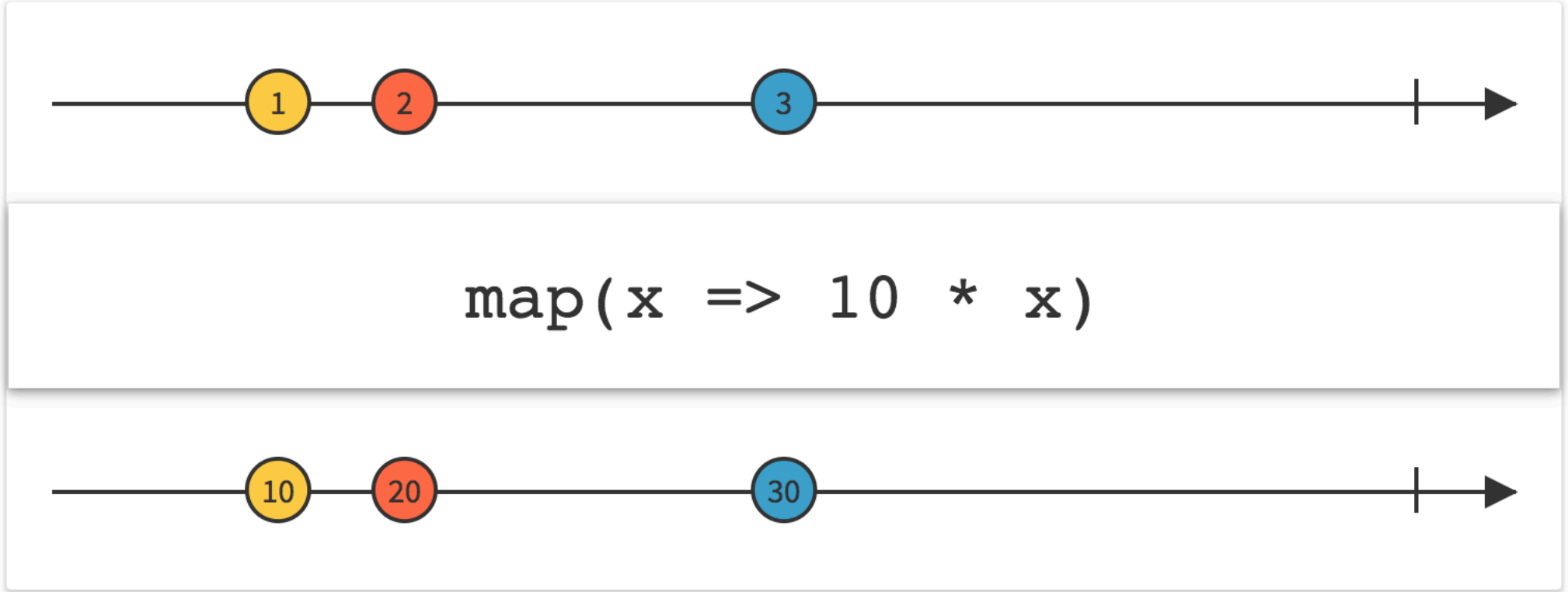
RxJS

RxJS



RxJS

- Reactive Programming
- RxJS & Angular
- Observable
- Operators

What is Reactive Programming?

- *a declarative programming paradigm concerned with data streams and the propagation of change*
- a way of responding to sets of events over a period of time
- a new mindset of thinking about events
- solve complex problems with a few lines of code

When to use Reactive Programming?

Responding to:

- User events
- Data changes
- State changes
- All other kinds of events

Why Reactive Programming?

- 1 way to think about data flow in the application
- Ability to compose a sequence of actions for an event
- Descriptive way of writing behaviour in code

Reactive Libraries

- RxJS
- RxJava
- Rx.NET
- RxSwift
- …

Example: double click

Imperative

```
previousClick = null;

onClick() {
  const thisClick = Date.now();
  if (thisClick - this.previousClick <= 250) {
    this.previousClick = null;
    double();
  } else {
    this.previousClick = thisClick;
    setTimeout(() => {
      if (this.previousClick) {
        this.previousClick = null;
        single();
      }
    }, 250);
  }
}
```

Example: double click

Reactive


```
@ViewChild('button') button;

ngAfterViewInit() {
  const clickStream = fromEvent(this.button.nativeElement, 'click');

  const finalClick = clickStream.pipe(debounceTime(250));

  clickStream
    .pipe(buffer(finalClick))
    .subscribe(clicks =>
      clicks.length === 1 ? single() : double()
    );
}
```

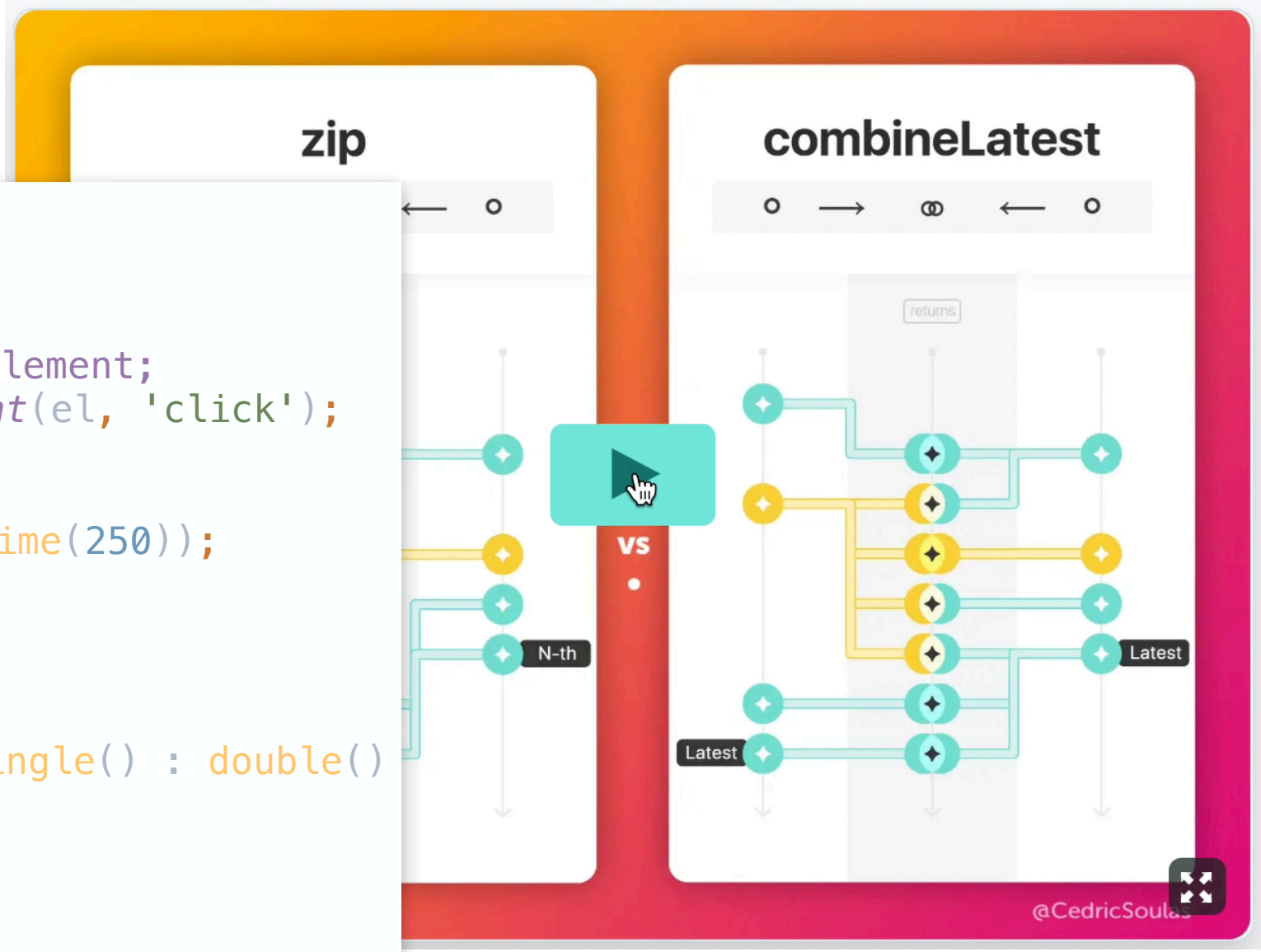
Example stream

```
@ViewChild('button') button;

ngAfterViewInit() {
  let el = this.button.nativeElement;
  const clickStream = fromEvent(el, 'click');

  const finalClick =
    clickStream.pipe(debounceTime(250));

  clickStream
    .pipe(buffer(finalClick))
    .subscribe(clicks =>
      clicks.length === 1 ? single() : double()
    );
}
```



RxJS & Angular

RxJS in Angular

Angular Http uses Observables

```
private http: HttpClient;

this.http
        .get<Contact[]>('/api/contacts')
        .subscribe()
            contacts => {

              // do something with contacts

            }
        );
```

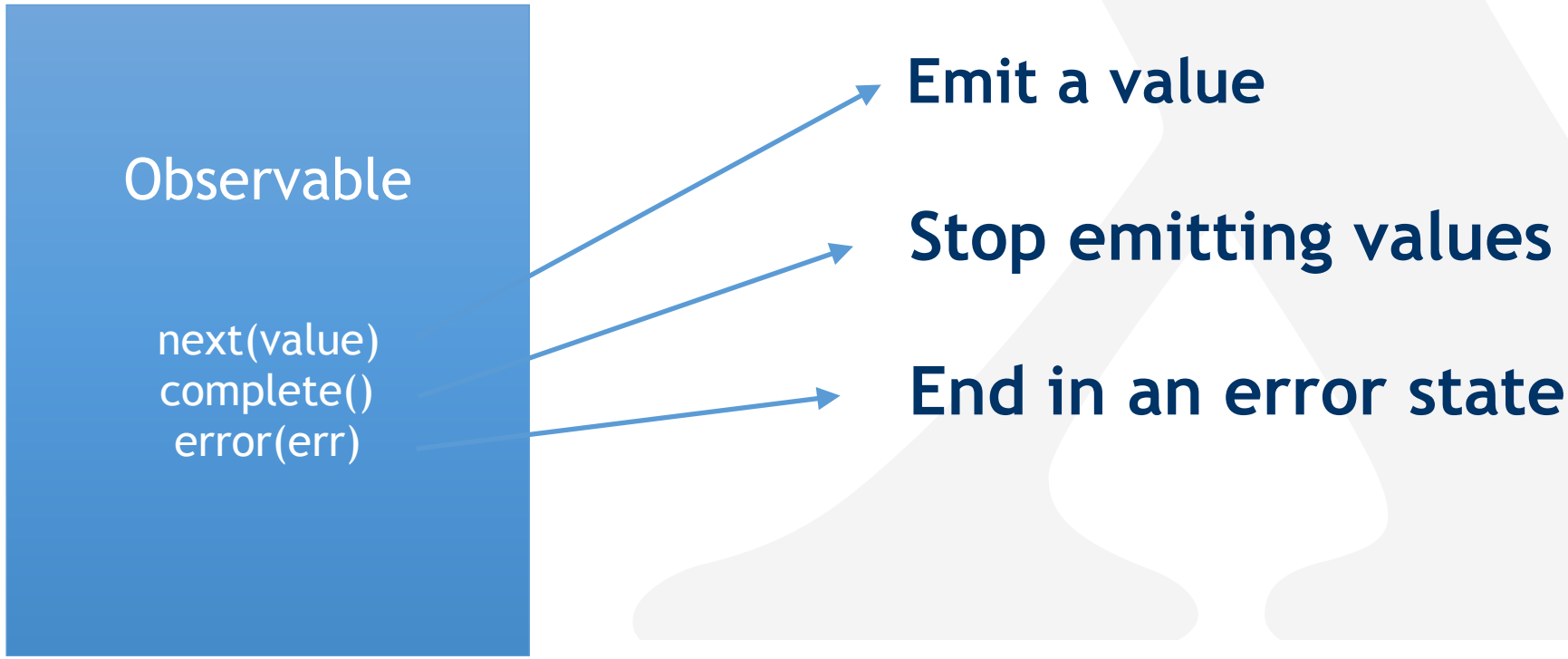
RxJS in Angular

- Angular Http uses Observables
- Change Detection uses Observables
- Router uses Observables

- In Angular, Observables are everywhere!

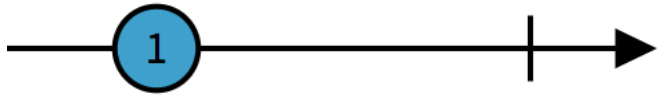
Observable

Observable: something that emits events



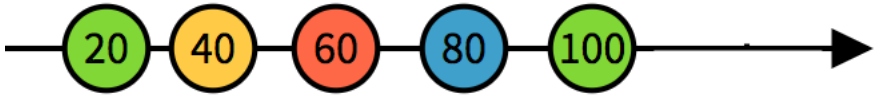
Example: Http <needs legend>

- An Http call
  - emits a single value
  - then completes
  - or ends in an error
- No need to unsubscribe!



Example: Router parameters

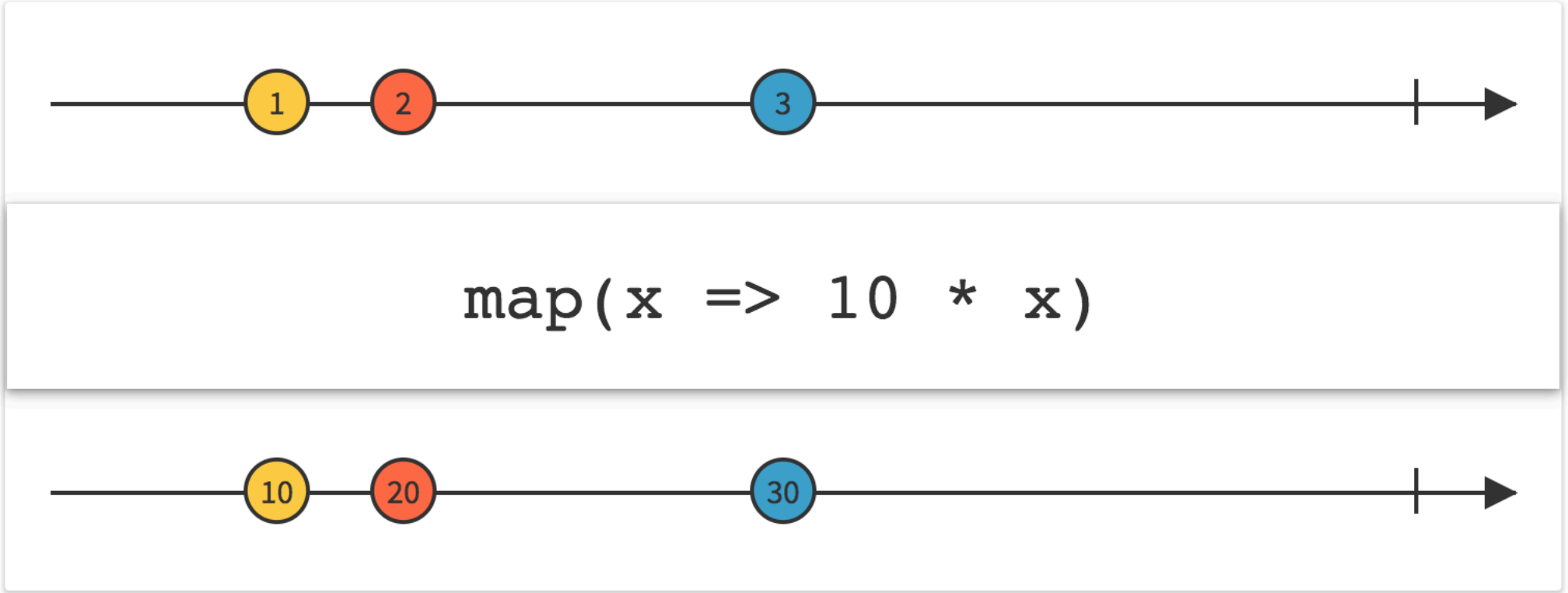
- Router parameters
- keeps emitting values
- unsubscribed by Angular



More Observables

- fromEvent(htmlElement, ‘event’)
- Subject
- BehaviorSubject
- ReplaySubject
- Manual unsubscribe!

Operators



Operators

- transform an Observable into another
- combine Observables
- operate on Observables

Operators make Observables powerful!

Operator syntax

Pipeable operators (RxJS >= 5.5)

```
this.http.get<Contact>('/who-am-i')
  .pipe(
    map(contact => contact.id),
    mergeMap(id => this.http.get<Details>(`/details/${id}`)),
    tap(details => console.log(details)),
    catchError(error => handle(error))
  )
  .subscribe(details => {
    // do something with contact details
  });
```

Operator syntax

Pipeable operators (RxJS >= 5.5)

```
this.http.get<Contact>('/who-am-i')
  .pipe(
    map(contact => contact.id),
    mergeMap(id => this.http.get<Details>(`/details/${id}`)),
    tap(details => console.log(details)),
    catchError(error => handle(error))
  )
  .subscribe(details => {
    // do something with contact details
  });
```

Operator syntax

map transforms the data going through the sequence

```
this.http.get<Contact>('/who-am-i')
  .pipe(
    map(contact => contact.id),
    mergeMap(id => this.http.get<Details>(`/details/${id}`)),
    tap(details => console.log(details)),
    catchError(error => handle(error))
  )
  .subscribe(details => {
    // do something with contact details
  });
```

Operator syntax

mergeMap changes the Observable

```
this.http.get<Contact>('/who-am-i')
  .pipe(
    map(contact => contact.id),
    mergeMap(id => this.http.get<Details>(`/details/${id}`)),
    tap(details => console.log(details)),
    catchError(error => handle(error))
  )
  .subscribe(details => {
    // do something with contact details
  });
```

Operator syntax

tap is for sideeffects

```
this.http.get<Contact>('/who-am-i')
  .pipe(
    map(contact => contact.id),
    mergeMap(id => this.http.get<Details>(`/details/${id}`)),
    tap(details => console.log(details)),
    catchError(error => handle(error))
  )
  .subscribe(details => {
    // do something with contact details
  });
```

Operator syntax

catchError is called when an Observable throws an error

```
this.http.get<Contact>('/who-am-i')
  .pipe(
    map(contact => contact.id),
    mergeMap(id => this.http.get<Details>(`/details/${id}`)),
    tap(details => console.log(details)),
    catchError(error => handle(error))
  )
  .subscribe(details => {
    // do something with contact details
  });
```

Operator syntax

subscribe() executes the sequence

```
this.http.get<Contact>('/who-am-i')
  .pipe(
    map(contact => contact.id),
    mergeMap(id => this.http.get<Details>(`/details/${id}`)),
    tap(details => console.log(details)),
    catchError(error => handle(error))
  )
  .subscribe(details => {
    // do something with contact details
  });
```

TakeUntil

terminate the observable when another observable emits/terminates

```
terminateObservable.pipe(
  takeUntil(whenThisEmitsObservable)
);
```

TakeUntil in action

```
destroy = new Subject<boolean>();

ngOnInit() {
  interval(1000)
    .pipe(takeUntil(this.destroy))
    .subscribe(console.log);
}

ngOnDestroy() {
  this.destroy.next(true);
}
```

TakeUntil in action

Create a Subject

```
destroy = new Subject<boolean>();

ngOnInit() {
  interval(1000)
    .pipe(takeUntil(this.destroy))
    .subscribe(console.log);
}

ngOnDestroy() {
  this.destroy.next(true);
}
```

TakeUntil in action

interval creates an Observables that fires every second

```
destroy = new Subject<boolean>();

ngOnInit() {
  interval(1000)
    .pipe(takeUntil(this.destroy))
    .subscribe(console.log);
}

ngOnDestroy() {
  this.destroy.next(true);
}
```

TakeUntil in action

takeUntil "destroy" emits

```
destroy = new Subject<boolean>();

ngOnInit() {
  interval(1000)
    .pipe(takeUntil(this.destroy))
    .subscribe(console.log);
}

ngOnDestroy() {
  this.destroy.next(true);
}
```

TakeUntil in action

When the component is destroyed, emit on destroy

```
destroy = new Subject<boolean>();

ngOnInit() {
  interval(1000)
    .pipe(takeUntil(this.destroy))
    .subscribe(console.log);
}

ngOnDestroy() {
  this.destroy.next(true);
}
```

More operators (120+)

- filter, distinct, elementAt
- join, merge, zip
- buffer, debounce, delay
- skipWhile, takeUntil
- average, count

Recap

- RxJS
- Observable
- Operators
- "Descriptive way of writing behaviour in code"

Code: Double click revisited

```
@ViewChild('button') button;

ngAfterViewInit() {
  const clickStream = fromEvent(this.button.nativeElement, 'click');

  const finalClick = clickStream.pipe(debounceTime(250));

  clickStream
    .pipe(buffer(finalClick))
    .subscribe(clicks =>
      clicks.length === 1 ? single() : double()
    );
}
```

Double click revisited

Observable from HTML click event.

```
@ViewChild('button') button;

ngAfterViewInit() {
  const clickStream = fromEvent(this.button.nativeElement, 'click');

  const finalClick = clickStream.pipe(debounceTime(250));

  clickStream
    .pipe(buffer(finalClick))
    .subscribe(clicks =>
      clicks.length === 1 ? single() : double()
    );
}
```

Double click revisited

Observable from HTML click event.

```
@ViewChild('button') button;

ngAfterViewInit() {
  const clickStream = fromEvent(this.button.nativeElement, 'click');

  const finalClick = clickStream.pipe(debounceTime(250));

  clickStream
    .pipe(buffer(finalClick))
    .subscribe(clicks =>
      clicks.length === 1 ? single() : double()
    );
}
```

Double click revisited

Second Observable: emits after 250ms

```
@ViewChild('button') button;

ngAfterViewInit() {
  const clickStream = fromEvent(this.button.nativeElement, 'click');

  const finalClick = clickStream.pipe(debounceTime(250));

  clickStream
    .pipe(buffer(finalClick))
    .subscribe(clicks =>
      clicks.length === 1 ? single() : double()
    );
}
```

Double click revisited

buffer(): collect all events until "finalClick" emits

```
@ViewChild('button') button;

ngAfterViewInit() {
  const clickStream = fromEvent(this.button.nativeElement, 'click');

  const finalClick = clickStream.pipe(debounceTime(250));

  clickStream
    .pipe(buffer(finalClick))
    .subscribe(clicks =>
      clicks.length === 1 ? single() : double()
    );
}
```