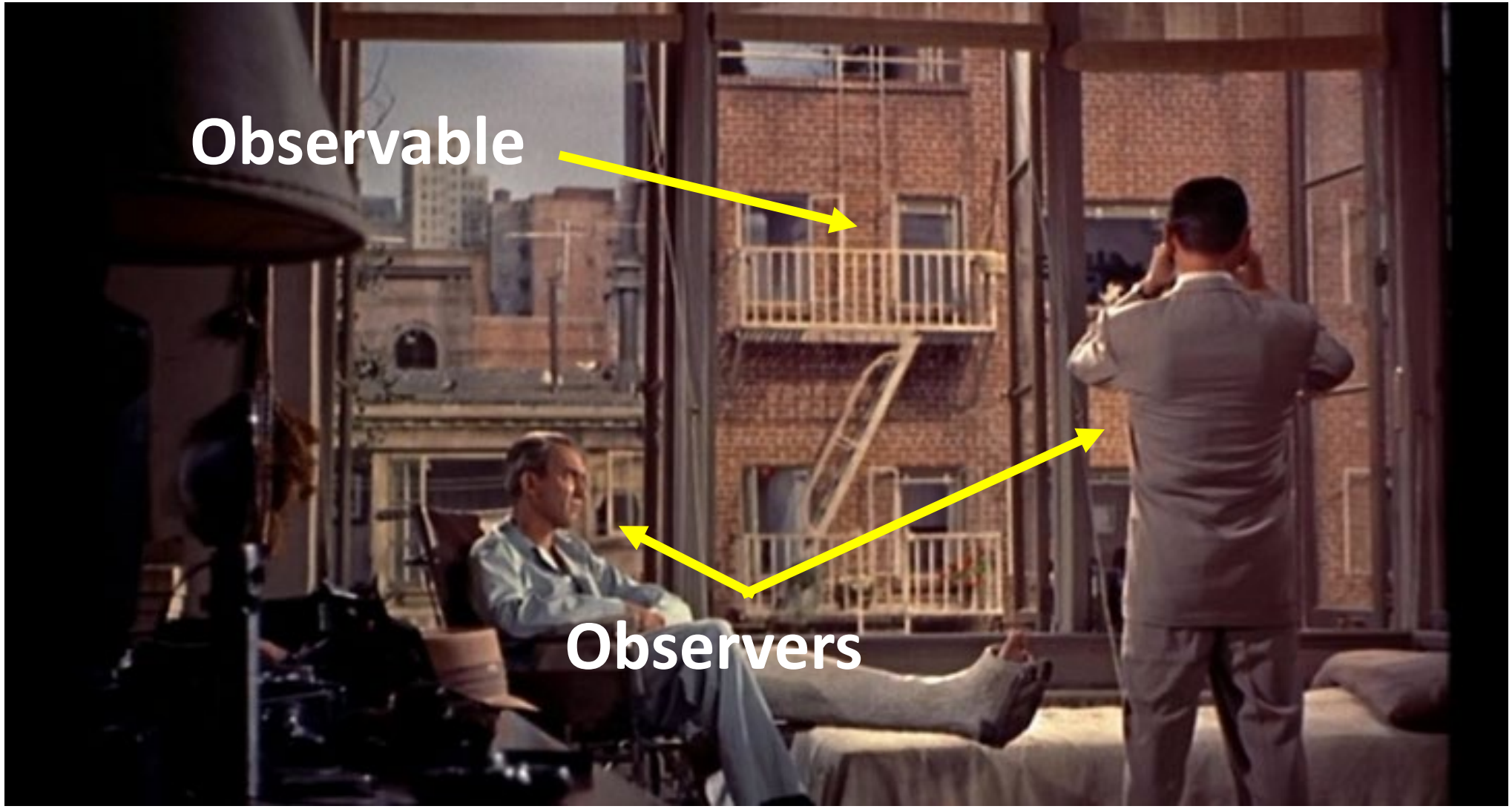
Day 37

Observer and Observable



From the movie “Rear Window”

Observable – EventEmitter

Observable

```
@Output() timesUp: EventEmitter<string> =
      new EventEmitter<string>();
```

```
this.timesUp.emit(this.message);
```

Data

Observer

```
<app-timer [duration]="10"
    (timesUp)="handleTimesUp($event)">
</app-timer>
```

```
handleTimesUp(msg: string) {
    console.log("timesup message: %s", msg);
  }
```

Observable

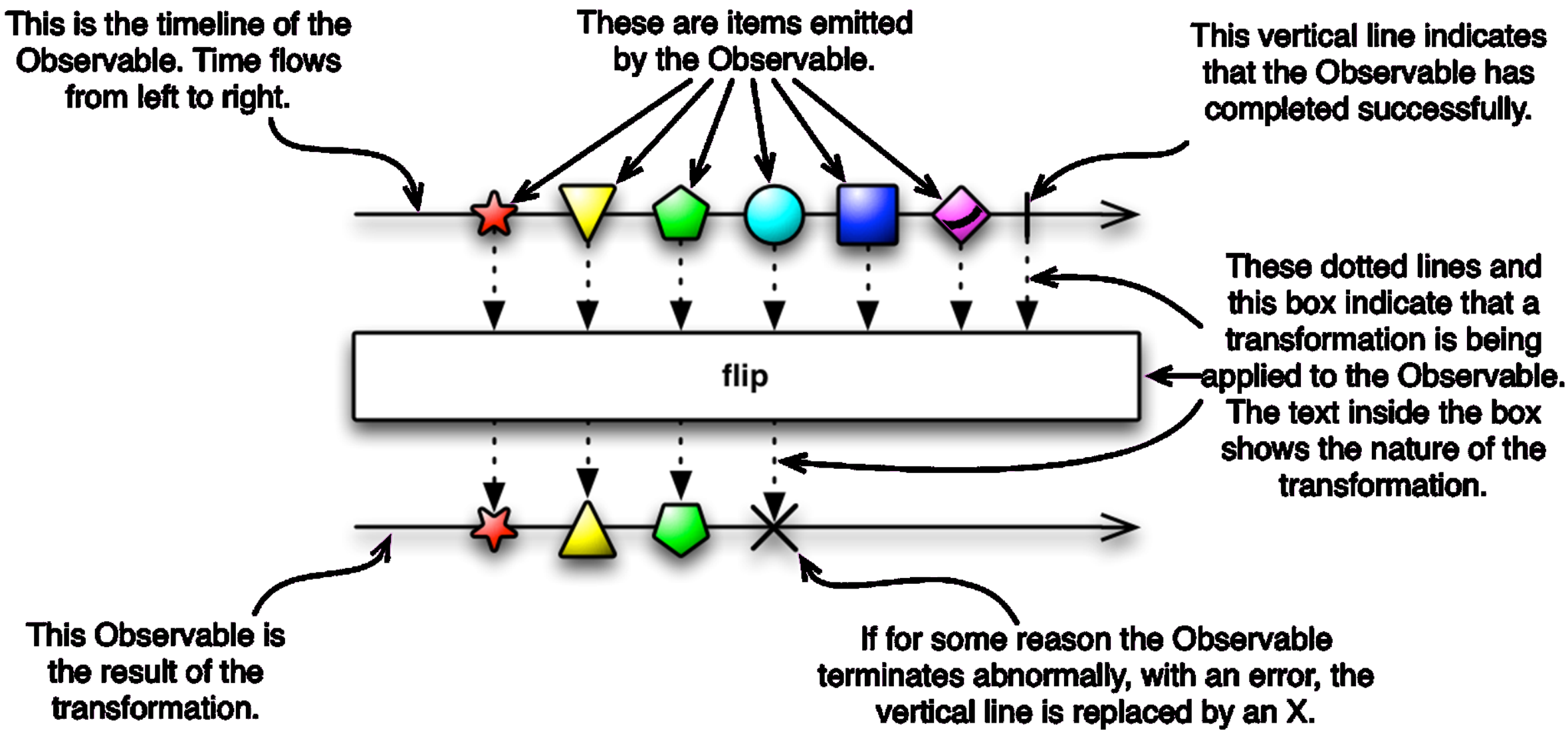


Image from http://reactivex.io/documentation/observable.html

Observable

- Observable is an abstraction of something that is of interest
- Three channels
  - Data channel – generating a stream of data
    - May or may not be a regular intervals
    - Eg. button clicks, stock updates,
  - Error channel – notifies when there are errors at the source
    - Eg. corrupted data
    - Eg. network connection failed at data source
  - Complete channel – no more data when the event/data source closes
    - Eg. no more stock update at the close of the day
    - Eg. no more button clicks because user have closed the application

`EventEmitter`

- `EventEmitter` is an observable
  - Subclass of `Subject` from rxjs module
    - https://reactive.io/rxjs
- `emit()` send data down the data channel
- `subscribe()` to get access to all the channels
  - Pass 3 callbacks: data, error and complete respectively
- `next()` – similar to emit()
- `error()` – notify the observer that there is an error
- `complete()` – the observable is closed. No more data will be sent

Observable Example

Subscribing to an observable

Data

Error

Complete

```
eventSource
  .subscribe(
    result => {
      console.info(`result = `, result);
    },
    error => {
      console.error(`error = `, error);
    },
    () => { /* never execute */ }
  )
```

EventEmitter

```
observable: EventEmitter<string>
  = new EventEmitter<string>()
```

Observable

```
observable.next(data);
observable.error(error);
observable.complete();
```

Observer

```
observable.subscribe(
  (data) => { ... },
  (error) => { ... },
  () = { ... }
)
```

Observable Operators

- Operators - operations to apply to events from an observable
  - `map` - transform an input event to a different output event by applying a function to the input
  - `tap` - like map but returns the same output
  - `take` - only take a specified number of events from a stream
  - `toPromise` - convert an event to a promise
- `pipe` is used to chain operators into a pipeline
  - Operators used in a pipe returns an `Observable`
  - `toPromise` operator cannot be used in a pipe because it does not return a ~~promise~~ observable
- Import operators from `rxjs/operators`

Operators Example

```
import { take, map, toArray, toPromise} from 'rxjs/operators';

eventSource
  .pipe(
    take(1000),
    map(v => v * 2),
    toArray()
  )
  .toPromise()
  .then(result => { console.info('result = ', result); })
  .catch(error => { console.error('error = ', error); });
```

Create a pipeline for operators to manipulate the data

Operators in a pipeline. Order is from left to right, top to bottom

Returns a Promise, so used outside of pipeline

Observable Classes

- `Observable` - different types of observable that produces event stream from different sources
  - `from` - from an array, a promise
  - `fromEvent` - from DOM event
  - Import `Observable` from `rxjs`
- Observable starts when there is an active subscription
- A subscriber gets 3 pieces of information
  - A continuous stream of data
  - Error. When an error is encountered the observable will stop emitting events/data
  - Complete - when the observable is no longer emitting events/data

Converting from Promise to Observable

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/fromPromise';

const promise = new Promise((resolve, reject) => {
  //resolve or reject here
});


Observable.fromPromise(promise)
  .subscribe(
    (data) => { /* resolve */ },
    (error) => { /* reject */ }
  )
```

Creates an observer from a promise. Observer will only fire once

Converting from Observable to Promise

```
import { EventEmitter } from '@angular/core';
import 'rxjs/add/operator/toPromise';
import 'rxjs/add/operator/take';

const event = new EventEmitter<any>();
event.next(data);
event.error(error);

event
    .pipe(take(1))
    .toPromise()
    .then((data) => { /* data */ })
    .catch((error) => { /* error */ })
```

Convert an observable to a promise

Differences between Observable and Promise

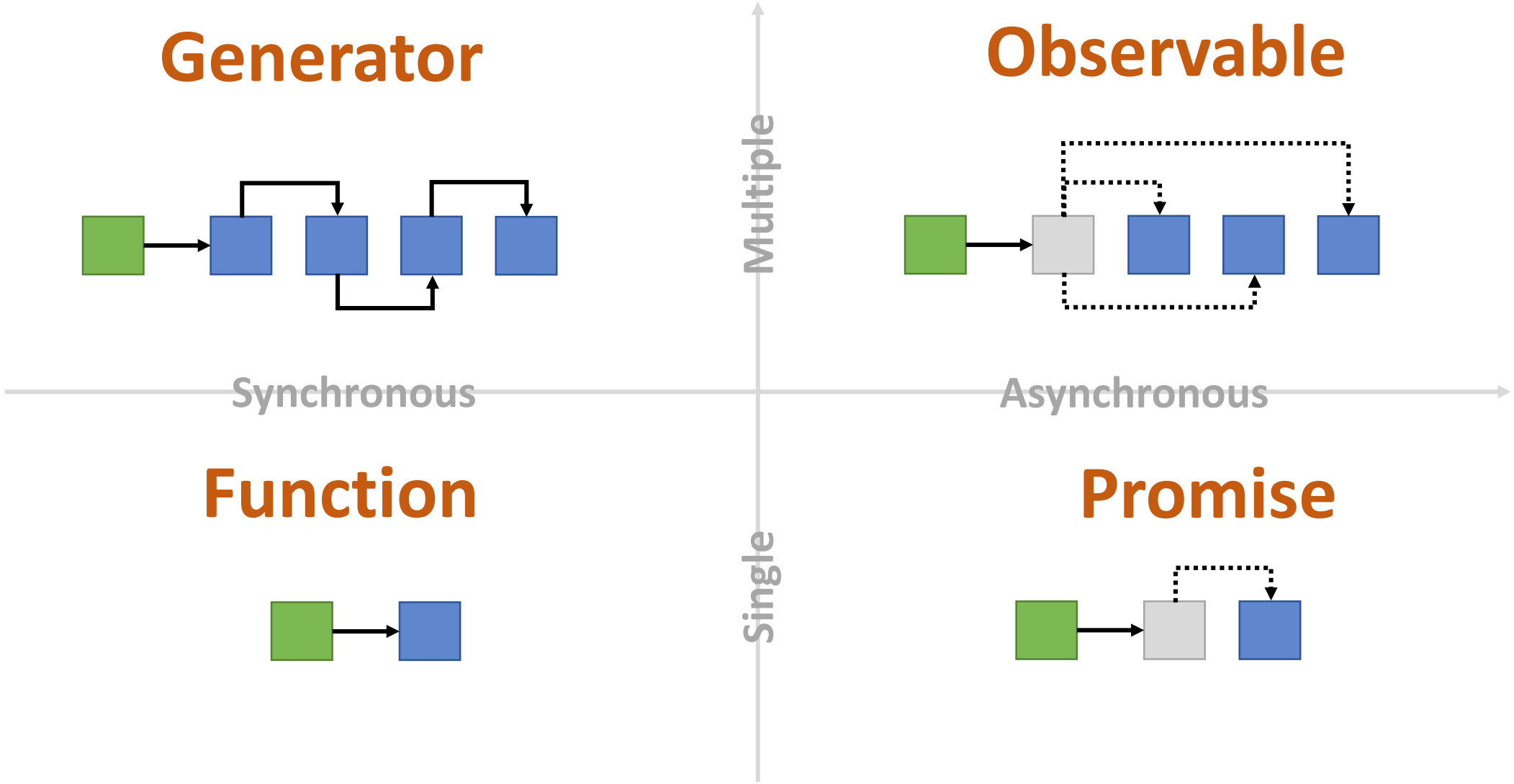
Observable

- Emit multiple values over a period of time
- Observables will not emit any events until there are subscribers
  - `subscribe()`
- Can be cancelled
  - `unsubscribe()`
- Event streams can be operated on and modified - eg. take, reduce, filter, etc.

Promise

- Emit only a single value
- Will emit the even with or without a subscriber
  - `then()`
- Cannot be cancelled
- No operators to modify the event

Differences between Observable and Promise



Redrawn from https://www.sitepoint.com/functional-reactive-programming-rxjs/

WebSocket on the Client

```
const ws = new WebSocket('ws://localhost:3000/chat/fred');

ws.onmessage = (data: any) => {
   console.log(data);
}

ws.send('hello server')
```

Connect to the WebSocket endpoint

ws connects the client to the endpoint

```
app.ws('/chat/:name', (ws, req) => {

  ws.on('message', (data) => {
    console.log(data)
  });

  ws.send('hello client')
})
```

Chat Service Example - 1

```
@Injectable()
export class ChatService {
   private ws: WebSocket;
   chatEmitter: EventEmitter<any>;

   connect(endpoint: string): EventEmitter<any> {
      this.ws = new WebSocket(endpoint);
      this.chatEmitter = new EventEmitter<any>();
      this.ws.onmessage = (data: any) => { this.chatEmitter.next(data); }
      this.ws.onerror = (error: any) => { this.chatEmitter.error(error); }
   }
   disconnect() {
      this.chatEmitter.complete();
      this.ws.close();
   }
   send(data: string) {
      this.ws.send(data);
   }
}
```

Notify subscribers of data or error event

Notify subscribers that we are closing the WebSocket connection

Don’t forget to provide the service

Chat Service Example - 2

```
export class AppComponent implements OnInit, OnDestroy {

  chatSub$: Subscription;

  constructor(private chatSvc: ChatService) { }

  ngOnInit() {
    this.chatSvc.connect('ws://...');
    this.chatSub$ = this.chatSvc.subscribe(
      (data: any) => { console.log(data); },
      (error: any) => { console.error(error); },
      () => { this.chatSub$.unsubscribe(); }
    );
  }
  ngOnDestroy() {
    this.chatSub$.unsubscribe();
  }
}
```

Subscription is the 'handle' to our subscription. Convention is to suffix with $

Need to unsubscribe to free resources