3.- Relación entre componentes

Repasaremos las opciones más importantes que tienen los componentes de Angular para comunicarse entre ellos.

3.1.- Comunicación padre-hijo a través de la etiqueta *@input*

Tenemos nuestra clase 'Padre':

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

import { ArrayObjects } from './classObject';

@Component({
  selector: 'app-parent',
  template: `
    <h2>{{master}} controls {{arrayObjects.length}} numbers</h2>
    <app-child *ngFor="let element of arrayObjects"
      [objectChild]="element"
      [master]="master">
    </app-child>
  `
})
export class ParentComponent {
  arrayObjects = ArrayObjects;
  master = 'Master';
}
```

Y nuestra clase 'Hijo':

```
import { Component, Input } from '@angular/core';

import { Object } from './classObject';

@Component({
  selector: 'app-child',
  template: `
    <h3>{{objectChild.name}} says:</h3>
    <p>I, {{objectChild.name}}, am at your service, {{masterName}}.</p>
  `
})
export class ChildComponent {
  @Input() objectChild: Object;
  @Input('master') masterName: string;
}
```

Como se ha podido comprobar la forma que tienen estos dos componentes es a través de la etiqueta @Input pero ¿qué quiere decir esto? Cuando colocamos esta etiqueta a un componente se

mantiene a la escucha de alguna variable que cumpla con los requisitos puestos, en este caso el primero es de una clase Object y el segundo es un string.

Una de las maneras de pasar al componente la información necesaria para recogerla en los @Input es colocando dicha información como atributo de la etiqueta del componente, como se vé en la etiqueta de ChildComponent dentro del HTML del padre.

3.2.- Comunicación entre componentes con *ngOnChanges()*

Tenemos nuestra clase 'Padre':

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

@Component({
  selector: 'app-version-parent',
  template: `
    <h2>Source code version</h2>
    <button (click)="newMinor()">New minor version</button>
    <button (click)="newMajor()">New major version</button>
    <app-version-child [major]="major" [minor]="minor"></app-version-child>
  `
})
export class VersionParentComponent {
  major = 1;
  minor = 23;

  newMinor() {
    this.minor++;
  }

  newMajor() {
    this.major++;
    this.minor = 0;
  }
}
```

Y tenemos nuestra clase 'Hijo':

```
import { Component, Input, OnChanges, SimpleChange } from '@angular/core';

@Component({
  selector: 'app-version-child',
  template: `
    <h3>Version {{major}}.{{minor}}</h3>
    <h4>Change log:</h4>
    <ul>
      <li *ngFor="let change of changeLog">{{change}}</li>
    </ul>
  `
})
export class VersionChildComponent implements OnChanges {
  @Input() major: number;
  @Input() minor: number;
  changeLog: string[] = [];
```

```
  ngOnChanges(changes: {[propKey: string]: SimpleChange}) {
    let log: string[] = [];
    for (let propName in changes) {
      let changedProp = changes[propName];
      let to = JSON.stringify(changedProp.currentValue);
      if (changedProp.isFirstChange()) {
        log.push(`Initial value of ${propName} set to ${to}`);
      } else {
        let from = JSON.stringify(changedProp.previousValue);
        log.push(`${propName} changed from ${from} to ${to}`);
      }
    }
    this.changeLog.push(log.join(', '));
  }
}
```

En este ejemplo se mantiene la misma forma de comunicación entre padre-hijo a través de la etiqueta @Input pero ahora añadimos la función *ngOnChanges()*. Esta función se implementa al crear la clase igual que el ngOnInit() y actúa cada vez que se detecten cambios en las directivas o atributos de dicho componente.

3.3.- Comunicación Hijo-Padre a través de eventos

Tenemos nuestra clase 'Padre':

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

@Component({
  selector: 'app-vote-taker',
  template: `
    <h2>Should mankind colonize the Universe?</h2>
    <h3>Agree: {{agreed}}, Disagree: {{disagreed}}</h3>
    <app-voter *ngFor="let voter of voters"
      [name]="voter"
      (voted)="onVoted($event)">
    </app-voter>
  `
})
export class VoteTakerComponent {
  agreed = 0;
  disagreed = 0;
  voters = ['Narco', 'Celeritas', 'Bombasto'];

  onVoted(agreed: boolean) {
    agreed ? this.agreed++ : this.disagreed++;
  }
}
```

Y tenemos nuestra clase ‘Hijo’:

```
import { Component, EventEmitter, Input, Output } from '@angular/core';

@Component({
  selector: 'app-voter',
  template: `
    <h4>{{name}}</h4>
    <button (click)="vote(true)"  [disabled]="didVote">Agree</button>
    <button (click)="vote(false)" [disabled]="didVote">Disagree</button>
  `
})
export class VoterComponent {
  @Input()  name: string;
  @Output() voted = new EventEmitter<boolean>();
  didVote = false;

  vote(agreed: boolean) {
    this.voted.emit(agreed);
    this.didVote = true;
  }
}
```

En este ejemplo el padre ‘escucha’ el evento del hijo a través del Objeto EventEmitter. Este objeto lanza el evento, en la etiqueta de dicho componente debemos colocar un atributo con el mismo nombre para capturarlo e igualarlo a alguna función del componente padre para que la lance al escuchar el evento. Para esta funcionalidad se debe de usar el objeto *@Output()* de Angular.

3.4.- Acceso componente Padre a variables del componente Hijo

Tenemos al componente ‘Padre’:

```
import { Component }                from '@angular/core';
import { CountdownTimerComponent }  from './countdown-timer.component';

@Component({
  selector: 'app-countdown-parent-lv',
  template: `
  <h3>Countdown to Liftoff (via local variable)</h3>
  <button (click)="timer.start()">Start</button>
  <button (click)="timer.stop()">Stop</button>
  <div class="seconds">{{timer.seconds}}</div>
  <app-countdown-timer #timer></app-countdown-timer>
  `,
  styleUrls: ['../assets/demo.css']
})
export class CountdownLocalVarParentComponent { }
```

Y tenemos al componente ‘Hijo’:

```
import { Component, OnDestroy, OnInit } from '@angular/core';

@Component({
  selector: 'app-countdown-timer',
  template: '<p>{{message}}</p>'
})
export class CountdownTimerComponent implements OnInit, OnDestroy {

  intervalId = 0;
  message = '';
  seconds = 11;

  clearTimer() { clearInterval(this.intervalId); }

  ngOnInit()    { this.start(); }
  ngOnDestroy() { this.clearTimer(); }

  start() { this.countDown(); }
  stop()  {
    this.clearTimer();
    this.message = `Holding at T-${this.seconds} seconds`;
  }

  private countDown() {
    this.clearTimer();
    this.intervalId = window.setInterval(() => {
      this.seconds -= 1;
      if (this.seconds === 0) {
        this.message = 'Blast off!';
      } else {
        if (this.seconds < 0) { this.seconds = 10; } // reset
        this.message = `T-${this.seconds} seconds and counting`;
      }
    }, 1000);
  }
}
```

De esta manera estamos vinculando una variable en el ‘Padre’ (#timer) para poder utilizar aquellas funciones o variables públicas del ‘Hijo’.

3.5.- Comunicación entre componentes a través de Services

Con este método podremos comunicar dos componentes sin la necesidad de que sean ‘Padre-Hijo’. Primero crearemos un Service que servirá de comunicación entre ambos componentes.

```
import { Injectable } from '@angular/core';
import { Subject }    from 'rxjs';
```

```
@Injectable()
export class MissionService {

  // Observable string sources
  private missionAnnouncedSource = new Subject<string>();
  private missionConfirmedSource = new Subject<string>();

  // Observable string streams
  missionAnnounced$ = this.missionAnnouncedSource.asObservable();
  missionConfirmed$ = this.missionConfirmedSource.asObservable();

  // Service message commands
  announceMission(mission: string) {
    this.missionAnnouncedSource.next(mission);
  }

  confirmMission(astronaut: string) {
    this.missionConfirmedSource.next(astronaut);
  }
}
```

Padre:

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

import { MissionService }     from './mission.service';

@Component({
  selector: 'app-mission-control',
  template: `
    <h2>Mission Control</h2>
    <button (click)="announce()">Announce mission</button>
    <app-astronaut *ngFor="let astronaut of astronauts"
      [astronaut]="astronaut">
    </app-astronaut>
    <h3>History</h3>
    <ul>
      <li *ngFor="let event of history">{{event}}</li>
    </ul>
  `,
  providers: [MissionService]
})
export class MissionControlComponent {
  astronauts = ['Lovell', 'Swigert', 'Haise'];
  history: string[] = [];
  missions = ['Fly to the moon!',
              'Fly to mars!',
              'Fly to Vegas!'];
  nextMission = 0;

  constructor(private missionService: MissionService) {
    missionService.missionConfirmed$.subscribe(
      astronaut => {
        this.history.push(`${astronaut} confirmed the mission`);
```

```
    });
  }

  announce() {
    let mission = this.missions[this.nextMission++];
    this.missionService.announceMission(mission);
    this.history.push(`Mission "${mission}" announced`);
    if (this.nextMission >= this.missions.length) { this.nextMission = 0; }
  }
}
```

Hijo:

```
import { Component, Input, OnDestroy } from '@angular/core';

import { MissionService } from './mission.service';
import { Subscription }   from 'rxjs';

@Component({
  selector: 'app-astronaut',
  template: `
    <p>
      {{astronaut}}: <strong>{{mission}}</strong>
      <button
        (click)="confirm()"
        [disabled]="!announced || confirmed">
        Confirm
      </button>
    </p>
  `
})
export class AstronautComponent implements OnDestroy {
  @Input() astronaut: string;
  mission = '<no mission announced>';
  confirmed = false;
  announced = false;
  subscription: Subscription;

  constructor(private missionService: MissionService) {
    this.subscription = missionService.missionAnnounced$.subscribe(
      mission => {
        this.mission = mission;
        this.announced = true;
        this.confirmed = false;
    });
  }

  confirm() {
    this.confirmed = true;
    this.missionService.confirmMission(this.astronaut);
  }

  ngOnDestroy() {
    // prevent memory leak when component destroyed
    this.subscription.unsubscribe(); }}
```

3.5.1.- ¡¡Importante!!

En el ejemplo anterior, seguimos utilizando la lógica componente Padre-Hijo pues en un componente tenemos el selector del otro componente por lo que ambos se están renderizando y visualizando en la misma vista en el navegador.

Esto es importante entenderlo pues por ese motivo al usar el objeto Subject en el Service me va a mostrar la variable modificada. Para poder mostrar una variable en dos componentes que no se renderizan en la misma vista se debe de cambiar ese objeto por BehaviorSubject de la misma librería. El cambio sería tan solo de esa parte, todo lo demás sigue la misma lógica de uso.

El código a cambiar del ejemplo anterior sería:

```
private missionAnnouncedSource = new Subject<string>();
```

Por:

```
private missionAnnouncedSource = new BehaviorSubject(‘’);
```