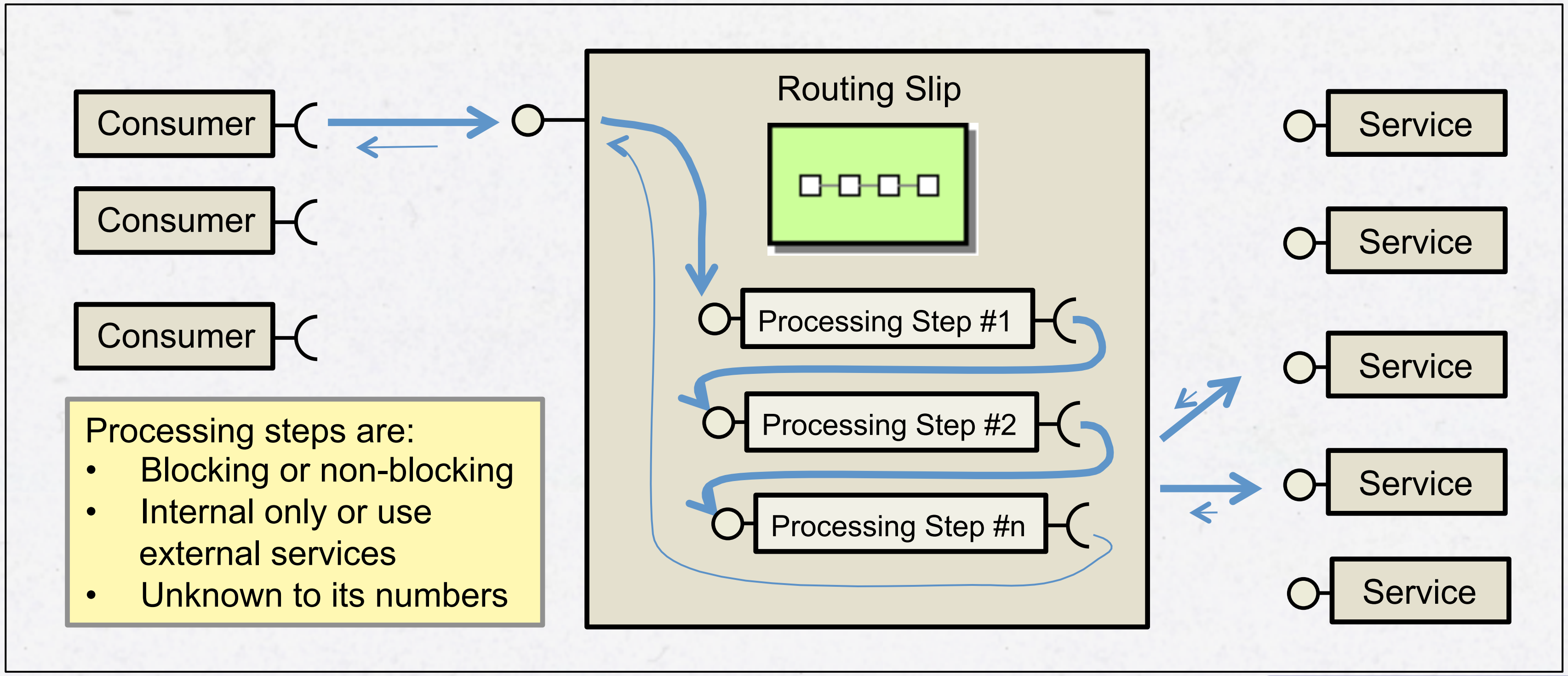
THEORY FOR EXERCISE #3 - REACTIVE PROGRAMMING WITH RX JAVA

- The road to Callback-hell…
- Introduction to RxJava
- Routing Slip – RxJava edition

THEORY FOR EXERCISE #3 - THE ROAD TO CALLBACK-HELL

- How to implement a routing slip pattern using non-blocking I/O?



Processing steps are:
- Blocking or non-blocking
- Internal only or use external services
- Unknown to its numbers

WHERE TO INITIATE THE NEXT PROCESSING STEP?

```
@RequestMapping("/router")
public DeferredResult<ResponseEntity<String>> router(@RequestParam String qry) {

    final DeferredResult<ResponseEntity<String>> dr = new DeferredResult<>();

    asyncHttpClient.execute(url + "/service?qry=" + qry,

        throwable -> {
                util.handleException(throwable, url, dr);
        },

        response -> {
            ...
        }
    );

    // Return to let go of the precious thread we are holding on to...
    return dr;
}
```

IT HAS TO GO INTO THE CALLBACK METHOD!

```
@RequestMapping("/router")
public DeferredResult<ResponseEntity<String>> router(@RequestParam String qry) {

    final DeferredResult<ResponseEntity<String>> dr = new DeferredResult<>();

    asyncHttpClient.execute(url + "/service?qry=" + qry,

        throwable -> {
                util.handleException(throwable, url, dr);
        },

        response -> {
            ...
        }
    );

    // Return to let go of the precious thread we are holding on to...
    return dr;
}
```

Here we can initiate the next processing step

CALLISTA — ENTERPRISE —

ENTERING "THE CALLBACK HELL"

```
// Send request #1
ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
  (Response r1) -> {
    processResult(r1.getResponseBody()); // Process response #1
    asyncHttpClient.execute(getUrl(2),   // Send request #2
      (Response r2) -> {
        processResult(r2.getResponseBody()); // Process response #2
        asyncHttpClient.execute(getUrl(3),   // Send request #3
          (Response r3) -> {
            processResult(r3.getResponseBody()); // Process response #3
            asyncHttpClient.execute(getUrl(4),   // Send request #4
              (Response r4) -> {
                processResult(r4.getResponseBody()); // Process response #4
                asyncHttpClient.execute(getUrl(5),   // Send request #5
                  (Response r5) -> {
                    processResult(r5.getResponseBody()); // Process response #5
                    // Get the total result and set it on the deferred result
                    dr.setResult(getTotalResult());
                    ...
```

ENTERING “THE CALLBACK HELL”

```
// Send request #1
ListenableFuture<Response> execute = asyncHttpClient.execute(getUrl(1),
  (Response r1) -> {
    processResult(r1.getResponseBody()); // Process response #1
    asyncHttpClient.execute(getUrl(2),   // Send request #2
      (Response r2) -> {
        processResult(r2.getResponseBody()); // Process response #2
        asyncHtt                               est #3
          (Respo
            proc                                 ss response #3
            asyn                                   request #4
              (Response r4) -> {
                processResult(r4.getResponseBody()); // Process response #4
                asyncHttpClient.execute(getUrl(5),   // Send request #5
                  (Response r5) -> {
                    processResult(r5.getResponseBody()); // Process response #5
                    // Get the total result and set it on the deferred result
                    dr.setResult(getTotalResult());
                    ...
```

A.k.a “Callback Hell”

ENTERING “THE CALLBACK HELL”

```
                                    ...
                                    dr.setResult(getTotalResult());
                                }
                            );
                        }
                    );
                }
            );
        }
    );

    return deferredResult;
}
```

THEORY FOR EXERCISE #3 - REACTIVE PROGRAMMING WITH RX JAVA

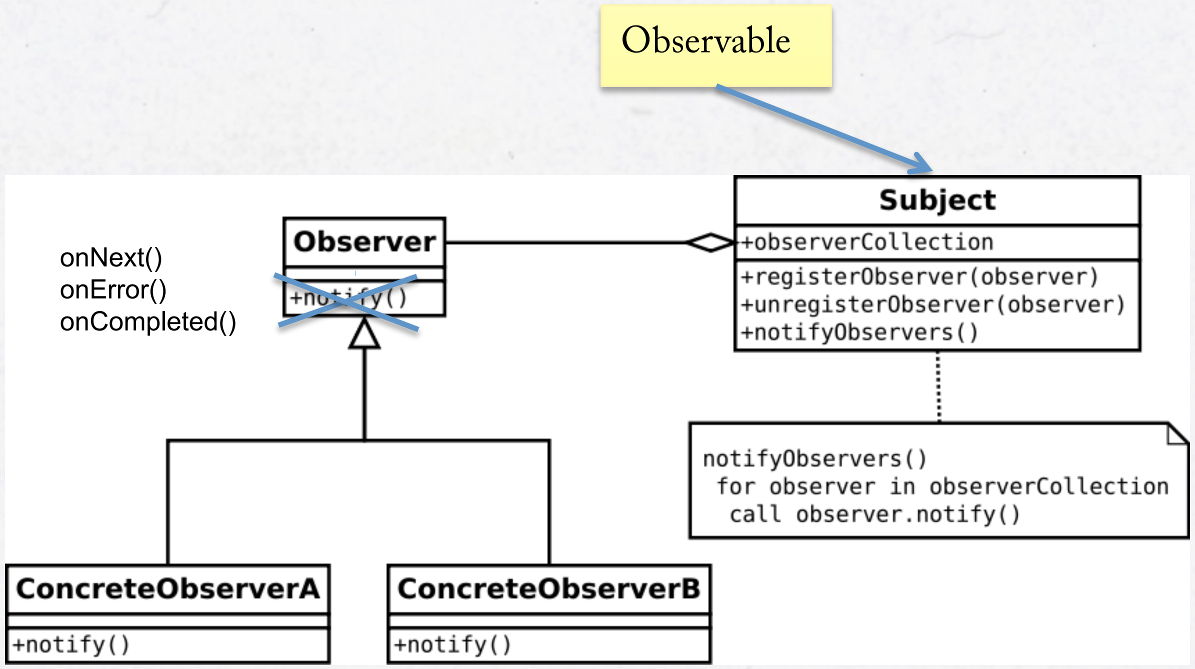
- The road to Callback-hell…
- Introduction to RxJava
- Routing Slip – RxJava edition

INTRODUCING RXJAVA

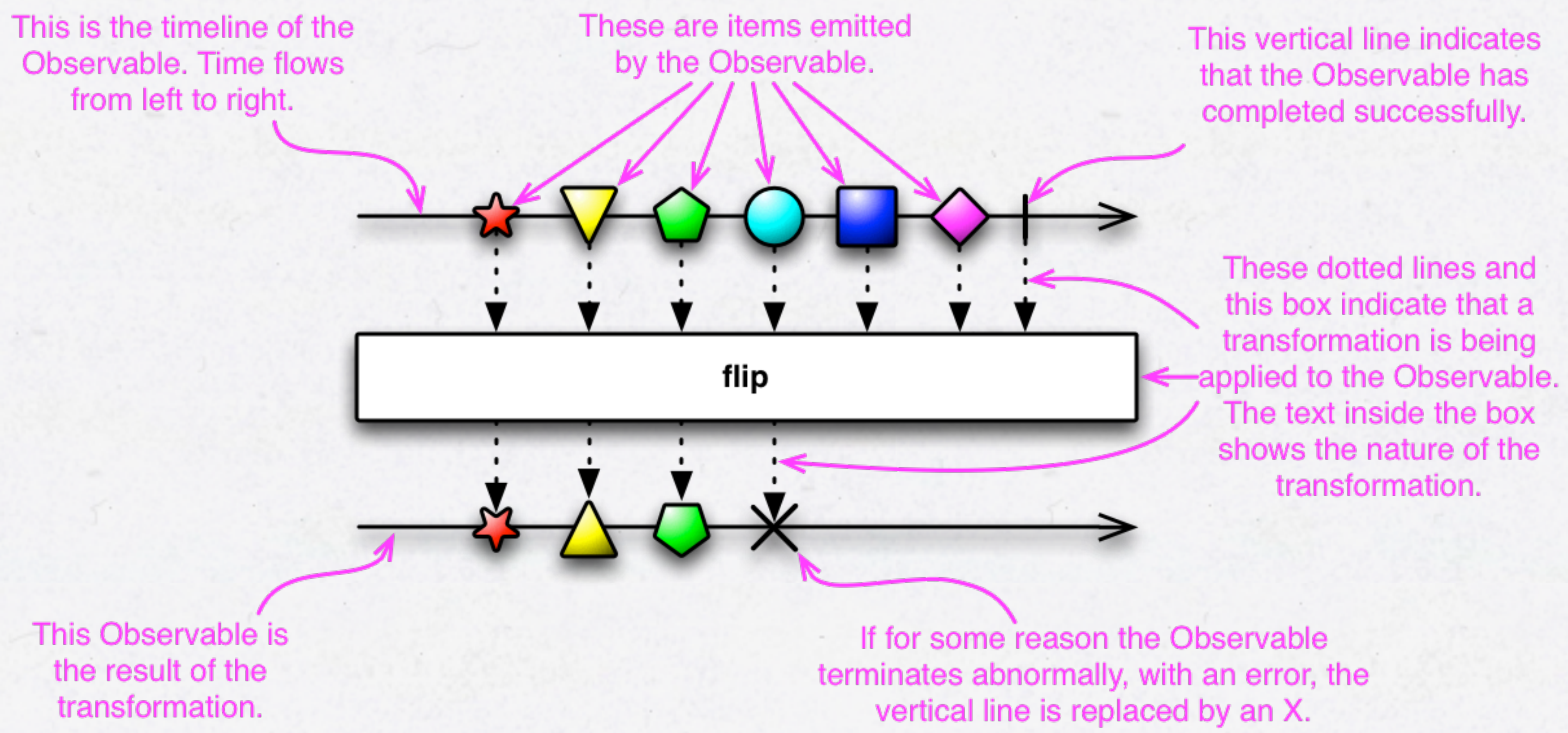
- A library for composing asynchronous and event-based programs
- Originated from Microsoft, https://rx.codeplex.com
  - Erik Meijer – the batik man
    - » http://vimeo.com/110554082, 4.45 – 5.40
- Netflix made the port to Java
- Several languages supported, http://reactivex.io
- Works on Java 7 (bye bye Lambdas…)
- Very clean API based on Observer/Observable
  - *“The Observer pattern done right!”*

OBSERVABLE

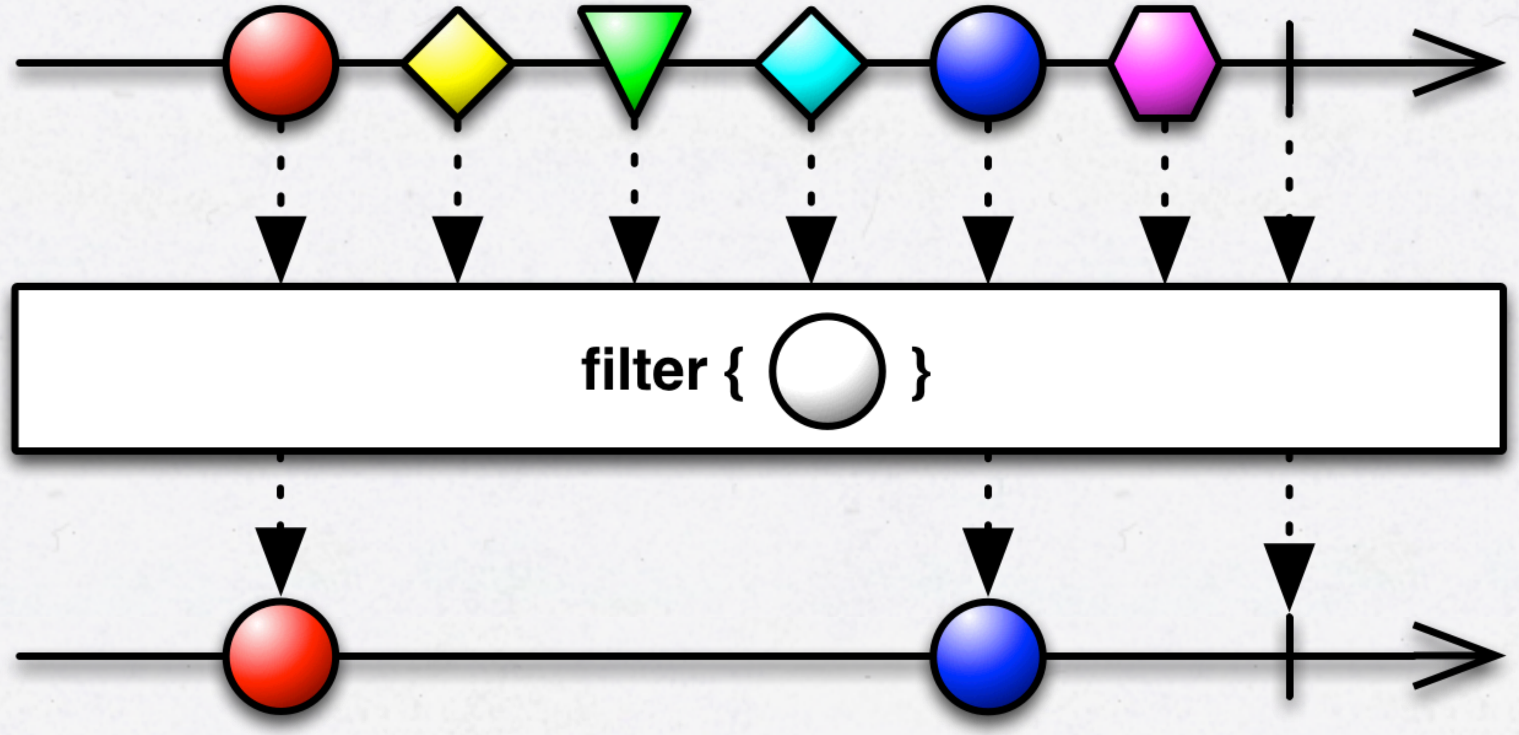
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


Observable
onNext()
onError()
onCompleted()
Observer
~~+notify()~~
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
notifyObservers()
for observer in observerCollection
call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

MARBLE DIAGRAMS (FROM RX JAVA DOCS)


This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
flip
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

FILTER FUNCTION

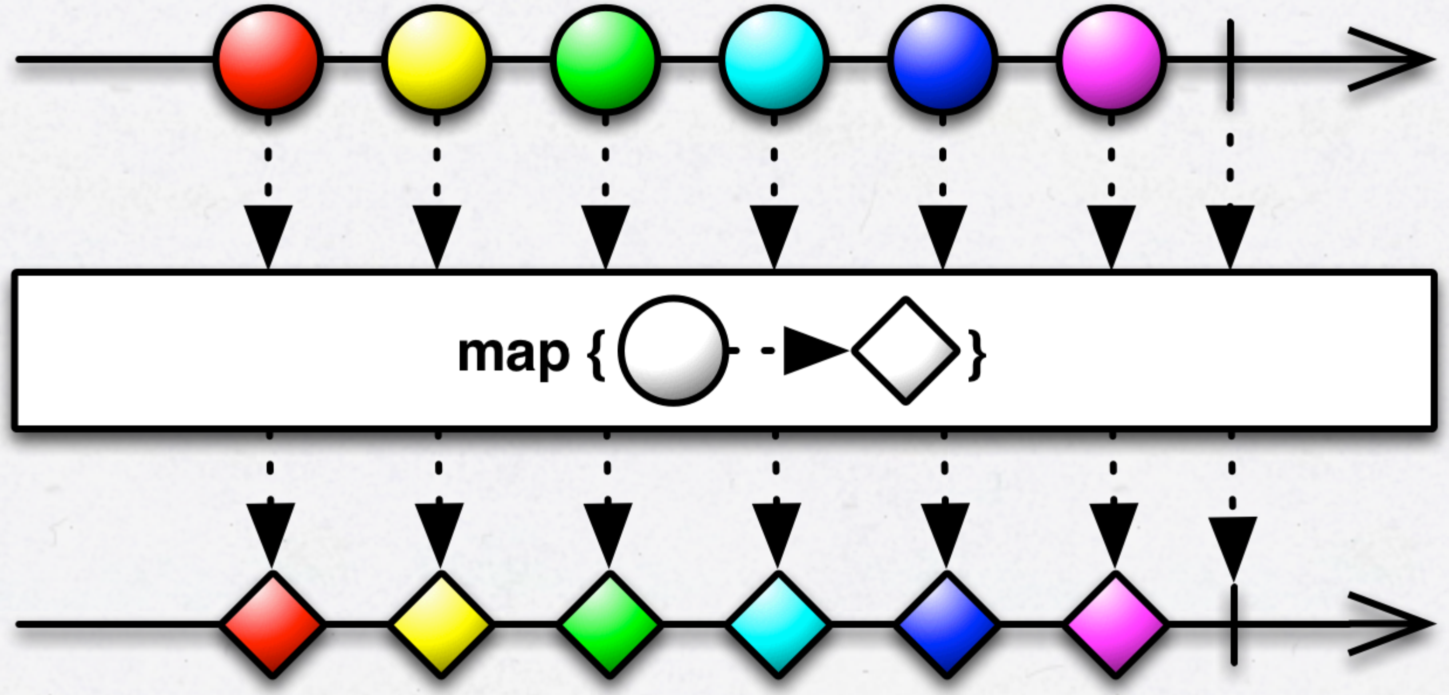


FILTER FUNCTION

```
Observable<Integer> o = Observable.range(1,10);

o.filter(s -> s % 3 == 0).subscribe(System.out::println);
```

```
3
6
9
```

MAP FUNCTION

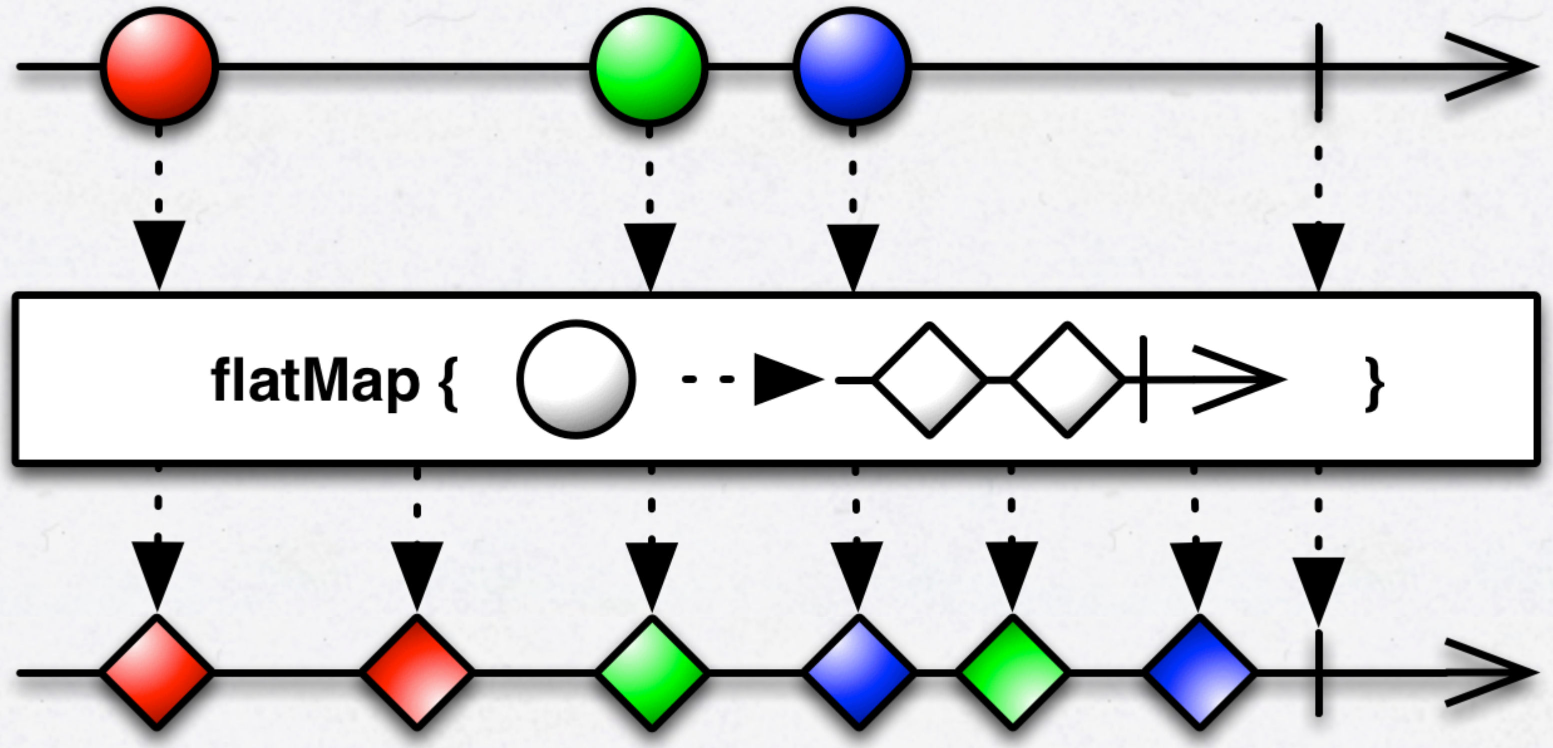


MAP FUNCTION

```java
Observable<Integer> o = Observable.range(1,5);

o.map(v -> v * 10).subscribe(System.out::println);
```

```
10
20
30
40
50
```

FLATMAP FUNCTION



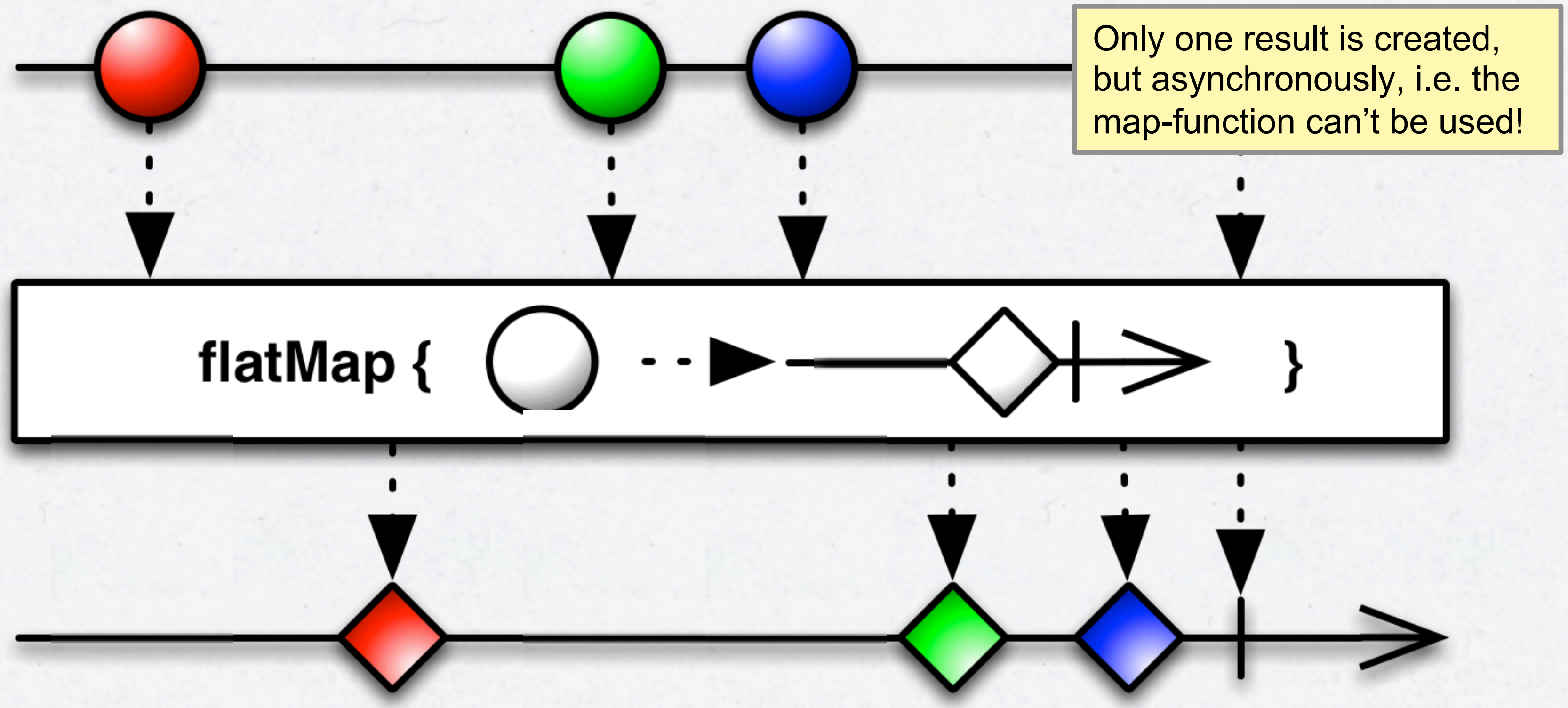
CALLISTA
— ENTERPRISE —

FLATMAP FUNCTION

```
Observable<Integer> o = Observable.range(1,5);

o.flatMap(v -> Observable.range(v*10, 2)).subscribe(System.out::println);
```

```
10
11
20
21
30
31
40
41
50
51
```

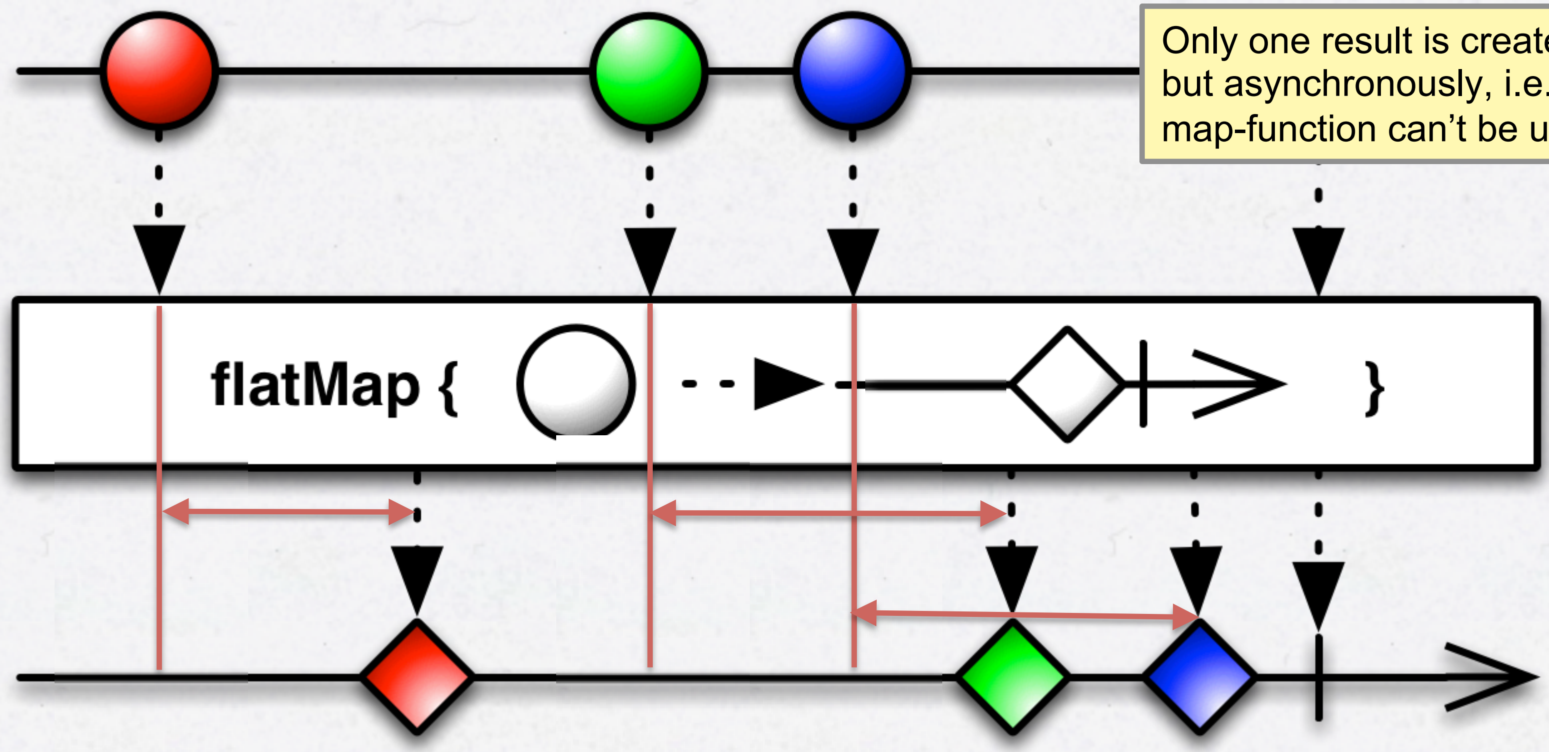
FLATMAP FUNCTION WITH ASYNCH HTTP CLIENT



Only one result is created, but asynchronously, i.e. the map-function can't be used!

CALLISTA
— ENTERPRISE —

FLATMAP FUNCTION WITH ASYNCH-HTTP-CLIENT



Only one result is created, but asynchronously, i.e. the map-function can't be used!

FLATMAP FUNCTION WITH ASYNCH-HTTP-CLIENT

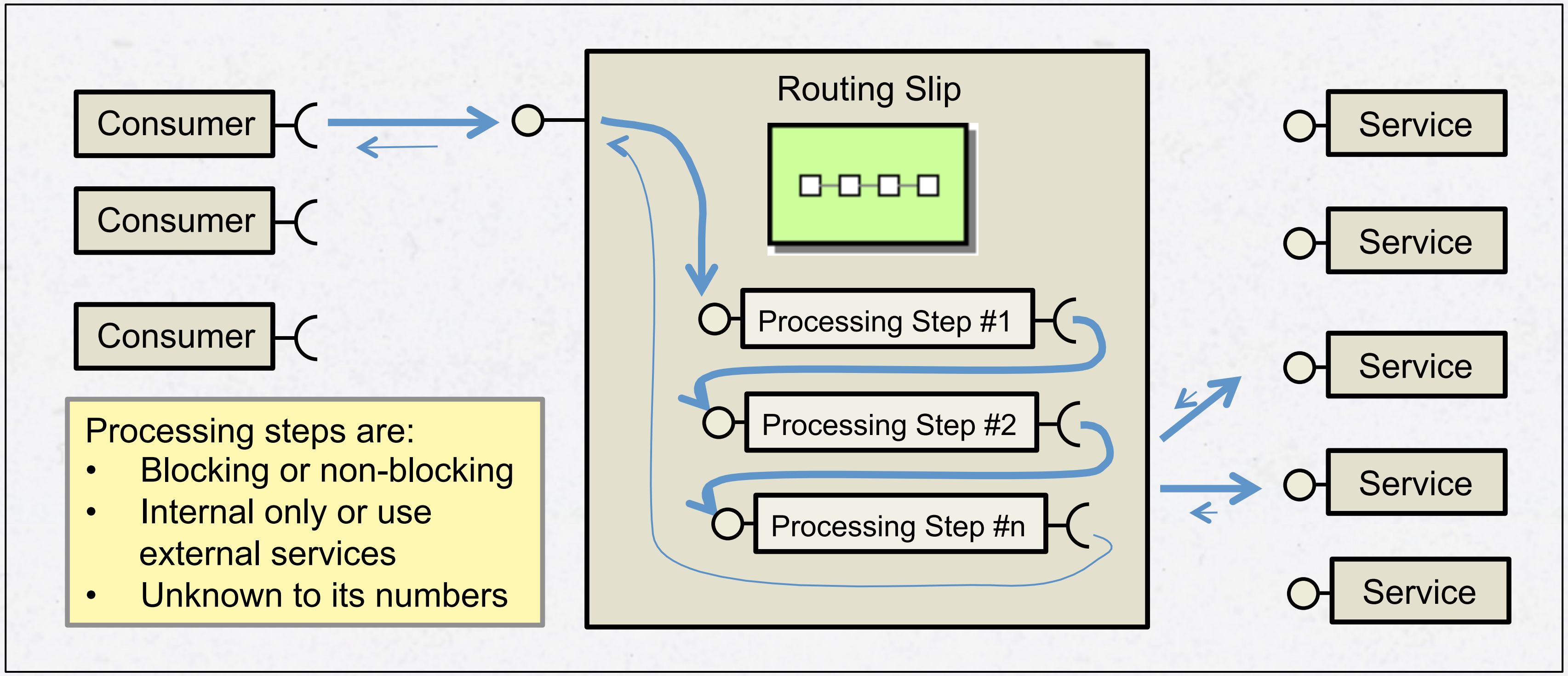
```
Observable<State> observable = Observable.just(new State(query))
  .flatMap(state -> doAsyncCall(state, 1))
  .flatMap(state -> doAsyncCall(state, 2))
  .flatMap(state -> doAsyncCall(state, 3))
  .flatMap(state -> doAsyncCall(state, 4));

Subscription subscription = observable.subscribe(
  state     -> deferredResult.setResult(state.getTotalResult()),
  throwable -> deferredResult.setErrorResult(handleException(throwable))
);
```

THEORY FOR EXERCISE #3 - REACTIVE PROGRAMMING WITH RX JAVA

- The road to Callback-hell…
- Introduction to RxJava
- Routing Slip – RxJava edition

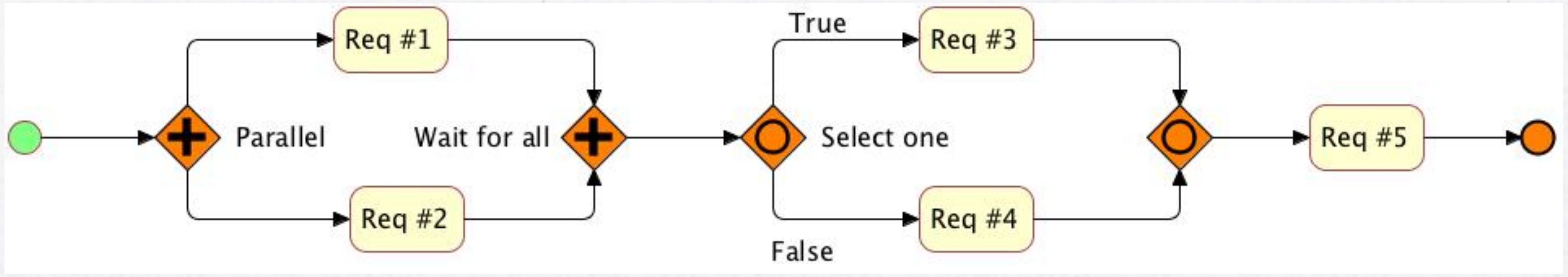
ATTACK THE CALLBACK-HELL WITH A FUNCTIONAL APPROACH!

- How to implement a routing slip pattern using RxJava?



Processing steps are:
- Blocking or non-blocking
- Internal only or use external services
- Unknown to its numbers

ROUTING SLIP EXAMPLE



- To improve scalability all requests should be executed non blocking (and therefore asynchronously)
- Req #1 and Req #2 can be executed in parallel to minimize latency

CALLISTA
— ENTERPRISE —

ROUTING SLIP USING BLOCKING I/O

```java
@RequestMapping("/routing-slip")
public ResponseEntity<String> routingSlip(@RequestParam String qry) {

    try {
        ResponseEntity<String> response;

        response = util.execute("#1", getUrl(1, qry));
        response = util.execute("#2", getUrl(2, qry));

        if (response.getBody().contains("(3)")) {
            response = util.execute("#3", getUrl(3, qry));
        } else {
            response = util.execute("#4", getUrl(4, qry));
        }

        response = util.execute("#5", getUrl(5, qry));
        return response;

    } catch(CommunicationException commEx) {
        return commEx.getErrorResponse();
    }
}
```

`util.execute()` encapsulate the processing of one step

CALLISTA
— ENTERPRISE —

ROUTING SLIP USING NON BLOCKING I/O

```
@RequestMapping("/routing-slip-callback")
public DeferredResult<ResponseEntity<String>> routingSlip(@RequestParam String qry) {

    DeferredResult<ResponseEntity<String>> deferredResult = new DeferredResult<>();

    util.execute(deferredResult, "#1", getUrl(1, qry),
        (Response r1) -> {

            util.execute(deferredResult, "#2", getUrl(2, qry),
                (Response r2) -> {

                    boolean contains3 = util.getResponseBody(r2).contains("(3)");
                    String req3or4 =  contains3 ? "#3" : "#4";
                    String url3or4 =  getUrl(contains3 ? 3 : 4, qry);

                    util.execute(deferredResult, req3or4, url3or4,
                        (Response r3or4) -> {

                            util.execute(deferredResult, "#5", getUrl(5, qry),
                                (Response r5) -> {

                                    deferredResult.setResult(util.createResponse(r5));
                                    ...
```

util.execute() encapsulate the processing of one step

ROUTING SLIP USING NON BLOCKING I/O

```
                                    ...
                                    deferredResult.setResult(util.createResponse(r5));
                                }
                            );
                        }
                    );
                }
            );
        }
    );

    return deferredResult;
}
```

A ROUTING SLIP PATTERN THE RXJAVA WAY

```
@RestController public class RoutingSlipControllerRx {

  @Autowired private UtilRx util;

  @RequestMapping("/routing-slip-rx")
  public DeferredResult<ResponseEntity<String>> routingSlip(@RequestParam String qry) {

    DeferredResult<ResponseEntity<String>> deferredResult = new DeferredResult<>();


...
```

A ROUTING SLIP PATTERN THE RXJAVA WAY

```
// Setup an observable, i.e. declare the processing
Observable<State> observable =

    // Run request #1 and #2 in parallel
    Observable.from(Arrays.asList(
        new Request("#1", getUrl(1, qry)),
        new Request("#2", getUrl(2, qry))))
    .flatMap(request -> util.execute(request))
    .buffer(2)

    // Run either request #3 or #4
    .flatMap(results -> {
        State state = new State();
        if (responseContains(results, "(3)")) {
            return util.execute(state, "#3", getUrl(3, qry));
        } else {
            return util.execute(state, "#4", getUrl(4, qry));
        }
    })

    // Wrap up with the final request #5
    .flatMap(result -> util.execute(result, "#5", getUrl(5, qry)));
...
```

A ROUTING SLIP PATTERN THE RXJAVA WAY

```
  ...

  // Subscribe to the observable, i.e. start the processing
  Subscription subscription = observable.subscribe(
    state -> {
      // We are done, create a response and send it back to the caller
      long processingTimeMs = System.currentTimeMillis() - timestamp;
      deferredResult.setResult(util.createResponse(state.getLastResponse()));
    },
    throwable -> {
      CommunicationException commEx = (CommunicationException) throwable;
      deferredResult.setErrorResult(commEx.getErrorResponse());
    }
  );

  // Unsubscribe, i.e. tear down any resources setup during the processing
  deferredResult.onCompletion(() -> subscription.unsubscribe());

  // Return to let go of the precious thread we are holding on to...
  return deferredResult;
}
```

A ROUTING SLIP PATTERN THE RXJAVA WAY – A SAMPLE RUN…

```
2015-01-20 21:23:36:068 DEBUG qtp743648472-20 s.c.c.t.r.e.e.RoutingSlipControllerRx:40
- Start processing routing slip request, query: 3
2015-01-20 21:23:36:069 DEBUG qtp743648472-20 s.c.c.t.r.u.r.UtilRx:55 - Start request #1
2015-01-20 21:23:36:069 DEBUG qtp743648472-20 s.c.c.t.r.u.r.UtilRx:55 - Start request #2
2015-01-20 21:23:36:070 DEBUG qtp743648472-20 s.c.c.t.r.e.e.RoutingSlipControllerRx:86
- Non-blocking processing setup, return the request thread to the thread-pool
2015-01-20 21:23:36:163 DEBUG New I/O worker #3 s.c.c.t.r.u.r.UtilRx:61 - Got response #1
2015-01-20 21:23:36:168 DEBUG New I/O worker #4 s.c.c.t.r.u.r.UtilRx:61 - Got response #2
2015-01-20 21:23:36:169 DEBUG New I/O worker #4 s.c.c.t.r.u.r.UtilRx:55 - Start request #3
2015-01-20 21:23:36:232 DEBUG New I/O worker #3 s.c.c.t.r.u.r.UtilRx:61 - Got response #3
2015-01-20 21:23:36:233 DEBUG New I/O worker #3 s.c.c.t.r.u.r.UtilRx:55 - Start request #5
2015-01-20 21:23:36:339 DEBUG New I/O worker #4 s.c.c.t.r.u.r.UtilRx:61 - Got response #5
2015-01-20 21:23:36:340 DEBUG New I/O worker #4 s.c.c.t.r.e.e.RoutingSlipControllerRx:72
- Processing complete, time: 272 ms
```