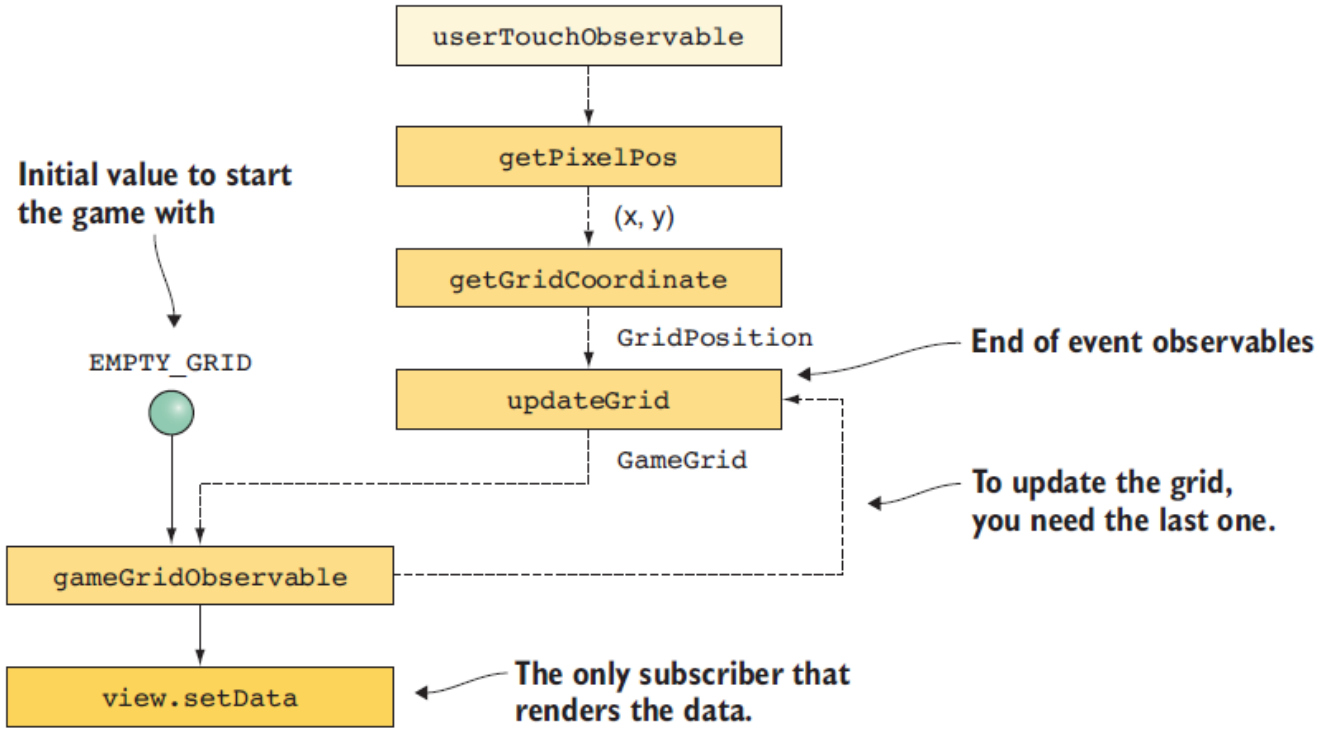
Get the grid position

- With this event observable, you can resolve a stream of GridPositions that allow you to later update the game grid.

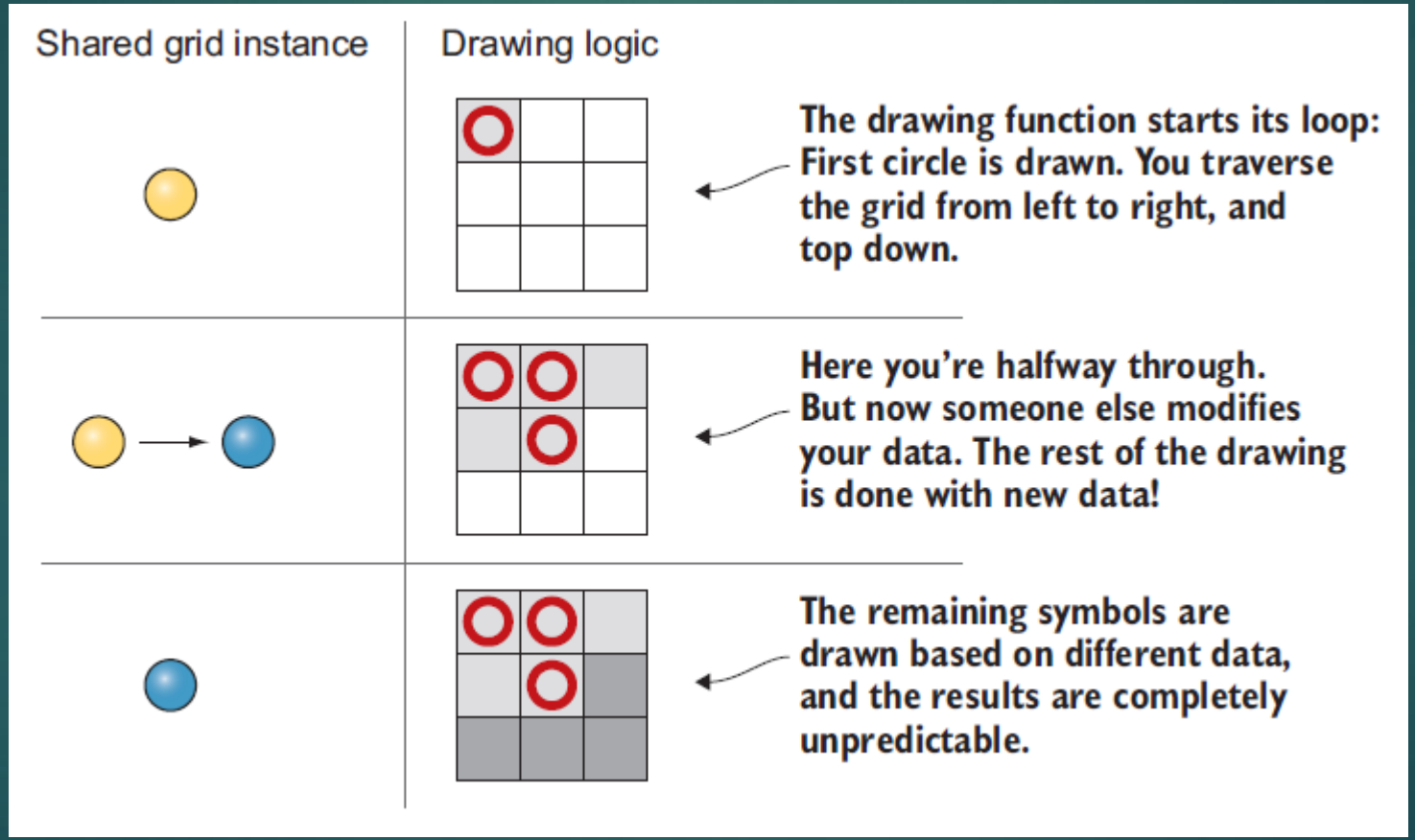
```
// Get the GridPosition from the pixel coordinate
Observable<GridPosition> gridPositionEventObservable =
  userTouchEventObservable
    .map(ev ->
      getGridPosition(
        ev.getX(), ev.getY(),
        gameGridView.getWidth(),
        gameGridView.getHeight(),
        GRID_WIDTH, GRID_HEIGHT
      )
    );
```

The extended graph structure

- To complete the graph on the previous page, you can observe that to update the grid, you need the last grid as well.
- Otherwise, you'll always be making the first move and forgetting the previous ones.

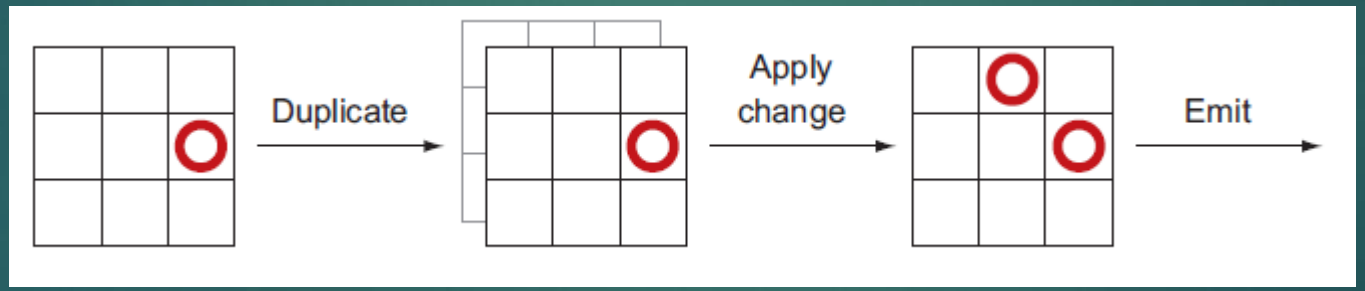


Immutable data and the game grid



Making a copy of the grid

- Java provides many ways to copy an array, but for our simple purposes, you can copy each row of the raw array with System.arraycopy, which is very efficient.
- It takes the source array, position in the source, destination, the position in the destination, as well as the total number of items to copy.



GameGrid type and functional setters

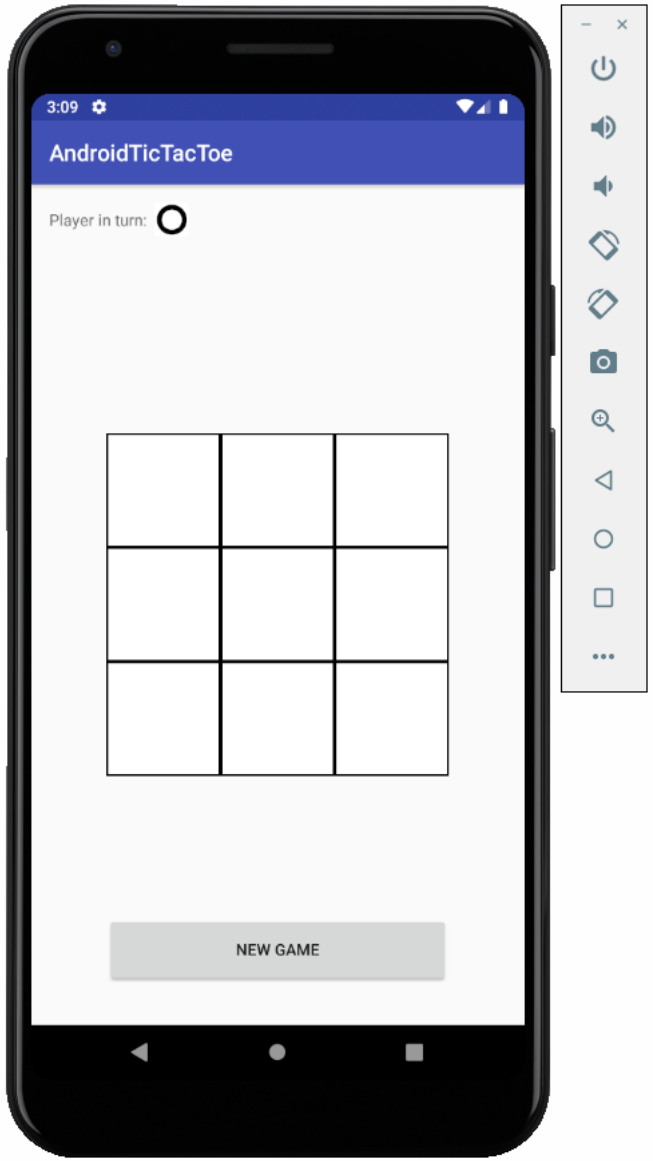
- What you'll create first is a class to hold the entire game grid.
- Previously, you had just a two-dimensional array GridSymbol[][], but to encapsulate more functionality, you'll change it into a type:

```
class GameGrid {
  GameSymbol getSymbolAt(int x, int y)
  GameGrid setSymbolAt(
    GameSymbol symbol, int x, int y)
}
```

Pay attention to this setter. It isn't the traditional kind of Java setter, but made in a functional style. It doesn't modify the original instance.

```
public GameGrid setSymbolAt(
   GameSymbol symbol, int i, int n) {
    GameGrid copy = this.copy();
    copy.grid[i][n] = symbol;
    return copy;
}
```

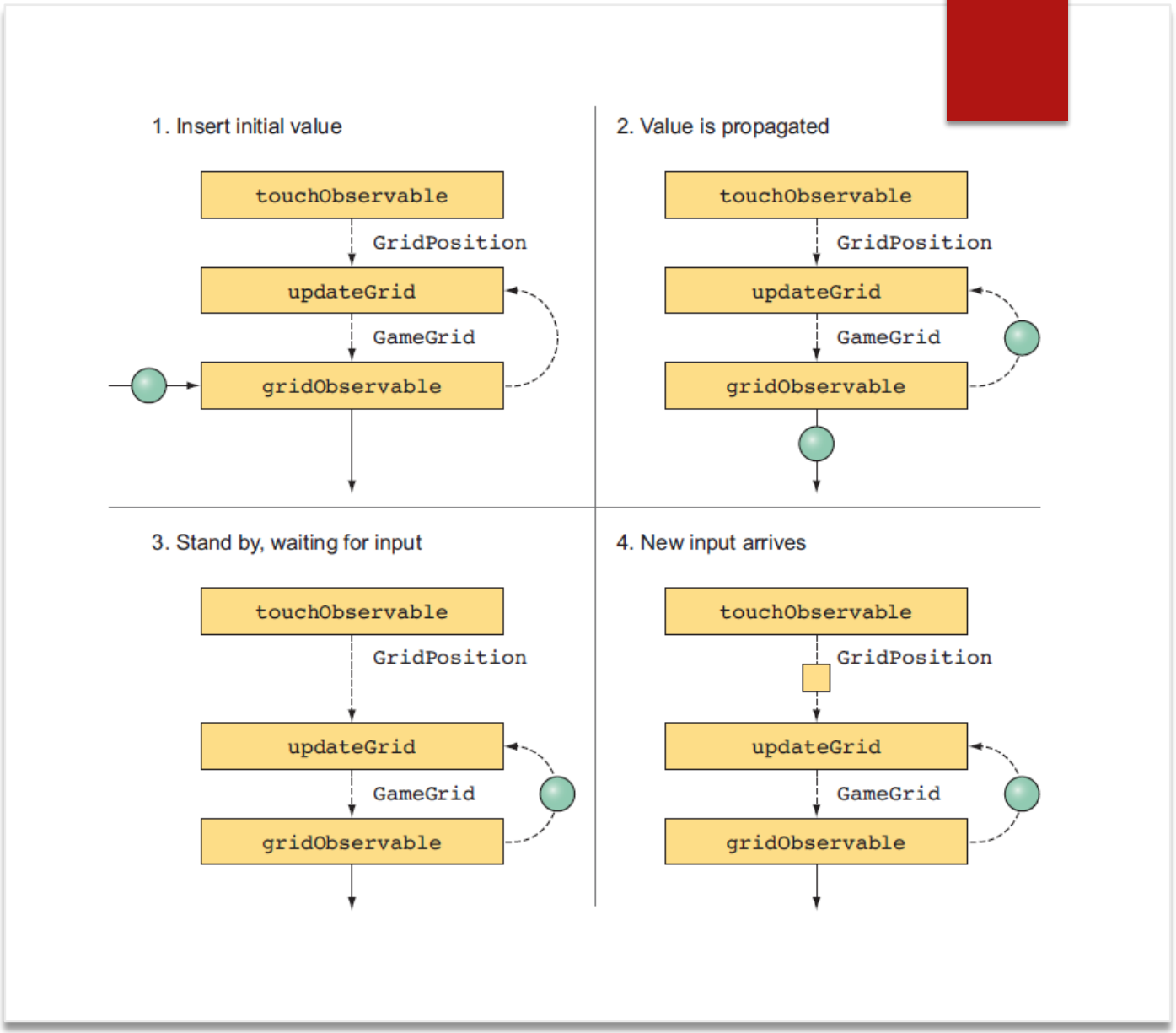
GameGrid type and functional setters



Putting it all together

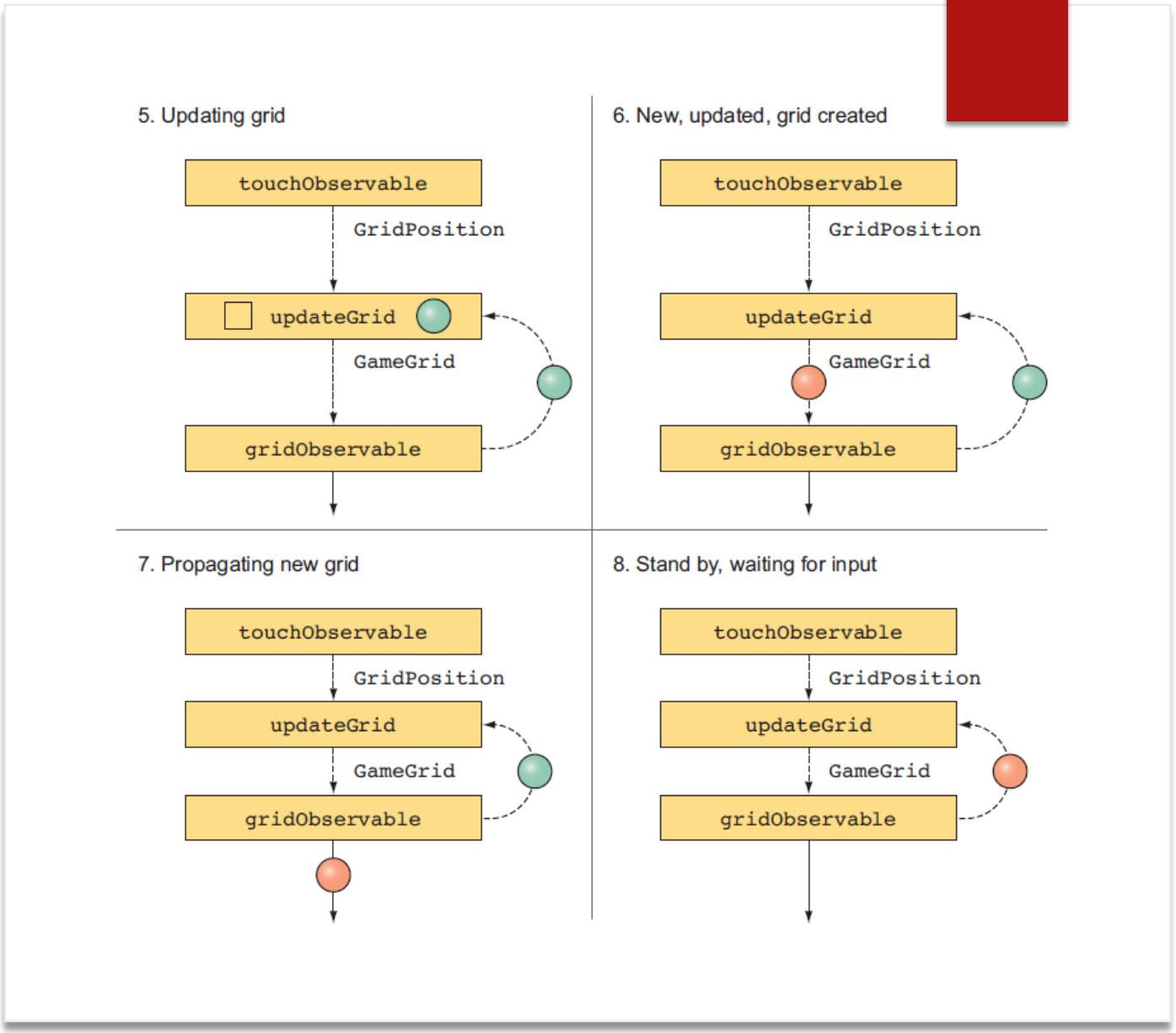
Cyclic graphs with .withLatestFrom

- In some cases, you need the last values from two observables to calculate a third one, but you want only one of them to trigger the chain below.



Cyclic graphs with .withLatestFrom

- Step 4 is where the .withLatestFrom function is doing its thing.
- It's a lot like combineLatest, but it's triggered on only new values from the primary observable.
- In this case, the triggering one is the observable that emits the GridCoordinates calculated from the touch events.



Cyclic graphs with .withLatestFrom

- You can think about it like this: you want to update the grid with the touch events only when the user is touching - not whenever you have a GameGrid for some other reason (also, in this case, that would create an infinite loop).

```
touchesOnGrid
  .withLatestFrom(gameGridSubject,
    (gridCoordinate, gameGrid) ->
      gameGrid.setSymbol(
        gridCoordinate, Symbol.CIRCLE)
  )
```

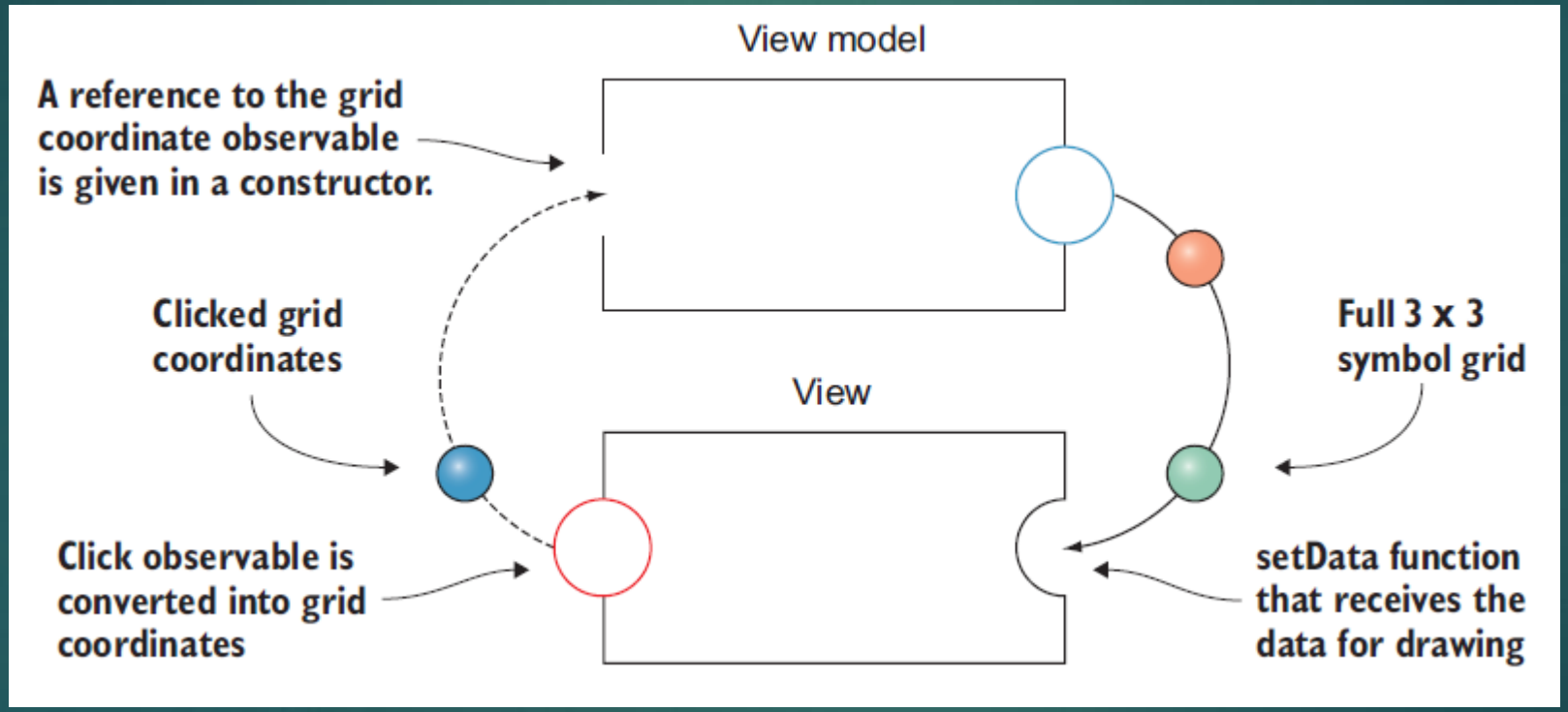
This lambda function is the combine function that's executed on all new values of the gridCoordinate.

Log the calculated positions

- Finally, until you have the grid updating function ready, you can see whether it works by logging the events. You'd expect values between (0, 0) and (2, 2).

```
// For starters just log the results
gridPositionEventObservable
  .subscribe(gridPosition ->
    Log.d(TAG, gridPosition.toString()));
```

Wrapping the logic into a view model



View model code

GameViewModel.java

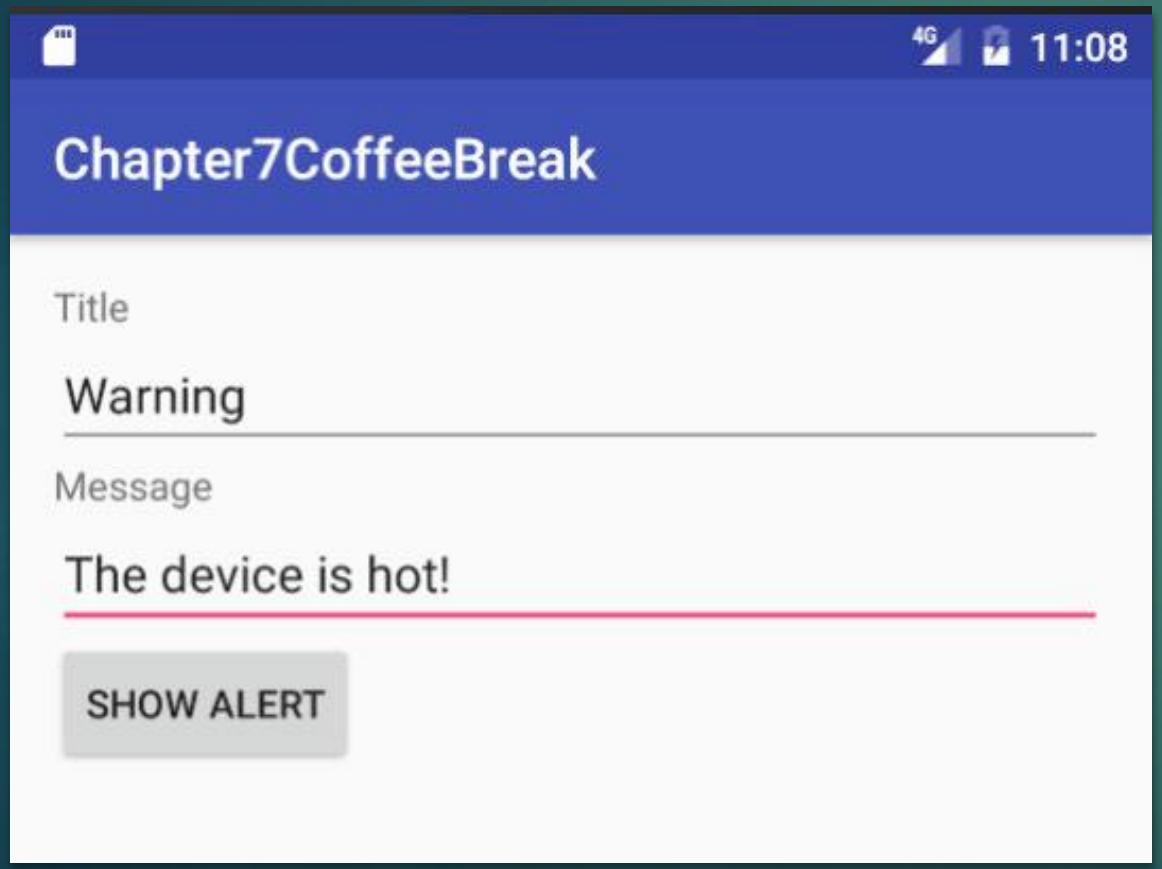
```
public class GameViewModel {
  private final CompositeDisposable subscriptions =
    new CompositeDisposable();

  private final BehaviorSubject<GameGrid>
    gameGridSubject = BehaviorSubject.create();

  private final Observable<GridPosition>
    touchEventObservable;

  public GameViewModel(
    Observable<GridPosition> touchEventObservable) {
    this.touchEventObservable = touchEventObservable;
  }

  public Observable<GameGrid> getGameGrid() {
    return gameGridSubject.hide();
  }
```

You'll hold the subscriptions you create here. They can then be released with the container lifecycle.

The input is an observable that gives events indicating which GridPosition was clicked.

The output is a behavior that returns the latest GameGrid. It'll be connected to the view that can draw it.

Coffee break



- You'll create a little app that can show alerts (dialogs) based on the input written in the text fields.
- You'll define the inputs as Observables, and it's your job to combine them to produce a dialog box.
- Keep in mind which observables emit events and which ones represent permanent states.