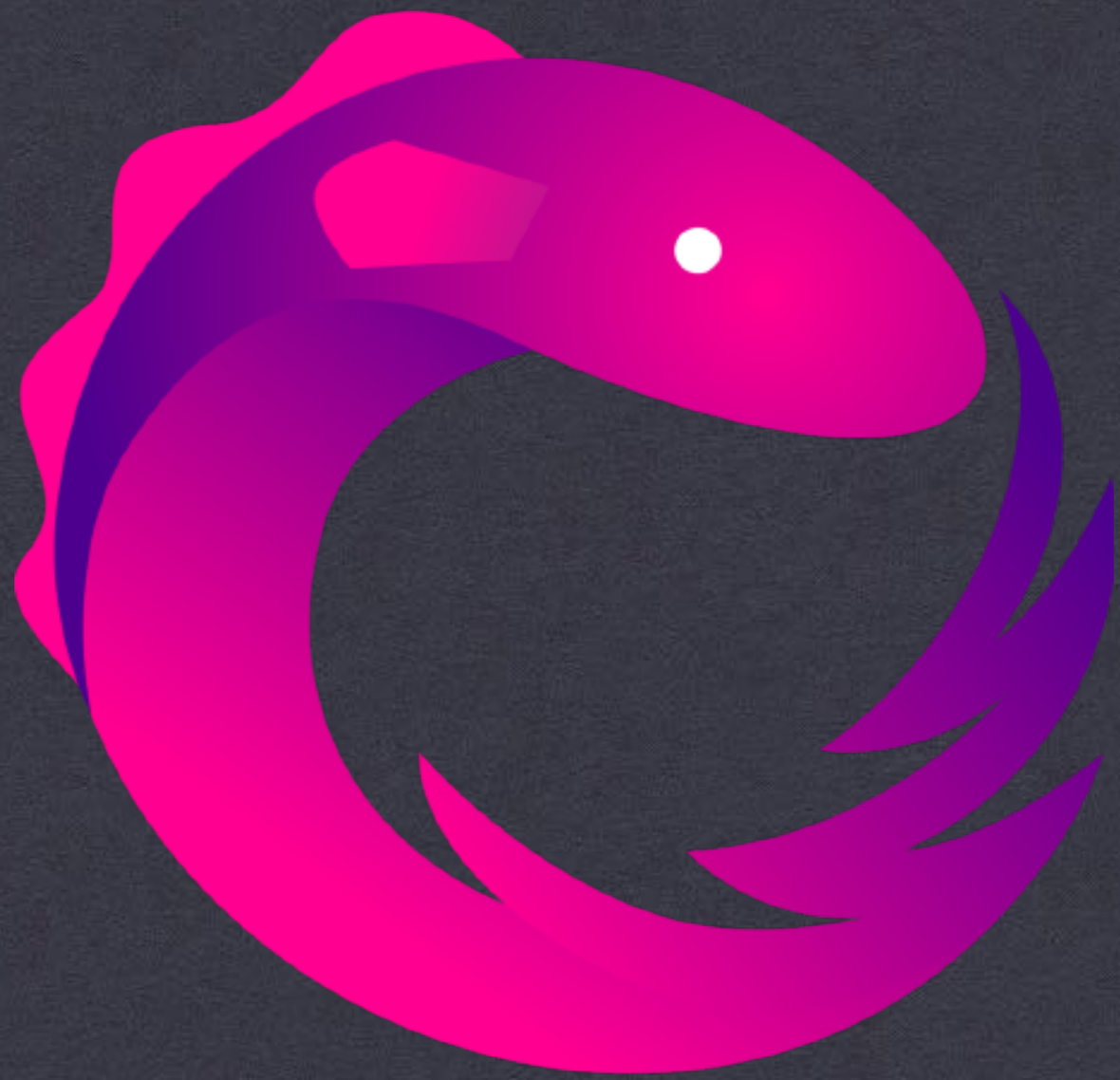
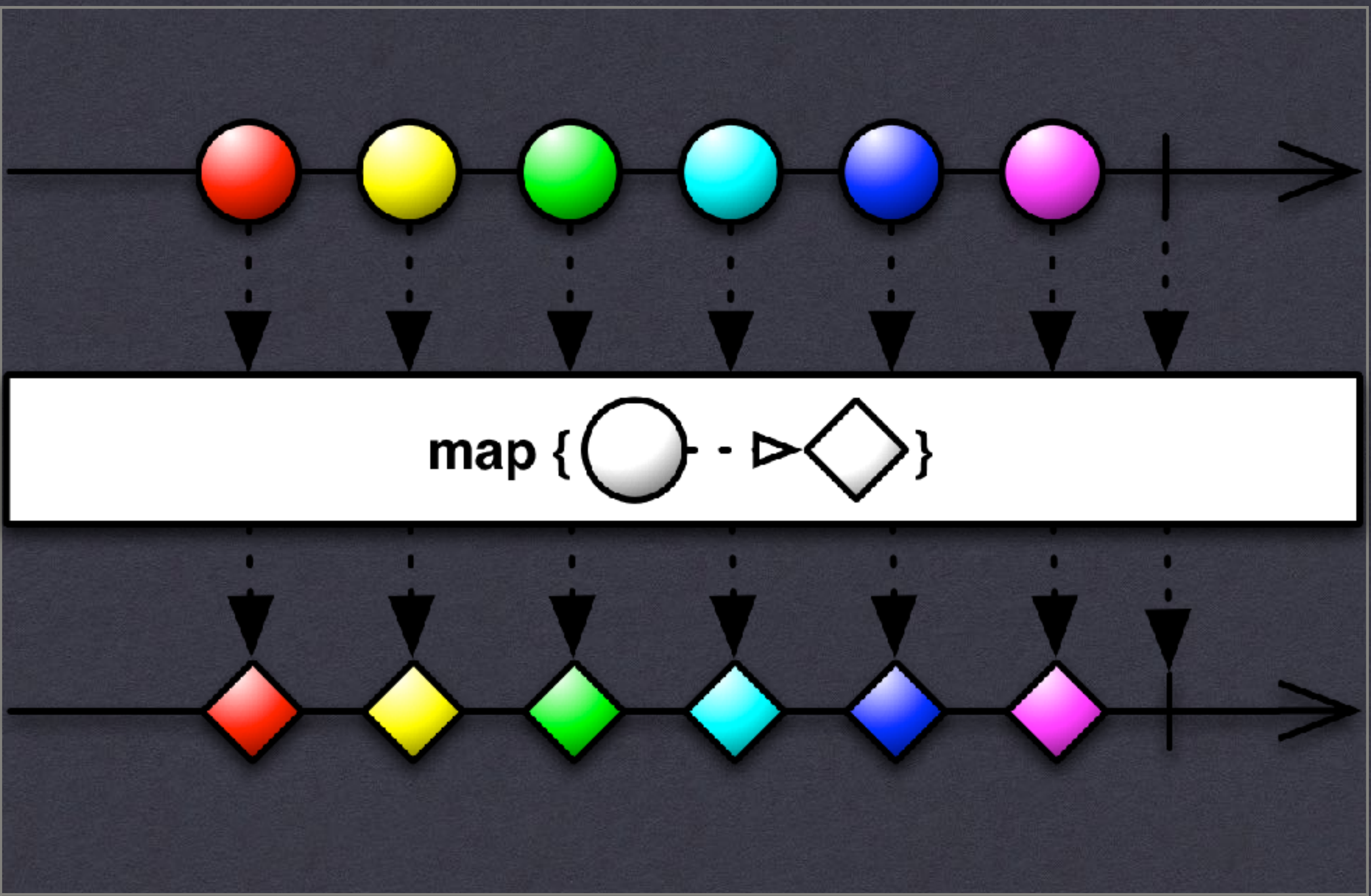


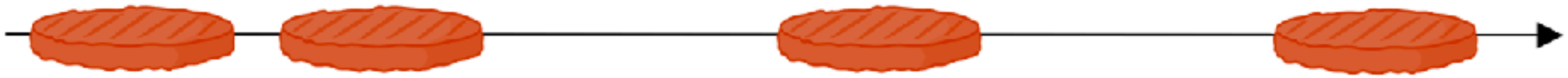
RXJAVA 2: REACTIVE REVOLUTION

JON HOLTAN



MARBLE DIAGRAM

```
filter(meat -> isMeatFresh(meat))
```



```
map(meat -> cook(meat))
```

REAL WORLD EXAMPLE

FRONTEND

CROSS-PLATFORM

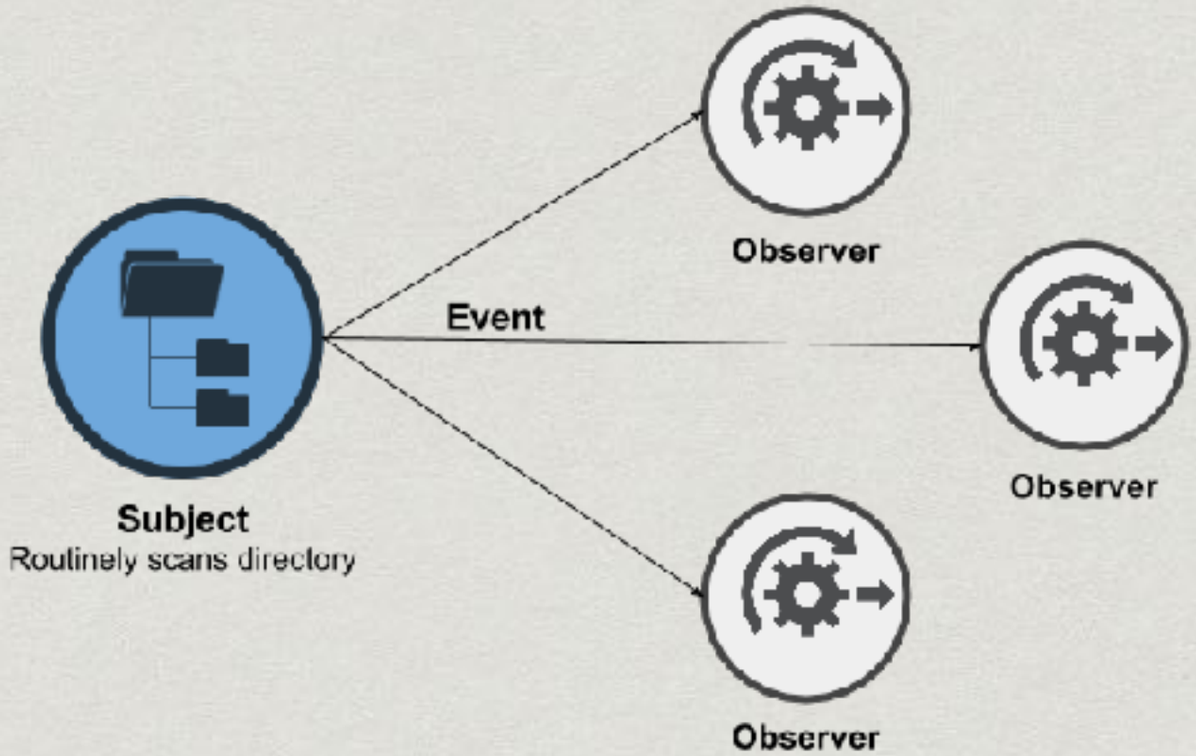
BACKEND

Why RxJava? Better Code!

- No Callback Hell
- Simple Error Handling
- Easy Multi-Threading
- Steep Learning Curve

Reactive Inspirations

- Observer Pattern
- Iterator Pattern
- Functional Programming



Basic Constructs

- A source **OBSERVABLE**
- One or more **OPERATORS**
- A target **SUBSCRIBER**

Variations

- Observable
- Flowable
- Single
- Completable
- Maybe

HOT & COLD OBSERVABLES

CONTRACT METHODS

Subscribe Contract

- Observer implements a subset of:
  - OnNext
  - OnError
  - OnCompleted

LANGUAGE COMPARISONS

.NET

```
IObservable<string> obj = Observable.Generate(
            0, //Sets the initial value like for loop
            _ => true, //Don't stop till i say so, infinite loop
            i => i + 1, //Increment the counter by 1 every time
            i => new string('#', i), //Append #
            i => TimeSelector(i));
        //Subscribe here
        using (obj.Subscribe(Console.WriteLine)) {
            Console.ReadLine();
        }
```

SWIFT

```
let helloSequence = Observable.from(["H","e","l","l","o"])
let subscription = helloSequence.subscribe { event in
  switch event {
      case .next(let value):
          print(value)
      case .error(let error):
          print(error)
      case .completed:
          print("completed")
  }
}
```

KOTLIN

```
val list = listOf("Alpha", "Beta", "Gamma", "Delta", “Epsilon")

    list.toObservable() // extension function for Iterables
            .filter { it.length >= 5 }
        .subscribeBy(  // named arguments for lambda Subscribers
                onNext = { println(it) },
                onError =  { it.printStackTrace() },
                onComplete = { println("Done!") }
        )
```

JS

```
const { Observable, Subject, ReplaySubject, from,
of, range } = require('rxjs');

const { map, filter, switchMap } = require('rxjs/
operators');

range(1, 200)

  .pipe(filter(x => x % 2 === 1), map(x => x + x))

  .subscribe(x => console.log(x));
```

THREADING

```
getBooks().subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribeWith(new DisposableObserver<Book>() {
            @Override
            public void onNext(@NonNull Book book) {
                // You can access your Book objects here
            }
            @Override
            public void onError(@NonNull Throwable e) {
                // Handler errors here
            }
            @Override
            public void onComplete() {
                // All your book objects have been fetched. Done!
            }
        });
```

Schedulers

- Schedulers.io()
- Schedulers.computation()
- Schedulers.newThread()
- Schedulers.single()
- Schedulers.from(Executor executor)
- AndroidSchedulers.mainThread()

*On()

- SubscribeOn
- ObserveOn

```
Observable.just(1, 2, 3, 4, 5, 6)
        .subscribeOn(Schedulers.co
mputation())
        .doOnNext({ println("Emitt
ing item $it on: $
{currentThread().name}") })
        .observeOn(AndroidSchedule
rs.mainThread())
        .subscribe({
println("Consuming item $it" })
```

```
getIntegersFromRemoteSource()
        .doOnNext(integer -> println("Emitting item " +
integer + " on: " + currentThread().getName()))
        .subscribeOn(Schedulers.io())
        .observeOn(Schedulers.computation())
        .map(integer -> {
            println("Mapping item " + integer + " on: " +
currentThread().getName());
            return integer * integer;
        })
        .observeOn(Schedulers.newThread())
        .filter(integer -> {
            println("Filtering item " + integer + " on: " +
currentThread().getName());
            return integer % 2 == 0;
        })
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(integer -> println("Consuming item " +
integer + " on: " + currentThread().getName()));
```

TESTING

Pre-Packaged

- Built in test friendly solutions
- TestScheduler to give you complete control
- TestObserver to verify Observables

Helpful Links

- Multi-Threading Like A Boss
- ReactiveX: Observable
- ReactiveX: Examples
- Reactive Streams
- Understanding Marble Diagrams for Rx Streams