Reactive Programming with RxJS

Orjan De Smet
FE Coach @ TVH Parts

https://github.com/orjandesmet/rxjs-course-material/tree/slides

JWORKS

POWERED BY ORDINA

```
let a = 10;

let b = 23;

let c = a + b;

// c = 33

a = 15;

// c = 33

c = a + b;

// c = 38
```

```
let a: number;

let b: number;

let c: number;// =a+b;

calcC() {
  c = a + b;
}

setA(value: number) {
  a = value;
  calcC();
}
```

```
setB(value: number) {
  b = value;
  calcC();
}

setA(10);
setB(23);

// c = 33

setA(15);

// c = 38
```

Callback Hell

```
let a: number;

let b: number;

let c: number;// = e-(a + b)

let d: number;

let e: number; // = [illegible]

calcC() {
  c = e - (a + b);
}

calcE() {
  e = b * d;
  calcC();
}
```

```
setA(value: number) {
  a = value;
  calcC();
}

setB(value: number) {
  b = value;
  calcC();
  calcD();
}

setD(value: number) {
  d = value;
  calcE();
}
```

Reactive Programming to the rescue 

What is Reactive Programming?

What is Reactive Programming?

BehaviorSubject

Callback

switchMap

Events

distinctUntilChanged

tap

Observable

combineLatest

Streams

Subscription

Declarative style

Asynchronous

flatMap

Operators

Propagation of change

What is Reactive Programming?

In computing, reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change. With this paradigm it is possible to express static (e.g., arrays) or dynamic (e.g., event emitters) data streams with ease, and also communicate that an inferred dependency within the associated execution model exists, which facilitates the automatic propagation of the changed data flow.

Source: **Wikipedia**

What is Reactive Programming?

Programming with asynchronous data streams

<<insert mandatory gif>>

What is a Stream

- Events (e.g. click, keyPress, ...)
- Ordered in time
- Can be observed

Wait a minute...

- Ordered in time ➠ synchronous
- Capture events as they occur
- Meanwhile next lines of code can be executed

➡ **Asynchronous**

Data Streams

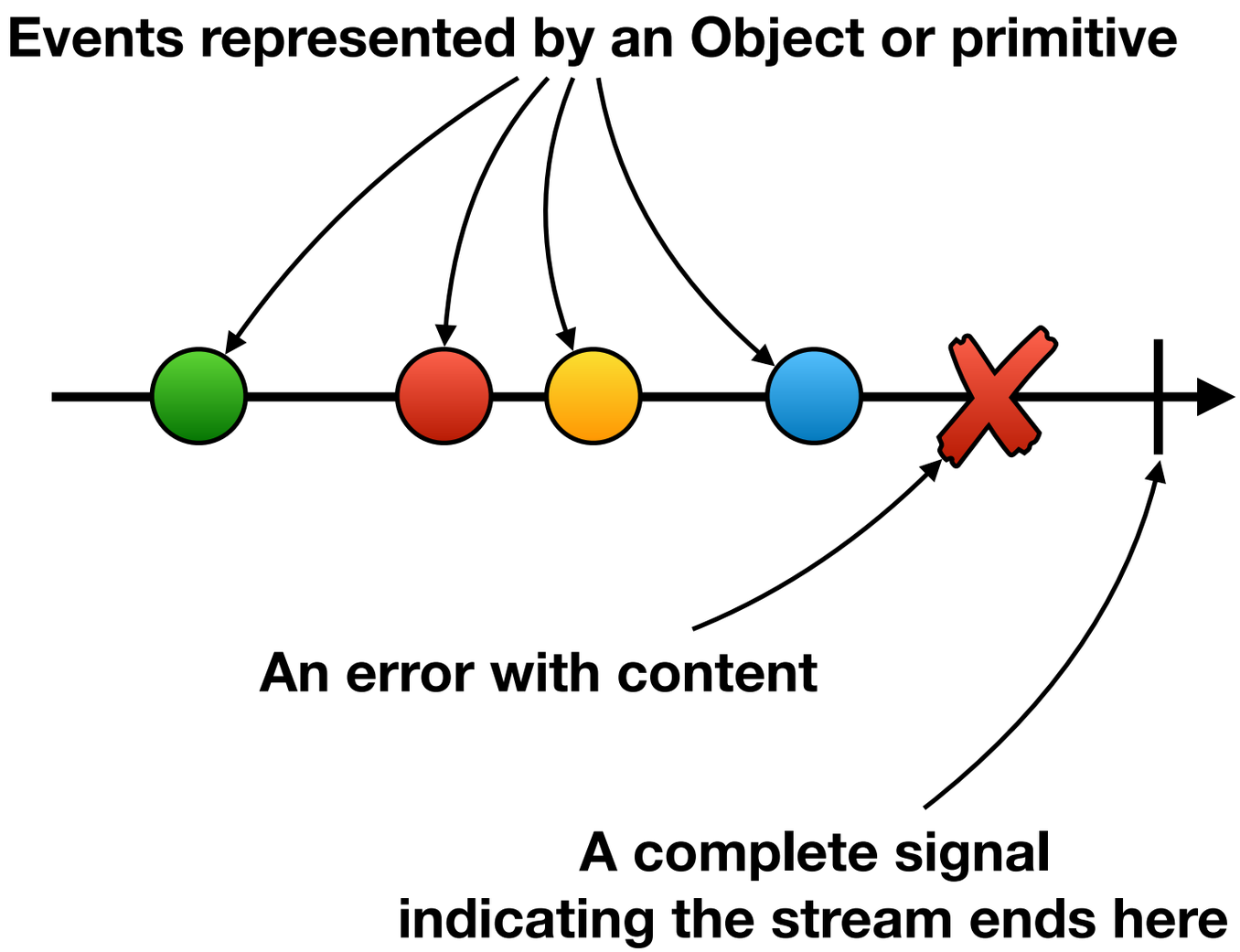
- Streams on steroids
- Not only events
- Everything can be a stream

Data Streams

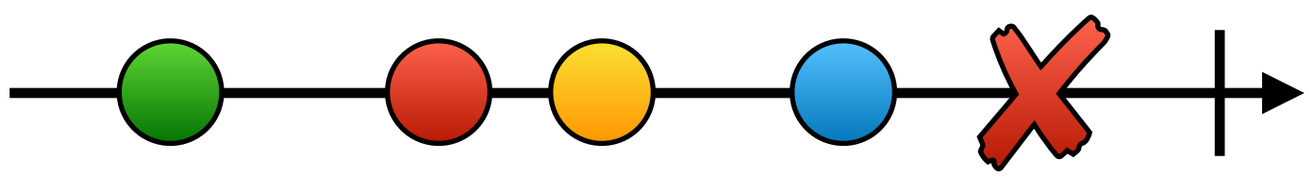
- Declarative functions
  - Create
  - Filter
  - Merge
  - Combine
  - ...
- Streams as input for other streams

How to represent this?

Marble Diagram



Marble Diagram



--a----b-c---d---#-|->

Capturing Events

- As they occur
- 3 functions
  - Value
  - Error
  - Complete

- Listening = Subscribing
- Functions = Observers
- Stream = subject being observed

➡ **Observer Design Pattern**

Observers only receive events after the subscription is created!

Subscribing

```
stream$.subscribe(
  (value: any) => { ... }, // next
  (error: any) => { ... }, // error
  () => { ... }            // complete
);
```

Subscribing

```
stream$.subscribe({
  next: (value: any) => { ... },
  error: (error: any) => { ... },
  complete: () => { ... }
});
```

Unsubscribing

```
let subscription;
onInit() {
  subscription = stream$.subscribe(...);
}

...

onDestroy() {
  subscription.unsubscribe();
}
```

Stream Functions

- Operators
- Creation operators
- Pipeable operators

Creation Operators

Create a stream

- fromEvent
- of
- from
- merge
- combineLatest
- ...

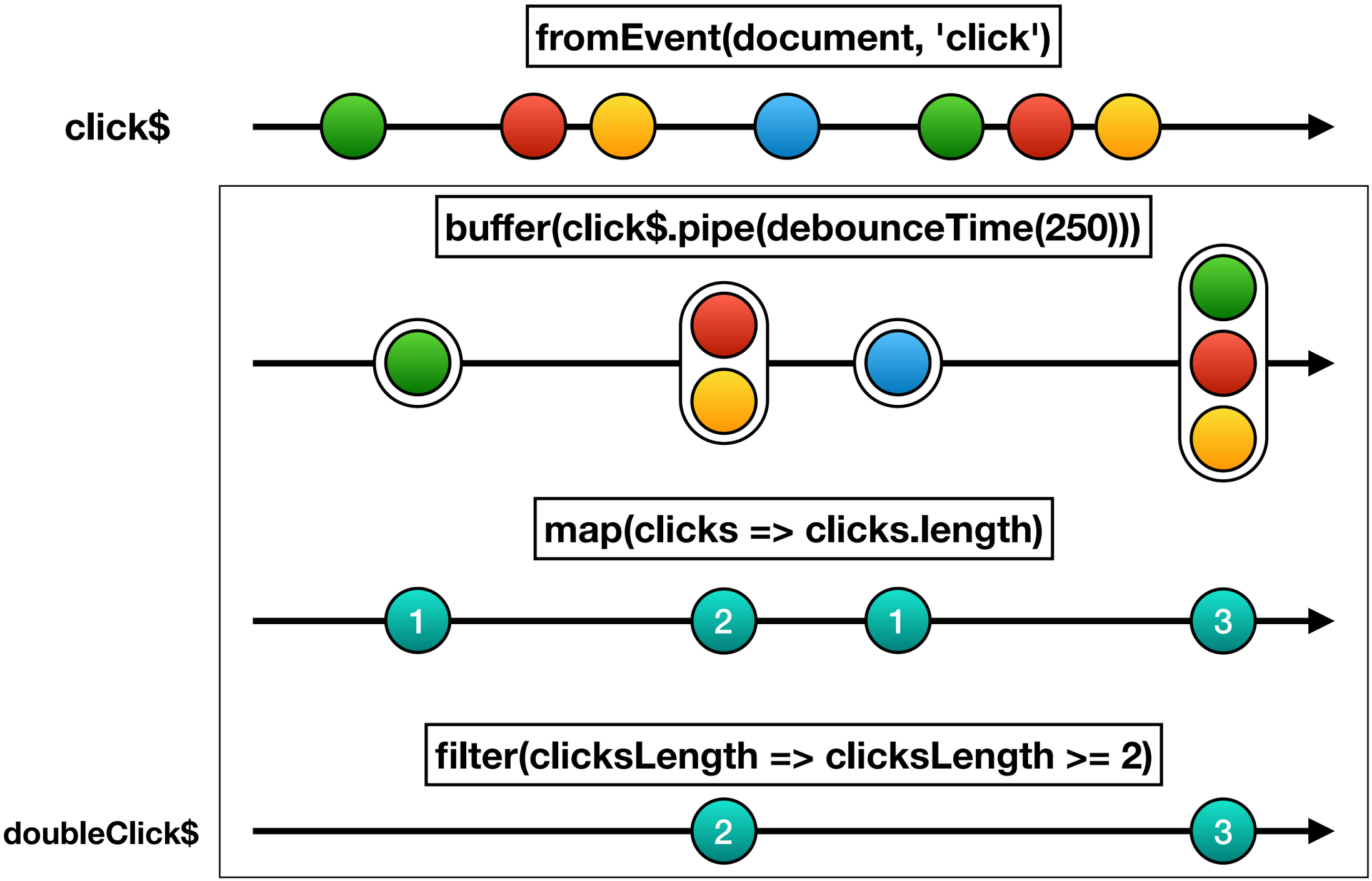
https://stackblitz.com/edit/creation-operators

Pipeable Operators

Transform a stream

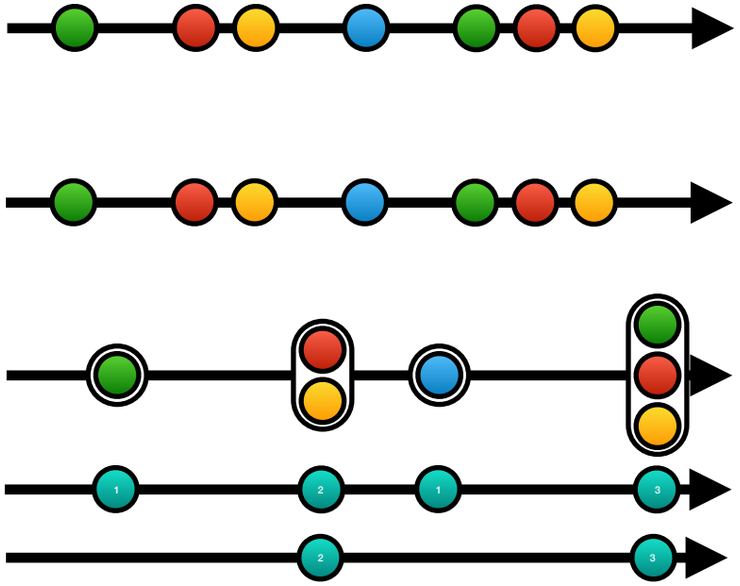
- map
- filter
- scan
- distinctUntilChanged
- switchMap
- ...

Example: Double Click



Example: Double Click

```
const click$ = fromEvent(document, 'click');

const doubleClick$ = click$
.pipe(
  buffer(click$.pipe(debounceTime(250))),
  map(clicks => clicks.length),
  filter(clicksLength => clicksLength >= 2)
);

doubleClick$.subscribe(_ => {
  console.log('double clicked detected', _)
});
```



https://stackblitz.com/edit/rxjs-doubleclick-example

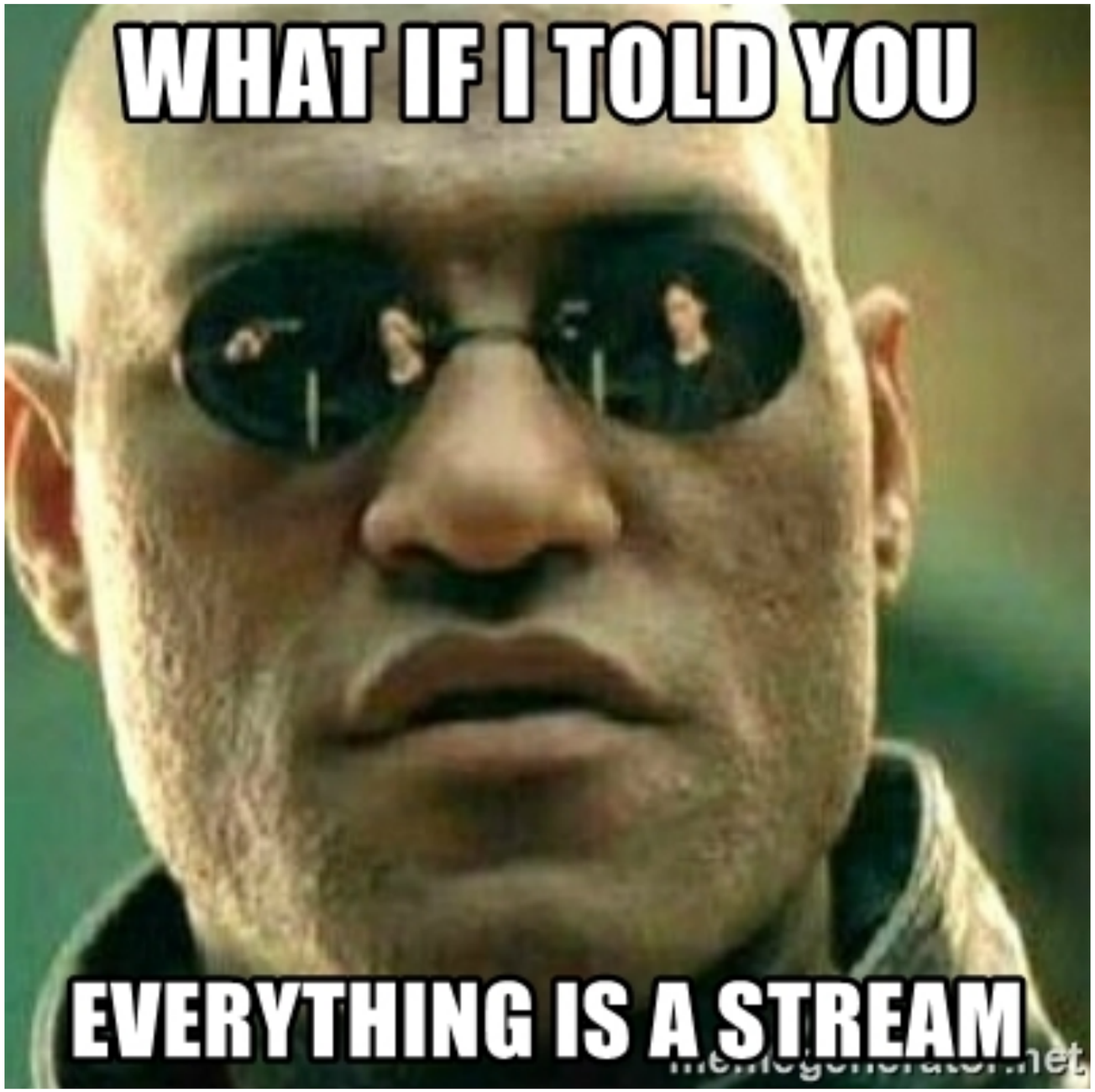
Why use RxJS

- Don't go to Callback Hell
- Raises abstraction
- Declarative programming
- Web applications with multitude of UI events
- **Pushing instead of Pulling data**

Thinking in Reactive Programming

```
let a: number;

let b: number;

let c: number;// = e-(a + b)

let d: number;

let e: number; // = b * d;

calcC() {
  c = e - (a + b);
}

calcE() {
  e = b * d;
  calcC();
}
```

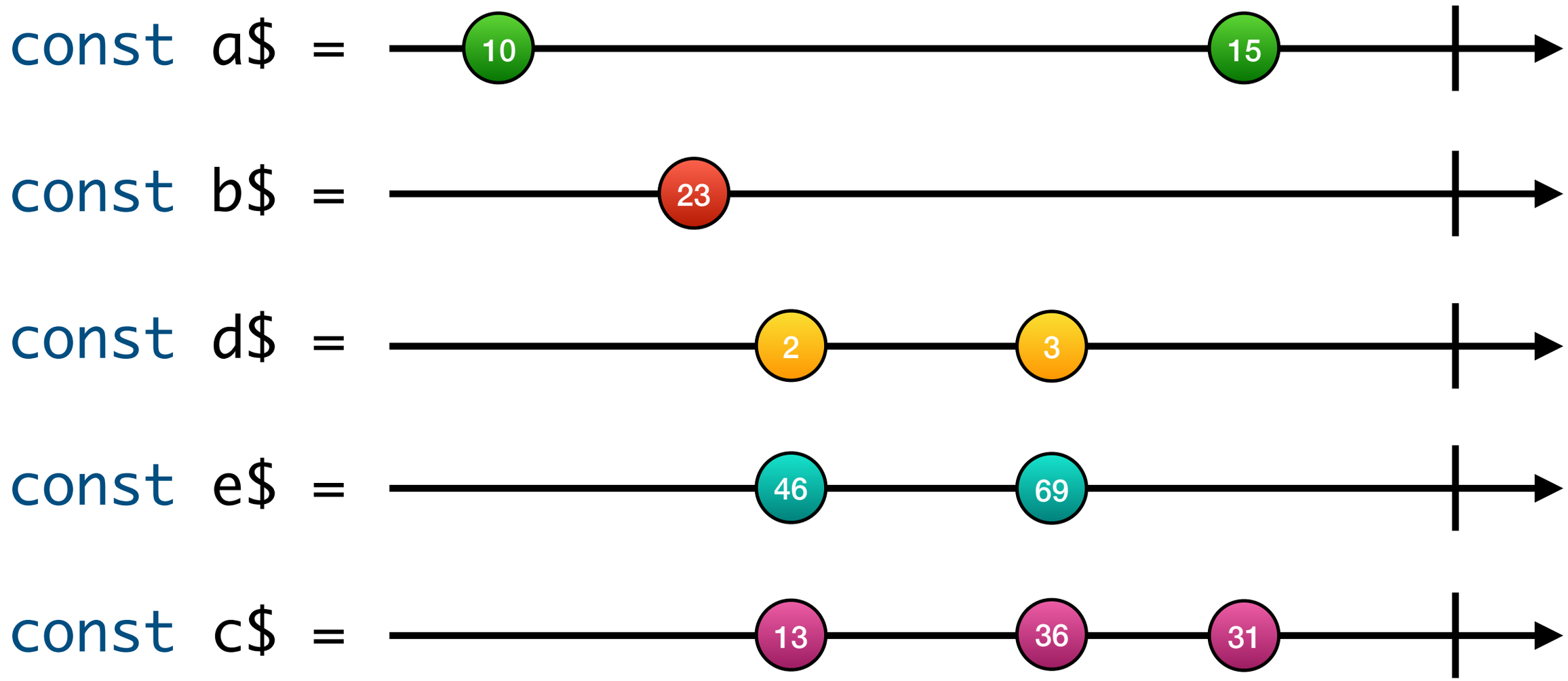
```
setA(value: number) {
  a = value;
  calcC();
}

setB(value: number) {
  b = value;
  calcE();
  calcC();
}

setD(value: number) {
  d = value;
  calcE();
}
```



SIP-principle

- Source
- Intermediary
- Presentational

```
e = b * d
c = e - (a + b)
```



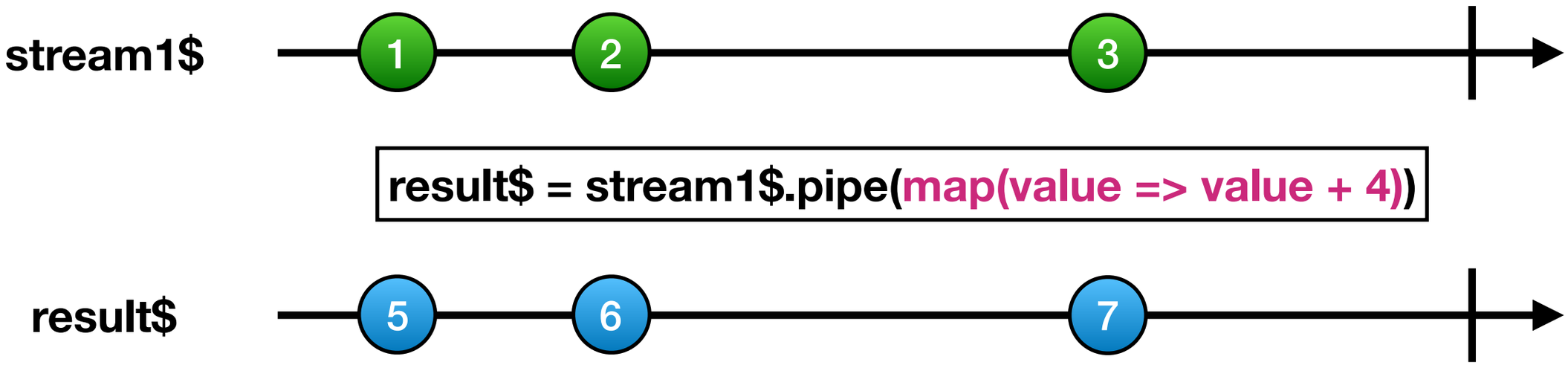
```
const a$ = of(10, 15);

const b$ = of(23);

const d$ = of(2, 3);

const e$ = combineLatest([b$, d$]).pipe(
  map(([b, d]) => b * d)
);

const c$ = combineLatest([a$, b$, e$]).pipe(
  map(([a, b, e]) => e - (a + b))
);

e$.subscribe(e => console.log(`value of e: ${e}`))

c$.subscribe(c => console.log(`value of c: ${c}`))
```

Example

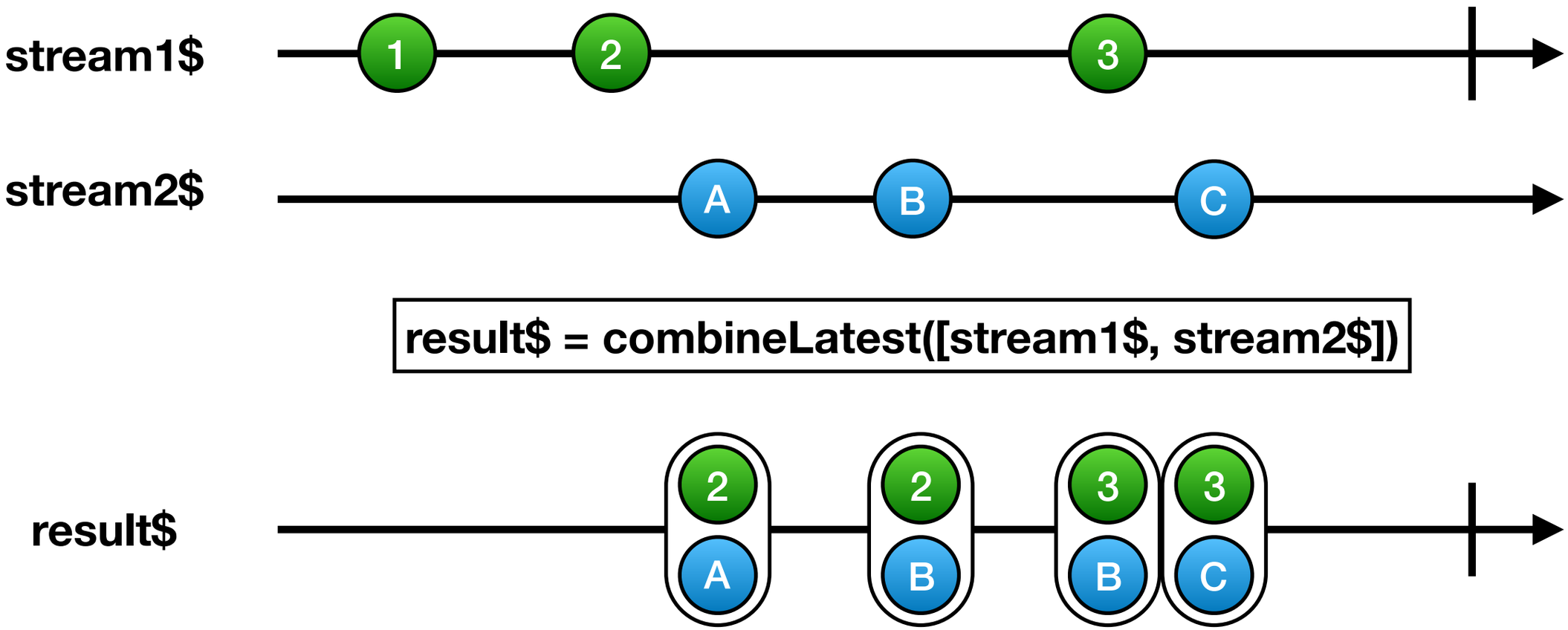
https://stackblitz.com/edit/calculation-example-no-rxjs

https://stackblitz.com/edit/calculation-example-with-rxjs

map

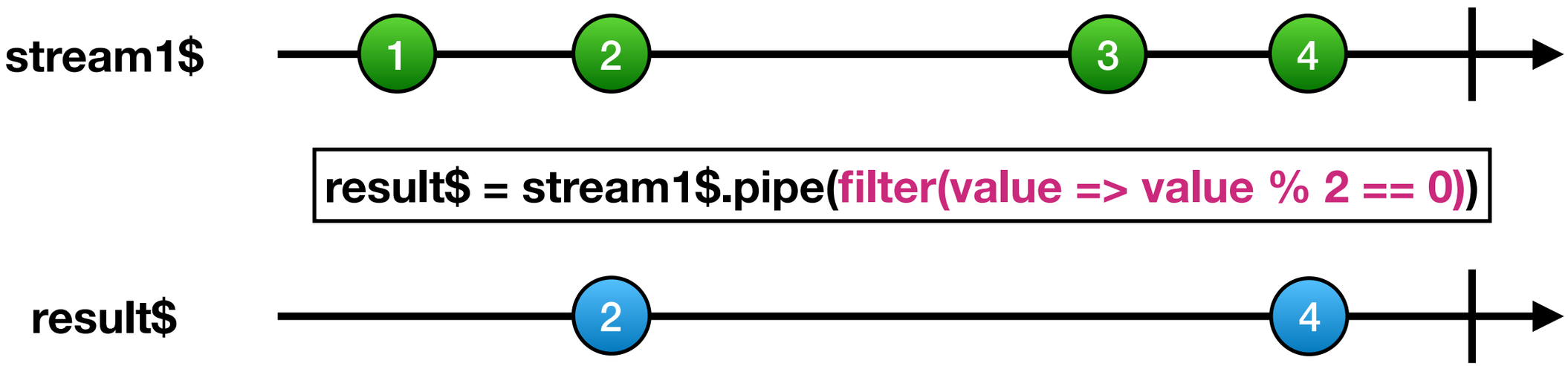


combineLatest

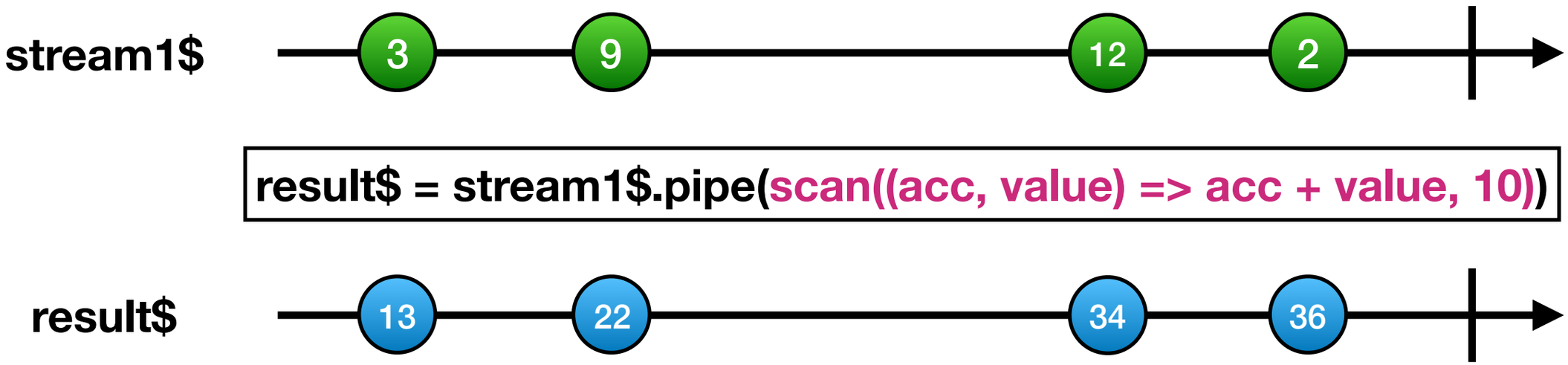


Other commonly used operators

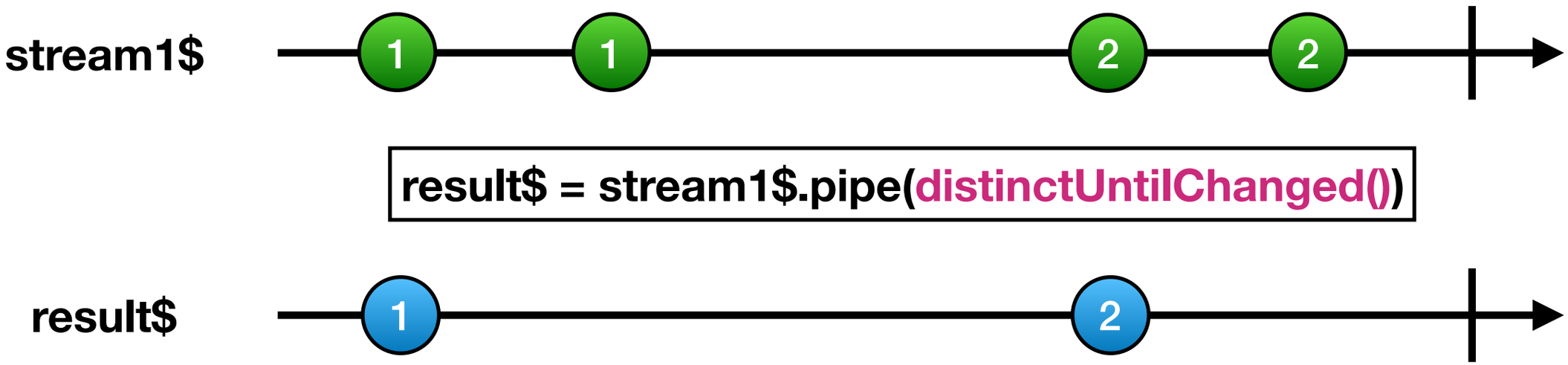
filter



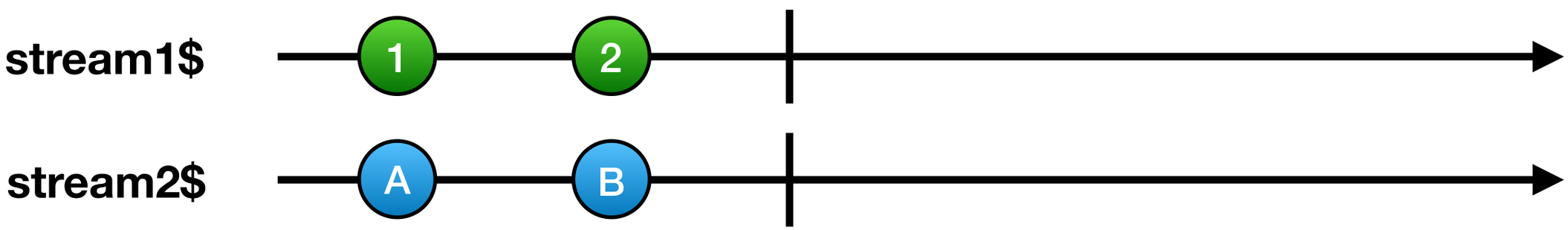
scan



distinctUntilChanged

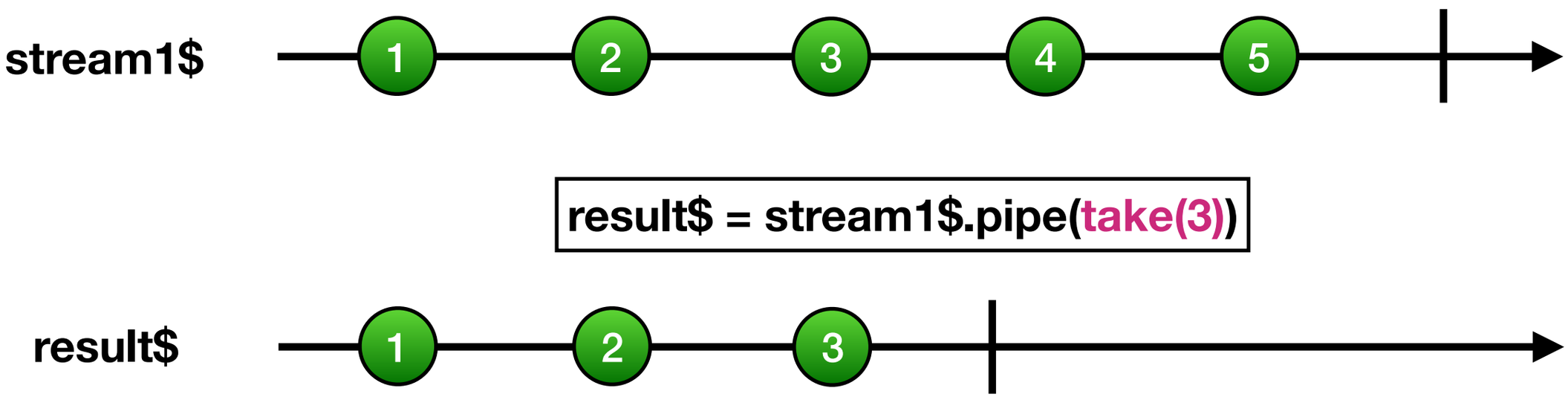


concat

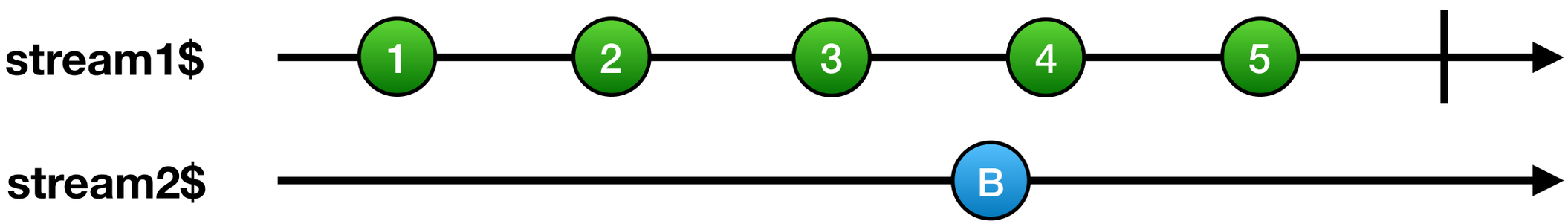


result$ = concat(stream1$, stream2$)

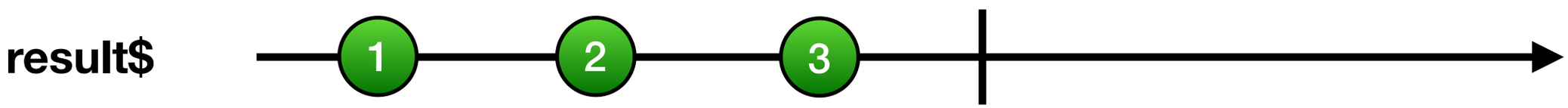
take



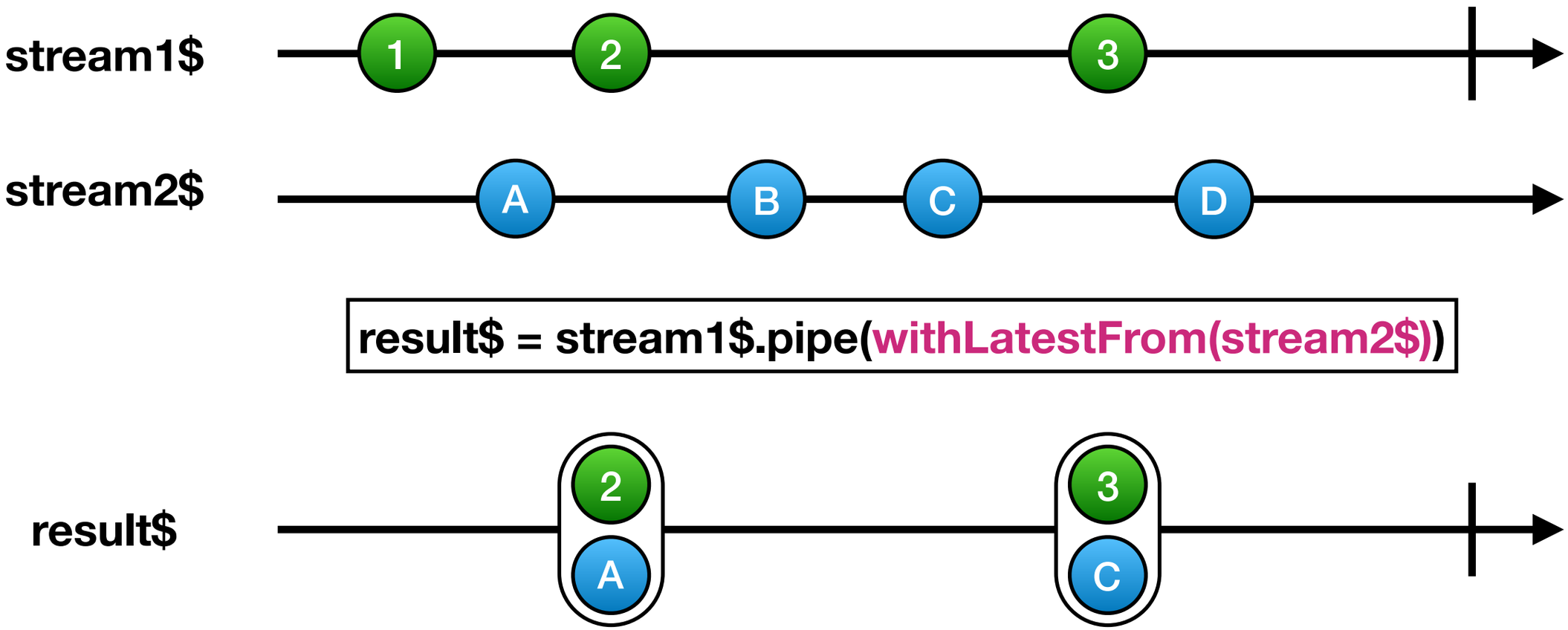
takeUntil



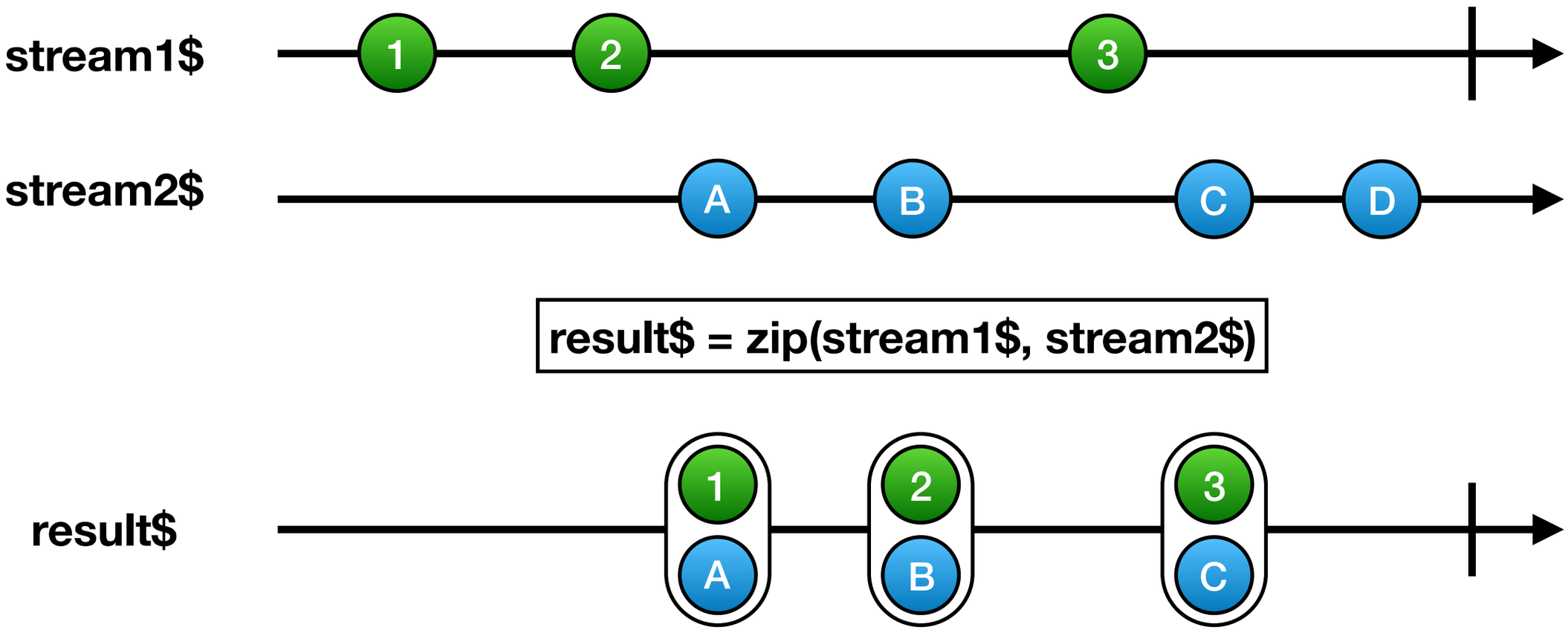

```
result$ = stream1$.pipe(takeUntil(stream2$))
```



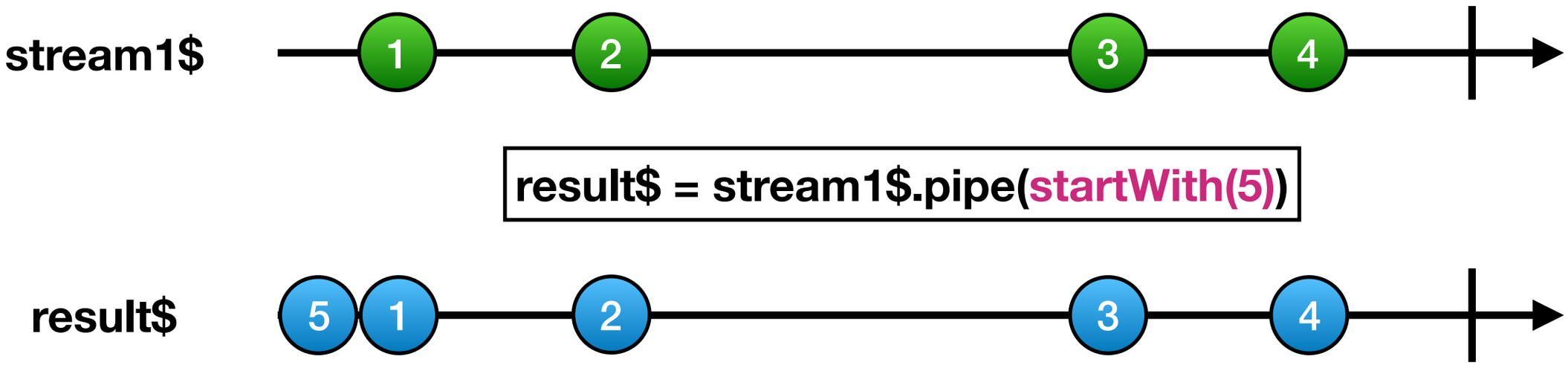
withLatestFrom



zip

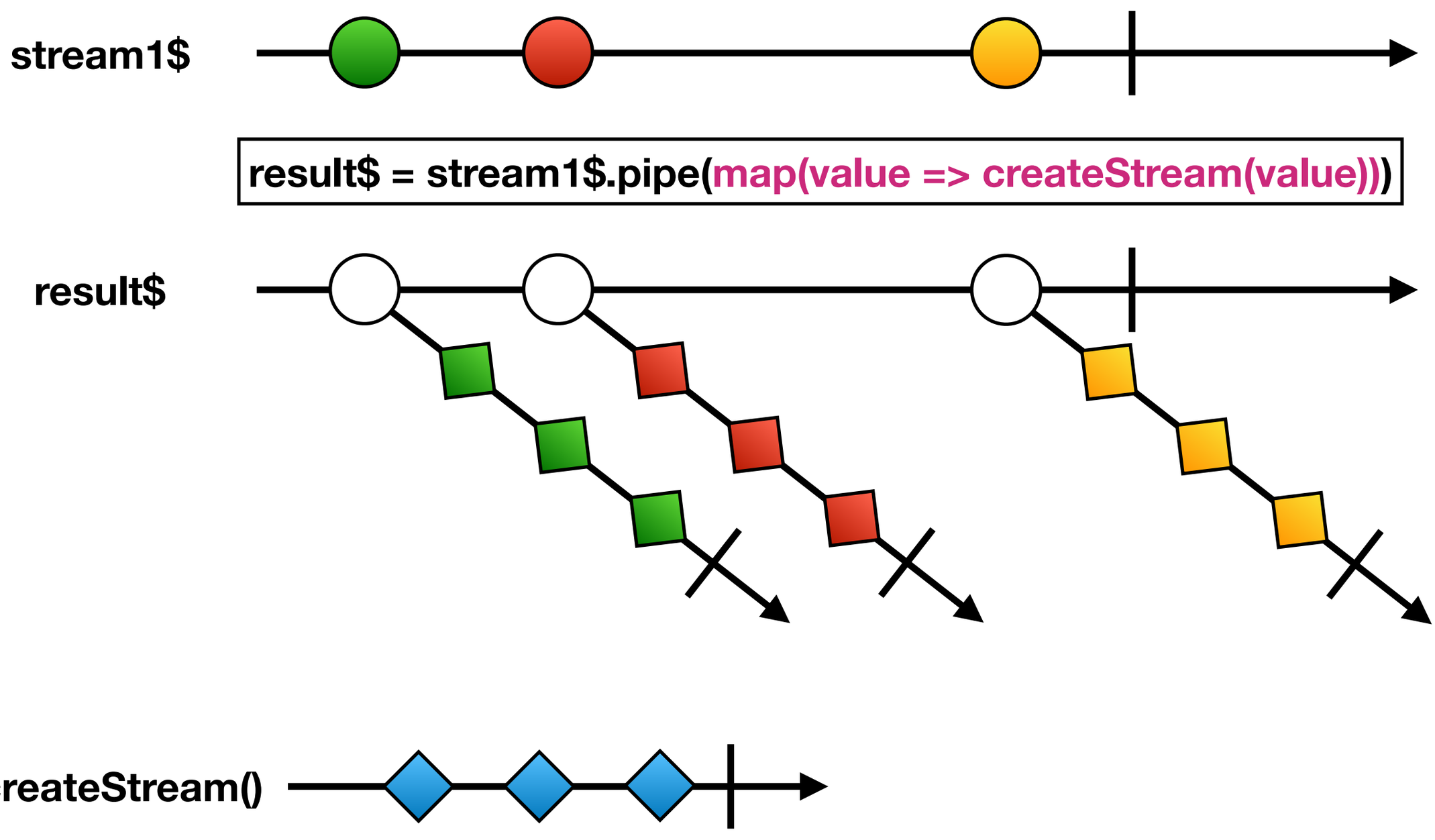


startWith



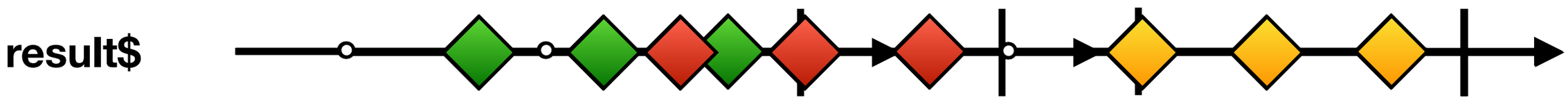
Let's get serious

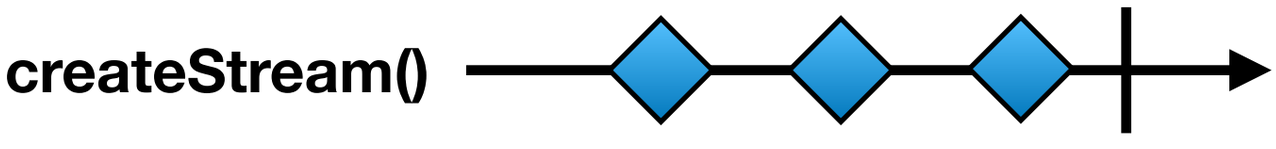
map



mergeMap

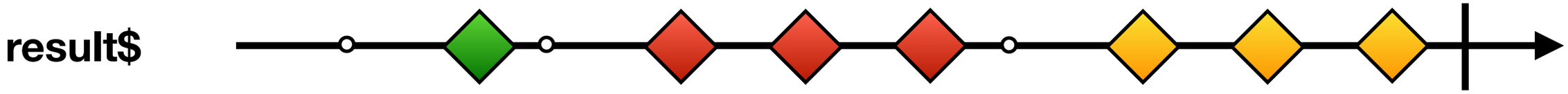

```
result$ = stream1$.pipe(mergeMap(value => createStream(value)))
```

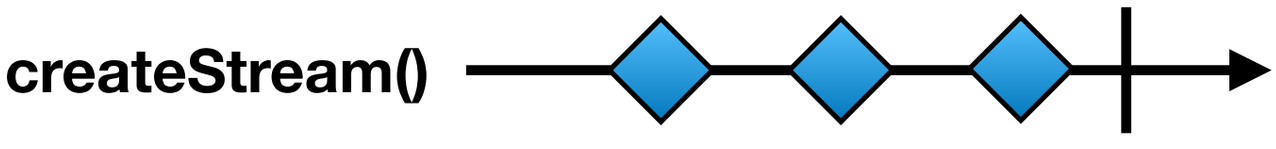




switchMap

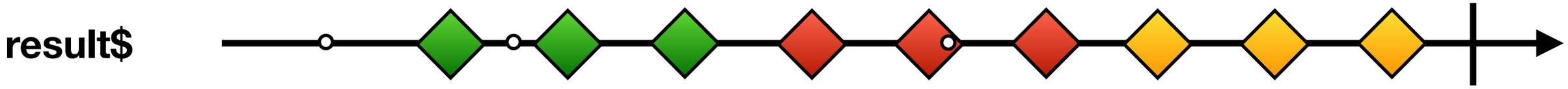

```
result$ = stream1$.pipe(switchMap(value => createStream(value)))
```

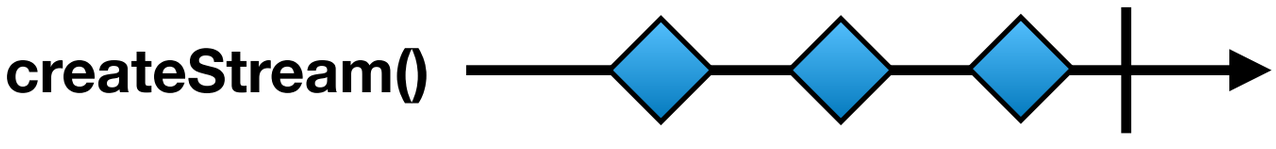




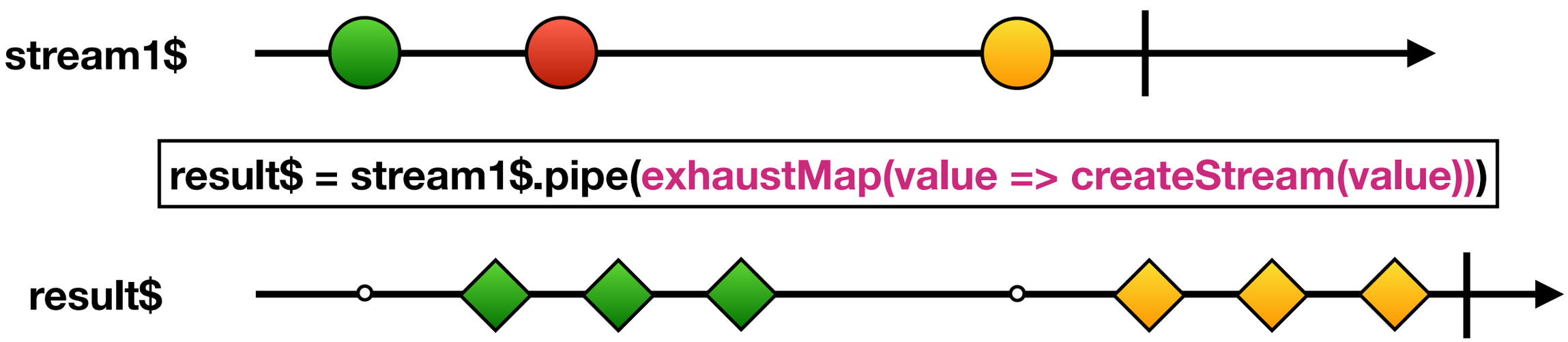
concatMap


```
result$ = stream1$.pipe(concatMap(value => createStream(value)))
```

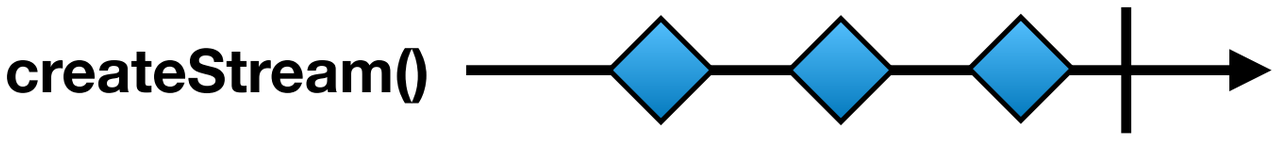




exhaustMap




```
result$ = stream1$.pipe(exhaustMap(value => createStream(value)))
```



Avoid nested subscriptions!

```
const sub = stream1$.subscribe(value1 => {
  console.log(`value1: ${value1}`);
  createStreamFrom(value1)
    .subscribe(
      value2 => console.log(`value2: ${value2}`)
    )
});

sub.unsubscribe();
```

Avoid nested subscriptions!

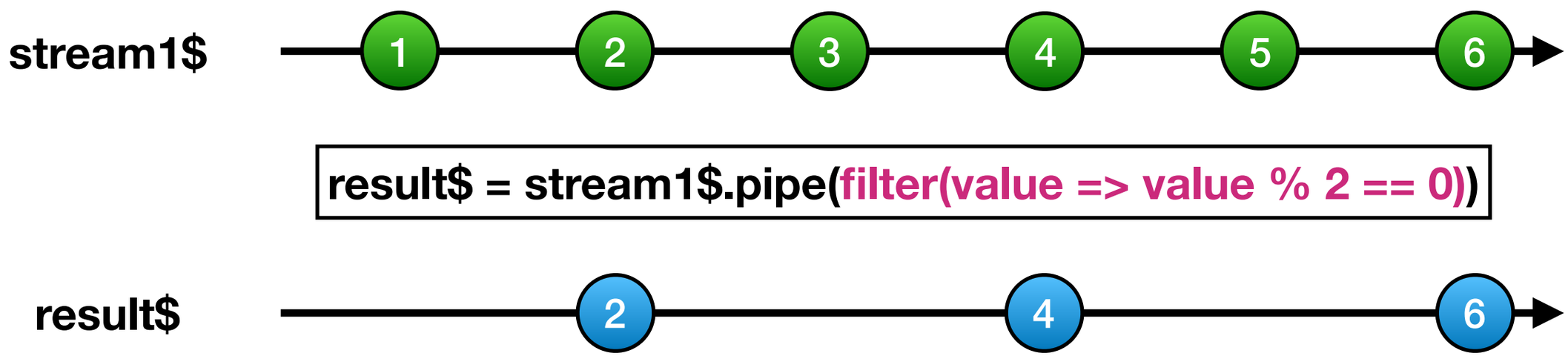
```
const sub = stream1$.pipe(
  mergeMap(value1 => {
    console.log(`value1: ${value1}`);
    return createStreamFrom(value1);
  })
).subscribe(
      value2 => console.log(`value2: ${value2}`)
);

sub.unsubscribe();
```

https://stackblitz.com/edit/nested-subscriptions

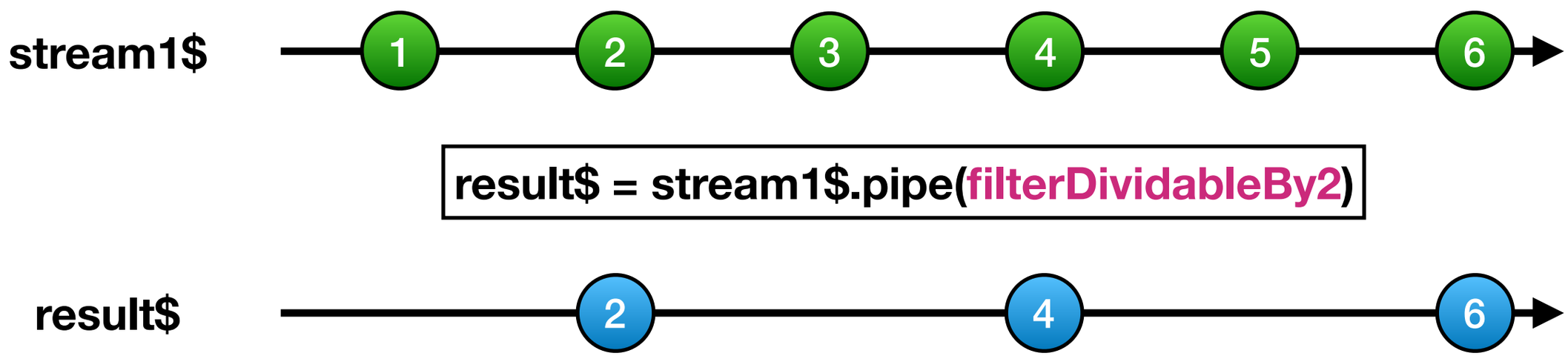
Creating your own operators

- Reusable
- Pure functions
- Testable

Example:
Filter Dividable by 2

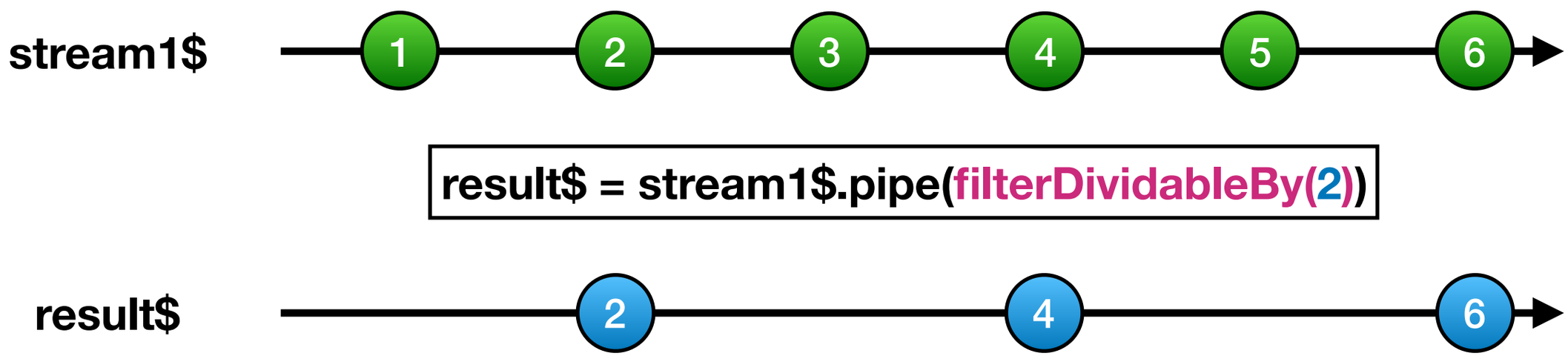


Example:
Filter Dividable by 2



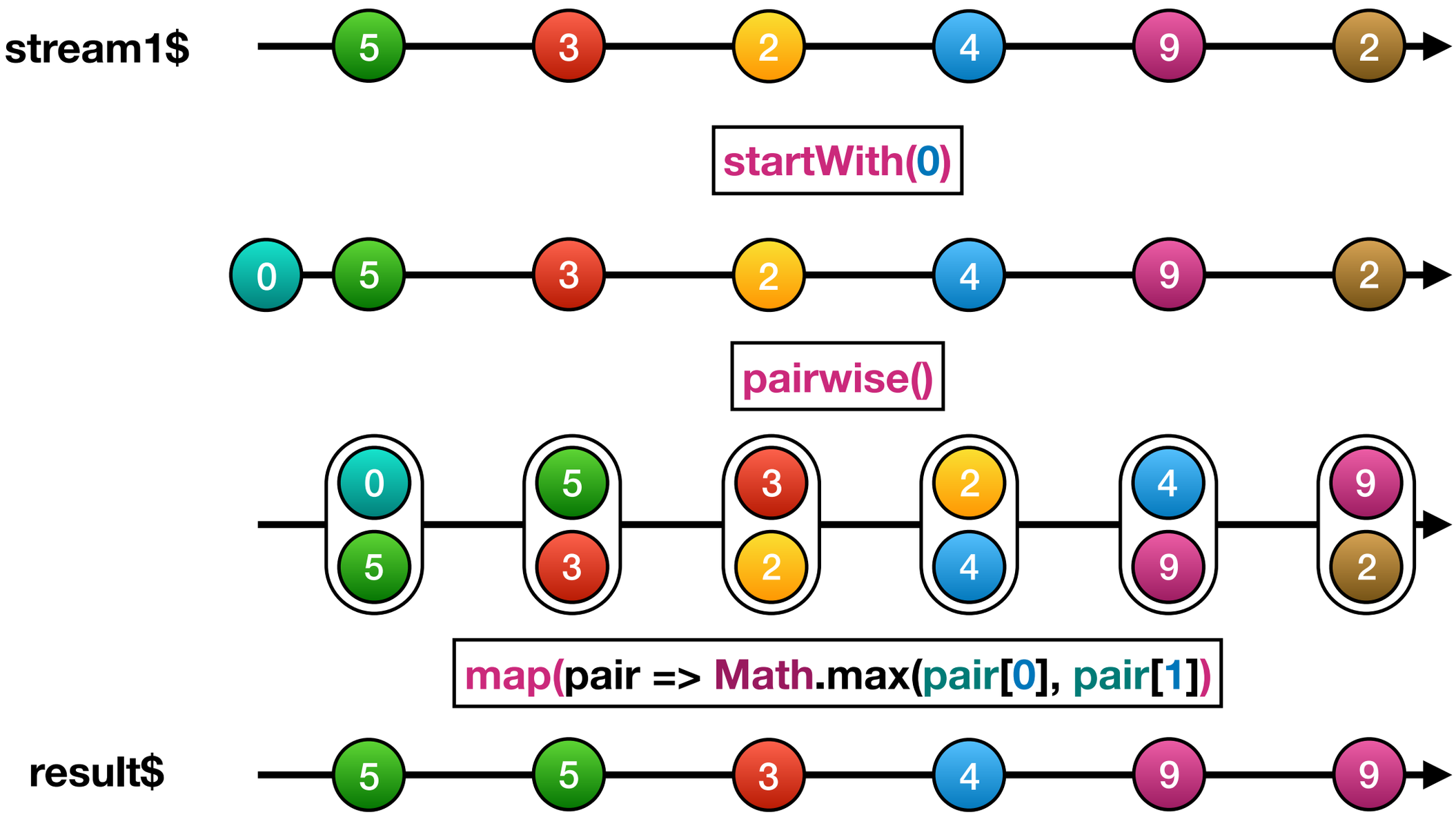

```
const filterDividableBy2 = filter<number>(value => value % 2 == 0)
```

Example:
Filter Dividable by n



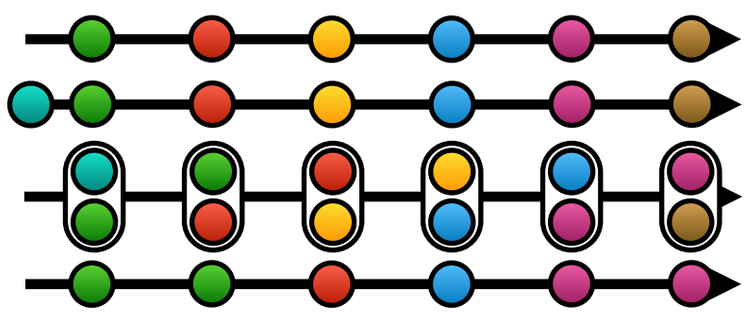

```
const filterDividableBy = (divider: number) =>
  filter<number>(value => value % divider == 0)
```

Example:
Pairwise comparison

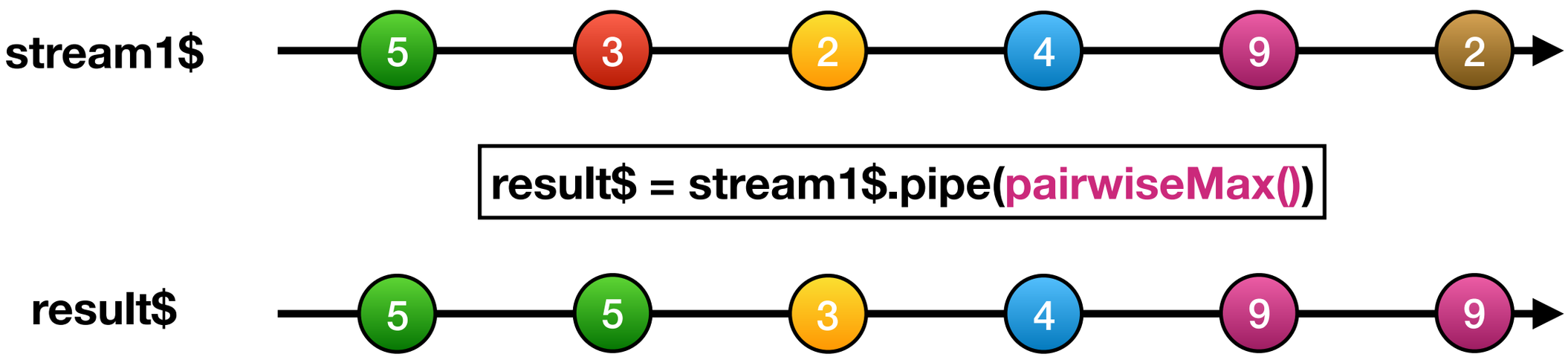


Example:
Pairwise comparison

```
const result$ = stream1$.pipe(
  startWith(0),
  pairwise(),
  map(pair => Math.max(pair[0], pair[1]))
);
```



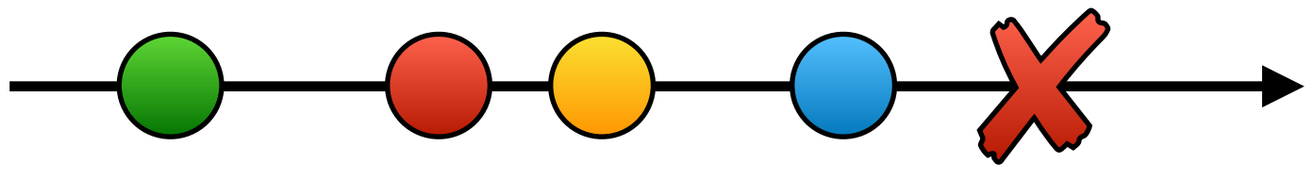
Example: Pairwise comparison




```
const pairwiseMax = () =>
  (source$: Observable<number>): Observable<number> =>
    source$.pipe(
      startWith(0),
      pairwise(),
      map(pair => Math.max(pair[0], pair[1]))
    );
```

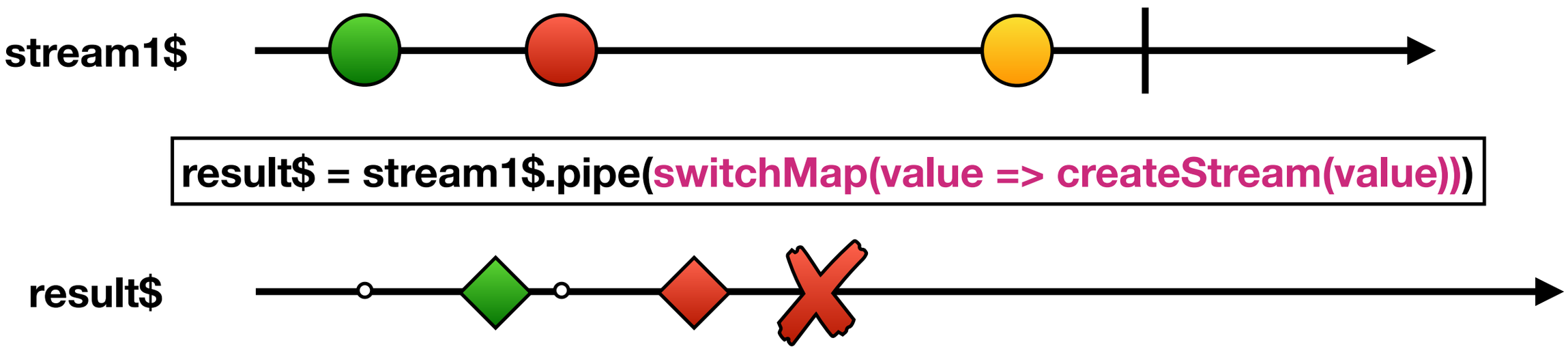
Error Handling

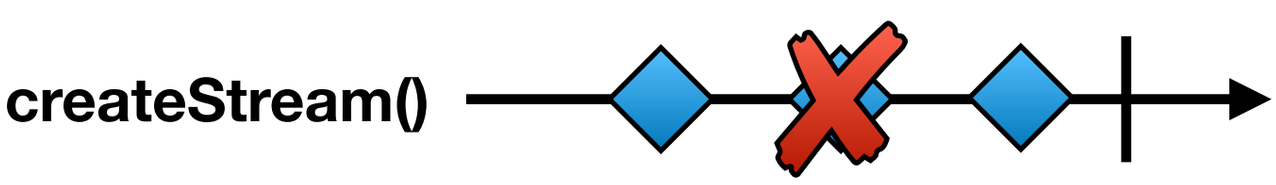
What happens on Error



- Calls Observer's error function
- Ends Subscription
- Does NOT call Observer's complete function

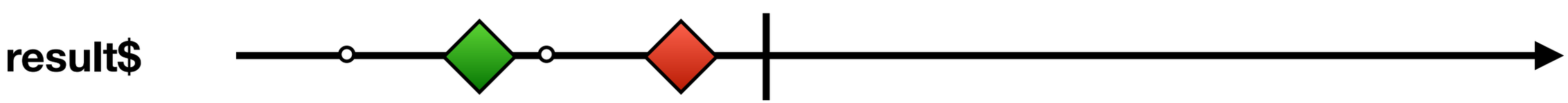
What if...

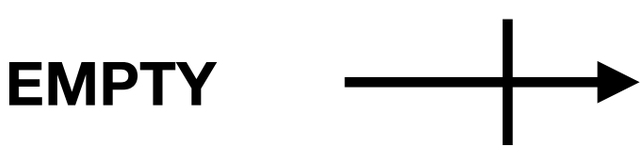


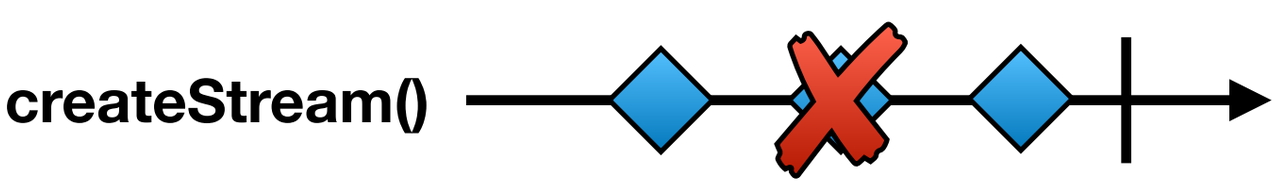


catchError

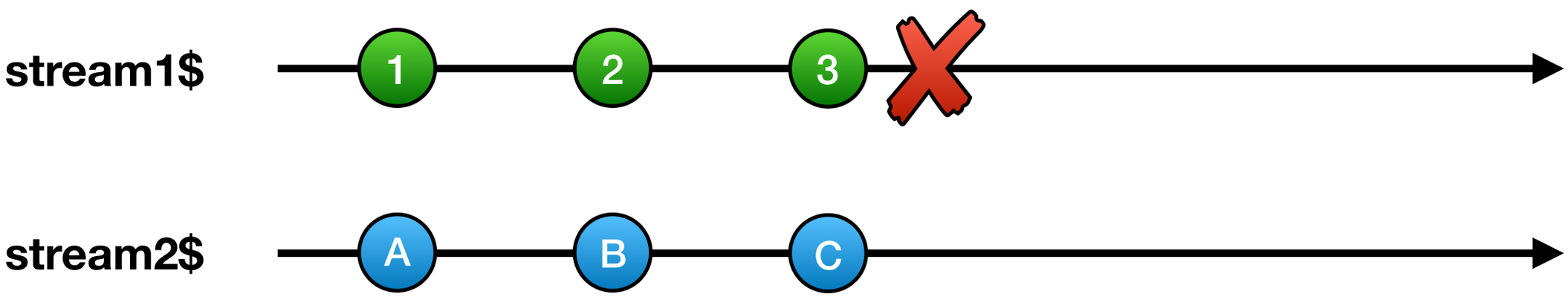

```
result$ = stream1$.pipe(
  switchMap(value => createStream(value))
  catchError(() => EMPTY)
)
```



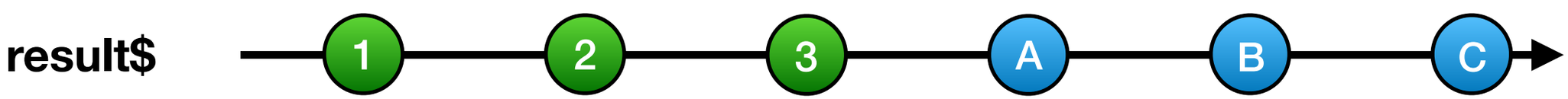




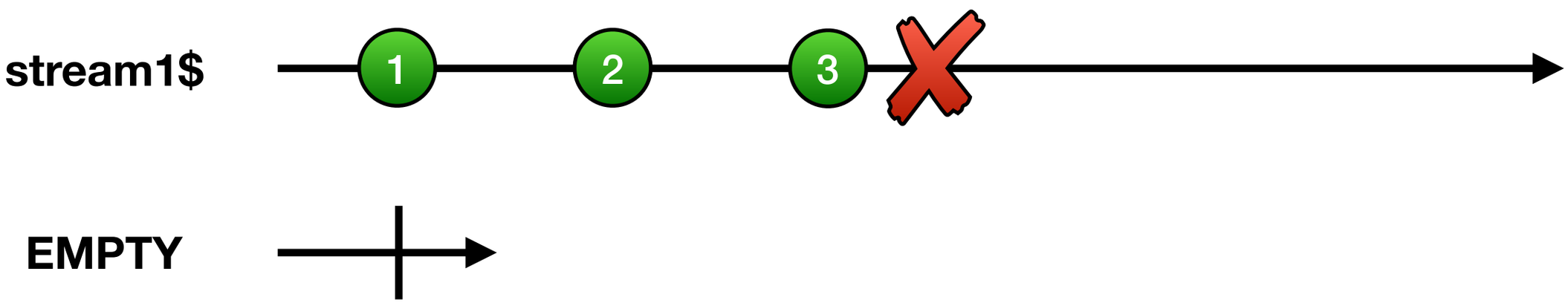
catchError



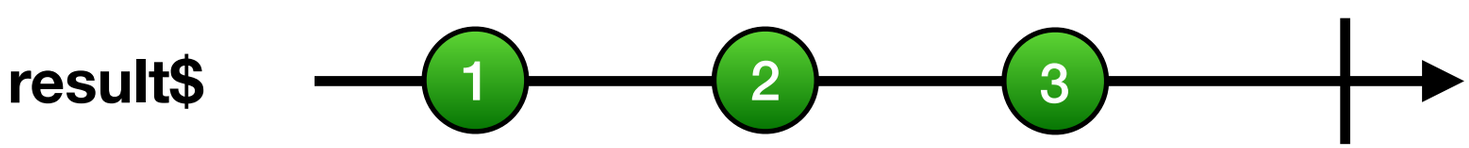

```
result$ = stream1$.pipe(catchError(error => {
  console.log(error);
  return stream2$;
}))
```



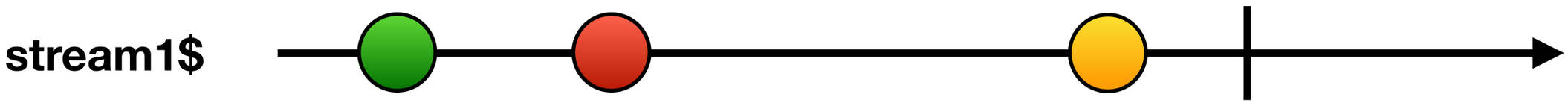
catchError



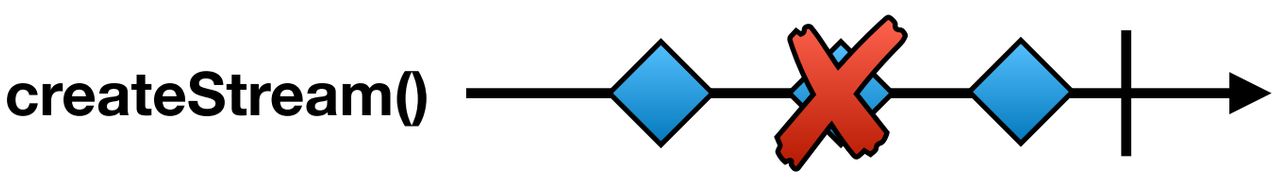

```
result$ = stream1$.pipe(catchError(error => {
  console.log(error);
  return EMPTY;
}))
```



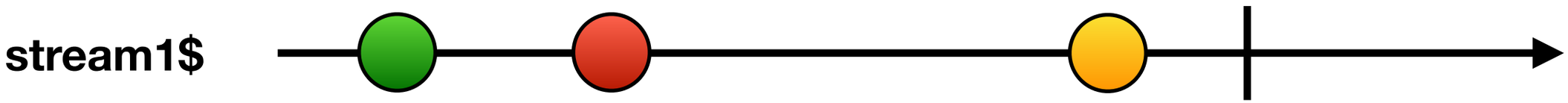
catchError



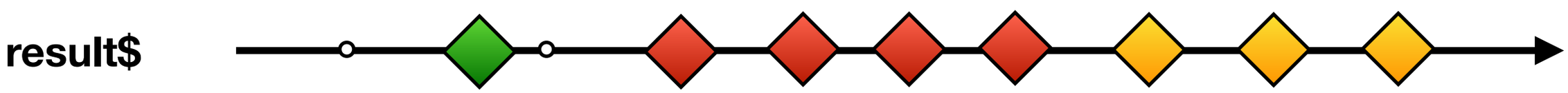

```
result$ = stream1$.pipe(
  switchMap(value => createStream(value).pipe(
    catchError(() => EMPTY)
  ))
)
```



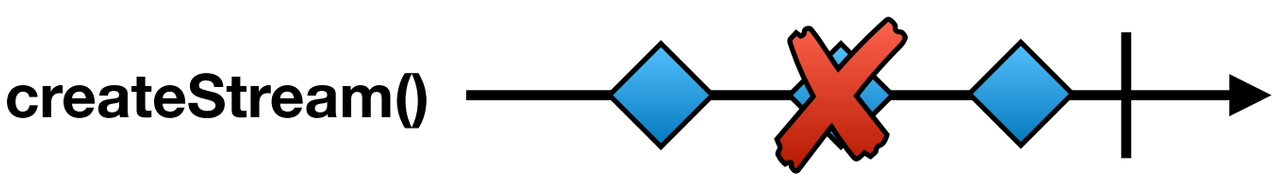
retry(n)




```
result$ = stream1$.pipe(
  switchMap(value => createStream(value).pipe(
    retry()
  ))
)
```



Be careful for infinite loops when retrying a stream that fails every time! Use a retry count or retryWhen



Hot vs. Cold Observables

- Cold Observables
  - Re-executes the subscription every time it's subscribed to
  - e.g. A HTTP Request
- Hot Observables
  - Routes subscriptions to the existing stream
  - e.g. A keyboard event stream

Hot vs. Cold Observables

- Cold to Hot Observable
  - share operator
  - shareReplay operator
- Hot to Cold Observable
  - Make the Observable the data producer
  - e.g.: by surrounding it by a function

Hot vs. Cold Observables

- Example: https://stackblitz.com/edit/rxjs-hot-vs-cold-observables

Recap

- Hot & Cold Observables
- Observers
- What about both?

Subject

- Observable
- AND Observer
  - next
  - error
  - complete

Subject

- Stream
- Events from its functions

Subject

```
const mySubject$ = new Subject();

mySubject$.subscribe(console.log);

fromEvent(document, 'click')
  .subscribe(mySubject$);
```

Subject

```
const mySubject$ = new Subject();

mySubject$.pipe(
  filter(value => value > 10)
).subscribe(console.log);

mySubject$.next(12);
mySubject$.next(9);
mySubject$.next(203);
```

Types of Subjects

- Subject
- AsyncSubject
- ReplaySubject
- BehaviorSubject

AsyncSubject

- Only emits last event before Complete
- OR the error when error is emitted

```
const mySubject$ = new AsyncSubject();

mySubject$.subscribe(console.log,
  err => console.log('error:', err),
  complete => console.log('completed')
);

mySubject$.next(12);
mySubject$.next(203);

mySubject$.complete();

// 203, completed
```

```
const mySubject$ = new AsyncSubject();

mySubject$.subscribe(console.log,
  err => console.log('error:', err),
  complete => console.log('completed')
);

mySubject$.next(12);
mySubject$.next(203);

mySubject$.error('Some error');

// error: Some error
```

ReplaySubject

- Replays events from before a subscription was created
- Set buffer size in construction

```
const mySubject$ = new ReplaySubject(2);

mySubject$.next(12);
mySubject$.next(9);
mySubject$.next(203);

mySubject$.subscribe(console.log);

mySubject$.next(84);

// 9, 203, 84
```

BehaviorSubject

- Replays 1 event from before a subscription was created
- Needs a default value
- Light-weight state storage solution

```
const mySubject$ = new BehaviorSubject(30);

mySubject$.subscribe(console.log);

mySubject$.next(84);

// 30, 84

console.log(mySubject$.getValue()); // 84
```

Unsubscribing

```
let subscription;
onInit() {
  subscription = stream$.subscribe(...);
}

...

onDestroy() {
  subscription.unsubscribe();
}
```

Unsubscribing

```
let subscription1;
let subscription2;
onInit() {
  subscription1 = stream1$.subscribe(...);
  subscription2 = stream2$.subscribe(...);
}

...

onDestroy() {
  subscription1.unsubscribe();
  subscription2.unsubscribe();
}
```

Unsubscribing

```
let subscription1;
let subscription2;
...
let subscriptionN;
onInit() {
  subscription1 = stream1$.subscribe(...);
  subscription2 = stream2$.subscribe(...);
  ...
  subscriptionN = streamN$.subscribe(...);
}

...

onDestroy() {
  subscription1.unsubscribe();
  subscription2.unsubscribe();
  ...
  subscriptionN.unsubscribe();
}
```

Unsubscribing

```
const subscriptions: Subscription[] = [];

onInit() {
  subscriptions.push(stream1$.subscribe(...));
  subscriptions.push(stream2$.subscribe(...));
  ...
  subscriptions.push(streamN$.subscribe(...));
}

...

onDestroy() {
  subscriptions
    .filter(s => !s.closed)
    .forEach(s => s.unsubscribe());
}
```

Unsubscribing

```
const destroy$ = new Subject();

onInit() {
  stream1$.pipe(takeUntil(destroy$)).subscribe(...);
  stream2$.pipe(takeUntil(destroy$)).subscribe(...);
  ...
  streamN$.pipe(takeUntil(destroy$)).subscribe(...);
}

...

onDestroy() {
  destroy$.next(null);
}
```

Debugging

Debugging

Don't use console.log!

You don't need to change your code so you can debug it.

Chrome Dev Tools

```
123   private getToteInformationFromToteNumber() {
124     return merge(
125       this.conveyableForm.get('carrier').statusChanges.pipe(
126         filter(status => status === 'VALID'),
127         withLatestFrom(this.conveyableForm.get('carrier').valueChanges),
128         map(([_, value]) => value),
129         switchMap(value => this.conveyableItemsSandbox.getToteInformation(value))
130       ),
131       this.conveyableForm.get('carrier').statusChanges.pipe(
132         filter(status => status !== 'VALID'),
133         mapTo(null)
134       )
135     );
136   }
```

Firefox Dev Tools

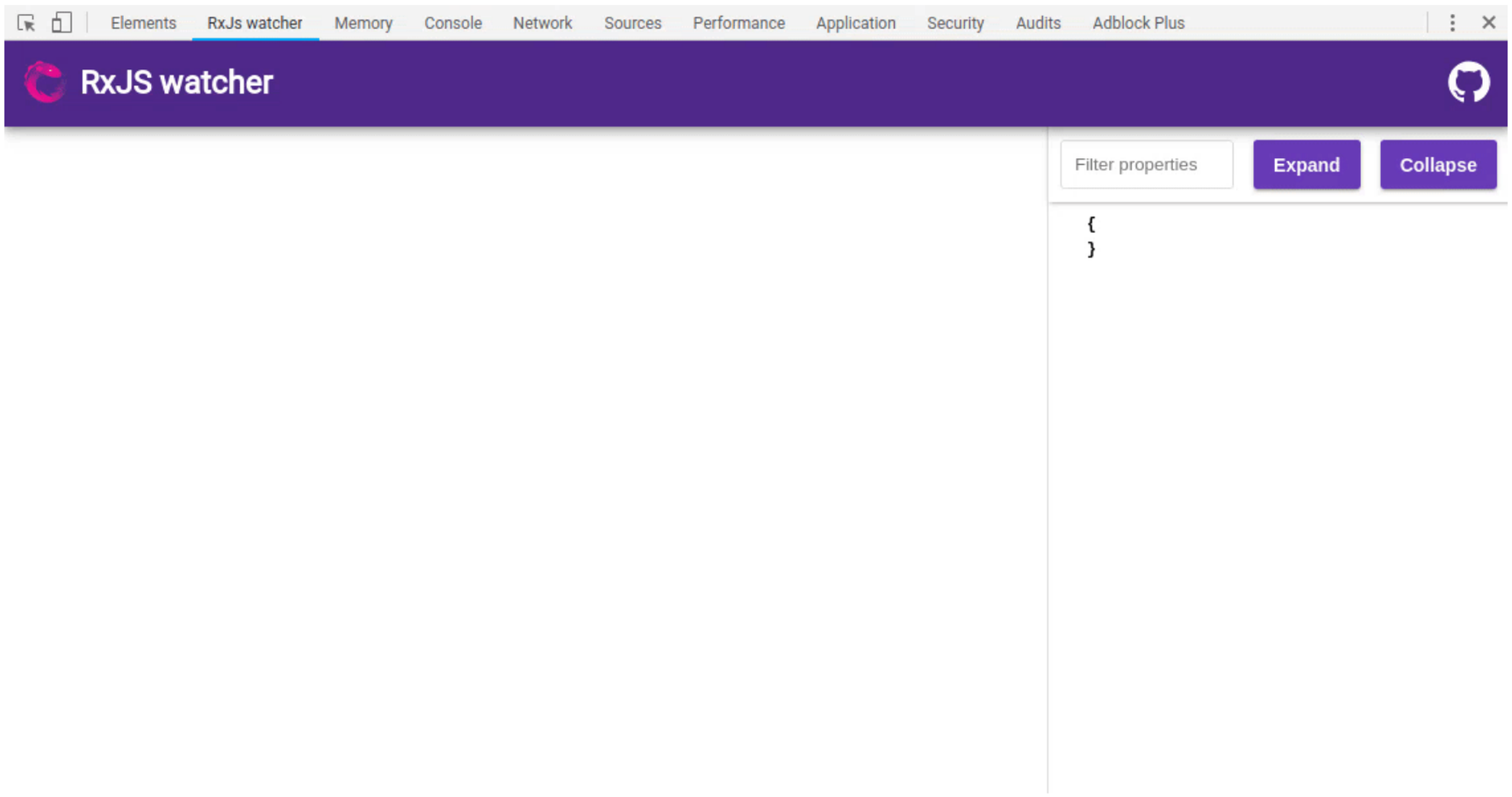
```
123   private getToteInformationFromToteNumber() {
124     return merge(
125       this.conveyableForm.get('carrier').statusChanges.pipe(
126         filter(status => status === 'V  "1"  '),
127         withLatestFrom(this.conveyable  .get('carrier').valueChanges),
128         map(([_, value]) => value),
129         switchMap(value => this.conveyableItemsSandbox.getToteInformation(value))
130       ),
131       this.conveyableForm.get('carrier').statusChanges.pipe(
132         filter(status => status !== 'VALID'),
133         mapTo(null)
134       )
135     );
136   }
```

RxJS Watcher

- Custom operator
- Can be installed via npm package
- Needs Chrome/Firefox extension
- Still better than console.log
- Displays streams in marble diagrams
- https://github.com/xripcsu/rxjs-watcher

RxJS Watcher

```
import { watch } from "rxjs-watcher";

interval(2000)
  .pipe(
    watch("Interval (2000)", 10),
    filter(v => v % 2 === 0),
    watch("Filter odd numbers out", 10),
  ).subscribe();
```



One more thing...

Unit Testing

Testing Promises

```
it('should do something', done => {
  createSomePromise()
    .then(result => {
      expect(result).toBeTruthy();
      done();
    });
});
```

Testing Subscriptions

```
it('should do something', done => {
  createSomeObservable()
    .subscribe(result => {
      expect(result).toBeTruthy();
      done();
    });
});
```

Testing Subscriptions

```
it('should do something', done => {
  let count = 0;
  createSomeObservable()
    .subscribe(result => {
      count++;
      if(count === 0) {
        expect(result).toBeTruthy();
      } else {
        expect(result).toBeFalsy();
        done();
      }
    });
});
```

Testing Subscriptions

```
it('should do something', done => {
  createSomeObservable()
    .pipe(take(1))
    .subscribe(result => {
        expect(result).toBeTruthy();
    });
  createSomeObservable()
    .pipe(skip(1))
    .subscribe(result => {
        expect(result).toBeFalsy();
        done();
      }
    });
});
```

Marble Testing

npm install rxjs-marbles --save-dev

Implements RxJS TestScheduler

Create tests using marble diagrams

Test asynchronous code synchronously

https://github.com/cartant/rxjs-marbles

https://github.com/ReactiveX/rxjs/blob/master/doc/marble-testing.md

Mock streams

```
const values = {
  'a': 'Ann',
  'b': 'Bob',
  'c': 'Curtis',
  'd': 'Dave',
  'e': 'Edna',
  'f': 'Frank',
}

m.hot('--a--b--b--a--c--d--|', values);
m.cold('----e----f---#', values);
```

1 character = 1 frame = 1ms (virtually)

Testing Subscriptions

```
it('should do something', marbles(m => {
  const values = {
    t: true,
    f: false
  };

  const actual$ = createSomeObservable();

  m.expect(actual$)
    .toBeObservable('-t--f--f--|', values);

}));
```

pairwiseMax

```
const pairwiseMax = () =>
  (source$: Observable<number>): Observable<number> =>
    source$.pipe(
      startWith(0),
      pairwise(),
      map(pair => Math.max(pair[0], pair[1]))
    );
```

pairwiseMax

```
describe('pairwiseMax', () => {
    it('should emit the max value pairwise', marbles(m => {
        const values = {
            'a': 5,
            'b': 3,
            'c': 2,
            'd': 4,
            'e': 9
        };
        const source$   = m.cold('--a--b--c--d--e--b-|', values);

        const result$ = pairwiseMax()(source$);

        m.expect(result$)
         .toBeObservable('--a--a--b--d--e--e-|', values);
    }));
});
```

pairwiseMax

```
describe('pairwiseMax', () => {
    it('should emit the max value pairwise', marbles(m => {
        const values = {
            'a': 5,
            'b': 3,
            'c': 2,
            'd': 4,
            'e': 9
        };
        const source$   = m.cold('--a--b--c--d--e--b-|', values);
        const expected$ = m.cold('--a--a--b--d--e--e-|', values);

        const result$ = pairwiseMax()(source$);

        m.equal(result$, expected$);
    }));
});
```

Unit testing
spies inside subscriptions

```
function myFunction(source$: Observable<number>) {
  source$.pipe(filterDividableBy2)
    .subscribe(value => {
      someFunction(value);
    });
}

describe('myFunction', () => {
    it('should only call someFunction when dividable by 2', marbles(m => {
        const values = {
            'a': 5,
            'b': 3,
            'c': 2,
            'd': 4,
            'e': 9
        };
        const source$   = m.cold('--a--b--c--d--e--|', values);
        myFunction(source$);

        m.flush();

        expect(someFunction).toHaveBeenCalledTimes(2);
    }));
});
```

Unit testing delays

```
describe('delay', () => {
    it('should delay a source for 20ms', marbles(m => {
        const values = {
            'a': 5,
            'b': 3,
            'c': 2,
        };
        const source$ = m.cold('--a--b--c--|', values);
        const result$ = source$.pipe(delay(20));

        m.expect(result$)
         .toBeObservable('20ms --a--b--c--|', values);
    }));
});
```

That's it...

Recap

- Streams are Observables
- Observers subscribe upon streams to listen to upcoming events
- Subjects are Observers and Observables
- catchError and retry on disposable inner streams
- Unit Testing with rxjs-marbles

And remember:

Useful links

- RxJS Official docs
  https://rxjs.dev

- RxJS Marbles:
  https://rxmarbles.com/

- Which creation operator do I use:
  http://xgrommx.github.io/rx-book/content/which_operator_do_i_use/creation_operators.html

- Which instance operator do I use:
  http://xgrommx.github.io/rx-book/content/which_operator_do_i_use/instance_operators.html

- Learn RxJS:
  https://www.learnrxjs.io/

- What are Schedulers in RxJS:
  https://blog.strongbrew.io/what-are-schedulers-in-rxjs/

- The introduction to Reactive Programming you've been missing:
  https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

- Marble Testing: https://github.com/ReactiveX/rxjs/blob/master/doc/marble-testing.md

Exercises

https://github.com/orjandesmet/rxjs-course-material/tree/exercises