Table of Contents

1. Learning Lamdas – 20 Minutes

Create a java class file : LearningLambdas.java in a java maven project.

Import the following.

```
package ostech.observables;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.function.Consumer;
```

Add the following method.

```
/*
     *  Using Lamdas to print the item in a loop
     */
    static void firstLambdas() {

        List<Integer> intSeq = Arrays.asList(1,2,3);

        intSeq.forEach(x -> {
           int y = x * 2;
```

```
            System.out.println(y);
        });

    }
```

Execute it. Does it print all the item in the List.

Add the following.

```
/*
     * Assigning a lamda function to a variable and using it.
     */

  static void myLambdas() {

       List<Integer> intSeq = Arrays.asList(1,2,3);

       java.util.function.Consumer<Integer> cnsmr = x ->  {
             x = x+1;
             System.out.println(x);};

       intSeq.forEach(cnsmr);

    }
```

Another way of using lamdas expression. It should print 2,3,4.

Add the following method, it again prints the item in the list.

```
/*
   *  Using a local variable in a lamda expression
   */
  static void localVC() {

    List<Integer> intSeq = Arrays.asList(1,2,3);
        int var = 10;
        intSeq.forEach(x -> System.out.println(x + var));

  }
```

Observe the local variable reference.

Execute the above method.

Again another lamdas way – using method reference.

```
/*
   *  using a method reference in Lamda
   */
  static void methodRef() {

      List<Integer> intSeq = Arrays.asList(1,2,3);
      int var = 10;
      intSeq.forEach(System.out::println);
```

```
  }
```

Execute the method and observe it. Think another ways of referencing static method about a String.

Using stream and lamdas.

```
/*
   * Using stream with Lamdas.
   */

  static void stream() {
      class Widget {
          String color;
          Integer weight;
        public String getColor() {
            return color;
        }
        public void setColor(String color) {
            this.color = color;
        }
        public Integer getWeight() {
            return weight;
        }
        public void setWeight(Integer weight) {
            this.weight = weight;
        }
```

```
    public Widget(String color, Integer weight) {
        super();
        this.color = color;
        this.weight = weight;
    }


}

Collection<Widget> widgets = new ArrayList<Widget>();
widgets.add(new Widget("RED", Integer.valueOf("3")) );
widgets.add(new Widget("RED", Integer.valueOf("5")) );
widgets.add(new Widget("BLUE", Integer.valueOf("4")) );
int sum = widgets.stream()
        .filter(w -> w.getColor() == "RED")
        .mapToInt(w -> w.getWeight())
        .sum();
System.out.print(sum);
}
```

Execute the above method.

It should only print: 8. Try to observe the flow of operators.

---------- Lab Ends Here ----------

2. RxJava Observables – 45 Minutes

The build blocks for RxJava code are the following:

- **observables** representing sources of data
- **subscribers (or observers)** listening to the observables
- a set of methods for modifying and composing the data

Start Eclipse

Create a simple maven project.

LearningRxJava

Include the following in the pom.xml

```
<dependencies>
          <dependency>
               <groupId>io.reactivex.rxjava2</groupId>
               <artifactId>rxjava</artifactId>
               <version>2.2.6</version>
          </dependency>
</dependencies>

<build>
```

```
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Create a package:

```
com.tos.learning
```

Import the following packages;

```
import java.util.concurrent.TimeUnit;

import io.reactivex.Observable;
```

Create a java Class having a static main method in the above package. – `Greeting`

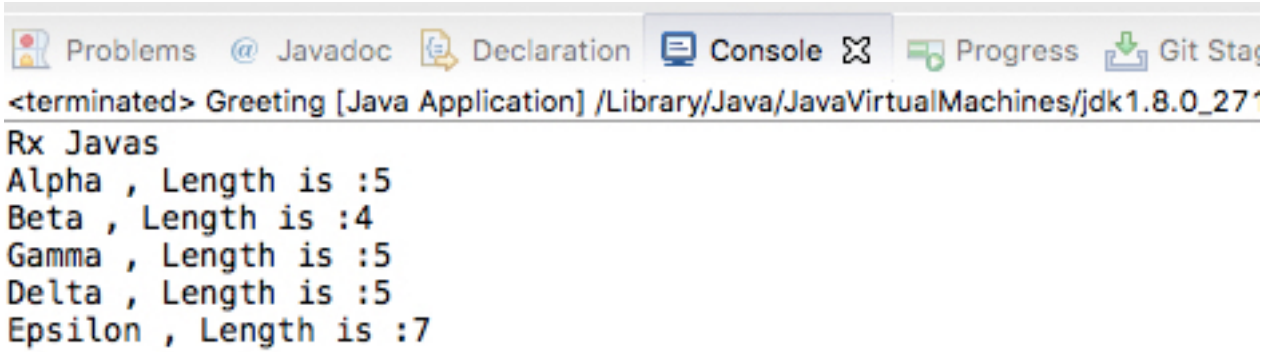
Add the following static method. It will create an observable of string objects and print out the length for each word in the observer.

```
public static void simpleObservables() {

          System.out.println("Rx Javas");

          Observable<String> myStrings = Observable.just("Alpha", "Beta", "Gamma", "Delta",
"Epsilon");
          myStrings.map(m -> m + " , Length is :" + m.length()).subscribe(s ->
System.out.println(s));

     }
```

Invoke the above method from the main method.

```
simpleObservables();
```

Execute the program.



As shown above it print out all the word along with its length.

Let us create a static method, which will create an Observable using an interval.

Add the following two static methods. It will emit an integer increment by 1 for 5 seconds.

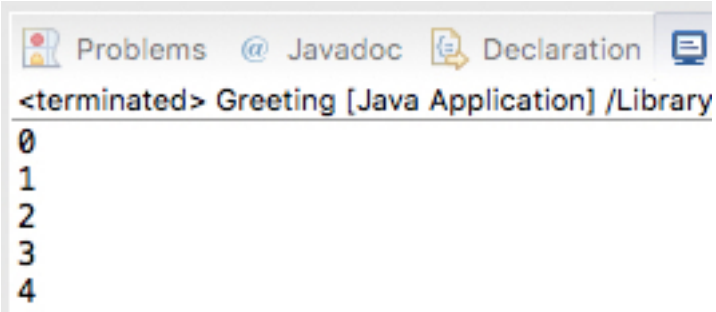
```
public static void interval() {
          Observable<Long> secondIntervals = Observable.interval(1, TimeUnit.SECONDS);
          secondIntervals.subscribe(s -> System.out.println(s));
          /*
           * Hold main thread for 5 seconds
           *
           * so Observable above has chance to fire
           */
          sleep(5000);
     }

     public static void sleep(long millis) {
          try {
```

```
            Thread.sleep(millis);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

Invoke the interval method from the main method.

```
interval();
```

Execute the main method.



Use Case:

In this section, we will simulate fetching Tweet Using Observable.

Data Fetching Pipeline: The Class dependency will be as follows:

TweetGateway → TweetServices → Tweet

Class	Purpose
Tweet	Class representing Tweet.
TweetServices	Data Access and Business Logic of Fetching Tweet

Create a class Tweet in package, com.tos.vo. It will be our Value object or DTO. Replace the class with the following code.

```java
package com.tos.vo;

public class Tweet {

	private String tweetID;
	private String text;


	@Override
	public String toString() {
		return "Tweet [tweetID=" + tweetID + ", text=" + text + "]";
	}
	public Tweet(String tweetID, String text) {
		super();
		this.tweetID = tweetID;
		this.text = text;
	}
	public String getTweetID() {
		return tweetID;
	}
	public void setTweetID(String tweetID) {
		this.tweetID = tweetID;
	}
	public String getText() {
		return text;
	}
	public void setText(String text) {
		this.text = text;
	}


}
```

Create a TweetServices Class. It will generate 6 tweets.

Use package: `com.tos.services`

Import the following packages.

```
import java.util.ArrayList;
import java.util.List;

import com.tos.vo.Tweet;
```

Add the following method that will return a List of Tweet.

```
public static List<Tweet> fetchTweetsFromSource() {
          List<Tweet> tweetList = new ArrayList<Tweet>();
          tweetList.add(new Tweet("1","Learning RxJava"));
          tweetList.add(new Tweet("2","Understaning Rx"));
          tweetList.add(new Tweet("3","RxJava Observables"));
          tweetList.add(new Tweet("4","Learning Transformation"));
          tweetList.add(new Tweet("5","Learning ErrorHandling"));
          tweetList.add(new Tweet("6","Learning BackPressure"));
          return tweetList;
     }
```

Create a class, TweetGateway in package; com.tos.gateway

It will contain two methods as shown below:

getTweets() : Fetch tweets by using the TweetService object.
getObservableTweets() : Provide Tweet in the form of Observable.

Import the following packages and classes.

```
import java.util.List;

import com.tos.services.TweetServices;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
```

Add the following methods as describe earlier.

```
public List<Tweet> getTweets(){

          return TweetServices.fetchTweetsFromSource();
     }

     /*
      *  Provide Tweet in the form of Observable
      */
     public static Observable<Tweet> getObservableTweets() {
          TweetGateway tc = new TweetGateway();
          Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
          return tweets;
     }
```

In the subsequent section, we will use the above classes to create Observable.

Create a class, LearningObservable in the package, com.tos.clients having main method.

Import the following.

```
import com.tos.gateway.TweetGateway;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
import io.reactivex.Observer;
import io.reactivex.disposables.Disposable;
```

Add the following methods.

```
	/*
	 * Subscriber having Only Next Callback.
	 *
	 */
	public static void understandObservables() {
		TweetGateway tc = new TweetGateway();
		Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
		tweets.subscribe(t -> System.out.println(t));

	}

/*
	 * Method to be invoked onCompletion.
	 */
```

```
public static void noMore() {
     System.out.println("No More Record");
}
```

It created an Observable of Tweet from a Tweet Array. And then Print out the tweet onSubscribe.

Add the next method;

```
/*
 * Wrapper holding all the callbacks.
 */
public static void observer() {

     TweetGateway tc = new TweetGateway();
     Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
     Observer<Tweet> observer = new Observer<Tweet>() {
          @Override
          public void onSubscribe(Disposable d) {
               // TODO Auto-generated method stub

          }

          @Override
          public void onNext(Tweet t) {
               System.out.println(t);

          }

          @Override
          public void onError(Throwable e) {
```

```
                    e.printStackTrace();

               }

               @Override
               public void onComplete() {
                    noMore();

               }

          };

          tweets.subscribe(observer);
     }
```

It demonstrates all the wrapper call back provided by Observable. On completion of the event, it will invoke the noMore() method.

Next add the following method, it demonstrates the usage of just, empty and error Operators.

```
/*
      * Different ways of Observable creation : just, empty and error
      */

     public static void differentWaystoObservables() {
          Observable<Tweet> tweets = Observable.just(new Tweet("a","My Tweets"));
          tweets.subscribe(System.out::println);
          Observable<Tweet> myts = Observable.empty();
          myts.subscribe(System.out::println);
          Observable<Tweet> mye = Observable.error(new Throwable("My Error"));
```

```
        mye.subscribe(System.out::println,Throwable::printStackTrace);

}
```

Let us invoke each of the method one at a time from the main method and examine the output.

Add the following line in the main method.

```
understandObservables();
```

```
18    /*
19     * Subscriber having Only Next Callback.
20     *
21     */
22    public static void understandObservables() {
23        TweetGateway tc = new TweetGateway();
24        Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
25        tweets.subscribe(t -> System.out.println(t));
26
27    }
28
29    /*
30     * Method to be invoked onCompletion.
31     */
32    public static void noMore() {
33        System.out.println("No More Record");
```

Problems Javadoc Declaration Console Progress Git Staging

<terminated> LearningObservable [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (05-Jan-20

```
Tweet [tweetID=1, text=Learning RxJava]
Tweet [tweetID=2, text=Understaning Rx]
Tweet [tweetID=3, text=RxJava Observables]
Tweet [tweetID=4, text=Learning Transformation]
Tweet [tweetID=5, text=Learning ErrorHandling]
Tweet [tweetID=6, text=Learning BackPressure]
```

Your output should be as shown above. It prints out all the tweet subsequently.

Next invoke the following, comment the above method invocation.

```
observer();
```

```
39    public static void observer() {
40
41        TweetGateway tc = new TweetGateway();
42        Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
43        Observer<Tweet> observer = new Observer<Tweet>() {
44            @Override
45            public void onSubscribe(Disposable d) {
46                System.out.println("Just Subscribe");
47            }
48
49            @Override
50            public void onNext(Tweet t) {
51                System.out.println(t);
52
53            }
```

Problems | Javadoc | Declaration | Console | Progress | Git Staging

<terminated> LearningObservable [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (05-Jan-2021, 2

```
Just Subscribe
Tweet [tweetID=1, text=Learning RxJava]
Tweet [tweetID=2, text=Understaning Rx]
Tweet [tweetID=3, text=RxJava Observables]
Tweet [tweetID=4, text=Learning Transformation]
Tweet [tweetID=5, text=Learning ErrorHandling]
Tweet [tweetID=6, text=Learning BackPressure]
No More Record
```

As you can observe, after printing out all Tweets, it finally invoke the onCompletion method.

Next invoke the following method.

```
differentWaystoObservables();
```

```
72      /*
73       * Different ways of Observable creation : just, empty and error
74       */
75
76      public static void differentWaystoObservables() {
77          Observable<Tweet> tweets = Observable.just(new Tweet("a","My Tweets"));
78          tweets.subscribe(System.out::println);
79          Observable<Tweet> myts = Observable.empty();
80          myts.subscribe(System.out::println);
81          Observable<Tweet> mye = Observable.error(new Throwable("My Error"));
82          mye.subscribe(System.out::println,Throwable::printStackTrace);
83
84
85      }
86  }
87
```

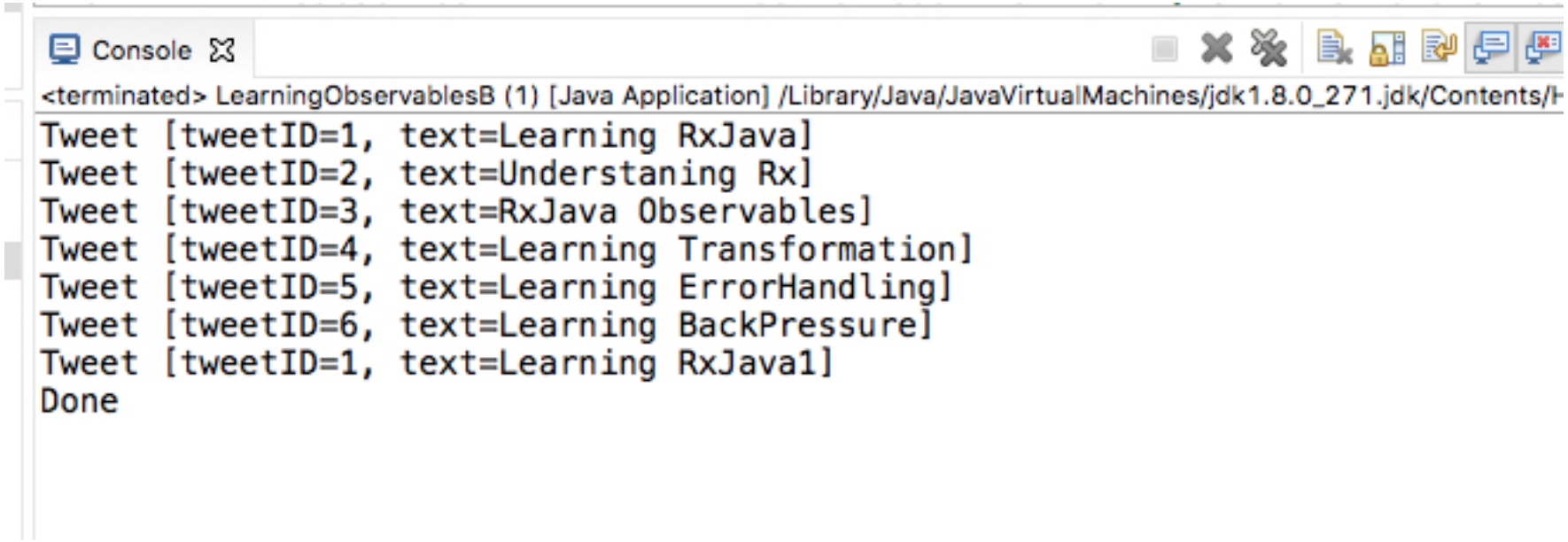
Problems @ Javadoc Declaration Console Progress Git Staging

```
terminated> LearningObservable [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java  (05-Jan-2
weet [tweetID=a, text=My Tweets]
ava.lang.Throwable: My Error
        at com.tos.clients.LearningObservable.differentWaystoObservables(LearningObservable.java:81)
        at com.tos.clients.LearningObservable.main(LearningObservable.java:15)
```

As expected,

1. It printed the Tweet
2. Empty so no action
3. Error Thrown and stack trace is printed.

```
    /*
     *  Observable using Iterator.
     *  Subscriber with All three methods - On Next , On error and On completion
     *  The first and Last method will get invoked.
     */

  static void myObservable() {

        Observable<Tweet> tweets =
Observable.fromIterable(TweetServices.fetchTweetsFromSource());
        tweets.subscribe(
                (Tweet tweet) -> System.out.println(tweet) ,
                (Throwable t) -> { t.printStackTrace(); },
                () -> System.out.println("Done")
                );

    }
```




```
  /*
   *  Method for disposing the Subscribtion.
   */
static void mySubscription() {

    Observable<Tweet> tweets =
Observable.fromIterable(TweetServices.fetchTweetsFromSource());
    Disposable tDisposable = tweets.subscribe(
            (Tweet tweet) -> System.out.println(tweet) ,
            (Throwable t) -> { t.printStackTrace(); },
            () -> System.out.println("Done")
            );
    tDisposable.dispose();
```

```
}
```

---------- Lab Ends Here ----------

3. Understanding Custom Observer – 40 Minutes

Understand the following feature of Observer and Observable:

- Observable Default Execution Model
- Multiple invoke of Subscription with multiple Subscription
- Using Cache

Create a class, MyObsCreate with static main method in package, com.tos.clients

Import the following packages,

```
import java.util.List;

import com.tos.gateway.TweetGateway;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
import io.reactivex.ObservableEmitter;
import io.reactivex.ObservableOnSubscribe;
```

Add the following method to create a custom Observable

```
/*
     *  Creating a custom Observable with Subscriber Method
     */
```

```
    public static void createObservable() {

        Observable<Integer> o = Observable.create(new
ObservableOnSubscribe<Integer >() { // Action
            @Override
            public void subscribe(ObservableEmitter<Integer> subscriber) throws
Exception {
                // TODO Auto-generated method stub
                try {
                    for (int i = 0; i < 10 ; i++) {
                        subscriber.onNext(i);
                    }

                    subscriber.onComplete();

                }
                catch (RuntimeException e) {

                    subscriber.onError(e);

                }
            }
        });

        o.subscribe(new Observer<Integer>() {

            @Override
```

```
            public void onSubscribe(Disposable d) {
                // TODO Auto-generated method stub
                System.out.println("\n On Subscribe " + d);
            }

            @Override
            public void onError(Throwable e) {
                // TODO Auto-generated method stub
                e.printStackTrace();
            }

            @Override
            public void onComplete() {
                // TODO Auto-generated method stub
                System.out.println("\nAll Done!");
            }

            @Override
            public void onNext(Integer t) {
                // TODO Auto-generated method stub
                System.out.println(t);
            }
        });

    }
```

Execute the method.

```
 On Subscribe CreateEmitter{null}
0
1
2
3
4
5
6
7
8
9

All Done!
```

Add the following method, to understand the thread model of Observable.

```
/*
	 * Default Thread Model in Observables.
	 */

	public static void defaultThread() {
		TweetGateway tc = new TweetGateway();
		Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
```

```
            log("Before");
            tweets.subscribe(t -> {
                 log("Subscribe");
                 System.out.println(t);

            });

            log("After");

      }

private static void log(Object msg) {
            System.out.println(Thread.currentThread().getName() + ": " + msg);
      }
```

It will print the thread detail, in which the operator executes.

Add the following method, to understand the thread for each callback.

```
/*
       * Low Level Operator of Create Method to understand the Thread.
       */
      public static void customObservable() {
            TweetGateway tc = new TweetGateway();
            Observable<Tweet> tweets = Observable.create(new ObservableOnSubscribe<Tweet>() {

                  @Override
                  public void subscribe(ObservableEmitter<Tweet> subscriber) throws Exception {
                        // TODO Auto-generated method stub
```

```
                List<Tweet> tweets = tc.getTweets();
                log("Create");
                for (Tweet tweet : tweets) {
                     subscriber.onNext(tweet);
                }
                subscriber.onComplete();
                log("Completed");
           }
      });

      log("Starting");
      tweets.subscribe(t -> {
           log("Tweet :");
           System.out.println(t);

      });

      log("After");

 }
```

Add the following method to understand that subscriber creation method is invoke multiple times as the number of subscription. If DB is invoked inside it then DB connectivity will be invoked Multiple times when subscribe for multiple times.

```
/*
      * Mimic just with create subscriber function invoke as much as subscription is
      * called. If DB is invoke inside it then DB connectivity will be invoked
      * multiples when subscribe for multiple times.
      */
```

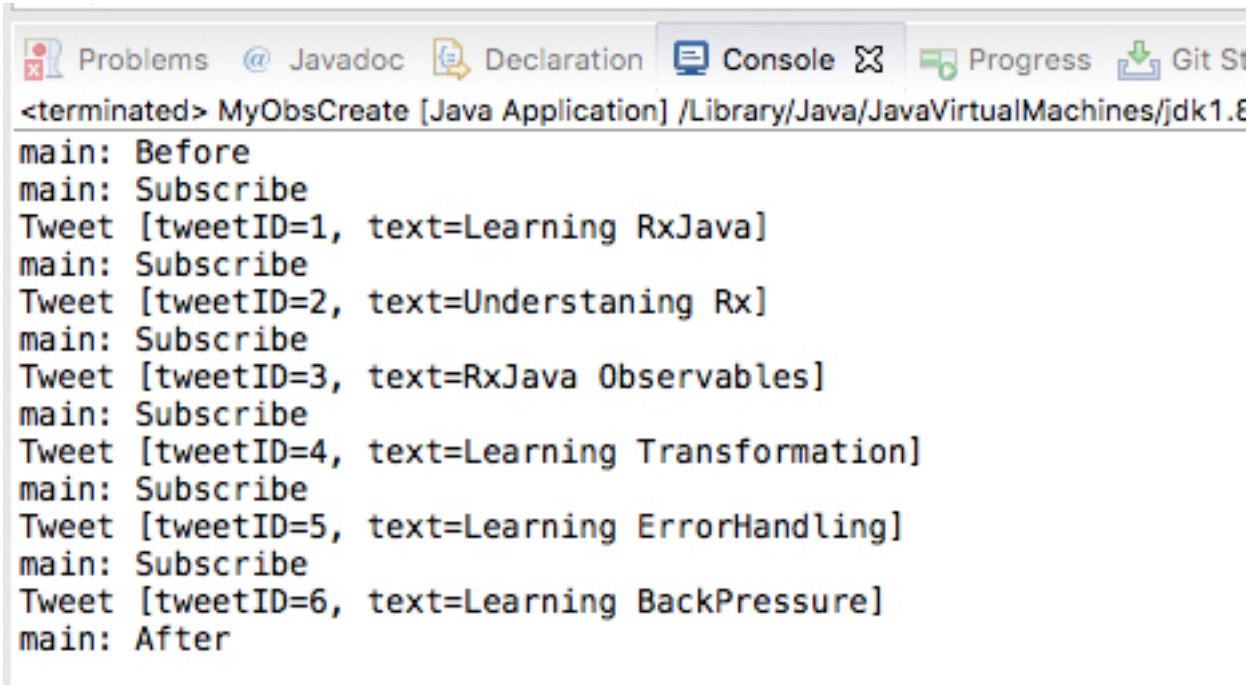
```
public static void justWithCreate() {
	TweetGateway tc = new TweetGateway();
	Observable<Tweet> tweets = Observable.<Tweet>create(subscriber -> {
		log("Inside");
		subscriber.onNext(new Tweet("1", "Just My Tweet"));
		subscriber.onComplete();
	});

	log("Before");
	tweets.subscribe(t -> {
		log("Subscribe");
		System.out.println(t);

	});
	tweets.subscribe(t -> {
		log("Subscribe");
		System.out.println(" >>" + t);

	});

	log("After");

}
```

Add the following method, that will cache the subscription method so that it wont invoke multiple times.

```
/*
	* Using cache to invoke the subscribe function once. Verify how many - Inside
	* message has been printed.
```

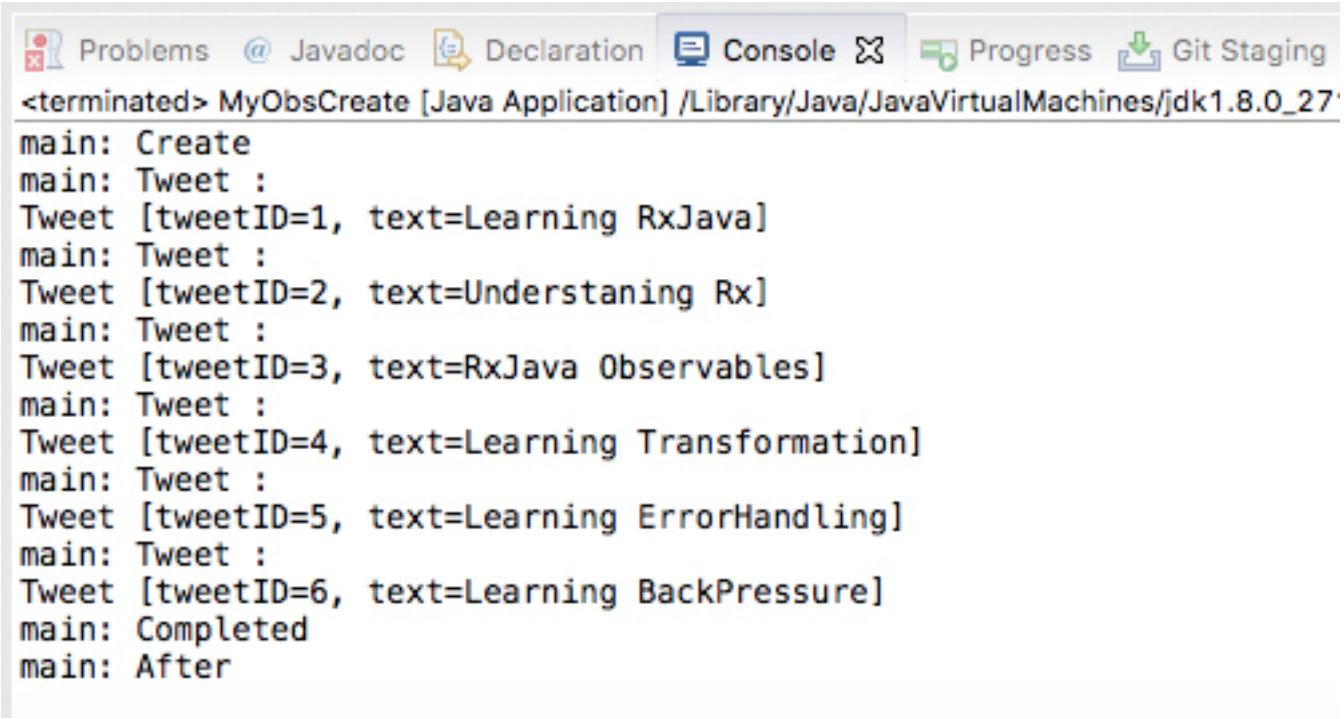
```
 */
public static void invokeCreateOnce() {
	TweetGateway tc = new TweetGateway();
	Observable<Tweet> tweets = Observable.<Tweet>create(subscriber -> {
		log("Inside");
		subscriber.onNext(new Tweet("1", "Just My Tweet"));
		subscriber.onComplete();
	}).cache();

	log("Before");
	tweets.subscribe(t -> {
		log("Subscribe");
		System.out.println(t);

	});
	tweets.subscribe(t -> {
		log("Subscribe");
		System.out.println(" >>" + t);

	});

	log("After");

}
```

Invoke the defaultThread() from the main method and execute the program.



As you can see everything is executed under the thread, main which is of the subscriber thread.

Now invoke `customObservable();` Comment any other method.



Verify the sequences of event that are invoked and under the client thread, main.

Now let us invoke, `justWithCreate()`

```
82    public static void justWithCreate() {
83        TweetGateway tc = new TweetGateway();
84        Observable<Tweet> tweets = Observable.<Tweet>create(subscriber -> {
85            log("Inside");
86            subscriber.onNext(new Tweet("1", "Just My Tweet"));
87            subscriber.onComplete();
88        });
89
90        log("Before");
91        tweets.subscribe(t -> {
92            log("Subscribe");
93            System.out.println(t);
94
95        });
96        tweets.subscribe(t -> {
97            log("Subscribe");
98            System.out.println(" >>" + t);
99
100       });
101
102       log("After");
103
```

Problems | Javadoc | Declaration | Console | Progress | Git Staging

```
<terminated> MyObsCreate [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java  (06-Jan-
main: Before
main: Inside
main: Subscribe
Tweet [tweetID=1, text=Just My Tweet]
main: Inside
main: Subscribe
 >>Tweet [tweetID=1, text=Just My Tweet]
main: After
```

ro

You can observe that “Inside” text has been printed twice. Why? Because there are 2 subscriptions. Perhaps we will want to invoke once only. For example, if it’s a fetch from DB, multiples database call will be required. Use cache. Try invoking the next method.

```
invokeCreateOnce()
```

Execute the program.

```
110    public static void invokeCreateOnce() {
111        TweetGateway tc = new TweetGateway();
112        Observable<Tweet> tweets = Observable.<Tweet>create(subscriber -> {
113            log("Inside");
114            subscriber.onNext(new Tweet("1", "Just My Tweet"));
115            subscriber.onComplete();
116        }).cache();
117
118        log("Before");
119        tweets.subscribe(t -> {
120            log("Subscribe");
121            System.out.println(t);
122
123        });
124        tweets.subscribe(t -> {
125            log("Subscribe");
126            System.out.println(" >>" + t);
127
128        });
129
```

Problems Javadoc Declaration Console Progress Git Staging

```
terminated> MyObsCreate [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (06
ain: Before
ain: Inside
ain: Subscribe
weet [tweetID=1, text=Just My Tweet]
ain: Subscribe
>>Tweet [tweetID=1, text=Just My Tweet]
ain: After
```

This time, text “Inside” is printed only once.

Things to do:
Convert Database Items to an Observable Items.

---------- Lab Ends Here ----------

4. Transformation – 60 Minutes

Following operations will be performed in this lab.

- Apply map and filter operators in custom Observable
- Create an instance of Observer and attach to a subscribe method

Create a package:
com.tos.transformation

Add a class, Launcher with main method.

Import the following packages and classes.

```
import io.reactivex.Observable;
import io.reactivex.Observer;
import io.reactivex.disposables.Disposable;
```

Add the following:

```
/*
     *  Using map : int 0..10 being transform by multiplying with 10.
     *  Using flatMap : int 0..10 being transform into an Observable with + and -
int.
```

```
     */
    public static void simpleTransformation() {
        Observable<Integer> o = Observable.range(0,10);
        Observable<Integer> timesTen =
                o.map(t -> { System.out.println(t);
                return t * 10;});
        timesTen.subscribe(System.out::println);
        System.out.println("----------");
        Observable<Integer> numbersAndNegatives =
                o.flatMap( t -> Observable.just(t, -t));
        numbersAndNegatives.map(
                val -> { System.out.println(val);
                return val;});
        numbersAndNegatives.subscribe(System.out::println);
    }
```

Execute and observe the output.

Add the following method.

```
/*
     * Create a custom Observable and apply map and filter operator to it.
     * It displays the item which length is greater or equal to 5
     */

public static void tranformation() {
```

```
        // TODO Auto-generated method stub
                  Observable<String> source = Observable.create(emitter -> {
                       try {
                            emitter.onNext("Alpha");
                            emitter.onNext("Beta");
                            emitter.onNext("Gamma");
                            emitter.onNext("Delta");
                            emitter.onNext("Epsilon");
                            emitter.onComplete();
                       } catch (Exception e) {
                            // TODO Auto-generated catch block
                            emitter.onError(e);;
                       }
                       });

        /*
         * Print only the object which is length is greater than 4.
         */
        Observable<Integer> lengths = source.map(String::length);
        Observable<Integer> filtered = lengths.filter(i -> i >= 5);
             filtered.subscribe(s -> System.out.println("RECEIVED: " + s));

             /*
              * Print only the object which is length is greater than 4, observable created
using just operator.
              */
             Observable<String> asource = Observable.just("Alpha", "Beta", "Gamma",
"Delta","Epsilon");
                       asource.map(String::length).filter(i -> i >= 5)
                       .subscribe(s -> System.out.println("RECEIVED: " + s));
```

```
    }
```

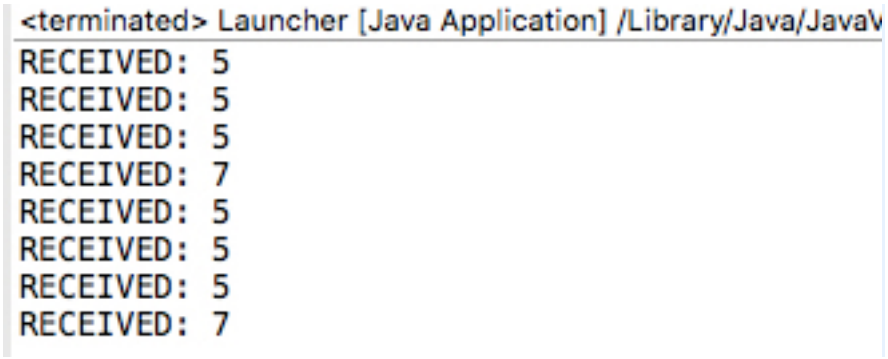
It created a custom Observable, source and emitted 5 strings mention in the onNext method. The operator map, convert each of the object to its length and the filter operator is applied to extract only the object which length is greater than 4. Finally, the subscriber or observer prints the length.

The last code performs the same action however using the just operator.

Invoke the above method in the main method.

```
tranformation();
```

Execute the program. Your result should be as shown below.



Comment the above method invocation from the main() method.

Add the following method.

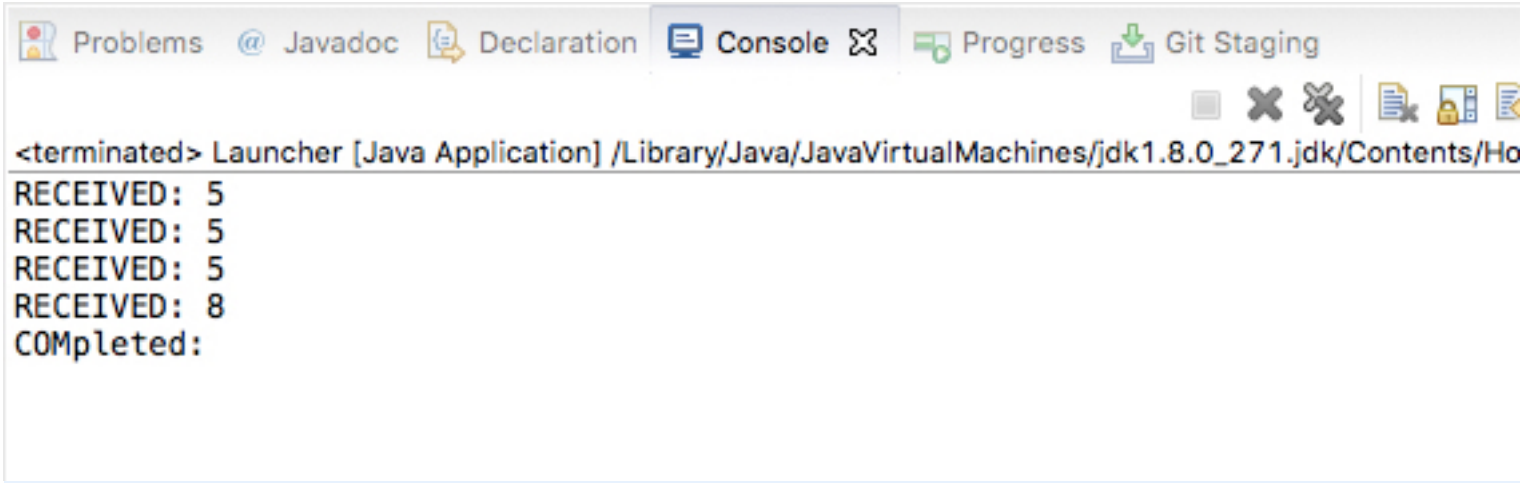
```
public static void myObserver() {

          Observable<String> source =
                    Observable.just("Alpha", "Beta", "Gamma", "Delta","Epsilons");
                    Observer<Integer> myObserver = new Observer<Integer>() {
                              @Override
                              public void onSubscribe(Disposable d) {
                                   //do nothing with Disposable, disregard for now
                              }
                              @Override
                              public void onNext(Integer value) {
                                   System.out.println("RECEIVED: " + value);
                              }
                              @Override
                              public void onError(Throwable e) {
                                   e.printStackTrace();
                              }
                              @Override
                              public void onComplete() {
                                   // TODO Auto-generated method stub
                                   System.out.println("COMpleted: ");
                              }
                    };

                    source.map(String::length).filter(i -> i >= 5).subscribe(myObserver);
     }
```

It demonstrates the another way of creating Observer instance and supply to subscribe method.

Invoke the above method from the main.

```
myObserver();
```

Verify the result. Ensure that you understand the flow of the event.



Now let us apply these operators to the Tweet Example.

Create a Class, `MyTransformation` with main method in package: `com.tos.transformation.`

Import the following:

```
import static java.util.concurrent.TimeUnit.SECONDS;

import java.util.concurrent.TimeUnit;
```

```
import com.tos.gateway.TweetGateway;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
```

Add the following method.

```
/*
      * Get only the Text Begin with Learning
      */
     public static void myFilter() {
          TweetGateway tc = new TweetGateway();
          Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
          Observable<Tweet> tweetsLearning = tweets.filter( t ->
t.getText().startsWith("Learning"));
          tweetsLearning.subscribe(t -> System.out.println(t));

     }
```

Using filter operation. It will print only the Tweet begin with Learning.

Next method, get only the Tweet begins with “Learning” and Add “Advance” to the tweet. Using filter and transformation – Map operators.

```
/*
      * Get only the Text Begin with Learning and Add Advance to the text.
      */
```

```
    public static void myMap() {
        TweetGateway tc = new TweetGateway();
        Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
        Observable<Tweet> tweetsLearning = tweets.filter( t ->
t.getText().startsWith("Learning"));
        Observable<Tweet> tweetsAdvance = tweetsLearning.map( t -> {
            t.setText(t.getText() + " - Advance");
            return t;
        });
        tweetsAdvance.subscribe(t -> System.out.println(t));

    }
```

Next method will demonstrate the flow of operators. You can view the message using doOnNext operator without touching the messages.

```
/*
     *  Understand the Transformation Sequences. A record by record from Begin to End - Async.
     */
    public static void transformationFlow() {

        Observable
                .just(8, 9, 10)
                .doOnNext(i -> System.out.println("A: " + i))
                .filter(i -> i % 3 > 0)
                .doOnNext(i -> System.out.println("B: " + i))
                .map(i -> "#" + i * 10)
                .doOnNext(s -> System.out.println("C: " + s))
                .filter(s -> s.length() < 4)
                .subscribe(s -> System.out.println("D: " + s));
```

```
    }
```

Execute the method and observe the output.

Let us fetch only 1 records - Add the take operator.

```
     */
    public static void transformationFlow() {

        Observable.just(8, 9, 10).doOnNext(i -> System.out.println("A: " + i)).fil
                .doOnNext(i -> System.out.println("B: " + i)).map(i -> "#" + i * 1
                .doOnNext(s -> System.out.println("C: " + s)).filter(s -> s.length
                // .take(1)
                .subscribe(s -> System.out.println("D: " + s));

        /*
         * Let us fetch only 1 records - Uncomment the take operator.
         */

    }

```

Using FlatMap to return multiples Observable. Multiple tweets.

```
/*
     * Get only the Text Begin with Rxjava and Split it into 2 Observable.
     */
    public static void myFlatMap() {
        TweetGateway tc = new TweetGateway();
        Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
```

```
          Observable<Tweet> tweetsLearning = tweets.filter( t ->
t.getText().startsWith("RxJava"));
          Observable<Tweet> tweetsAdvance = tweetsLearning.flatMap( t -> {
               String topics[] = t.getText().split(" ");
               Tweet item1 = new Tweet(t.getTweetID(),topics[0]);
               Tweet item2 = new Tweet(t.getTweetID(),topics[1]);
               return Observable.<Tweet>just(item1, item2);
          });
          tweetsAdvance.subscribe(t -> System.out.println(t));

     }
```

Execute the method

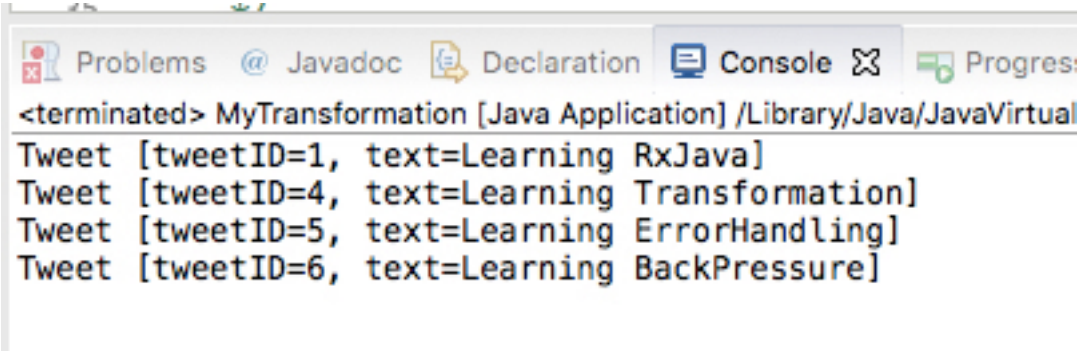
Delaying Tweet using defer.

```
/*
      * Delay emission by 5 Seconds
      */
     public static void delayTweet() {

          TweetGateway tc = new TweetGateway();
          Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new
Tweet[0])).delay(2, SECONDS);
          Observable<Tweet> tweetsLearning = tweets.filter( t ->
t.getText().startsWith("Learning"));
          tweetsLearning.subscribe(t -> System.out.println(t));

          Observable.just("Learning","RxJava","REactive").delay(word ->
Observable.timer(word.length(), SECONDS))
              .subscribe(System.out::println);
```

```
            try {
                TimeUnit.SECONDS.sleep(15);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
    }
```

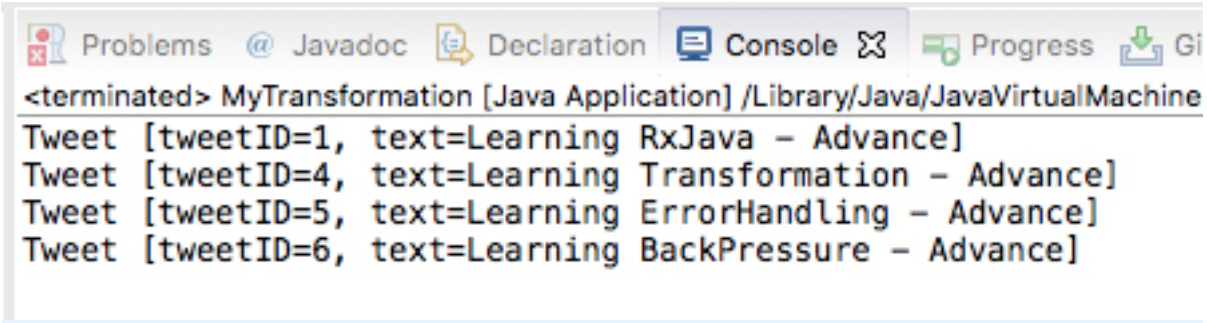
Now let us invoke the method one at a time and execute it, to understand its feature.

```
myFilter();
```



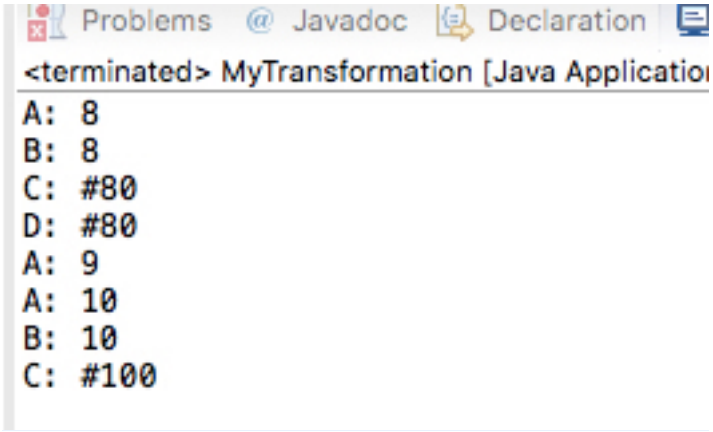
It prints only the Tweet begins with “Learning”

```
myMap();
```



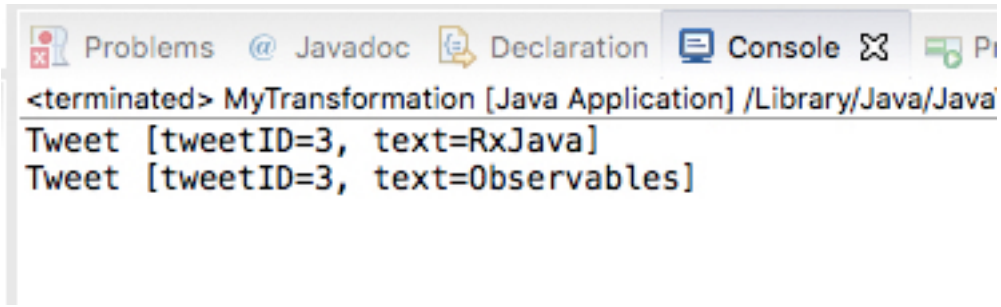
It applies filter and transform the tweet by adding Advance text on it.

```
transformationFlow();
```



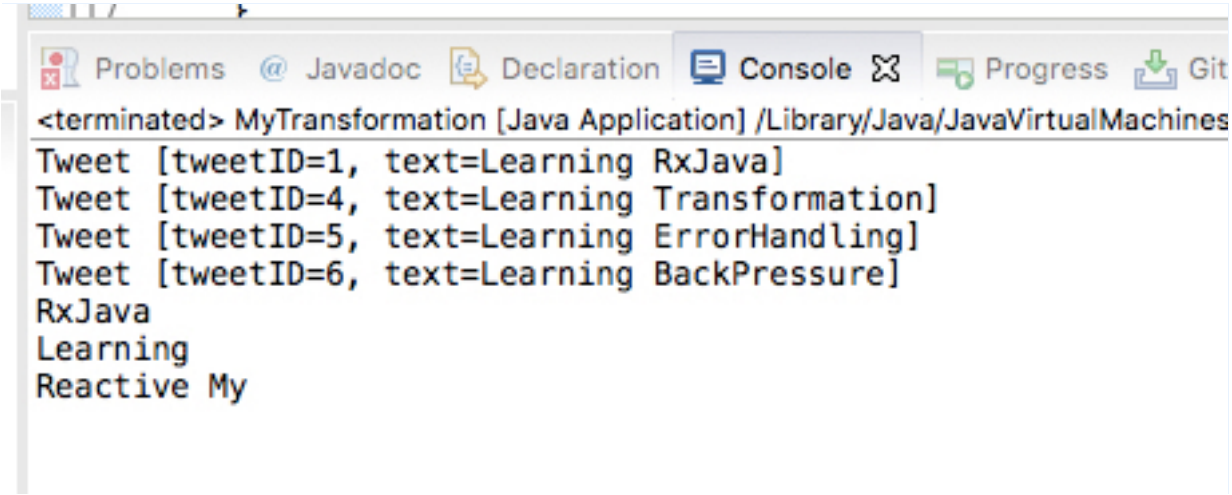
Focus on the sequence of each message, e.g flow of 8 message then 9 etc.

```
myFlatMap();
```



Tweet id 3, is converted into Two Tweets.

```
delayTweet();
```



Add

```
/*
   * Operators can be chained.
   *  Item should be greater than 11 and display only the last item
   */
    public static void operatorChaining() throws Exception {

        Observable<Integer> integerObservable = Observable.just(1, 2, 3);
        integerObservable // emits 1, 2, 3
                .map(i -> i + 10) // adds 10 to each item; emits 11, 12, 13
                .filter(i -> i > 11) // emits items that satisfy condition; 12,
13
                .lastElement() // emits last item in observable; 13
                // unlimited operators can be added ...
                .subscribe(System.out::println); // prints 13

    }
```

Execute the method.

Does it print only 13?

Add the next method and execute.

```
/*
   *  understanding Onnext to debug each of the flow.
   */
    public static void doOnNextOperator() {

        Observable.range(1, 3)
                .doOnNext(value -> System.out.println("before transform: " +
value))
                .map(value -> value * 2).
                doOnNext(value -> System.out.println("after transform: " +
value))
                .subscribe(System.out::println);
    }
```

Observe the text that prints after each operator.

Add the repeat operator:

```
/*
  * It will display int continously.
  */
    public static void repeatOperator() {

        Observable.just(1, 2, 3).repeat()
                .subscribe(next -> System.out.println("next: " + next),
```

```
                error -> System.out.println("error: " + error),
                () -> System.out.println("complete"));
    }
```

Execute and understand it.

Next add the method to understand the concatMap

```
/*
    *  Verify the Order of Emiting Name i.e Mark  - flatMap ( Not in Order).
    *  Verify the Order of Emiting - concatMap (In Order).
    *  If you run the example, you can see the order is maintained as source
observable
    *   i.e Mark, John, Trump, Obama and it always maintains the same order.
    */
    public static void concatMapOperator() {

        getUsersObservable()
            .subscribeOn(Schedulers.io())
            //flatMap / concatMap
           .concatMap( user -> {
                // getting each user address by making another network call
                return getAddressObservable(user);
            }
        )
        .blockingSubscribe(new Observer<User>() {
            @Override
            public void onSubscribe(Disposable d) {
```

```
            }

            @Override
            public void onNext(User user) {
                TweetServices.log("onNext: " + user.getName() + ", " +
user.getGender()
                 + ", " + user.getAddress().getAddress());
            }

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onComplete() {
             TweetServices.log( "All users emitted!");
            }
        });



    /**
     * Assume this as a network call
     * returns Users with address filed added
     */
    private static Observable<User> getAddressObservable(final User user) {
```

```
        final String[] addresses = new String[]{
                "1600 Amphitheatre Parkway, Mountain View, CA 94043",
                "2300 Traverwood Dr. Ann Arbor, MI 48105",
                "500 W 2nd St Suite 2900 Austin, TX 78701",
                "355 Main Street Cambridge, MA 02142"
        };

        return Observable
                .create(new ObservableOnSubscribe<User>() {
                    @Override
                    public void subscribe(ObservableEmitter<User> emitter) throws
Exception {
                        Address address = new Address();
                        address.setAddress(addresses[new Random().nextInt(2) +
0]);
                        if (!emitter.isDisposed()) {
                            user.setAddress(address);


                            // Generate network latency of random duration
                            int sleepTime = new Random().nextInt(1000) + 500;

                            Thread.sleep(sleepTime);
                            emitter.onNext(user);
                            emitter.onComplete();
                        }
```

```
                    }
                }).subscribeOn(Schedulers.io());
    }

    /**
     * Assume this is a network call to fetch users
     * returns Users with name and gender but missing address
     */
    private static Observable<User> getUsersObservable() {
        String[] maleUsers = new String[]{"Mark", "John", "Trump", "Obama"};

        final List<User> users = new ArrayList<>();

        for (String name : maleUsers) {
            User user = new User();
            user.setName(name);
            user.setGender("male");

            users.add(user);
        }

        return Observable
                .create(new ObservableOnSubscribe<User>() {
                    @Override
                    public void subscribe(ObservableEmitter<User> emitter) throws
Exception {
                        for (User user : users) {
```

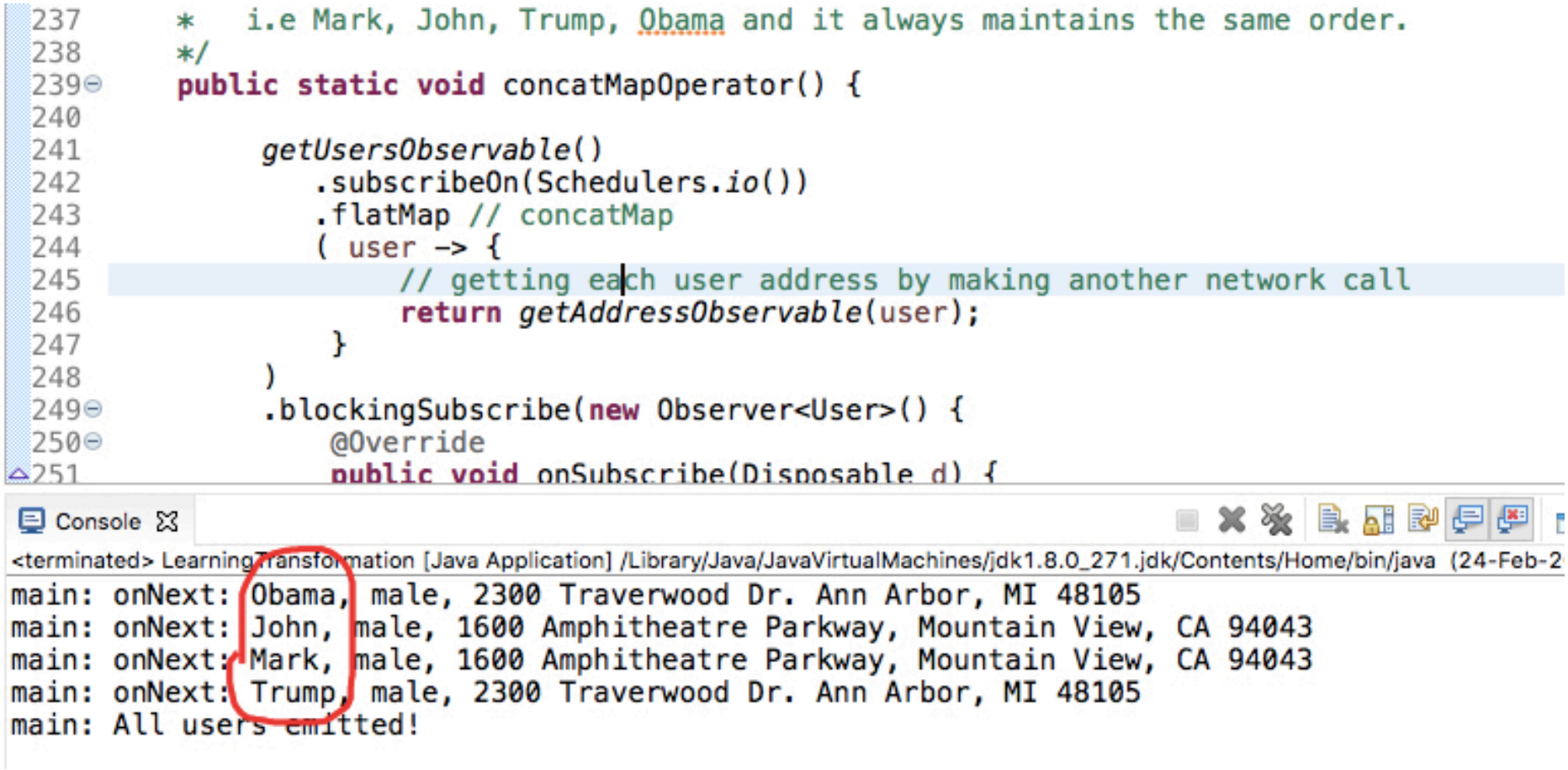
```
                        if (!emitter.isDisposed()) {
                            emitter.onNext(user);
                        }
                    }

                    if (!emitter.isDisposed()) {
                        emitter.onComplete();
                    }
                }
            }).subscribeOn(Schedulers.io());
}
public static class User {
    String name;
    String email;
    String gender;
    Address address;
     public String getName() {
         return name;
     }
     public void setName(String name) {
         this.name = name;
     }
     public String getEmail() {
         return email;
     }
     public void setEmail(String email) {
         this.email = email;
```

```
    }
    public String getGender() {
        return gender;
    }
    public void setGender(String gender) {
        this.gender = gender;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }

   // getters and setters
}

public static class Address{
String address;

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
```

```
    }

        }
```

Execute the method and observe.

Replace the concatMap with flatMap and observe the print.


237 * i.e Mark, John, Trump, Obama and it always maintains the same order.
238 */
239 public static void concatMapOperator() {
240
241 getUsersObservable()
242 .subscribeOn(Schedulers.io())
243 .flatMap // concatMap
244 (user -> {
245 // getting each user address by making another network call
246 return getAddressObservable(user);
247 }
248)
249 .blockingSubscribe(new Observer<User>() {
250 @Override
251 public void onSubscribe(Disposable d) {
Console
<terminated> LearningTransformation [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (24-Feb-2
main: onNext: Obama, male, 2300 Traverwood Dr. Ann Arbor, MI 48105
main: onNext: John, male, 1600 Amphitheatre Parkway, Mountain View, CA 94043
main: onNext: Mark, male, 1600 Amphitheatre Parkway, Mountain View, CA 94043
main: onNext: Trump, male, 2300 Traverwood Dr. Ann Arbor, MI 48105
main: All users emitted!

Next Add the following to understand the switchMap

```
    /*
     * SwitchMap always return the latest Observable and emits the items from it.
     * Ex - Get the Latest Tweet.
     */
    public static void switchMap() throws Exception {
    Observable<Tweet> it =
Observable.fromIterable(TweetServices.fetchTweetsFromSource());

    it.switchMap(t -> {
        return Observable.just(t).delay(1, TimeUnit.SECONDS);
    }).subscribe(System.out::println);


    Thread.sleep(2000);
    }
```

Execute the method.

Does it print? `Tweet [tweetID=1, text=Learning RxJava1]`

---------- Lab Ends Here ----------

5. Hot Observable – 40 Minutes

Hot and Cold Observable:

Create a customer Class.

```
package com.tos.vo;

public class Customer {

	private long id;
	private String name;


	public Customer(long id, String name) {
		super();
		this.id = id;
		this.name = name;
	}
	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
```

```
    public void setName(String name) {
        this.name = name;
    }


}
```

Define a class, CustomerService for Fetching Customer Data.

Add the following method that will create the following Data required for the next program.:

```
public static List<Customer> getCustomers(){
        ArrayList<Customer> custList = new ArrayList<Customer>();
        custList.add(new Customer(12, "John P"));
        custList.add(new Customer(14, "Akbar M"));
        custList.add(new Customer(16, "Swammy N"));
        custList.add(new Customer(18, "Jennifer P"));
        custList.add(new Customer(20, "Ramaya K"));
        return custList;

    }
```

Create a class for defining cold and hot observables method. – `LearningHotColdObservables`

Import the following:

```
import java.util.Iterator;
import java.util.List;

import com.tos.services.CustomerService;
import com.tos.vo.Customer;

import io.reactivex.Observable;
import io.reactivex.ObservableEmitter;
import io.reactivex.ObservableOnSubscribe;
```

Add the following methods:

Logging method.

```
private static void log(Object msg) {
        System.out.println(Thread.currentThread().getName() + ": " + msg);
    }
```

To demonstrate the cold

```
    /*
     * Cold Observable - Fetch not Happen till it get subscribed.
     */
    public  static Observable<Customer> fetchCustomers() throws Exception {

        Observable<Customer> custObs = Observable.create(new
ObservableOnSubscribe<Customer>() {

            @Override
            public void subscribe(ObservableEmitter<Customer> emitter) throws
Exception {

                List<Customer> custs = CustomerService.getCustomers();
                for (Iterator iterator = custs.iterator(); iterator.hasNext();) {
                    Customer customer = (Customer) iterator.next();
                    emitter.onNext(customer);
                };

                emitter.onComplete();

            }
        });

        return custObs;
    }
```

```
// Invoke without Subscription.
private static void fetchCustomersWOSubscribe() throws Exception {
  Observable<Customer>    custObs = fetchCustomers();
}
```

Invoke with subscription and fetching only 3 records.

```
private static void fetchCustomersWithSubscribe() throws Exception {
    // Invoke with subscription.
    Observable<Customer> custObs = fetchCustomers().take(3);
    custObs.subscribe(
                cust -> log(cust.getName()),
                err -> log(err)
               );


}
```

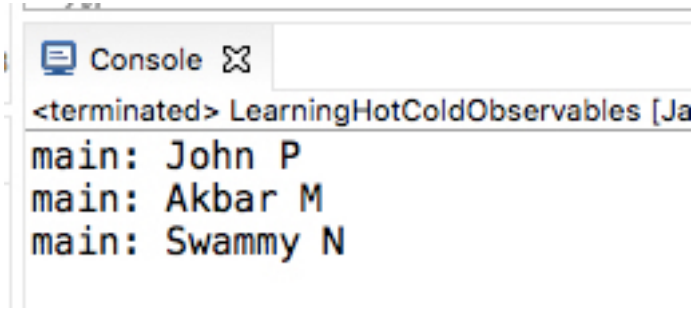
Invoke the above method one at a time.

```
 fetchCustomersWOSubscribe();
```

What do you observe? Why?

```
fetchCustomersWithSubscribe();
```

What do you observe in this case?



Printing 3 customers name. Invoke with subscription – Hints.

Hot Observables

1. Observables that don't wait for any subscription. They start emitting items when created.
2. They don't emit the sequence of items again for a new subscriber.
3. When an item is emitted by hot observable, all the subscribers that are subscribed will get the emitted item at once.

```
/*
     * Hold Observable – emit Before it is subscribe.
     * Fetch tweet with random thread sleep to simulate the real scenario, where
tweet can
     * emit randomly.
     */
    public  static Observable<Tweet> fetchTweets() throws Exception {

        Observable<Tweet> custObs = Observable.create(new
ObservableOnSubscribe<Tweet>() {

            @Override
            public void subscribe(ObservableEmitter<Tweet> emitter) throws
```

```
Exception {

                    List<Tweet> tweets = TweetServices.fetchTweetsFromSource();
                    for (Iterator<Tweet> iterator = tweets.iterator();
iterator.hasNext();) {
                        Tweet tweet = (Tweet) iterator.next();
                        log("Emitting " + tweet.getText());
                        emitter.onNext(tweet);
                        Thread.sleep(1000, 5000);
                    };

                    emitter.onComplete();

                }
            });

            return custObs;
        }

/*
         * Method to generate random number. Use in Thread sleep randomly.
         */
        public static int getRandomInteger(int maximum, int minimum){
            return ((int) (Math.random()*(maximum - minimum))) + minimum; }
```

Invoke the Hot observables and understand the publish and Replay method.

```
public static void understandHotObservable() throws Exception {

    Observable<Tweet> tweets = fetchTweets();
    ConnectableObservable<Tweet> hotTweets =
            //tweets.publish();
           tweets.replay();
    hotTweets.connect();
    hotTweets.subscribe( t -> log(" First Subscriber :" + t.getText()));
    Thread.sleep(200);  // Simulate Delay.
    hotTweets.subscribe( t -> log(" Subscriber Subscriber :" + t.getText()));
    Thread.sleep(5000);

}
```

Execute the method using publish()

```
102
103    public static void understandHotObservable() throws Exception {
104
105        Observable<Tweet> tweets = fetchTweets();
106        ConnectableObservable<Tweet> hotTweets =
107                tweets.publish();
108                //tweets.replay();
109        hotTweets.connect();
110
111        hotTweets.subscribeOn(Schedulers.computation()).
112              subscribe( t -> log(" First Subscriber :" + t.getText()),
113                      x -> log(x),() -> log("Done"));
114        Thread.sleep(100);  // Simulate Delay.
```

Console

<terminated> LearningHotColdObservables [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin

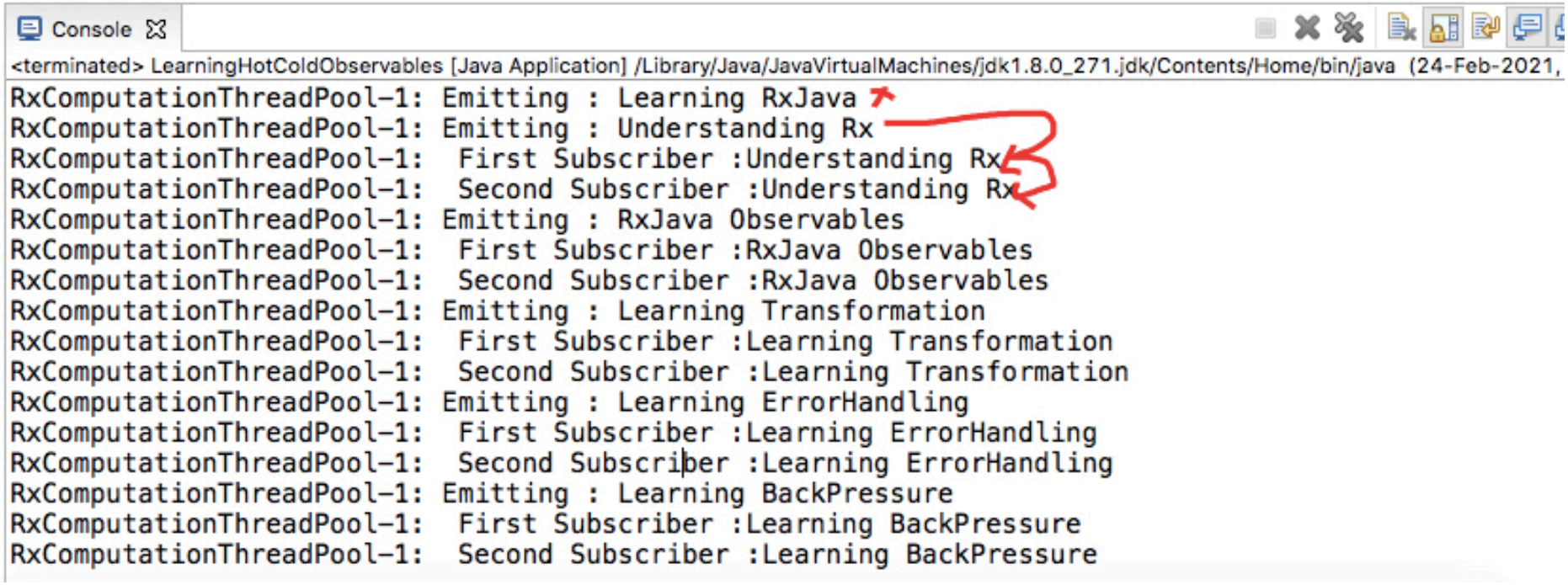
```
main: Emitting Learning RxJava
main: Emitting Understaning Rx
main: Emitting RxJava Observables
main: Emitting Learning Transformation
main: Emitting Learning ErrorHandling
main: Emitting Learning BackPressure
```

It only publishes from the Observable subscription method. No Subscribers will be consuming it, since it get registered after the item get produce.

Try adding a scheduler to the fetchTweets.

```
104      * Publish -> Consume whatever Tweet that comes after subscription.
105      * Replay  -> Consume all tweets before it subscribe too.
106      *  Try Publish and replay.
107      */
108     public static void understandHotObservable() throws Exception {
109
110         Observable<Tweet> tweets = fetchTweets().subscribeOn(Schedulers.computation());
111         ConnectableObservable<Tweet> hotTweets =
112                 tweets.publish();
113                 //tweets.replay();
```

Execute it.



As you can observe above, Learning RxJava tweet is being never consume by the First or Second ubscriber. Why? Because it is produce before the subscription.

Execute the method using replay()

```
104
105         Observable<Tweet> tweets = fetchTweets();
106         ConnectableObservable<Tweet> hotTweets =
107                 //tweets.publish();
108                 tweets.replay();
109         hotTweets.connect();
110
111         hotTweets.subscribeOn(Schedulers.computation()).
112               subscribe( t -> log(" First Subscriber :" + t.getText()),
113                       x -> log(x),() -> log("Done"));
114         Thread.sleep(100);  // Simulate Delay.
115         hotTweets.subscribe( t -> log(" Subscriber Subscriber :" + t.getText()));
116
117         Thread.sleep(5000);
```

Console

<terminated> LearningHotColdObservables [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (12-Feb-2

```
main: Emitting Learning ErrorHandling
main: Emitting Learning BackPressure
RxComputationThreadPool-1:  First Subscriber :Learning RxJava
RxComputationThreadPool-1:  First Subscriber :Understaning Rx
RxComputationThreadPool-1:  First Subscriber :RxJava Observables
RxComputationThreadPool-1:  First Subscriber :Learning Transformation
RxComputationThreadPool-1:  First Subscriber :Learning ErrorHandling
RxComputationThreadPool-1:  First Subscriber :Learning BackPressure
RxComputationThreadPool-1: Done
main:  Subscriber Subscriber :Learning RxJava
main:  Subscriber Subscriber :Understaning Rx
main:  Subscriber Subscriber :RxJava Observables
main:  Subscriber Subscriber :Learning Transformation
main:  Subscriber Subscriber :Learning ErrorHandling
```

It gets process in all the subscriber.

---------- Lab Ends Here ----------

6. Combining Observables – 60 Minutes

Let us familiarize some of the combining operators.

Create a class with static method : `LearningCombineAndAggregrate`

Import the following.

```
package ostech.observables;

import java.util.Map;

import com.tos.services.TweetServices;
import com.tos.vo.Category;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
import io.reactivex.Single;
import java.util.concurrent.TimeUnit;

import com.tos.services.TweetServices;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
```

```
import io.reactivex.Observer;
import io.reactivex.functions.BiFunction;
import io.reactivex.functions.Consumer;
import io.reactivex.functions.Function;
import io.reactivex.observables.GroupedObservable;
import io.reactivex.observers.DisposableObserver;
import io.reactivex.schedulers.Schedulers;
```

Add the following method to understand the merge.

```
/*
     *  It combine the first and second Observable as a single Observable.
     *  The two observable items may overlap.
     */
    public static void merge() {
        Observable<Integer> first = Observable.range(0,10);
        Observable<Integer> second = Observable.range(10, 20);
        Observable<Integer> merged = Observable.merge(first, second);
        merged.subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer t) throws Exception {
                // TODO Auto-generated method stub
                System.out.println("Hello " + t + "!");
            }
        });
    }
```

Execute and verify the result.

Add the following method to understand the concat.

```
/*
     *  It combine the first and second Observable as a single Observable.
     *  The two observable items wont overlap.
     */
    public static void concat() {
        Observable<Integer> first = Observable.range(0,5);
        Observable<Integer> second = Observable.range(5, 10);
        Observable<Integer> concat = Observable.concat(first, second);
        concat.subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer t) throws Exception {
                // TODO Auto-generated method stub
                System.out.println( t);
            }
        });
    }
```

Execute the method and verify its result.

Add the following method to understand the zip.

```
    /*
     * combine two observables by applying a function to the item.
     */
    public static void zip() {
        Observable<Integer> first = Observable.just(1, 2, 3);
        Observable<String> second = Observable.just("A","B", "C", "D");
        Observable<String> zipped = Observable.zip(first, second , (x,y) ->
String.valueOf(x)+y);
        Observable<String> zippedinst = first.zipWith(second, (x, y) ->
String.valueOf(x)+y);
        zippedinst.subscribe(new Consumer<String>() {
            @Override
            public void accept(String t) throws Exception {
                // TODO Auto-generated method stub
                System.out.println( t);
            }
        });
    }
```

Execute it.

Add the following method to understand the debounce.

```
 /*
  * Debounce operators emits items only when a specified timespan is passed.
  *  Emit the Tweet after that 30 ms.
  */
    public static void debounce() throws Exception {
         Observable<Tweet> it =
                 Observable.fromIterable(TweetServices.fetchTweetsFromSource());
        Thread main = Thread.currentThread();
        it.debounce(30, TimeUnit.MILLISECONDS)
        .subscribeOn(Schedulers.io())
       .subscribeWith(searchQuery());

        Thread.sleep(2000);
    }

/*
   * Part of the debounce Logic
   */
    private static Observer<Tweet> searchQuery() {

        DisposableObserver<Tweet> dT =    new DisposableObserver<Tweet>() {

            @Override
            public void onNext(Tweet t) {
                TweetServices.log(t);
```

```
            }

            @Override
            public void onError(Throwable e) {
                // TODO Auto-generated method stub

            }

            @Override
            public void onComplete() {
                // TODO Auto-generated method stub

            }

        };
        return dT;
    }
```

Execute it.

Add the following method to understand the skip.

```
/*
     *   It will skip 2 record and display the item.
     */
    public static void skip() throws Exception {
        Observable<Tweet> it =
                Observable.fromIterable(TweetServices.fetchTweetsFromSource());
       // Display after Two Tweets.
        it.skip(2)
        .subscribeWith(searchQuery());

        System.out.println("----------");
        // Display without the last 2 Tweets.
        it.skipLast(2)
        .subscribeWith(searchQuery());

        System.out.println("----------");
        // Display the First 2 Tweets only.
        it.take(2)
        .subscribeWith(searchQuery());

        System.out.println("----------");
        // Display the last 2 Tweets only.
        it.takeLast(2)
        .subscribeWith(searchQuery());
```

```
        Thread.sleep(2000);
    }
```

Execute it.

Add the following method to understand the distinct.

```
    /*
     * Remove the Distinct Tweet.
     * Steps:
     *  Add or uncomment a same Tweet in the TweetServices.
     *  Execute the method without equal or define the Key as shown below.
     *  Add equals and observe the output.
     */
    public static void distinctOperator() throws Exception{

        Observable<Tweet> it =
                Observable.fromIterable(TweetServices.fetchTweetsFromSource());
        it.distinct( t -> t.getTweetID())
         .subscribeWith(searchQuery());
        Thread.sleep(2000);

    }
```

Execute it.

Add the following method to understand the count.

```
    /*
     * Determine the count of the Observable.
     *
     */
    public static void count() throws Exception{

        Observable<Tweet> it =
                Observable.fromIterable(TweetServices.fetchTweetsFromSource());
        it.distinct( t -> t.getTweetID())
          .count()
          .subscribe(System.out::println);
         Thread.sleep(1000);

    }
```

Add the following method to understand the reduce.

```
    /*
     * Apply reduce to sum the IDs of the Tweet.
     */
 public static void reduce() throws Exception{

        Observable<Tweet> it =
                Observable.fromIterable(TweetServices.fetchTweetsFromSource());
        Observable<Tweet> dT =    it.distinct( t -> t.getTweetID());
```

```
        Observable<String> tId = dT.map(t -> t.getTweetID());
         tId.reduce( new BiFunction<String, String, String>() {

             @Override
             public String apply(String t1, String t2) throws Exception {
                 // TODO Auto-generated method stub
                 return String.valueOf( Integer.parseInt(t1) +
Integer.parseInt(t2));
             }
        } )
          .subscribe(System.out::println);
         Thread.sleep(1000);

    }
```

Add the following method to understand the defer.

```
 /*
  *  Emit the item after subscription only.
  */
 public static void defer() throws Exception{

        Observable<Tweet> it =
                Observable.fromIterable(TweetServices.fetchTweetsFromSource());
        /*
         * Defer the Item emission till it get subsription
         */
```

```
Observable.defer(() -> it);
// .subscribe(System.out::println);

 class SomeType {
  private String value;

  public void setValue(String value) {
    this.value = value;
  }

  /*
   *  Just assign value at the creation time
   *  while defer initialize laster on subscription
   */
  public Observable<String> valueObservable() {
     // return Observable.defer( () -> Observable.just(value));
     return Observable.just(value);
  }
}

 SomeType instance = new SomeType();
 Observable<String> value = instance.valueObservable();
 instance.setValue("Learning rx Defer");
 value.subscribe(System.out::println);
/*
 * Emit Item after 2 seconds.
 */
```

```
        Observable.timer(2, TimeUnit.SECONDS)
          .flatMap( l ->  Observable.just(new Tweet("12","Operator")))
          .subscribe(System.out::println);
         Thread.sleep(5000);

    }
```

Execute it

Add the following method to understand the GroupBy.

```
 /*
  * Grouping logic needs to be defined in the Function that is passed to groupBy
operator.
  *  In the Function, you need to define logic that identifies group
  *  for each item emitted by source observable and return group id for the item.
  *
  * Below example, groups Tweet into even and odd groups,
  * and prints even group by subscribing to even GroupedObservable.
  */
  public static void groupBy() {
      Observable<Tweet> it =
                  Observable.fromIterable(TweetServices.fetchTweetsFromSource());
      Observable<GroupedObservable<String, Tweet>> groupedObservable =
              it.groupBy(new Function<Tweet, String>() {

                  @Override
                  public String apply(Tweet t) throws Exception {
```

```
                    if( Integer.valueOf(t.getTweetID()) % 2 > 0){
                        return "Odd Number";
                    }else{
                        return "Even Number";
                    }
                }
            });

    groupedObservable.subscribe(s -> {
       TweetServices.log("grouped observable item key is  "+s.getKey());
      if("Even Number".equals(s.getKey())){
          s.subscribe ( groupedObservavleItem -> {
              TweetServices.log("Items from the even number group observable "
             + groupedObservavleItem);});
      } /*else {
          s.subscribe ( groupedObservavleItem -> {
              TweetServices.log("Items from the Odd number group observable "
             + groupedObservavleItem);});
      } */
      });
}
```

Add the following method to understand the scan.

```
  /*
   * Apply a function recursively and emit the item individually.
   * reduce() -> Apply the fuction and return the final value.
   *
   */
  public static void scan() throws Exception {
      Observable<String> myObservable = Observable.just("a", "b", "c", "d", "e");
      myObservable.scan((x, y) -> x + y)
                  .subscribe(System.out::println);
  }
```

Add the following method

```
// May Interleave
    public static void understandMerge() {
        Observable<Integer> first = Observable.range(0,5);
        Observable<Integer> second = Observable.range(5,5);
        Observable<Integer> merged
             = Observable.merge(first, second);
       merged.subscribe(System.out::println);
    }
```

The above method merge the content of the first and second Observable.

```
30
31      // No Interleave
32      public static void understandConcat() {
33          Observable<Integer> first = Observable.range(0,5);
34          Observable<Integer> second = Observable.range(5,5);
35          Observable<Integer> concat
36              = Observable.concat(first, second);
37          concat.subscribe(System.out::println);
38      }
39
40      // Combine and Emmit single item.
41      public static void understandzip() {
```

```
Console
<terminated> LearningCombineObs [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java  (10
0
1
2
3
4
5
6
7
8
9
```

As you can observe the two observables are merge into single. It may interleave the item from different observables.

The following function using concat, will never interleave the items in the final observable.

```
// No Interleave
    public static void understandConcat() {
        Observable<Integer> first = Observable.range(0,5);
        Observable<Integer> second = Observable.range(5,5);
        Observable<Integer> concat
            = Observable.concat(first, second);
       concat.subscribe(System.out::println);
    }
```

Execute the method.

```
31      // No Interleave
32      public static void understandConcat() {
33          Observable<Integer> first = Observable.range(0,5);
34          Observable<Integer> second = Observable.range(5,5);
35          Observable<Integer> concat
36              = Observable.concat(first, second);
37          concat.subscribe(System.out::println);
38      }
39
40      // Combine and Emmit single item.
41      public static void understandzip() {
42          Observable<Integer> first = Observable.just(1,2,3,4,5,6);
43          Observable<String> second = Observable.just("A","B","C","D","E");
44          Observable<String> zipped
45              = Observable.zip(first, second
```

Console

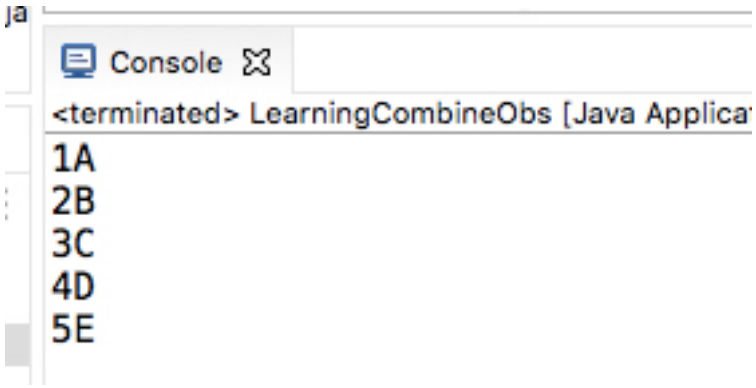
<terminated> LearningCombineObs [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bi

```
0
1
2
3
4
5
6
7
8
9
```

Next, it will combine and emit a single item using a combine function

```
// Combine and Emmit single item.
     public static void understandzip() {
          Observable<Integer> first = Observable.just(1,2,3,4,5,6);
          Observable<String> second = Observable.just("A","B","C","D","E");
          Observable<String> zipped
               = Observable.zip(first, second ,
                         (x,y) -> String.valueOf(x)+y);
          zipped.subscribe(System.out::println);
     }
```

Execute the method. You should observe as shown below.



Next let us add a method that will perform the followings:

- Fetch tweets from two sources
- Fetch categories for each tweet.
- Merge the first two observables and determine the category of each tweet using HashMap.

```
public static void tweetCategory() {

            // Get Tweets from different sources – Movies and IT
            Observable<Tweet> it =
Observable.fromIterable(TweetServices.fetchTweetsFromSource());
            Observable<Tweet> movies =
Observable.fromIterable(TweetServices.fetchTweetsFromMovieSource());

            // Combine all the tweets.
            Observable<Tweet> comTweets
                 = Observable.concat(it, movies);

            Observable<Category> categories =
                     Observable.fromIterable(TweetServices.fetchCategories());



          Single<Map<Object, Category>> catMap =
                  categories.toMap(cat -> cat.getCategoryID());
```

```
            Observable<String> combResult =comTweets.flatMap( tweet -> {
               String cate =
catMap.blockingGet().get(tweet.getCategory()).getCategoryText();
                return Observable.just(tweet.getText() + " : " + cate);
            });

            combResult.subscribe(System.out::println);

        }
```

Execute the method.

```
18      //understandzip();
19          tweetCategory();
20      }
```

Console

<terminated> LearningCombineObs [Java Application] /Library/Java/JavaVirtualMachines

```
Observable thread: main
Learning RxJava : IT
Understaning Rx : IT
RxJava Observables : IT
Learning Transformation : IT
Learning ErrorHandling : IT
Learning BackPressure : IT
Braveheart : Movie
Matrix reload : Movie
```

Add the Datasource as shown below:

```
package com.tos.services;

import java.util.ArrayList;
import java.util.List;

import com.tos.vo.Category;
import com.tos.vo.Tweet;

public class TweetServices {
```

```
    public static List<Tweet> fetchTweetsFromSource() {
        List<Tweet> tweetList = new ArrayList<Tweet>();
        tweetList.add(new Tweet("1","Learning RxJava","1"));
        tweetList.add(new Tweet("2","Understaning Rx"));
        tweetList.add(new Tweet("3","RxJava Observables"));
        tweetList.add(new Tweet("4","Learning Transformation"));
        tweetList.add(new Tweet("5","Learning ErrorHandling"));
        tweetList.add(new Tweet("6","Learning BackPressure"));
        return tweetList;
    }

    public static List<Tweet> fetchTweetsFromMovieSource() {
        List<Tweet> tweetList = new ArrayList<Tweet>();
        tweetList.add(new Tweet("11","Braveheart","2"));
        tweetList.add(new Tweet("12","Matrix reload","2"));
        return tweetList;
    }

    public static List<Tweet> fetchTweet() {
        List<Tweet> tweetList = new ArrayList<Tweet>();
        tweetList.add(new Tweet("1","Learning RxJava"));
         System.out.println("Observable thread: " +
Thread.currentThread().getName());
        return tweetList;
    }
```

```
    public static List<Category> fetchCategories() {
        List<Category> catList = new ArrayList<Category>();
        catList.add(new Category("1","IT"));
        catList.add(new Category("2","Movie"));
        System.out.println("Observable thread: " +
Thread.currentThread().getName());
        return catList;
    }


}
```

It reflects each Tweet with that of the category. i.e are IT and Movie respectively.

Let us understand some of the conditional Operators.

```
    /*
     * Does all items less then 10, if yes then success or false.
     */
  static void all() throws Exception {
        Observable.range(1, 9)
                  .all( r -> r < 10)
```

```
                    .subscribe(System.out::println);
    }
```

Add the above and execute it.

Add the next method and execute it.

```
/*
   *  Understand skip - it will skip as long as the item is less than 5
   */
    static void understandSkip() throws Exception {

        Observable.range(1, 9)
           .skipWhile( el -> el < 5)
           .take(3)
           .reduce((el1,el2) -> el1 + el2)
           .subscribe(System.out::println);
    }
```

---------- Lab Ends Here ----------

7. Demo Operators – TweetRx – 60 Minutes

In this use case,

Given an author or username, it will perform the following:

1. Fetch the User Profile Details
2. Fetch the Tweet of the above user.
3. All the above 1 & 2 will be performed in a separate thread.
4. Perform the reduce operator on the 2 step to determine the Tweet with that of the highest retweet count.
5. Perform combine operation using zip to form a single object with Profile and Tweet Details.

Create a maven Project.

Group ID : com.ostech
Artifact ID: TweetRx.

Update the pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ostech</groupId>
  <artifactId>TweetRx</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.22</version>
        </dependency>
        <dependency>
            <groupId>io.reactivex.rxjava2</groupId>
            <artifactId>rxjava</artifactId>
            <version>2.2.6</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.google.guava/guava -->
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
```

```
            <version>30.1-jre</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.slf4j/log4j-over-slf4j -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>log4j-over-slf4j</artifactId>
            <version>2.0.0-alpha1</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Domain Objects.

```
class Tweet {
    String text;
    int favorite_count;
    String author;
    int author_followers;
}
```

```
class Profile {
    String screen_name;
    String name;
    String location;
    int statuses_count;
    int followers_count;
}
```

```
class UserWithTweet {
    Profile profile;
    Tweet tweet;
}
```

Define these classes in a package.

```java
package com.osttech.vo;

public class Profile {

    String screen_name;
    String name;
    String location;
    int statuses_count;
    int followers_count;
    public Profile() {
        super();
        // TODO Auto-generated constructor stub
    }
    public Profile(String screen_name, String name, String location, int
statuses_count, int followers_count) {
        super();
        this.screen_name = screen_name;
        this.name = name;
        this.location = location;
        this.statuses_count = statuses_count;
        this.followers_count = followers_count;
    }
    public String getScreen_name() {
        return screen_name;
    }
    public void setScreen_name(String screen_name) {
        this.screen_name = screen_name;
```

```
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getLocation() {
    return location;
}
public void setLocation(String location) {
    this.location = location;
}
public int getStatuses_count() {
    return statuses_count;
}
public void setStatuses_count(int statuses_count) {
    this.statuses_count = statuses_count;
}
public int getFollowers_count() {
    return followers_count;
}
public void setFollowers_count(int followers_count) {
    this.followers_count = followers_count;
}
@Override
public String toString() {
```

```
        return "Profile [screen_name=" + screen_name + ", name=" + name + ",
location=" + location + ", statuses_count="
                + statuses_count + ", followers_count=" + followers_count + "]";
    }


}
```

```
package com.osttech.vo;

public class Tweet {

    String text;
    int favorite_count;
    String author;
    int author_followers;
    public int retweet_count;
    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }
    public int getFavorite_count() {
        return favorite_count;
    }
```

```
    public void setFavorite_count(int favorite_count) {
        this.favorite_count = favorite_count;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public int getAuthor_followers() {
        return author_followers;
    }
    public void setAuthor_followers(int author_followers) {
        this.author_followers = author_followers;
    }
    public Tweet(String text, int favorite_count, String author, int
author_followers,int ret_count) {
        super();
        this.text = text;
        this.favorite_count = favorite_count;
        this.author = author;
        this.author_followers = author_followers;
        this.retweet_count = ret_count;
    }
    public Tweet() {
        super();
        // TODO Auto-generated constructor stub
```

```
    }
    @Override
    public String toString() {
        return "Tweet [text=" + text + ", favorite_count=" + favorite_count + ",
author=" + author
                + ", author_followers=" + author_followers + ", retweet_count=" +
retweet_count + "]";
    }



}
```

UserWithTweet.java

```
package com.osttech.vo;

public class UserWithTweet {

    Profile profile;
    Tweet tweet;
    public UserWithTweet() {
        super();
        // TODO Auto-generated constructor stub
    }
    public UserWithTweet(Profile profile, Tweet tweet) {
```

```
        super();
        this.profile = profile;
        this.tweet = tweet;
    }
    public Profile getProfile() {
        return profile;
    }
    public void setProfile(Profile profile) {
        this.profile = profile;
    }
    public Tweet getTweet() {
        return tweet;
    }
    public void setTweet(Tweet tweet) {
        this.tweet = tweet;
    }
    @Override
    public String toString() {
        return "UserWithTweet [profile=" + profile + "\n, tweet=" + tweet +
                //",\n getProfile()=" + getProfile()
                //+ "\n, getTweet()=" + getTweet() +
                "]";
    }


}
```

```java
package com.ostech.bo;

import java.util.ArrayList;
import java.util.List;

import com.osttech.vo.Tweet;



public class TweetRepository {

    public static List<Tweet> fetchTweetsFromSource() {
        List<Tweet> tweetList = new ArrayList<Tweet>();
        tweetList.add(new Tweet("Learning RxJava",15,"rx_java",128,3));
        tweetList.add(new Tweet("Learning Kafka",25,"kafka",2128,4));
        tweetList.add(new Tweet("Learning Spark",35,"spark",3128,5));
        tweetList.add(new Tweet("Rx Error handling",45,"rx_java",4128,2));
        tweetList.add(new Tweet("Kafka Quorum",15,"kafka",5128,7));
        tweetList.add(new Tweet("Kafka Client",75,"kafka",6128,6));
        tweetList.add(new Tweet("Spark Dataframe",85,"spark",7128,5));
        tweetList.add(new Tweet("Rx Backpressure",95,"rx_java",8128,6));
        return tweetList;
    }
}
```

A repository to simulate the Tweet data source.

User.java

Class that fetch the Details of Users from the Repository and Apply Business Logic on It.

Import the following:

```
package com.ostech.bo;

import com.osttech.vo.Profile;
import com.osttech.vo.Tweet;

import io.reactivex.Observable;
```

Add the following method:

```
/*
     * Get Profile Details Using ScreenName - Sync or Traditional Ways
     */
    public Profile getUserProfile(String screenName) {

        Profile prof = null;
        switch (screenName) {
        case "rx_java":
            prof = new Profile("rx_java", "Rx Java", "Imphal", 3, 120);
            break;
        case "kafka":
```

```
            prof = new Profile("kafka", "Apache kafka", "Mumbai", 43, 80020);
            break;
        default:
            prof = new Profile("kafka", "Apache kafka", "Mumbai", 23, 220);
            break;
        }

        return prof;
    }
```

```
/*
     * Get Profile Using Observable
     */
    public Observable<Profile> getUserProfileAsync(String screenName) {
        return Observable
                .just(screenName)
                .flatMap(sn -> {
                  Profile prof = null;
                    switch (screenName) {
                    case "rx_java":
                        prof = new Profile("rx_java", "Rx Java", "Imphal", 3,
120);
                        break;
                    case "kafka":
                        prof = new Profile("kafka", "Apache kafka", "Mumbai", 43,
80020);
                        break;
```

```
                    default:
                        prof = new Profile("kafka", "Apache kafka", "Mumbai", 23,
220);
                        break;
                    }
                    return Observable.just(prof);
                });
        }
User profile information using Observable.
```

Add the following - Get Profile Using Observable along with Exception handling.

```
/*
     * Get Profile Using Observable along with Exception handling.
     *
     * parmeter : auth - true (Authenticated) | false (Not Authenticated)
     */
    public Observable<Profile> getUserProfileAsyncErrorsH(String screenName,
boolean auth) {
        if(auth) {
            return Observable
                    .just(screenName)
                    .flatMap(sn -> {
                      Profile prof = null;
                        switch (screenName) {
                        case "rx_java":
```

```
                                prof = new Profile("rx_java", "Rx Java", "Imphal",
3, 120);
                                break;
                            case "kafka":
                                prof = new Profile("kafka", "Apache kafka",
"Mumbai", 43, 80020);
                                break;
                            default:
                                prof = new Profile("Spark", "Apache Spark",
"Mumbai", 23, 220);
                                break;
                            }
                            return Observable.just(prof);
                        });
        }else {
            return Observable.error(new RuntimeException("Can not connect to
twitter" ));
        }
    }
```

Next Method: Get all Tweets for a Specific Author.

```
/*
     * Get all Tweets for a Specific Author.
     */
```

```
    public Observable<Tweet> getUserRecentTweets(String author) {
        Observable<Tweet> tweets =
Observable.fromIterable(TweetRepository.fetchTweetsFromSource());
        Observable<Tweet> auT = tweets.filter( t -> t.getAuthor().equals(author));
        return auT;
    }
```

TweetGateway – The Client Class.

Import the following:

```
package com.ostech.client;

import com.ostech.bo.User;
import com.osttech.vo.Profile;
import com.osttech.vo.Tweet;
import com.osttech.vo.UserWithTweet;

import io.reactivex.Maybe;
import io.reactivex.Observable;
import io.reactivex.functions.BiFunction;
import io.reactivex.schedulers.Schedulers;
```

Add the following method. It performs the following

- Get the User Profile in a Separate Thread - IO Thread.

- Get the Tweet for the Author and Fetch the one with the highest Tweet Count - Use Reduce operator and Thread IO
- Combine the Profile and Tweet for a user using zip Operator.

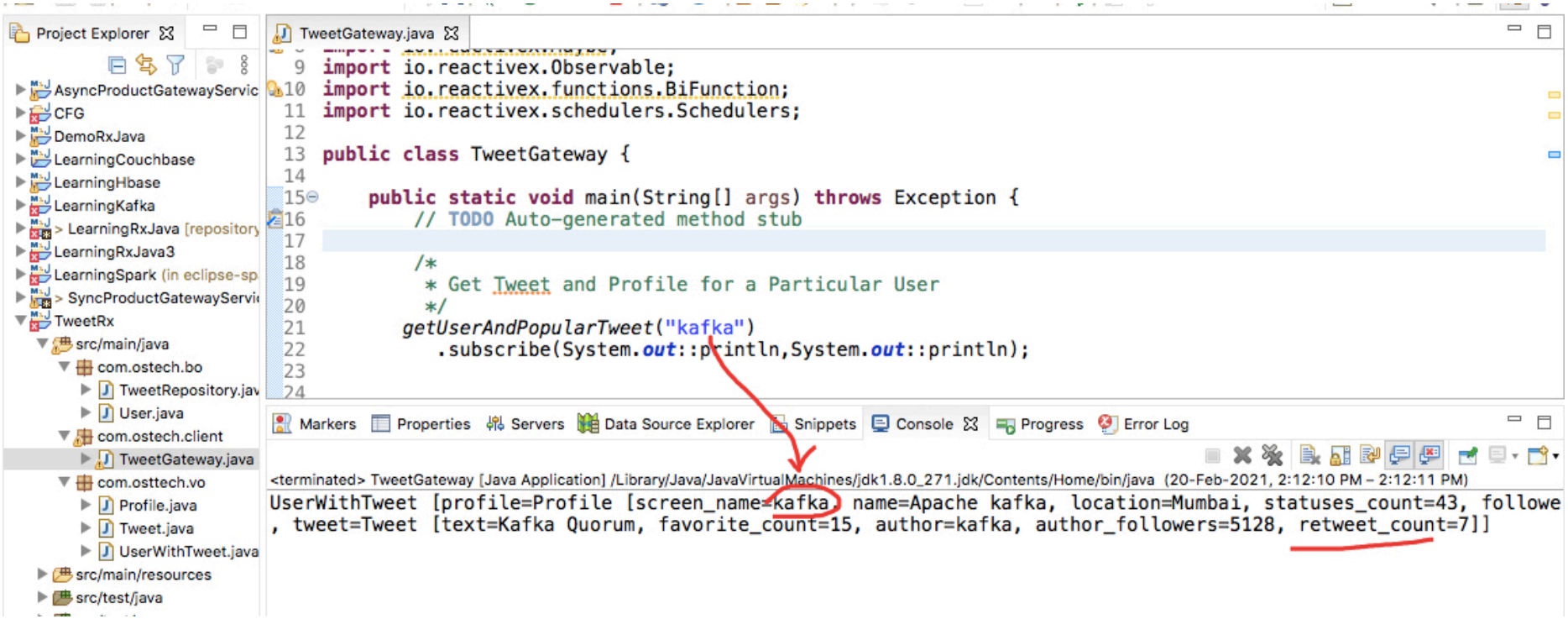
```
    /*
     *  Get the User Profile in a Separate Thread – IO Thread.
     *  Get the Tweet for the Author and Fetch the one with the highest Tweet
Count – Use Reduce operator and Thread IO
     *  Combine the Profile and Tweet for a user using zip Operator.
     */
    static Observable<UserWithTweet> getUserAndPopularTweet(String author){
        User client = new User();
        return Observable.just(author)
        .flatMap(u -> {
                Observable<Profile> profile =
                    client
                    // .getUserProfileAsyncErrorsH(u, false)  // Simulate Error
Handling with this Method
                     .getUserProfileAsync(u)
                    .subscribeOn(Schedulers.io()); // Using Seperate IO Thread

        Observable<Tweet> tweet = client.getUserRecentTweets(u)
                    .reduce((t1, t2) ->     // Return the Tweet with the Highest
retweet count.
                        t1.retweet_count > t2.retweet_count ? t1 : t2) // As it
return Maybe; convert into Observable
```

```
                        .toObservable()   // Converting from Maybe to Observable
                        .subscribeOn(Schedulers.io());
                    return Observable.zip(profile,tweet, UserWithTweet::new);
                    });
        }
```

Now invoke the above method from the main method();

```
public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub

        /*
         * Get Tweet and Profile for a Particular User
         */
       getUserAndPopularTweet("kafka")
          .subscribe(System.out::println,System.out::println);


       Thread.sleep(500); // Waiting for the Observable to complete.
    }
```

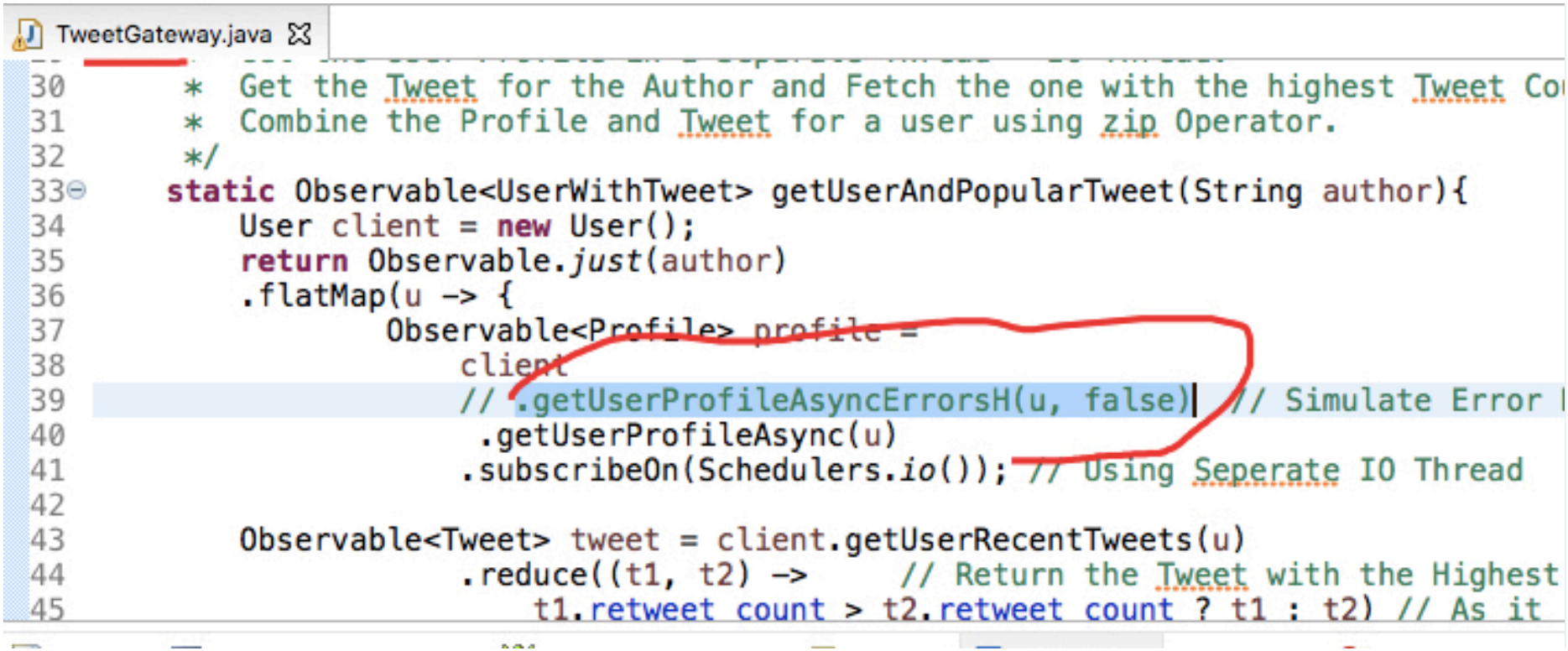
Execute the above.



Verify the Output,
Here, Profile details is of kafka User and the “Kafka Quorom” tweet, which have the highest retweet count.

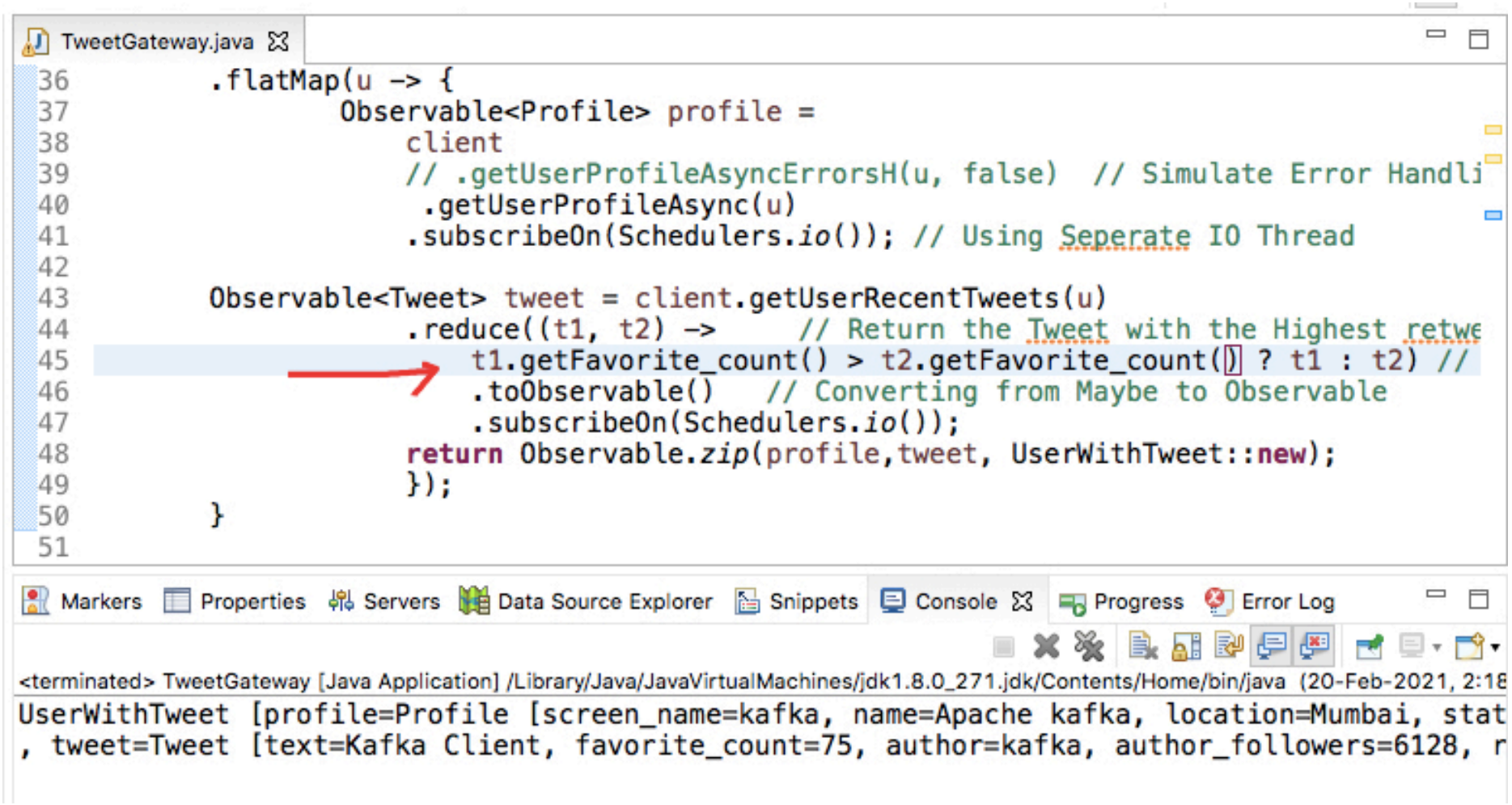
Tasks:

Try changing the user to rs_java and execute it. Try to understand the logic flow.
Try using the `.getUserProfileAsyncErrorsH(u, false) instead of without error.`



Return the tweet with the Highest Favorite count?

Hints:



---------- Lab Ends Here ----------

8. Case Study – Microservices – 3 Hrs

Use cases:

To Fetch the Product details. It will have the following classes.

Client : To get the product information by Specifying the product ID. The Main Entry client.

ProductGateway :- Entry point for getting the product details. A gateway service to Product. It assembles the Product information fetching various information from the following applications or services.

- **ProductService** :- It use **ProductDAO** to fetch the basic Product information.
- **PriceDAO** :- It fetch the Price of the Product.
- **InventoryDAO** :- It fetch the inventory information of the Product.

Gateway uses the above classes to derive the Product information. In real scenarios, these classes can be different system or application.

Model or value classes are declared inside the package: `com.ostech.product.model`

In this use case,

- Basic information, Price and Inventory of the Product are maintained in different system.

- A Gateway is use to construct the Product by fetching information from the 3 different systems mention above.
- Finally, a client will invoke the gateway to derive the product information.

We can design the system in the following ways:

- Synchronously
- Async.

Case I – Sync Mode.
Synchronous Product Gateway Services.

Create a simple maven project with the following information.

Maven Project Details.
Group ID: com.ostech
Artifact : SyncProductGatewayService
Version : snapshot

Update the pom.xml with the following content.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.ostech</groupId>
    <artifactId>SyncProductGatewayService</artifactId>
    <version>0.0.1-SNAPSHOT</version>
<dependencies>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.22</version>
        </dependency>
    </dependencies>
</project>
```

Some information or data is required for this exercise. Install DB and Insert all the require data.

You can have MYSQL as a separate system or install in the same machine you are developing.

If you are using Docker to install MySQL server, then follow the steps given below:

- Using Docker
- Install Docker

- Install Mysql container.
- You can refer the link
    - https://www.percona.com/blog/2019/11/19/installing-mysql-with-docker/

Docker command to instantiate the Mysql Server.

```
docker run --name mysql-latest  \
-p 3306:3306 -p 33060:33060  \
-e MYSQL_ROOT_HOST='%' -e MYSQL_ROOT_PASSWORD='root213'   \
-d mysql/mysql-server:latest

docker exec -it mysql-latest bash
mysql -u root -p
Enter password : root213

docker start mysql-latest
```

The above commands, start the mysql server and access the server using the CLI.

On Windows server (Refer the following link or any online tutorial):
https://www.liquidweb.com/kb/install-mysql-windows/

Create the following tables:

Table	Remarks
Product	It stores the Product basic information
Options	It stores the details information including the price.

Connect to CLI and execute the following commands.

```
/****************************** DDL & DML ***************************************/

CREATE DATABASE ostech;
use ostech;

CREATE TABLE product (
   id int,
   description varchar(255)
);


CREATE TABLE options (
   prodid int,
   optionid int,
   description varchar(255),
   price float,
```

```
   inventory int
);

// Insert into scripts

// Products Data.

INSERT INTO product VALUES (1,'Car');
INSERT INTO product VALUES (2,'Laptop');
INSERT INTO product VALUES (3,'Tablet');

INSERT INTO options VALUES (1,1,'Basic',6000000,12);
INSERT INTO options VALUES (1,2,'Stereo',8000,7);
INSERT INTO options VALUES (1,3,'GPS',6000,10);

INSERT INTO options VALUES (2,1,'Basic',200000,20);
INSERT INTO options VALUES (2,2,'Mouse',2000,5);
INSERT INTO options VALUES (2,3,'Bag',6000,10);

INSERT INTO options VALUES (3,1,'Basic',100000,10);
INSERT INTO options VALUES (3,2,'Cam',2000,5);
INSERT INTO options VALUES (3,3,'Bag',3000,10);

/****************************** DDL & DML ******************************************/
```

Create the following model classes:

- `BaseProduct`
- `BaseOption`
- `Option`
- `Product`

It contained the information or data of the Product and its Option.

```
package com.ostech.product.model;

public abstract class BaseOption {

    public  Long id = null;
    public String description = "";
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
```

```
    }


}
```

```
package com.ostech.product.model;

import java.util.List;

public abstract class BaseProduct {

    public Long id;
    public String description;
    public List<BaseOption> baseOptions;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
```

```
    public List<BaseOption> getBaseOptions() {
        return baseOptions;
    }
    public void setBaseOptions(List<BaseOption> baseOptions) {
        this.baseOptions = baseOptions;
    }


}
```

```
package com.ostech.product.model;

public class Option extends BaseOption {

    private Double price;
    private Integer inventory;
    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
        this.price = price;
    }
    public Integer getInventory() {
        return inventory;
    }
    public void setInventory(Integer inventory) {
        this.inventory = inventory;
```

```
    }


}
```

```
package com.ostech.product.model;

import java.util.List;

public class Product extends BaseProduct {

    private Integer totalInventory;
    private Double displayPrice;
    private List<Option> options;

    public Integer getTotalInventory() {
        return totalInventory;
    }

    public void setTotalInventory(Integer totalInventory) {
        this.totalInventory = totalInventory;
    }

    public Double getDisplayPrice() {
        return displayPrice;
    }
```

```
    public void setDisplayPrice(Double displayPrice) {
        this.displayPrice = displayPrice;
    }

    public List<Option> getOptions() {
        return options;
    }

    public void setOptions(List<Option> options) {
        this.options = options;
    }

}
```

Now we have created all the classes that will be used as Data transfer or Value Object. They are just an abstract of our domain data.
Let us define classes to connect to each of the different system which have databases in the backend.

InventoryDAO.class : Fetching inventory details from the Inventory system.

```
package com.ostech.data.access;

import java.sql.Connection;
```

```
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class InventoryDAO {

    public Integer getInventory(Long productId, Long optionId) throws SQLException
{
        Integer inventory = 0;
        String sql = "select inventory from options where prodid=? and optionid
=?";

        try
          {
             String url = "jdbc:mysql://localhost:3306/ostech";
            Connection conn = DriverManager.getConnection(url,"root","root213");

           // create the prepared statement and add the criteria
           PreparedStatement ps = conn.prepareStatement(sql);
           ps.setLong(1, productId );
           ps.setLong(2, optionId );
           // process the results
           ResultSet rs = ps.executeQuery();
           while ( rs.next() )
           {
             inventory = rs.getInt("inventory");
```

```
            }
            rs.close();
            ps.close();
          }
          catch (SQLException se)
          {
            // log exception;
            throw se;
          }
        return inventory;

    }
}
```

PriceDAO.class : Fetch the Price details of a product.

```
package com.ostech.data.access;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class PriceDAO {

    public Double getPrice(Long productId, Long optionId) throws SQLException {
        Double price = 0.0;
        String sql = "select price from options where prodid=? and optionid =?";

        try
          {
             String url = "jdbc:mysql://localhost:3306/ostech";
            Connection conn = DriverManager.getConnection(url,"root","root213");

           // create the preparedstatement and add the criteria
           PreparedStatement ps = conn.prepareStatement(sql);
           ps.setLong(1, productId );
           ps.setLong(2, optionId );
           // process the results
           ResultSet rs = ps.executeQuery();
```

```
            while ( rs.next() )
            {
              price = rs.getDouble("price");
            }
            rs.close();
            ps.close();
          }
          catch (SQLException se)
          {
            // log exception;
            throw se;
          }
        return price;

    }
}
```

ProductDAO.class : Fetch the Product Base information.

```
package com.ostech.data.access;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import com.ostech.product.model.BaseOption;
import com.ostech.product.model.BaseProduct;
import com.ostech.product.model.Option;
import com.ostech.product.model.Product;

public class ProductDAO {

    public BaseProduct getBaseProduct(Long productId) throws SQLException {
        Product prod = new Product();
        String sql ="select a.description ad,b.prodid,b.optionid,b.description bd
from product a, options b where a.id = b.prodid and a.id =?";
        List<Option> optionList = new ArrayList<Option>();
        List<BaseOption> boptionList = new ArrayList<BaseOption>();
        Option bo = null;
        try
```

```
{
    String url = "jdbc:mysql://localhost:3306/ostech";
   Connection conn = DriverManager.getConnection(url,"root","root213");
  // create the preparedstatement and add the criteria
  PreparedStatement ps = conn.prepareStatement(sql);
  ps.setLong(1, productId );

  // process the results
  ResultSet rs = ps.executeQuery();
  boolean flag = true;
  while ( rs.next() )
  {
  if(flag) {
       prod.id = rs.getLong("prodid");
       prod.description = rs.getString("ad");
       flag = false;
  }
   bo = new Option();
       bo.setDescription(rs.getString("bd"));
       bo.setId(rs.getLong("optionid"));
       optionList.add(bo);
       boptionList.add(bo);
  }
  rs.close();
  ps.close();
}
catch (SQLException se)
```

```
            {
              // log exception;
              throw se;
            } finally {
                prod.setOptions(optionList);
                prod.setBaseOptions(boptionList);
            }
          return prod;

    }

}
```

The above 3 classess, connect to the backend databases tables define in a mysql Server.

Next we will create a Business layer.

ProductService.class -> It just invoke the Data Access Layer to fetch the base information. Here, you can add your business logic.

```
package com.ostech.product.services;

import java.sql.SQLException;

import com.ostech.data.access.ProductDAO;
import com.ostech.product.model.BaseProduct;

public class ProductService {
    ProductDAO prodDAO = new ProductDAO();
    BaseProduct getBaseProduct(Long productId) throws SQLException {

        return prodDAO.getBaseProduct(productId);
    }
}
```

Finally add the ProductGateway.class. Assembling of Product will happens here.

Import the following packages.

```
package com.ostech.product.services;

import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import com.ostech.data.access.InventoryDAO;
import com.ostech.data.access.PriceDAO;
import com.ostech.product.model.BaseOption;
import com.ostech.product.model.BaseProduct;
import com.ostech.product.model.Option;
import com.ostech.product.model.Product;
```

Declare the variables. It just instantiates the objects.

```
ProductService baseProductClient = new ProductService();
    PriceDAO priceClient = new PriceDAO();
    InventoryDAO inventoryClient = new InventoryDAO();
```

Add the following method.

```
public Product getProduct(Long productId) throws SQLException {

        BaseProduct bp = baseProductClient.getBaseProduct(productId); // Blocking
WS Call
        Double displayPrice = Double.MAX_VALUE;
```

```
        List<Option> optionList = new ArrayList<Option>();
        int totalInventory = 0;
        Option  option = null;
        Double totalPrice = new Double(0.0);
        for (BaseOption bo : bp.getBaseOptions()) {

            Double price = priceClient.getPrice(bp.id,bo.getId()); // Blocking WS
Call
            Integer inventory =
inventoryClient.getInventory(bp.getId(),bo.getId()); // Blocking WS Call
            option = new Option();
            option.setDescription(bo.getDescription());
            option.setId(bo.getId());
            option.setInventory(inventory);
            option.setPrice(price);
            optionList.add(option);
            totalInventory += inventory;
            totalPrice = Double.sum(totalPrice, price);

        }
        Product prod = new Product();
        prod.setId(bp.getId());
        prod.setDescription(bp.getDescription());
        prod.setTotalInventory(totalInventory);
        prod.setDisplayPrice(totalPrice);
        prod.setOptions(optionList);
```

```
        return prod;
    }
```

The above method performs the following:

- Using the specified product ID, base product information is fetched along with its option.
- Each product can have various multiples option.
- For each option of the product, the following are performed:
    - Fetch Price
    - Fetch Inventory
- And the Option along with its inventory and price details are added to the Product object.

Let us invoke the product gateway and fetch the product information from a client.

```java
package com.ostech.client;

import java.sql.SQLException;

import com.ostech.data.access.PriceDAO;
import com.ostech.product.model.Product;
import com.ostech.product.services.ProductGateway;

public class Client {

    public static void main(String[] args) {

        getProduct();
        //getPrice();
    }


    public static void getProduct() {
        ProductGateway gateway = new ProductGateway();
        try {
        Product prod =   gateway.getProduct(new Long(3));
        System.out.println(">>> " + prod.getDescription()
              +" Price Option: "+ prod.getOptions().get(0).getPrice() +
              " Product: " + prod.getDisplayPrice());
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
```

```
        }

    }
    public static void getPrice() {

        PriceDAO priceDAO = new PriceDAO();
        try {
            Double price = priceDAO.getPrice(new Long("1"),new Long("2"));
            System.out.println(">>>" + price);
        } catch (NumberFormatException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        } catch (SQLException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
}
```

Execute the above class:

```
18    public static void getProduct() {
19        ProductGateway gateway = new ProductGateway();
20        try {
21        Product prod =  gateway.getProduct(new Long(3));
22        System.out.println(">>> " + prod.getDescription()
23              +" Price Option: "+ prod.getOptions().get(0).getPrice() +
24              " Product: " + prod.getDisplayPrice()
25              +" Inventory : " + prod.getTotalInventory());
26        } catch (SQLException e) {
27            // TODO Auto-generated catch block
```

Console

<terminated> Client [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (17-Feb-2021, 3:1

```
>>> Tablet Price Option: 100000.0 Product: 105000.0 Inventory : 25
```

As you can observe, For a product ID -3 , It display the Produce base description, product price and inventory of it as shown above.

Now in the Next Project, we will implement it using Async with RxJava.

Create Another Maven Project.

Here, let us invoke the following Asynchronously using RxJava's Observable.

- Inventory
- Price &
- Product Basic information

GroupID: AsyncProductGatewayService
ArtifactID: AsyncProductGatewayService
Version : Default

Update the pom.xml with the following information.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.ostech</groupId>
    <artifactId>AsyncProductGatewayService</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>mysql</groupId>
```

```
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.22</version>
        </dependency>
        <dependency>
            <groupId>io.reactivex.rxjava2</groupId>
            <artifactId>rxjava</artifactId>
            <version>2.2.6</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.google.guava/guava -->
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>30.1-jre</version>
        </dependency>

        <!-- https://mvnrepository.com/artifact/org.slf4j/log4j-over-slf4j -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>log4j-over-slf4j</artifactId>
            <version>2.0.0-alpha1</version>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
```

```
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Define the model classes, there is no any changes in it.

```
package com.ostech.product.model;

public abstract class BaseOption {

    public  Long id = null;
    public String description = "";
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getDescription() {
```

```
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }


}
```

```
package com.ostech.product.model;

import java.util.List;

public abstract class BaseProduct {

    public Long id;
    public String description;
    public List<BaseOption> baseOptions;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getDescription() {
        return description;
```

```
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public List<BaseOption> getBaseOptions() {
        return baseOptions;
    }
    public void setBaseOptions(List<BaseOption> baseOptions) {
        this.baseOptions = baseOptions;
    }


}
```

```
package com.ostech.product.model;

public class Option extends BaseOption {

    private Double price;
    private Integer inventory;
    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
        this.price = price;
    }
```

```
    public Integer getInventory() {
        return inventory;
    }
    public void setInventory(Integer inventory) {
        this.inventory = inventory;
    }


}
```

```
package com.ostech.product.model;

import java.util.List;

public class Product extends BaseProduct {

    private Integer totalInventory;
    private Double displayPrice;
    private List<Option> options;

    public Integer getTotalInventory() {
        return totalInventory;
    }

    public void setTotalInventory(Integer totalInventory) {
        this.totalInventory = totalInventory;
```

```
}

public Double getDisplayPrice() {
    return displayPrice;
}

public void setDisplayPrice(Double displayPrice) {
    this.displayPrice = displayPrice;
}

public List<Option> getOptions() {
    return options;
}

public void setOptions(List<Option> options) {
    this.options = options;
}

@Override
public String toString() {
    // TODO Auto-generated method stub

    return " Total Inventory : " + getTotalInventory() +
           "\n Display Price : " + getDisplayPrice() +
           "\n ID :" + getId() +
           "\n Product Info: " + getDescription();
}
```

```
}
```

We have defined 4 Value classes.

Now we will define the Data Access Layer. Here, you need to observe that, the method returns an Observable instead of a primitive type or normal object.

InventoryDAO.class

```
package com.ostech.data.access;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import io.reactivex.Observable;

public class InventoryDAO {

     public Observable<Integer> getInventory(Long productId, Long optionId) throws
SQLException {
          Integer inventory = 0;
          String sql = "select inventory from options where prodid=? and optionid =?";
```

```
        try
          {
              String url = "jdbc:mysql://localhost:3306/ostech";
             Connection conn = DriverManager.getConnection(url,"root","root213");

            // create the preparedstatement and add the criteria
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setLong(1, productId );
            ps.setLong(2, optionId );
            // process the results
            ResultSet rs = ps.executeQuery();
            while ( rs.next() )
            {
              inventory = rs.getInt("inventory");
            }
            rs.close();
            ps.close();
          }
          catch (SQLException se)
          {
            // log exception;
            throw se;
          }
        return Observable.just(inventory);

    }
}
```

It returns the Observable of type Integer.

PriceDAO.class

```
package com.ostech.data.access;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import io.reactivex.Observable;

public class PriceDAO {

    public Observable<Double> getPrice(Long productId, Long optionId) throws
SQLException {
        Double price = 0.0;
        String sql = "select price from options where prodid=? and optionid =?";

        try
          {
              String url = "jdbc:mysql://localhost:3306/ostech";
```

```
            Connection conn = DriverManager.getConnection(url,"root","root213");

            // create the preparedstatement and add the criteria
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setLong(1, productId );
            ps.setLong(2, optionId );
            // process the results
            ResultSet rs = ps.executeQuery();
            while ( rs.next() )
            {
              price = rs.getDouble("price");
            }
            rs.close();
            ps.close();
          }
          catch (SQLException se)
          {
            // log exception;
            throw se;
          }
        return Observable.just(price);

    }
}
```

Similarly, it returns observable of type Double.

In the case of the Product, we will keep it as a normal Object. We will convert into Observable from the Business Object layer.

```
package com.ostech.data.access;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import com.ostech.product.model.BaseOption;
import com.ostech.product.model.BaseProduct;
import com.ostech.product.model.Option;
import com.ostech.product.model.Product;

public class ProductDAO {

    public BaseProduct getBaseProduct(Long productId) throws SQLException {
        Product prod = new Product();
        String sql ="select a.description ad,b.prodid,b.optionid,b.description bd
from product a, options b where a.id = b.prodid and a.id =?";
        List<Option> optionList = new ArrayList<Option>();
        List<BaseOption> boptionList = new ArrayList<BaseOption>();
        Option bo = null;
```

```
try
  {
     String url = "jdbc:mysql://localhost:3306/ostech";
     Connection conn = DriverManager.getConnection(url,"root","root213");
    // create the preparedstatement and add the criteria
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setLong(1, productId );

    // process the results
    ResultSet rs = ps.executeQuery();
    boolean flag = true;
    while ( rs.next() )
    {
    if(flag) {
         prod.id = rs.getLong("prodid");
         prod.description = rs.getString("ad");
         flag = false;
    }
     bo = new Option();
         bo.setDescription(rs.getString("bd"));
         bo.setId(rs.getLong("optionid"));
         optionList.add(bo);
         boptionList.add(bo);
    }
    rs.close();
    ps.close();
  }
```

```
            catch (SQLException se)
            {
              // log exception;
              throw se;
            } finally {
                prod.setOptions(optionList);
                prod.setBaseOptions(boptionList);
            }
          return prod;

    }

}
```

Up till now, we have defined the Data Access Layer.

Next is the Business Layer.

Let us redefine the ProductService.class , instead of a POJO, it will convert the BaseProduct Value Object to an Observable.

```
package com.ostech.product.services;

import java.sql.SQLException;
```

```
import com.ostech.data.access.ProductDAO;
import com.ostech.product.model.BaseProduct;

import io.reactivex.Observable;
import io.reactivex.Scheduler;
import io.reactivex.schedulers.Schedulers;

public class ProductService {
    ProductDAO prodDAO = new ProductDAO();
    Observable<BaseProduct> getBaseProduct(Long productId) throws SQLException {
        //subscribeOn(Schedulers.io()
        return Observable.just(prodDAO.getBaseProduct(productId));
    }
}
```

It uses the ***just*** method to convert into an Observable.

Next let us rewrite the `ProductGateway.class`

Import the following.

```
package com.ostech.product.services;

import java.sql.SQLException;
```

```
import com.ostech.data.access.InventoryDAO;
import com.ostech.data.access.PriceDAO;
import com.ostech.product.model.BaseOption;
import com.ostech.product.model.BaseProduct;
import com.ostech.product.model.Option;
import com.ostech.product.model.Product;

import io.reactivex.Observable;
```

Instantiate the following

```
ProductService baseProductClient = new ProductService();
    PriceDAO priceClient = new PriceDAO();
    InventoryDAO inventoryClient = new InventoryDAO();
```

Add the following method that perform the actual Product object assembly.

```
    public Observable<Product> getProduct(Long productId) throws SQLException {
        // Get all the Base Product
        Observable<BaseProduct> bp = baseProductClient.getBaseProduct(productId);
// Blocking WS Call
        return bp.flatMap( basep -> {
```

```
            // Get the options for Each Product.
            Observable<Option> options =
Observable.fromArray(basep.getBaseOptions().toArray(new BaseOption[0]))
                .flatMap( option -> {
                    /*
                     * Get the Price and Inventory of each option.
                     */
                    Observable<Double> price =
priceClient.getPrice(basep.id,option.getId()); // Blocking WS Call
                    Observable<Integer> inventory =
inventoryClient.getInventory(basep.getId(),option.getId()); // Blocking WS Call
                   /*
                    * Join two Observable and return a single Observable.
                    */
                    return Observable.zip(price , inventory,
                            (p , i) -> {
                           Option toption = new Option();
                               toption.setDescription(option.getDescription());
                               toption.setId(option.getId());
                               toption.setInventory(i);
                               toption.setPrice(p);
                               return toption;
                            });
                });


            /*
```

```
            * Create the final Product populated with the Price and Inventory.
            */
            int totalInventory = 0;
            Double totalPrice = new Double(0.0);

           // Calculate the inventory and price.
            for (Option o : options.blockingIterable()) {
                totalInventory += o.getInventory();
                totalPrice = Double.sum(totalPrice, o.getPrice());
            }


              Product p = new Product();
                    p.setId(new Long(bp.blockingFirst().getId()));
                    p.setDescription(bp.blockingFirst().getDescription());
                    p.setDisplayPrice(totalPrice);
                    p.setOptions(options.toList().blockingGet());
                    p.setTotalInventory(totalInventory);
             return Observable.just(p);
        });
    }
```

You can observe the following in the above code:

1. Product information is retrieve as an Observable.

```
Observable<BaseProduct> bp = baseProductClient.getBaseProduct(productId);
```

2. Use flatMap to transform the BaseProduct's option item

```
bp.flatMap
```

3. Fetch the Price and Inventory as an observable.

```
Observable<Double> price = priceClient.getPrice(basep.id,option.getId()); //
Blocking WS Call
                    Observable<Integer> inventory =
inventoryClient.getInventory(basep.getId(),option.getId()); // Blocking WS Call
                   /*
```

4. Merge the above two observable using zip operator.

```
return Observable.zip(price , inventory,
                             (p , i) -> {
                            Option toption = new Option();
                                toption.setDescription(option.getDescription());
                                toption.setId(option.getId());
                                toption.setInventory(i);
                                toption.setPrice(p);
                                return toption;
                                     });
```

5. Finally an observable product is return after all the transformation.

```
 return Observable.just(p);
```

Let us create a TestClient to invoke the GatewayService.class

```
package com.ostech.client;

import java.sql.SQLException;

import com.ostech.data.access.PriceDAO;
import com.ostech.product.model.Product;
import com.ostech.product.services.ProductGateway;

import io.reactivex.Observable;
import io.reactivex.Scheduler;
import io.reactivex.schedulers.Schedulers;

public class TestClient {

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

        getProduct();

        //Thread.sleep(1500);
    }


    public static void getProduct() {
        ProductGateway gateway = new ProductGateway();
        try {
```

```
            Observable<Product> prod =    gateway.getProduct(new Long(2));
               prod
                .subscribeOn(Schedulers.io())
                .subscribe(p -> {
                 System.out.println(p.toString());

               });

            } catch (SQLException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

        }

}
```

Execute the above.

```
13 public class TestClient {
14
15     public static void main(String[] args) throws Exception {
16
17         getProduct();
18
19         //Thread.sleep(1500);
20     }
21
22
23     public static void getProduct() {
24         ProductGateway gateway = new ProductGateway();
25         try {
26
27         Observable<Product> prod =  gateway.getProduct(new Long(2));
28            prod
29             .subscribeOn(Schedulers.io())
```

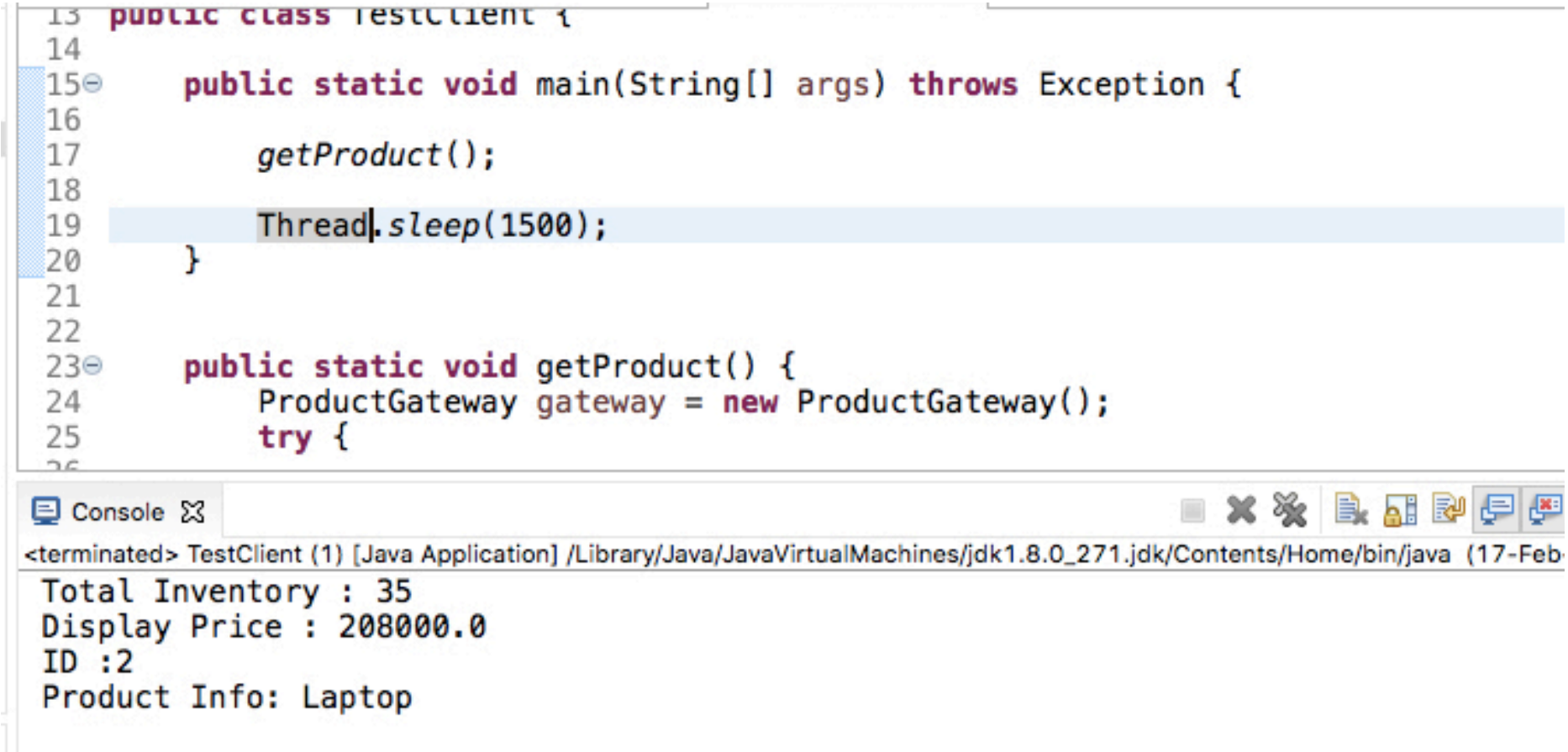
Console

```
<terminated> TestClient (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java  (17-Feb-2021, 4:24:3
```

You wont observe anything as shown above. Surprise?? (Hints-> Async)

Uncomment the following comment in the main method and execute again.

// Thread.sleep(1500)



This time you get the result. We will discuss about Async and scheduler later.

---------- Lab Ends Here ----------

9. Handling Errors – 60 Minutes

In this lab, various ways of handling errors in RxJava will be demonstrated.

Create a class, `MyErrors` in package: `com.tos.errors.`

Import the following packages.

```
import java.io.IOException;
import java.util.Random;

import com.tos.gateway.TweetGateway;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
```

Add the following method.

```
/*
 * Subscriber having – Exception Handling and Completion Event.
 *
 */
public static void understandObservablesWithExceptionCompletionSubscriber() {
     TweetGateway tc = new TweetGateway();
     Observable<Tweet> tweets = Observable.fromArray(tc.getTweets().toArray(new Tweet[0]));
     tweets.subscribe(t -> {
          System.out.println(t);
     }, e -> {
          e.printStackTrace();
```

```
        }, () -> {
             System.out.println("Completed Processing");
        });
        /*
         * Java 8 method references. /* tweets.subscribe( System.out::println,
         * Throwable::printStackTrace, LearningObservable::noMore);
         */
    }


    public static void noMore() {
        System.out.println("No More Record");
    }
```

It has 3 functions,
OnNext , OnError and OnCompletion.

Next method propagates Error using Try Catch Block.

```
    /*
     * Propagate Error using Try Catch Block.
     *
     */
    public static void errorPopogation() {
        TweetGateway tc = new TweetGateway();
        Observable<Tweet> tweets = Observable.<Tweet>create(subscriber -> {
            try {
                boolean flag = true;
                subscriber.onNext(new Tweet("1", "Just My Tweet"));
                if(flag)
```

```
                    throw new RuntimeException(" Random Error ");
                subscriber.onComplete();
            } catch (Exception e) {
                // TODO Auto-generated catch block
                subscriber.onError(e);
            }
        });

        tweets.subscribe(t -> {
            System.out.println(" 1: " + t);

        });
        tweets.subscribe(t -> {
            System.out.println(" 2: " + t);

        });

    }
```

Subscriber without error Handler.

```
/*
     *  Without Error handler
     */
    public static void withoutErrorCallBack() {

        Observable.create(subscriber -> {
            try {
                subscriber.onNext(1 / 0);
            } catch (Exception e) {
                subscriber.onError(e);
```

```
            }
        })
        //BROKEN, missing onError() callback
        .subscribe(System.out::println);

    }
```

With Error Callback().

```
/*
     *  With Error Callback
     */
    public static void withErrorCallBack() {

        Observable.create(subscriber -> {
            try {
                subscriber.onNext(1 / 0);
            } catch (Exception e) {
                subscriber.onError(e);
            }
        })
        //BROKEN, missing onError() callback
        //.subscribe(System.out::println,throwable -> log.error("That escalated quickly",
throwable));
        .subscribe(System.out::println,System.out::println);

        // Try below syntax.. All are same
        //Observable.create(subscriber -> subscriber.onNext(1 /
0)).subscribe(System.out::println,System.out::println);
        //Observable.fromCallable(() -> 1 /
```

```
0).subscribe(System.out::println,System.out::println);

      }
```

Errors handling when Exception occurs during transformation(operator).

```
/*
      * Error while performing Operator.
      */
     public static void errorsInOperator() {

          Observable.just(1,2 ,0).doOnNext(System.out::println).map(x -> 10 /
x).subscribe(System.out::println,System.out::println);
          Observable.just("Lorem", "ipsum",null).doOnNext(System.out::println).map(x -> x)

.doOnNext(System.out::println).filter(String::isEmpty).subscribe(System.out::println,System.out:
:println);

     }
```

Stopping Execution with error

```
/*
      *  Stopping execution with Error.
      */
    public static void doOnError() {
          Observable<Tweet> tweets = TweetGateway.getObservableTweets();
          tweets.filter(t -> t.getTweetID() =="2").flatMap( t -> {
     return Observable.error(new IOException("Something went wrong with Tweet ID : " +
t.getTweetID()))
                    .doOnError(error -> System.err.println("The error message is: " +
error.getMessage()) )
                    ;
          }).subscribe(
            x -> System.out.println("onNext should never be printed!"),
            Throwable::printStackTrace,
            () -> System.out.println("onComplete should never be printed!"));

     }
```

Proceed execution after Error. When tweet ID 2 give error it will be skipped and proceed.

```
/*
     *  When tweet 2 give error it will be skipped and proceed.
     */

    public static void onErrorResumeNext() {
          Observable<Tweet> tweets = TweetGateway.getObservableTweets();
          tweets.flatMap( t -> {
          if (t.getTweetID() =="2" )    {
     return Observable.error(new IOException("Something went wrong with Tweet ID : " +
```

```
t.getTweetID()))
                    .doOnError(error -> System.err.println("The error message is: " +
error.getMessage()))
                    .onErrorResumeNext(Observable.empty());
          }else
                return Observable.just(t);
          }).subscribe(
            x -> System.out.println(" Get Tweet : " + x),
            Throwable::printStackTrace,
            () -> System.out.println("Completed!"));

     }
```

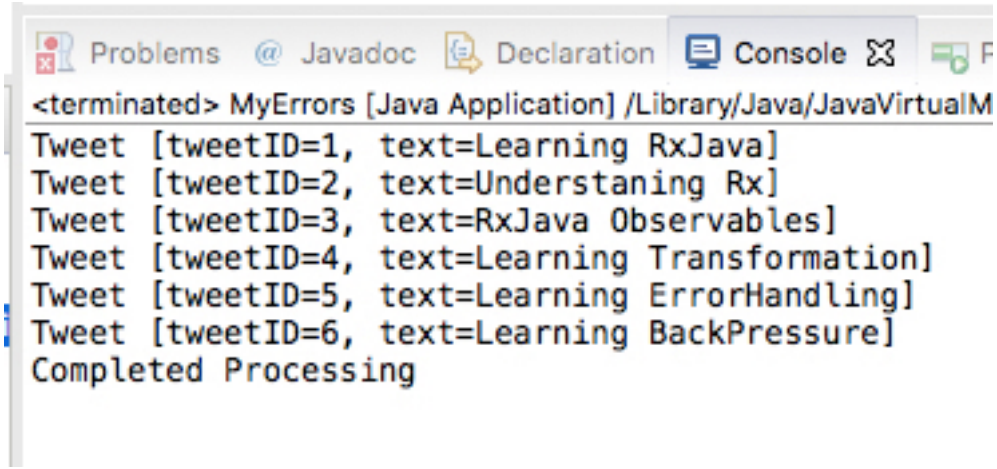
Retry Errors.

```
/*
     *  When  random error is generated  it will be retried.
     */

    public static void onErrorResumeNextWithRetry() {
          Observable<Tweet> tweets = TweetGateway.getObservableTweets();
          tweets.flatMap( t -> {
          if ((Math.random() < 0.1) )   {
     return Observable.error(new IOException("Something went wrong with Tweet ID : " +
t.getTweetID()))
                    .doOnError(error -> System.err.println("The error message is: " +
error.getMessage()));
       }else
                return Observable.just(t);
          }).retry((retrycount, error) -> retrycount < 5 ).distinct().subscribe(
            x -> System.out.println(" Get Tweet : " + x),
```

```
                Throwable::printStackTrace,
                () -> System.out.println("Completed!"));

        // Try without distinct --> It will retry from the begining of the List.
    }
```

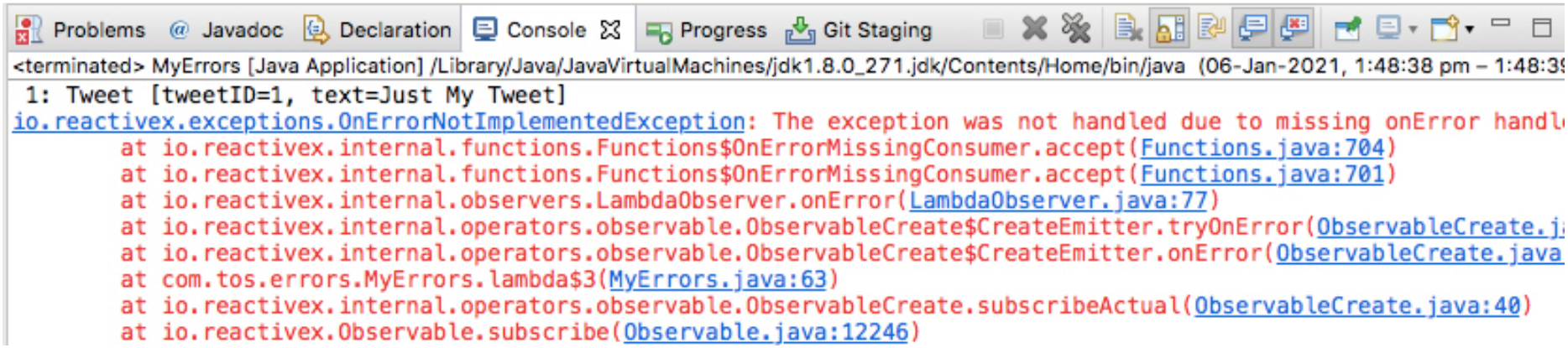
Let us execute the method one at a time and review the output.

```
understandObservablesWithExceptionCompletionSubscriber();
```



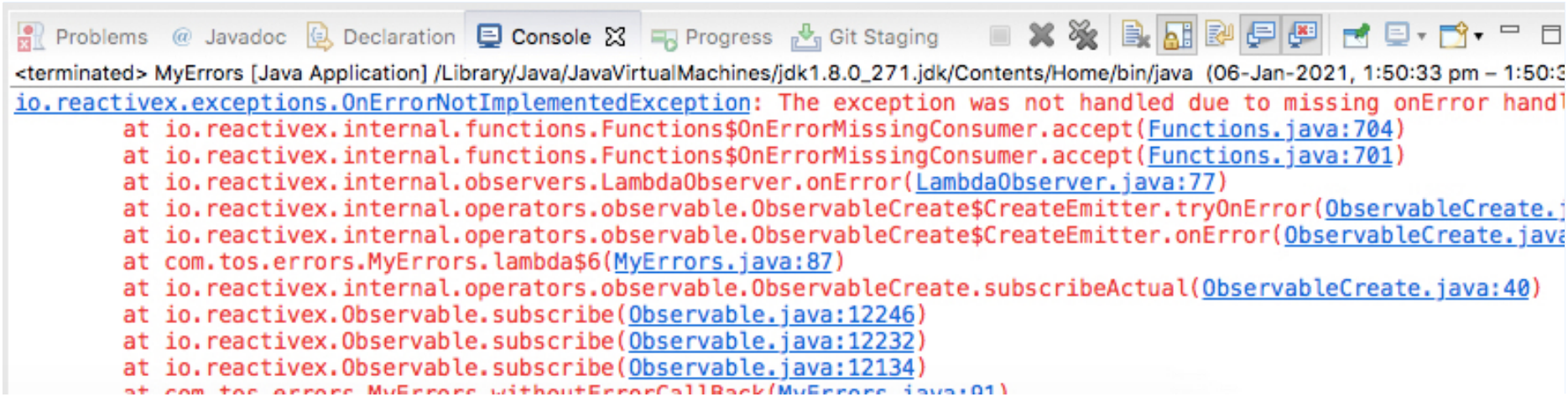
OnNext and OnCompletion being invoked.

```
errorPopogation();
```



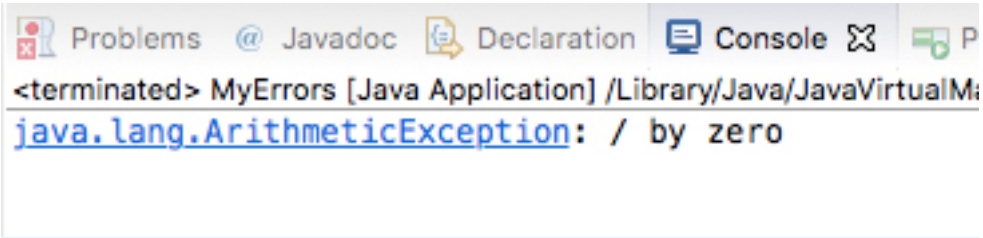
Error propagated but no handler found in the subscriber.

```
withoutErrorCallBack();
```



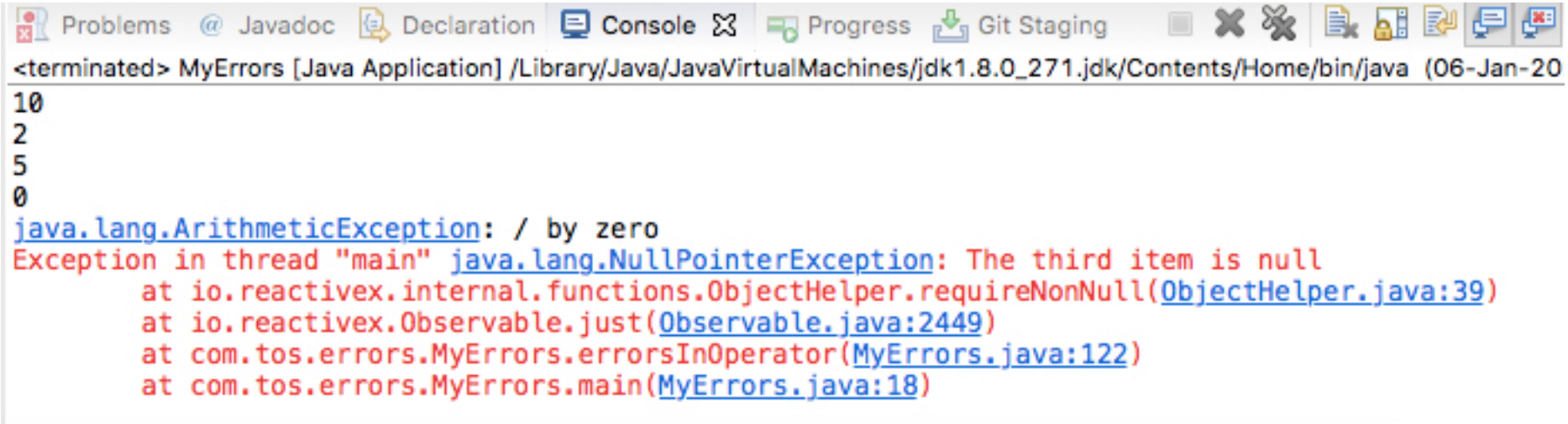
Another way of Error propagation without handler.

```
withErrorCallBack();
```



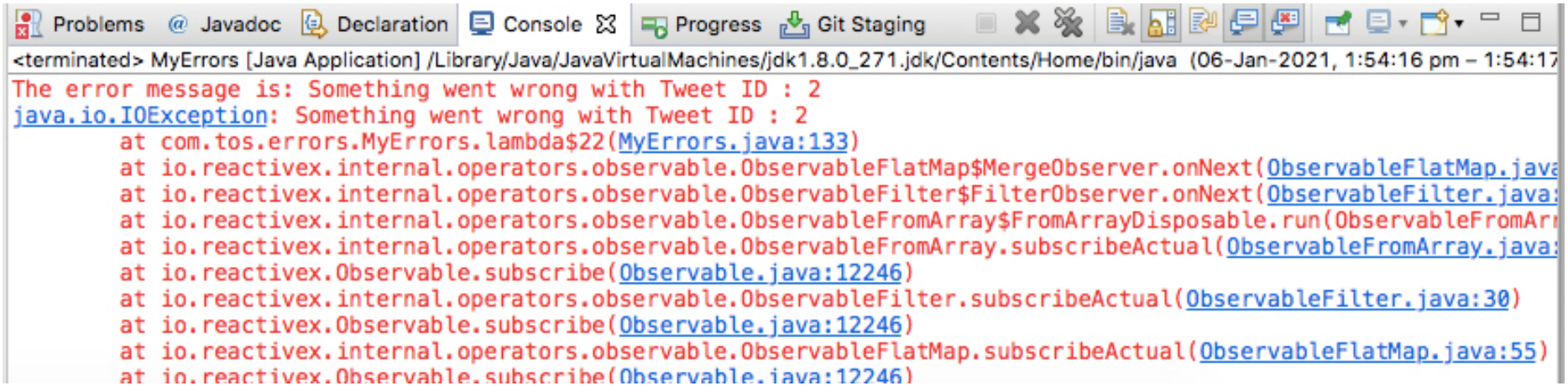
Error being consumed by the Error handler of subscriber.

```
errorsInOperator();
```



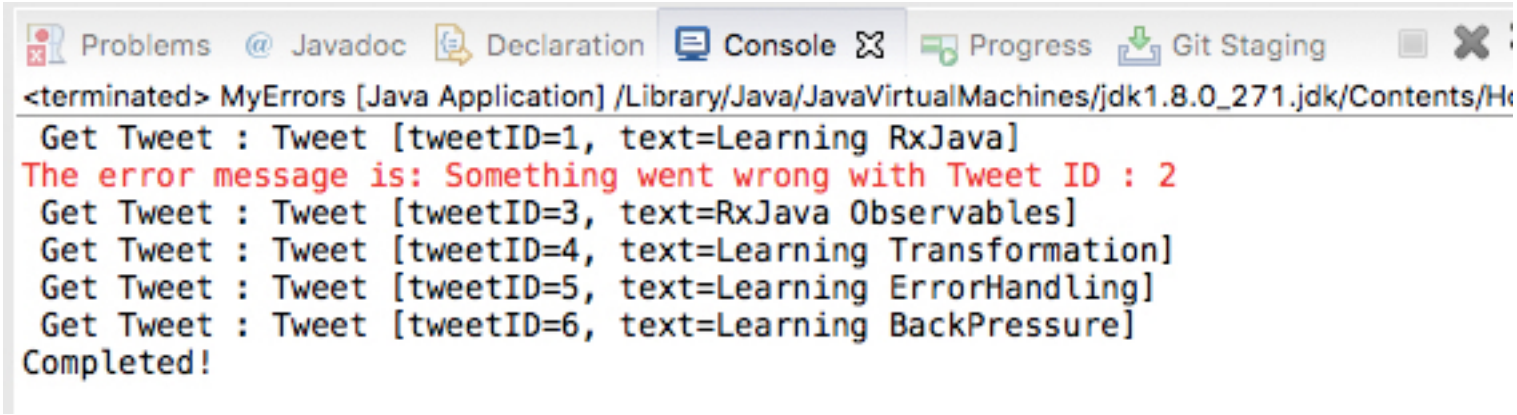
Exception while transformation.

```
doOnError();
```



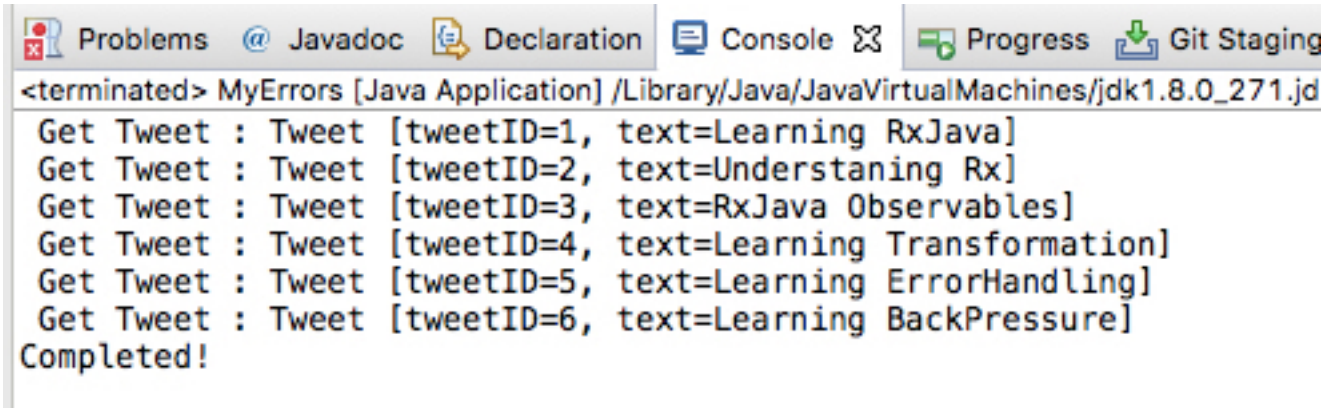
Stop Execution when error encountered.

```
onErrorResumeNext();
```



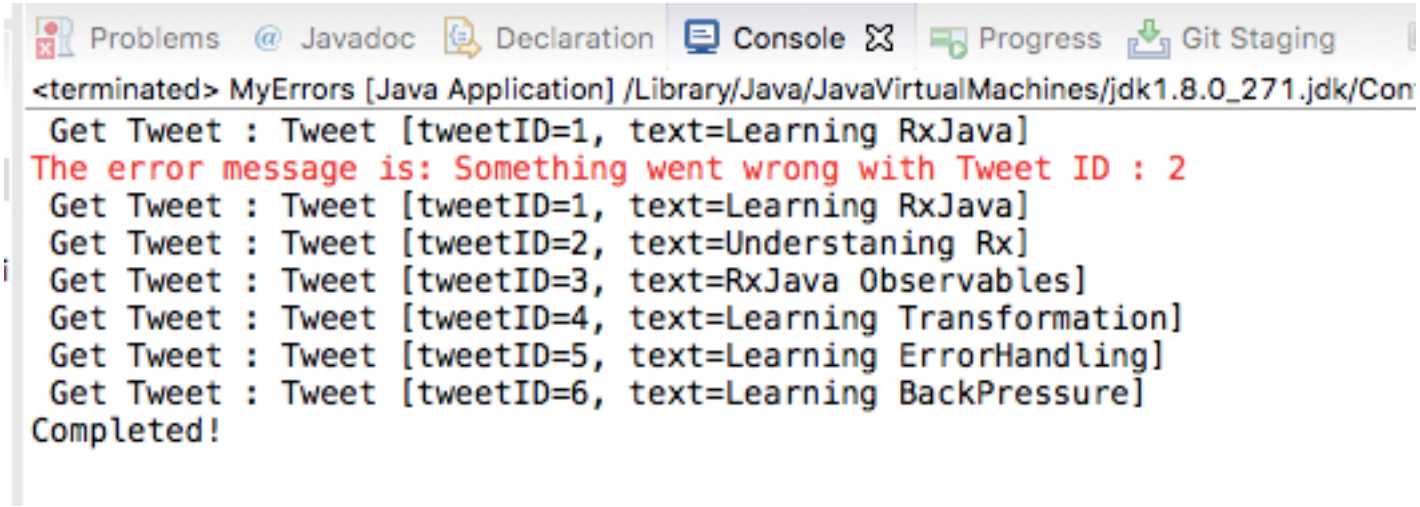
Skipped Tweet ID 2.

```
onErrorResumeNextWithRetry();
```



It will be retried 5 times, when there is any errors.

Try without distinct --> It will retry from the beginning of the List.



Task to be Done.

Implements Exception handling in the TweetRx Application:

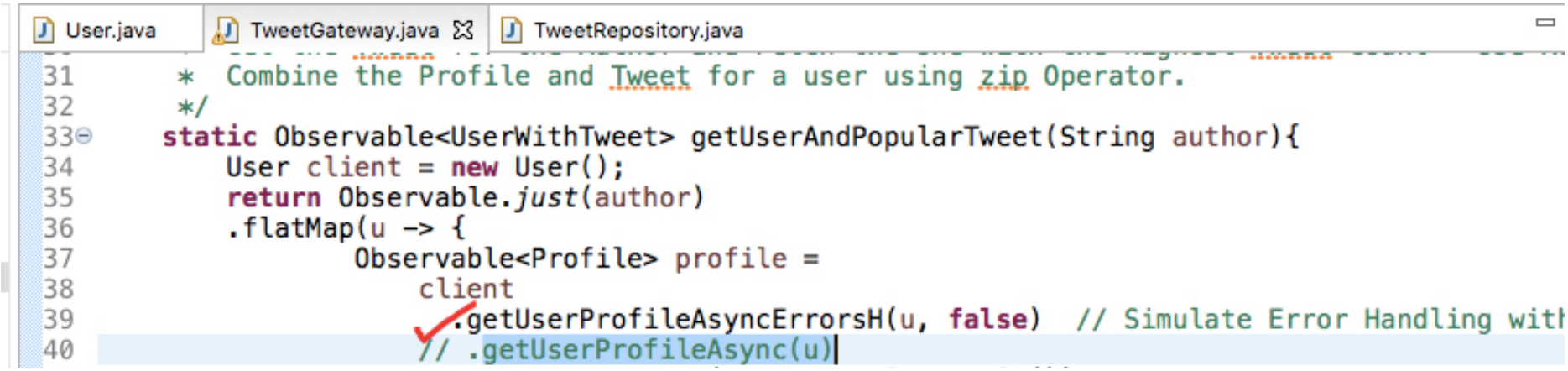
Scenario:

There is an Error while fetching the Profile. Assign a default Profile to it and the application should proceed as it is.

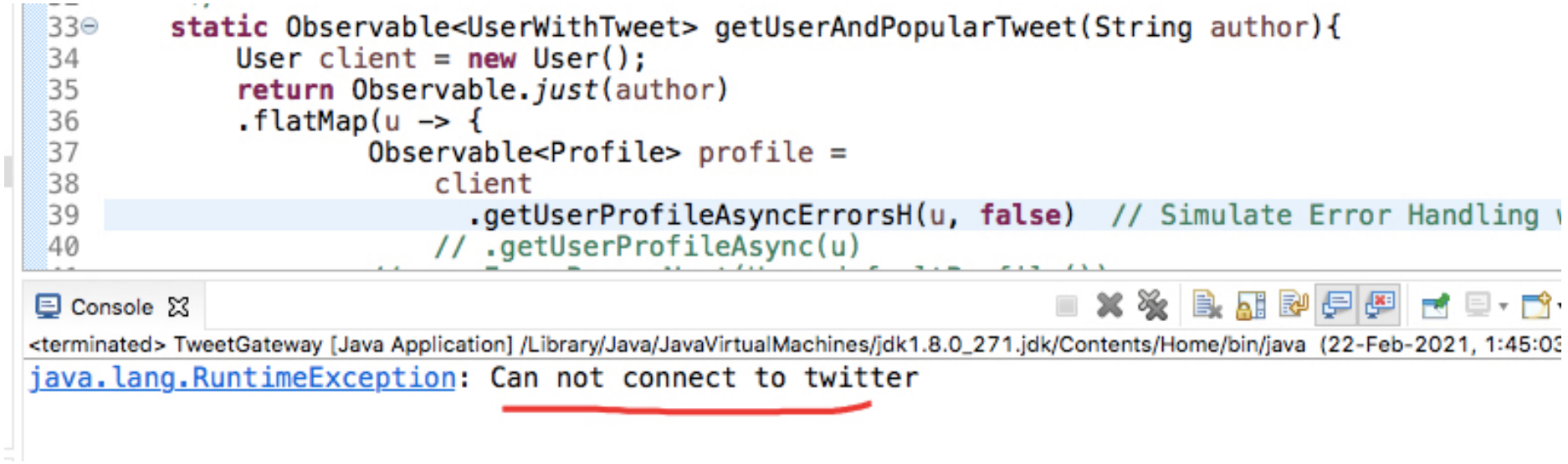
Hints:

In the TweetGateway Class, the method `getUserAndPopularTweet should`

use `getUserProfileAsyncErrorsH(u, false)` to simulate Error instead of `getUserProfileAsync(u)` to fetch the Profile details.



Now execute the program.

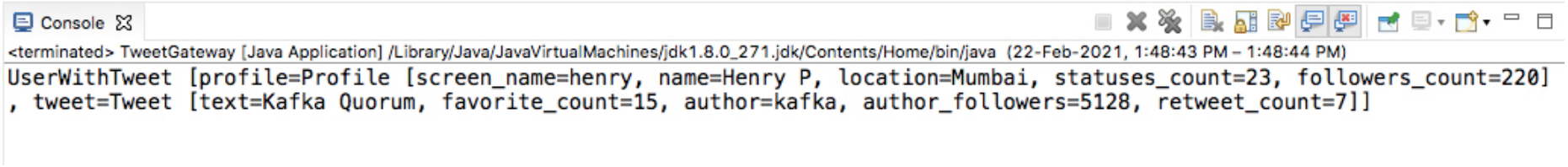


Now the application will stop as shown below.

You should define a default profile whenever such error comes and allows to proceed the application.

Implements it using `onErrorResumeNext(defaultProfile())`

When you execute the Program after implementing the above, it should be as shown below.



Solution:
Define a default profile.

```
95
96      /*
97       *  On Error, return this Profile User.
98       */
99     public   static Observable<Profile> defaultProfile(){
100         Profile myProfile = new Profile("henry", "Henry P", "Mumbai", 23, 220);
101         return Observable.just(myProfile);
102
103     }
```

Invoke the profile using OnErrorResume Next.

```
 *  Combine the Profile and Tweet for a user using zip Operator.
 */
static Observable<UserWithTweet> getUserAndPopularTweet(String author){
    User client = new User();
    return Observable.just(author)
    .flatMap(u -> {
            Observable<Profile> profile =
                client
                  .getUserProfileAsyncErrorsH(u, false)  // Simulate Error Handling with this Method
                // .getUserProfileAsync(u)
                .onErrorResumeNext(User.defaultProfile())
                .subscribeOn(Schedulers.io()); // Using Seperate IO Thread
```

----------Lab Ends Here ----------

10. Schedulers – 60 Minutes

Create a package to store all the related classes for this lab.

`com.tos.scheduler`

Let us create a Class with main method - MyScheduler

Import the following classes and packages.

```
import static java.util.concurrent.Executors.newFixedThreadPool;

import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ThreadFactory;

import com.google.common.util.concurrent.ThreadFactoryBuilder;
import com.tos.gateway.TweetGateway;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
import io.reactivex.ObservableEmitter;
import io.reactivex.ObservableOnSubscribe;
import io.reactivex.Scheduler;
import io.reactivex.schedulers.Schedulers;
```

Add the following two methods, which will be used for logging the execution steps and hold the thread for a particular period.

```
static long start = System.currentTimeMillis();

/*
      * A logging method
      */
     static void log(Object label) {
          System.out.println(
                    System.currentTimeMillis() - start + "\t| " +
Thread.currentThread().getName() + "\t| " + label);
     }

     static void sleepOneSecond() {
          try {
               Thread.sleep(10000);
          } catch (InterruptedException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
          }
     }
```

Add the following method, it set the thread factory Method with the display format.

```
     /*
      * Executor factory method for verbose print with Thread - Custom Name
      */

     private static ThreadFactory threadFactory(String pattern) {
          return new ThreadFactoryBuilder().setNameFormat(pattern).build();
     }
```

Add this method to create different type of ThreaPool depending on the Flag.

```
static Scheduler myScheduler(String flag) {
          Scheduler scheduler = null;
          switch (flag) {
          case "A":
               ExecutorService poolA = newFixedThreadPool(10, threadFactory("Sched-A-%d"));
               scheduler = Schedulers.from(poolA);
               break;
          case "B":
               ExecutorService poolB = newFixedThreadPool(10, threadFactory("Sched-B-%d"));
               scheduler = Schedulers.from(poolB);
               break;
          default:
               ExecutorService poolC = newFixedThreadPool(10, threadFactory("Sched-C-%d"));
               scheduler = Schedulers.from(poolC);
               break;
          }

          return scheduler;

     }
```

Understand the Trampoline Scheduling with the following method.

```
/*
      * Understanding Trampoline .
      */
     static void trampoline() {
          // Scheduler scheduler = Schedulers.from(executor).;
          Scheduler scheduler = Schedulers.trampoline();
          Scheduler.Worker worker = scheduler.createWorker();

          log("Main start");
          worker.schedule(() -> {
               log(" Outer start");
               sleepOneSecond();
               worker.schedule(() -> {
                    log(" Middle start");
                    sleepOneSecond();
                    worker.schedule(() -> {
                         log(" Inner start");
                         sleepOneSecond();
                         log(" Inner end");
                    });
                    log(" Middle end");
               });
               log(" Outer end");
          });
          log("Main end");
     }
```

Execute the method().

MyScheduler.java

```
83          Scheduler.Worker worker = scheduler.createWorker();
84
85          log("Main start");
86          worker.schedule(() -> {
87              log(" Outer start");
88              sleepOneSecond();
89              worker.schedule(() -> {
90                  log(" Middle start");
91                  sleepOneSecond();
92                  worker.schedule(() -> {
93                      log(" Inner start");
94                      sleepOneSecond();
95                      log(" Inner end");
96                  });
97                  log(" Middle end");
98              });
99              log(" Outer end");
100         });
101         log("Main end");
102     }
103
```

Console

<terminated> MyScheduler [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Hom

```
62      | main | Main start
160     | main | Outer start
10167   | main | Outer end
10167   | main | Middle start
20175   | main | Middle end
20175   | main | Inner start
30179   | main | Inner end
30183   | main | Main end
```

Complete Each Job before scheduling another One.

Same thread

As you can observe- Everything schedule in main thread and Outer Job gets completed before scheduling the inner job.

Add the following method to demonstrate the trampoline in Observable Object.

```
/*
     * All jobs that subscribes on trampoline() will be queued and executed one by one.
     * we have more than one observable and we want them to execute in order
     * Use Case: Complete execution of the first Observable and then the Second Observable.
     */
    static void observablesUsingTrampoline() {
        Observable.just(2, 4, 6, 8, 10)
          .subscribeOn(Schedulers.io())
          //.subscribeOn(Schedulers.trampoline())
          .subscribe( obj -> log("Processing DB Update for Order ID :" + obj));
        Observable.just(1, 3, 5, 7, 9)
         .subscribeOn(Schedulers.io())
        // .subscribeOn(Schedulers.trampoline())
        .subscribe(obj -> log(obj));
       try {
            Thread.sleep(100000);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
```

First Execute as it is using io scheduler.

MyScheduler.java

```
109      */
110     static void observablesUsingTrampoline() {
111         Observable.just(2, 4, 6, 8, 10)
112           .subscribeOn(Schedulers.io())
113           //.subscribeOn(Schedulers.trampoline())
114           .subscribe( obj -> log("Processing DB Update for Order ID :" + obj));
115         Observable.just(1, 3, 5, 7, 9)
116          .subscribeOn(Schedulers.io())
117         // .subscribeOn(Schedulers.trampoline())
118         .subscribe(obj -> log(obj));
119         try {
120             Thread.sleep(100000);
121         } catch (InterruptedException e) {
122             // TODO Auto-generated catch block
123             e.printStackTrace();
124         }
125     }
```

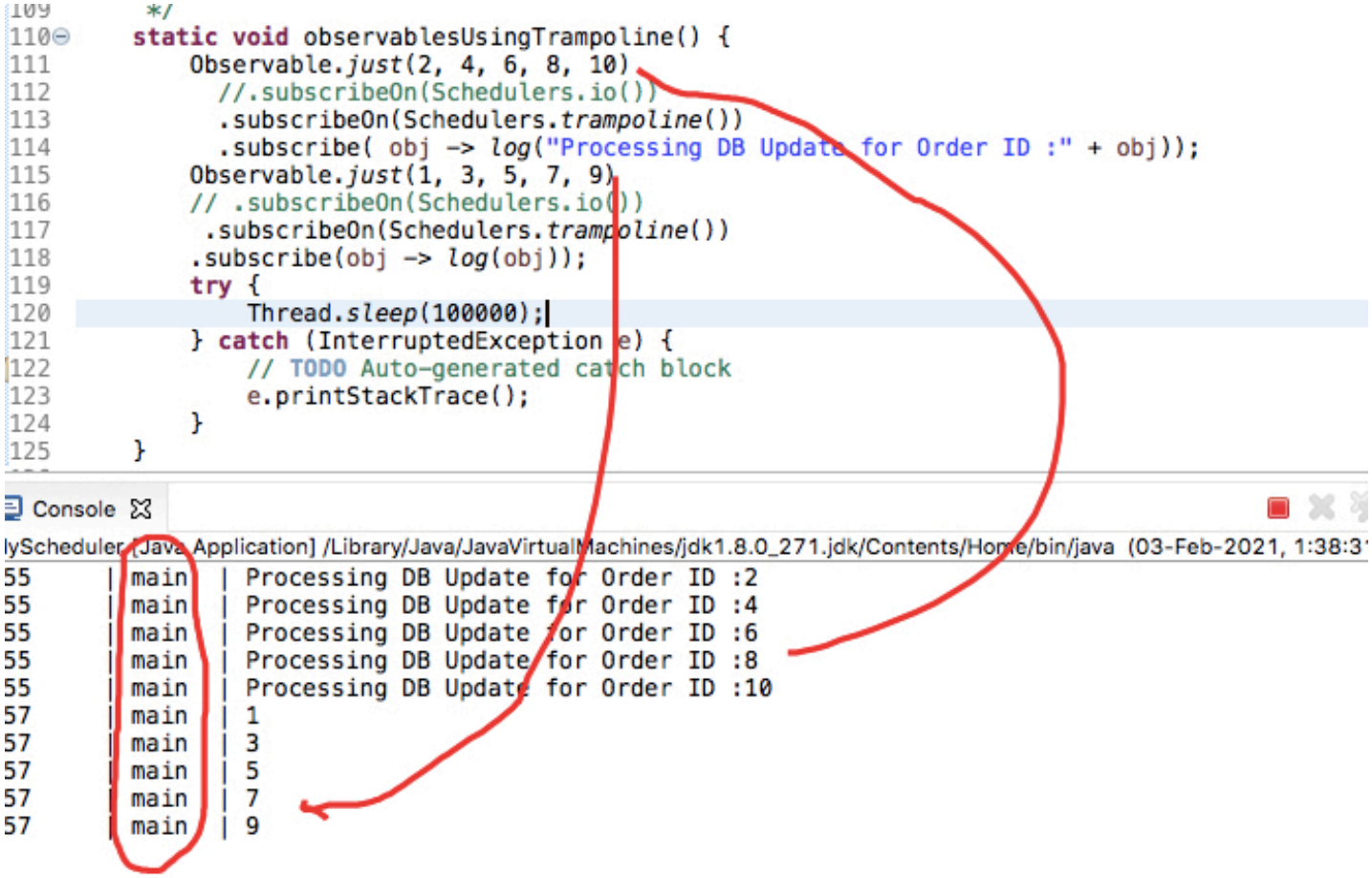
Console

```
<terminated> MyScheduler [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java
267     | RxCachedThreadScheduler-1     | Processing DB Update for Order ID :2
267     | RxCachedThreadScheduler-2     | 1
267     | RxCachedThreadScheduler-1     | Processing DB Update for Order ID :4
267     | RxCachedThreadScheduler-2     | 3
267     | RxCachedThreadScheduler-1     | Processing DB Update for Order ID :6
267     | RxCachedThreadScheduler-2     | 5
267     | RxCachedThreadScheduler-1     | Processing DB Update for Order ID :8
267     | RxCachedThreadScheduler-2     | 7
267     | RxCachedThreadScheduler-1     | Processing DB Update for Order ID :10
267     | RxCachedThreadScheduler-2     | 9
```

As you can observer there is an overlap in the processing of the Two Observables.

Let us comment the io scheduler are replace with that of the trampoline.

Execute again:



As you can observe, the whole program executes in main thread and First observables objects get processed before the processing of the Next Observable.

Add this method that will create an Observable Objects of type Tweet.

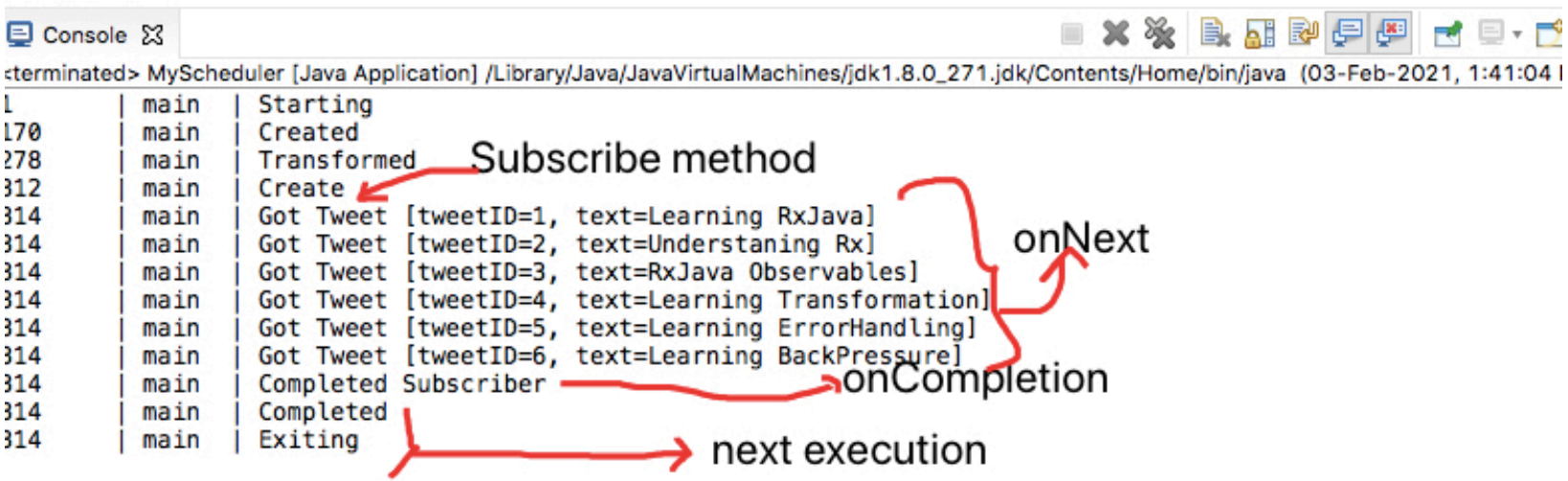
```
/*
      * Creating Observable Tweet Objects.
      */
     static Observable<Tweet> simple() {
          TweetGateway tc = new TweetGateway();
          Observable<Tweet> tweets = Observable.create(new ObservableOnSubscribe<Tweet>() {
               @Override
               public void subscribe(ObservableEmitter<Tweet> subscriber) throws Exception {
                    List<Tweet> tweets = tc.getTweets();
                    log("Create");
                    for (Tweet tweet : tweets) {
                         subscriber.onNext(tweet);
                    }
                    subscriber.onComplete();
                    log("Completed");
               }
          });

          return tweets;

     }
```

The Next method display the Normal Thread Mechanism in Observable. It is Single Thread Model Using Client. All events are sequential and use Main Thread Only.

```
/*
      * Normal Thread Mechanism in Observable. - Single Thread Using Client. ALl
      * events are sequential and use Main Thread Only.
      */
     static void understandingNormalThread() {

          log("Starting");
          final Observable<Tweet> obs = simple();
          log("Created");
          final Observable<Tweet> obs2 = obs.map(x -> x).filter(x -> true);
          log("Transformed");
          obs2.subscribe(x -> log("Got " + x), Throwable::printStackTrace, () -> log("Completed
Subscriber"));
          log("Exiting");
     }
```

Execute the method and review the execution flow.

```java
/*
 * Normal Thread Mechanism in Observable. - Single Thread Using Client. ALl
 * events are sequential and use Main Thread Only.
 */
static void understandingNormalThread() {

    log("Starting");
    final Observable<Tweet> obs = simple();
    log("Created");
    final Observable<Tweet> obs2 = obs.map(x -> x).filter(x -> true);
    log("Transformed");
    obs2.subscribe(x -> log("Got " + x), Throwable::printStackTrace, () -> log("Completed Su
    log("Exiting");
}
```



Try to understand the flow execution: subscriber Method , Multiple onNext → OnComplete.

You can use Scheduler for invoking Subscriber in different threads then that of the caller thread as demonstrate below.

```
/*
      * Using Scheduler for invoking Subscriber in different thread
      *
      */
     static void understandingScheduler() {

          log("Starting");
          final Observable<Tweet> obs = simple();
          log("Created");
          final Observable<Tweet> obs2 = obs.map(x -> x).filter(x -> true);
          log("Transformed");
          obs2.subscribeOn(myScheduler("A")) // Using A Scheduler.
                    .subscribe(x -> log("Got " + x), Throwable::printStackTrace, () ->
log("Completed"));
          log("Exiting");
     }
```

Execute the method.

```
169      */
170     static void understandingScheduler() {
171
172         log("Starting");
173         final Observable<Tweet> obs = simple();
174         log("Created");
175         final Observable<Tweet> obs2 = obs.map(x -> x).filter(x -> true);
176         log("Transformed");
177         obs2.subscribeOn(myScheduler("A")) // Using A Scheduler.
178                 .subscribe(x -> log("Got " + x), Throwable::printStackTrace, () -> log("Completed"))
179         log("Exiting");
180     }
181
```

Console

MyScheduler [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (03-Feb-2021, 2:33:07 PM)

```
0       | main      | Starting
156     | main      | Created
273     | main      | Transformed
335     | main      | Exiting
339     | Sched-A-0       | Create
339     | Sched-A-0       | Got Tweet [tweetID=1, text=Learning RxJava]
339     | Sched-A-0       | Got Tweet [tweetID=2, text=Understaning Rx]
339     | Sched-A-0       | Got Tweet [tweetID=3, text=RxJava Observables]
339     | Sched-A-0       | Got Tweet [tweetID=4, text=Learning Transformation]
339     | Sched-A-0       | Got Tweet [tweetID=5, text=Learning ErrorHandling]
339     | Sched-A-0       | Got Tweet [tweetID=6, text=Learning BackPressure]
339     | Sched-A-0       | Completed
340     | Sched-A-0       | Completed
```

It got executed using the A scheduler thread Pool.

Next, Using different threads in various Object Transformation Sequences. Various events travel sequentially using the scheduler's thread to finally reach the Subscriber. If we want to Invoke each

Tweet Using Different Threads? Then Check Next Method Using FlatMap.

```
/*
     * Using Object Transformation Sequences. events travel sequentially through the
     * scheduler's thread to finally reach the Subscriber. If we want to Invoke each
     * Tweet Using Different Thread? Check Next Method Using FlatMap
     *
     */
    static void understandingSchedulerExecutionTransformation() {

        log("Starting");
        final Observable<Tweet> obs = simple();
        log("Created");
        final Observable<Tweet> obs2 = obs.doOnNext(MyScheduler::log).map(x -
> {
            x.setText(x.getText() + "1");
            return x;
        }).doOnNext(MyScheduler::log).map(x -> {
            x.setText(x.getText() + "2");
            return x;
        }).doOnNext(MyScheduler::log);
        log("Transformed");
        obs2.subscribeOn(myScheduler("A")).subscribe(x -> log("Got " + x),
Throwable::printStackTrace, () -> log("Completed"));
        log("Exiting");
```

```
}
```

Execute the method.

MyScheduler.java

```
18/      */
188      static void understandingSchedulerExecutionTransformation() {
189
190          log("Starting");
191          final Observable<Tweet> obs = simple();
192          log("Created");
193          final Observable<Tweet> obs2 = obs.doOnNext(MyScheduler::log).map(x -> {
194              x.setText(x.getText() + "1");
195              return x;
196          }).doOnNext(MyScheduler::log).map(x -> {
197              x.setText(x.getText() + "2");
198              return x;
199          }).doOnNext(MyScheduler::log);
200          log("Transformed");
201          obs2.subscribeOn(myScheduler("A")).subscribe(x -> log("Got " + x), Throwable::p
202          log("Exiting");
```

Console

```
MyScheduler [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java  (03-Feb-2021, 2:
115      | main   | Created
220      | main   | Transformed
268      | main   | Exiting
272      | Sched-A-0      | Create
272      | Sched-A-0      | Tweet [tweetID=1, text=Learning RxJava]
272      | Sched-A-0      | Tweet [tweetID=1, text=Learning RxJava1]
272      | Sched-A-0      | Tweet [tweetID=1, text=Learning RxJava12]
273      | Sched-A-0      | Got Tweet [tweetID=1, text=Learning RxJava12]
273      | Sched-A-0      | Tweet [tweetID=2, text=Understaning Rx]
273      | Sched-A-0      | Tweet [tweetID=2, text=Understaning Rx1]
273      | Sched-A-0      | Tweet [tweetID=2, text=Understaning Rx12]
273      | Sched-A-0      | Got Tweet [tweetID=2, text=Understaning Rx12]
273      | Sched-A-0      | Tweet [tweetID=3, text=RxJava Observables]
273      | Sched-A-0      | Tweet [tweetID=3, text=RxJava Observables1]
273      | Sched-A-0      | Tweet [tweetID=3, text=RxJava Observables12]
273      | Sched-A-0      | Got Tweet [tweetID=3, text=RxJava Observables12]
273      | Sched-A-0      | Tweet [tweetID=4, text=Learning Transformation]
```

All transformation execution happen in the same thread A.

Adding method to simulate sending SMS asynchronously. There is a random thread sleep to simulate latency while sending SMS.

```
/*
	 * Obtaining Async per message across the transformation pipeline.
	 */
	/*
	 * Casting return object to Observable.
	 */
	static public Observable<Tweet> sms(Tweet t) {
		return Observable.fromCallable(() -> (Tweet) sendSMS(t));
	}

	static public Object sendSMS(Tweet obj) {

		log(" Sending SMS  " + obj.toString());
		try {
			int random = (int) (5 * Math.random() + 3 );
			Thread.sleep(random * 1000);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		log(" Sent SMS " + obj.getTweetID());
		return obj;
	}
```

Using SubscribeOn to Process Tweet Asynchronously.

```
/*
     *  Using SubscribeOn to Process Asynchrnously
     */
     static void understandingAsyncExecutionPerMesage() {

          log("Starting");
          final Observable<Tweet> obs = simple();
          log("Created");
          final Observable<Tweet> obs2 = obs.filter(x -> true)
                    .flatMap(x -> sms(x).subscribeOn(myScheduler("B")));
          log("Transformed");
          obs2.subscribeOn(myScheduler("A"))// Using A Scheduler.
                    .subscribe(x -> {
                    }, Throwable::printStackTrace, () -> log("Completed1"));
          log("Exiting");
```

Execute the method.

```
 */
static void understandingAsyncExecutionPerMesage() {

    log("Starting");
    final Observable<Tweet> obs = simple();
    log("Created");
    final Observable<Tweet> obs2 = obs.filter(x -> true)
            .flatMap(x -> sms(x).subscribeOn(myScheduler("B")));
    log("Transformed");
    obs2.subscribeOn(myScheduler("A"))// Using A Scheduler.
            .subscribe(x -> {
            }, Throwable::printStackTrace, () -> log("Completed1"));
    log("Exiting");
}
```

Console

```
MyScheduler [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (03-Feb-2021, 3:00:37
1    | main      | Starting
141  | main      | Created
366  | main      | Transformed
430  | main      | Exiting
438  | Sched-A-0 | Create
441  | Sched-B-0 |  Sending SMS  Tweet [tweetID=1, text=Learning RxJava]
441  | Sched-B-0 |  Sending SMS  Tweet [tweetID=4, text=Learning Transformation]
441  | Sched-B-0 |  Sending SMS  Tweet [tweetID=3, text=RxJava Observables]
441  | Sched-B-0 |  Sending SMS  Tweet [tweetID=2, text=Understaning Rx]
442  | Sched-B-0 |  Sending SMS  Tweet [tweetID=5, text=Learning ErrorHandling]
442  | Sched-A-0 | Completed
442  | Sched-B-0 |  Sending SMS  Tweet [tweetID=6, text=Learning BackPressure]
3447 | Sched-B-0 |  Sent SMS 5
3447 | Sched-B-0 |  Sent SMS 2
3447 | Sched-B-0 |  Sent SMS 1
5447 | Sched-B-0 |  Sent SMS 3
5447 | Sched-B-0 |  Sent SMS 6
7446 | Sched-B-0 |  Sent SMS 4
```

As you can observe there are three thread, caller thread -> main, the subscription thread → A and the SMS thread → Thread B.

Understanding ObserveOn

```
/*
      * Using ObserverOn:
      * All of the operators above observeOn are executed within client thread, which happens
     * to be the default in RxJava.
     * But below observeOn(), the operators are executed within the supplied Scheduler
      */
     static void understandingAsyncUsingObserverOn() {

          log("Starting");
          final Observable<Tweet> obs = simple();
          log("Created");
          final Observable<Tweet> obs2 = obs.doOnNext(x -> log("Found 1 - Tweet ID:" +
x.getTweetID() ))
                    .observeOn(myScheduler("A"))
               .doOnNext(x -> log("Found 2 - Tweet ID :" + x.getTweetID() ));
          log("Transformed");
          obs2.subscribe(x -> log("Got 1 - Tweet ID:" + x.getTweetID()),
Throwable::printStackTrace, () -> log("Completed"));
          log("Exiting");

     }
```

Execute the method.

```java
/*
 * Using ObserverOn: All of the operators above observeOn are executed within
 * client thread, which happens to be the default in RxJava. But below
 * observeOn(), the operators are executed within the supplied Scheduler
 */
static void understandingAsyncUsingObserverOn() {

    log("Starting");
    final Observable<Tweet> obs = simple();
    log("Created");
    final Observable<Tweet> obs2 = obs.doOnNext(x -> log("Found 1 - Tweet ID:" + x.getTweetID()))
            .observeOn(myScheduler("B")).doOnNext(x -> log("Found 2 - Tweet ID :" + x.getTweetID()));
    log("Transformed");
    obs2.subscribeOn(myScheduler("A"))
        .subscribe(x -> log("Got 1 - Tweet ID:" + x.getTweetID()), Throwable::printStackTrace,
            () -> log("Completed1"));
    log("Exiting");

```

Console

```
MyScheduler [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (03-Feb-2021, 3:11:21 PM)
355     | Sched-A-0     | Found 1 - Tweet ID:6
355     | Sched-B-0     | Got 1 - Tweet ID:1
356     | Sched-A-0     | Completed
356     | Sched-B-0     | Found 2 - Tweet ID :2
356     | Sched-B-0     | Got 1 - Tweet ID:2
356     | Sched-B-0     | Found 2 - Tweet ID :3
356     | Sched-B-0     | Got 1 - Tweet ID:3
356     | Sched-B-0     | Found 2 - Tweet ID :4
356     | Sched-B-0     | Got 1 - Tweet ID:4
356     | Sched-B-0     | Found 2 - Tweet ID :5
356     | Sched-B-0     | Got 1 - Tweet ID:5
356     | Sched-B-0     | Found 2 - Tweet ID :6
356     | Sched-B-0     | Got 1 - Tweet ID:6
356     | Sched-B-0     | Completed1
```

Normal code execute in main thread, Found – 1 * execute in Subscribeon Thread- A , Found 2 * & Got 1 * execute in observeOn thread – B and Completion also execute in Thread B which is of observeOn.

Observation Using ObserverOn:

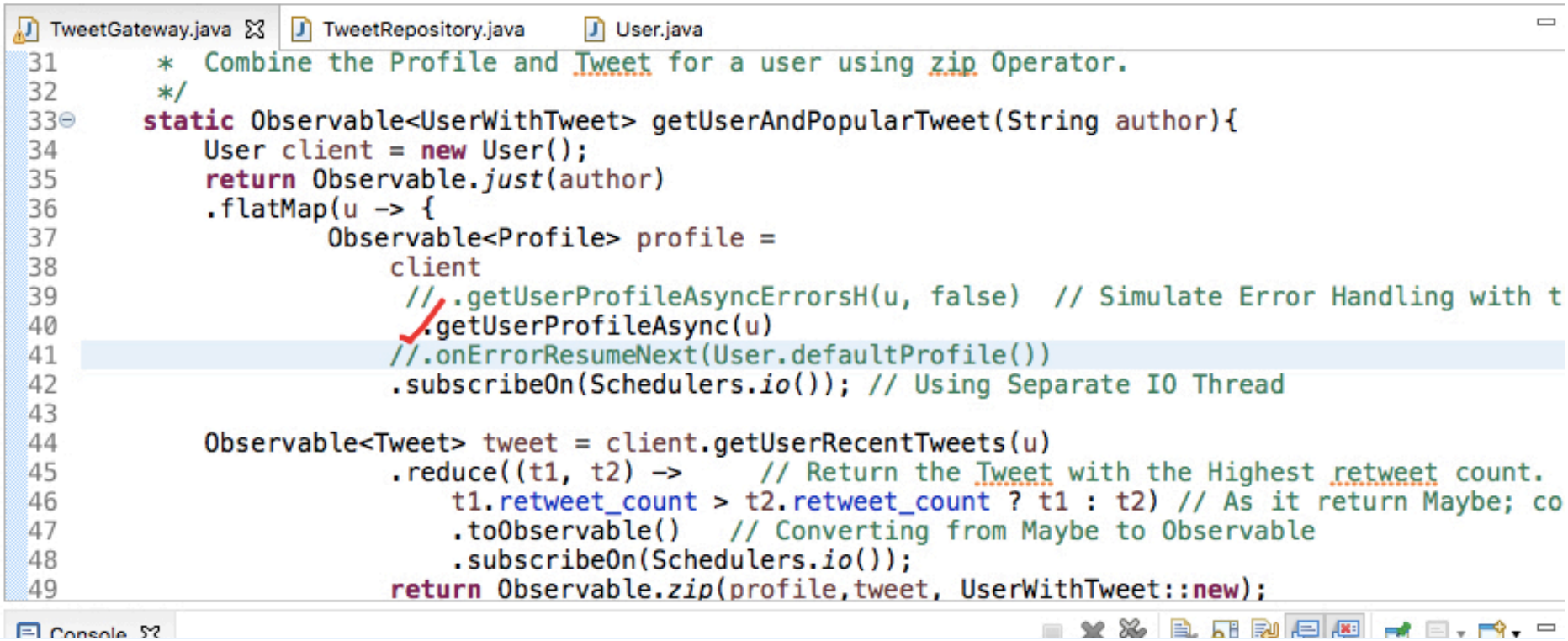
All operators above observeOn are executed within client subscribeOn thread, which happens to be the default in RxJava.
But remaining transformation below observeOn(), all operators are executed within the supplied Scheduler of observeOn.

Task to be done:

Understand the workflow in the TweetRX.

Execute the program without exception handler i.e

Use : `getUserProfileAsync(u) instead of getUserProfileAsyncErrorsH(u, false)`



Determine the Thread in which the following are being invoked in the `TweetGateway.`

```
getUserAndPopularTweet("kafka")
          .subscribe(System.out::println,System.out::println);
```

User Class:

```
public Observable<Profile> getUserProfileAsync(String screenName)
```

```
public Observable<Tweet> getUserRecentTweets(String author)
```

Define a method to log the thread in which the codes run.

```
    public static void log(Object msg) {
        System.out.println(Thread.currentThread().getName() + ": " + msg);
    }
```

Add the above method in any class, here I will define in the TweetGateway.java

Invoke the above method in the following areas.

1) TweetGateway.java

```
13 public class TweetGateway {
14
15     public static void main(String[] args) throws Exception {
16         // TODO Auto-generated method stub
17
18         /*
19          * Get Tweet and Profile for a Particular User
20          */
21         log(" Starting ");
22         getUserAndPopularTweet("kafka")
23             .subscribe(System.out::println,System.out::println);
24
25         log(" Ending ");
26         Thread.sleep(500); // Waiting for the Observable to complete.
27     }
```

2) User.java

TweetGateway.java | TweetRepository.java | User.java | ProductService.java | TweetServices.java

```
34          return prof;
35      }
36
37      /*
38       * Get Profile Using Observable
39       */
40      public Observable<Profile> getUserProfileAsync(String screenName) {
41          TweetGateway.log(" fetching Profile ");
42          return Observable
43                  .just(screenName)
44                  .flatMap(sn -> {
45                      Profile prof = null;
46                      switch (screenName) {
47                      case "rx_java":
48                          prof = new Profile("rx_java", "Rx Java", "Imphal", 3, 120);
49                          break;
50                      case "kafka":
51                          prof = new Profile("kafka", "Apache kafka", "Mumbai", 43, 80020);
52                          break;
53                      default:
54                          prof = new Profile("kafka", "Apache kafka", "Mumbai", 23, 220);
55                          break;
56                      }
57                      return Observable.just(prof);
58                  });
59          }
60      /*
```

3) User.java

TweetGateway.java | TweetRepository.java | User.java | ProductService.java | TweetServices.java

```
                                prof = new Profile("kafka", "Apache kafka", "Mumbai", 43, 80020);
                                break;
                            default:
                                prof = new Profile("Spark", "Apache Spark", "Mumbai", 23, 220);
                                break;
                            }
                            return Observable.just(prof);
                        });
            }else {
                return Observable.error(new RuntimeException("Can not connect to twitter" ));
            }
        }

        /*
         * Get all Tweets for a Specific Author.
         */
        public Observable<Tweet> getUserRecentTweets(String author) {
            TweetGateway.log(" fetching Recent Tweets ");
            Observable<Tweet> tweets = Observable.fromIterable(TweetRepository.fetchTweetsFromSource());
            Observable<Tweet> auT = tweets.filter( t -> t.getAuthor().equals(author));
            return auT;
        }
```

Now execute the Main method of TweetGateway.java

```
        /*
         * Get Tweet and Profile for a Particular User
         */
        log(" Starting ");
       getUserAndPopularTweet("kafka")
        //  .subscribeOn(Schedulers.computation())
          .subscribe( x ->{
              log(" Subscibe: ");
              System.out.println(x);
           }
            ,System.out::println);
       log(" Ending ");
       Thread.sleep(500); // Waiting for the Observable to complete.
    }

    /*
     *  Get the User Profile in a Separate Thread - IO Thread.
```

Console

<terminated> TweetGateway [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (22-Feb-2021, 2:27:38 PM – 2:27:40 PM

```
main:  Starting
main:  fetching Profile
main:  fetching Recent Tweets
main:  Ending
RxCachedThreadScheduler-2:  Subscibe:
UserWithTweet [profile=Profile [screen_name=kafka, name=Apache kafka, location=Mumbai, statuses_count=43
, tweet=Tweet [text=Kafka Quorum, favorite_count=15, author=kafka, author_followers=5128, retweet_count=
```

As you can observe all methods in the main method get invoked in the same thread-main, except the Subcriber Thread. If the code is not same as above, make the necessary changes so that you can write logic in the subscription method.

Let us invoke the fetching also in different threads.

```
19          * Get Tweet and Profile for a Particular User
20          */
21         log(" Starting ");
22        getUserAndPopularTweet("kafka")
23           .subscribeOn(Schedulers.computation())
24            .subscribe( x ->{
25                log(" Subscibe: ");
26                System.out.println(x);
27              }
28               ,System.out::println);
29        log(" Ending ");
30        Thread.sleep(500); // Waiting for the Observable to complete.
31    }
32
33    /*
34     *  Get the User Profile in a Separate Thread - IO Thread.
```

Console

```
<terminated> TweetGateway [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (22-Feb-2021, 2:32:01 PM
main:  Starting
main:  Ending
RxComputationThreadPool-1:  fetching Profile
RxComputationThreadPool-1:  fetching Recent Tweets
RxCachedThreadScheduler-2:  Subscibe:
UserWithTweet [profile=Profile [screen_name=kafka, name=Apache kafka, location=Mumbai, statuses
, tweet=Tweet [text=Kafka Quorum, favorite_count=15, author=kafka, author_followers=5128, retwe
```

Enable the subscribeOn and invoke the program.

---------- Lab Ends Here ----------

11. Using Subject & Throttles – 40 Minutes

Add the following class in the package : `com.tos.scheduler`

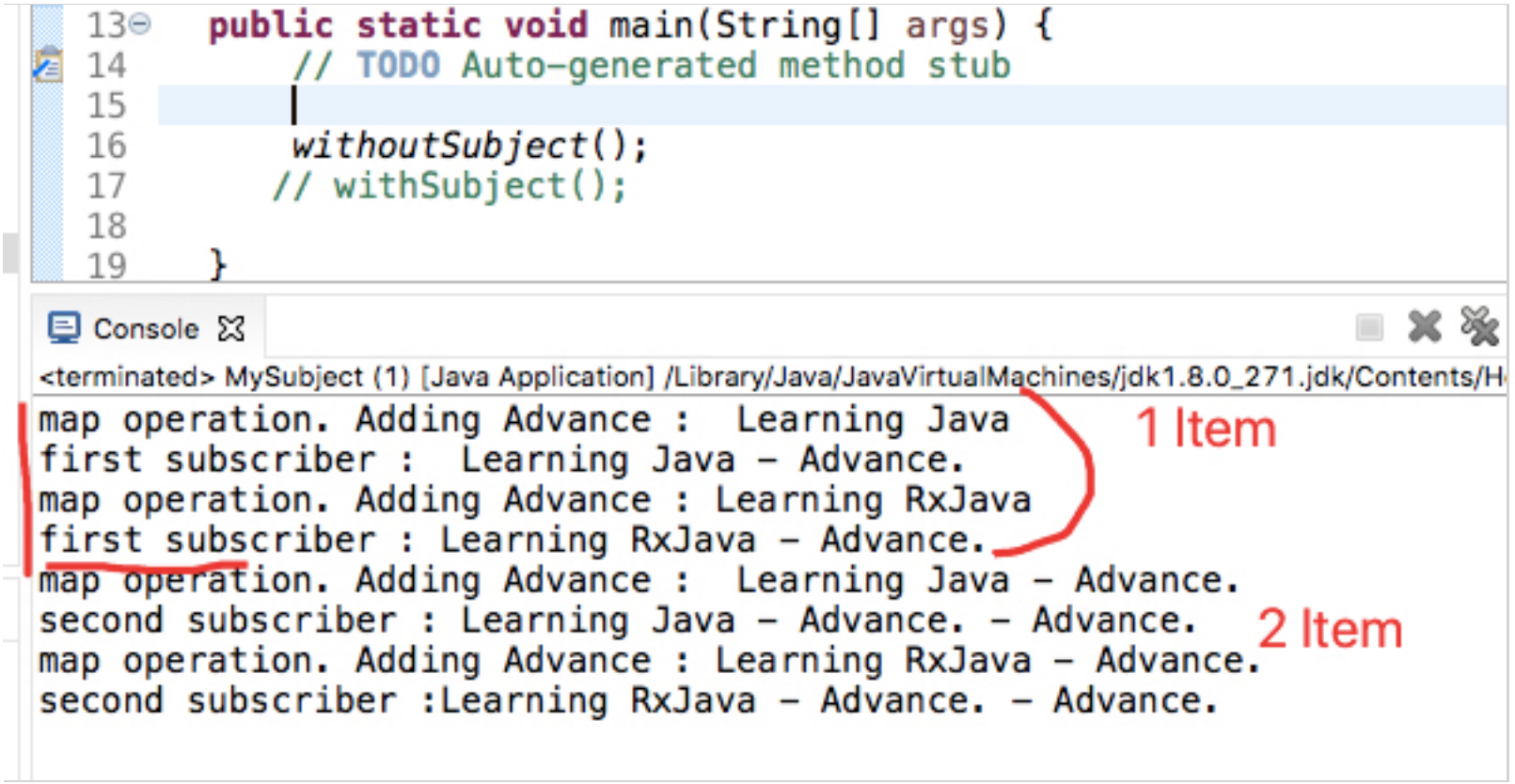
Create a class MySubject, it will demonstrate the PublishSubject object.

Add the following method.

```
/*
	*
	* The map operation is computed each time for each observer Without Subject.
	*
	*/
	public static void withoutSubject() {
		Observable<Tweet> observable =
				Observable.<Tweet>just(new Tweet("1"," Learning Java"), new
Tweet("2","Learning RxJava"))
				//.subscribeOn(Schedulers.computation())
				.map(val -> {
					System.out.println("map operation. Adding Advance : " +
val.getText());
					 val.setText(val.getText() + " - Advance.");
					return val;
				});
		// First Subscriber
		observable.subscribe( val -> {System.out.println("first subscriber : " +
val.getText());}  );
		// Second Subscriber
		observable.subscribe(l -> System.out.println("second subscriber :" + l.getText()));
	}
```

The above function has two subscribers on the same observable of type Tweet. The Transformation function map() get executed twice for each of the subscriber.

Invoke the above method from the main method and execute it.



Now let us implements the above function using Subscriber.

Let's assume you'd like to emit your own events, but you can't be sure when those events will be fired or how many of them will be there. Clearly, just() and from() can't help here and you don't want to create() an Observable which spins on some state either. The best would be an object that is both Observable, so clients can chain onto it, and Observer so you can emit values and terminal events as well. The combination of these two is what's called Subject

```
/*
      *   Let's assume you'd like to emit your own events,
      *   but you can't be sure when those events will be fired or
      *   how many of them will be there. Clearly, just() and from() can't help
here and
      *   you don't want to create() an Observable which spins on some state
either.
      *   The best would be an object that is both Observable, so clients can
chain onto it,
      *    and Observer so you can emit values and terminal events as well. The
combination of these two is what's called Subject
      */

 public static void withSubject() {
         PublishSubject<Tweet> pSubject = PublishSubject.create();
         pSubject.onNext(new Tweet("13"," Learning Quantum - Beginner"));
         pSubject.onNext(new Tweet("3"," Learning Quantum  - Intermediate"));
         pSubject.subscribe(new Observer<Tweet>() {
```

```
        @Override
        public void onSubscribe(Disposable d) {
            System.out.println("First Subscriber");
        }

        @Override
        public void onNext(Tweet tweet) {
            System.out.println("-->"+ tweet.getText());
        }

        @Override
        public void onError(Throwable e) {
            System.out.println("onError");
        }

        @Override
        public void onComplete() {
            System.out.println("onComplete");
        }
    });


pSubject.onNext(new Tweet("3"," Learning Quantum Advance"));

pSubject.subscribe(new Observer<Tweet>() {
        @Override
        public void onSubscribe(Disposable d) {
```

```
                System.out.println("Second Subscriber");
            }

            @Override
            public void onNext(Tweet tweet) {
             System.out.println(" Second --> "+tweet.getText());
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("onError");
            }

            @Override
            public void onComplete() {
                System.out.println("onComplete");
            }
        });

    pSubject.onNext(new Tweet("3"," Learning Quantum Expert"));

}
```

Execute the above function, what do you observe?

```
18          //withoutSubject();
19          withSubject();
20
21      }
```

Console

<terminated> MySubject (1) [Java Application] /Library/Java/JavaVirtualMachines/j

```
First Subscriber
--> Learning Quantum Advance
Second Subscriber
--> Learning Quantum Expert
 Second -->  Learning Quantum Expert
```

As you can observe,
Two Tweets – 13 and 23 being missed -
Tweet - Advance being consume by First Subscriber. [Hints – it subscribes before emitting the Tweet.]
Tweet – Expert getting consume by First and Second Subscriber. [Hints – Both subscribe before emitting the Tweet.]

Verify the logic carefully. Especially the flow of Tweets.

Next, let us understand the Sampling and the Backpressure strategies.

Create a class – LearningBackpressure

Import the following:

```
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.TimeUnit;

import com.tos.gateway.TweetGateway;
import com.tos.services.TweetServices;
import com.tos.vo.Tweet;

import io.reactivex.Observable;
import io.reactivex.ObservableEmitter;
import io.reactivex.ObservableOnSubscribe;
import io.reactivex.Observer;
```

Add the following method to simulate sampling:

```
public static void sampling() throws Exception {
        TweetGateway tc = new TweetGateway();
        Observable<Tweet> tweets = Observable.create(new
ObservableOnSubscribe<Tweet>() {

            @Override
            public void subscribe(ObservableEmitter<Tweet> emitter) throws
Exception {
                List<Tweet> ts = tc.getTweets();
                for (Iterator emi = ts.iterator(); emi.hasNext();) {

                    Tweet tweet = (Tweet) emi.next();
                    TweetServices.log("Sending : " + tweet);
                    emitter.onNext(tweet);
                    Thread.sleep(500);  // Simulating a delay of 30 seconds

                }

            }
        });

        tweets
        .sample(1, TimeUnit.SECONDS)  // Sampling at 1 seconds
        .map(ts -> ts.getText())
        .take(3)    // Fetch only 3 records.
        .subscribe(System.out::println);
```

```
        Thread.sleep(5000);
    }
```

Execute the method.

```
18      public static void main(String[] args) throws Exception {
19          // TODO Auto-generated method stub
20          sampling();
21      }
22
```

Console

<terminated> LearningBackpressure [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents

```
main: Sending : Tweet [tweetID=1, text=Learning RxJava]
main: Sending : Tweet [tweetID=2, text=Understaning Rx]
Understaning Rx
main: Sending : Tweet [tweetID=3, text=RxJava Observables]
main: Sending : Tweet [tweetID=4, text=Learning Transformation]
Learning Transformation
main: Sending : Tweet [tweetID=5, text=Learning ErrorHandling]
main: Sending : Tweet [tweetID=6, text=Learning BackPressure]
Learning BackPressure
```

As you can see, the tweet 1 is skip as tweet 3. Why? Sampling is done every 1 second and we are emitting the Tweet every 30 Seconds.

Try
```
.throttleFirst(1, TimeUnit.SECONDS)
```

and
```
.throttleLast(1, TimeUnit.SECONDS)
```

instead of sample.

Actually it emits the First or the last item of the window duration.

```
41          tweets
42         // .sample(1, TimeUnit.SECONDS)  // Sampling at 1 seconds
43          .throttleFirst(1, TimeUnit.SECONDS)
44          //.throttleLast(1, TimeUnit.SECONDS)
45          .map(ts -> ts.getText())
46          .take(3)     // Fetch only 3 records.
```

Console

<terminated> LearningBackpressure [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Hon

```
main: Sending : Tweet [tweetID=1, text=Learning RxJava]
Learning RxJava
main: Sending : Tweet [tweetID=2, text=Understaning Rx]
main: Sending : Tweet [tweetID=3, text=RxJava Observables]
RxJava Observables
main: Sending : Tweet [tweetID=4, text=Learning Transformation]
main: Sending : Tweet [tweetID=5, text=Learning ErrorHandling]
Learning ErrorHandling
main: Sending : Tweet [tweetID=6, text=Learning BackPressure]
```

Understand Buffer.

Add the following artifacts to simulate Database operations.

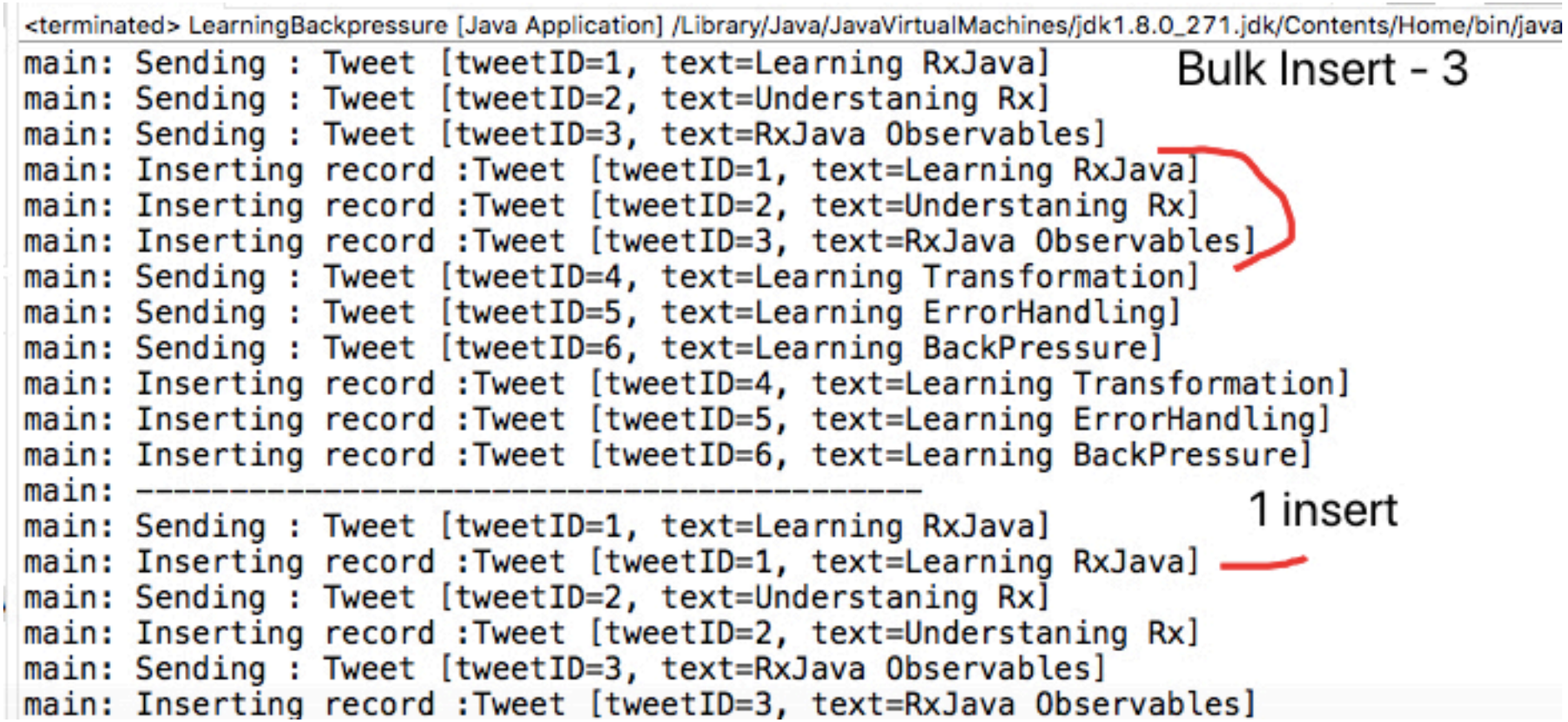
```
interface Repository {
        void store(Tweet record);
```

```
        void storeAll(List<Tweet> records);
    }

    static class TweetRepository implements Repository{

        @Override
        public void store(Tweet record) {

            TweetServices.log( "Inserting record :" + record);
        }

        @Override
        public void storeAll(List<Tweet> records) {
           for (Tweet tweet : records) {
            TweetServices.log( "Inserting record :" + tweet);
          }
        }

    }


public static void bufferDBOperation() throws Exception {
        TweetGateway tc = new TweetGateway();
        Observable<Tweet> tweets = Observable.create(new
ObservableOnSubscribe<Tweet>() {

            @Override
```

```
            public void subscribe(ObservableEmitter<Tweet> emitter) throws
Exception {
                List<Tweet> ts = tc.getTweets();
                for (Iterator emi = ts.iterator(); emi.hasNext();) {

                    Tweet tweet = (Tweet) emi.next();
                    TweetServices.log("Sending : " + tweet);
                    emitter.onNext(tweet);
                    Thread.sleep(500);  // Simulating a delay of 30 seconds
                }

            }
        });

        Repository tweetRepo = new TweetRepository();
        tweets
       .buffer(3)
        .subscribe(tweetRepo::storeAll);

        TweetServices.log("----------------------------------------");
        tweets
       .subscribe(tweetRepo::store);


    }
```

Using Buffer. It invokes multiple data pushes to the Database depending on the requirement.

Execute the method.



Debounce operator.

Debounce operator enables to emit the last item after a configurable debounce period.

Add the following method:

```
/*
     * Debounce operator enables to emit the last item after a configurable
     * debounce period.
     */
    public static void debounceOperation() throws Exception {
        TweetGateway tc = new TweetGateway();
        Observable<Tweet> tweets = Observable.create(new
ObservableOnSubscribe<Tweet>() {

            @Override
            public void subscribe(ObservableEmitter<Tweet> emitter) throws
Exception {
                    Tweet tweet = new Tweet("1","Rx Java");
                    emitter.onNext(tweet);
                    Thread.sleep(500);  // Simulating a delay of 30 seconds
                    tweet = new Tweet("2","Rx Java");
                    Thread.sleep(500);  // Simulating a delay of 30 seconds
                    emitter.onNext(tweet);
                    tweet = new Tweet("3","Rx Java");
```

```
                    Thread.sleep(500);  // Simulating a delay of 30 seconds
                    emitter.onNext(tweet);
                    tweet = new Tweet("4","Rx Java");
                    Thread.sleep(500);  // Simulating a delay of 30 seconds
                    emitter.onNext(tweet);
                }

        });

        // Using debounce, pushing record after debounce period i.e 1500 milli
seconds.
        Repository tweetRepo = new TweetRepository();
        tweets
       .debounce(1500, TimeUnit.MILLISECONDS)
        .map(item -> item.getTweetID())
       .subscribe(System.out::println);

        /*
         * It will emits only the 4 - Tweet ID since the debounce period is 1500
ms.
         *
         */
        Thread.sleep(5000);
    }
```

Execute the method.

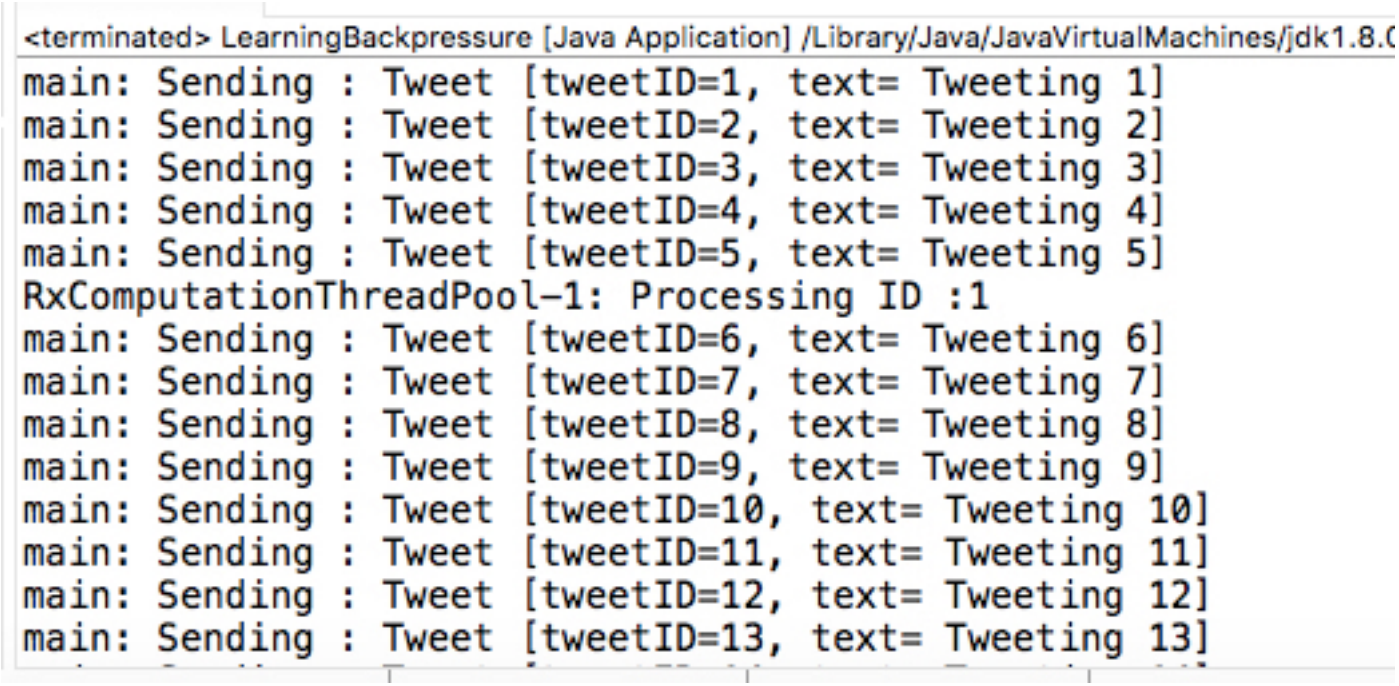
Backpressure.

Without Backpressure, the main publisher can push the way it is. Subscriber will be consuming at its own rate. It may lose some of the messages.

```
     /*
      * Understanding the BackPressure:
      *  Pushing Messages more then it can consume.
      */
public static void UnderstandingBackpressure() throws Exception {

          TweetGateway tc = new TweetGateway();
          Observable<Tweet> tweets = Observable.create(new ObservableOnSubscribe<Tweet>() {
           /*
            * Pushing continously Tweet.
            */
              @Override
              public void subscribe(ObservableEmitter<Tweet> emitter) throws Exception {
                   int from = 0 ;
                   int count = 15;
                   int i = from;
                   while (i < from + count) {
                        i++;
                        Tweet tweet = new Tweet( String.valueOf(i),  " Tweeting " + i);
                        TweetServices.log("Sending : " + tweet);
                        emitter.onNext(tweet);
                   }
              }

          });
```

```
        tweets.map( t -> t.getTweetID())
              // .subscribeOn(Schedulers.io())
               .observeOn(Schedulers.computation())
               .subscribe(id -> {
                /*
                 * Processing with Delay
                 */
                TweetServices.log("Processing ID :" + id);
                Thread.sleep(500);
               });

        Thread.sleep(9000);
    }
```

Execute the method.



Publishing more message than the processing speed of the subscriber.

Using Buffer strategy.

```
/*
      * Using Buffer to hold the message.
      */
     public static void backpressureUsingFlow() throws Exception {
          Flowable<Tweet> tweets = Flowable.create(new FlowableOnSubscribe<Tweet>() {
               /*
                * Pushing continously Tweet.
```

```
            */
                        @Override
                public void subscribe(FlowableEmitter<Tweet> emitter) throws Exception {
                     // TODO Auto-generated method stub
                     int from = 0 ;
                     int count = 1000;
                     int i = from;
                     while (i < from + count) {
                          i++;
                          Tweet tweet = new Tweet( String.valueOf(i),  " Tweeting " + i);
                          TweetServices.log("Sending : " + tweet);
                          emitter.onNext(tweet);
                          //Thread.sleep(400);  // Simulating a delay of 30 seconds
                     }
                }

            },BackpressureStrategy.BUFFER);

DefaultSubscriber<String> ds = new DefaultSubscriber<String>() {

     @Override
     protected void onStart() {
          // TODO Auto-generated method stub
          super.onStart();
          TweetServices.log("Start");
          request(1);
     }
     @Override
     public void onNext(String t) {
          // TODO Auto-generated method stub
           TweetServices.log("Processing ID :" + t);
           request(1);
```

```
            try {
          Thread.sleep(500);
              }
             catch(Exception e) {
                    onError(e);
             }
      }

      @Override
      public void onError(Throwable e) {
           // TODO Auto-generated method stub

      }

      @Override
      public void onComplete() {
           // TODO Auto-generated method stub
          TweetServices.log("Done");
      }


};


tweets.map( t -> t.getTweetID())
.observeOn(Schedulers.computation())
.subscribe(ds);



      Thread.sleep(9000);
}
```

Invoke the previous method.

```
240                     // TODO Auto-generated method stub
241                     int from = 0 ;
242                     int count = 1000;
243                     int i = from;
244                     while (i < from + count) {
245                         i++;
246                         Tweet tweet = new Tweet( String.valueOf(i),  " Tweeting " + i);
247                         TweetServices.log("Sending : " + tweet);
248                         emitter.onNext(tweet);
249                         //Thread.sleep(400);  // Simulating a delay of 30 seconds
250                     }
251                 }
252
253             },BackpressureStrategy.BUFFER);
254
255     DefaultSubscriber<String> ds = new DefaultSubscriber<String>() {
256
257         @Override
```

Console

```
<terminated> LearningBackpressure [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java
RxComputationThreadPool-1: Processing ID :6
RxComputationThreadPool-1: Processing ID :7
RxComputationThreadPool-1: Processing ID :8
RxComputationThreadPool-1: Processing ID :9
RxComputationThreadPool-1: Processing ID :10
RxComputationThreadPool-1: Processing ID :11
RxComputationThreadPool-1: Processing ID :12
RxComputationThreadPool-1: Processing ID :13
RxComputationThreadPool-1: Processing ID :14
RxComputationThreadPool-1: Processing ID :15
RxComputationThreadPool-1: Processing ID :16
RxComputationThreadPool-1: Processing ID :17
RxComputationThreadPool-1: Processing ID :18
```

Another ways of specify Backpressure Strategy.

```
/*
 * using Backpressure
 */
     public static void backPressureHotSource() throws Exception {
          PublishProcessor<Integer> source = PublishProcessor.create();

          DisposableSubscriber<Integer> ds = new DisposableSubscriber<Integer>() {
             @Override
             public void onStart() {
                 request(1);
             }

             public void onNext(Integer v) {
                 try {
                          compute(v);
                    } catch (Exception e) {
                          // TODO Auto-generated catch block
                          onError(e);
                    }

                 request(1);
             }

             @Override
             public void onError(Throwable ex) {
                 ex.printStackTrace();
             }

             @Override
```

```
            public void onComplete() {
                System.out.println("Done!");
            }
        };

        source
          .onBackpressureBuffer()
          .observeOn(Schedulers.computation())
          .subscribe(ds);
        for (int i = 0; i < 100; i++) {
            source.onNext(i);
            TweetServices.log("Sening: " + i);
        }



        Thread.sleep(20000);

    }

    public static void compute(int i) throws Exception {
         TweetServices.log("Processing:" + i);
          Thread.sleep(300);
    }
```

Execute it

```
38      //  backPressureUsingFlow();
39          backPressureHotSource();
40
41      }
42
```

Console

```
LearningBackpressure [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java  (16-
RxComputationThreadPool-1: Processing:18
RxComputationThreadPool-1: Processing:19
RxComputationThreadPool-1: Processing:20
RxComputationThreadPool-1: Processing:21
RxComputationThreadPool-1: Processing:22
RxComputationThreadPool-1: Processing:23
RxComputationThreadPool-1: Processing:24
RxComputationThreadPool-1: Processing:25
RxComputationThreadPool-1: Processing:26
RxComputationThreadPool-1: Processing:27
RxComputationThreadPool-1: Processing:28
RxComputationThreadPool-1: Processing:29
RxComputationThreadPool-1: Processing:30
```

Notes:

periodic sampling is enough. The most obvious case is receiving measurements from some device; for example, temperature

The buffer() operator aggregates batches of events in real time into a List. However, unlike the toList() operator, buffer() emits several lists grouping some number of subsequent events as opposed to just one containing all events (like toList()).

Window is same with buffer but return Observable.

Debounce

Backpressure.
Different strategy.

----------Lab Ends Here ----------

12. Further Reading

https://blog.avenuecode.com/reactive-streams-and-microservices-a-case-study
https://medium.com/joinpre/using-observables-to-render-responsive-lists-an-rxjava-case-study-part-1-of-3-53dd83af2c08
https://www.slideshare.net/InfoQ/functional-reactive-programming-in-the-netflix-api