
Managing
The Reactive World
Jake Wharton


Managing
The Reactive World
(with RxJava)
Jake Wharton

Why Reactive?

Unless you can model your entire system synchronously...

Why Reactive?

Unless you can model your entire system synchronously, a single asynchronous source breaks imperative programming.

Why Reactive?

```
interface UserManager {
  User getUser();
}
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name);
  void setAge(int age);
}
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name);
  void setAge(int age);
}

UserManager um = new UserManager();
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name);
  void setAge(int age);
}

UserManager um = new UserManager();
System.out.println(um.getUser());
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name);
  void setAge(int age);
}

UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe");
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name);
  void setAge(int age);
}

UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe");
System.out.println(um.getUser());
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name); // <-- now async
  void setAge(int age); // <-- now async
}
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name);
  void setAge(int age);
}

UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe");
System.out.println(um.getUser());
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name, Runnable callback);
  void setAge(int age, Runnable callback);
}
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name, Runnable callback);
  void setAge(int age, Runnable callback);
}

UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe", new Runnable() {
  @Override public void run() {
    System.out.println(um.getUser());
  }
});
```

Why Reactive?

```
interface UserManager {
  User getUser();
  void setName(String name, Listener listener);
  void setAge(int age, Listener listener);

  interface Listener {
    void success(User user);
    void failure(IOException e);
  }
}
```

Why Reactive?

```
UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe");
```

Why Reactive?

```
UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe", new UserManager.Listener() {
  @Override public void success() {
    System.out.println(um.getUser());
  }

  @Override public void failure(IOException e) {
    // TODO show the error...
  }
});
```

Why Reactive?

```
UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe", new UserManager.Listener() {
  @Override public void success() {
    System.out.println(um.getUser());
  }
  @Override public void failure(IOException e) {
    // TODO show the error...
  }
});

um.setAge(40, new UserManager.Listener() {
  @Override public void success() {
    System.out.println(um.getUser());
  }
  @Override public void failure(IOException e) {
    // TODO show the error...
  }
});
```

Why Reactive?

```
UserManager um = new UserManager();
System.out.println(um.getUser());

um.setName("Jane Doe", new UserManager.Listener() {
  @Override public void success() {
    System.out.println(um.getUser());

    um.setAge(40, new UserManager.Listener() {
      @Override public void success() {
        System.out.println(um.getUser());
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
  @Override public void failure(IOException e) {
    // TODO show the error...
  }
});
```

Why Reactive?

```
public final class UserActivity extends Activity {
  private final UserManager um = new UserManager();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.user);
    TextView tv = (TextView) findViewById(R.id.user_name);
    tv.setText(um.getUser().toString());

    um.setName("Jane Doe", new UserManager.Listener() {
      @Override public void success() {
        tv.setText(um.getUser().toString());
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
}
```

Why Reactive?

```
public final class UserActivity extends Activity {
  private final UserManager um = new UserManager();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.user);
    TextView tv = (TextView) findViewById(R.id.user_name);
    tv.setText(um.getUser().toString());

    um.setName("Jane Doe", new UserManager.Listener() {
      @Override public void success() {
        tv.setText(um.getUser().toString());
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
}
```

Why Reactive?

```
public final class UserActivity extends Activity {
  private final UserManager um = new UserManager();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.user);
    TextView tv = (TextView) findViewById(R.id.user_name);
    tv.setText(um.getUser().toString());

    um.setName("Jane Doe", new UserManager.Listener() {
      @Override public void success() {
        if (!isDestroyed()) {
          tv.setText(um.getUser().toString());
        }
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
}
```

Why Reactive?

```
public final class UserActivity extends Activity {
  private final UserManager um = new UserManager();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.user);
    TextView tv = (TextView) findViewById(R.id.user_name);
    tv.setText(um.getUser().toString());

    um.setName("Jane Doe", new UserManager.Listener() {
      @Override public void success() {
        if (!isDestroyed()) {
          tv.setText(um.getUser().toString());
        }
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
}
```

Why Reactive?

```
public final class UserActivity extends Activity {
  private final UserManager um = new UserManager();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.user);
    TextView tv = (TextView) findViewById(R.id.user_name);
    tv.setText(um.getUser().toString());

    um.setName("Jane Doe", new UserManager.Listener() {
      @Override public void success() {
        if (!isDestroyed()) {
          tv.setText(um.getUser().toString());
        }
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
}
```

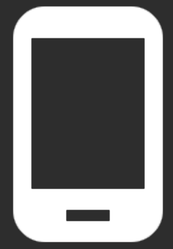
Why Reactive?

```
public final class UserActivity extends Activity {
  private final UserManager um = new UserManager();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.user);
    TextView tv = (TextView) findViewById(R.id.user_name);
    tv.setText(um.getUser().toString());

    um.setName("Jane Doe", new UserManager.Listener() {
      @Override public void success() {
        runOnUiThread(new Runnable() {
          @Override public void run() {
            if (!isDestroyed()) {
              tv.setText(um.getUser().toString());
            }
          }
        });
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
}
```

Why Reactive?

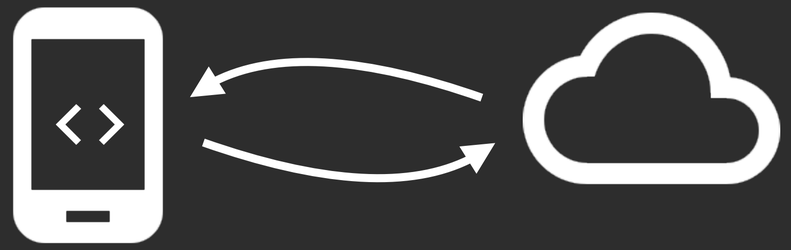
```
public final class UserActivity extends Activity {
  private final UserManager um = new UserManager();

  @Override protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.user);
    TextView tv = (TextView) findViewById(R.id.user_name);
    tv.setText(um.getUser().toString());

    um.setName("Jane Doe", new UserManager.Listener() {
      @Override public void success() {
        runOnUiThread(new Runnable() {
          @Override public void run() {
            if (!isDestroyed()) {
              tv.setText(um.getUser().toString());
            }
          }
        });
      }
      @Override public void failure(IOException e) {
        // TODO show the error...
      }
    });
  }
}
```

Why Reactive?

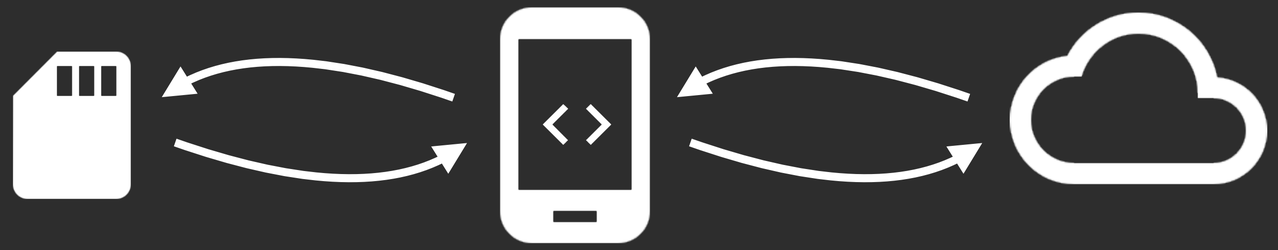


Why Reactive?

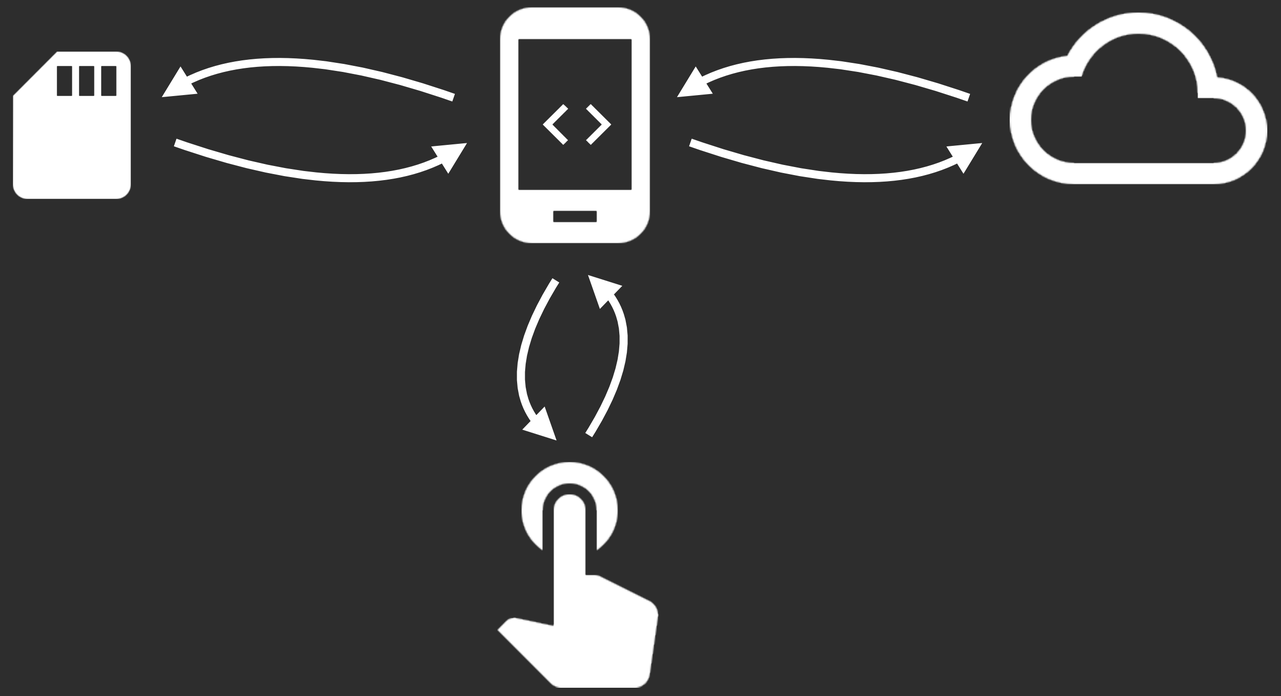
Why Reactive?



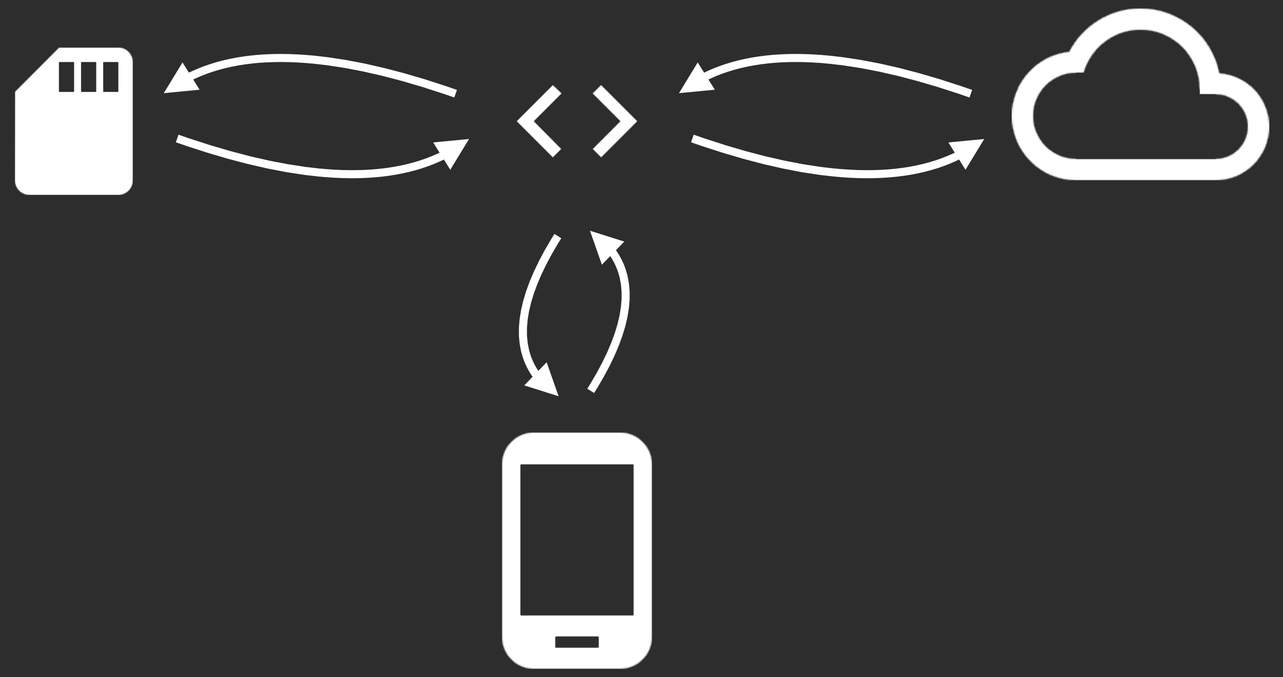
Why Reactive?



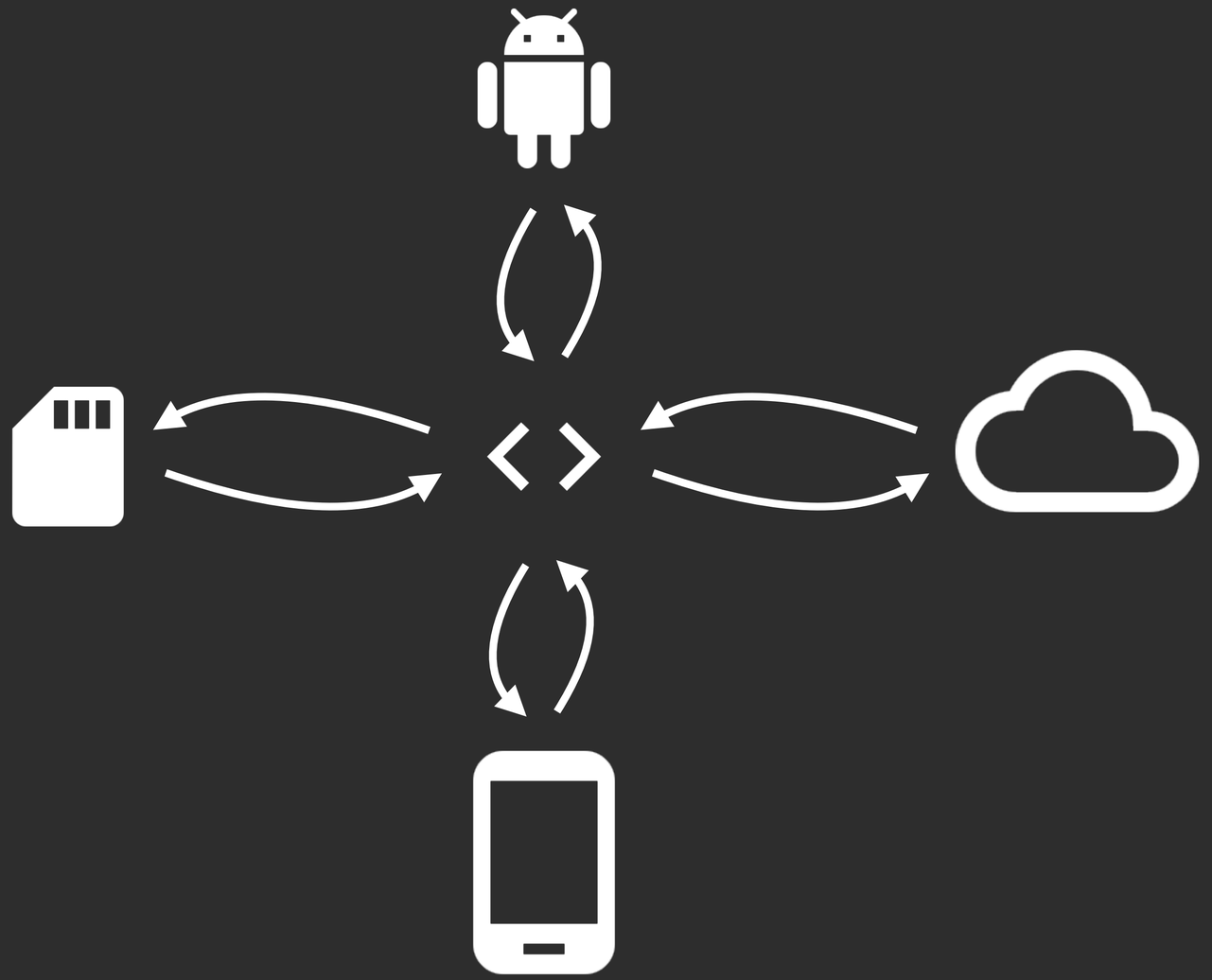
Why Reactive?



Why Reactive?



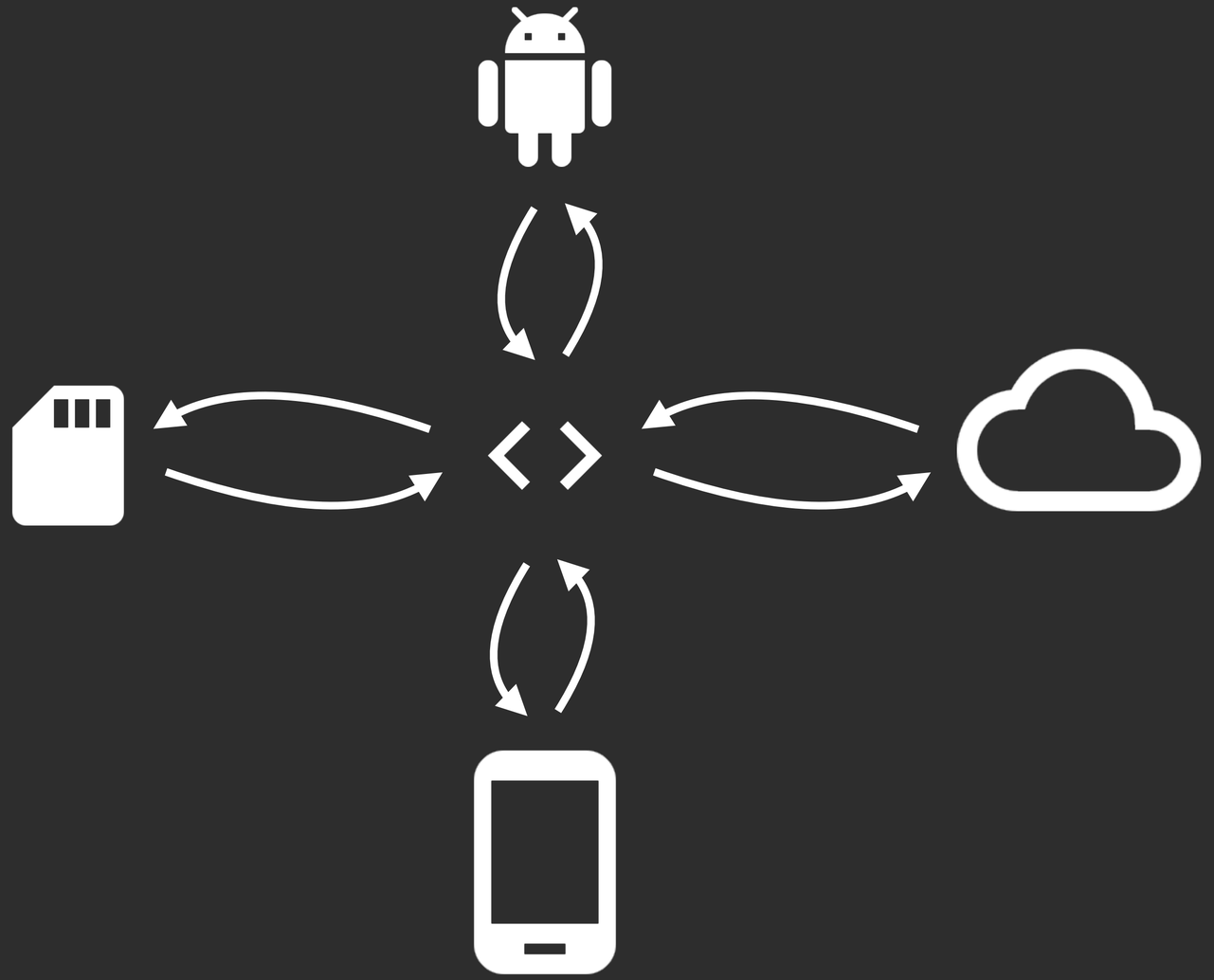
Why Reactive?



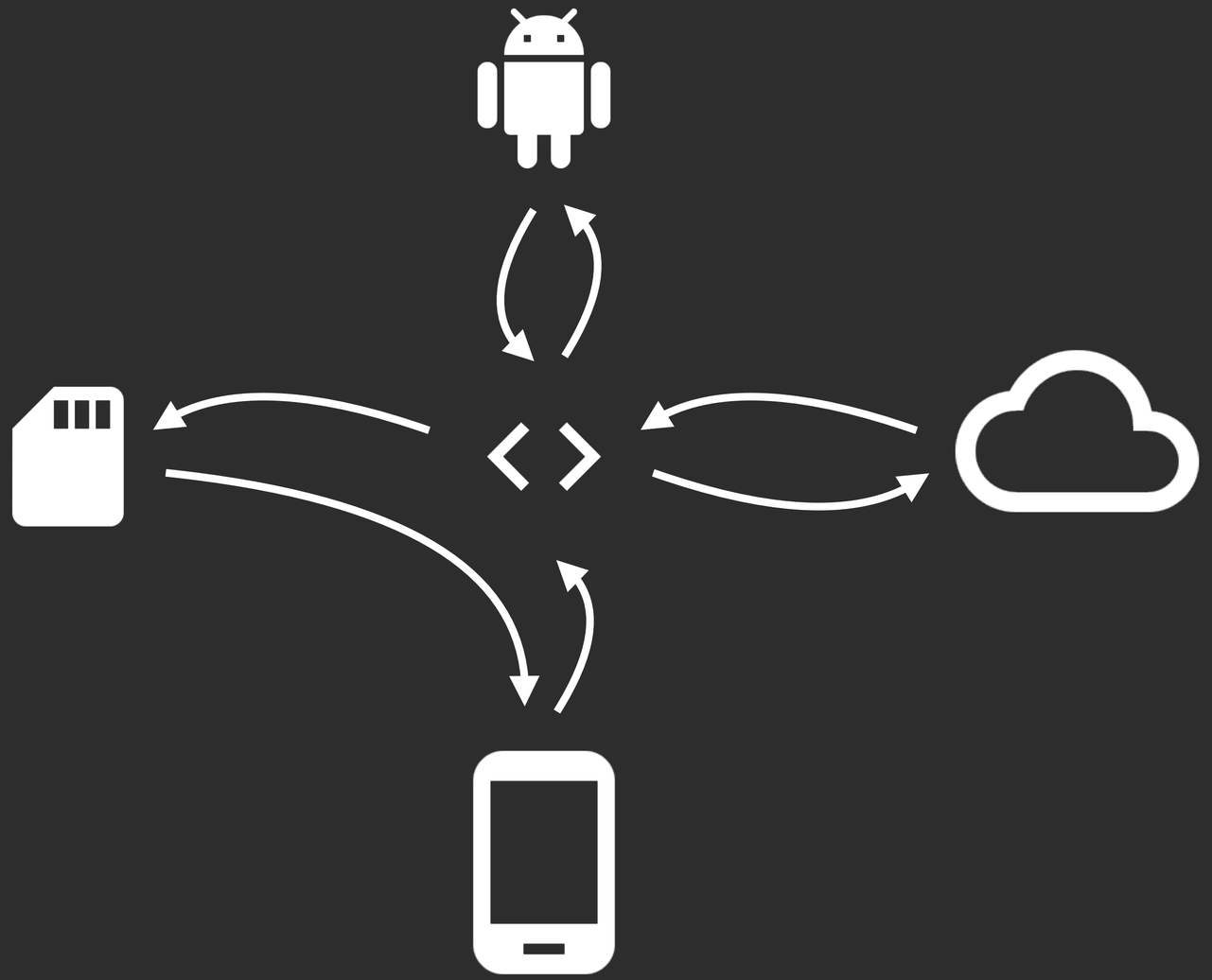
Why Reactive?

Unless you can model your entire system synchronously, a single asynchronous source breaks imperative programming.

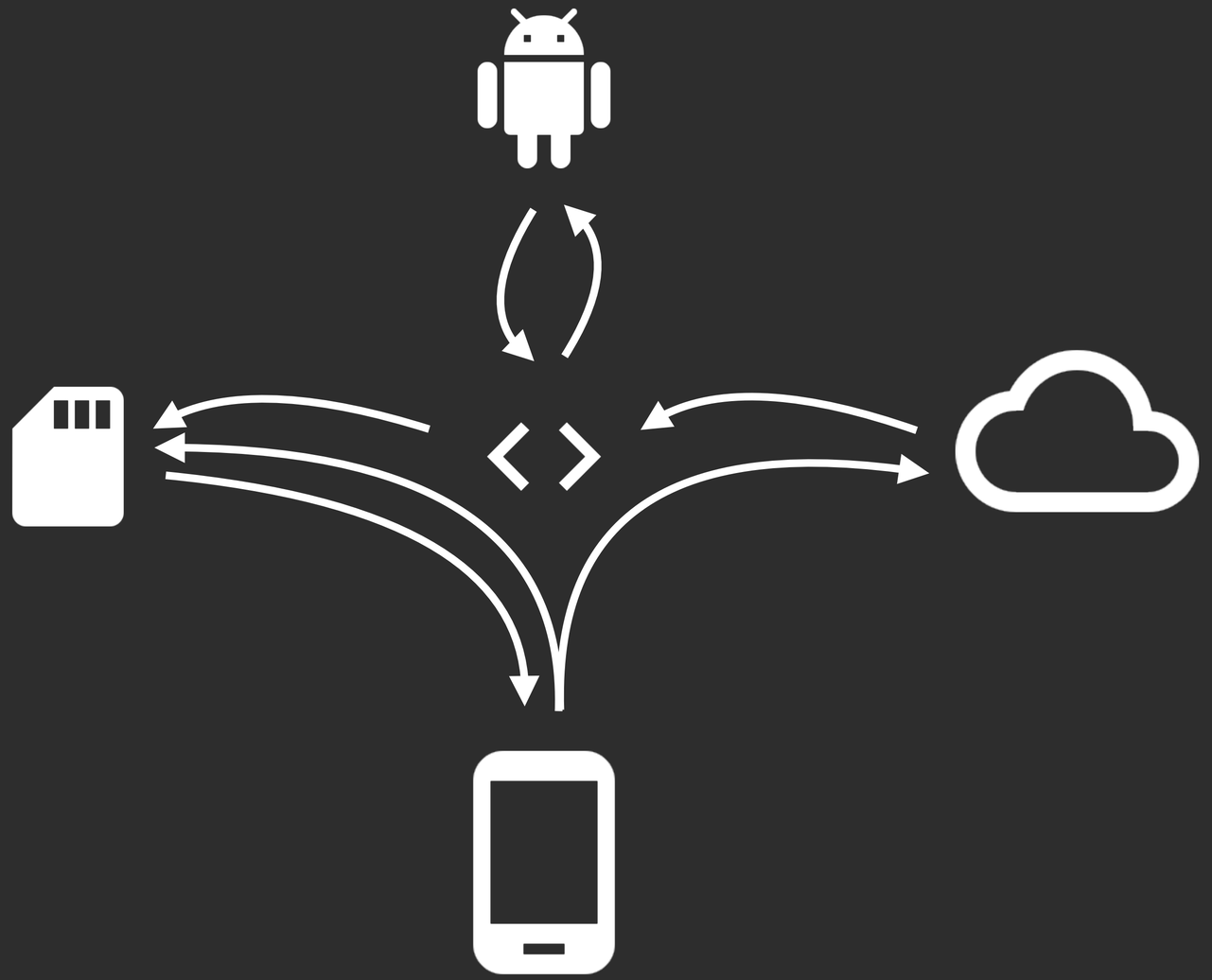
Why Reactive?



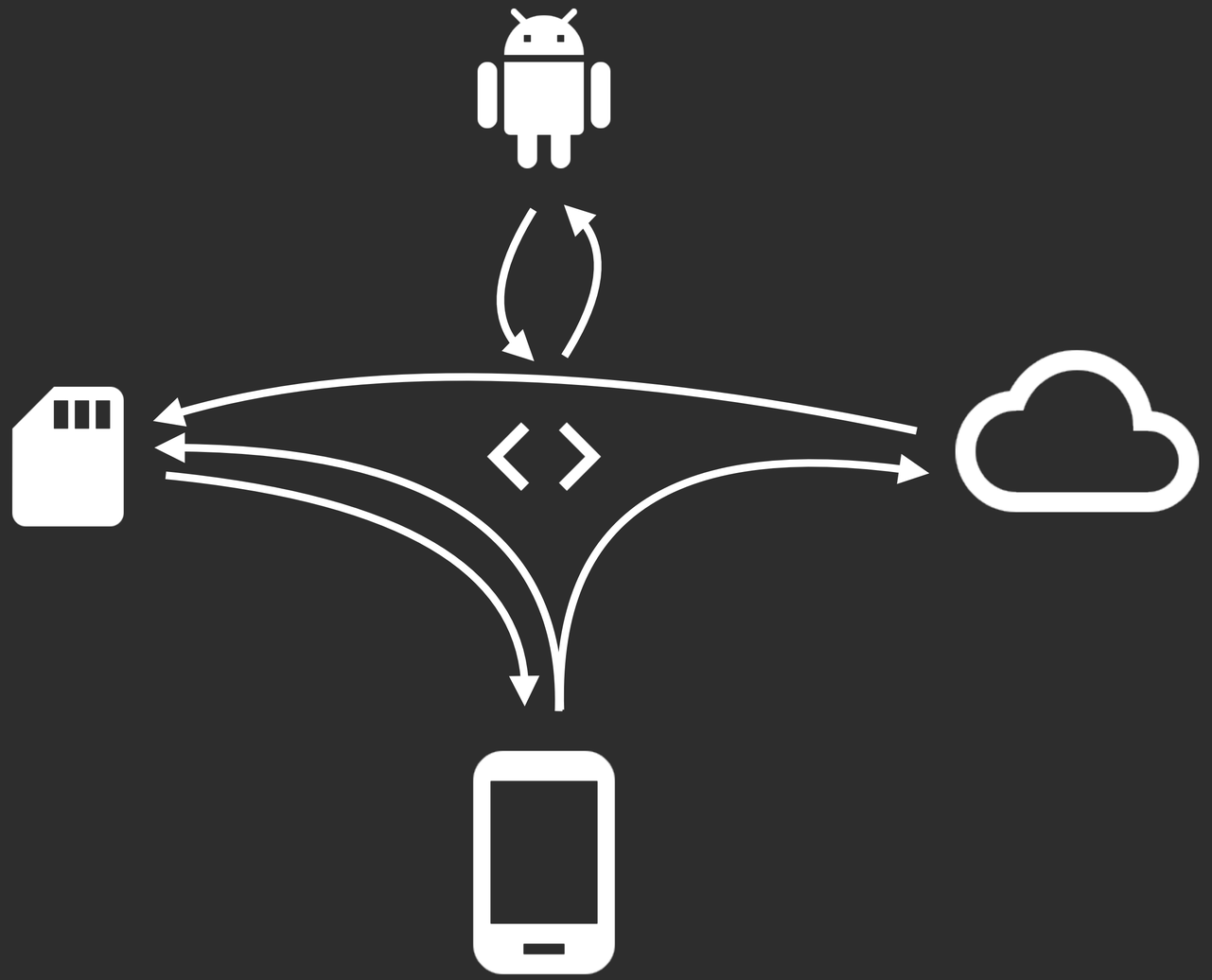
Why Reactive?



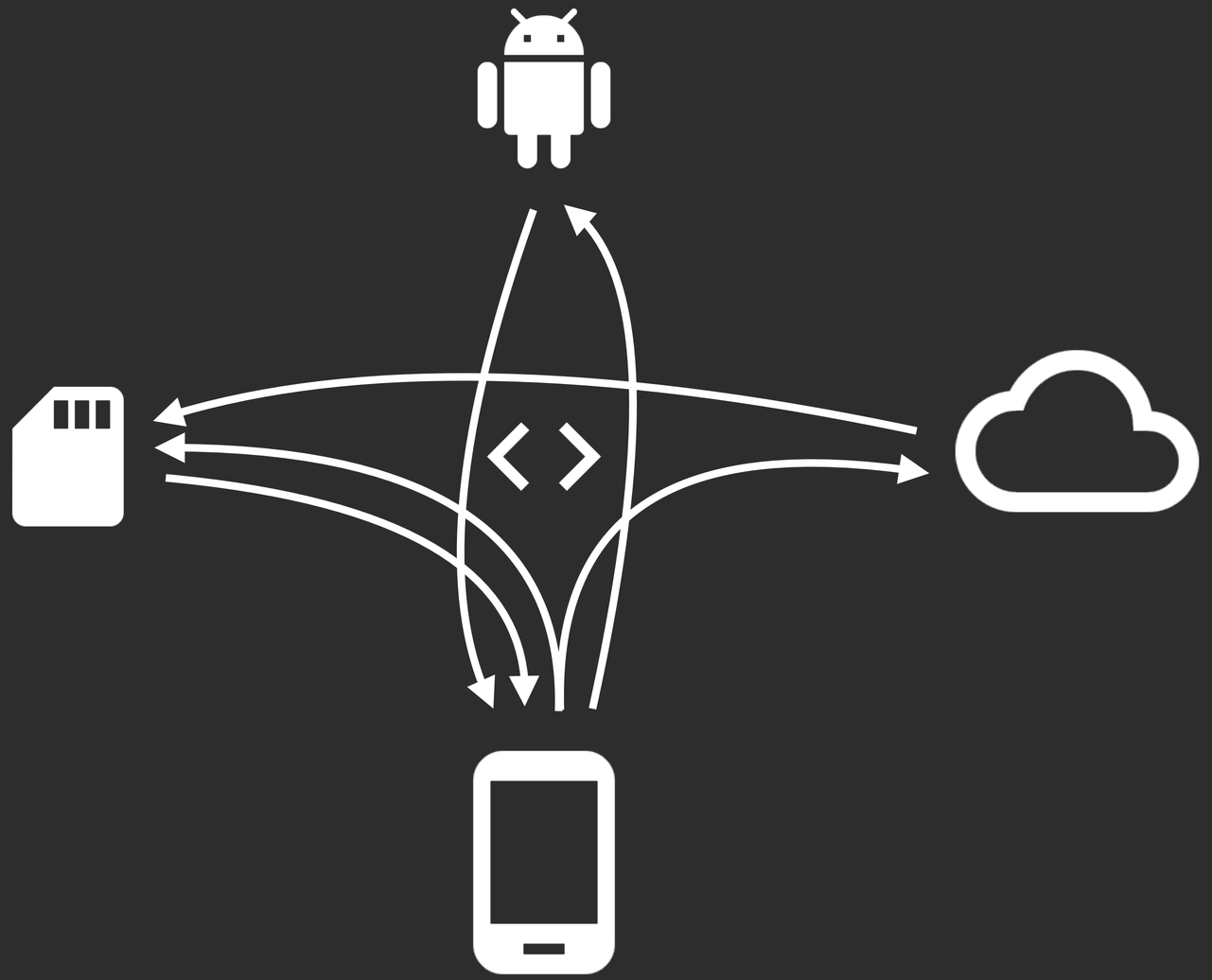
Why Reactive?



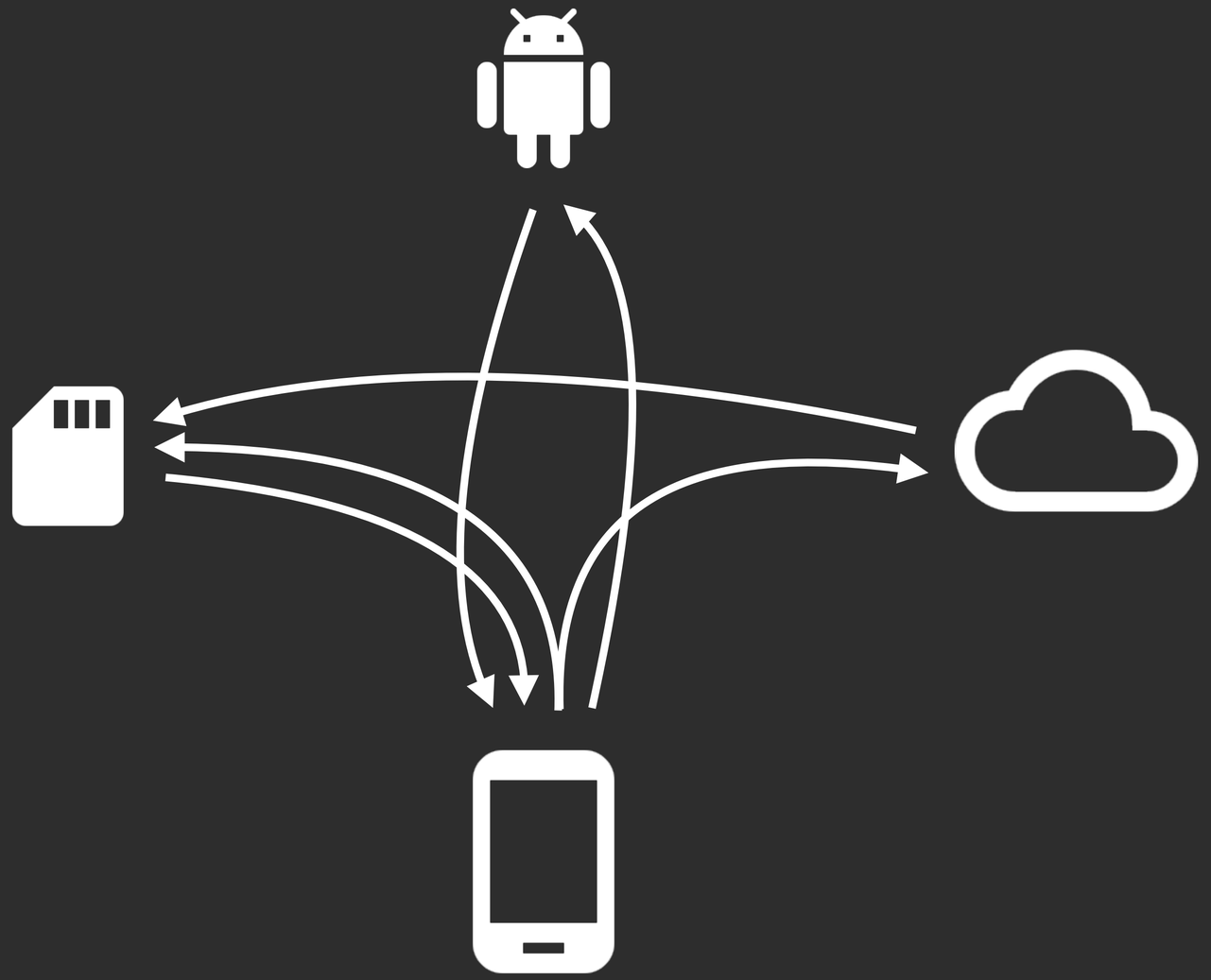
Why Reactive?



Why Reactive?



Why Reactive?



RxJava

RxJava

- A set of classes for representing sources of data.

RxJava

- A set of classes for representing sources of data.
- A set of classes for listening to data sources.

RxJava

- A set of classes for representing sources of data.
- A set of classes for listening to data sources.
- A set of methods for modifying and composing the data.

RxJava

- A set of classes for representing sources of data.
- A set of classes for listening to data sources.
- A set of methods for modifying and composing the data.

Sources

Sources

- Usually do work when you start or stop listening.

Sources

- Usually do work when you start or stop listening.
- Synchronous or asynchronous.

Sources

- Usually do work when you start or stop listening.
- Synchronous or asynchronous.
- Single item or many items.

Sources

- Usually do work when you start or stop listening.
- Synchronous or asynchronous.
- Single item, many items, or empty.

Sources

- Usually do work when you start or stop listening.
- Synchronous or asynchronous.
- Single item, many items, or empty.
- Terminates with an error or succeeds to completion.

Sources

- Usually do work when you start or stop listening.
- Synchronous or asynchronous.
- Single item, many items, or empty.
- Terminates with an error or succeeds to completion.
- May never terminate!

Sources

- Usually do work when you start or stop listening.
- Synchronous or asynchronous.
- Single item, many items, or empty.
- Terminates with an error or succeeds to completion.
- May never terminate!
- Just an implementation of the Observer pattern.

Sources

- `Observable<T>`
- `Flowable<T>`

Sources

- `Observable<T>`
  - Emits `0` to `n` items.
  - Terminates with complete or error.
- `Flowable<T>`
  - Emits `0` to `n` items.
  - Terminates with complete or error.

Sources

- `Observable<T>`
  - Emits `0` to `n` items.
  - Terminates with complete or error.
  - Does not have backpressure.
- `Flowable<T>`
  - Emits `0` to `n` items.
  - Terminates with complete or error.
  - Has backpressure.

Flowable vs. Observable

Flowable vs. Observable

- Backpressure allows you to control how fast a source emits items.

Flowable vs. Observable

- Backpressure allows you to control how fast a source emits items.
- RxJava 1.x added backpressure late in the design process.

Flowable vs. Observable

- Backpressure allows you to control how fast a source emits items.
- RxJava 1.x added backpressure late in the design process.
- All types exposed backpressure but not all sources respected it.

Flowable vs. Observable

- Backpressure allows you to control how fast a source emits items.
- RxJava 1.x added backpressure late in the design process.
- All types exposed backpressure but not all sources respected it.
- Backpressure must be designed for.

Flowable vs. Observable

- Backpressure must be designed for.

Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

```
Observable<Row> rows
    = db.createQuery("SELECT * …");
```

Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

```
Observable<Row> rows
    = db.createQuery("SELECT * …");
```

Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

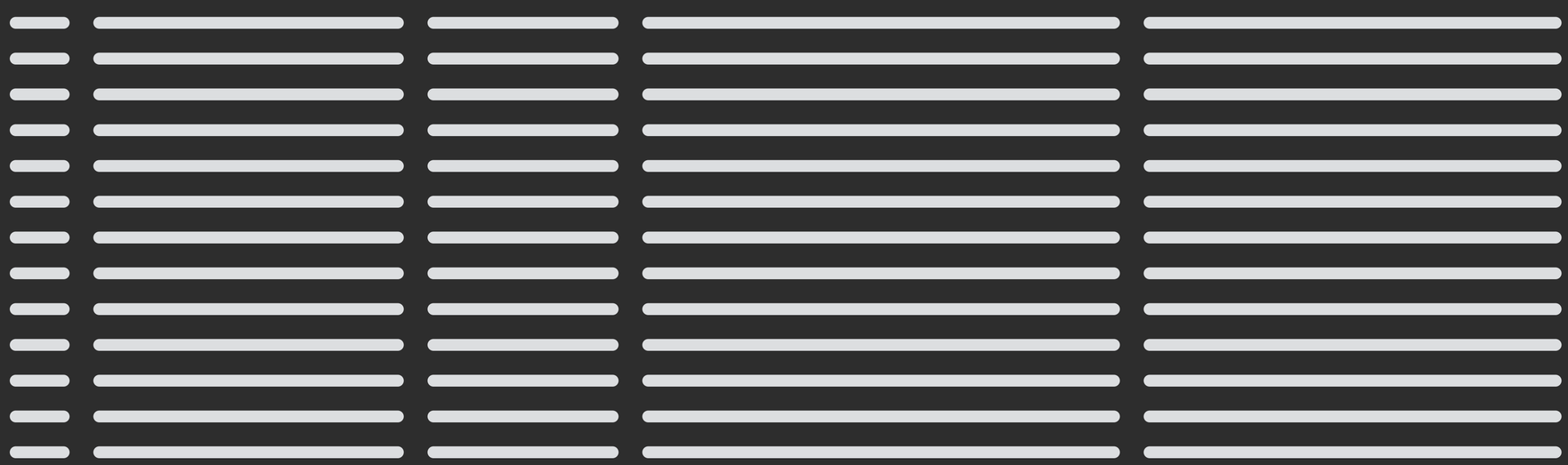
```
Observable<Row> rows
    = db.createQuery("SELECT * …");
```

Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

```
Observable<Row> rows
    = db.createQuery("SELECT * …");
```



Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

```
Observable<Row> rows
    = db.createQuery("SELECT * …");
```

```
MissingBackpressureException
```

Flowable vs. Observable

- Backpressure must be designed for.

```
Observable<MotionEvent> events
    = RxView.touches(paintView);
```

```
Flowable<Row> rows
    = db.createQuery("SELECT * …");
```

Flowable vs. Observable

```
Observable<MotionEvent>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}
```

```
Flowable<Row>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Flowable vs. Observable

```
Observable<MotionEvent>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}
```

```
Flowable<Row>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Flowable vs. Observable

```
Observable<MotionEvent>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}
```

```
Flowable<Row>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Flowable vs. Observable

```
Observable<MotionEvent>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}
```

```
Flowable<Row>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Flowable vs. Observable

```
Observable<MotionEvent>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}
```

```
Flowable<Row>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Flowable vs. Observable

```
Observable<MotionEvent>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}


interface Disposable {
  void dispose();
}
```

```
Flowable<Row>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Flowable vs. Observable

```
Observable<MotionEvent>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}


interface Disposable {
  void dispose();
}
```

```
Flowable<Row>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}


interface Subscription {
  void cancel();
  void request(long r);
}
```

	Backpressure	No Backpressure
0…n items, complete|error	`Flowable`	`Observable`

Reactive Streams

...is an initiative to provide a standard
for asynchronous stream processing
with non-blocking back pressure.

Reactive Streams

```
interface Publisher<T> {
  void subscribe(Subscriber<? super T> s);
}
```

Reactive Streams

```
interface Publisher<T> {
  void subscribe(Subscriber<? super T> s);
}

interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Reactive Streams

```
interface Publisher<T> {
  void subscribe(Subscriber<? super T> s);
}

interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}

interface Subscription {
  void request(long n);
  void cancel();
}
```

Reactive Streams

```
interface Publisher<T> {
  void subscribe(Subscriber<? super T> s);
}

interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}

interface Subscription {
  void request(long n);
  void cancel();
}

interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

	Reactive Streams (Backpressure)	No Backpressure
0…n items, complete|error	`Flowable`	`Observable`

Sources

```
interface UserManager {
  User getUser();
  void setName(String name);
  void setAge(int age);
}
```

Sources

```
interface UserManager {
  Observable<User> getUser();
  void setName(String name);
  void setAge(int age);
}
```

Source Specializations

- Encoding subsets of `Observable` into the type system.

Single

- Either succeeds with an item or errors.
- No backpressure support.

Single

- Either succeeds with an item or errors.
- No backpressure support.
- Think "reactive scalar".

Completable

- Either completes or errors. Has no items!
- No backpressure support.

Completable

- Either completes or errors. Has no items!
- No backpressure support.
- Think "reactive runnable".

Maybe

- Either succeeds with an item, completes with no items, or errors.
- No backpressure support.

Maybe

- Either succeeds with an item, completes with no items, or errors.
- No backpressure support.
- Think "reactive optional".

Source Specializations

- Encoding subsets of `Observable` into the type system.
  - `Single` – Item or error. Think "scalar".
  - `Completable` – Complete or error. Think "runnable".
  - `Maybe` – Item, complete, or error. Think "optional".

	Reactive Streams (Backpressure)	No Backpressure
0…n items, complete|error	`Flowable`	`Observable`
item|complete|error		`Maybe`
item|error		`Single`
complete|error		`Completable`

Sources

```
interface UserManager {
  Observable<User> getUser();
  void setName(String name);
  void setAge(int age);
}
```

Sources

```
interface UserManager {
  Observable<User> getUser();
  Completable setName(String name);
  Completable setAge(int age);
}
```

Creating Sources

```
Flowable.just("Hello");
Flowable.just("Hello", "World");

Observable.just("Hello");
Observable.just("Hello", "World");

Maybe.just("Hello");

Single.just("Hello");
```

Creating Sources

```
String[] array = { "Hello", "World" };
List<String> list = Arrays.asList(array);

Flowable.fromArray(array);
Flowable.fromIterable(list);

Observable.fromArray(array);
Observable.fromIterable(list);
```

Creating Sources

```
Observable.fromCallable(new Callable<String>() {
  @Override public String call() {
    return getName();
  }
});
```

Creating Sources

```
Observable.fromCallable(new Callable<String>() {
  @Override public String call() throws Exception {
    return getName();
  }
});
```

Creating Sources

```
OkHttpClient client = // …
Request request = // …

Observable.fromCallable(new Callable<String>() {
  @Override public String call() throws Exception {
    return client.newCall(request).execute();
  }
});
```

Creating Sources

```
Flowable.fromCallable(() -> "Hello");

Observable.fromCallable(() -> "Hello");

Maybe.fromCallable(() -> "Hello");

Single.fromCallable(() -> "Hello");

Completable.fromCallable(() -> "Ignored!");
```

Creating Sources

```
Flowable.fromCallable(() -> "Hello");

Observable.fromCallable(() -> "Hello");

Maybe.fromCallable(() -> "Hello");
Maybe.fromAction(() -> System.out.println("Hello"));
Maybe.fromRunnable(() -> System.out.println("Hello"))

Single.fromCallable(() -> "Hello");

Completable.fromCallable(() -> "Ignored!");
Completable.fromAction(() -> System.out.println("Hello"));
Completable.fromRunnable(() -> System.out.println("Hello"));
```

Creating Sources

```
Observable.create();
```

Creating Sources

```
Observable.create(new ObservableOnSubscribe<String>() {
  @Override
  public void subscribe(ObservableEmitter<String> e) throws Exception {
    e.onNext("Hello");
    e.onComplete();
  }
});
```

Creating Sources

```
Observable.create(new ObservableOnSubscribe<String>() {
  @Override
  public void subscribe(ObservableEmitter<String> e) throws Exception {
    e.onNext("Hello");
    e.onComplete();
  }
});
```

Creating Sources

```
Observable.create(new ObservableOnSubscribe<String>() {
  @Override
  public void subscribe(ObservableEmitter<String> e) throws Exception {
    e.onNext("Hello");
    e.onComplete();
  }
});
```

Creating Sources

```
Observable.create(new ObservableOnSubscribe<String>() {
  @Override
  public void subscribe(ObservableEmitter<String> e) throws Exception {
    e.onNext("Hello");
    e.onComplete();
  }
});
```

Creating Sources

```
Observable.create(e -> {
  e.onNext("Hello");
  e.onComplete();
});
```

Creating Sources

```
Observable.create(e -> {
  e.onNext("Hello");
  e.onNext("World");
  e.onComplete();
});
```

Creating Sources

```
OkHttpClient client = // …
Request request = // …

Observable.create(e -> {
  client.newCall(request).enqueue(new Callback() {
    @Override public void onResponse(Response r) throws IOException {
      e.onNext(r.body().string());
      e.onComplete();
    }
    @Override public void onFailure(IOException e) {
      e.onError(e);
    }
  });
});
```

Creating Sources

```
OkHttpClient client = // …
Request request = // …

Observable.create(e -> {
  Call call = client.newCall(request);
  call.enqueue(new Callback() {
    @Override public void onResponse(Response r) throws IOException {
      e.onNext(r.body().string());
      e.onComplete();
    }
    @Override public void onFailure(IOException e) {
      e.onError(e);
    }
  });
});
```

Creating Sources

```
OkHttpClient client = // …
Request request = // …

Observable.create(e -> {
  Call call = client.newCall(request);
  e.setCancelation(() -> call.cancel());
  call.enqueue(new Callback() {
    @Override public void onResponse(Response r) throws IOException {
      e.onNext(r.body().string());
      e.onComplete();
    }
    @Override public void onFailure(IOException e) {
      e.onError(e);
    }
  });
});
```

Creating Sources

```
View view = // …

Observable.create(e -> {
  e.setCancelation(() -> view.setOnClickListener(null));
  view.setOnClickListener(v -> e.onNext(v));
});
```

Creating Sources

```
Flowable.create(e -> { … });

Observable.create(e -> { … });

Maybe.create(e -> { … });

Single.create(e -> { … });

Completable.create(e -> { … });
```

Observing Sources

```
Observable<String>


interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}
```

```
Flowable<String>


interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}
```

Observing Sources

Observable<String>

```
interface Observer<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Disposable d);
}

interface Disposable {
  void dispose();
}
```

Flowable<String>

```
interface Subscriber<T> {
  void onNext(T t);
  void onComplete();
  void onError(Throwable t);
  void onSubscribe(Subscription s);
}

interface Subscription {
  void cancel();
  void request(long r);
}
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

o.subscribe(new Observer<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }

  @Override public void onSubscribe(Disposable d) {
    ???
  }
});
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

o.subscribe(new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
});
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

o.subscribe(new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
});

// TODO how do we dispose???
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

DisposableObserver observer = new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
}
o.subscribe(observer);
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

DisposableObserver observer = new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
}
o.subscribe(observer);

observer.dispose();
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

o.subscribe(new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
});
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

o.subscribeWith(new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
});
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

Disposable d = o.subscribeWith(new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
});

d.dispose();
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");

CompositeDisposable disposables = new CompositeDisposable();

disposables.add(o.subscribeWith(new DisposableObserver<String>() {
  @Override public void onNext(String s) { … }
  @Override public void onComplete() { … }
  @Override public void onError(Throwable t) { … }
}));

disposables.dispose();
```

Observing Sources

```
Observable<String> o = Observable.just("Hello");
o.subscribeWith(new DisposableObserver<String>() { … });

Maybe<String> m = Maybe.just("Hello");
m.subscribeWith(new DisposableMaybeObserver<String>() { … });

Single<String> s = Single.just("Hello");
s.subscribeWith(new DisposableSingleObserver<String>() { … });

Completable c = Completable.completed();
c.subscribeWith(new DisposableCompletableObserver<String>() { … });
```

Observing Sources

```
Flowable<String> f = Flowable.just("Hello");
f.subscribeWith(new DisposableSubscriber<String>() { … });

Observable<String> o = Observable.just("Hello");
o.subscribeWith(new DisposableObserver<String>() { … });

Maybe<String> m = Maybe.just("Hello");
m.subscribeWith(new DisposableMaybeObserver<String>() { … });

Single<String> s = Single.just("Hello");
s.subscribeWith(new DisposableSingleObserver<String>() { … });

Completable c = Completable.completed();
c.subscribeWith(new DisposableCompletableObserver<String>() { … });
```

Observing Sources

```
Flowable<String> f = Flowable.just("Hello");
Disposable d1 = f.subscribeWith(new DisposableSubscriber<String>() { … });

Observable<String> o = Observable.just("Hello");
Disposable d2 = o.subscribeWith(new DisposableObserver<String>() { … });

Maybe<String> m = Maybe.just("Hello");
Disposable d3 = m.subscribeWith(new DisposableMaybeObserver<String>() { … });

Single<String> s = Single.just("Hello");
Disposable d4 = s.subscribeWith(new DisposableSingleObserver<String>() { … });

Completable c = Completable.completed();
Disposable d5 = c.subscribeWith(new DisposableCompletableObserver<String>() { … });
```

RxJava

- A set of classes for representing sources of data.
- A set of classes for listening to data sources.
- A set of methods for modifying and composing data.

RxJava

- A set of classes for representing sources of data.
- A set of classes for listening to data sources.
- A set of methods for modifying and composing data.

Operators

- Manipulate or combine data in some way.
- Manipulate threading in some way.
- Manipulate emissions in some way.

Operators

```
String greeting = "Hello";
```

Operators

```
String greeting = "Hello";
String yelling = greeting.toUppercase();
```

Operators

```
Observable<String> greeting = Observable.just("Hello");
String yelling = greeting.toUppercase();
```

Operators

```
Observable<String> greeting = Observable.just("Hello");
Observable<String> yelling = greeting.map(s -> s.toUppercase());
```

Operators

```
Observable<String> greeting = Observable.just("Hello");
Observable<String> yelling = greeting.map(s -> s.toUppercase());
```

Operators

```
String greeting = "Hello, World!";
```

Operators

```
String greeting = "Hello, World!";
String[] words = greeting.split(" ");
```

Operators

```
Observable<String> greeting = Observable.just("Hello, World!");
String[] words = greeting.split(" ");
```

Operators

```
Observable<String> greeting = Observable.just("Hello, World!");
Observable<String[]> words = greeting.map(s -> s.split(" "));
```

Operators

```
Observable<String> greeting = Observable.just("Hello, World!");
Observable<Observable<String>> words =
    greeting.map(s -> Observable.fromArray(s.split(" ")));
```

Operators

```
Observable<String> greeting = Observable.just("Hello, World!");
Observable<String> words =
    greeting.flatMap(s -> Observable.fromArray(s.split(" ")));
```

Operators

```
Observable<String> greeting = Observable.just("Hello, World!");
Observable<String> words =
    greeting.flatMap(s -> Observable.fromArray(s.split(" ")));
```

Operators

```
@Override public void success() {
  runOnUiThread(new Runnable() {
    @Override public void run() {
      tv.setText(um.getUser().toString());
    }
  });
}
```

Operators

```
Observable<User> user = um.getUser();
```

Operators

```
Observable<User> user = um.getUser();
Observable<User> mainThreadUser =
    user.observeOn(AndroidSchedulers.mainThread());
```

Operators

```
Observable<User> user = um.getUser();
Observable<User> mainThreadUser =
    user.observeOn(AndroidSchedulers.mainThread());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Response response = client.newCall(request).execute();
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
 return client.newCall(request).execute();
});
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
 return client.newCall(request).execute();
});
Observable<Response> backgroundResponse =
    response.subscribeOn(Schedulers.io());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
 return client.newCall(request).execute();
});
Observable<Response> backgroundResponse =
    response.subscribeOn(Schedulers.io());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
 return client.newCall(request).execute();
});
Observable<Response> backgroundResponse =
    response.subscribeOn(Schedulers.io());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
     return client.newCall(request).execute();
    })
    .subscribeOn(Schedulers.io());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
     return client.newCall(request).execute();
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
     return client.newCall(request).execute();
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .map(response -> response.body().string());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
     return client.newCall(request).execute();
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .map(response -> response.body().string()); // NetworkOnMainThread!
```

Operators

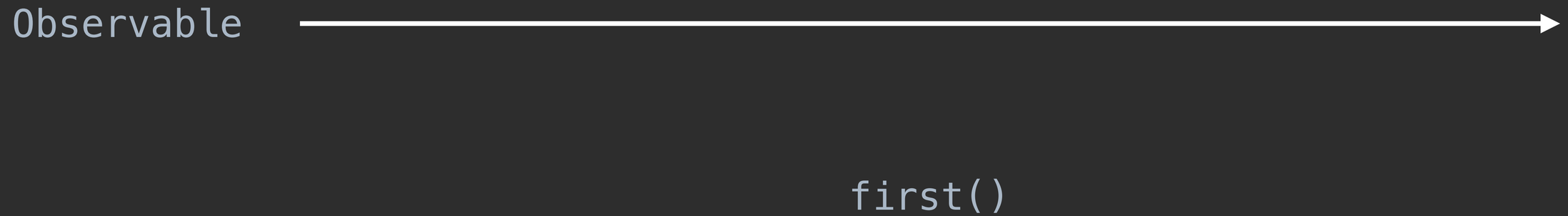
```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
     return client.newCall(request).execute();
    })
    .subscribeOn(Schedulers.io())
    .map(response -> response.body().string())  // Ok!
    .observeOn(AndroidSchedulers.mainThread());
```

Operators

```
OkHttpClient client = // …
Request request = // …

Observable<Response> response = Observable.fromCallable(() -> {
     return client.newCall(request).execute();
    })
    .subscribeOn(Schedulers.io())
    .map(response -> response.body().string())
    .flatMap(s -> Observable.fromArray(s.split(" ")))
    .observeOn(AndroidSchedulers.mainThread());
```

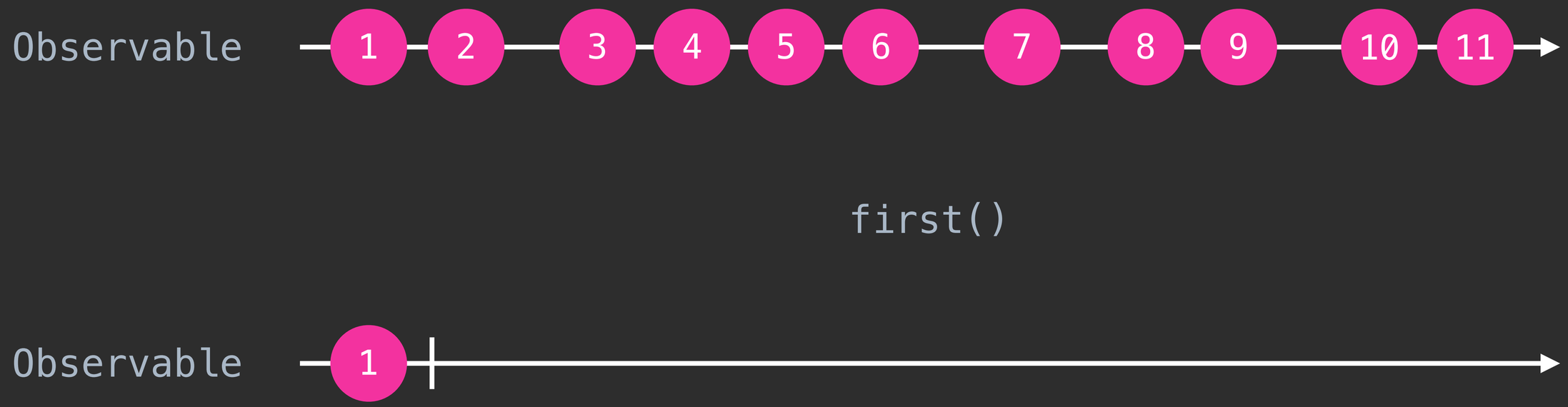
Operator Specialization

Operator Specialization

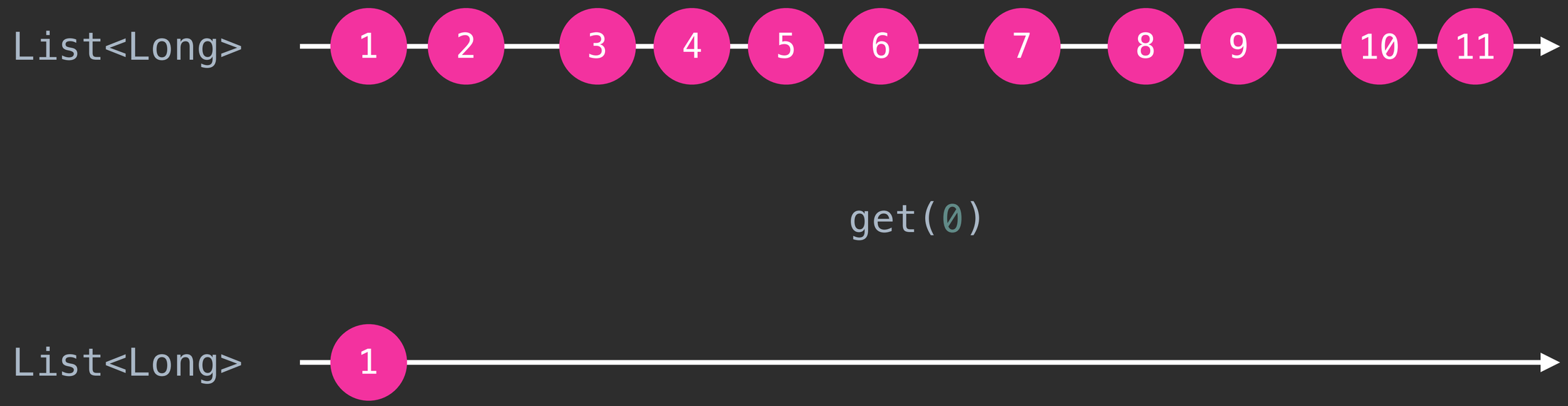


Operator Specialization

Operator Specialization

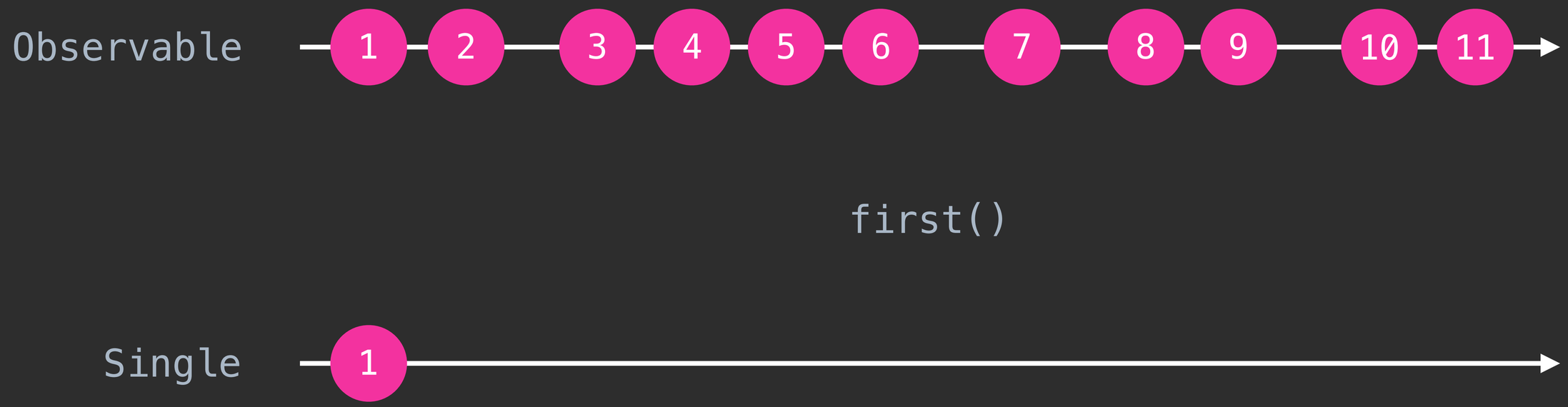


Operator Specialization

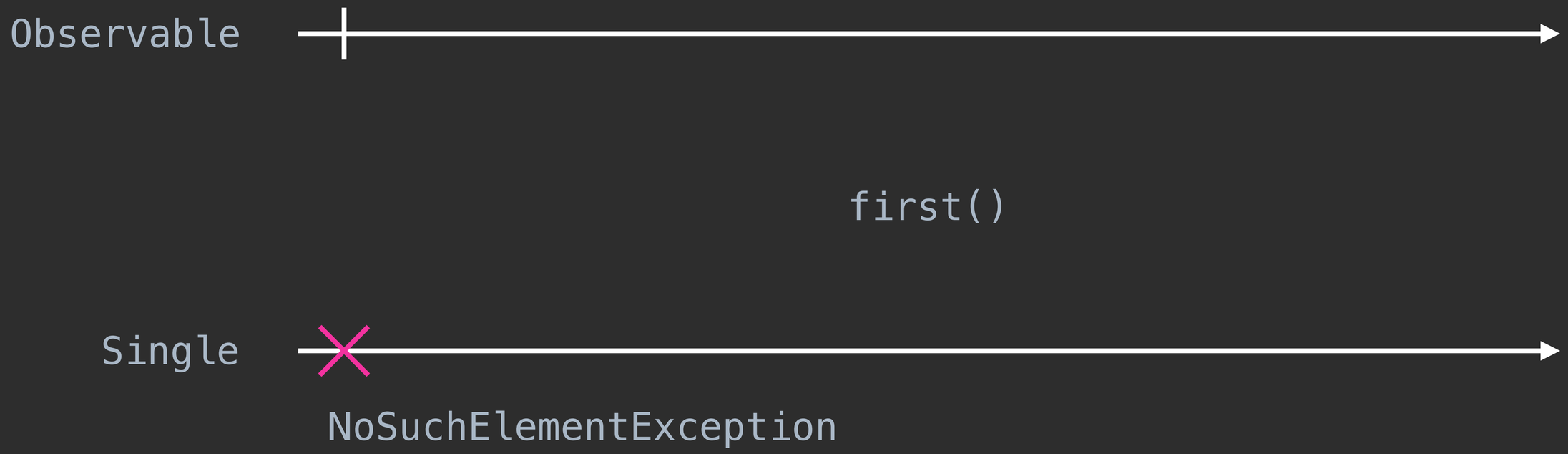


Operator Specialization

Operator Specialization



Operator Specialization

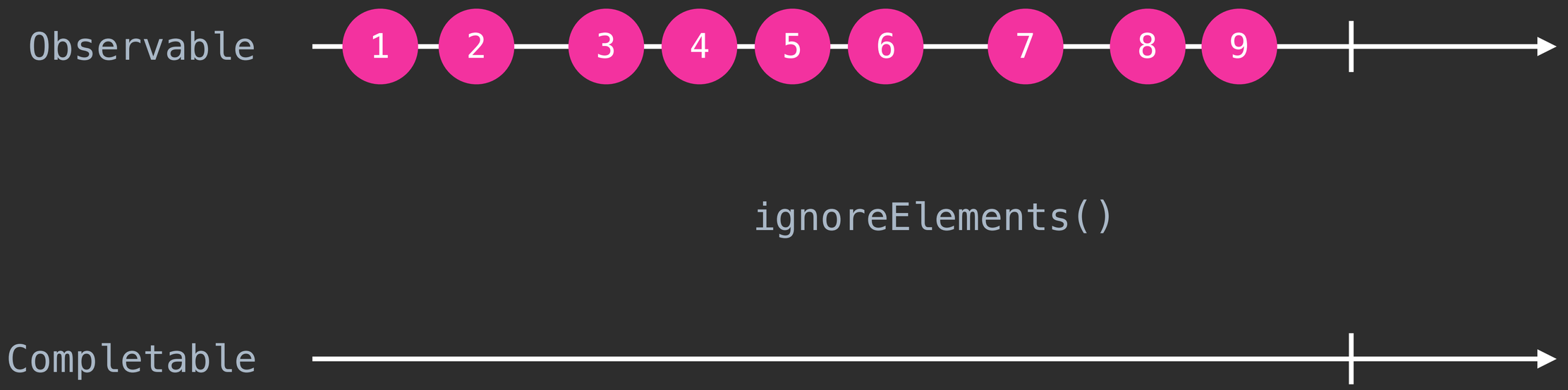


Operator Specialization

Operator Specialization

Operator Specialization

Operator Specialization



Operator Specialization

Operator Specialization

Operator Specialization

From \ To	Flowable	Observable	Maybe	Single	Completable
Flowable		toObservable()	reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrError() last()/lastOrError() single/singleOrError() all()/any()/count() (and more)	ignoreElements()
Observable	toFlowable()		reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrError() last()/lastOrError() single/singleOrError() all()/any()/count() (and more)	ignoreElements()
Maybe	toFlowable()	toObservable()		toSingle() sequenceEqual()	toCompletable()
Single	toFlowable()	toObservable()	toMaybe()		toCompletable()
Completable	toFlowable()	toObservable()	toMaybe()	toSingle() toSingleDefault()	

From \ To	Flowable	Observable	Maybe	Single	Completable
Flowable		toObservable()	reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrError() last()/lastOrError() single/singleOrError() all()/any()/count() (and more)	ignoreElements()
Observable	toFlowable()		reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrError() last()/lastOrError() single/singleOrError() all()/any()/count() (and more)	ignoreElements()
Maybe	toFlowable()	toObservable()		toSingle() sequenceEqual()	toCompletable()
Single	toFlowable()	toObservable()	toMaybe()		toCompletable()
Completable	toFlowable()	toObservable()	toMaybe()	toSingle() toSingleDefault()	

From \ To	Flowable	Observable	Maybe	Single	Completable
Flowable		toObservable()	reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrError() last()/lastOrError() single/singleOrError() all()/any()/count() (and more)	ignoreElements()
Observable	toFlowable()		reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrError() last()/lastOrError() single/singleOrError() all()/any()/count() (and more)	ignoreElements()
Maybe	toFlowable()	toObservable()		toSingle() sequenceEqual()	toCompletable()
Single	toFlowable()	toObservable()	toMaybe()		toCompletable()
Completable	toFlowable()	toObservable()	toMaybe()	toSingle() toSingleDefault()	

Being Reactive

```
um.getUser()
```

Being Reactive

```
um.getUser()
  .observeOn(AndroidSchedulers.mainThread())
```

Being Reactive

```
um.getUser()
  .observeOn(AndroidSchedulers.mainThread())
  .subscribeWith(new DisposableObserver<User>() {
    @Override public void onNext(User user) {
    }
    @Override public void onComplete() { /* ignored */ }
    @Override public void onError(Throwable t) { /* crash or show */ }
  });
```

Being Reactive

```
um.getUser()
  .observeOn(AndroidSchedulers.mainThread())
  .subscribeWith(new DisposableObserver<User>() {
    @Override public void onNext(User user) {
      tv.setText(user.toString());
    }
    @Override public void onComplete() { /* ignored */ }
    @Override public void onError(Throwable t) { /* crash or show */ }
  });
```

Being Reactive

```
disposables.add(um.getUser()
  .observeOn(AndroidSchedulers.mainThread())
  .subscribeWith(new DisposableObserver<User>() {
    @Override public void onNext(User user) {
      tv.setText(user.toString());
    }
    @Override public void onComplete() { /* ignored */ }
    @Override public void onError(Throwable t) { /* crash or show */ }
  }));
```

Being Reactive

```
// onCreate
disposables.add(um.getUser()
  .observeOn(AndroidSchedulers.mainThread())
  .subscribeWith(new DisposableObserver<User>() {
    @Override public void onNext(User user) {
      tv.setText(user.toString());
    }
    @Override public void onComplete() { /* ignored */ }
    @Override public void onError(Throwable t) { /* crash or show */ }
  }));

// onDestroy
disposables.dispose();
```

Being Reactive

```
um.setName("Jane Doe")
```

Being Reactive

```
um.setName("Jane Doe")
  .subscribeOn(Schedulers.io())
```

Being Reactive

```
um.setName("Jane Doe")
  .subscribeOn(Schedulers.io())
  .observeOn(AndroidSchedulers.mainThread())
```

Being Reactive

```
um.setName("Jane Doe")
  .subscribeOn(Schedulers.io())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribeWith(new DisposableCompletableObserver() {
    @Override public void onComplete() {
    }
    @Override public void onError(Throwable t) {
      // retry or show
    }
  });
```

Being Reactive

```
um.setName("Jane Doe")
  .subscribeOn(Schedulers.io())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribeWith(new DisposableCompletableObserver() {
    @Override public void onComplete() {
      // success! re-enable editing
    }
    @Override public void onError(Throwable t) {
      // retry or show
    }
  });
```

Being Reactive

```
disposables.add(um.setName("Jane Doe")
  .subscribeOn(Schedulers.io())
  .observeOn(AndroidSchedulers.mainThread())
  .subscribeWith(new DisposableCompletableObserver() {
    @Override public void onComplete() {
      // success! re-enable editing
    }
    @Override public void onError(Throwable t) {
      // retry or show
    }
  }));
```

Java 9

- JEP 266: More Concurrency Updates

Java 9

- JEP 266: More Concurrency Updates

> Interfaces supporting the Reactive Streams publish-subscribe framework, nested within the new class `Flow`, along with a utility class `SubmissionPublisher` that developers can use to create custom components. These (very small) interfaces correspond to those defined with broad participation (from the Reactive Streams initiative) and support interoperability across a number of async systems running on JVMs. Nesting the interfaces within a class is a conservative policy allowing their use across various short-term and long-term possibilities. The proposed `java.util.concurrent` components have been offered in pre-release since January 2015, and have benefitted from several rounds of review. There are no plans to provide network- or I/O-based `java.util.concurrent` components for distributed messaging, but it is possible that future JDK releases will include such APIs in other packages.

```
final class Flow {
  private Flow() {}

  interface Publisher<T> {
    void subscribe(Subscriber<? super T> subscriber);
  }

  interface Subscriber<T> {
    void onSubscribe(Subscription subscription);
    void onNext(T item);
    void onError(Throwable throwable);
    void onComplete();
  }

  interface Subscription {
    void request(long n);
    void cancel();
  }

  interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
  }
}
```


DEVOXX
UNITED STATES
Managing State
with RxJava
Jake Wharton
jakes.link/rx-state

Managing The Reactive World

twitter.com/ jakewharton

github.com/ jakewharton

jakewharton .com