Challenges

- Sequential composition of operations
  - Check identity in database, update database, query remote service, …
- Multiple subscribers to the same stream
- Back to the chat application!

Sequential blocking operation

```
var s_flow = loop.read(conn)
        .observeOn(computation())
        .lift(new LineSplitOperator())
        .map(bb -> UTF_8.decode(bb))


        ???



        .map(s -> UTF_8.encode(s));
loop.write(s_flow, conn);
```

```
connection
        .createStatement("INSERT ...")
        .bind("$1", s)
        .execute();
```

Returns a reactive stream

Sequential blocking operation

```
var s_flow = loop.read(conn)
        .observeOn(computation())
        .lift(new LineSplitOperator())
        .map(bb -> UTF_8.decode(bb))

        .observeOn(io())
        .map(s -> ...)
        .observeOn(computation())


        .map(s -> UTF_8.encode(s));
loop.write(s_flow, conn);
```

R2DBC

```
connection
        .createStatement("INSERT ...")
        .bind("$1", s)
        .execute()
        .blockingSubscribe();
```

Composition operator

```
public class MapOperator implements FlowableOperator<R,T> {
    private Function<T,R> op;

    public Subscriber<T> apply(Subscriber<R> child) throws Throwable {
        return new Subscriber<T>() {

            public void onNext(T data) {
                R result = op.apply(data);
                child.onNext(result);
            }
     }
}
```

Database code would run here….

Transformation

- The generic strategy for declaring a non-blocking operation
  - For some function:

    ```
    R op(T t)
    ```

  - transform it to:

    ```
    Flowable<R> op(T t)
    ```

Composition operator

```
public class BetterMapOperator implements FlowableOperator<R,T> {
    private Function<T,Flowable<R>> op;

    public Subscriber<T> apply(Subscriber<R> child) throws Throwable {
        return new Subscriber<T>() {

            public void onNext(T data) {
                Flowable<R> rflow = op.apply(data);
                child.onNext(???);
            }
     }
}
```

```
connection
        .createStatement("INSERT ...")
        .bind("$1", s)
        .execute()
        ~~.blockingSubscribe();~~
```

Transformation

- The generic strategy to consume the result from a non-blocking operation:
  - Instead of:

    ```
    var result = op();
    other(result);
    ```

  - do:

    ```
    op().subscribe((result) -> other(result));
    ```

IMPORTANT

Composition operator

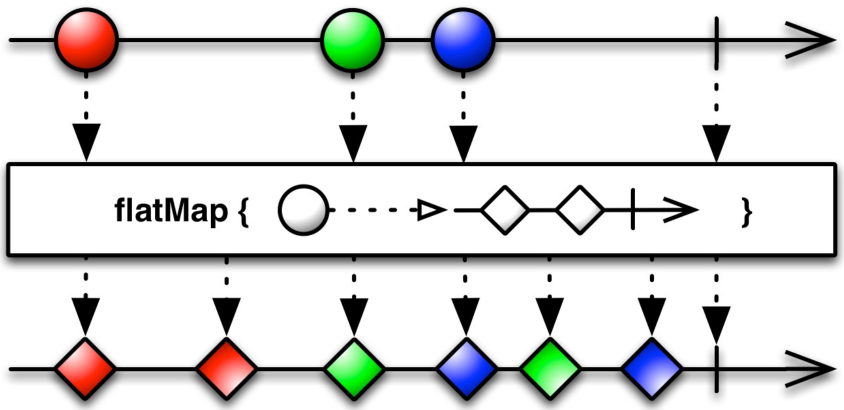
```
public class BetterMapOperator implements FlowableOperator<R,T> {
   private Function<T,Flowable<R>> op;

   public Subscriber<T> apply(Subscriber<R> child) throws Throwable {
      return new Subscriber<T>() {

         public void onNext(T data) {
            Flowable<R> rflow = op.apply(data);
            rflow.subscribe(result -> {
                  child.onNext(result);
            });
         }
    }
}
```

⚠ needs thread synchronization

FlatMap operator



- Asynchronous sequential composition, if each inner publisher returns one result
- map(v->f(v)) <=> flatMap(v->just(f(v))
- filter(v->cond(v)) <=> flatMap(v→cond.test(v)?just(v):empty())
- Out-of-order parallel processing by using subscribeOn() in inner streams

Sequential composition with flatMap

```
var s_flow = loop.read(conn)
        .observeOn(computation())
        .lift(new LineSplitOperator())
        .map(bb -> UTF_8.decode(bb))

        .flatMap(s -> connection
                .createStatement("INSERT ...")
                .bind("$1", s)
                .execute())

        .map(s -> UTF_8.encode(s));
loop.write(s_flow, conn);
```

Actually… we need to open the connection that is itself a blocking operation...

Sequential composition with flatMap

```
var cf = ConnectionFactories
            .get("r2dbc:h2:mem:///testdb");
```

```
var s_flow = loop.read(conn)
      .observeOn(computation())
      .lift(new LineSplitOperator())
      .map(bb -> UTF_8.decode(bb))
      .flatMap(s -> Flowable.fromPublisher(cf.create())
            .flatMap(connection -> connection
                  .createStatement("INSERT ...")
                  .bind("$1", s)
                  .execute()
            )
      .map(s -> UTF_8.encode(s));
loop.write(s_flow, conn);
```

Manual translation

```
R op() {
    var a, b, … = …

    a = blockingOp1();

    b = op2(a);
    a = blockingOp3(b, a);

    return a;
}
```

```
Flowable<R> op() {
    var v0 = (a, b, …)
    return Flowable.just(v0).
        .flatMap(v1 -> blockingOp1()
            .map(r -> (r, v1.b, …)
            .map(v2 -> (v2.a, op2(v2.b), …)
            .flatMap(v3 -> blockingOp3()
                .map(r -> (a, v3.b, …)
                .map(v4 -> v4.a);
}
```

- Possible and mechanical, if blocking operations return a single result (i.e., don’t “fork”)

Manual translation

- Also possible with control flow: if, while, try, catch, …
  - … but MUCH harder to do manually!
  - see Single.repeat(), repeatUntil(), …
- Key difference to imperative code:
  - the reactive code is lazy
    - nothing actually happens until the return value is used (i.e., subscribed to)
    - the reactive code is only creating a structure in memory with paths for future execution

Automatic translation with async/await

```
async R op() {
    var a, b, … = …

    a = await
        blockingOp1();
    b = op2(a);
    a = await
        blockingOp3(b, a);
    return a;
}
```

```
Flowable<R> op() {
    var v0 = (a, b, …)
    return Flowable.just(v0).
        .flatMap(v1 -> blockingOp1()
            .map(r -> (r, v1.b, …)
            .map(v2 -> (v2.a, op2(v2.b), …)
            .flatMap(v3 -> blockingOp3()
                .map(r -> (a, v3.b, …)
                .map(v4 -> v4.a);
}
```

- Many languages do the translation with async/await keywords: Python, JavaScript, Rust, …

Async/await in Java

- Library by ElectronicArts based on CompletableFuture (not lazy, always restricted to a single value):
  - https://github.com/electronicarts/ea-async
- Fibers from “Project Loom” in Java 20, without explicit async/await keywords

Stream composition

- There are other stream composition operators:
  - merge(), concat(), zip(), join(), …
  - … but flatMap() does a lot!

- Remember that:
  - We have a stream of SocketChannel
  - And read() maps SocketChannel to stream of ByteBuffer

Complete simple application + flatMap

```
public class Server {
        public void server() throws Exception {
                var loop = new MainLoop();
                loop.accept(ServerSocketChannel.open(new InetSocketAddress(12345));)
                        .flatMap(conn -> loop.read(conn)
                                .observeOn(computation())
                                .lift(new LineSplitOperator())
                                .map(bb -> StandardCharsets.UTF_8.decode(bb))
                        )
                        .subscribe(s -> System.out.println(s));
        }
}
```

Flatten all incoming streams to one instead of subscribing them

Subscribing to the resulting stream

Complete simple application + flatMap

```
public class Server {
    public void server() throws Exception {
        var loop = new MainLoop();
        loop.accept(ServerSocketChannel.open(new InetSocketAddress(12345));)
            .flatMap(conn -> loop.read(conn)
                .observeOn(computation())
                .lift(new LineSplitOperator())
            )
            .map(bb -> StandardCharsets.UTF_8.decode(bb))
            .subscribe(s -> System.out.println(s));
    }
}
```

Separate line split for each stream

Need only one conversion operator (stateless)

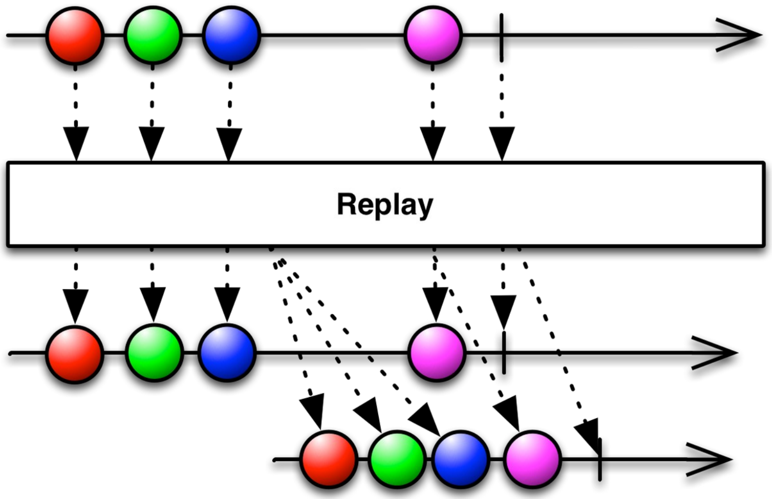
Complete simple application + flatMap

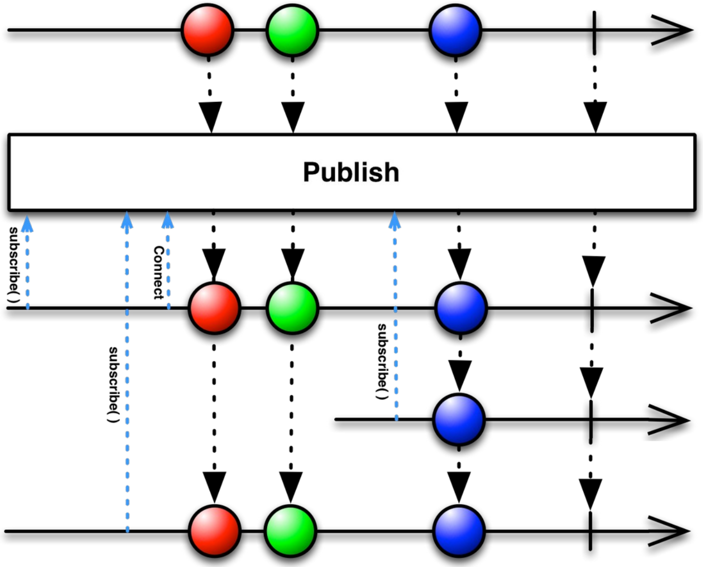
```
public class Server {
	private Flowable<CharBuffer> chat ;
	public void server() throws Exception {
		var loop = new MainLoop();
		chat = loop.accept(ServerSocketChannel.open(new InetSocketAddress(12345));)
			.flatMap(conn -> loop.read(conn)
				.observeOn(computation())
				.lift(new LineSplitOperator())
			)
			.map(bb -> StandardCharsets.UTF_8.decode(bb));
		chat.subscribe(s -> System.out.println(s));
	}
}
```

Keep a reference to the merged stream

Multiple subscriptions

- We can have multiple subscribers to the same stream
  - A <u>cold stream</u> will restart and replay from scratch
  - A <u>hot stream</u> will simply forward new messages to the new subscription
- Can autoConnect() on first subscription
- replay(n) combines both





Complete chat application

```
public class Server {
        private Flowable<CharBuffer> chat ;
        public void server() throws Exception {
                var loop = new MainLoop();
                chat = loop.accept(ServerSocketC                                     ress(12345));)
                        .flatMap(conn -> {
                                loop.write(chat
                                        .map(s -> StandardCharsets.UTF_8.encode(s.duplicate())), conn);
                                return loop.read(conn)
                                        .observeOn(computation())
                                        .lift(new LineSplitOperator());
                        })
                        .map(bb -> StandardCharsets.UTF_8.decod
                        .publish().autoConnect();
                chat.subscribe();
        }
}
```

Implicitly subscribes to the chat stream

Buffer will be used more than once

Allow multiple subscriptions

Do we need this? Yes!

Complete chat application

```
public class Server {
    private Flowable<CharBuffer> chat ;
    public void server() throws Exception {
        var loop = new MainLoop();
        chat = loop.accept(ServerSocketChannel.op                    345));)
            .flatMap(conn -> {
                loop.write(chat.onBackpressureDrop()
                    .map(s -> StandardCharsets.UTF_8.encode(s.duplicate()), conn);
                return loop.read(conn)
                    .observeOn(computation())
                    .lift(new LineSplitOperator());
            })
            .map(bb -> StandardCharsets.UTF_8.decode(bb)
            .publish().autoConnect());
        chat.subscribe();
    }
}
```

Drop separately for each destination

References

- ReactiveX operators: https://reactivex.io/documentation/operators.html
- Tomasz Nurkiewicz and Ben Christensen. *Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications.* O'Reilly, 2017.
  - Chaps. 4-5

Image sources: Microsot Office Clipart; Flaticon.com; RxJava documentation; and product logos.