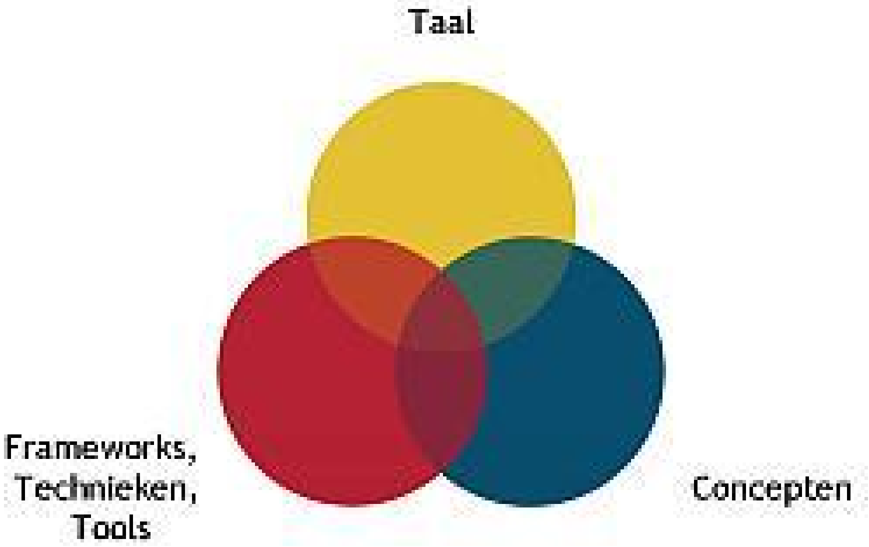
Reactive Extensions

rwolter@sogyo.nl
kvdvlist@sogyo.nl

About Sogyo

- Founded in 1995
- Office at “Landgoed Sandwijck” @ De Bilt
- ~ 80 employees
- Fascinated by *software innovation*: design, development and integration of software.



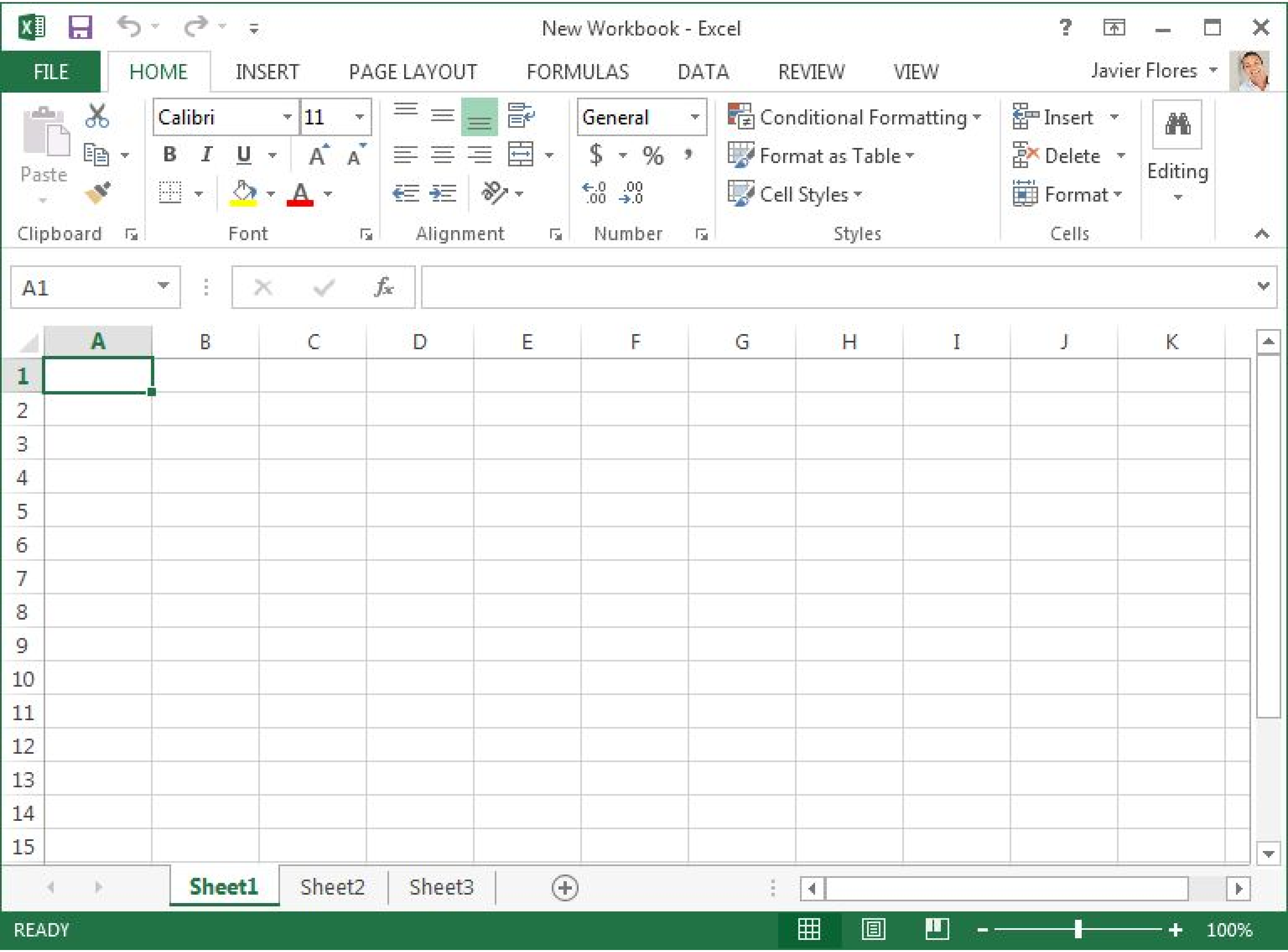
Reactive Extensions

What is it?

Reactive Extensions

*ReactiveX is a **library** for **composing asynchronous** and **event-based** programs by using **observable sequences**. It extends the **observer pattern** to support sequences of data and/or events and **adds operators** that allow you to **compose** sequences together **declaratively** while abstracting away concerns about things like low-level **threading**, **synchronization**, **thread-safety**, **concurrent data structures**, and **nonblocking I/O**.*

- ReactiveX.io


New Workbook - Excel
FILE
HOME
INSERT
PAGE LAYOUT
FORMULAS
DATA
REVIEW
VIEW
Javier Flores
Paste
Calibri
11
B
I
U
General
$
%
Conditional Formatting
Format as Table
Cell Styles
Insert
Delete
Format
Editing
Clipboard
Font
Alignment
Number
Styles
Cells
A1
fx
A
B
C
D
E
F
G
H
I
J
K
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Sheet1
Sheet2
Sheet3
READY
100%

Reactive Extensions

Where did it come from?

Rx origins and influences

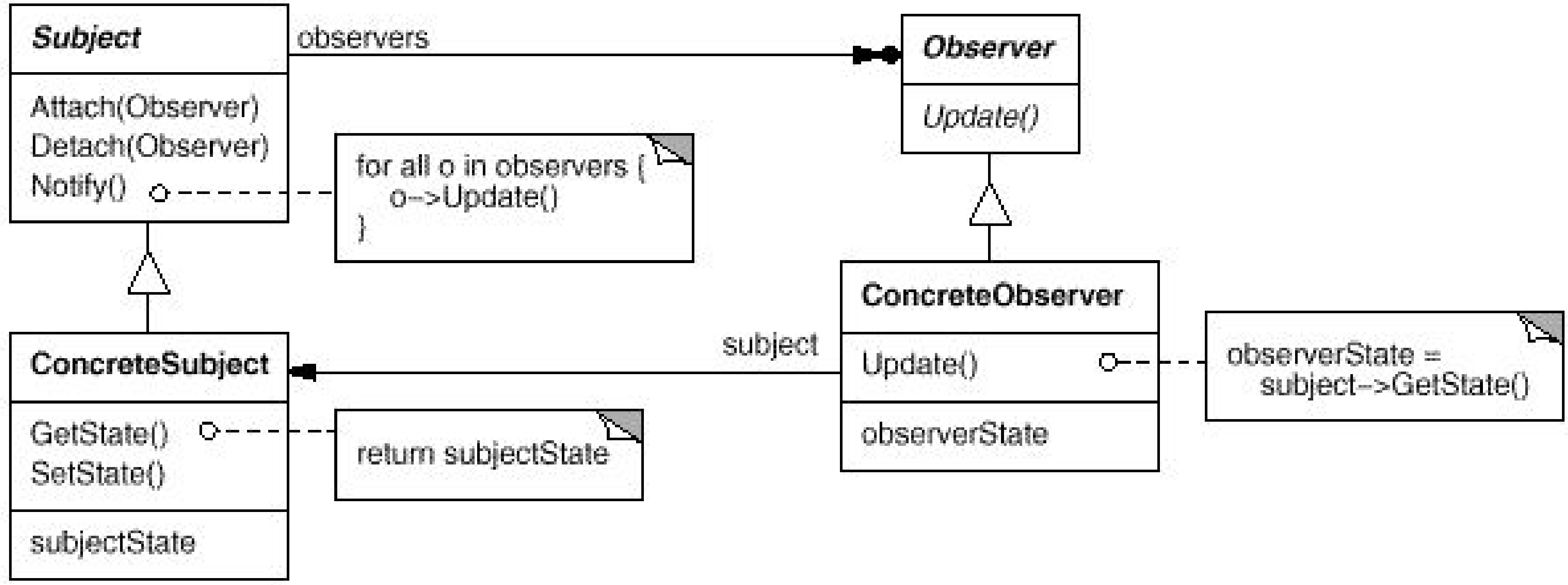
- Observable pattern
- Iterator pattern
- Asynchronous code

Observer pattern

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and update automatically.

- Design Patterns, Gamma et al

Observer pattern



Observer pattern

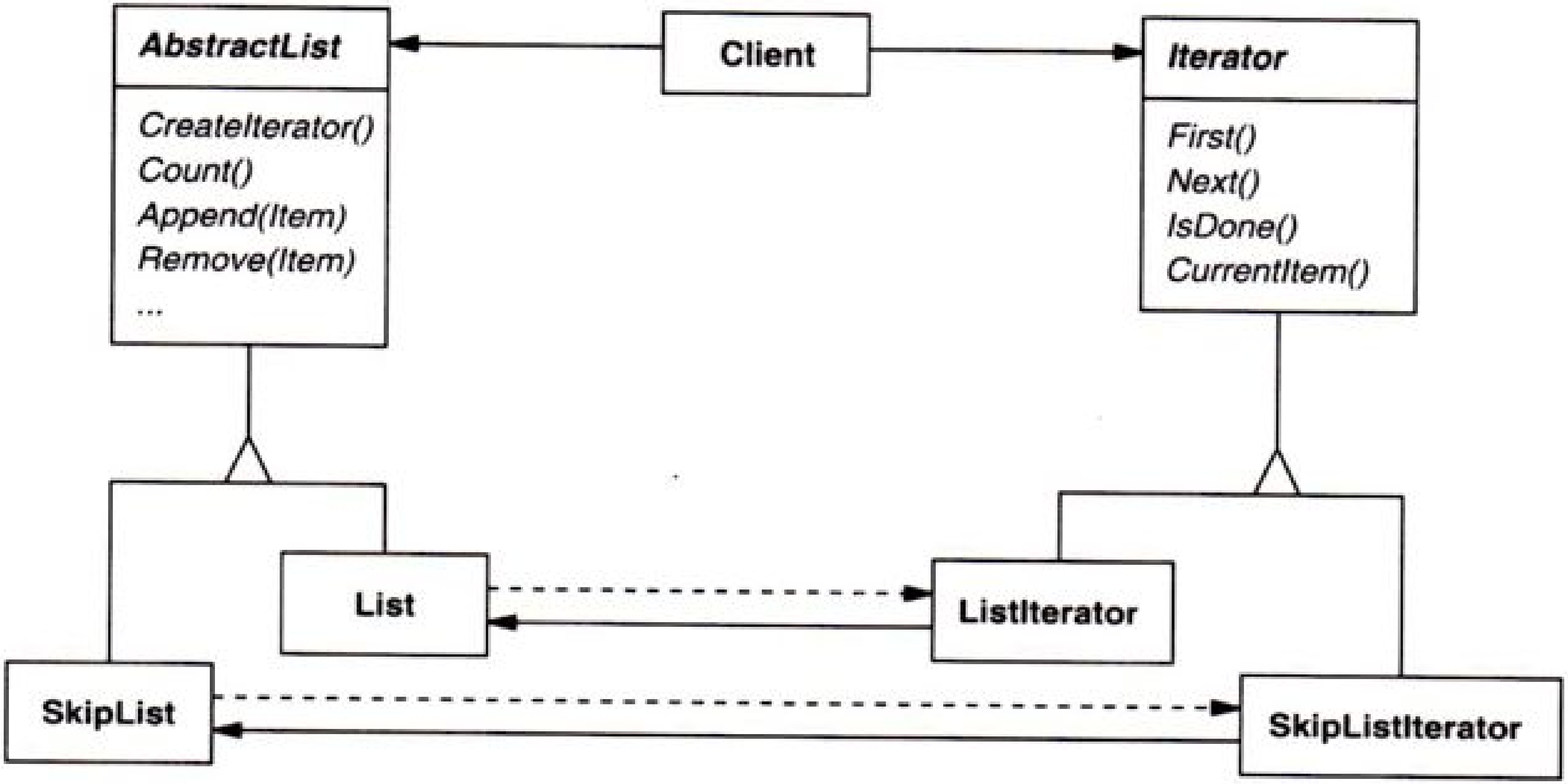
- Loosely coupled, change-driven components
- (un)subscribe at will
- No change semantics (“something changed” instead of “change foo occurred”)
- No distinction between the nature of changes (failures, success, timers etc are all identical)
- No sensible way to terminate (Subject can’t signal completion to observers)

Iterator pattern

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Design Patterns, Gamma et al

Iterator pattern



Iterator + LINQ

- LINQ: Language INtegrated Query)
- Microsoft C# language extension
- Allows manipulation of collections via a single, streaming fluent API.

Iterator pattern - LINQ

```
var results = SomeCollection
    .Where ( c = > c.Foo < 10)
    .Select ( c = > new { c.Foo , c.Bar }) ;

// or as language extension :
var results = from c in SomeCollection
    where c.Foo < 10
    select new { c.Foo , c.Bar };
```

Iterator pattern - LINQ

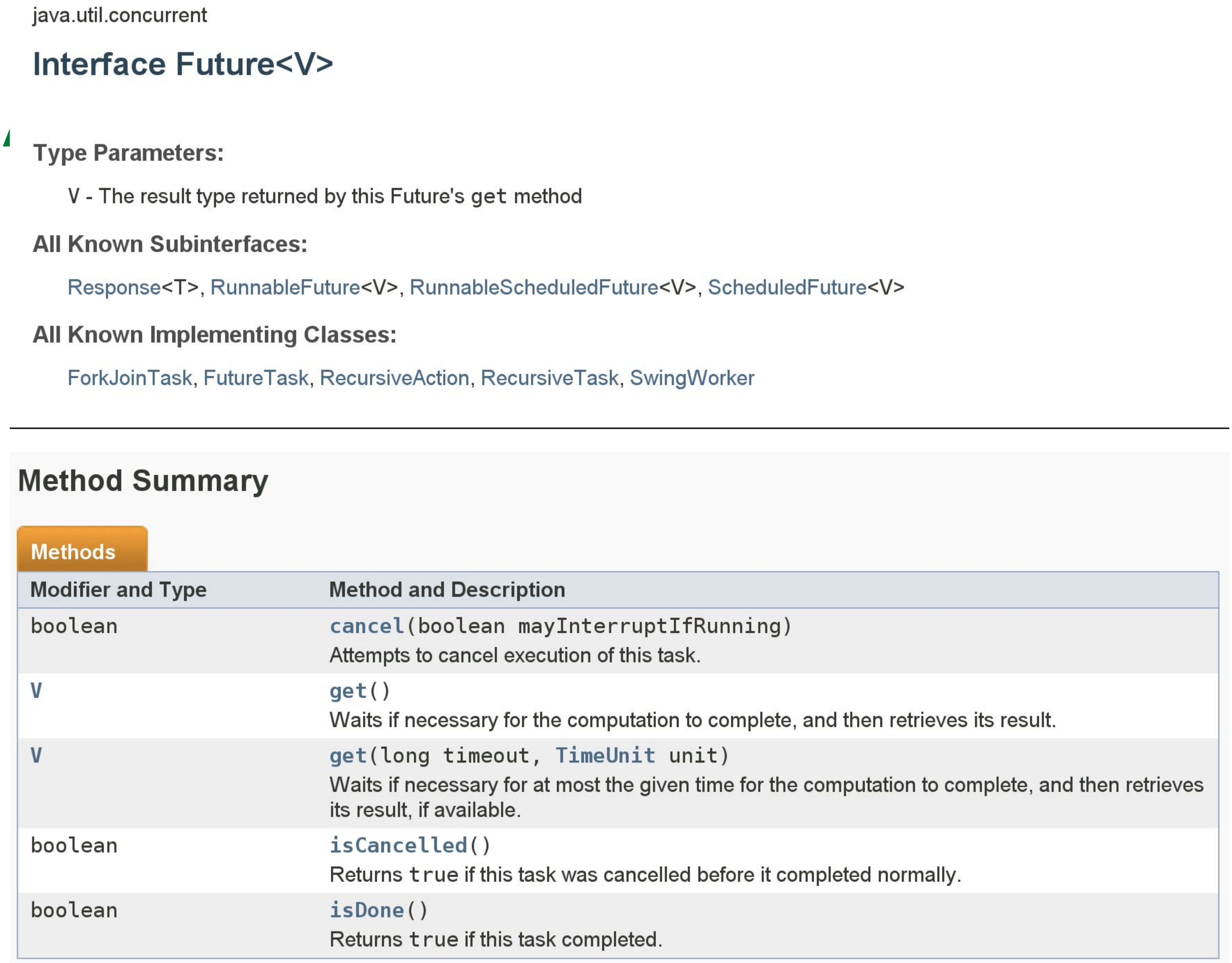
```
List < String > items = new List<string>{"Foo", "Bar"};
var results = from s in items select s;

foreach (var result in results) {
      Console.WriteLine(result) ;
}

items.add ("Baz") ;
foreach (var result in results) {
      Console.WriteLine(result) ;
}
```

Iterable + LINQ

- Generic way to express collections, or series of data
- LINQ: Delayed execution
- LINQ: Query always reflects the state of the underlying structure
- Time is an issue (future data can’t exist in an iterable collection)
- Doesn’t cope well with endless streams
- Controlled by the consumer of the data (e.g. pull-based)

Async

- Everything is asynchronous
    - File IO
    - Network IO
    - Business processes
    - …
- Problem: how should we deal with this in code?
    - Various push-based solutions are in wide use



java.util.concurrent

Interface Future<V>

Type Parameters:

V - The result type returned by this Future's `get` method

All Known Subinterfaces:

Response<T>, RunnableFuture<V>, RunnableScheduledFuture<V>, ScheduledFuture<V>

All Known Implementing Classes:

ForkJoinTask, FutureTask, RecursiveAction, RecursiveTask, SwingWorker

Method Summary

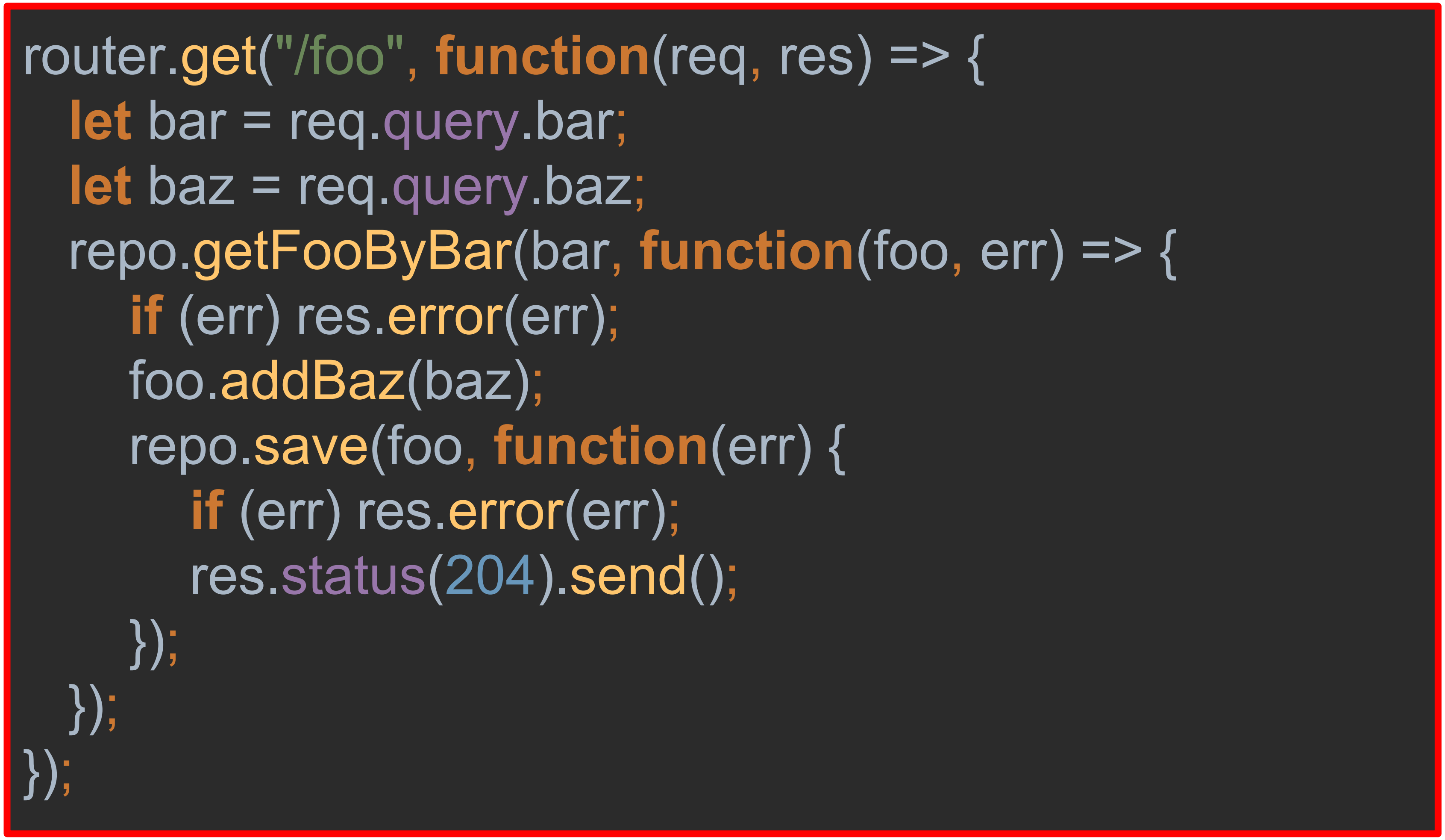
Methods

Modifier and Type	Method and Description
`boolean`	`cancel(boolean mayInterruptIfRunning)` Attempts to cancel execution of this task.
`V`	`get()` Waits if necessary for the computation to complete, and then retrieves its result.
`V`	`get(long timeout, TimeUnit unit)` Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
`boolean`	`isCancelled()` Returns `true` if this task was cancelled before it completed normally.
`boolean`	`isDone()` Returns `true` if this task completed.

Async - Futures

- Futures
  - Annoying API’s
  - Hard to compose
  - Only applicable to a single ‘unit’
    - Future<List<T>>
    - Future<T>
    - etc

Async - Callbacks

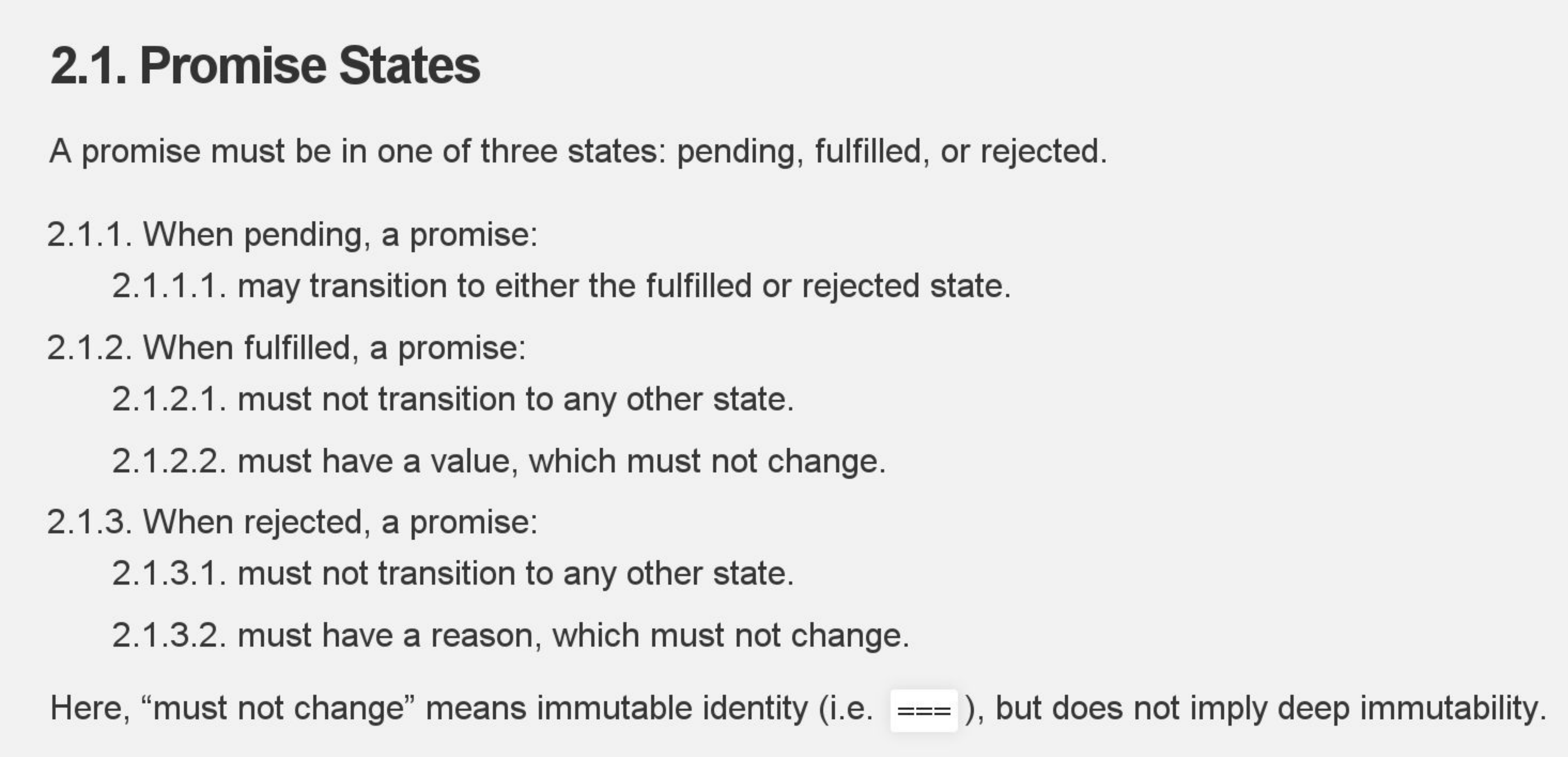


```
router.get("/foo", function(req, res) => {
  let bar = req.query.bar;
  let baz = req.query.baz;
  repo.getFooByBar(bar, function(foo, err) => {
    if (err) res.error(err);
    foo.addBaz(baz);
    repo.save(foo, function(err) {
      if (err) res.error(err);
      res.status(204).send();
    });
  });
});
```

Async - Callbacks

- Callbacks
  - Flexible API’s
  - Unreadable code
  - State is tricky
    - Progress of dispatched task can not be known
    - Multiple results; multiple callback invocations?
    - Closures?

Async - Promises



2.1. Promise States

A promise must be in one of three states: pending, fulfilled, or rejected.

2.1.1. When pending, a promise:

- 2.1.1.1. may transition to either the fulfilled or rejected state.

2.1.2. When fulfilled, a promise:

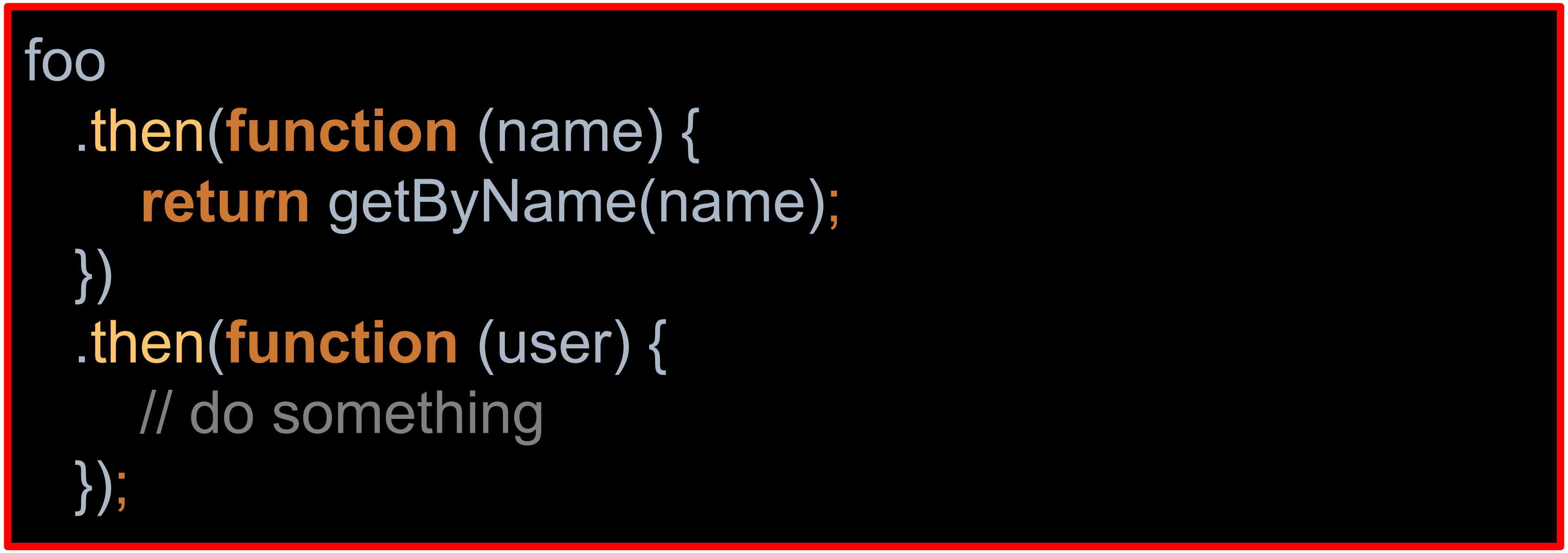
- 2.1.2.1. must not transition to any other state.
- 2.1.2.2. must have a value, which must not change.

2.1.3. When rejected, a promise:

- 2.1.3.1. must not transition to any other state.
- 2.1.3.2. must have a reason, which must not change.

Here, “must not change” means immutable identity (i.e. `===`), but does not imply deep immutability.

Async - Promises



```
foo
   .then(function (name) {
       return getByName(name);
   })
   .then(function (user) {
       // do something
   });
```

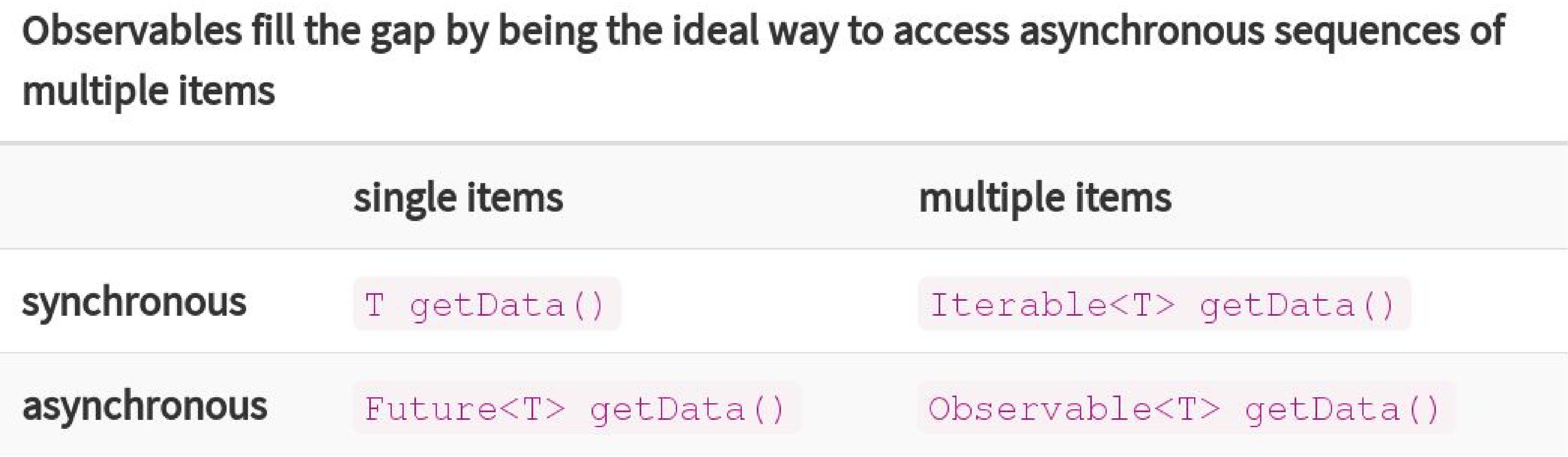
Async - Callbacks

- Promises
  - Async code starts to look like sequential code
  - Clear API
  - Eager execution: no laziness
  - Coordination (like cancellation) of larger, more complex sequences is almost impossible.

Reactive Extensions

The essence of Rx

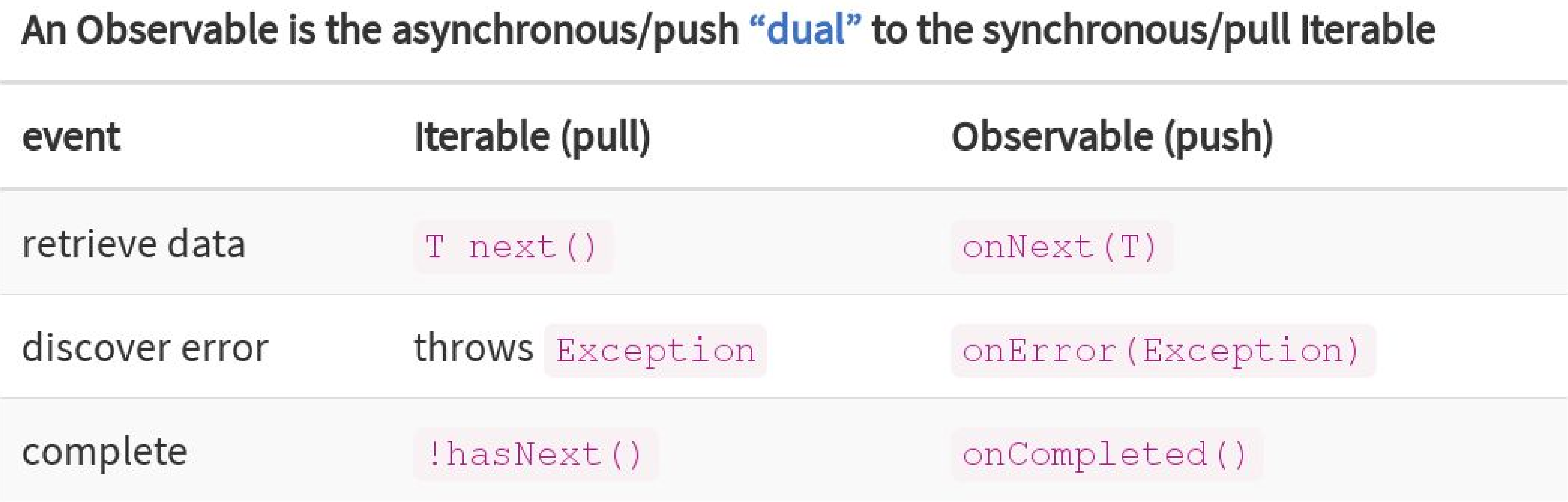
Rx: Unification of collections



Observables fill the gap by being the ideal way to access asynchronous sequences of multiple items

	single items	multiple items
synchronous	`T getData()`	`Iterable<T> getData()`
asynchronous	`Future<T> getData()`	`Observable<T> getData()`

Rx: pull vs push based



An Observable is the asynchronous/push “dual” to the synchronous/pull Iterable

event	Iterable (pull)	Observable (push)
retrieve data	`T next()`	`onNext(T)`
discover error	throws `Exception`	`onError(Exception)`
complete	`!hasNext()`	`onCompleted()`

Example Single

```
app.get('/api/transactions/:id', function(req, res) {
   Rx.Observable.just(req)
       .map(req => req.params.id)
       .map(findById)
       .forEach(t => res.json(t));
});
```

Example Multiple

```
app.get('/api/users/:userId/transactions', function(req,
res) {
   Rx.Observable.just(req)
       .map(req => req.params.userId)
       .flatMap(findByUserId)
       .forEach(t => res.json(t));
});
```

Example Multiple Sources

```
app.get('/api/users/:userId/transactions', function(req, res) {
  const stream = Rx.Observable.just(req)
     .map(req => req.params.userId);
  Rx.Observable.merge(
     stream.flatMap(findByUserId),
     stream.flatMap(otherFindByUserId)
  ).forEach(t => res.json(t));
});
```

Rx: why should I use this?

- Easily composable
- Polyglot (At time of writing there are 13 official implementations for various languages)
- Streams -- everything is a stream

Reactive Extensions

Everything is a stream

Rx: types of data

- A single value
- An array of values
- A promise
- A callback
- An event
- A series of events over time
- ...

Rx: types of data

- A single value: a stream of one value
- An array of values: a stream of values
- A promise: a stream of one value
- A callback: an event and ...
- An event: a stream of one value
- A series of events over time: a stream of values
- ...

Rx: types of data

- Interval timer: stream of ‘ticks’
- Mouse: stream of mouse location events
- Bitcoin ticker: BTC <-> EUR trading values

Everything is a stream

Marble diagrams

… or “how can I read Rx documentation”

Rx: why should I use this?

- Used to visually explain Observables and their transformations
- API documentation can take some time to get used to:

  The FlatMap operator transforms an Observable by applying a function that you specify to each item emitted by the source Observable, where that function returns an Observable that itself emits items. FlatMap then merges the emissions of these resulting Observables, emitting these merged results as its own sequence.

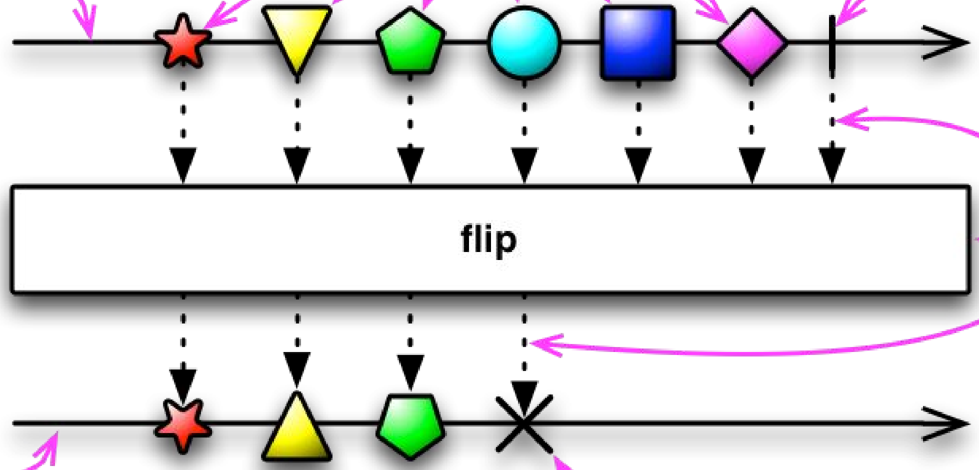
- http://reactivex.io/documentation/operators/flatmap.html

Marble diagrams

This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

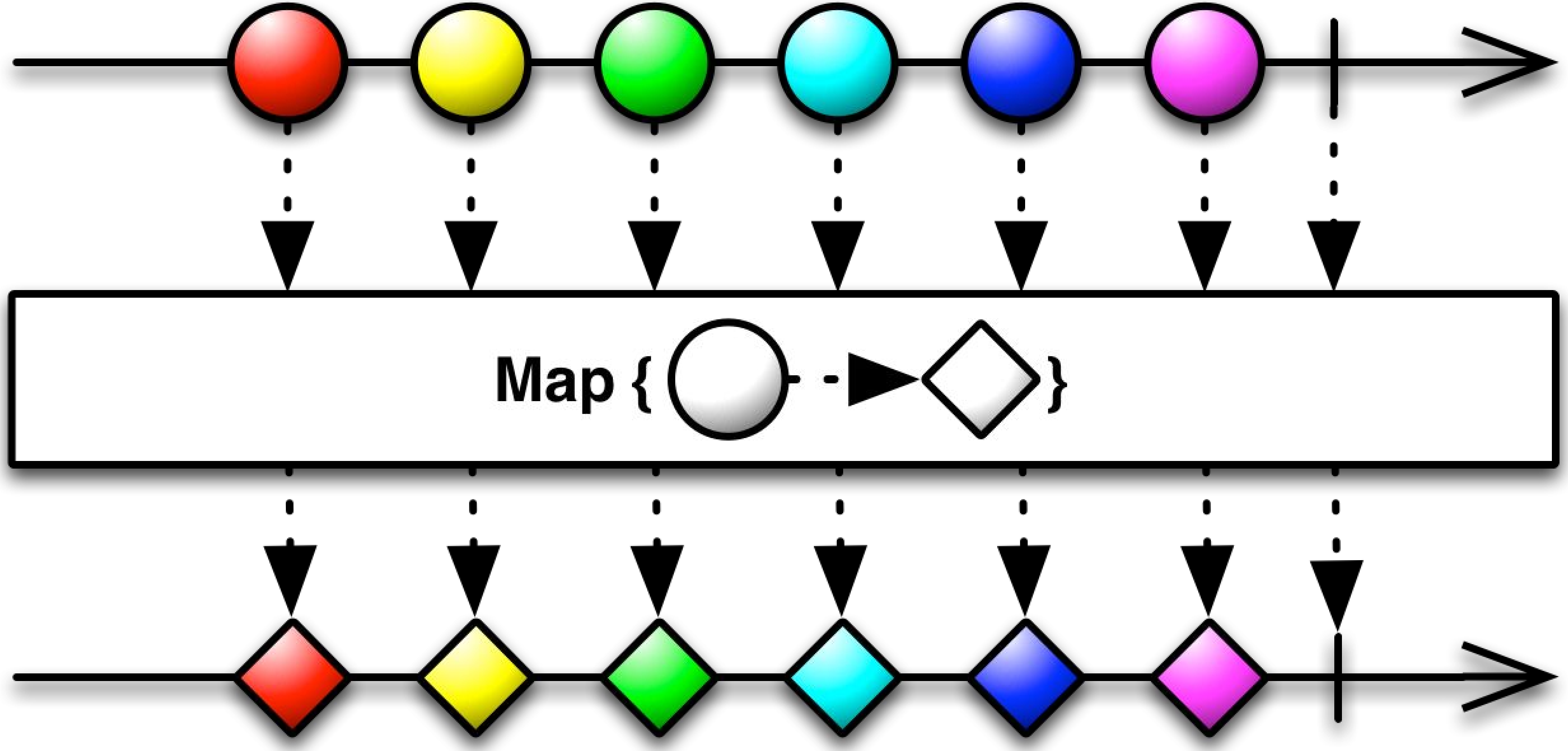


These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

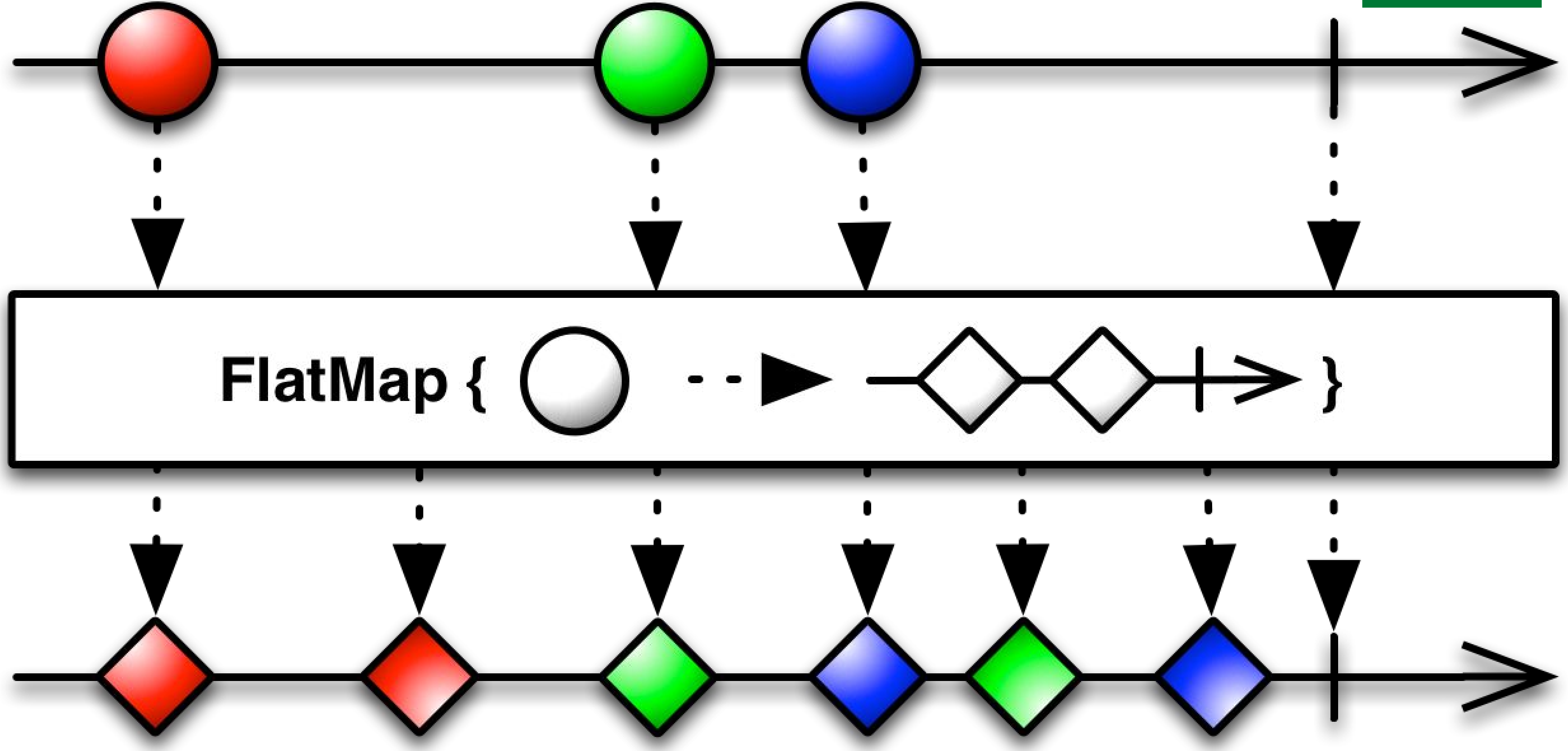
This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Map



Map



Search box

How, when and why to use Rx?

Search box

- Domain:
  - A web page with a search box
  - The search box can query a back-end service
  - The back-end will respond with every that has the same prefix as the search request.
- You are asked to implement the front end code to implement the search box.

Search box

- http://localhost:8000/

Other references

References: interesting links

- Hystrix
- Vert.x
- Fluorine

References: interesting links

- Andre staltz' “Intro you've been missing”
- ReactiveX.io
- Your mouse is a database
- https://github.com/Sogyo/rx-samples
-