Reactive programming

by Romans

What is it?

Programming paradigm where data (I/O) is treated as a bunch of streams

- Centered around Observable/Observer/Scheduler

Why?

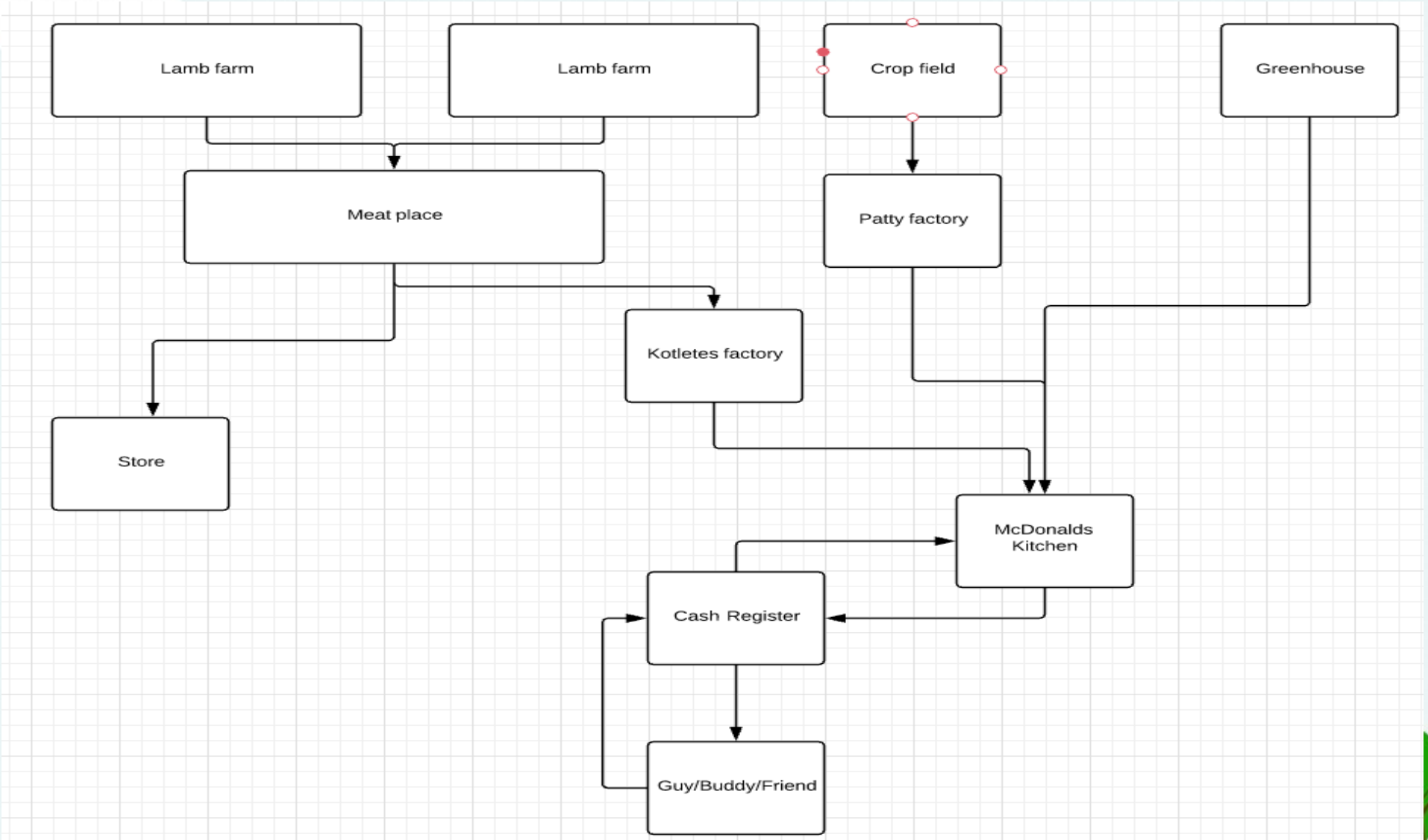
- Why not?
- Hype-driven development
- Threading
- Error handling
- Observe events as you go

Simple example

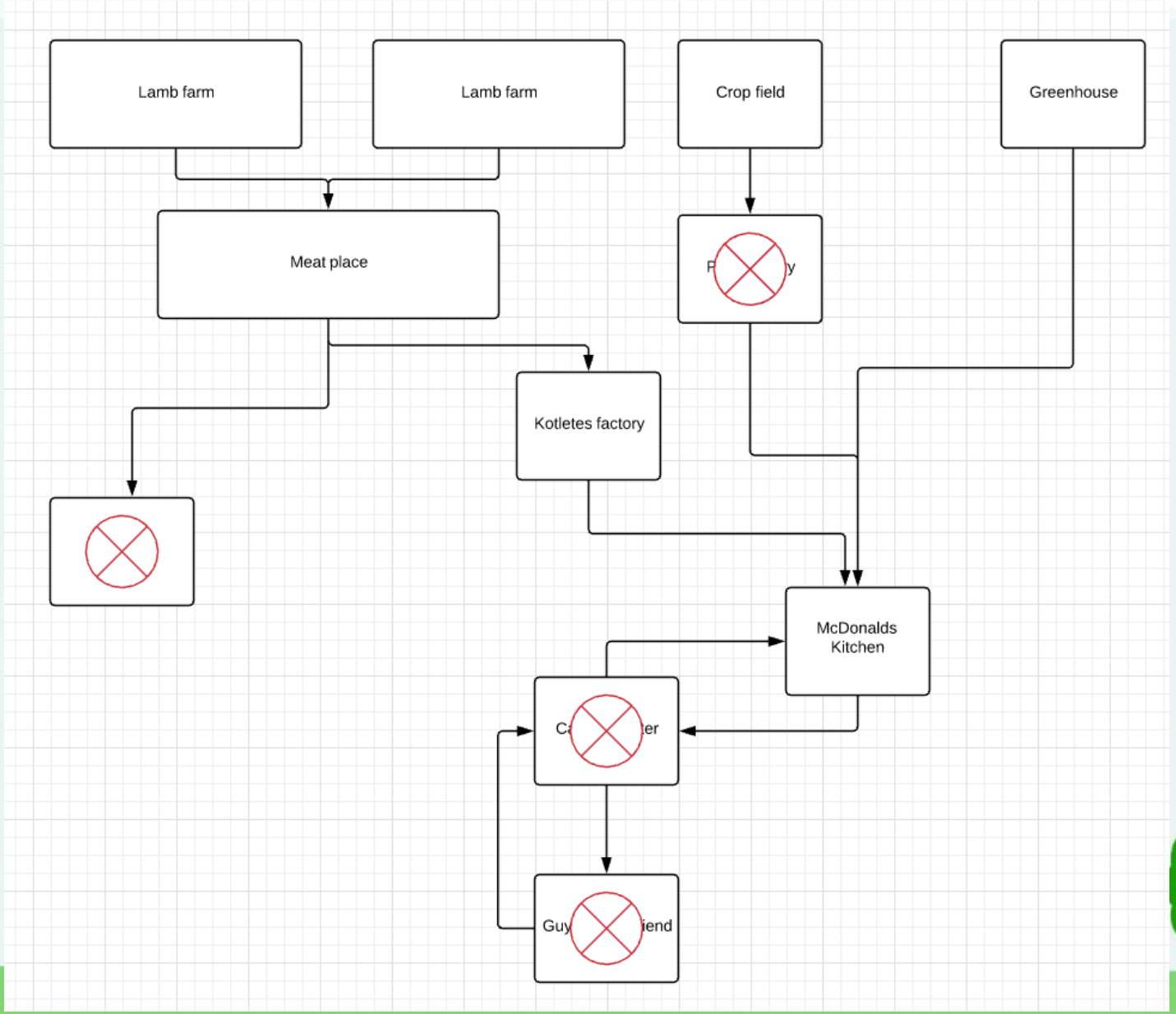
```
public class Ex1 {

    private static Observable<Integer> observable = Observable.just( item1: 1,  item2: 2,  item3: 3);

    public static void main(String[] args) throws InterruptedException {

        observable.subscribeOn(Schedulers.computation())
                .doOnNext(e -> System.out.println(
                        String.format("Emiting %s on %s", e, Thread.currentThread().getName())))
                .observeOn(Schedulers.computation())
                .subscribe(e ->
                        System.out.println(
                                String.format("Receiving %s on %s", e, Thread.currentThread().getName())),
                        e -> { },
                        () -> System.out.println(String.format("Completed on %s", Thread.currentThread().getName())));
        Thread.sleep( millis: 100);
    }
}
```

```
Emiting 1 on RxComputationThreadPool-1
Emiting 2 on RxComputationThreadPool-1
Emiting 3 on RxComputationThreadPool-1
Receiving 1 on RxComputationThreadPool-2
Receiving 2 on RxComputationThreadPool-2
Receiving 3 on RxComputationThreadPool-2
Completed on RxComputationThreadPool-2
```

Simple Reactive app



Simple Reactive app



Lamb Farms

```
static Observable<Lamb> farm1 =
        Observable.interval( period: 1, TimeUnit.SECONDS, Threader.scheduler( name: "Lamb Farm #1"))
                .map(e -> new Lamb( farm: "Farm #1"));
static Observable<Lamb> farm2 =
        Observable.interval( period: 1500, TimeUnit.MILLISECONDS, Threader.scheduler( name: "Lamb Farm #2"))
                .map(e -> new Lamb( farm: "Farm #2"));
```

```
Observable<List<Lamb>> farmResult = farm1.mergeWith(farm2).buffer(2);

long startTime = System.currentTimeMillis();

farmResult.subscribe(moos -> System.out.println(String.format("Received %s Mooos after %s", moos.size(),
        System.currentTimeMillis() - startTime)));
```

```
Received 2 Mooos after 1520
Received 2 Mooos after 3008
Received 2 Mooos after 4008
Received 2 Mooos after 5008
Received 2 Mooos after 6020
Received 2 Mooos after 7519
Received 2 Mooos after 9008
Received 2 Mooos after 10008
Received 2 Mooos after 11007
Received 2 Mooos after 12020
Received 2 Mooos after 13520
```

Lamb to kotlete

```
Observable<List<Kotlete>> kotleteSupply =
        farmResult.map(meatPlace::process)
                .flatMapSingle(kotletesPlace::process);
```

```
static class MeatPlace {
    List<RawMeat> process(List<Lamb> lamb) {
        System.out.println(String.format("Received meat on %s from %s", Threader.threadName(), lamb.st
        return IntStream.of(10).mapToObj(i -> new RawMeat()).collect(Collectors.toList());
    }
}
```

```
static class KotletesPlace {

    private Scheduler kotletesPlaceScheduler = Threader.scheduler( name: "Kotletes place");

    Single<List<Kotlete>> process(List<RawMeat> meatMMM) {
        return Single.just(meatMMM)
                .zipWith(Single.timer(new Random().nextInt( bound: 4), TimeUnit.SECONDS, kotletesP
                        (kotlete, time) -> generateKotletes())
                .subscribeOn(kotletesPlaceScheduler);

    }

    private List<Kotlete> generateKotletes() {
        System.out.println(String.format("Starting to cook Kotletes on %s", Threader.threadName()
        return IntStream.of(new Random().nextInt( bound: 10)).mapToObj(i -> new Kotlete()).collect
    }
}
```

Other burger supplies

```
static Observable<List<Patty>> cropField =
        Observable.interval( period: 100, TimeUnit.MILLISECONDS, Threader.scheduler( name: "Patty Farm"))
        .map(e -> new Patty()).buffer(13);

static Observable<List<Tomato>> tomatoGreenHouse =
        Observable.interval( period: 140, TimeUnit.MILLISECONDS, Threader.scheduler( name: "Greenhouse"))
                .map(e -> new Tomato()).buffer(8);
```

```
public static void main(String[] args) throws InterruptedException {

    Observable<List<Lamb>> farmResult = farm1.mergeWith(farm2).buffer(2);


    Observable<List<Kotlete>> kotleteSupply =
            farmResult.map(meatPlace::process)
                    .flatMapSingle(kotletesPlace::process);

    Observable.zip(kotleteSupply, cropField, tomatoGreenHouse, mcDonalds::process)
            .observeOn(Threader.scheduler( name: "Cashier"))
            .subscribe(e -> System.out.println(String.format("Received burger on %s", Threader.threadName())));
    Thread.sleep( millis: 100000);
}
```

Operators

- flatMap
- concatMap
- zip
- just
- timer
- combineLatest
- debounce
- retry
- Map
- create

Fail-safe network service

```
public static void main(String[] args) {
    Single<String> callOtherServiceOrDefault = networkRequest()
            .ambWith(Single.timer( delay: 2000, TimeUnit.MILLISECONDS, local).map(i -> "Returned local response"));

    Observable.range( start: 0,  count: 10)
            .flatMapSingle(i -> callOtherServiceOrDefault).blockingSubscribe(System.out::println);

}

static Scheduler network = Threader.scheduler( name: "Generic network");
static Scheduler local = Threader.scheduler( name: "Generic local I/O");

private static Single<String> networkRequest() {
    return Single.fromCallable(() -> new Random().nextInt( bound: 9) * 500 + 500)
            .flatMap(delay -> Single.timer(delay, TimeUnit.MILLISECONDS, network).map(e -> {
                if (new Random().nextInt( bound: 9) < 4) {
                    throw new IllegalArgumentException("Bad request");
                }
                return "Received 200 OK";
            })).onErrorReturn(e -> "Unknown response");
}
```

Thanks

- https://github.com/romansbobans/CitadeleReactive