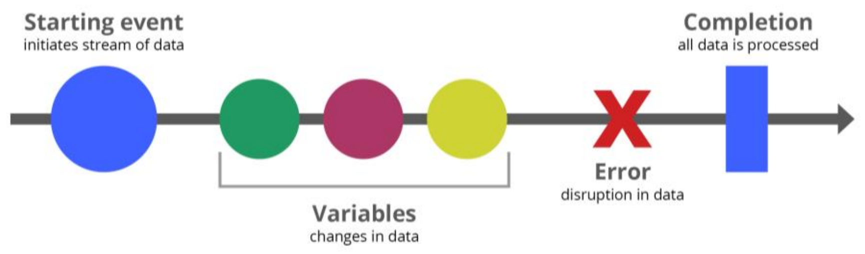
Angular 14 - 13

RxJS and Angular

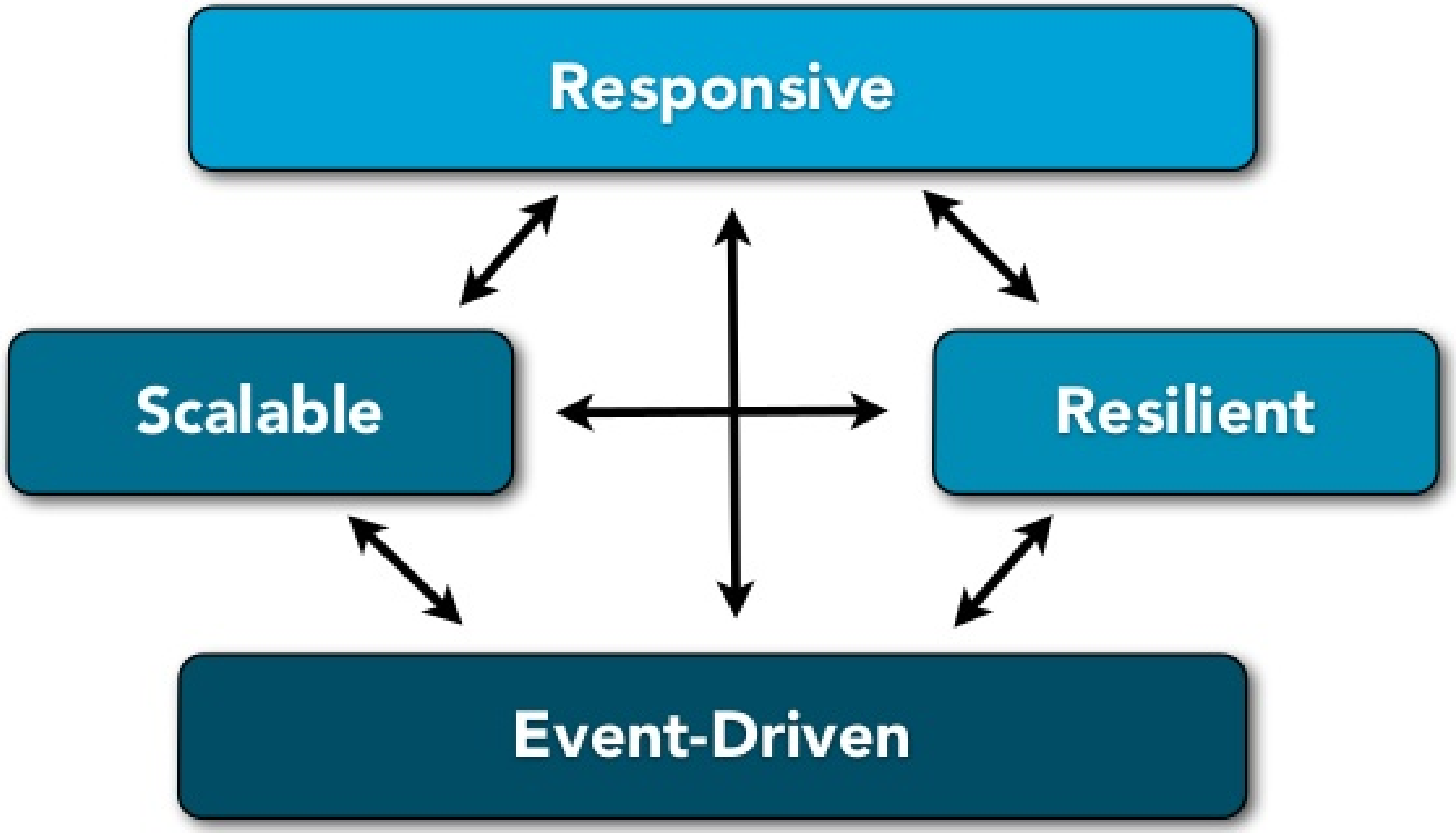
Reactive programming



"Reactive programming is programming with asynchronous data streams"

- Reactive Programming is a **microarchitecture style and paradigm** that involves the intelligent routing and consumption of events, all combined, to change the behavior of an application.
  - https://reactivex.io/
- In a way, this is nothing new. **Event buses** or typical events, like clicks, are actually a sequence of asynchronous events, which can be observed and thus produce some side effects accordingly.
- You can create streams of anything, not just events
  - In reality, streams are cheap and ubiquitous.
  - Anything can be a stream: variables, user input, properties, caches, data structures, etc.
  - For example, viewing a Twitter feed is a stream of data in the same way as click events; you can listen to that broadcast and react accordingly.

The Reactive Manifesto



https://www.reactivemanifesto.org/

The Reactive Manifesto

- Published in 2014, its goal is to condense knowledge on how to **design highly scalable and reliable applications** with a set of architecture features, and define a common vocabulary to enable communication between all participants in this process: developers, architects, project leaders. , engineering management, CTOs.
- It proposes that systems built as reactive systems are more **flexible, loosely coupled, and scalable**.
- This makes them **easier to develop and evolve**. They are significantly more tolerant to failure, and when it does occur, they satisfy it with elegance rather than disaster.
- Reactive systems are **highly responsive** and provide users with effective interactive feedback.
- Therefore, it proposes to build **Responsive, Resilient, Elastic and Message-Guided** systems.

Components of a Reactive library

- Put simply, in Rx programming, the data **streams emitted** by a component and the underlying structure provided by the Rx libraries will **propagate those changes** to another component that is registered to receive those data changes.
- The requirements of a library that handles the entire asynchronous process would be:
- **Execution**
- **Easy thread management**
- **Easily combinable**
- **Minimal Side Effects**

Rx Components

RX = OBSERVABLE + OBSERVER + SCHEDULER

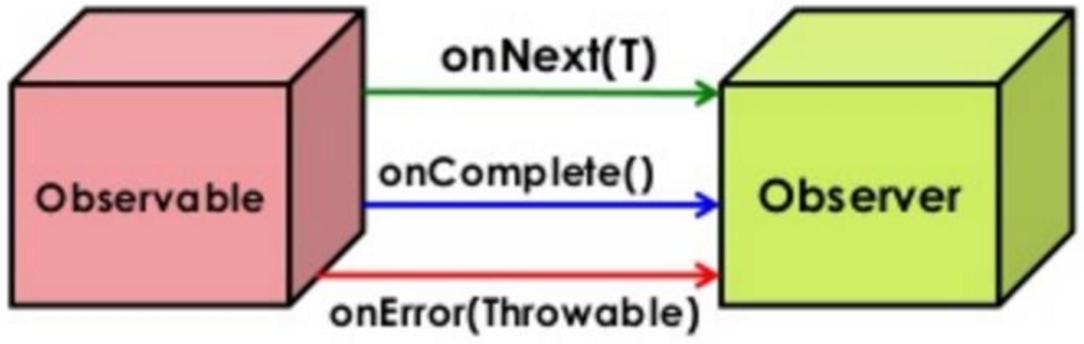
Observables:

- Observables are the data streams.
- Observable packages data that can be passed from one thread to another.
- They basically emit the data periodically or just once in their life cycle depending on your settings.
- There are various operators that can help the observer to output some specific data based on certain events (we will discuss them in next parts).
- For now, you can think of observables as providers. They process and supply the data to other components.

Rx Components

Observers:

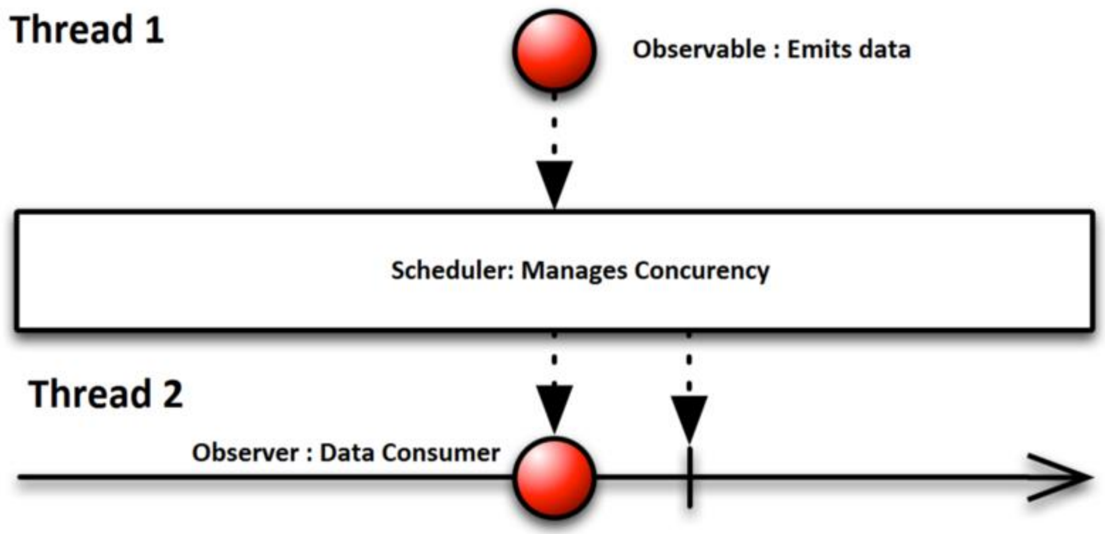
- Observers consume the data stream emitted by the observable.
- Observers subscribe to the observable using the subscribeOn() method to receive the data emitted by the observable.
- Every time the observable emits the data, every registered observer receives the data in the onNext() callback.
- Here they can perform various operations, such as parsing the JSON response or updating the UI.
- If an observable error occurs, the observer will receive it in onError().



Rx Components

Schedulers:

- Rx is used in asynchronous programming and therefore we need a thread manager.
- That's where schedulers come into the picture.
- They are the component that tells the observables and observers, in which thread they should be executed.
- You can use the observeOn() method to tell observers which thread to watch.
- Also, you can use scheduleOn() to tell the observable which thread it should run on.



Observables vs Promises

Promises

- They return a single value
- They are not cancelable
- They can also be used in Angular (there is some example of use in the official documentation).

Observables

- They return multiple values over time.
- can be canceled
- They support modern operators like map , reduce , filter , retry , etc.
- They use an architecture similar to Web Sockets.

Use Rx in our applications

- An implementation of the paradigm for different programming languages can be found in the ReactiveX library
  - http://reactivex.io.
- Especially for Javascript there is the RxJS variant
  - https://github.com/ReactiveX/rxjs
- Angular already comes with the RxJS library
  - https://angular.io/guide/rx-library
- This can be used in the implementation of asynchronous processes, especially those that involve communication with APIs.

Use Rx in Angular

- To use RxJs in an Angular component we must import the desired function from the rxjs library

```
import { Component } from '@angular/core';
import { Observable } from 'rxjs';

@Component({
 selector: 'app-stopwatch',
 templateUrl: './stopwatch.component.html'
})

export class StopwatchComponent {
 stopwatchValue: number;
 stopwatchValue$: Observable<number>;

 start() {
    this.stopwatchValue$.subscribe(num =>
       this.stopwatchValue = num
    );
 }
}
```

Observables and Subjects: Observables

Observables have some qualities:

- **Observables are lazy**
    - You can think of lazy observables as newsletters.
    - For each subscriber, a new newsletter is created.
    - These are only sent to subscribed people, and to no one else.

- **Observables can have multiple values over time**
    - If we keep the subscription to the newsletter open, we will receive a new one from time to time.
    - The sender decides when to send it, all we have to do is wait until it arrives directly in our inbox.

Observables and Subjects: Push vs Pull

A key concept to understand when using observables is that observables push.

Pull

- By pulling, the data consumer decides when to get the data from the data producer. The producer does not know when the data will be delivered to the consumer.
- JavaScript functions behave like this. The function is a Data Producer, and the code that calls the function is consuming it by "extracting" a single return value from its call.

Push

- Pushing works the other way around. The data producer decides when the consumer gets the information.
- We can create our own Observables, but for this we need a Subject

Observables and Subjects: Subject

- According to the RxJS documentation:

  "A Subject is a kind of bridge or proxy [...] that acts as an observer and as an observable. Because it's an Observer, it can subscribe to one or more Observables, and because it's an Observable, it can loop through the items it observes by re-emitting them, and it can also emit new items."

  http://reactivex.io/documentation/subject.html

- Therefore a Subject is capable of observing an event and emitting a message; while being able to be observed by another element.

- To define a Subject in Angular we can do it as follows .
  - Observables are usually **prefixed with $** by convention.

```
import { Injectable } from '@angular/core';
import { Subject } from "rxjs/Subject";
import { Model } from './model;

@Injectable()
export class ModelService {
  public $models= new Subject<Model>();
  constructor() { }

  public fetchModels():{
    return $models;
  }
}
```

Observables and Subjects: Subject

- Also, for good performance it is good practice to create a Subscription object when we subscribe to an Observable in the **ngOnInit** method, to be able to unsubscribe when the component is destroyed, in the **ngOnDestroy** method.
  - In this way we will avoid memory leaks.

```
export class ModelComponent implements OnInit, OnDestroy {

  @Input() model: Model;
  private _modelSubscription: Subscription;

  constructor(private modelService: ModelService) { }

  ngOnInit() {
    this.modelSubscription = this.modelService.$visible.subscribe((model: Model) => {
      this.model.isVisible = model.title === this.model.title;
    })
  }

  ngOnDestroy() {
    if (this.modelSubscription) {
      this.i_temSubscription.unsubscribe();
    }
  }
}
```

Rx in Angular

- Observable, Subject etc.

```
import { Observable, Subject } from 'rxjs';
```

- Operators

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

  - https://rxjs.dev/guide/operators

- Functions for creating observables

```
import { of } from 'rxjs';
```

  - https://rxjs.dev/api

The 4 types of observables

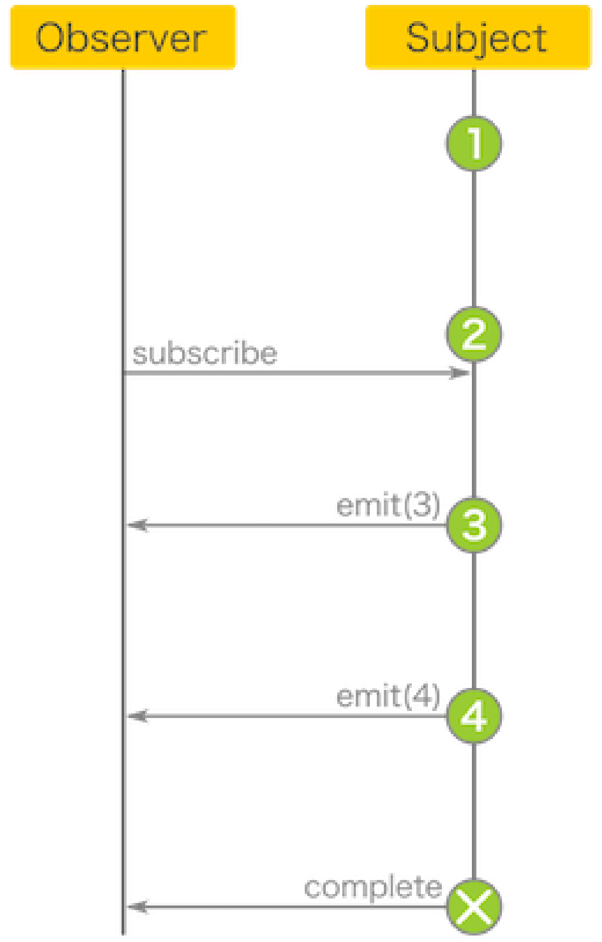
- RxJS provides us with 4 different types of Subject that allow us to create an Observable:
  - **Subject**
  - **BehaviorSubject**
  - **ReplaySubject**
  - **AsyncSubject**

- The main difference between them is the way they respond to a subscription via **subscribe()**.

The 4 types of observables

Subject

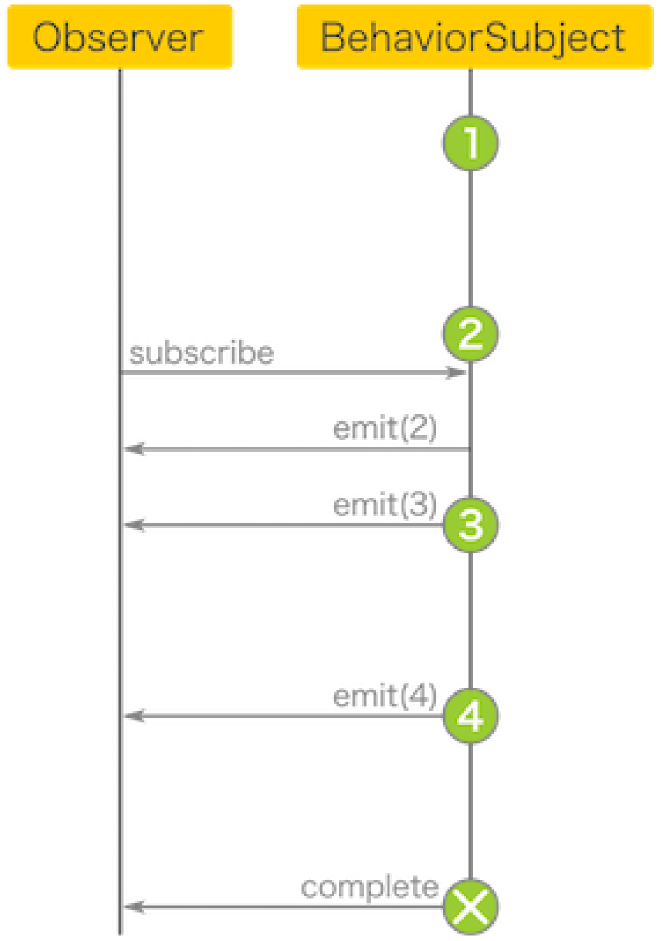
- When we subscribe to a Subject, we get all the **events** that this Subject emits **after being subscribed** as shown in the figure.
- That is, with Subject, the observer will never be notified of event 1 and 2



The 4 types of observables

BehaviorSubject

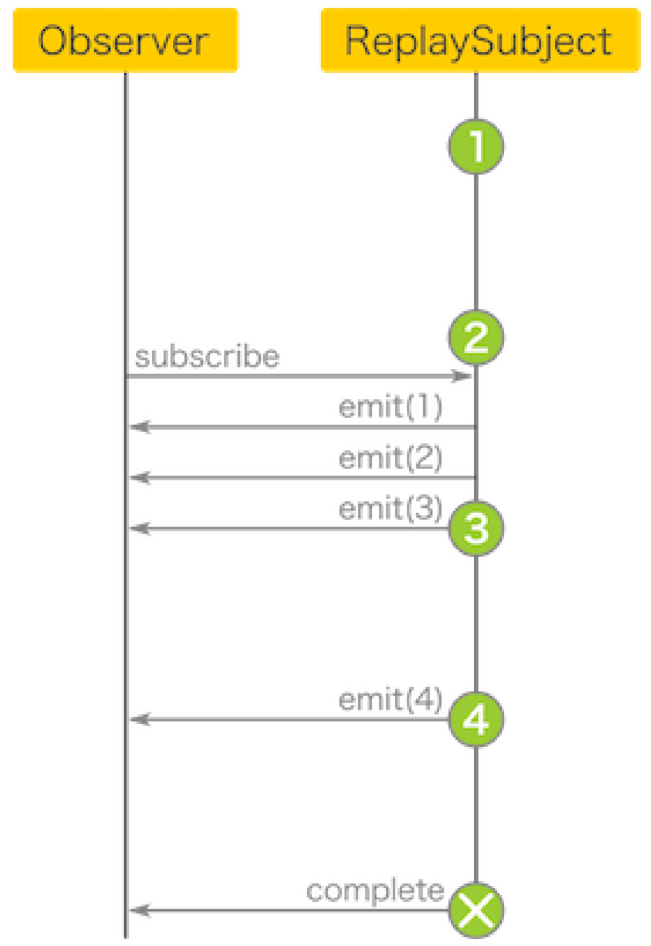
- A BehaviorSubject behaves like a Subject, except that the Observer **also receives the last event** that occurred before the subscription.
- It then receives all the events that occur after the subscription, just like for a regular Subject.
- In this case, the watcher will be notified of event 2 when they subscribe, but not of event 1.



The 4 types of observables

ReplaySubject

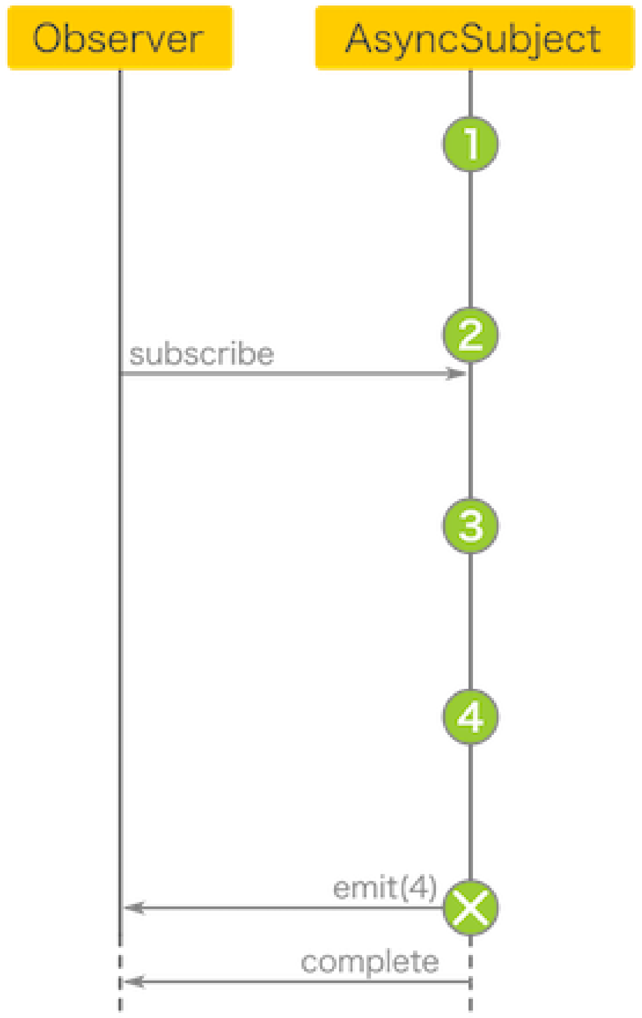
- With a ReplaySubject, the Observer receives **all past events when it subscribes**.
- It then receives all the events that occur after the subscription, just like for a regular subject.
- The observer will know all the previous events, i.e. 1 and 2.



The 4 types of observables

AsyncSubject

- It has a peculiar behavior. AsyncSubject **will wait to complete itseelf to emit the last event** and then the **complete event**.
- The observer will only be notified of event 4 (and complete)



Let's put it into practice: Tasks/Projects App

1. Create services using a BehaviorSubject to deliver data to components, especially when adding new tasks and projects.

Next steps

We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

Thanks!

Follow us: