REACTIVE WITH RXJAVA

WHAT IS REACTIVE

- Remember swing events? Windows event loop? Know about UI thread?
- They look deceivingly similar to Java 8 streams but are not.
- Reactive deals with data as events. And events as data.
- Very subtle difference with respect to Functional Reactive Programming
  - FRP is about dynamic state management
  - Reactive is mostly stateless

SETUP

```
<dependency>
    <groupId>io.reactivex.rxjava2</groupId>
    <artifactId>rxjava</artifactId>
    <version>2.1.14</version>
</dependency>

<dependency>
    <groupId>com.github.davidmoten</groupId>
    <artifactId>rxjava2-jdbc</artifactId>
    <version>0.1-RC35</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.11</version>
</dependency>

Observable.fromArray(strs).subscribe(s -> System.out.println(s));
```

BASICS

- Basic type is an Observable
- An Observable<T> pushes objects of type T thru a series of operators till it arrives at an Observer
- Major difference: Observable pushes items, while the others pull

```
String[] strs = new String[]{"str1", "str2", "strs3"};

for(String s: strs) {
    System.out.println(s);
}


Stream.of(strs).forEach(s -> System.out.println(s));


Observable.fromArray(strs).subscribe(s -> System.out.println(s));
```

EVENTS

- Github repo: https://github.com/maruthir/training
- Generate events with Observable.interval and listen to them (Ex_1)

```
Observable.interval(1, TimeUnit.SECONDS)
```

  - Print the thread name in the subscriber

- Use Observable.create to create an observable from scratch (Ex_1)

  - Raise error and complete events

```
Observable<Long> sourceObs = Observable.create(e -> {
    e.onNext(3L);
    e.onNext(9L);
    e.onNext(5L);
    //e.onError(new Exception());
    e.onComplete();
});
```

OBSERVER

- Create a subscribe() with new Observer() (Ex_2)

```
sourceObs.subscribe(new Observer<Long>() {…
```

- Look at all the over loaded subscribers with Consumer parameters - use lambdas to do the same thing as Observer()

USING DB AS OBSERVABLE

- (Ex_3)

```
Database db = Database.from("jdbc:mysql://root:admin123@localhost:3306/
training", 5);

db.select("select name from users")
        .getAs(String.class)
        .subscribe(s ->
System.out.println(Thread.currentThread().getName() + ":" + s));
```

COLD OBSERVABLES

- Read records using observable. Prefix all names with “Mr.” (Ex_4)
- Now make an update to one of the names based on ID
- Create another subscription to print names
- Call db.close() when done
- What are the observations

HOT OBSERVABLES

- Get a ConnectableObservable (Ex_5)

```
ConnectableFlowable<Tuple2<String, String>> rows = db.select("select name, email_id from
users")
        .getAs(String.class, String.class).publish();
```

- blockingSubscribe once
- make an update using blockingSubscribe
- blockingSubscribe again
- call rows.connect
- Change blocking subscribes in selects to normal subscribes
- Understand publish().autoConnect and publish().refCount (Ex_6)

HTTP WITH REACTIVE

- Clone and import project into IDE: https://github.com/ReactiveX/RxNetty.git
- Run io.reactivex.netty.examples.http.helloworld.HelloWorldServer
- Understand code
- Can we hook up the earlier DB access to http now? (Now we run into a problem here that netty wants RxJava and DB library is RxJava2)

TRANSACTED DB CALLS

- Use Observable.fromCallable to create a callable class that is initialised with a connection and updates name in the DB table based on ID (Ex_7)
- **db.connection()** Gives us a connection. Use this to init the callable and after the update is over, subscribe to the “hot” row of Ex_6
- Log Thread info in each of the operations

ASYNC HTTP WORK

- Observable.empty will be a way to build empty responses
- Build async request processing to update DB records extending the previous work using callables (Ex_8)
- Log messages after request processing and db updates including thread names

HANDLING ERRORS WITH CALLABLES

- Modify http server to process 2 numbers, divide and return result. An url like this: http://localhost:65134/?numbers=4,0 (Ex_9)

```
Observable compute = Observable.just(numer/denom);
return rx.Observable.create(e -> {
    compute.subscribe(res -> e.onNext("" + res));
    e.onCompleted();
});
```

- Improve this so that if there is an error such as div by zero, it sends out a 0

OBSERVABLES FROM FILES

- Dump the contents of a file to http response in a reactive way (Ex_10)
  - Who controls the rendering speed? and who controls the file reading speed?

BASIC OPERATORS

- Can we print each line with a
 using “map” operator (Ex_10)
- And prefix with html and body tags with “startWith” operator
- Cut out lines that are less than 50 chars with “filter”

MORE OPERATORS

- DB has two additional tables - sales, sales2 with these columns: id, month, sales (Ex_11)
- Print sequence of Months in each table
- Print sequence of cumulative sales values with scan operator
- Join months in both tables using “mergeWith” operator and print
- Join months sequentially with “concatWith” operator
- concat sales from both tables and apply cumulative scan on it

MORE OPERATORS

- Use a reduce operation to find the cumulative sales (Ex_11)
- Make the allSales a hot flowable so that it does not repeat queries to DB
- Read names, email in a tuple from user table we had earlier - sort by name using sorted() operator

```
Flowable<Tuple2<String, String>> rows = db.select("select name, email_id from
users").getAs(String.class, String.class);
```

- Pull up only distinct names using the distinct() operator.
    - Ponder over the behaviour of sorted/distinct and performance

COMPUTE ACTIVITY

- Find a bunch of PNG files and resize them. Resize logic (Ex_13)

```
READ:
BufferedImage originalImage = ImageIO.read(new File("c:\\image\\my.png"));

RESIZE:
int type = originalImage.getType() == 0? BufferedImage.TYPE_INT_ARGB : originalImage.getType();
BufferedImage resizeImageJpg = resizeImage(originalImage, type);

WRITE:
ImageIO.write(resizeImageJpg, "png", new File("c:\\image\\my_small.png"));

private static BufferedImage resizeImage(BufferedImage originalImage, int type) {
    BufferedImage resizedImage = new BufferedImage(IMG_WIDTH, IMG_HEIGHT, type);
    Graphics2D g = resizedImage.createGraphics();
    g.drawImage(originalImage, 0, 0, IMG_WIDTH, IMG_HEIGHT, null);
    g.dispose();
    return resizedImage;
}
```

- Implement with java streams (No loops)
- Implement with Rx
- Understand subscribeOn and observeOn

FLATMAP

- Take a file and make a stream of lines from it (Ex_15)
- Starting from this stream print individual words on a separate line without looping

PARALLELISATION

- Each observer method is guaranteed to be called on a single thread as part of the reactive context. So how do we run a subscription processing in multiple threads? (Ex_16)
- Flatmap to the rescue -> allows each event to be converted to an observable and subscribed on a separate thread.

```
paths.filter(p -> p.getFileName().toString().toLowerCase().contains(IMG_EXTN))
        .flatMap(p -> Observable.just(p).subscribeOn(Schedulers.io()).map(path -> {
            try {
               return ImageIO.read(path.toFile());
            } catch (IOException e) {
                e.printStackTrace();
                return null;
            }
        }))
```

MISMATCHED SENDER/RECEIVER

- Sender and receiver can be mismatched if they run on different threads. This just causes Rx's Cache to get filled up eventually leading to out of memory. Example (Ex_17):

```
Observable.range(1, 10000)
        .doOnNext(i -> System.out.println("Sending item: " + i))
        .observeOn(Schedulers.io())
        .subscribe(i -> {
            TimeUnit.MILLISECONDS.sleep(20);
            System.out.println("Processed item: " + i);
        });
```

- What happens when Observable is replaced with Flowable?
- Back pressure is a protocol defined across a Rx chain

CUSTOM FLOWABLE WITH BACKPRESSURE

- Approach 1: use a built in backpressure strategy (Ex_18)

```
Flowable.create(e -> {
    for(int i=0; i< 10000; i++) {
        if (e.isCancelled()) return;
        e.onNext(i);
    }
    e.onComplete();
}, BackpressureStrategy.LATEST)
```

- Approach 2: make the source really backpressured. Read a file and write contents to DB (or just delay n print) in backpressured way

```
Flowable.generate(() -> lineReader.readLine(), (prevLine, e) -> {
    String line = lineReader.readLine();
    if (line != null) {
        e.onNext(line);
    } else {
        lineReader.close();
        e.onComplete();
    }
})
```

DISPOSING

```
Observable<String> linesFromFile = Observable.create(e -> {
    LineNumberReader lineReader = new LineNumberReader(
            new InputStreamReader(new FileInputStream("/users/maruthir/scripts.log")));
    String line;
    while((line = lineReader.readLine())!=null) {
        if (e.isDisposed()) {
            System.out.println("Disposing observable");
            lineReader.close();
            e.onComplete();
            return;
        }
        e.onNext(line);
    }
    lineReader.close();
    e.onComplete();
});
linesFromFile.subscribe(new Observer<String>() {
    Disposable disposable;

    public void onSubscribe(@NonNull Disposable disposable) {
        this.disposable = disposable;
    }

    public void onNext(@NonNull String s) {
        System.out.println(s);
        disposable.dispose();
    }

    public void onError(@NonNull Throwable throwable) {

    }

    public void onComplete() {

    }
});
```

- Bufferring

```
Observable.range(1,100).buffer(10).subscribe(System.out::println);
SomeLiveObservable.buffer(1, TimeUnit.SECONDS).subscribe(System.out::println);
```

- Windowing
  - Same as buffering - returns one observable for each group

```
Observable.range(1,100).window(10).subscribe(obs -> obs.subscribe(System.out::print));
Observable.range(1,100).window(10).flatMap(obs -> obs).subscribe(System.out::println);
```

- Throttling

```
Observable.interval(200, TimeUnit.MILLISECONDS).throttleLast(1, TimeUnit.SECONDS)
        .subscribe(System.out::println);
```

C10K PROBLEM

- What is scalability
- 10000 Concurrent Connections to a server - hard to achieve with traditional thread model
  - What are we going to do with this many connections? Is our processing so trivial?
  - What bottlenecks exist outside the system
  - How much of IO is wrapped-blocking?

REACTIVE IN PRACTICE

- Reactive will make very little difference in most apps because the bottleneck is usually the DB layer
  - We spend a lot of effort trying to scale DB before we even get to app layer
- Reactive programming is not natural and is hard
  - Simple sequential algorithms get complicated
  - Transaction propagation is hard to achieve
  - It makes sense when you can have end to end reactive
- Frameworks like Vert.x, Spring reactive make life easier in regular usecases

SPRING REACTIVE WITH RXJAVA

```
@RestController
public class RxJavaController {

    @Autowired
    private final RxJavaService aService;

    @RequestMapping(path = "/handleMessageRxJava", method = RequestMethod.POST)
    public Observable<MessageAcknowledgement> handleMessage(@RequestBody Message message) {
        return this.aService.handleMessage(message);
    }

}
```