OBSERVABLES

JOHAN KUSTERMANS

PLANNING

- Context
- Introduction of Observables
- Components & Observables
- Operators
- Examples

Reactive Extensions

ReactiveX - RX

http://reactivex.io

"An API for asynchronous programming based on Observable streams"

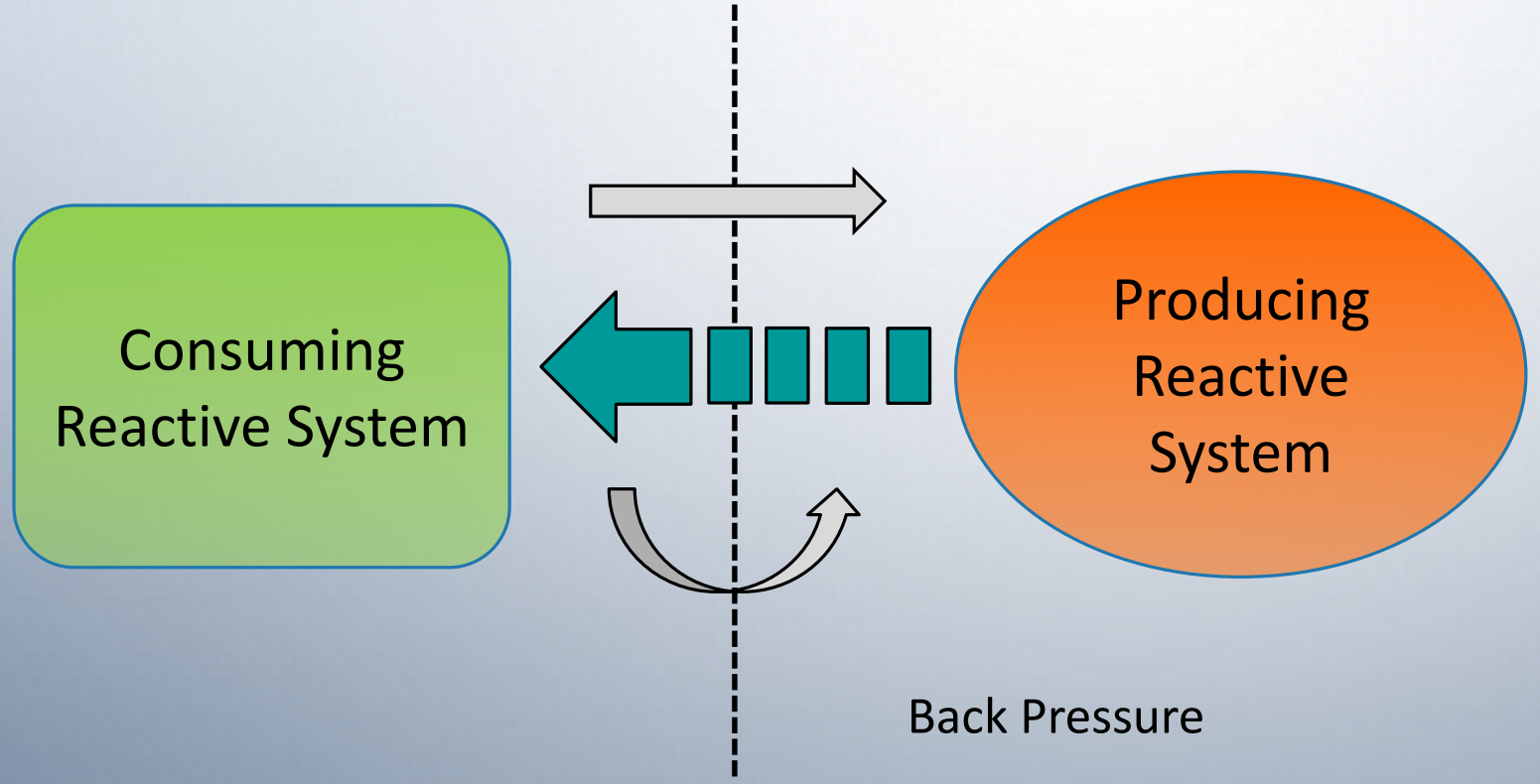
Functional Reactive Programming

Multiple Implementations:

- C# (origins, Erik Meijer)
- Javascript : RxJS 5
- Java : RxJava (1 & 2)

Reactive Streams

“Govern the exchange of stream data accross an aynchronous border”



Implementations

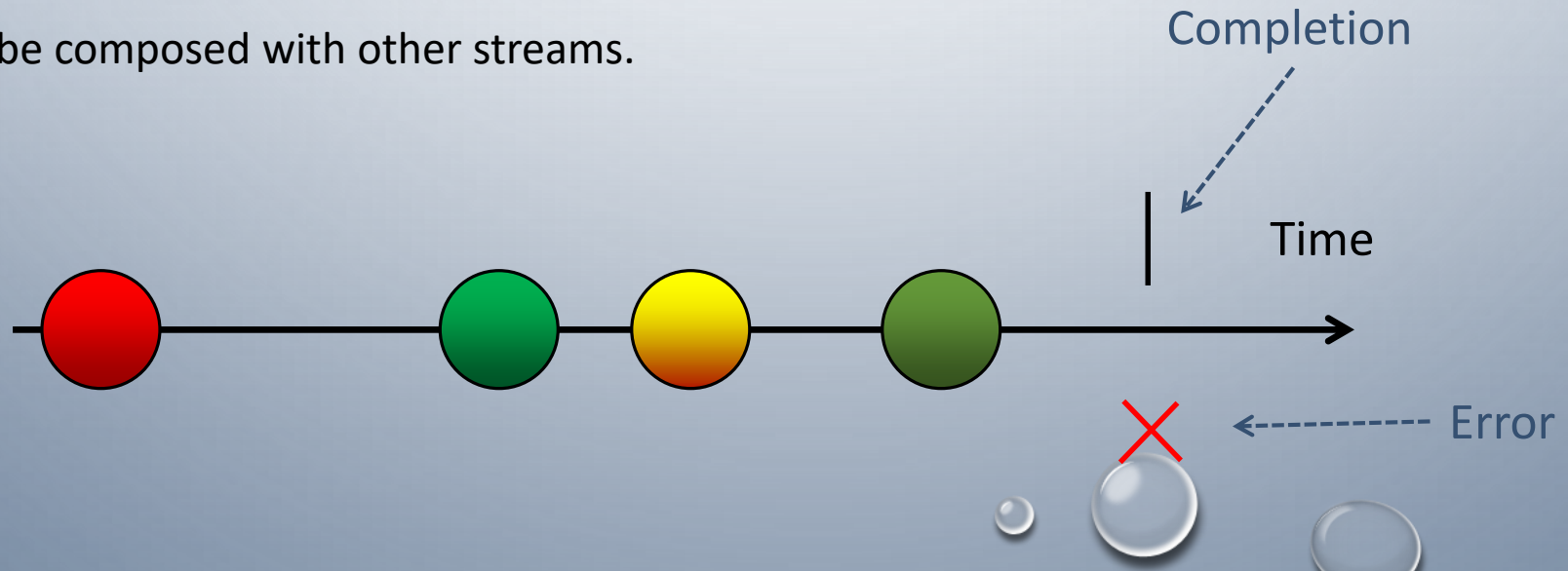
- RxJava
- Spring Reactor -> Spring 5
- Play & Akka Streams

Observables

Observables (RXJS)

- Angular 2 heavily relies on observables: event system, routing, http, forms, QueryList.

- An Observable<T> is a stream of data items (of type T) over time that
  - Can be observed in an asynchronous way
  - Can be transformed into another stream.
    e.g. transform each element in the stream to another type.
  - Can be composed with other streams.

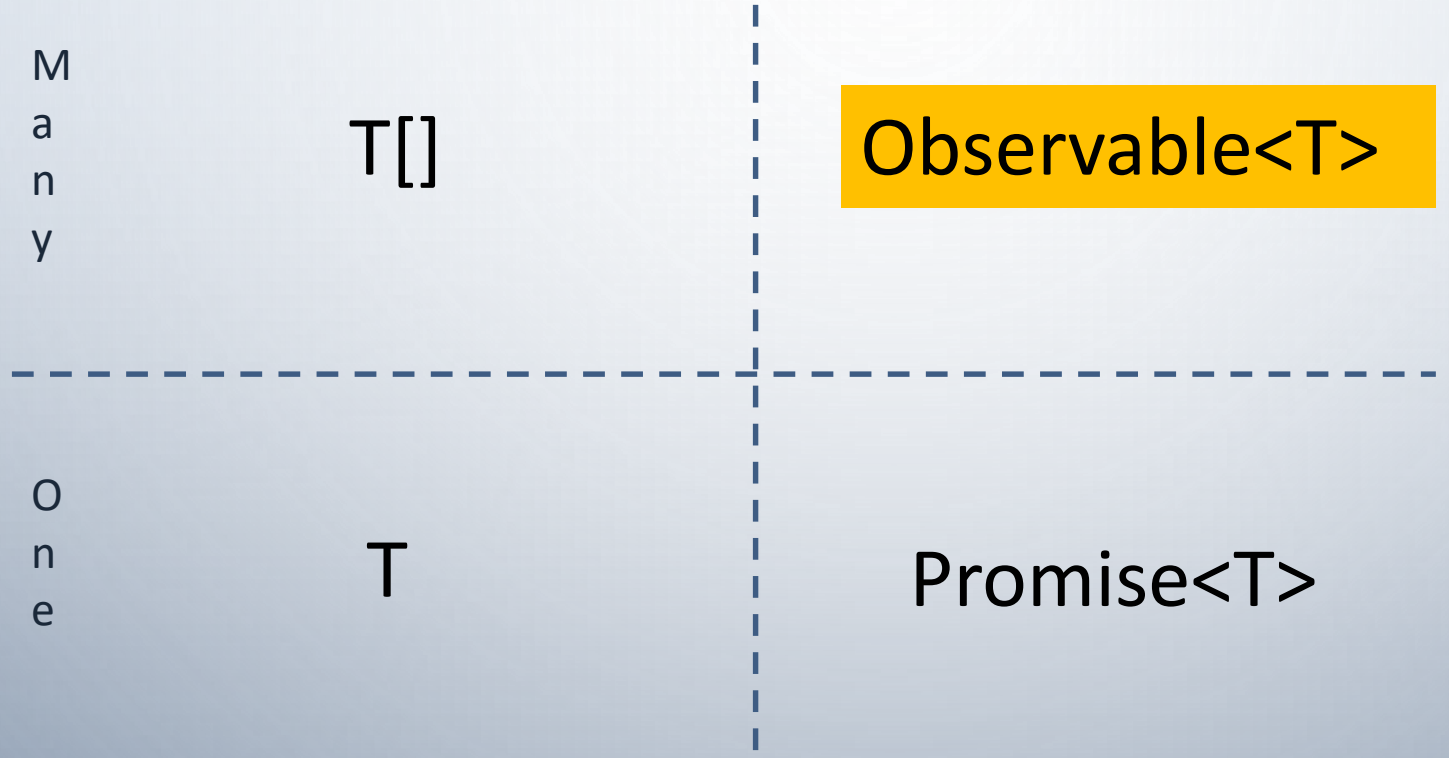


Observables

- Examples
  - Stream of click events from a button.
  - Result of a HTTP call.
  - Stream of data over a web socket.
  - Stream of messages from a queue.

- Typical example
  - Autocomplete in angular 2 (see later)

As a result of a method call



Pull

Synchronous:
Values are immediately available.

Asynchronous:
Values are available in the futurer.

Push

Observable Subscription

Most basic way to consume an observable: via a subscription.

```
Let obs : Observable <T> = … retrieved e.g. from http call

subscription = obs.subscribe(
      item => … handle  item … ,
      error => … handle error …,
      ()        => … handle completion …
)
```

A subscription can be canceled! (e.g. the underlying http request will be canceled)

```
subscription.unsubscribe()
```

Observable Creation Methods

- In a lot of cases, you do not create an observable, but you get one as a starting point.
- You derive an observable by applying an operator on an existing observable.
- You create a Subject using the constructor.

```
Let subject = new Subject<T>();
```

You drive this subject by calling one of the following methods:

```
subject.next(value:T)
subject.error(error:any)
Subject.complete();
```

Special cases:

- BehaviorSubject. This type of Subject remembers the last value, and sends it to a subscriber upon subscription (generalized by ReplaySubject).
- In Angular 2: EventEmitter. This has an “async“ flag.

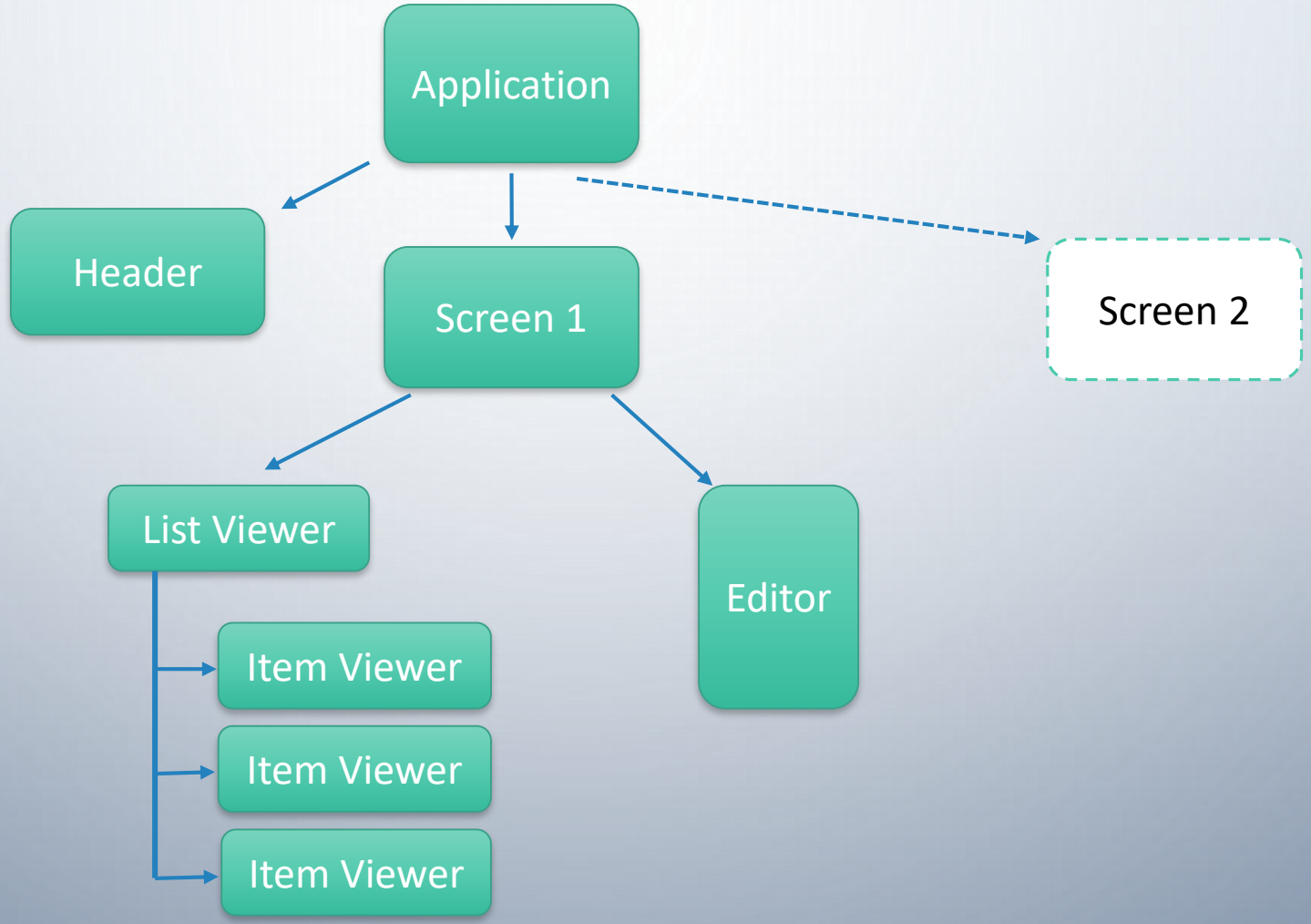
Observable Creation Methods

- Through factory methods:
  - Observable.of
  - Observable.fromPromise
  - Observable.throw
  - Observable.interval

- Observable.create((subscriber:Subscriber) => TeardownLogic)
  Example: source code of xhr_backend

Components

Component Tree



Component

Component : self descriptive unit

- State and behaviour defined in class
- View via HTML Template
- Defines Input & Output API
- Wiring via dependency injection
- Well defined Lifecycle & callbacks: onInit, onDestroy, …

Anatomy of a Component

application-header.ts

```
@Component({
    selector: "application-header"
    templateUrl: "application-header.html"
})
class ApplicationHeaderComponent {

    applicationName : string;

    search() : void {
        … perform search …
    }
}
```

application-header.html

```
<span>{{applicationName}}<span>

<button (click)="search()">Search</button>
```

application.html

```
<application-header></application-header>
<div>
                … application content …
</div>
```

Components

application-header.ts

```
@Component({
    selector: "application-header"
    templateUrl: "application-header.html"
})
class ApplicationHeaderComponent {

    @Input()
    applicationName : string;

}
```

application.ts

```
@Component({
    selector: "application"
    templateUrl: "application.html"
})
class ApplicationComponent {

  name : string = "Angular 2 Workshop";


}
```

application.html

```
<application-header [applicationName]="name" />
<div>
                … application content …
</div>
```

Components & Observables

Outputs

application-header.ts

```
@Component({})
class ApplicationHeaderComponent {

    search(value:string) : void {
        this.onSearch.next(value);
    }

    @Output()
    onSearch : EventEmitter<string>;
```

application-header.html

```
<input type="text" #searchInput>

<i (click)="search(searchInput.value)" />
```

application.html

```
<application-header  (onSearch)="performSearch($event)" />
<div>
                … application content …
</div>
```

Operators

Operator API : Composable & Rich

- Most operators on an observable return a new observable that can be further operated on
  - => This allows for operator chaining!
- API: http://reactivex.io/rxjs/
  Manual: http://reactivex.io/rxjs/manual
  In Depth: http://reactivex.io/documentation/operators

  Transform: map, reduce, …

  Filter: filter, debounce, take, takeWhile, skip, sample, …

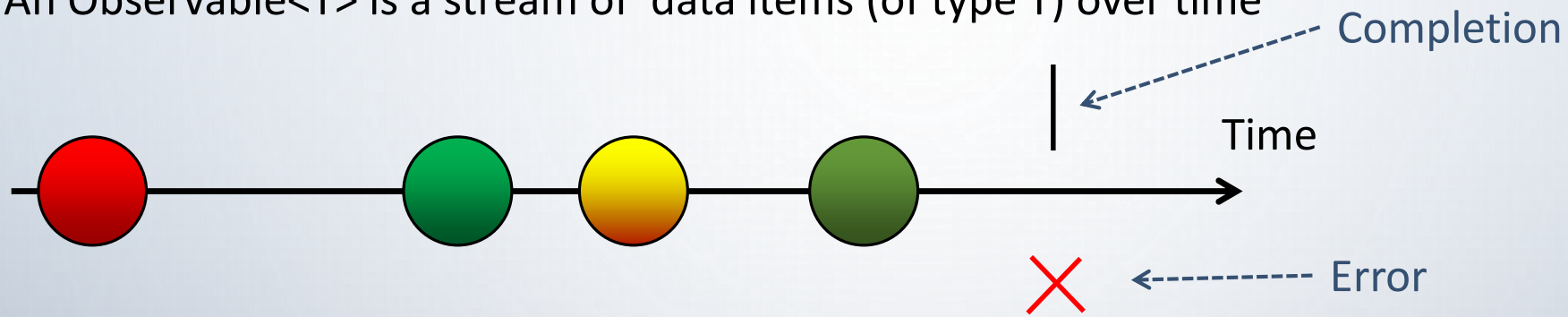
  Combine: merge, concat, combineLatest,
  mergeAll, mergeMap, concatMap, groupBy, switchMap, …

  Error Handling: catch, retry, onErrorResumeNext
- Cold / hot observables

Recapitulation

Recapitulation (RXJS 5)

- An Observable<T> is a stream of data items (of type T) over time



- You consume an observable by subcribing to it:

```
subscription = observable.subscribe(
      item => … handle  item … ,
      error => … handle error …,
      ()       => … handle completion …
)
```

Recapitulation

- An observable can be
  - Unicast : Each subscriber gets it's own execution context.
  - Multicast : Different subscribers share an execution context → Subject

- An operator creates a new observable from an existing one(s)

  Operator chaining provides a fluent API!
  - Instance methods

```
Example: Consider obs : Observable<T> and a function f: T → U
Then obs.map(f) is a new Observable<U>
```

  - Static methods

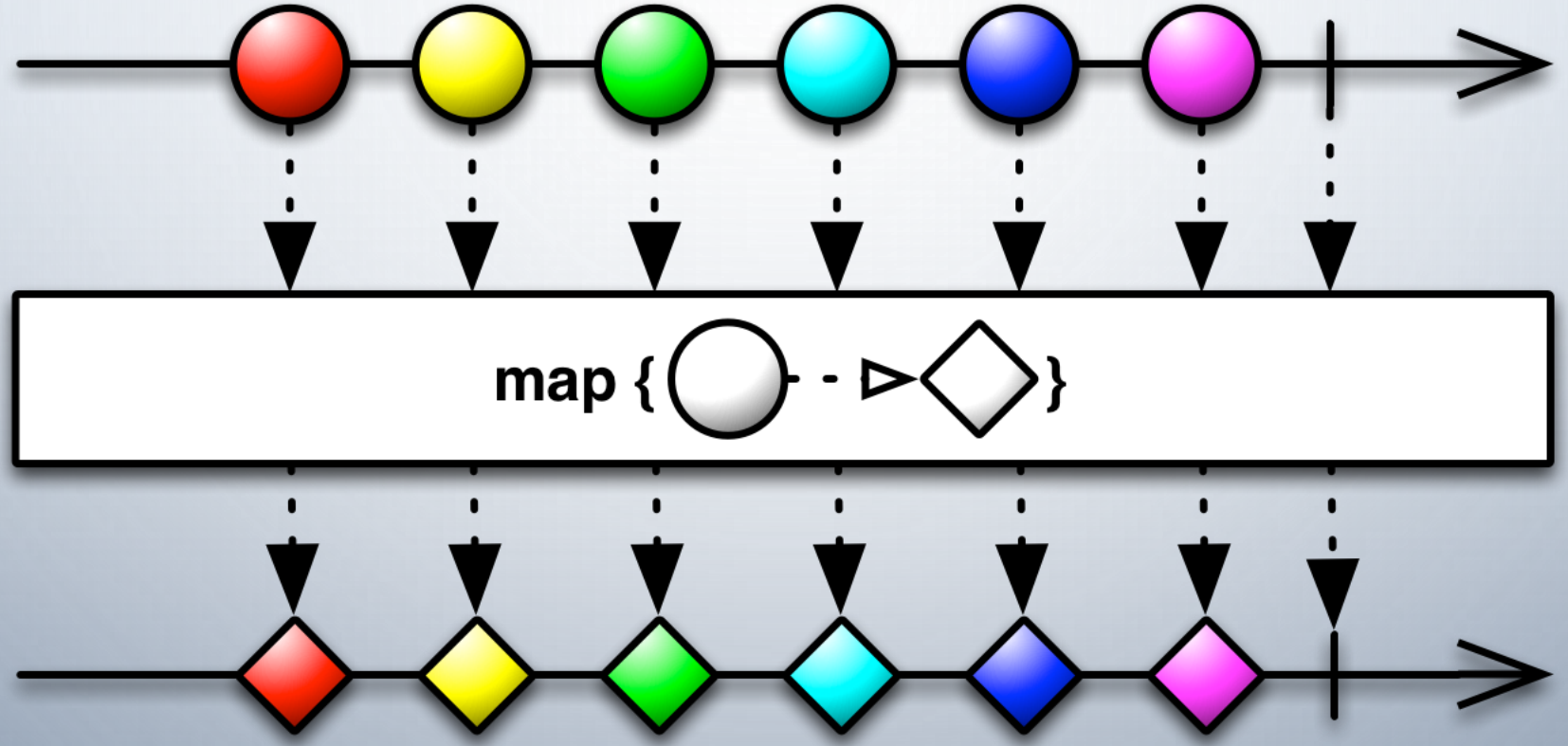
```
Consider obs1, obs2 : Observable<T>
Then Observable.merge(obs1, obs2) is a new Observable<T>
```

Recapitulation

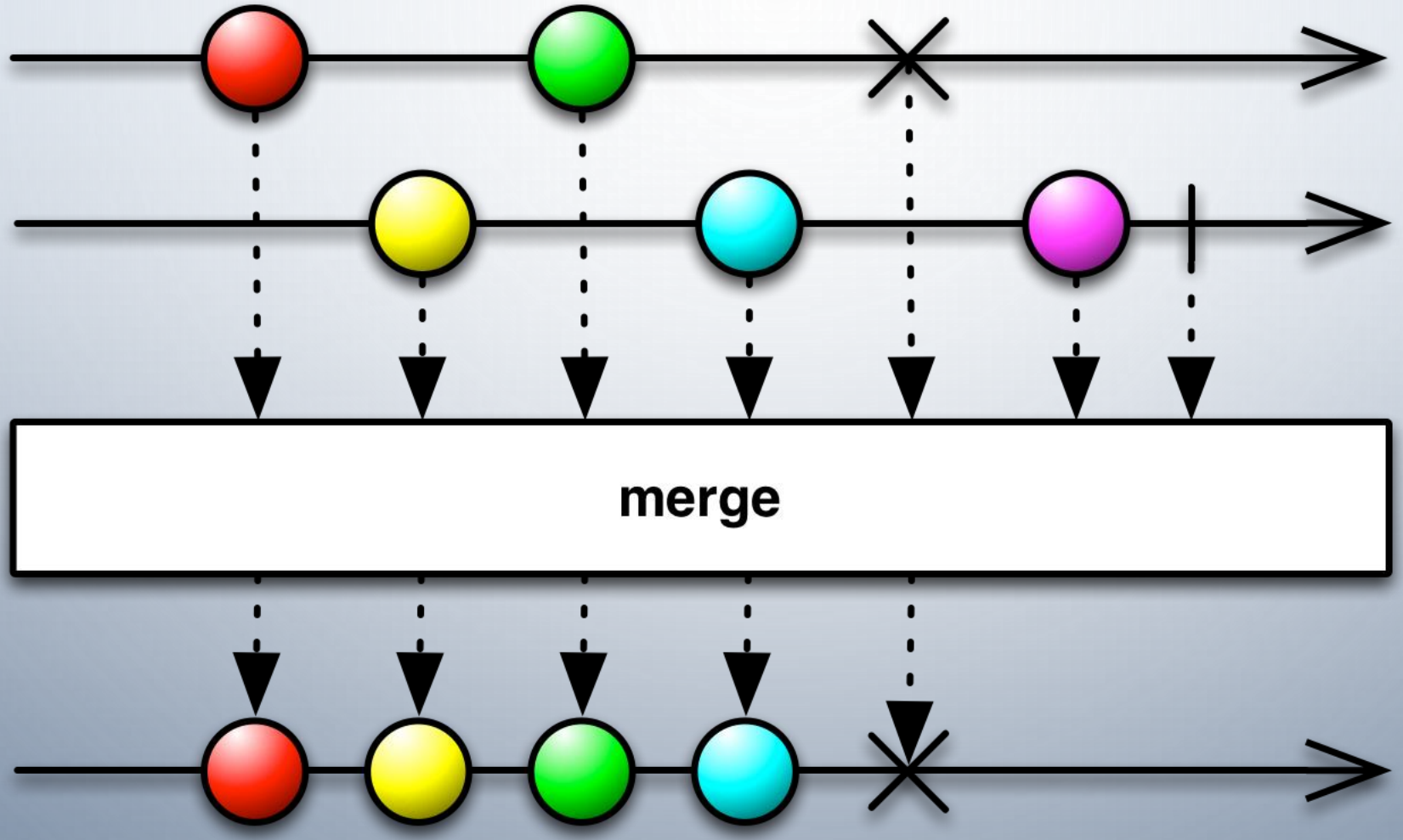
- References:
  - Operator API: http://reactivex.io/rxjs/
  - Operator Manual: http://reactivex.io/rxjs/manual

- First use case in Angular 2:
  An Output of a component is an observable.

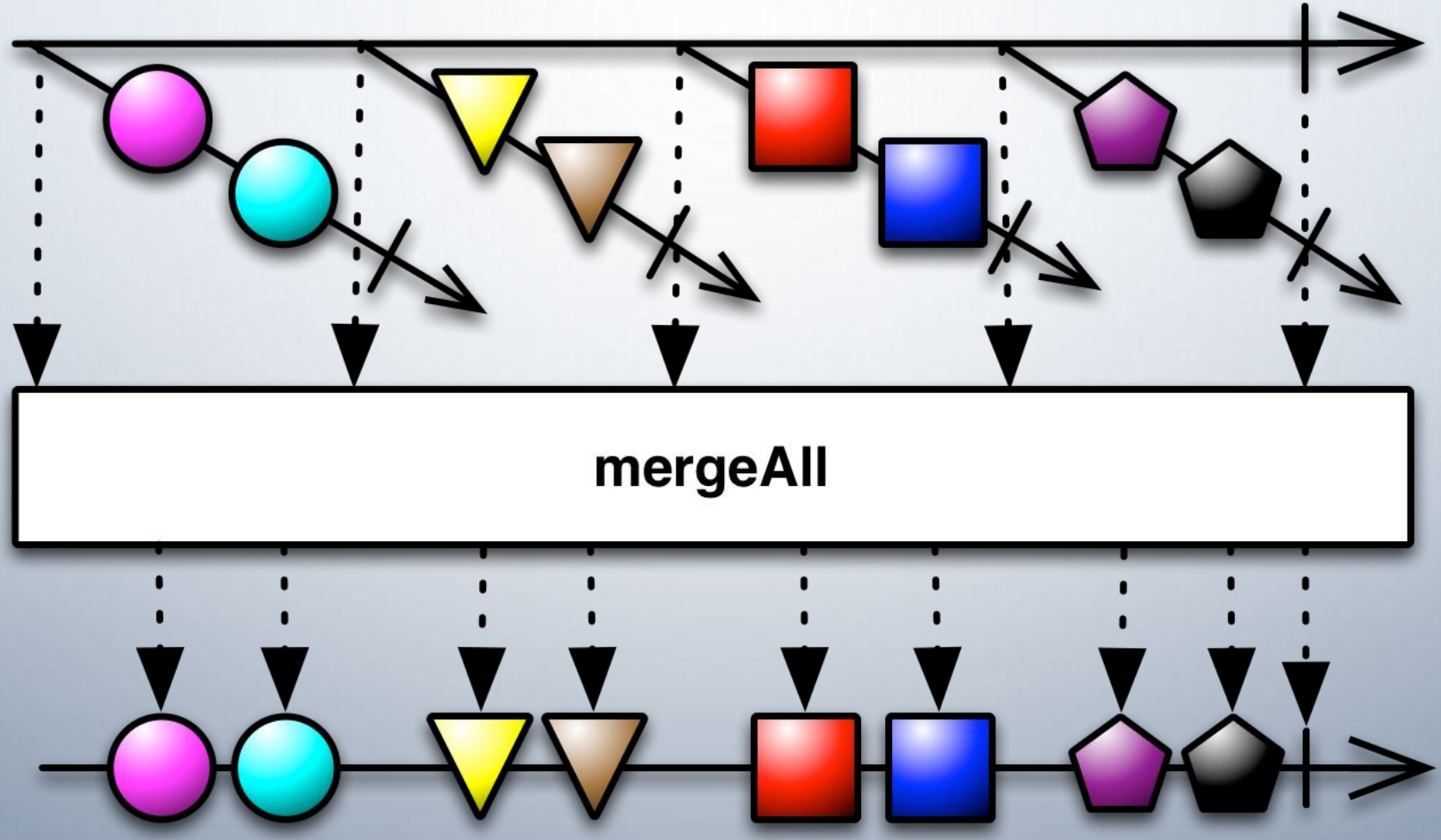
Map



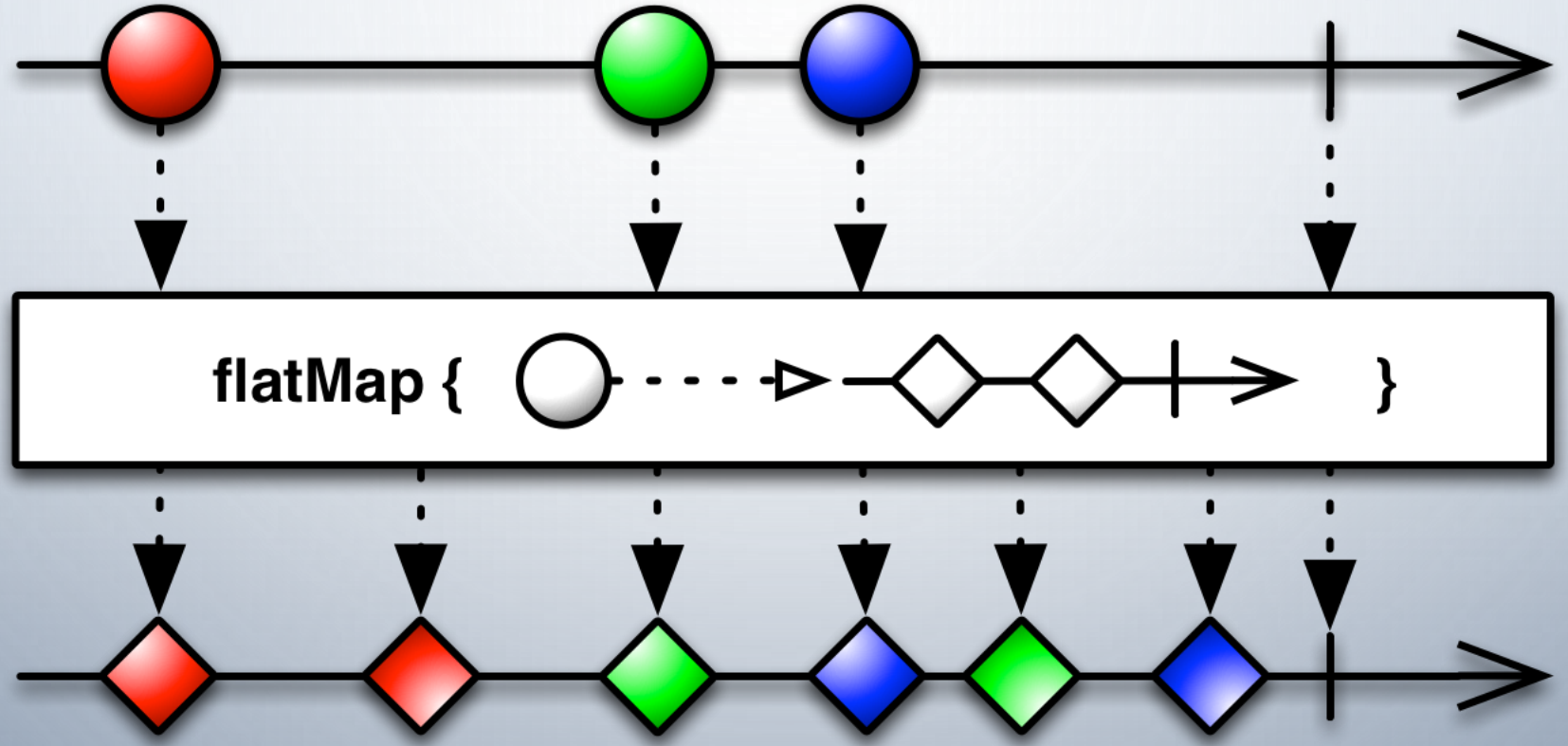
Merge



MergeAll

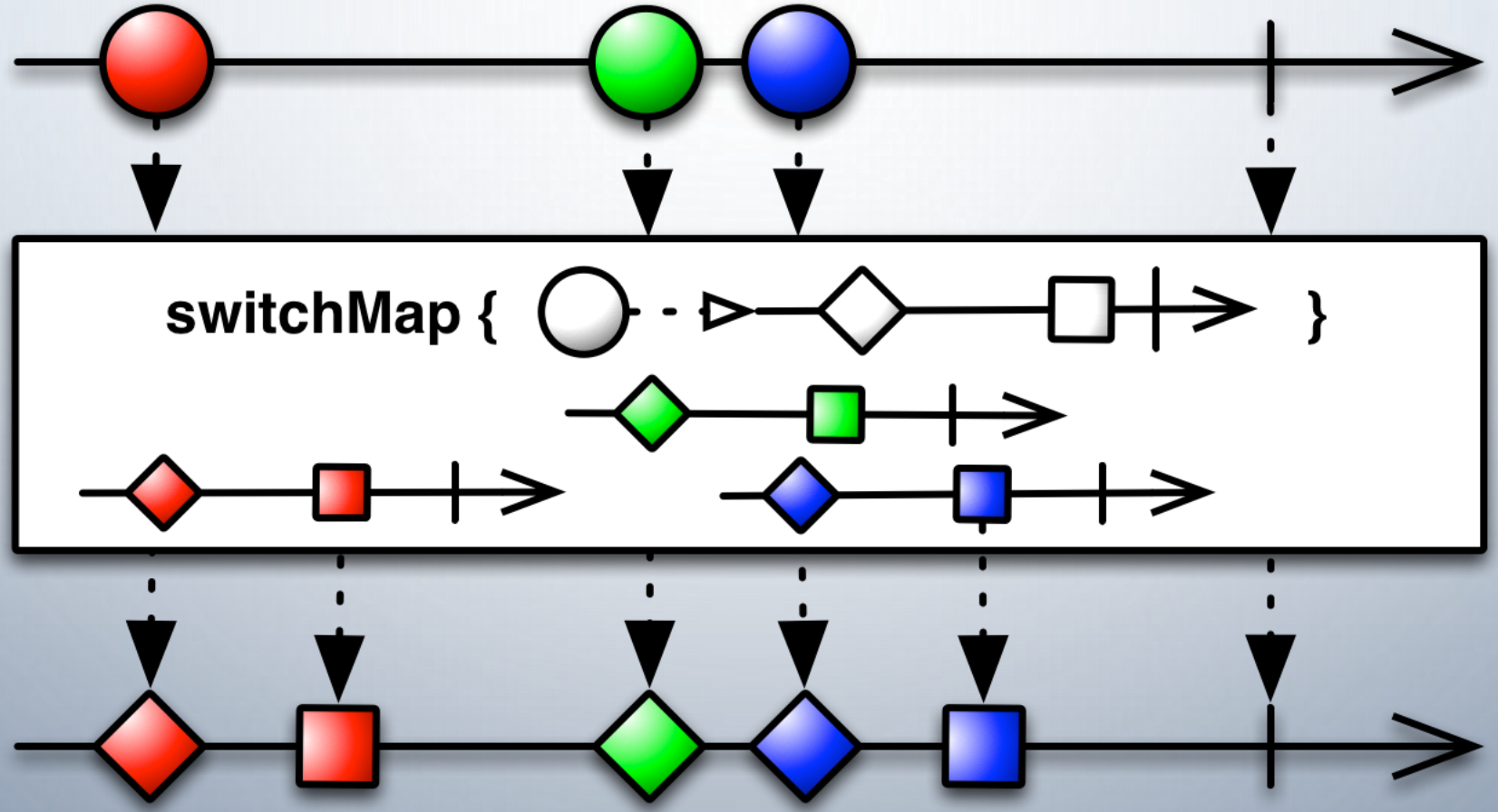


MergeMap (FlatMap)



Autocomplete Example

SwitchMap



Releasing Observable Resources

Disposing Observable Execution

- Some observable executions are disposed of because they are finished.
- Other observable executions are only disposed if all subscriptions have unsubscribed.

In Angular 2 components you can do this in the ngOnDestroy lifecycle callback.

```
@Component({})
class MyComponent implements OnDestroy {


      ngOnDestroy() : void {
            this.mySubscription.unsubscribe();
      }
}
```

Some Use Cases

Some use cases

- Http
- Routing (see next time)
- Forms: observe changes
- Cache: inform components of data updates
- Listen to server updates (Websocket, Stomp)

OBSERVABLES: REFERENCES

- ReactiveX
  - API: http://reactivex.io/rxjs
  - Manual: http://reactivex.io/rxjs/manual
  - In Depth: http://reactivex.io/documentation/operators

- Blogs Thoughtram
  - http://blog.thoughtram.io/angular/2016/01/06/taking-advantage-of-observables-in-angular2.html