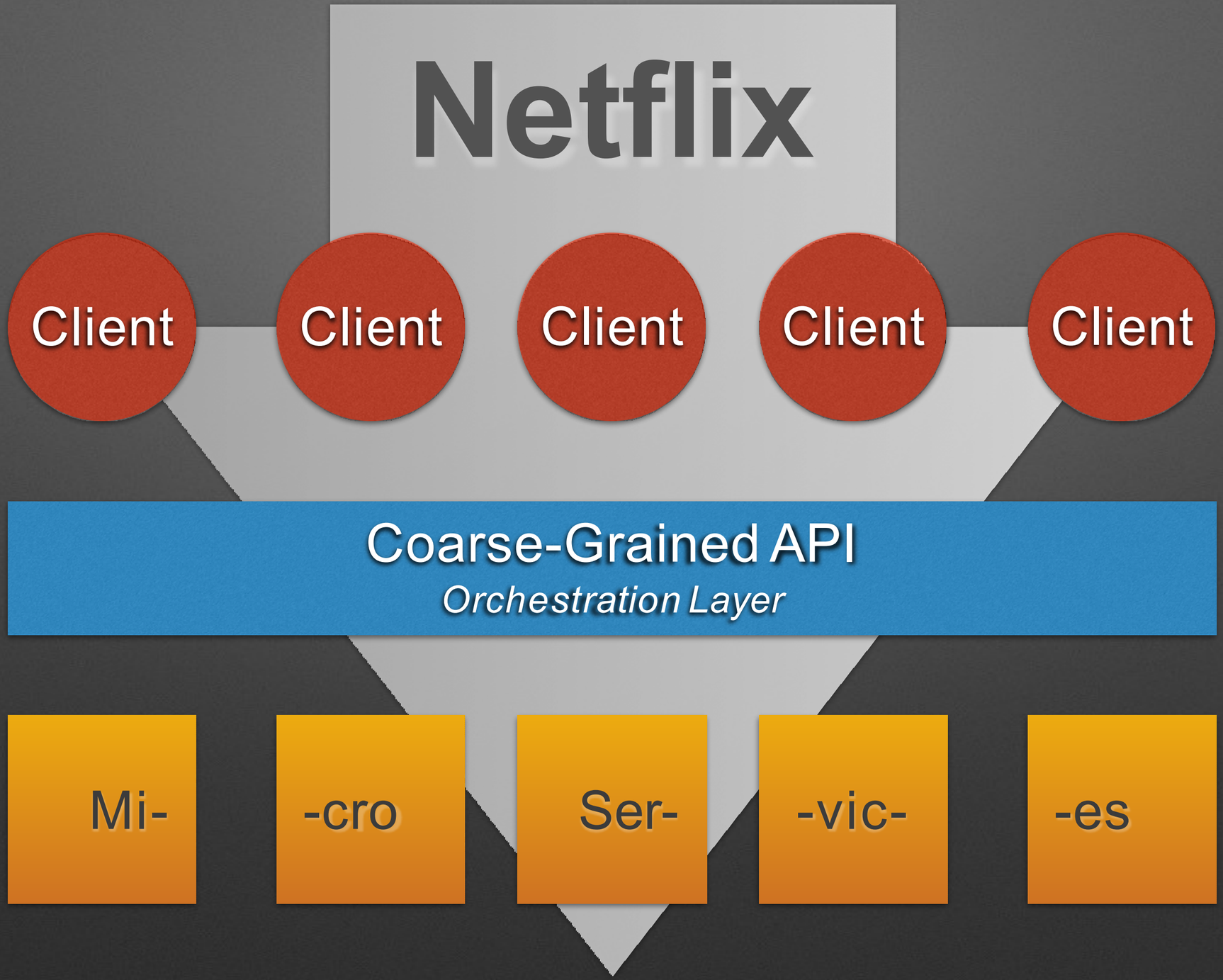
Building Scalable Stateless Applications with

RxJava

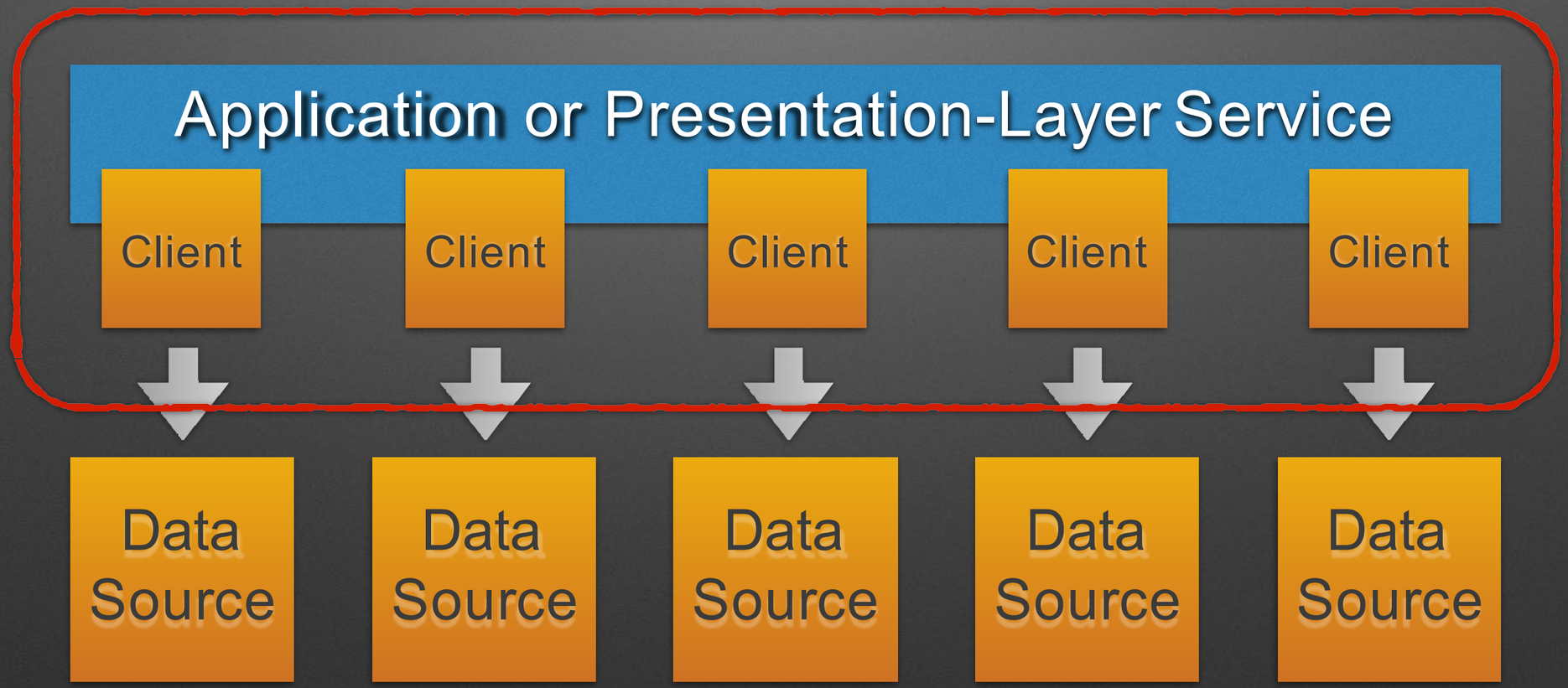
RxJava

- Encapsulates data sequences: `Observable`
- Provides composable operations on them
- Abstracts away threading and synch

- Port from Microsoft .NET Reactive Extensions
- Java 6+, Groovy, Clojure, JRuby, Kotlin, and Scala; Android-compatible


Netflix
Client
Client
Client
Client
Client
Coarse-Grained API
Orchestration Layer
Mi-
-cro
Ser-
-vic-
-es

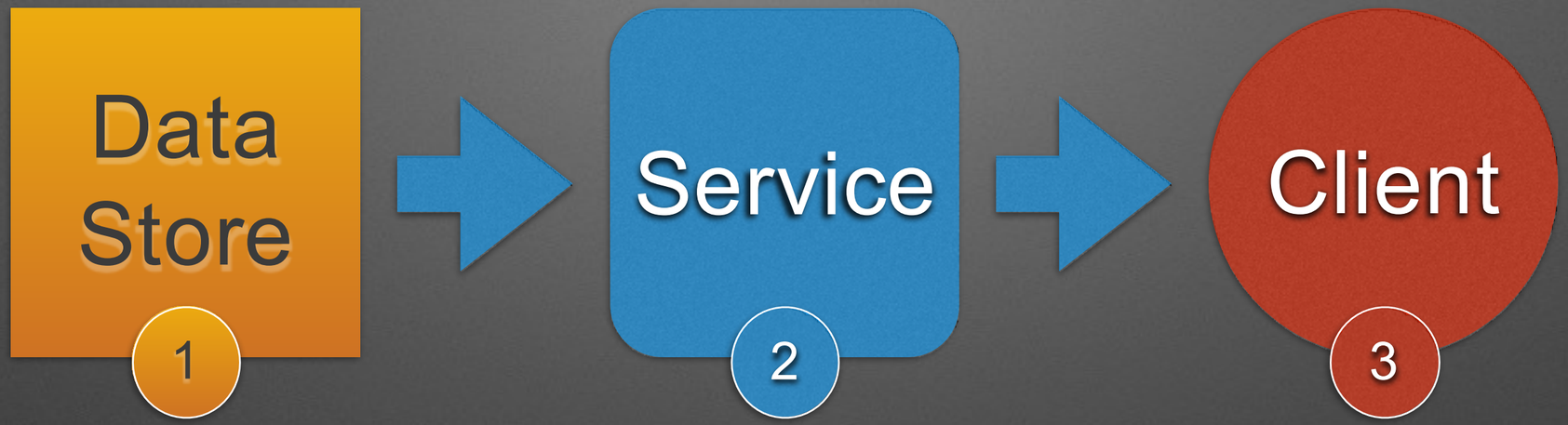
Applicability



3 Problems

1. Streaming queries
2. Rate-limited APIs
   (with retries)
3. Using `Futures`

Streaming Queries



1. Query lots of data from NoSQL store
2. Enrich, filter, and score on the fly
3. Deliver “best” results to client

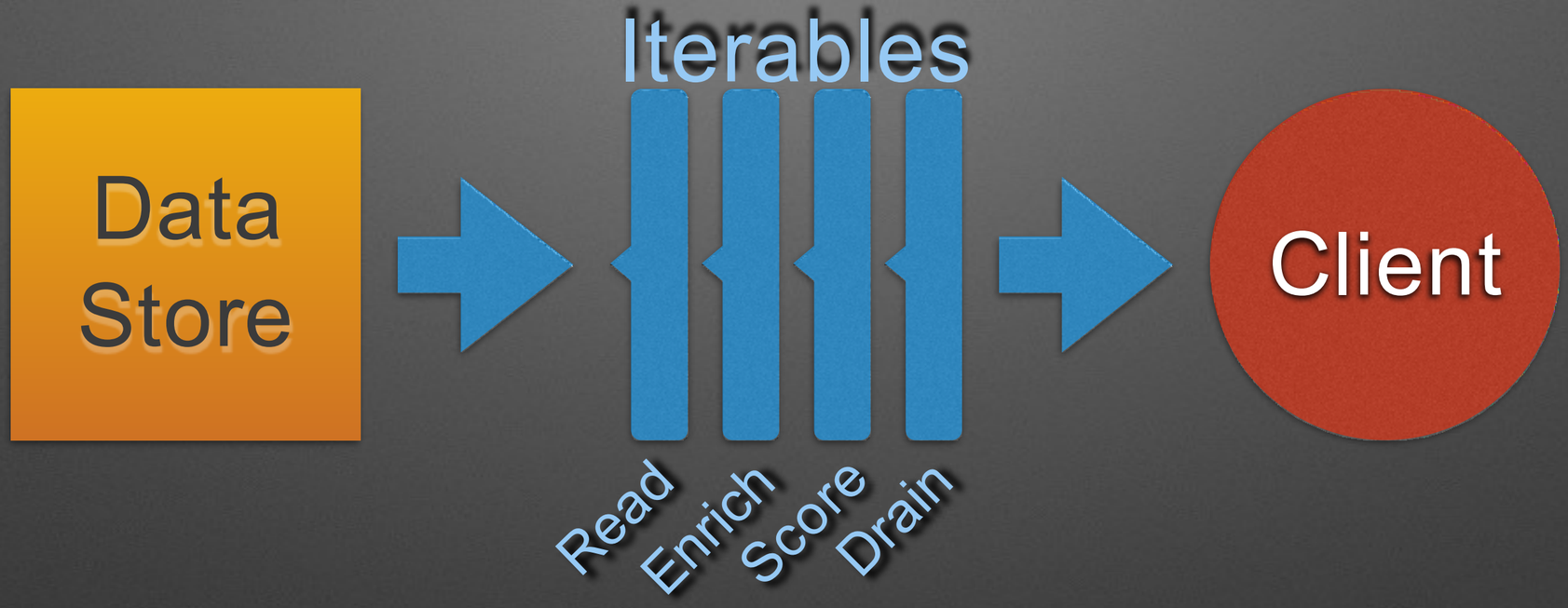
Streaming Queries

Too Much Latency

Streaming Queries

More Efficient

Streaming Queries



- *Iterator* pattern for fast start and lower footprint
- Composed as *Decorators* for flexibility
- **Limitation**: Doesn't abstract timing, concurrency

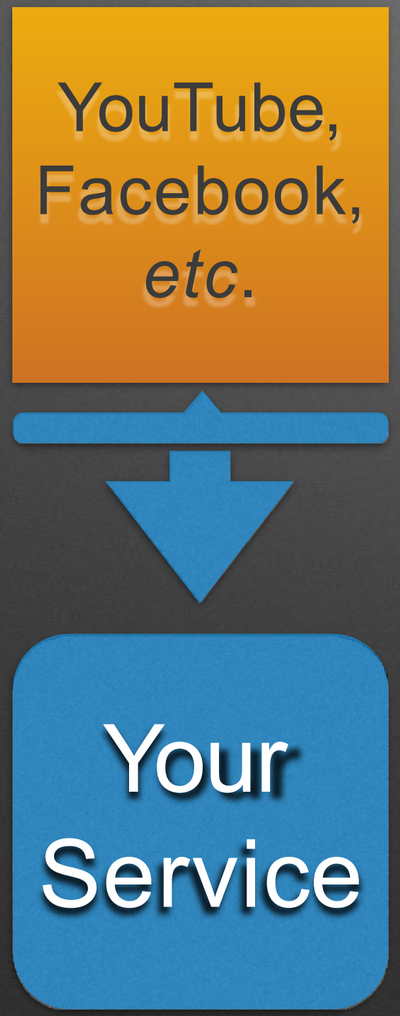
Rate-Limited APIs

Scenario: Ingest data from public API—they will refuse rapid requests!

Rate-Limited APIs

Insight: The module responsible
for *what* data is available
is also responsible
for *when* data is available

Rate-Limited APIs



Solution: *Facade* API with augmented *Visitors* to encapsulate timing and retries

```
interface DataVisitor<T>
extends java.io.Closeable {
  void accept(T data);
}
```

Limitation: Scheduling doesn't generalize

Using Futures

```
javax.ws.rs.client.Invocation
---------------------------------
Response invoke()

Future<Response> submit()
```

- API and implementation cluttered with concurrency variants
- Caller responsible for concurrency, but has least information
- What can you do with a `Future` besides block?

Using Futures

What about Guava's ListenableFuture?

```
ListeningExecutorService service =
  MoreExecutors.listeningDecorator(Executors.newFixedThreadPool(10))
  ;
ListenableFuture<Explosion> explosion = service.submit(new Callable<Explosion>() {
  public Explosion call() {
    return pushBigRedButton();
  }
});
Futures.addCallback(explosion, new FutureCallback<Explosion>() {
  public void onSuccess(Explosion explosion) {
    walkAwayFrom(explosion);
  }
  public void onFailure(Throwable thrown) {
    battleArchNemesis();
  }
});
```

- Requires access to ExecutorService!
- Still hard to compose
- Doesn't generalize to multiple results

Work

Interactions should:

- Support multiple values
- Be efficient & incremental
- Encapsulate timing
- Compose operations

RxJava: `Observable`

- **Like `Iterable`,** but offers fluent chaining operations
- **Like `Stream`,** but supports async termination and error handling

Iterable Example

```
try {
   for (E elem : elements) {
      onNext(elem);
   }
   onCompleted();
} catch (Throwable ex) {
   onError(ex);
}
```

Iterable Example

```
try {
  elements.forEach(elem ->
    onNext(elem)
  );
  onCompleted();
} catch (Throwable ex) {
  onError(ex);
}
```

First-class
Visitor
(Consumer)

Stream Example

```
try {
   elements.parallelStream()
      .filter(condition)
      .map(transformation)
      .forEach(elem -> onNext(elem));
   onCompleted();
} catch (Throwable ex) {
   onError(ex);
}
```

Parallelize operations

Still serial

Observable Example

```
elements
    .filter(condition)
    .map(transformation)
    .subscribe(
      elem -> onNext(elem),
      ex ->onError(ex),  ()
      -> onCompleted());
```

Fully async'able

Creating an Observable

```
import rx.*;

Observable<String> o = Observable.just(
  "a", "b", "c");

Iterable<String> it = ImmutableList.of(
  "a", "b", "c");
Observable<String> o = Observable.from(it);

Future<String> f = exec.submit(myTask);
Observable<String> o = Observable.from(
  f,
  Schedulers.from(exec));
```

Example: JDBC

```
public finalclass ObservableDB {
    private final DataSource db;

    public ObservableDB(final DataSource source) {
        this.db = Objects.requireNonNull(source);
    }

    /** Each emitted List represents a row in the ResultSet. */
    public Observable<List<Object>> select(
            final String sql,
            final Iterable<?> params) {
        return Observable.create(new Observable.OnSubscribe<List<Object>>() {
            @Override
            public void call(final Subscriber<? super List<Object>> sub) {
               // ...
            }
        });
    }
}
```

Example: JDBC

```
@Override
public void call(final Subscriber<? super List<Object>> sub) {
   try {
        try (Connection cx = db.getConnection();
             PreparedStatement stmt = cx.prepareStatement(sql)) {
            int i = 0; for (final Object p : params) { stmt.setObject(i++, p); }
            try (ResultSet results = stmt.executeQuery()) {
               while (results.next() && !sub.isUnsubscribed()) {
                    final Lists<Object> row = new ArrayList<>();
                    for (int col = 0; col < results.getMetaData().getColumnCount(); ++col) {
                        row.add(results.getObject(col));
                    }
                    sub.onNext(row);
                }
            }
        }
        if (!sub.isUnsubscribed()) {
            sub.onCompleted();
        }
    } catch (final Exception ex) {
        if (!sub.isUnsubscribed()) {
            sub.onError(ex);
        }
    }
}
```

Call *n* times

Call one or the other, once

Serialize

Example: JDBC

```
public                          void
        printProductCatalog(
        final DataSource source,
        final boolean awesome) {
    ObservableDB db = new ObservableDB(source);
    Subscription subscription = db.select(
            "SELECT * FROM products WHERE isAwesome=?",
            Collections.singleton(awesome))
        // Func1<List<Object>, Product> :
        .map(unmarshallRowIntoProduct())
        // Func1<Product, Observable<ProductWithPrice>> :
        .flatMap(remoteLookupPriceForProduct(this.priceService))
        .take(NUM_PAGES * NUM_PRODUCTS_PER_PAGE)
        .window(NUM_PAGES)
        .subscribe(new Observer<Observable<ProductWithPrice>>() {
            ...
        });
    // Some time later, if we change our minds:
    //subscription.unsubscribe();
}
```

```
interface PriceService {
  Observable<ProductWithPrice>
    getPrice(Product p);
}
```

Example: JDBC

```
.subscribe(new Observer<Observable<ProductWithPrice>>() {
    private final AtomicInteger pageNumber = new AtomicInteger(1);

    @Override
    public void onNext(final Observable<ProductWithPrice> page) {
        System.out.println("Page " + pageNumber.getAndIncrement());
        page.forEach(new Action1<ProductWithPrice>() {
            @Override
            public void call(final ProductWithPrice product) {
                System.out.println("Product:" + product);
            }
        });
    }

    @Override
    public void onError(final Throwable ex) {
        System.err.println("This is how you handle errors? Srsly?");
    }

    @Override
    public void onCompleted() {
        System.out.println("Copyright 2014 ACME Catalog Company");
    }
});
```

Operations

Useful stuff, built in

Content Filtering

- `Observable<T> filter(Func1<T, Boolean> predicate)`
- `Observable<T> skip(int num)`
- `Observable<T> take(int num)`
- `Observable<T> takeLast(int num)`
- `Observable<T> elementAt(int index)`
- `Observable<T> distinct()`
- `Observable<T> distinctUntilChanged()`

Time Filtering

- `Observable<T> throttleFirst(long duration, TimeUnit unit)`
- `Observable<T> throttleLast(long duration, TimeUnit unit)`
- `Observable<T> timeout(long duration, TimeUnit unit)`

Transformation

- `Observable<R> map(Func1<T, R> func)`
- `Observable<R> flatMap(Func1<T, Observable<R>> func)`
- `Observable<R> cast(Class<R> klass)`
- `Observable<GroupedObservable<K, T>> groupBy(Func1<T, K> keySelector)`

Concurrency

Encapsulated, so usually you don’t care

Concurrency

1. Single-threaded and synchronous by default:
   RxJava doesn't magically create new threads for you

2. When creating an `Observable`, invoke `Subscriber`
   from any thread you like (or use Actors, *etc.*)

3. Derive new `Observables` by binding subscription
   and/or observation to `Schedulers`

Rescheduling

```
Observable<T> rescheduled = obs
  .subscribeOn(mySubScheduler)
  .observeOn(myObsScheduler);
```

What's a `Scheduler`?

Policy for time-sharing among Workers

- **Worker**: Serial collection of scheduled Actions
- **Action**: Callable unit of work, e.g. `Observable`'s subscription or notification

Built-in `Schedulers`

- **Immediate**—`Schedulers.immediate()`
  - Run without scheduling in calling thread
- **Trampoline**—`Schedulers.trampoline()`
  - Enqueue Actions in thread-local priority queue
- **Computation**—`Schedulers.computation()`
  - Enqueue in event loop with as many threads as cores

Built-in `Schedulers`

- **New Thread**—`Schedulers.newThread()`
  - Each `Worker` is a single-thread `ExecutorService`
- **I/O**—`Schedulers.io()`
  - Mostly like New Thread, but with some pooling
- **Executor Service**—`Schedulers.from(ExecutorService)`
  - Bind new `Scheduler` to arbitrary `ExecutorService`, with external serialization of `Actions`. Observations may hop threads!

Bring Data to Calling Thread

Avoid whenever possible

```
myObservable.toBlocking()
```

Operations:

- `first()`
- `last()`
- `toFuture()`
- `toIterable()`

Lab : TweetRx