Observer Pattern

RX-Java simplifies the use of `Observer-pattern` (https://en.wikipedia.org/wiki/Observer_pattern) in applications.

A Subject registers and maintains a list of `Observers` with a specific 'notify' method on them. It calls these observers when it receives a new message by invoking their notify method.

First, in the observer pattern implementation, the subject maintains a list of observers subscribed to it. Now, the actual event or message generation may be separated from this subject. In other words, some other entity generates the data and then sends message to the subject. The subject then notifies the observers.

Now, in the Rx world, the event generation is mostly innate to the Subject(the observer pattern) and this subject is called an `Observable`. This observable can generate one message or a stream of messages and multiple subscribers (observers from the pattern) can subscribe to it.

In many cases, there may be only one subscriber since there may be only one action desired for an event. The notify method in observer is called `onNext()` in Rx

Subject is a concept where an observer is fused with a subject. In the Rx parlance, a subscriber is fused with an observable. This allows one to call `onNext()` method on the subject in order to generate a message. This message is then in turn relayed to the subscriber to perform actions.

Perpetual Machine

An observable can be turned into a perpetual machine to generate events, as described in the snippet on observer pattern from wiki,

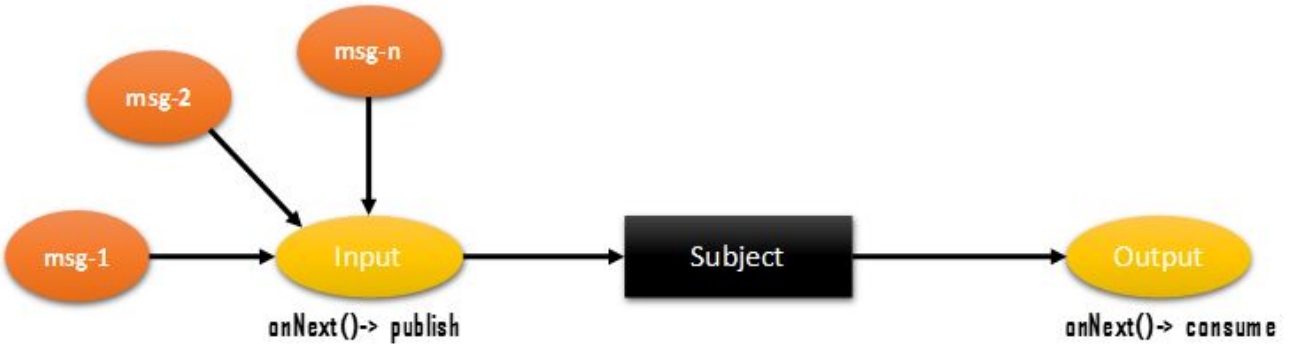
```
import java.util.Observable;
import java.util.Scanner;

class EventSource extends Observable implements Runnable {
  public void run() {
    while (true) {
      String response = new Scanner(System.in).next();
      setChanged();
      notifyObservers(response);
    }
  }
}
```

Important thing to realize in this is that first of all, we have a perpetual `while` loop and second the event comes from an external source which in this case is keyboard.

Now comes the important question. How do we 'inject' a new event on a perpetual basis into this observable?

This is where the concept of subject as observable comes in handy. We want the subject to be hot - continuously generating events. We would like it be receive events from outside by providing methods for external consumption. In effect, the subject needs to have an input as well as an output.

Event Bus



This looks more and more like an event bus which has the ability to receive events and redirect events. One crucial thing to note from the picture is that the act of publishing is happening to the input end and additionally, this publishing is occurring from different parts of the application, at different times in an arbitrary and perpetual manner. Consequently, there are different threads performing the act of publishing.

Now, this breaks the observable contract in Rx Java which states that the events need to be in a predictable sequence so as not to cause a race condition.

In order to prevent this catastrophe, we need to serialize the subject so that multiple threads can still publish to it.

In the context of a Spring application, we can use a simple procedure to implement this subject for message consumption and processing.

```
safeSource = PublishSubject.create().toSerialized();
safeSource.subscribe(new Subscriber() {

    @Override
    public void onCompleted() {
      LOGGER.debug("Batch completed..");
    }

    @Override
    public void onError(Throwable e) {
      LOGGER.error("ERR: in batch processing", e);
    }

    @Override
    public void onNext(String string) {
      LOGGER.debug(">>> received :{}", string);
    }
 }
);
```

In the `onNext()` method, we can use any processing function or logic. In this way, we can wire a processor or output into the subject or the event bus.

Now, how to do we inject or input messages into this bus?

```
safeSource.onNext(s);
```

This completes the input and output ends of the event bus. It's a self-contained, perpetual processing machine.

Wiring

Now, we need to create this machine inside an application and start it so that it can perform its job.

In the context of a Spring based java application, we can use a straightforward procedure to achieve this.

First, we can create a 'Component' to house this logic. This will be any class with a '@Component' annotation. We can add the subject creation in the constructor so that on the application startup, the event bus is ready to process data.

Next, we need to provide an interface to external parts of the application to make use of this bus. This can be done by providing a public method to call the 'onNext()' method.

This wiring can be done as described below,

```
@Component
public class MessageProcessor {
private SerializedSubject safeSource;

@Autowired
public MessageProcessor() {
safeSource = PublishSubject.create().toSerialized();
safeSource.subscribe(new Subscriber() {
@Override
public void onCompleted() {
LOGGER.debug("Batch completed..");
}

@Override
public void onError(Throwable e) {
LOGGER.error("ERR: in batch processing", e);
}

@Override
public void onNext(String string) {
LOGGER.debug(">>> received :{}", string);
}
});

}

public void publish(String s) {
LOGGER.debug("[{}]... publishing to the eventBus",    Thread.currentThread().getName());
safeSource.onNext(s);
}
}
```

Bulk processing

Now, let's say that we need to use this bus to perform bulk or batch processing. One use case is that we need to perform database inserts. Rather than insert a single record at a time, if we can use batch processing, then the application can improve its data processing performance. The application can leverage various JDBC batching mechanisms to improve its throughput.

First imperative for bulk processing is that we need to convert this perpetual and arbitrary stream of data into a batch of predictable size. This is similar to a leaky-bucket for shaping the network traffic, Leaky bucket. We have a burst of water droplets arriving and we capture it in a bucket and release the bucket at a constant and predictable rate.

There are two primitives by which such a bucket or batch can be generated - one is based on time and the other is based on number of messages. We can say that after a certain period of time, the batch is complete regardless of number of messages in it and release it. We can also say that the batch shall contain at most x number of messages and release it when it does accumulate x messages regardless of time taken to accrue them.

These two primitives are implemented in Rx-Java via a concept called, 'buffer'. A buffer can be built based on count or a time interval. It is explained in more detail here,
http://reactivex.io/documentation/operators/buffer.html

The implementation is ridiculously trivial via Rx-Java,

```
safeSource = PublishSubject.create().toSerialized();
safeSource
.buffer(5, TimeUnit.SECONDS, 50)
.subscribe(new Subscriber() {
@Override
public void onCompleted() {
LOGGER.debug("Batch completed..");
}

@Override
public void onError(Throwable e) {
LOGGER.error("ERR: in batch processing", e);
}

@Override
public void onNext(List strings) {
LOGGER.debug(">>> Batch size:{}", strings.size());
}
});
```

The important thing to note is the method signature of the 'onNext()' method. The method now receives a parameter of type 'List' as opposed to just a String as in the previous step. This is a consequence of the 'buffer()' clause. The buffer clause takes 2 arguments - time and count. This buffer is constructed based on a time window of 5 seconds or a message count of 50, whichever occurs first. That's it.

Backpressure

What happens when the subscriber cannot keep up with the rate of production by the observable? This is where the performance and response time start to suffer. the worst part of this degradation is that it is unpredictable or unplanned. Backpressure moves the problem up the chain of processors and adds predictability to the process.

First defense is added by the 'buffer' capability itself. Since it prevents the observable from emitting each individual message, the load on the subscriber is reduced already.

Secondly, we can specify a backpressure buffer which will hold the messages into a buffer before releasing them to the subscribers.

Again, the implementation is trivial in Rx-Java,

```
safeSource
.buffer(5, TimeUnit.SECONDS, 50)
.onBackpressureBuffer(bufferSize)
.subscribe(new Subscriber() {
@Override
public void onCompleted() {
LOGGER.debug("Batch completed..");
}

@Override
public void onError(Throwable e) {
LOGGER.error("ERR: in batch processing", e);
}

@Override
public void onNext(List strings) {
LOGGER.debug(">>> Batch size:{}", strings.size());
}
});
```

Multithreading

RX-Java performs operations in a sequential manner and as such the stream of events in serialized. There are two methods to change the thread of execution - 'observeOn()' and 'subscribeOn()'. Method 'observeOn()' is for the 'Observer' which in RX-Java parlance is the subscriber, whereas 'subscribeOn()' is for the Observable.

```
safeSource
.subscribeOn(Schedulers.newThread())
.buffer(bufferSpanSeconds, TimeUnit.SECONDS, bufferSize)
.observeOn(Schedulers.from(taskExecutor))
.subscribe(new Subscribe() {
..
} );
```

On running this code, we find the output from the onNext() method in the subscriber with the name of the thread from our task executor pool.

```
DEBUG 8380 --- [readScheduler-5] o.a.k.service.MessageProcessor : Batch size=50
```

However, if this operation takes moderate amount of time, that blocks the next call to the method. to demonstrate, we can add a 'sleep' call in the onNext() method,

```
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
e.printStackTrace();
}
```

This should output,

```
LOGGER.debug("Batch size:{}", strings.size());
```

With this code in the onNext() method, one might expect that multiple threads will be spawned and while one thread takes 10 seconds, the next message is processed and so on so forth.

However, we find an output that is sequential. Each message is serially processed and that too by the same thread.

This makes Observables not really parallel at all. What we have achieved is just push the execution to a different thread, but multiple threads are not spawned. Observables are sequential and the subscribeOn() or observeOn() won't intermingle the messages. That's the Observable contract.

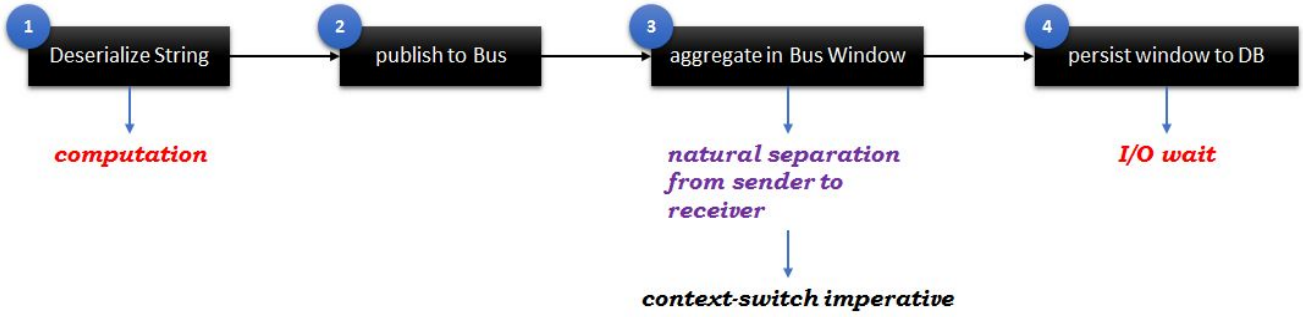
There are two ways to make this parallel. One, the computation in the onNext() method can simply be done using a new thread. Two, we can make use of the Observable again and spawn the work on to another thread as follows,

```
@Override
public void onNext(List strings) {
Observable.just(strings).
observeOn(Schedulers.newThread()).
subscribe(strings1 -> LOGGER.debug(">> Batch size={}",
strings1.size()));
}
```

Observable.just() converts the List into an Observable and then emits it on a new thread.

Long-running computation

It's neither sufficient nor necessary to just move the computation off to another thread. In fact, it may even impact the performance adversely due to unnecessary context switching. Let's consider an example where we wish to convert a string to an object and then put it on a bus to aggregate by window and finally to persist the window as a batch to the database. Various computation activities may be identified as follows,



There are two long running, compute intensive tasks - desrialize using jackson and jdbc batch persist to db. Naturally, it's important to quickly hand these tasks out on different threads so that more requests can be serviced. Additionally, there is a natural breaking point in the process - the aggregation in the event bus. This aggregation is by message count as well as time. Therefore there is an inherent separation between deserializing, publishing and the rest of the processing. As such, there will be a context switch and it should be ok to move this second phase of computation on to a different thread.

Concurrent Emission

As seen from above, we need to achieve multi-threaded emission. This can be achieved by converting the incoming message to a new Observable via `flatmap` as follows,

```
public void publish(String s) {
LOGGER.debug("[{}]... publishing now",
Thread.currentThread().getName());
Observable.just(s).
flatMap(s1 -> Observable.just(s1)
.subscribeOn(Schedulers.computation())
.map(this::unmarshalMetadata))
.subscribe(kafkaMessage -> {
LOGGER.info(">> sent to Bus on->{}",
Thread.currentThread().getName());
safeKafkaMessageSource.onNext(kafkaMessage);
});
LOGGER.debug(">>[{}]->Publishing completed",
Thread.currentThread().getName());
}
```

In this case, `unmarshalMetadata` is an expensive computation, relatively speaking and as such is put on a new thread. The last line should tell us whether there is any blocking. This prints following output,

```
2017-07-24 09:43:19.516 DEBUG 12136 --- [nio-8080-exec-9] o.a.k.service.MessageProcessor :
[http-nio-8080-exec-9]... publishing now
2017-07-24 09:43:19.548 DEBUG 12136 --- [nio-8080-exec-9] o.a.k.service.MessageProcessor : >>
[http-nio-8080-exec-9]->Publishing completed
2017-07-24 09:43:19.548 INFO 12136 --- [tionScheduler-3] o.a.k.service.MessageProcessor : >>
Unmarshal on->RxComputationScheduler-3
2017-07-24 09:43:23.751 INFO 12136 --- [tionScheduler-3] o.a.k.service.MessageProcessor : >>
sent to Bus on->RxComputationScheduler-3
```

We can see that the publishing starts on thread `http-nio-8080-exec-9` and finishes on it in 32 ms. The deserialization task proceeds on `RxComputationScheduler-3` and finishes after 4 seconds. if we add the following statements to the bus Subject,

```
@Override
public void onNext(List strings) {
LOGGER.debug("Arrived on Bus on:{}",
Thread.currentThread().getName());
Observable.just(strings).observeOn(Schedulers.newThread())
.subscribe(strings1-> LOGGER.debug("Batch size={}",
strings1.size()));
}
```

we now see the output as,

```
2017-07-24 09:43:26.716 DEBUG 12136 --- [ RXbus-1] o.a.k.service.MessageProcessor : >>>>>
Arrived on Bus on:RXbus-1
2017-07-24 09:43:26.716 DEBUG 12136 --- [readScheduler-2] o.a.k.service.MessageProcessor : >>
Batch size=1
```

The subscriber method is invoked on a new thread with a batch of deserialized messages and then the actual long-running db persist task is handed out to a different thread, in this case `readScheduler-2` while the onNext() method is ready to serve the next batch of messages.

In this manner, we see a clear hand-off at desired points.