FP Seminar » March 28, 2017 » Mike Parker

An Introduction to Functional Reactive Programming

What is Functional Reactive Programming?

- If functional programming and the observer pattern had a 👶
- ReactiveX is a not-so-formal specification (http://reactivex.io/)
- Implementations include:
  - ReactiveCocoa ★ Star 16,907
  - RxJava ★ Star 22,683
  - RxJS ★ Star 15,192

Theory

What is the observer pattern?

```
public interface ConnectivityObserver {
  void connectivityChanged(boolean isConnected);
}
```

What is the observer pattern?

```
public class ConnectivityMonitor {
  private final Set<ConnectivityObserver> observers = new HashSet<>();

  public void addObserver(ConnectivityObserver observer) {
    observers.add(observer);
  }
  public void removeObserver(ConnectivityObserver observer) {
    observers.remove(observer);
  }
  ...
}
```

What is the observer pattern?

```
public class ConnectivityMonitor {
  ...
  private void notifyConnectivityObservers(boolean isConnected) {
    observers.stream().forEach(
        observer -> observer.connectivityChanged(isConnected));
  }
}
```

Observable

- `Observable<T>` is something that emits events of type `T`
- Also has a built-in lifecycle.
- Emits 0 or more events of type T to its Observers, and then...
  - Completes normally
  - Completes with an error

Observer

```
public interface Observer<T> {
  void onNext(T value);
  void onCompleted();
  void onError(Throwable exception);
}
```

A toy example

```
Observable<Integer> observable =
    Observable.just(1, 2, 3, 4, 5);
// Creates an Observable that emits 1, 2, 3, 4, 5
// and then completes

observable.subscribe(value -> System.out.println(value));
// Creates an Observer with the given method as onNext
// and prints 1, 2, 3, 4, 5
```

Subject: Bridging Imperative to Rx

```
Subject<Integer> subject = PublishSubject.create();
subject.subscribe(value -> System.out.println(value));

subject.onNext(1);
subject.onNext(2);
subject.onNext(3);
subject.onCompleted();
```

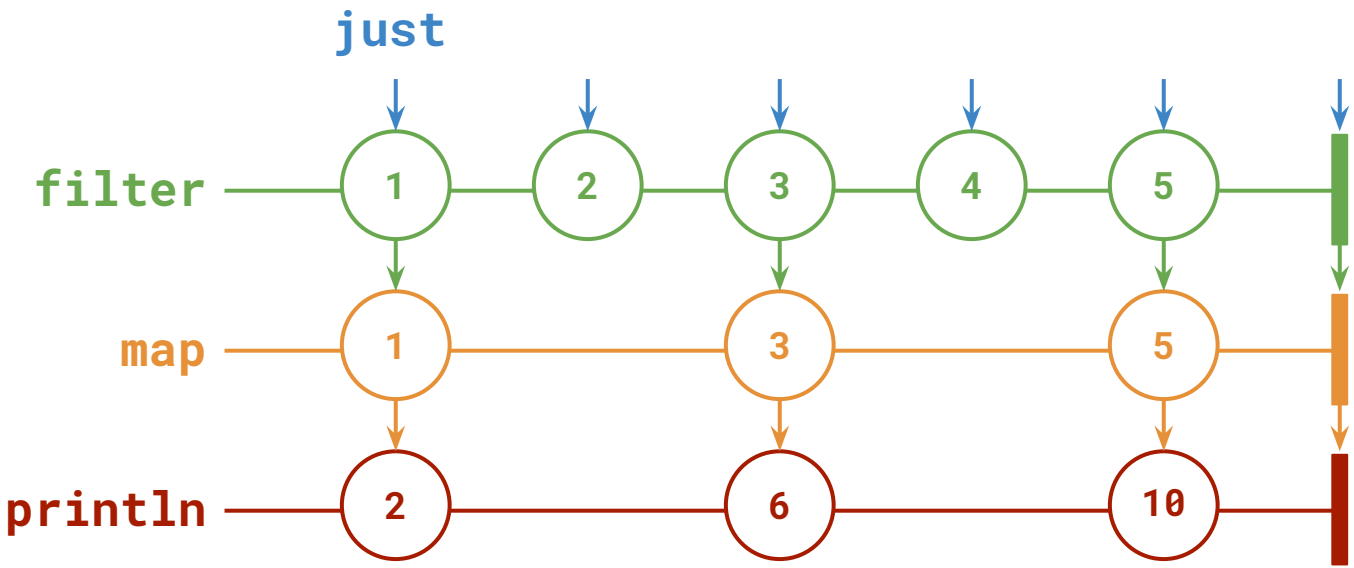
A slightly-less-toy example

```
Observable<Integer> o1 = Observable.just(1, 2, 3, 4, 5);

Observable<Integer> o2 = o1.filter(value -> (value % 2) == 1);
// o2 emits 1, 3, 5 to next Observable

Observable<Integer> o3 = o2.map(value -> 2 * value);
// o3 emits 2, 6, 10 to next Observable

o3.subscribe(value -> System.out.println(value));
```

A slightly-less-toy example

```
Observable.just(1, 2, 3, 4, 5)
    .filter(value -> (value % 2) == 1)
    .map(value -> 2 * value)
    .subscribe(value -> System.out.println(value));
```

Marble diagrams



The pattern

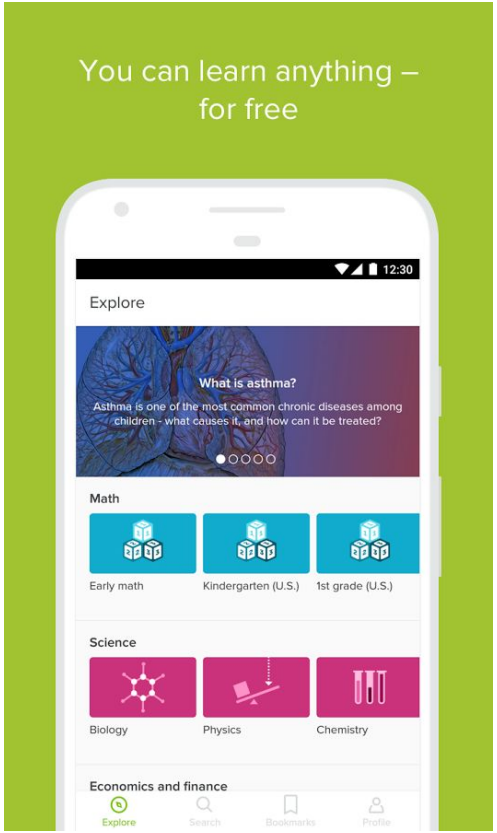
- The most *upstream* `Observable` may do *work* when subscribed to and in turn emits *values*
- We then chain *operators* that
  - filter values
  - transform values
  - combine values
  - aggregate, debounce, thread-hop, recover errors, etc. on values
- Finally an `Observer` subscribes to the most *downstream* `Observable` to perform an *action*

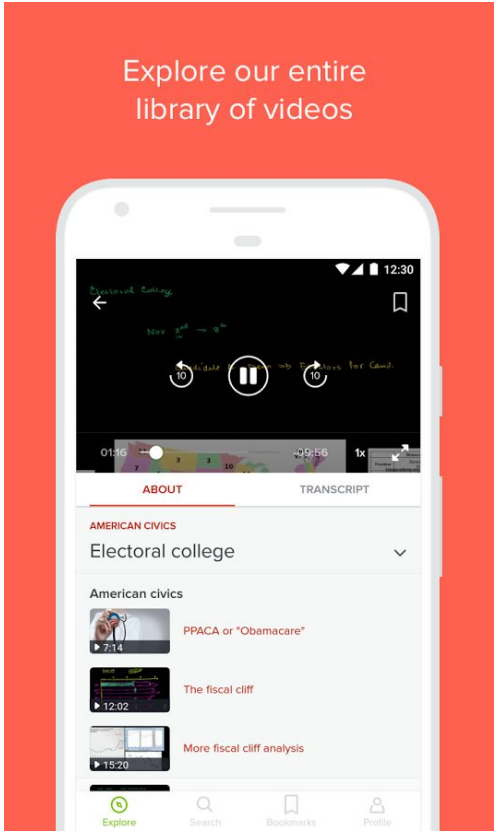
Modeling with Observables

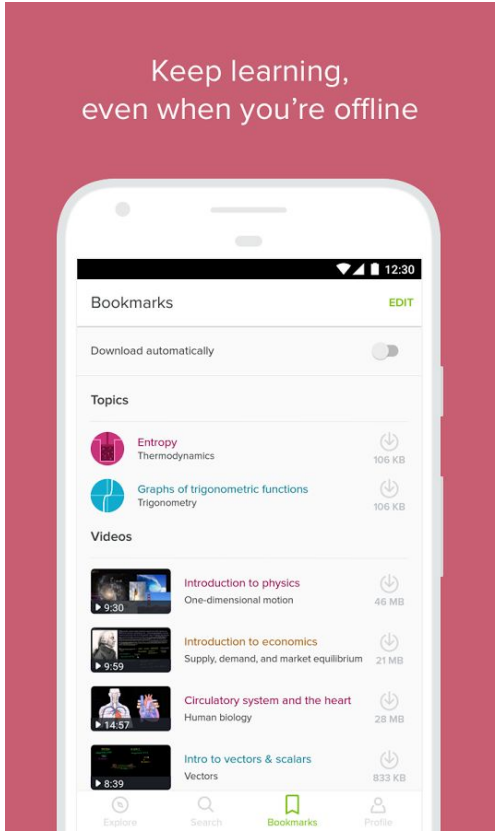
- **Adjusting a volume slider**
  - `Observable<Integer>` **emits 0 or more values and then completes**
- **Making an HTTP request**
  - `Observable<HttpResponse>` **emits 0 or 1 value and then completes**
  - **Also called a** `Single`
- **Writing a string to a file**
  - `Observable<Void>` **emits 0 values and then completes**
  - **Also called a** `Competable`

Practice

Previously at Khan Academy...

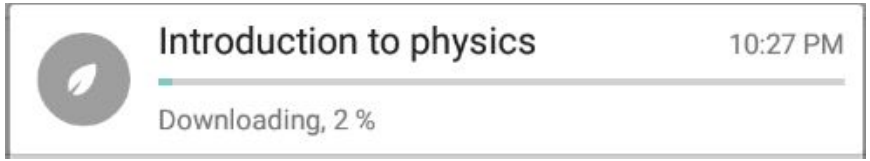






Download Notifications

Download notifications



- Can download individual videos or entire tutorials
- Progress displayed in the notification bar
- Should not update too quickly, except when completed

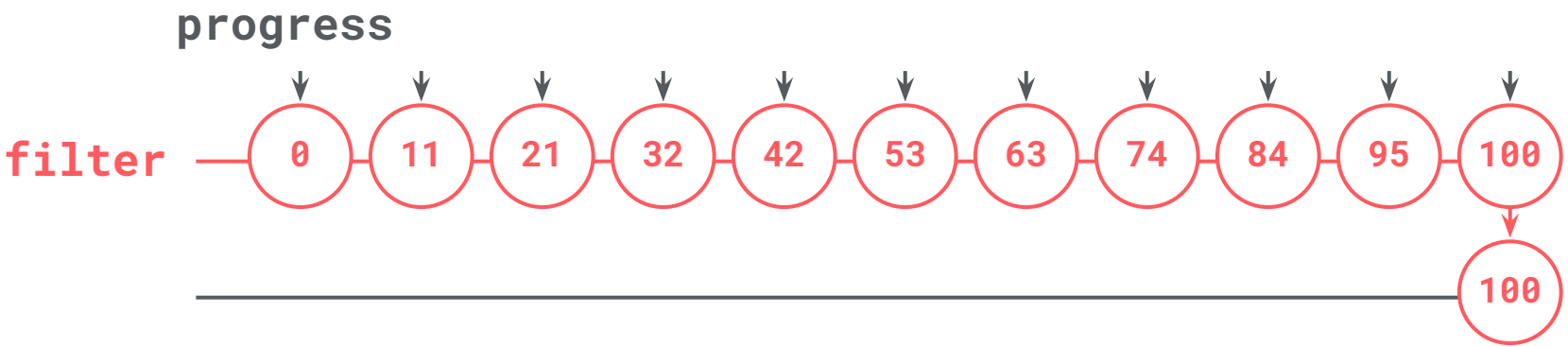
Download progress

```
public final class DownloadProgress {
  private final int bytesDownloaded;
  private final int percentComplete;
  ...

  public boolean isComplete() {
    return percentComplete == 100;
  }
}
```

Step 1: filtering the completed value

```
Observable<DownloadProgress> completeObservable =
    downloadProgressObservable
        .filter(progress -> progress.isComplete());
```

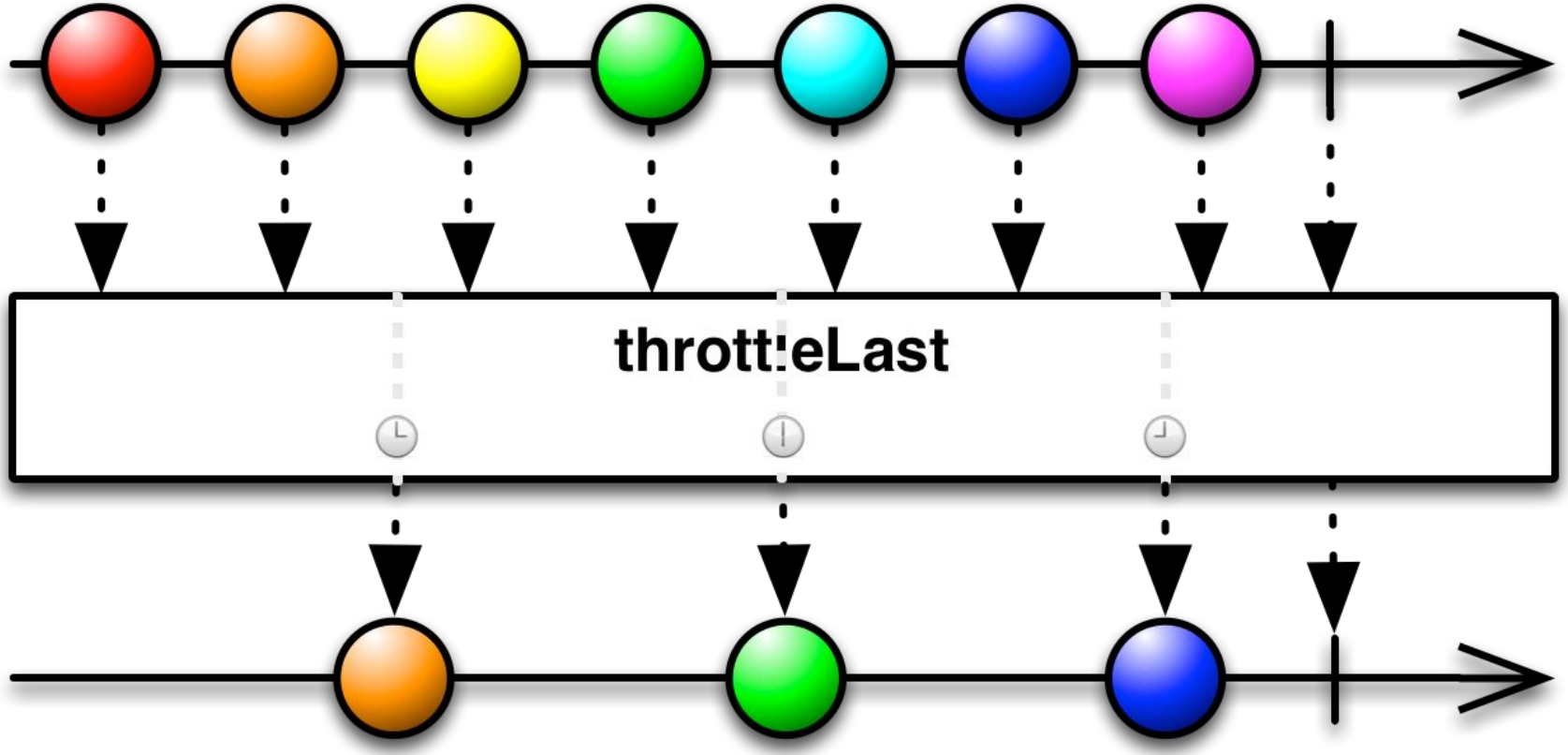
Step 1: filtering the completed value



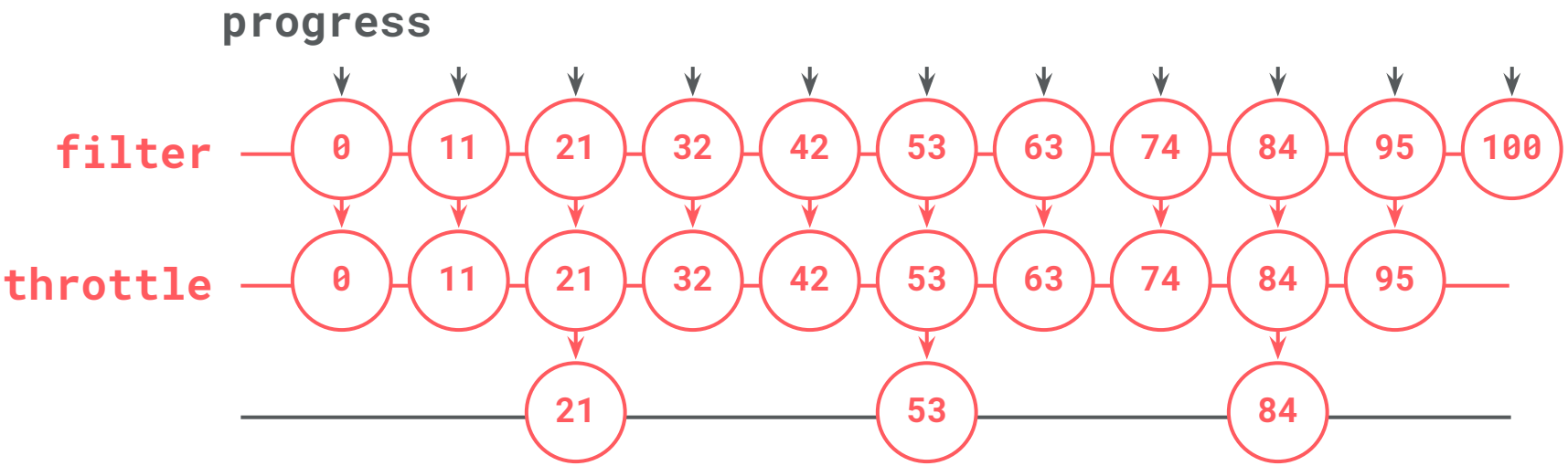
Step 2: throttling the incomplete values

```
Observable<DownloadProgress> incompleteObservable =
    downloadProgressObservable
        .filter(progress -> !progress.isComplete())
        .throttleLast(1L, TimeUnit.SECONDS);
```

Step 2: throttleLast



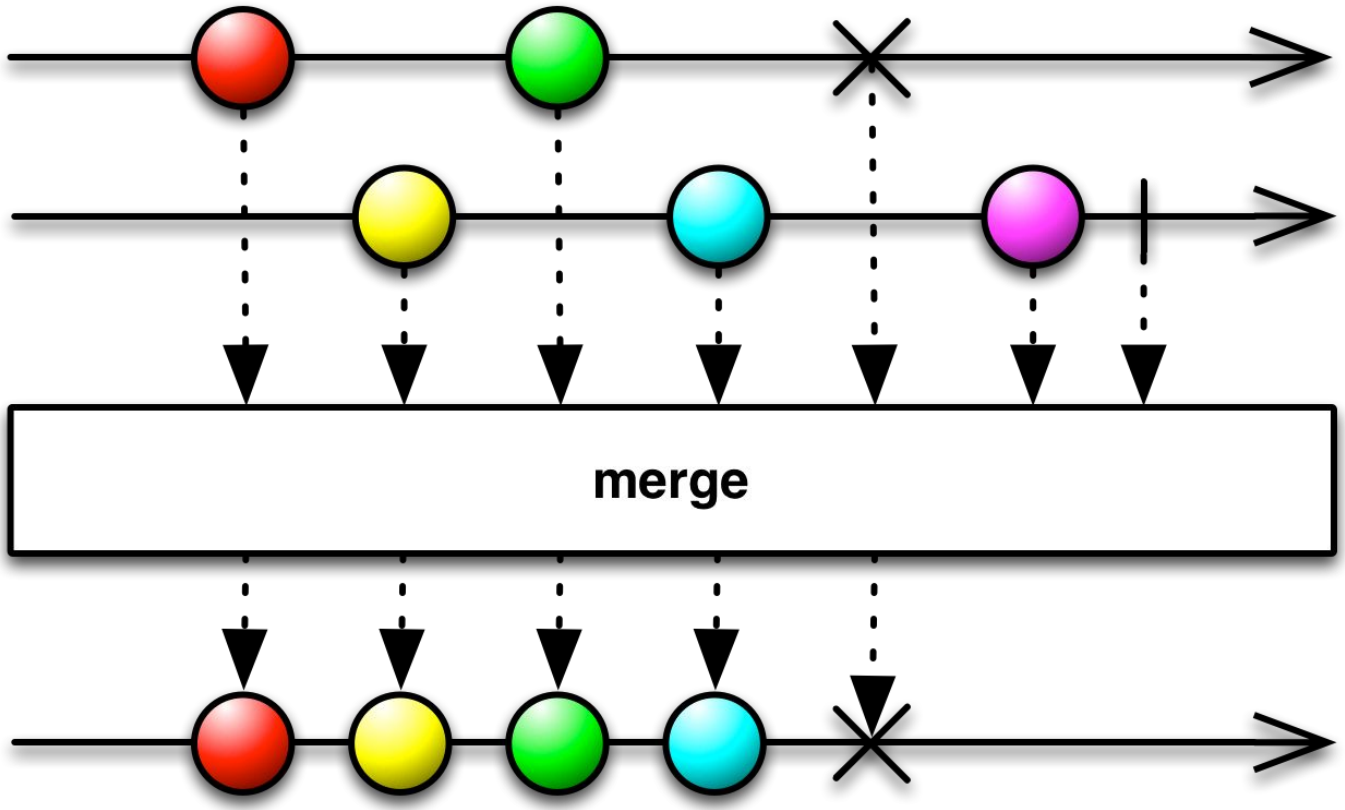
Step 2: throttling the incomplete values



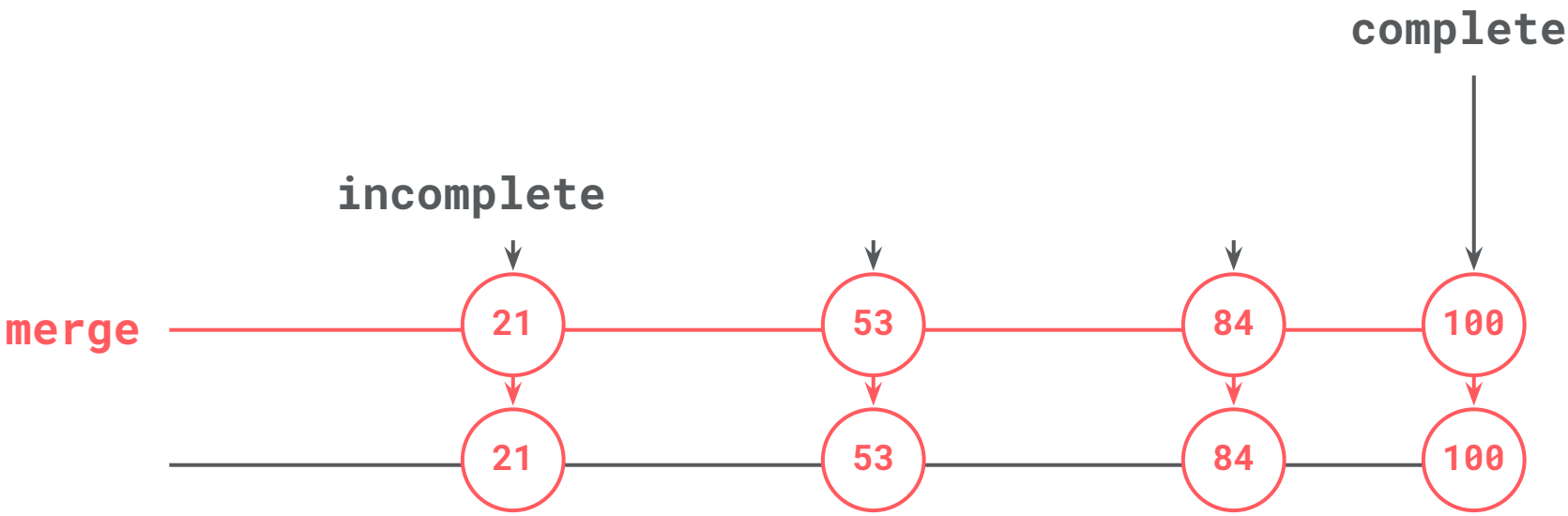
Step 3: merge

```
Observable<DownloadProgress> throttledObservable =
    Observable.merge(
        incompleteObservable,
        completeObservable
    );
```

Step 3: merge

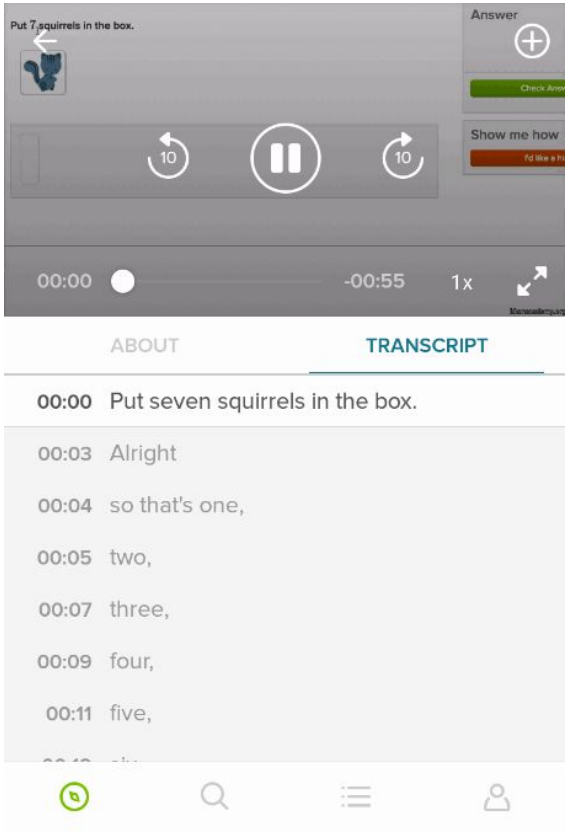


Step 3: merge



Video Transcripts

Video player



- **Video player for over 6,500 videos**
- **As the video plays, highlights the corresponding line of the transcript**

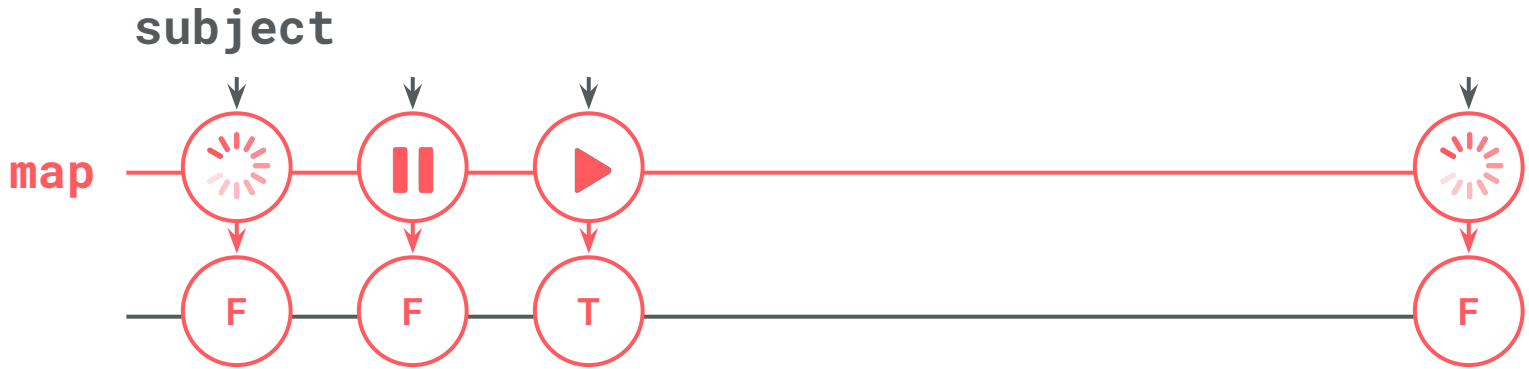
The imperative solution

- A listener on the video player for the various states
- A recurring timer that is part of the video player state
- When the video player changes state:
  - If the video started, start the timer
  - If the video stopped, stop the timer
- When the timer executes:
  - Read the current time from the video player
  - Scroll to the subtitle at that time
  - Schedule timer to execute again 250ms later

Step 1: map

```
playerStateSubject
    .map(state -> state == VideoPlayerState.PLAYING)
```

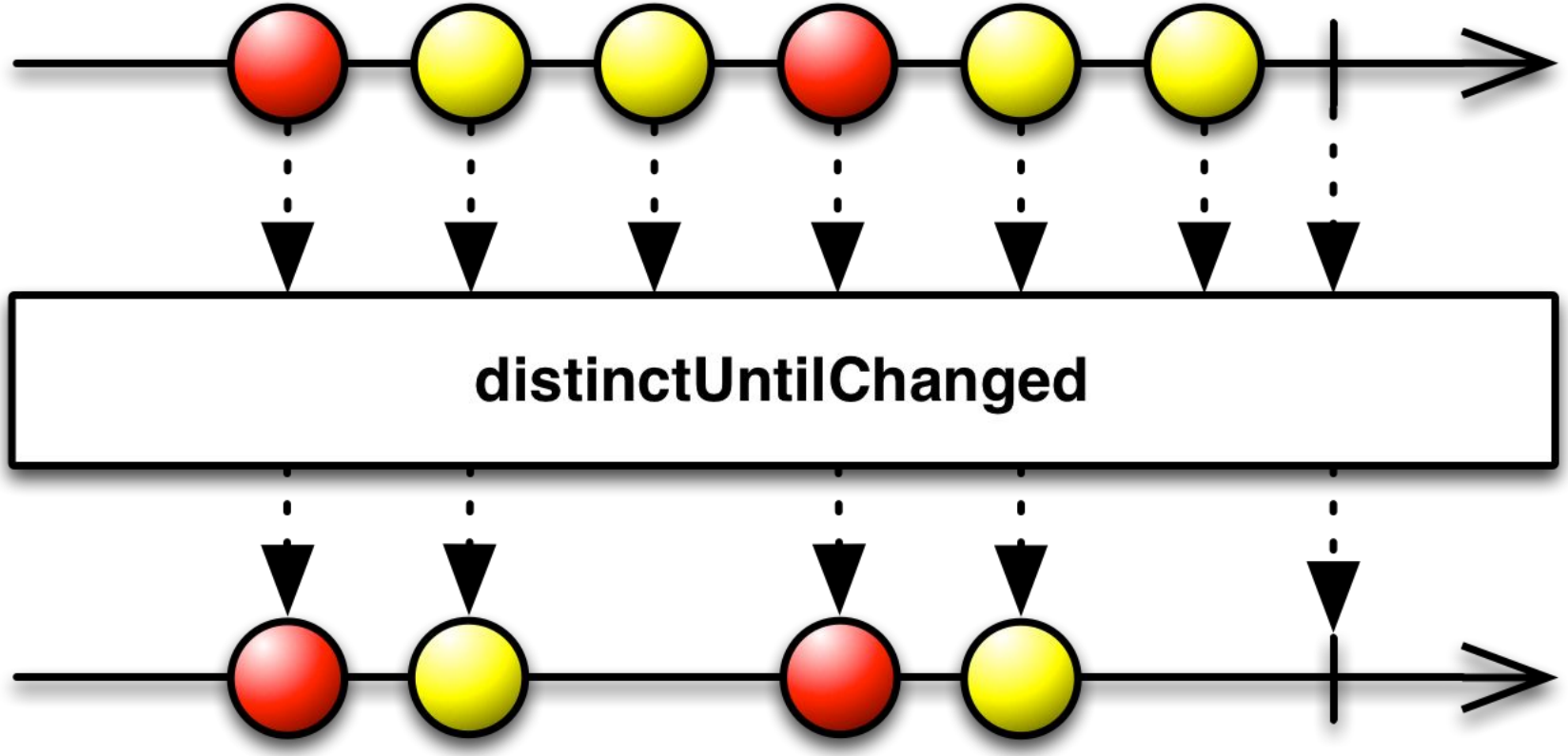
Step 1: map



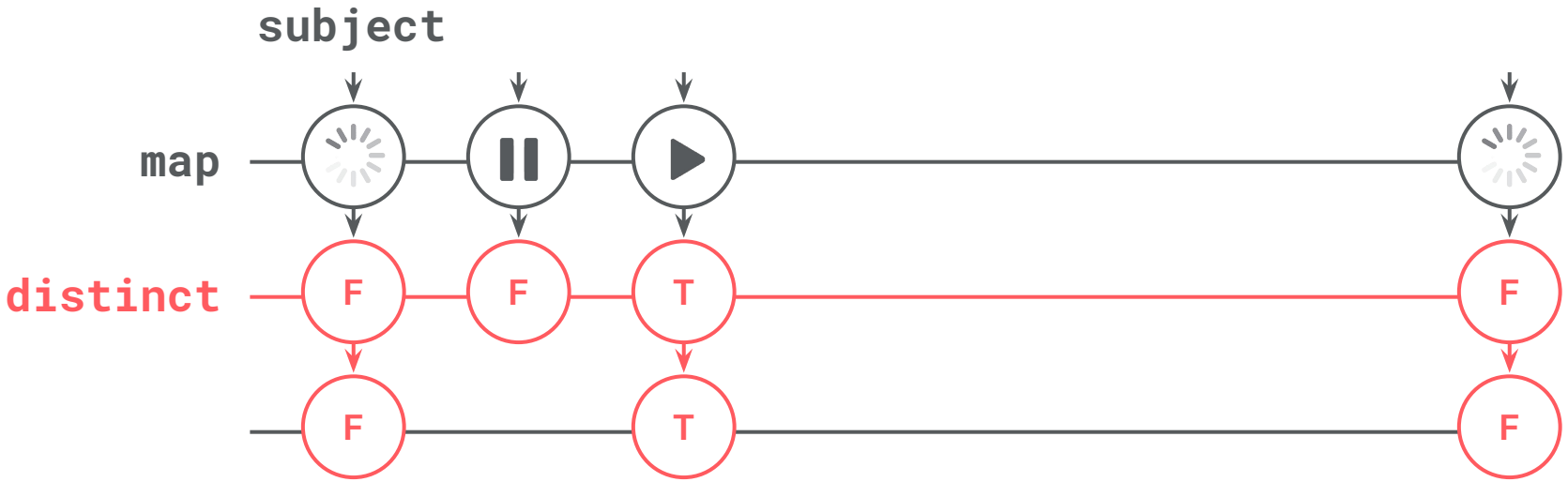
Step 2: distinctUntilChanged

```
playerStateSubject
    .map(state -> state == VideoPlayerState.PLAYING)
    .distinctUntilChanged()
```

Step 2: distinctUntilChanged



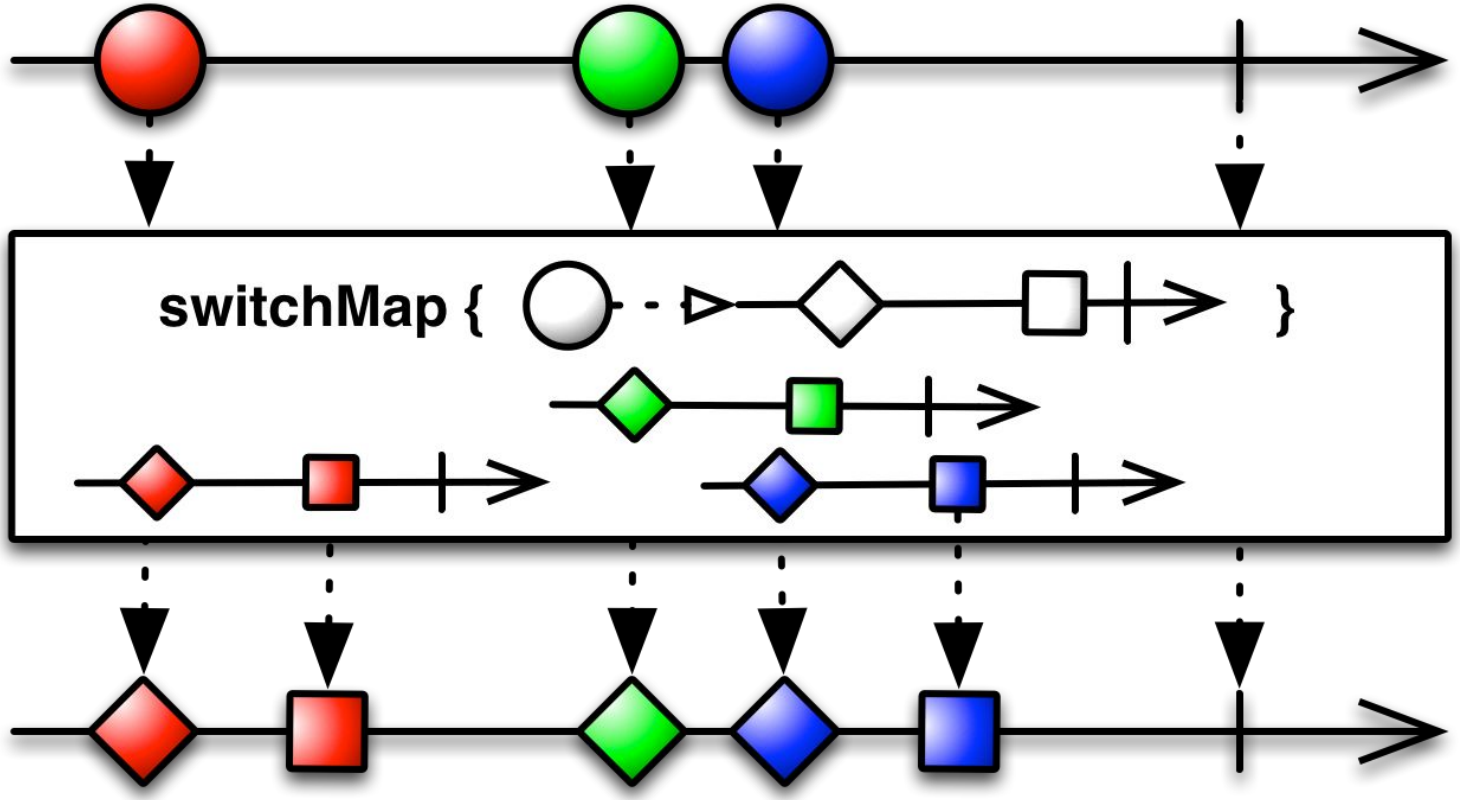
Step 2: distinctUntilChanged



Step 3: switchMap

```
playerStateSubject
    .map(state -> state == VideoPlayerState.PLAYING)
    .distinctUntilChanged()
    .switchMap(isPlaying -> /* return some Observable<T> */)
```

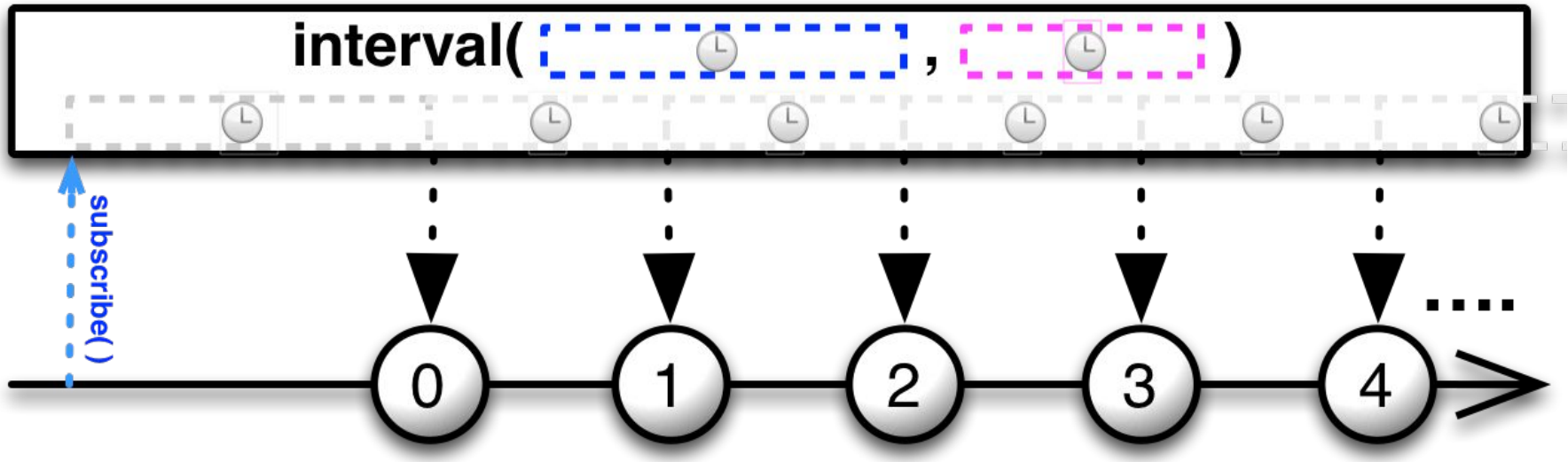
Step 3: switchMap



Step 3a & 3b: interval and never

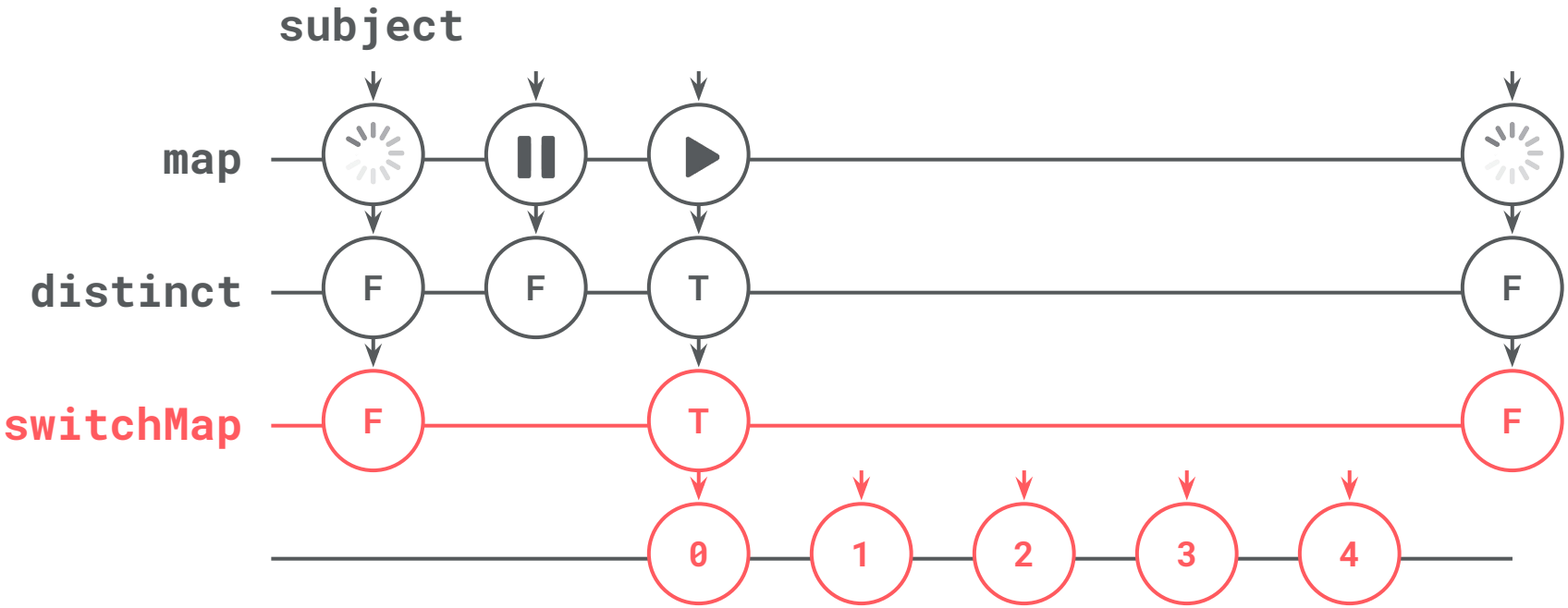
```
playerStateSubject
    .map(state -> state == VideoPlayerState.PLAYING)
    .distinctUntilChanged()
    .switchMap(isPlaying -> {
      return isPlaying ?
          Observable.interval(0, 250, TimeUnit.MILLISECONDS) :
          Observable.never();
    })
```

Step 3a: interval



Step 3b: never

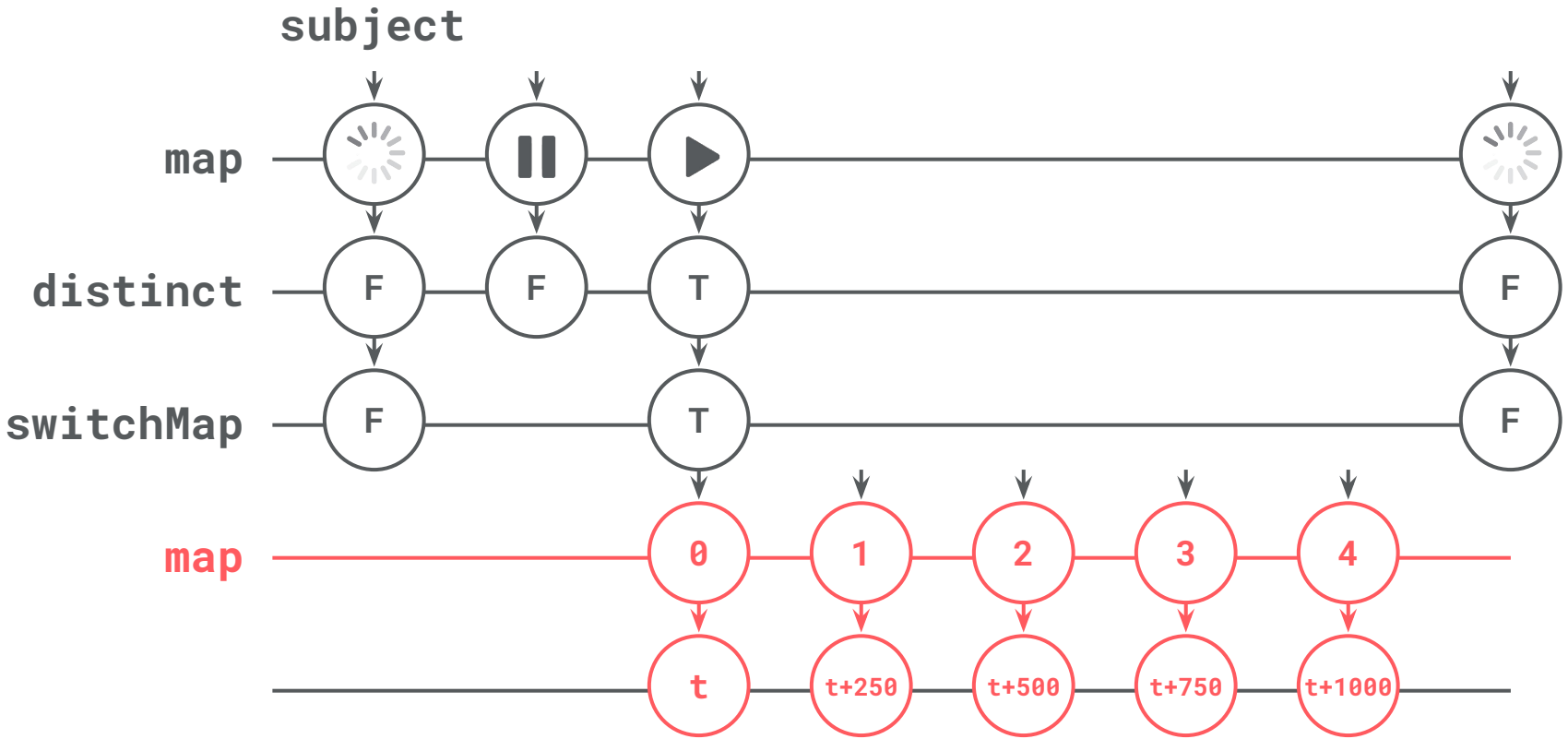
Step 3: switchMap



Step 4: observeOn & map

```
playerStateSubject
    .map(state -> state == VideoPlayerState.PLAYING)
    .distinctUntilChanged()
    .switchMap(isPlaying -> {
      return isPlaying ?
          Observable.interval(0, 250, TimeUnit.MILLISECONDS) :
          Observable.never();
    })
    .observeOn(AndroidSchedulers.mainThread())
    .map(ignored -> exoPlayer.getCurrentPosition())
```

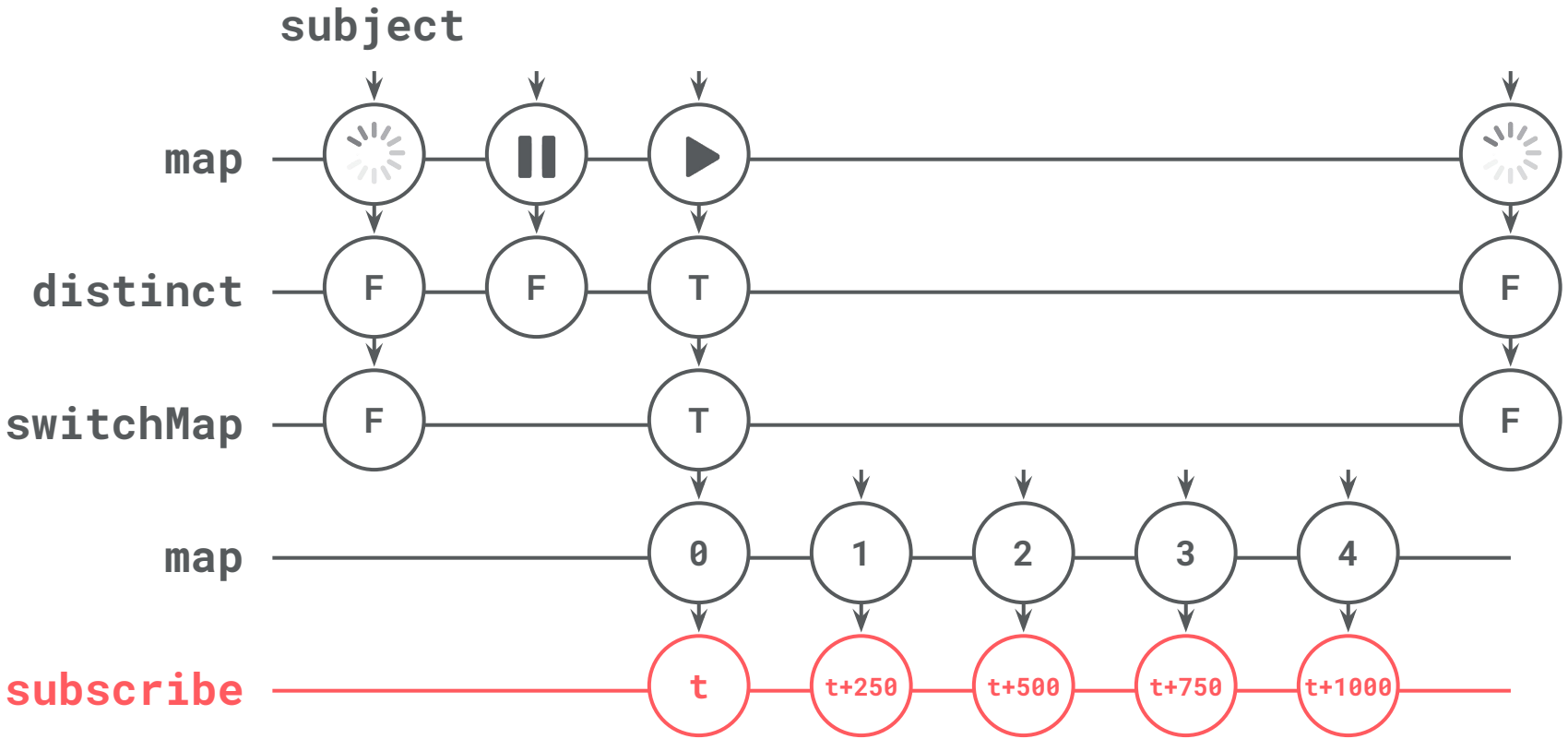
Step 4: observeOn & map



Step 5: subscribe

```
playerStateSubject
    .map(state -> state == VideoPlayerState.PLAYING)
    .distinctUntilChanged()
    .switchMap(isPlaying -> {
      return isPlaying ?
          Observable.interval(0, 250, TimeUnit.MILLISECONDS) :
          Observable.never();
    })
    .observeOn(AndroidSchedulers.mainThread())
    .map(ignored -> exoPlayer.getCurrentPosition())
    .subscribe(timeMillis -> scrollToSubtitle(timeMillis));
```

Step 5: subscribe



Etc.

But you write services, not mobile apps

- Use Lombok's `@Value` annotation (not `@Data`) to create small, focused, immutable value types
- Use Guava's `ImmutableList`, `ImmutableSet`, and `ImmutableMap` to create immutable collections from immutable values
- Use the `Stream` class and its operators to transform immutable collections into other immutable collections

Back to our slightly-less-toy example

```
Observable.just(1, 2, 3, 4, 5)
    .filter(value -> (value % 2) == 1)
    .map(value -> 2 * value)
    .subscribe(value -> System.out.println(value));
```

Using immutable collections & Stream

```
List<Integer> list = ImmutableList.of(1, 2, 3, 4, 5);

List<Integer> otherList = list.stream()
    .filter(value -> (value % 2) == 1)
    .map(value -> 2 * value)
    .collect(GuavaCollectors.immutableList());
```

Caveats

- API surface area is large
- Mental model is different
- Many details to get right
  - Lifecycle concerns
  - Hot vs cold Observables
  - Choosing the right operators, e.g. `flatMap` vs `switchMap`
- Lot of framework code to wade through in stacktraces

Questions?