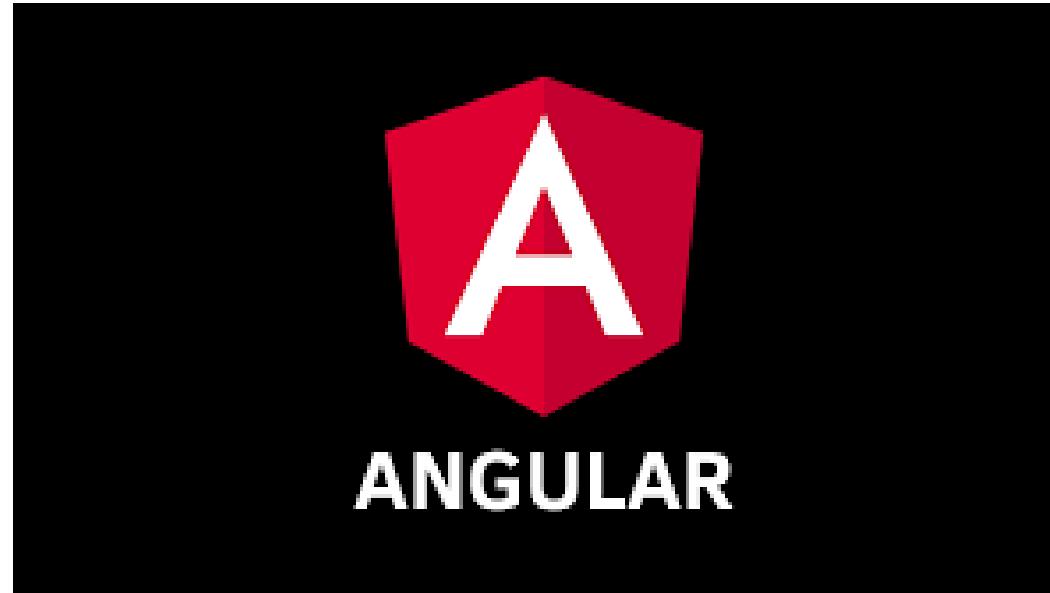




INSTITUTO TÉCNICO  
DE ESTUDIOS  
PROFESIONALES



# ¿Qué es Angular?

es un framework para el desarrollo de aplicaciones web. Está pensado para dividir un proyecto en componentes y ser reutilizadas en proyectos medianos y grandes.

Sus principales competidores son [Vue](#) (proyecto iniciado por Evan You en 2014) y [React](#) ( proyecto iniciado por Facebook en 2013)

Para desarrollar en forma efectiva una aplicación en Angular debemos instalar al menos dos herramientas básicas:  
**Node.js**  
Angular CLI (Command Line Interface - Interfaz de Línea de comandos)



tiene su salida al mercado en 2016 pero tiene una versión previa no compatible y solo mantenida para proyectos antiguos llamada [Angular.js](#)(2010)

El proyecto de Angular es propiedad de la empresa de Google.

## TypeScript Tutorial

- ➊ TypeScript - Home
- ➋ TypeScript - Overview
- ➌ TypeScript - Environment Setup
- ➍ TypeScript - Basic Syntax
- ➎ TypeScript - Types
- ➏ TypeScript - Variables
- ➐ TypeScript - Operators
- ➑ TypeScript - Decision Making
- ➒ TypeScript - Loops
- ➓ TypeScript - Functions
- ➔ TypeScript - Numbers
- ➕ TypeScript - Strings
- ➖ TypeScript - Arrays
- ➗ TypeScript - Tuples
- ➘ TypeScript - Union
- ➙ TypeScript - Interfaces
- ➚ TypeScript - Classes
- ➛ TypeScript - Objects
- ➜ TypeScript - Namespaces
- ➝ TypeScript - Modules
- ➞ TypeScript - Ambients



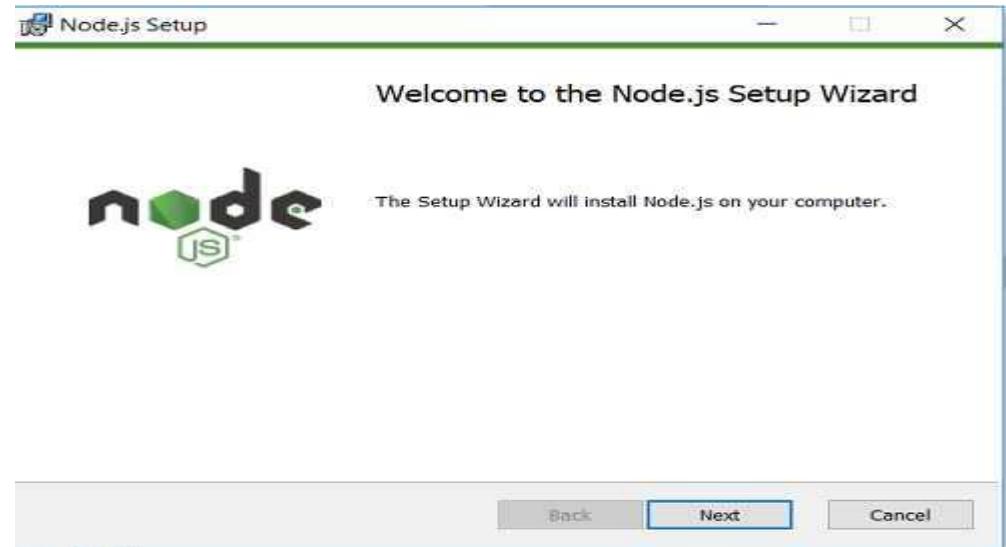
## Angular7 Tutorial

- ➊ Angular7 - Home
- ➋ Angular7 - Overview
- ➌ Angular7 - Environment Setup
- ➍ Angular7 - Project Setup
- ➎ Angular7 - Components
- ➏ Angular7 - Modules
- ➐ Angular7 - Data Binding
- ➑ Angular7 - Event Binding
- ➒ Angular7 - Templates
- ➓ Angular7 - Directives
- ➔ Angular7 - Pipes
- ➕ Angular7 - Routing
- ➖ Angular7 - Services
- ➗ Angular7 - Http Client
- ➘ Angular7 - CLI Prompts
- ➙ Angular7 - Forms
- ➚ Materials/CDK-Virtual Scrolling
- ➛ Angular7 - Materials/CDK-Drag & Drop
- ➜ Angular7 - Animations
- ➝ Angular7 - Materials
- ➞ Testing & Building Angular7 Project

## Instalación de Node.js

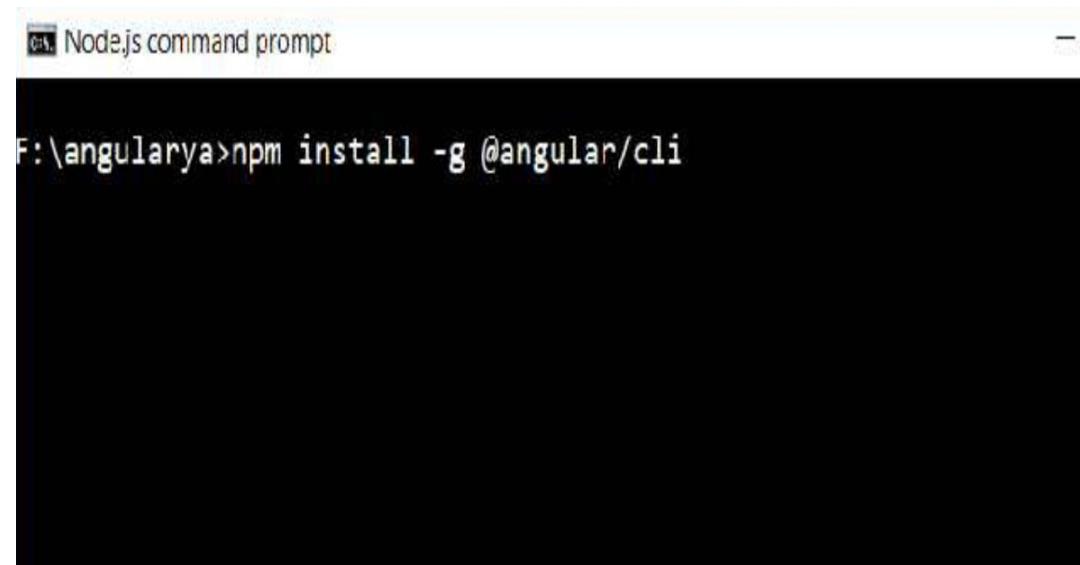
La primer herramienta a instalar será Node.js, esto debido a que gran cantidad de programas para el desarrollo en Angular están implementadas en Node. Debemos Descargar e instalar la última versión estable de [Node.js](#):

Una vez instalado Node.js desde la línea de comandos del mismo podemos comprobar su correcto funcionamiento averiguando su versión:

A screenshot of a 'Node.js command prompt' window. The command 'node -v' is entered, resulting in the output 'v8.9.4'. The prompt ends with a cursor at 'F:\angularaya>'.

## Instalación de Angular CLI

Para instalar este software lo hacemos desde la misma línea de comandos de Node.js (por eso lo instalamos primero), debemos ejecutar el siguiente comando:



A screenshot of a terminal window titled "Node.js command prompt". The command "F:\angularaya>npm install -g @angular/cli" is visible on the screen.

Es importante el -g para que se instale en forma global.

## Creación de un proyecto y prueba de su funcionamiento

Creación de un proyecto y prueba de su funcionamiento

Este comando crea la carpeta proyecto001 e instala una gran cantidad de herramientas que nos auxiliarán durante el desarrollo del proyecto (320mb). Como Angular está pensado para aplicaciones de complejidad media o alta no hay posibilidad de instalar menor herramientas.

El proceso de generar el proyecto lleva bastante tiempo ya que deben descargarse de internet muchas herramientas.

Se genera una aplicación con el esqueleto mínimo, para probarlo debemos descender a la carpeta que se acaba de crear y lanzar el siguiente comando desde Node:

Este comando arranca un servidor web en forma local y abre el navegador para la ejecución de la aplicación.  
En el navegador tenemos como resultado:

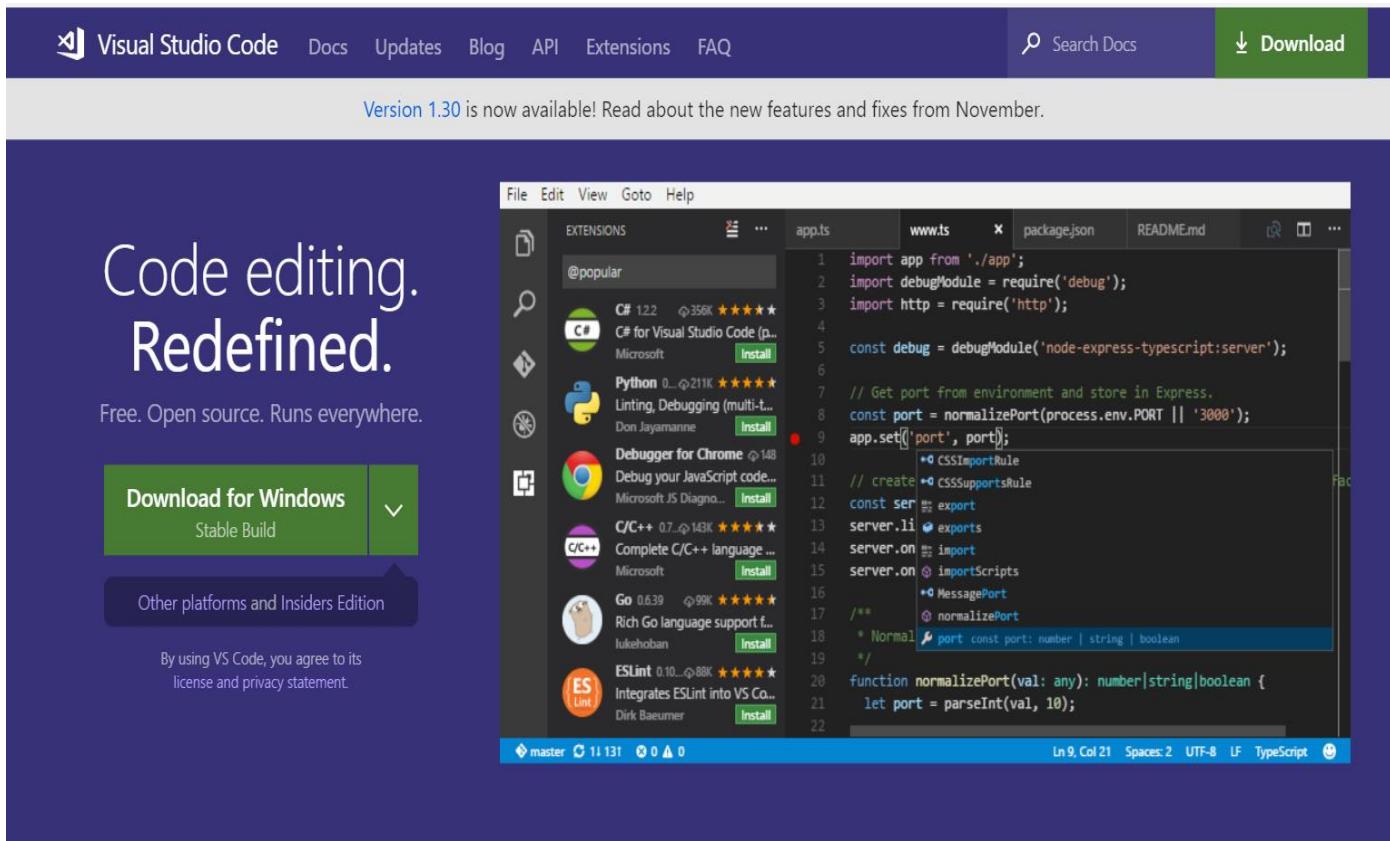
```
Node.js command prompt  
F:\angular\ya>ng new proyecto001  
create proyecto001/e2e/app.e2e-spec.ts (293 bytes)  
create proyecto001/e2e/app.po.ts (208 bytes)  
create proyecto001/e2e/tsconfig.e2e.json (235 bytes)  
create proyecto001/karma.conf.js (923 bytes)  
create proyecto001/package.json (1296 bytes)  
create proyecto001/protractor.conf.js (722 bytes)  
create proyecto001/README.md (1027 bytes)  
create proyecto001/tsconfig.json (363 bytes)  
create proyecto001/tslint.json (3012 bytes)
```

```
cd my-app  
ng serve --open
```



## Creación de un proyecto y prueba de su funcionamiento

Debemos utilizar un editor de texto para codificar la aplicación. Yo recomiendo el [Visual Studio Code](#), puede visitar luego un tutorial completo del editor [VS Code](#)



# Archivos y carpetas básicas de un proyecto en Angular

Vimos en el concepto anterior que para crear un proyecto en Angular utilizamos la herramienta Angular CLI y desde la línea de comandos escribimos:

No haremos por el momento un estudio exhaustivo de todos los archivos y carpetas que se crean (más 29000 archivos y 3700 carpetas en la versión de Angular 6.x), sino de aquellas que se requieren modificar según el concepto que estemos estudiando.

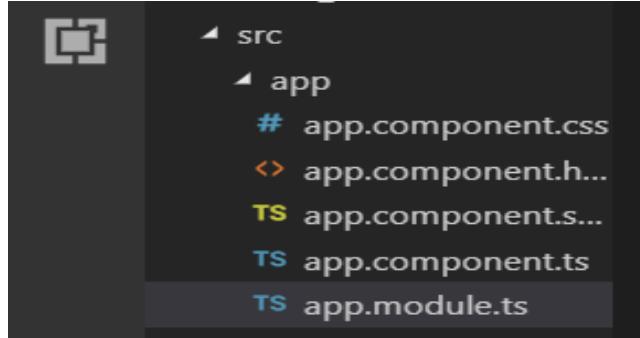
Angular CLI nos crea una única componente llamada 'AppComponent' que se distribuye en 4 archivos:

Todos estos archivos se localizan en la carpeta 'app' y esta carpeta se encuentra dentro de la carpeta 'src':

```
ng new proyecto001
```

En Angular la pieza fundamental es la 'COMPONENTE'. Debemos pensar siempre que una aplicación se construye a base de un conjunto de componentes (por ejemplo pueden ser componentes: un menú, lista de usuarios, login, tabla de datos, calendario, formulario de búsqueda etc.)

```
# app.component.css
<> app.component.h...
TS app.component.s...
TS app.component.ts
TS app.module.ts
```



```
src
  app
    # app.component.css
    <> app.component.h...
    TS app.component.s...
    TS app.component.ts
    TS app.module.ts
```

# Archivos y carpetas básicas de un proyecto en Angular

En Angular se programa utilizando el lenguaje TypeScript que vamos a ir aprendiéndolo a lo largo del curso. El archivo donde se declara la clase AppComponent es 'app.component.ts':

La Clase AppComponent define un atributo llamado 'title' y lo inicializa con el string 'app':

```
8  export class AppComponent {  
9    title = 'cursoangular';  
10 }  
11
```

TS app.component.ts ×

```
1  import { Component } from '@angular/core';  
2  
3  @Component({  
4    selector: 'app-root',  
5    templateUrl: './app.component.html',  
6    styleUrls: ['./app.component.css']  
7  })  
8  export class AppComponent {  
9    title = 'cursoangular';  
10 }  
11
```

Dijimos anteriormente que la clase completa se distribuye en otros archivos y podemos ver que mediante la función decoradora @Component le indicamos los otros archivos que pertenecen a esta componente:

```
2  
3  @Component({  
4    selector: 'app-root',  
5    templateUrl: './app.component.html',  
6    styleUrls: ['./app.component.css']  
7  })
```

El archivo 'app.component.html' tiene la parte visual de nuestra componente 'AppComponent' y está constituido mayormente por código HTML (cada vez que realicemos un proyecto a este código lo borraremos para resolver nuestro problema):

```
 app.component.html x
1  <!--The content below is only a placeholder and can be replaced.-->
2  <div style="text-align:center">
3    <h1>
4      | Welcome to {{ title }}!
5    </h1>
6    Tour of Heroes</a></h2>
12   </li>
13   <li>
14     | <h2><a target="_blank" rel="noopener" href="https://github.com/angular/angular-cli/wiki">CLI Documentation</a></h2>
15   </li>
16   <li>
17     | <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
18   </li>
19 </ul>
20
21
```

Lo único que analizaremos ahora de este trozo de HTML es donde aparece el atributo 'title' de la componente:

```
3  <h1>
4    | Welcome to {{ title }}!
5  </h1>
```

Cuando ejecutamos nuestra aplicación desde la línea de comandos de Node.js:



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.17134.523]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

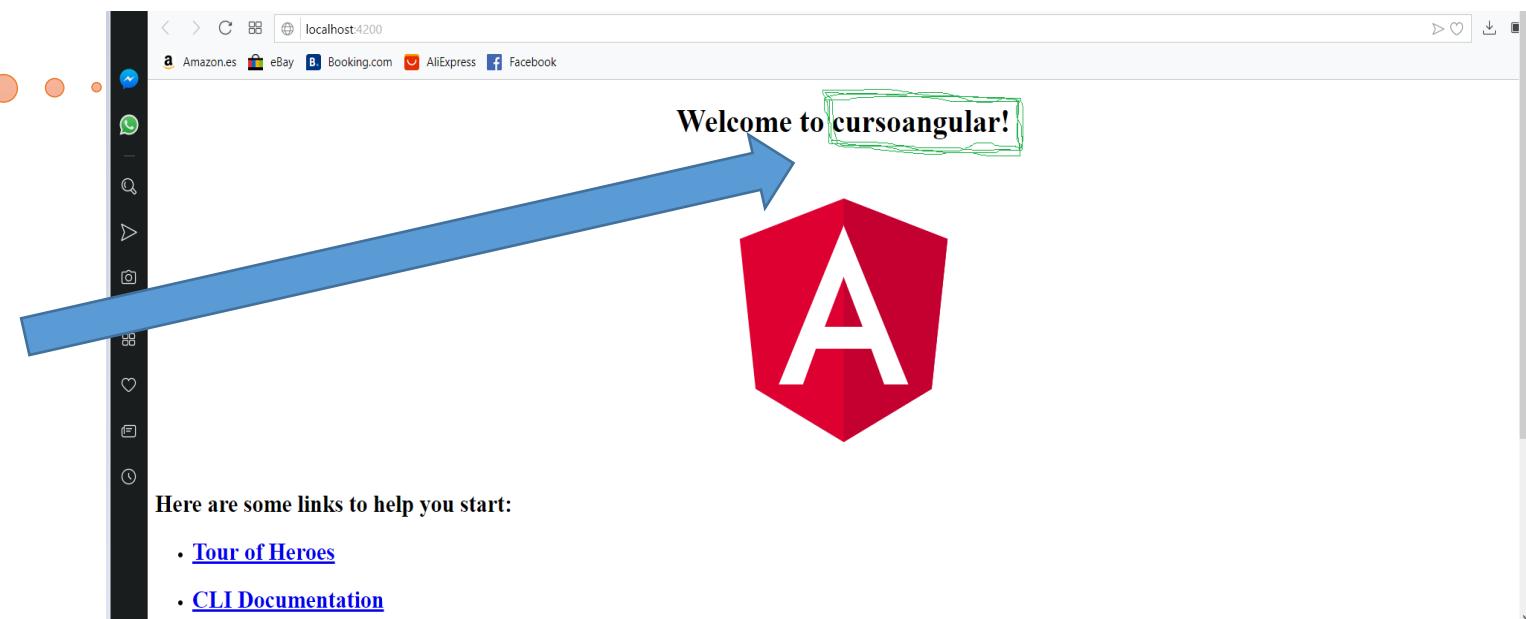
C:\Users\codig>ng serve --open
```

En el navegador aparece el contenido de la propiedad 'title':

Podemos ver que aparece el string 'cursoangular' y no {{ title }}:

```
title = 'cursoangular';
```

Este concepto de sustitución se llama  
interpolación

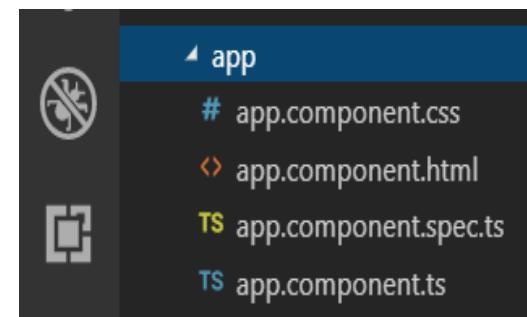


Otro archivo que se asocia a la componente 'AppComponent' es 'app.component.css' donde se almacenan todos los estilos que se van a aplicar solo a dicha componente, es decir que quedarán encapsulados en la componente 'AppComponent'.



En la carpeta raíz del proyecto hay un archivo llamado 'styles.css' donde podemos definir estilos que se aplicarán en forma global a todas las componentes de nuestra aplicación:

Ya hemos nombrado los tres archivos fundamentales que definen toda componente:



Queda uno llamado 'app.component.spec.ts' que tiene por objetivo definir código de testing para medir el correcto funcionamiento de la componente



## app

```
# app.component.css  
< app.component.html  
TS app.component.spec.ts  
TS app.component.ts  
TS app.module.ts
```

Nombramos este archivo porque como mínimo una aplicación en Angular debe tener un módulo. Podemos ver que en la función `@NgModule` en la propiedad 'declaraciones' se le pasa un vector con un elemento que es nuestra componente 'AppComponent'. La aplicación mínima en Angular debe tener un módulo y dentro de dicho módulo como mínimo una componente:

### AppModule

AppComponent



Otro archivo fundamental que nos crea Angular CLI es 'app.module.ts' en la misma carpeta donde se encuentran los 4 archivos de la componente 'AppComponent':

### app.module.ts

```
1 import { BrowserModule } from '@angular/platform-browser';  
2 import { NgModule } from '@angular/core';  
3  
4 import { AppComponent } from './app.component';  
5  
6 @NgModule({  
7   declarations: [  
8     AppComponent  
9   ],  
10  imports: [  
11    BrowserModule  
12  ],  
13  providers: [],  
14  bootstrap: [AppComponent]  
15 })  
16 export class AppModule {}  
17
```

Un proyecto grande se divide en diferentes módulos con un conjunto de componentes cada uno. Por ejemplo podemos tener un módulo 'Cientes' con tres componentes que resuelven distintas partes visuales de la aplicación:

### Cientes

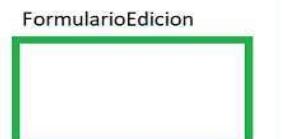
ListadoCientes



Busqueda



FormularioEdicion



## Interpolación en los archivos HTML de Angular

Una de las características fundamentales en Angular es separar la vista del modelo de datos. En el modelo de datos tenemos las variables y en la vista implementamos como se muestran dichos datos.  
Modificaremos el proyecto001 para ver este concepto de interpolación.

Abriremos el archivo que tiene la clase AppComponent (app.component.ts) y lo modificaremos con el siguiente código:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre = 'Rodriguez Pablo';
  edad = 40;
  email = 'rpablo@gmail.com';
  sueldos = [1700, 1600, 1900];
  activo = true;

  esActivo() {
    if (this.activo)
      return 'Trabajador Activo';
    else
      return 'Trabajador Inactivo';
  }

  ultimos3Sueldos() {
    let suma=0;
    for(let x=0; x<this.sueldos.length; x++)
      suma+=this.sueldos[x];
    return suma;
  }
}
```

La clase 'AppComponent' representa los datos de un empleado. Definimos e inicializamos 5 propiedades:

```
nombre = 'Rodriguez Pablo';
edad = 40;
email = 'rpablo@gmail.com';
sueldos = [1700, 1600, 1900];
activo = true;
```

Definimos dos métodos, en el primero según el valor que almacena la propiedad 'estado' retornamos un string que informa si es un empleado activo o inactivo:

```
esActivo() {
    if (this.activo)
        return 'Trabajador Activo';
    else
        return 'Trabajador Inactivo';
}
```

El segundo método retorna la suma de sus últimos 3 meses de trabajo que se almacenan en la propiedad 'sueldos':

```
ultimos3Sueldos() {
    let suma=0;
    for(let x=0; x<this.sueldos.length; x++)
        suma+=this.sueldos[x];
    return suma;
}
```

Veamos ahora el archivo html que muestra los datos, esto se encuentra en 'app.component.html':

```
<div>
  <p>Nombre del Empleado: {{nombre}}</p>
  <p>Edad: {{edad}}</p>
  <p>Los últimos tres sueldos son: {{sueldos[0]}}, {{sueldos[1]}} y {{sueldos[2]}}</p>
  <p>En los últimos 3 meses ha ganado: {{ultimos3Sueldos()}}</p>
  <p>{{esActivo()}}</p>
</div>
```

Para acceder a las propiedades del objeto dentro del template del HTML debemos disponer dos llaves abiertas y cerradas y dentro el nombre de la propiedad:

Cuando se tratan de vectores la primer forma que podemos acceder es mediante un subíndice:

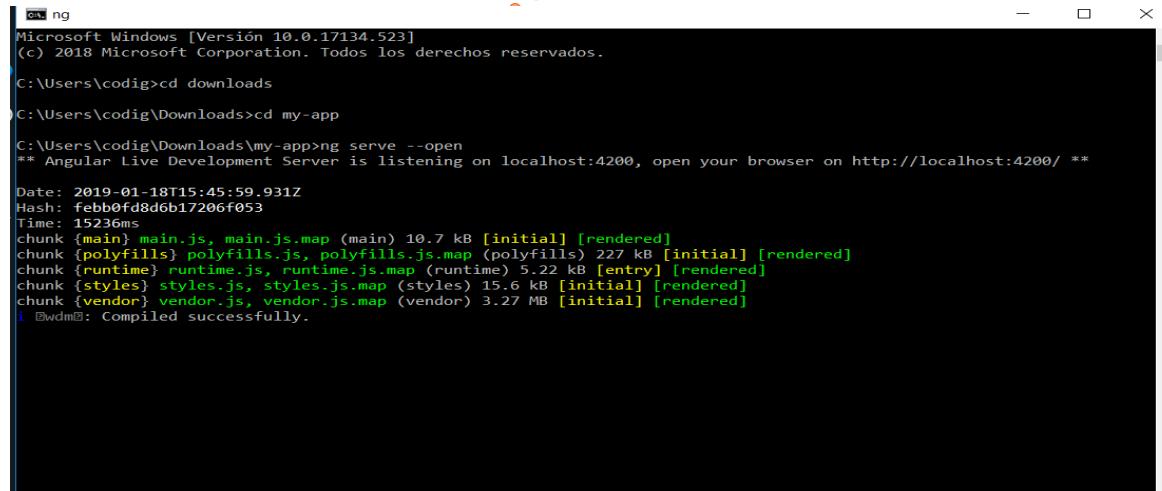
```
<p>Nombre del Empleado:{{nombre}}</p>
```

```
<p>Los últimos tres sueldos son: {{sueldos[0]}}, {{sueldos[1]}} y {{sueldos[2]}}</p>
```

Finalmente podemos llamar a métodos que tiene por objetivo consultar el valor de propiedades:

```
<p>En los últimos 3 meses ha ganado: {{ultimos3Sueldos()}}</p>
<p>{{esActivo()}}</p>
```

Cuando ejecutamos nuestra aplicación desde la línea de comandos de Node.js:



```
git:~ ng
Microsoft Windows [Versión 10.0.17134.523]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\codig>cd downloads
C:\Users\codig\Downloads>cd my-app

C:\Users\codig\Downloads\my-app>ng serve --open
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

Date: 2019-01-18T15:45:59.931Z
Hash: febb0fd8d6b17206f053
Time: 15236ms
chunk {main} main.js, main.js.map (main) 10.7 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 227 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 5.22 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 15.6 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 3.27 MB [initial] [rendered]
i wdm: Compiled successfully.
```

En el navegador aparece el contenido de la vista pero con los valores sustituidos donde dispusimos las llaves {{}}:



## Directivas \*ngIf y \*ngFor

Las directivas \*ngIf y \*ngFor son atributos que podemos agregarle a los elementos HTML que nos permiten en el caso del \*ngIf condicionar si dicha marca debe agregarse a la página HTML.

La directiva \*ngFor nos permite generar muchos elementos HTML repetidos a partir del recorrido de un arreglo de datos.

Para analizar con un ejemplo estas directivas procederemos nuevamente a modificar el proyecto001.

En el archivo 'app.component.ts' procedemos a codificar la clase AppComponent con la definición de 3 propiedades:

Hemos definido las propiedades nombre, edad y sueldos en la clase AppComponent:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre = 'Rodriguez Pablo';
  edad = 40;
  sueldos = [1700, 1600, 1900];
}
```

Ahora procedemos a modificar el archivo app.component.html:

```
<div>
  <p>Nombre del Empleado:{{nombre}}</p>
  <p>Edad:{{edad}}</p>
  <p *ngIf="edad>=18">Es mayor de edad.</p>
  <table border="1">
    <tr>
      <td>Sueldos</td>
    </tr>
    <tr *ngFor="let sueldo of sueldos">
      <td>{{sueldo}}</td>
    </tr>
  </table>
</div>
```

En el navegador aparece el siguiente contenido:

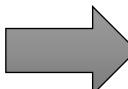


La directiva \*ngIf verifica la condición que indicamos entre comillas, en el caso de verificarse verdadero se agrega el elemento HTML 'p':



```
<p *ngIf="edad>=18">Es mayor de edad.</p>
```

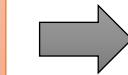
Probemos de modificar la propiedad edad en la clase AppComponent por el valor 7:



```
export class AppComponent {  
  nombre = 'Rodriguez Pablo';  
  edad = 7;  
  sueldos = [1700, 1600, 1900];  
}
```

Al recargar la página podemos comprobar que no aparece el mensaje contenido en dicho párrafo: 'Es mayor de edad.'

La directiva \*ngFor nos genera posiblemente muchos elementos HTML repetidos, en este ejemplo una serie de filas de una tabla HTML:



```
<tr *ngFor="let sueldo of sueldos">  
  <td>{{sueldo}}</td>  
</tr>
```

En cada repetición en la variable 'sueldo' se almacena una componente del arreglo 'sueldos'. De esta forma podemos mostrar los datos del arreglo mediante la directiva \*ngFor.

La directiva \*ngIf podemos plantear un else con la siguiente sintaxis:



```
<p *ngIf="edad>=18; else menor">Es mayor de edad.</p>  
<ng-template #menor><p>Es un menor de edad.</p></ng-template>
```

En el caso que la condición del \*ngIf se verifique falso le indicamos con un else un nombre que debe luego especificarse en un elemento ng-template. Lo que disponemos dentro del elemento ng-template es lo que se muestra.

## Captura de eventos

El evento más común que podemos encontrar en cualquier aplicación es la presión de un botón. Modificaremos nuevamente el proyecto para que la componente AppComponent muestre un etiqueta con un número 0 y luego dos botones que permitan incrementar o decrementar en uno el contenido de la etiqueta.

Otra actividad muy común en una aplicación es la captura de eventos. La presión de un botón, la presión de una tecla, el desplazamiento de la flecha del mouse etc. son eventos que podemos capturar

Nuevamente debemos modificar el archivo 'app.component.ts' con el siguiente código:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  contador = 1;

  incrementar() {
    this.contador++;
  }

  decrementar() {
    this.contador--;
  }
}
```

Definimos en la clase la propiedad 'contador' y lo iniciamos con el valor '1':

```
export class AppComponent {  
  contador = 1;
```

Luego otros dos métodos de la clase AppComponent, que serán llamados al presionar alguno de los botones, incrementan en uno o decrementan en uno el valor almacenado en la propiedad contador:

```
incrementar() {  
  this.contador++;  
}  
  
decrementar() {  
  this.contador--;  
}
```

Recordar que las propiedades dentro de los métodos debemos anteceder la palabra clave 'this'

El segundo archivo donde se encuentra la vista de la componente es app.component.html:

```
<div>  
  <p>{{ contador }}</p>  
  <button (click)="incrementar()">Sumar 1</button>  
  <button (click)="decrementar()">Restar 1</button>  
</div>
```

Como ya conocemos mostramos el contenido de la propiedad contador mediante interpolación de string:

```
<p>{{contador}}</p>
```

Luego definimos dos elementos HTML de tipo 'button' y definimos los eventos click (deben ir entre paréntesis los nombres de los eventos) y luego entre comillas el nombre del método que se llama:

```
<button (click)="incrementar()">Sumar 1</button>  
<button (click)="decrementar()">Restar 1</button>
```

En el navegador aparece la siguiente interfaz:



Cuando se presiona el botón 'Sumar 1' se llama el método 'incrementar()', en dicho método si recordamos se modifica el contenido de la propiedad 'contador':

```
incrementar() {  
    this.contador++;  
}
```

Lo más importante notar que Angular detecta cuando se modifican valores almacenados en propiedades y automáticamente se encarga de actualizar la interfaz visual sin tener que llamar a algún método.

Este concepto se conoce como 'binding' en una dirección (cambio en atributos de la clase se actualizan en la vista)

## Directiva ngModel

Esta directiva nos permite tener un enlace unidireccional entre una propiedad de nuestra clase con el valor que se muestra en un control de formulario HTML de tipo input, textarea etc.

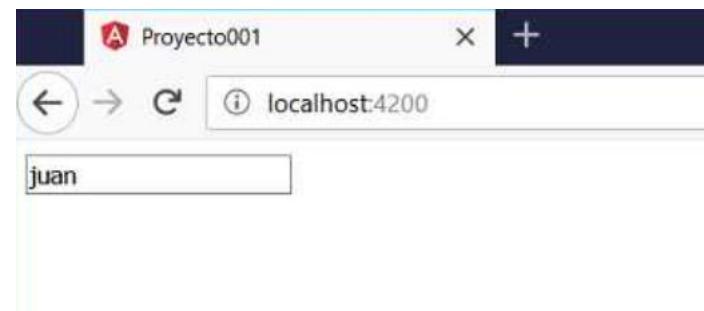
Por ejemplo si en la clase AppComponent tenemos la propiedad 'nombre' con el valor 'juan':

```
nombre='juan';
```

Luego en la vista definimos la directiva ngModel entre corchetes y le asignamos el nombre de la propiedad definida en la clase:

```
<input type="text" [ngModel]="nombre">
```

Cuando arrancamos la aplicación podemos observar que el control input aparece automáticamente con el valor 'juan':



Lo que hay que tener en cuenta es que es un enlace en una dirección: el valor de la propiedad de la clase se refleja en la interfaz visual. Si el operador cambia el contenido del control 'input' por ejemplo por el nombre 'ana' luego la propiedad 'nombre' de la clase sigue almacenando el valor 'juan'.

Si queremos que el enlace sea en las dos direcciones debemos utilizar la siguiente sintaxis:

Un primer ejemplo muy corto que podemos hacer es modificar el proyecto001 para que se ingrese el nombre y apellido de una persona y se muestre inmediatamente en la parte inferior.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
    ,FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

```
<input type="text" [(ngModel)]="nombre">
```

Cuando utilizamos la directiva ngModel debemos importar la clase 'FormsModule' en el archivo 'app.module.ts' y especificarla en la propiedad 'imports':

Luego nuestro archivo 'app.component.ts' debe tener:

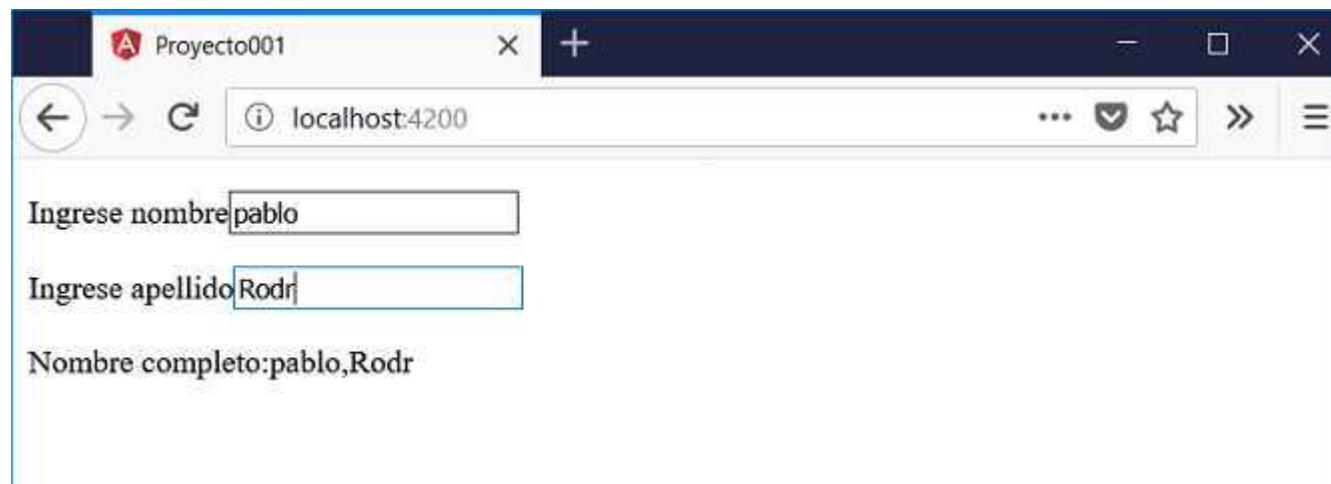
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre='';
  apellido='';
}
```

```
<div>
  <p>Ingrese nombre<input type="text" [(ngModel)]="nombre"></p>
  <p>Ingrese apellido<input type="text" [(ngModel)]="apellido"></p>
  <p>Nombre completo:<{nombre}>, <{apellido}></p>
</div>
```

En el navegador nos muestra como se actualizan las propiedades cada vez que ingresamos un carácter en los controles 'input':

...



## Problema

Confeccionar una aplicación que permita administrar un vector de objetos que almacena en cada elemento el código, descripción y precio de un artículo. Se debe poder agregar, borrar y modificar los datos de un artículo.

La interfaz visual de la aplicación debe ser similar a esta:

The screenshot shows a web browser window titled "Proyecto002" at "localhost:4200". The main content is a table titled "Administración de artículos" containing the following data:

| Codigo | Descripcion | Precio | Borrar  | Seleccionar |
|--------|-------------|--------|---------|-------------|
| 1      | papas       | 10.55  | Borrar? | Seleccionar |
| 2      | manzanas    | 12.1   | Borrar? | Seleccionar |
| 3      | melon       | 52.3   | Borrar? | Seleccionar |
| 4      | cebollas    | 17     | Borrar? | Seleccionar |
| 5      | calabaza    | 20     | Borrar? | Seleccionar |

Below the table, there are input fields for adding new items:

Codigo:

descripcion:

precio:

Buttons at the bottom: Agregar and Modificar

## Proceso del problema

Como trabajaremos con un formulario donde el operador ingresará el código, descripción y precio de productos lo más conveniente es enlazar los controles 'input' mediante la directiva 'ngModel'. Debemos entonces importar la clase 'FormsModule' en el archivo 'app.module.ts':

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
    , FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

## Proceso del problema

En el archivo app.component.html implementamos:

```
<div>
  <h1>Administración de artículos</h1>
  <table border="1" *ngIf="hayRegistros(); else sinarticulos">
    <tr>
      <td>Codigo</td><td>Descripción</td><td>Precio</td><td>Borrar</td><td>Seleccionar</td>
    </tr>
    <tr *ngFor="let art of articulos">
      <td>{{art.codigo}}</td>
      <td>{{art.descripcion}}</td>
      <td>{{art.precio}}</td>
      <td><button (click)="borrar(art)">Borrar?</button></td>
      <td><button (click)="seleccionar(art)">Seleccionar</button></td>
    </tr>
  </table>
  <ng-template #sinarticulos><p>No hay artículos.</p></ng-template>
  <div>
    <p>
      Código:<input type="number" [(ngModel)]="art.codigo" />
    </p>
    <p>
      descripción:<input type="text" [(ngModel)]="art.descripcion" />
    </p>
    <p>
      precio:<input type="number" [(ngModel)]="art.precio" />
    </p>
    <p><button (click)="agregar()">Agregar</button>
      <button (click)="modificar()">Modificar</button></p>
    </div>
  </div>
```

## Proceso del problema

La clase app.component.ts es:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  art={
    codigo:null,
    descripcion:null,
    precio:null
  }

  articulos = [{codigo:1, descripcion:'papas', precio:10.55},
  {codigo:2, descripcion:'manzanas', precio:12.10},
  {codigo:3, descripcion:'malon', precio:52.30},
  {codigo:4, descripcion:'cebollas', precio:17},
  {codigo:5, descripcion:'calabaza', precio:20},
  ];

  hayRegistros() {
    return this.articulos.length>0;
  }

  borrar(art) {
    for(let x=0;x<this.articulos.length;x++)
      if (this.articulos[x].codigo==art.codigo)
      {
        this.articulos.splice(x,1);
        return;
      }
  }

  agregar() {
    for(let x=0;x<this.articulos.length;x++)
      if (this.articulos[x].codigo==this.art.codigo)
      {
        alert('ya existe un articulo con dicho codigo');
        return;
      }
    this.articulos.push({codigo:this.art.codigo,
                        descripcion:this.art.descripcion,
                        precio:this.art.precio });
    this.art.codigo=null,
    this.art.descripcion=null,
    this.art.precio=null,
  }

  seleccionar(art) {
    this.art.codigo=art.codigo,
    this.art.descripcion=art.descripcion,
    this.art.precio=art.precio,
  }

  modificar() {
    for(let x=0;x<this.articulos.length;x++)
      if (this.articulos[x].codigo==this.art.codigo)
      {
        this.articulos[x].descripcion=this.art.descripcion,
        this.articulos[x].precio=this.art.precio,
        return;
      }
    alert('No existe el codigo de articulo ingresado');
  }
}
```

## Explicación de problema

### Listado

Definimos en el modelo (app.component.ts) un vector de objetos llamado 'articulos' y almacenamos 5 elementos:

En la vista (app.component.html) generamos una tabla HTML que muestre los datos del modelo y lo recorremos mediante la directiva \*ngFor:

Disponemos en cada fila dos botones y definimos sus respectivos eventos 'click' para que al ser presionados llamen a métodos del modelo para borrar o seleccionar el artículo respectivo. Los métodos envían como parámetro el artículo para saber cual borrar o seleccionar.

En la vista disponemos una serie de 'input' que nos permiten ingresar el código, descripción y precio de un artículo:

Pasemos a analizar las distintas partes de nuestra aplicación. Los archivos app.component.ts y app.component.html están totalmente integrados y con objetivos bien definidos cada uno. El archivo '.html' almacena la vista y el archivo '.ts' almacena el modelo de datos.

```
articulos = [{codigo:1, descripcion:'papas', precio:10.55},  
             {codigo:2, descripcion:'manzanas', precio:12.10},  
             {codigo:3, descripcion:'melon', precio:52.30},  
             {codigo:4, descripcion:'cebollas', precio:17},  
             {codigo:5, descripcion:'calabaza', precio:20},  
         ];
```

```
<table border="1" *ngIf="hayRegistros(); else sinarticulos">  
  <tr>  
    <td>Codigo</td><td>Descripcion</td><td>Precio</td><td>Borrar</td><td>Seleccionar</td>  
  </tr>  
  <tr *ngFor="let art of articulos">  
    <td>{{art.codigo}}</td>  
    <td>{{art.descripcion}}</td>  
    <td>{{art.precio}}</td>  
    <td><button (click)="borrar(art)">Borrar</button></td>  
    <td><button (click)="seleccionar(art)">Seleccionar</button></td>  
  </tr>  
</table>
```

```
<div>  
  <p>  
    Código:<input type="number" [(ngModel)]="art.codigo" />  
  </p>  
  <p>  
    descripción:<input type="text" [(ngModel)]="art.descripcion" />  
  </p>  
  <p>  
    precio:<input type="number" [(ngModel)]="art.precio" />  
  </p>  
  <p><button (click)="agregar()">Agregar</button>  
    <button (click)="modificar()">Modificar</button></p>  
</div>
```

## Agregado

Podemos comprobar que los controles HTML tienen la directiva 'ngModel' bidireccional, es decir que cuando el operador carga un dato en el primer 'input' se actualiza automáticamente en el modelo el dato cargado en 'art.codigo':  
Podemos comprobar que en el modelo tenemos definido un objeto llamado art con tres propiedades:

En este método primero recorremos el vector articulos para comprobar si hay algún otro artículo con el mismo código. En el caso que no exista procedemos a añadir un nuevo elemento llamando al método push y pasando un objeto que creamos en dicho momento con los datos almacenados en el objeto 'art' que se encuentra enlazado con el formulario.

Luego asignamos null a todas las propiedades del objeto art con el objetivo de borrar todos los 'input' del formulario.

Al agregar un elemento al vector 'Angular' se encarga de actualizar la vista sin tener que indicar nada en nuestro código.

```
art={  
  codigo:null,  
  descripcion:null,  
  precio:null  
}
```

Al presionar el botón agregar se ejecuta el método 'agregar':

```
agregar() {  
  for(let x=0;x<this.articulos.length;x++)  
  if (this.articulos[x].codigo==this.art.codigo)  
  {  
    alert('ya existe un articulo con dicho codigo');  
    return;  
  }  
  this.articulos.push({codigo:this.art.codigo,  
                      descripcion:this.art.descripcion,  
                      precio:this.art.precio});  
  
  this.art.codigo=null;  
  this.art.descripcion=null;  
  this.art.precio=null;  
}
```

## Borrado

Cuando se presiona el botón de borrar se ejecuta el método 'borrar':

```
borrar(art) {  
    for(let x=0;x<this.articulos.length;x++)  
        if (this.articulos[x].codigo==art.codigo)  
        {  
            this.articulos.splice(x,1);  
            return;  
        }  
}
```

Recorremos el vector y controlamos uno a uno el código del artículo seleccionado con cada uno de los elementos del vector. El que coincide lo eliminamos del vector llamando al método splice indicando la posición y cuantas componentes borrar a partir de ese.

## Selección

Cuando se presiona el botón de seleccionar se ejecuta el método 'seleccionar':

```
seleccionar(art) {  
    this.art.codigo=art.codigo;  
    this.art.descripcion=art.descripcion;  
    this.art.precio=art.precio;  
}
```

Lo único que hacemos es actualizar el objeto art del modelo con el artículo que acaba de seleccionar el operador (llega como parámetro el artículo seleccionado)

## Modificación

Cuando presiona el botón de modificación se ejecuta el método:

```
modificar() {
  for(let x=0;x<this.articulos.length;x++)
    if (this.articulos[x].codigo==this.art.codigo)
    {
      this.articulos[x].descripcion=this.art.descripcion;
      this.articulos[x].precio=this.art.precio;
      return;
    }
  alert('No existe el código de articulo ingresado');
}
```

Buscamos el código de articulo del control 'input' dentro del vector, en caso de encontrarlo procedemos a modificar la descripción y precio. Por ultimo decir que hemos utilizado la directiva \*ngIf para no mostrar la tabla HTML en caso que el vector 'articulos' se encuentre vacío:

```
<table border="1" *ngIf="hayRegistros(); else sinarticulos">
```

Llamamos al método 'hayRegistros()' que retorna true o false dependiendo si el vector tiene o no componentes:

```
hayRegistros() {
  return this.articulos.length>0;
}
```

## Componentes: creación

Hasta ahora siempre hemos desarrollado toda la lógica en la componente que se crea por defecto al crear un proyecto con Angular CLI.

La realidad es que en proyectos de mediano y gran tamaño no podemos disponer toda la lógica en una única componente.

Las componentes son una de las características fundamentales de Angular. Ayudan a extender las características básicas de las etiquetas HTML y encapsular código.

Para crear otras componentes la herramienta Angular CLI nos provee la posibilidad de crearlas desde la línea de comandos de Node.js

## Ejercicio de la clase

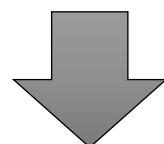
Implementar una aplicación que muestre tres datos. Crear una componente 'dato' además de la componente que crea por defecto Angular CLI

Desde la línea de comandos de Node.js procedemos a crear el proyecto:

```
f:\angularya> ng new proyecto003
```

Primero descendemos a la carpeta proyecto003 y nuevamente desde la línea de comandos procedemos a crear la componente 'dato' escribiendo:

```
f:\angularya\proyecto003> ng generate component dato
```



Al ejecutar este comando se crean 4 archivos y se modifica uno:

```
Node.js command prompt
F:\angularya>cd proyecto003

F:\angularya\proyecto003>ng generate component dato
create src/app/dato/dato.component.html (23 bytes)
create src/app/dato/dato.component.spec.ts (614 bytes)
create src/app/dato/dato.component.ts (261 bytes)
create src/app/dato/dato.component.css (0 bytes)
update src/app/app.module.ts (390 bytes)

F:\angularya\proyecto003>
```

Además dentro de la carpeta 'app' se crea una carpeta llamada 'dato' y dentro de ella se localizan los cuatro archivos creados:



El archivo que se modifica es 'app.module.ts' donde podemos comprobar que se importa la componente que acabamos de crear:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { DadoComponent } from './dado/dado.component';

@NgModule({
  declarations: [
    AppComponent,
    DadoComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Es decir que ahora nuestro módulo tiene 2 componentes:

```
declarations: [
  AppComponent,
  DadoComponent
],
```

En nuestro tercer paso vamos a implementar la vista de la componente 'dato' y su modelo. Abrimos el archivo 'dato.component.ts' y codificamos:

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

@Component({
  selector: 'app-dado',
  templateUrl: './dato.component.html',
  styleUrls: ['./dato.component.css']
})
export class DadoComponent implements OnInit {
  valor: number;
  constructor() { }

  ngOnInit() {
    this.valor = Math.trunc(Math.random() * 6) + 1;
  }
}
```

Codificamos ahora el archivo 'dato.component.html':

```
<div class="forma">
  {{valor}}
</div>
```

Como podemos ver solo mostramos el valor almacenado en la propiedad 'valor' definido en el modelo.

Para definir la hoja de estilo del 'dato' abrimos el archivo 'dato.component.css' y codificamos:

```
.forma {  
    width: 5rem;  
    height: 5rem;  
    font-size: 3rem;  
    color:white;  
    background-color: black;  
    border-radius: 1rem;  
    display: inline-flex;  
    justify-content: center;  
    align-items: center;  
    margin:10px;  
}
```

Finamente nos falta definir tres objetos de nuestra clase 'DadoComponent', si volvemos a ver el archivo 'dato.component.ts' podemos identificar en la llamada a @Component que tiene una propiedad llamada 'selector' con el valor 'app-dado':

```
@Component({  
    selector: 'app-dado',  
    templateUrl: './dato.component.html',  
    styleUrls: ['./dato.component.css']  
})
```

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de tres datos:

```
<div style="text-align:center">  
    <app-dado></app-dado>  
    <app-dado></app-dado>  
    <app-dado></app-dado>  
</div>
```

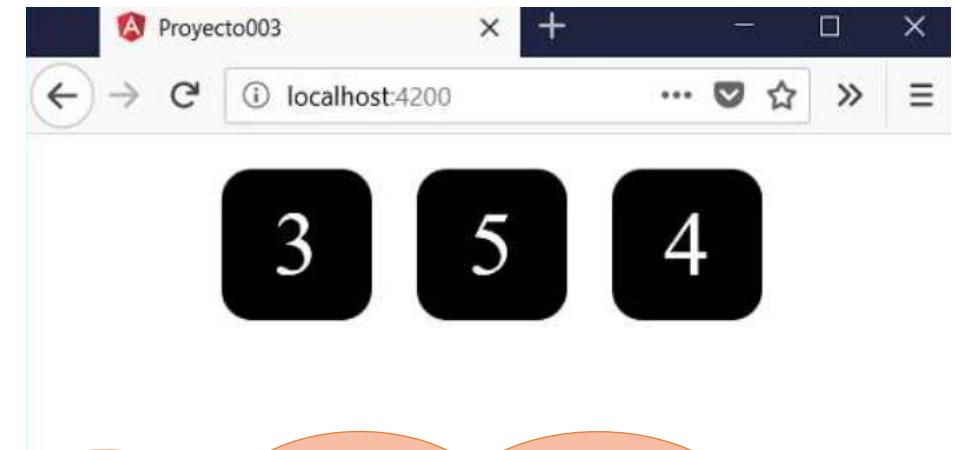
Este es el selector que debemos utilizar para definir objetos de la clase DadoComponent en las vistas.

Si ejecutamos ahora el proyecto:

Podemos ver que tenemos los tres dados en pantalla:

## Puntualización

La división de un proyecto en componentes en Angular nos permite crear piezas independientes y reutilizables. Siempre debe haber una primer componente donde arranca la aplicación, si utilizamos la herramienta Angular CLI se llama 'AppComponent'. Luego podemos crear otras componentes como en nuestro caso de 'DadoComponent'. Tenemos que toda componente tiene un nombre de clase, por ejemplo 'DadoComponent' y luego un nombre de selector definida para dicha componente 'app-dado'. En las vistas para definir objetos de una determinada componente debemos hacer referencia al nombre del selector:



## Componentes: pasar datos de la componente padre a la componente hija

Vimos en el concepto anterior que una aplicación con Angular está conformada por componentes. Siempre hay una componente padre y esta puede tener una o más componentes hijas, a su vez las componentes hijas pueden tener componentes hijas de ellas y así sucesivamente.

En este concepto veremos una técnica para pasar datos de la componente padre a la componente hija.  
Podemos pasar datos a una componente en el momento que definimos una etiqueta de la misma:

```
<app-dado valor="3"></app-dado>
```

Cuando declaramos la etiqueta app-dado definimos una propiedad llamada 'valor' y le pasamos el dato a dicha componente. Es idéntico a lo que hacemos a cuando definimos etiquetas HTML con sus propiedades.

## Ejercicio

Implementar una aplicación que muestre tres dados (crear una componente llamada 'dado') y un botón. Cuando se presione el botón generar tres valores aleatorios y pasarlo a las componentes respectivas para que se muestren. La clase principal debe mostrar un mensaje que el usuario ganó si los tres dados tienen el mismo valor.

Desde la línea de comandos de Node.js procedemos a crear el proyecto004:

```
f:\angularya> ng new proyecto004
```

Primero descendemos a la carpeta proyecto004 y nuevamente desde la línea de comandos procedemos a crear la componente 'dado' escribiendo:

```
f:\angularya\proyecto004> ng generate component dado
```

Recordemos que al ejecutar este comando se crean 4 archivos y se modifica uno.

Además dentro de la carpeta 'app' se crea una carpeta llamada 'dato' y dentro de ella se localizan los cuatro archivos creados.

El archivo que se modifica es 'app.module.ts' donde podemos comprobar que se importa la componente que acabamos de crear:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { DadoComponent } from './dato/dado.component';

@NgModule({
  declarations: [
    AppComponent,
    DadoComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Es decir que ahora nuestro módulo tiene 2 componentes:

```
declarations: [
  AppComponent,
  DadoComponent
],
```

En nuestro tercer paso vamos a implementar la vista de la componente 'dato' y su modelo. Abrimos el archivo 'dato.component.ts' y codificamos:

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

@Component({
  selector: 'app-dado',
  templateUrl: './dato.component.html',
  styleUrls: ['./dato.component.css']
})
export class DadoComponent implements OnInit {

  @Input() valor: number;

  constructor() { }

  ngOnInit() {
  }

}
```

En la clase DadoComponent podemos identificar la sintaxis para definir una propiedad que llega como parámetro de la componente padre:

```
@Input() valor: number;
```

No debemos asignar ningún dato a la propiedad valor ya que se cargará cuando creemos un objeto de esta clase.

Para definir el decorador @Input() debemos importar la clase Input:

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

Codificamos ahora el archivo  
'dato.component.html':

```
<div class="forma">  
  {{valor}}  
</div>
```

Como podemos ver solo mostramos  
el valor almacenado en la propiedad  
'valor' definido en el modelo (no hay  
cambios con el ejemplo anterior)

Para definir la hoja de estilo del 'dato' abrimos el archivo  
'dato.component.css' y codificamos:

```
.forma {  
  width: 5rem;  
  height: 5rem;  
  font-size: 3rem;  
  color:white;  
  background-color: black;  
  border-radius: 1rem;  
  display: inline-flex;  
  justify-content: center;  
  align-items: center;  
  margin:10px;  
}
```

Finamente nos falta definir tres objetos de nuestra clase 'DadoComponent' y pasar los valores que queremos que se muestren, si volvemos a ver el archivo 'dado.component.ts' podemos identificar en la llamada a @Component que tiene una propiedad llamada 'selector' con el valor 'app-dado':

```
@Component({
  selector: 'app-dado',
  templateUrl: './dado.component.html',
  styleUrls: ['./dado.component.css']
})
```

Este es el selector que debemos utilizar para definir objetos de la clase DadoComponent en las vistas.

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de tres dados y mediante interpolación pasamos el valor para cada dado:

```
<div style="text-align:center">
  <app-dado valor="{{valor1}}></app-dado>
  <app-dado valor="{{valor2}}></app-dado>
  <app-dado valor="{{valor3}}></app-dado>
  <hr>
  <button (click)="tirar()">Tirar</button>
  <hr>
  <p>Resultado: {{resultado}}</p>
</div>
```

Ahora codificamos la clase AppComponent donde generamos los tres valores aleatorios que mostrarán las componentes, implementamos además el método que captura el click del botón:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  valor1: number;
  valor2: number;
  valor3: number;
  resultado: string;
  constructor() {
    this.valor1 = this.retornarAleatorio();
    this.valor2 = this.retornarAleatorio();
    this.valor3 = this.retornarAleatorio();
  }

  retornarAleatorio() {
    return Math.trunc(Math.random() * 6) + 1;
  }

  tirar() {
    this.valor1 = this.retornarAleatorio();
    this.valor2 = this.retornarAleatorio();
    this.valor3 = this.retornarAleatorio();
    if (this.valor1==this.valor2 && this.valor1==this.valor3)
      this.resultado='Ganó';
    else
      this.resultado='Perdió';
  }
}
```

Podemos ver que tenemos los tres dados en pantalla, el botón de 'tirar' y el mensaje que se actualiza cada vez que jugamos:



Dependiendo del problema nos conviene definir propiedades privadas a una componente o definir propiedades que lleguen los datos desde la componente padre. En éste problema como tenemos que controlar si los tres dados tienen el mismo valor es más conveniente que el dado tenga solo la responsabilidad de mostrar un valor y que el control de los tres números se debe hacer en la componente principal 'AppComponent'

## Componentes: disparo de eventos de la componente hija a la componente padre

En el concepto anterior vimos como podemos pasar datos de la componente padre a la componente hija mediante la definición de propiedades en el mismo selector de la componente:

```
<app-dado valor="3"></app-dado>
```

Ahora veremos como podemos capturar un evento en la componente padre que emite la componente hija:

```
<app-cronometro inicio="15" (multiplo10)="actualizar($event)"></app-cronometro>
```

En esta componente tenemos una propiedad llamada inicio que le enviamos un dato y capturamos un evento llamado 'multiplo10' que emite la componente app-cronometro.

## Ejercicio

Confeccionar una aplicación con dos componentes llamadas 'AppComponent' y 'CronometroComponent'. La componente 'CronometroComponent' muestra un cronómetro que se actualiza cada un segundo, cada vez que su valor es múltiplo de 10 informa a la componente padre de dicha situación informando el segundo actual.

La componente 'AppComponent' define un cronómetro e informa cada vez que el cronómetro tiene un valor múltiplo de 10.

Desde la línea de comandos de Node.js procedemos a crear el proyecto005:

```
f:\angularya> ng new proyecto005
```

Primero descendemos a la carpeta proyecto005 y nuevamente desde la línea de comandos procedemos a crear la componente 'cronometro' escribiendo:

```
f:\angularya\proyecto005> ng generate component cronometro
```

Recordemos que al ejecutar este comando se crean 4 archivos y se modifica uno.

Además dentro de la carpeta 'app' se crea una carpeta llamada 'cronometro' y dentro de ella se localizan los cuatro archivos creados.

El archivo que se modifica es 'app.module.ts' donde podemos comprobar que se importa la componente que acabamos de crear:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { CronometroComponent } from './cronometro/cronometro.component';

@NgModule({
  declarations: [
    AppComponent,
    CronometroComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Es decir que ahora nuestro módulo tiene 2 componentes:

```
declarations: [  
  AppComponent,  
  CronometroComponent  
,
```

En nuestro tercer paso vamos a implementar la vista de la componente 'cronometro' y su modelo. Abrimos el archivo 'cronometro.component.ts' y codificamos:

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';  
  
@Component({  
  selector: 'app-cronometro',  
  templateUrl: './cronometro.component.html',  
  styleUrls: ['./cronometro.component.css']  
})  
export class CronometroComponent implements OnInit {  
  segundo=0;  
  @Input() inicio;  
  @Output() multiplo10 = new EventEmitter();  
  constructor() { }  
  
  ngOnInit() {  
    this.segundo = this.inicio;  
    setInterval(() => {  
      this.segundo++;  
      if (this.segundo % 10 == 0)  
        this.multiplo10.emit(this.segundo);  
    }, 1000);  
  }  
}
```

En la clase CronometroComponent podemos identificar la sintaxis para definir un evento:

```
@Output() multiplo10 = new EventEmitter();
```

Para definir el decorador @Output() debemos importar:

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

Codificamos ahora el archivo 'cronometro.component.html':

```
<div class="cronometro">  
    {{segundo}} Seg.  
</div>
```

Para definir la hoja de estilo del 'cronometro' abrimos el archivo 'cronometro.component.css' y codificamos:

```
.cronometro {  
    width: 8rem;  
    height: 3rem;  
    font-size: 2rem;  
    color:white;  
    background-color: black;  
    border-radius: 10px;  
    display: inline-flex;  
    justify-content: center;  
    align-items: center;  
    margin:10px;  
}
```

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de un cronometro y un mensaje que se muestra mediante interpolación:

```
<div style="text-align:center">  
    <h1>Prueba de la componente cronometro</h1>  
    <app-cronometro inicio="15" (multiplo10)="actualizar($event)"></app-cronometro>  
    <h2>Evento</h2>  
    <h3>{{mensaje}}</h3>  
</div>
```

Ahora codificamos la clase AppComponent donde definimos el método que captura el evento emitido por el cronómetro:

```
import { Component } from '@angular/core';  
  
@Component({  
    selector: 'app-root',  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
)  
export class AppComponent {  
    mensaje='';  
  
    actualizar(t) {  
        this.mensaje = t + '(se actualiza cada 10 segundos)';  
    }  
}
```

Si ejecutamos ahora el proyecto:



Podemos ver que cada vez que el cronómetro tiene un valor múltiplo de 10 la componente principal actualiza un mensaje gracias al evento que emite la componente 'cronometro':

Componentes: llamar a métodos de la componente hija desde el template del padre

Otra forma de comunicarnos de la componente padre a la componente hija es la posibilidad de llamar a métodos de la componente hija definiendo una variable en el template HTML donde declaramos la componente hija.

Para llamar a los métodos la componente debe definir una variable en el template del HTML:

```
<app-dado #dado1></app-dado>
<button (click)="dado1.tirar()">Tirar</button>
```

Para definir una variable local le antecedemos el carácter # al nombre. Luego podemos llamar a métodos indicando el nombre de la variable y el método a llamar.

## Ejercicio

Crear una componente llamada 'selectornumerico' que le pasemos el mínimo y el máximo valor que puede mostrar. Dentro de la componente definir dos botones que puedan incrementar o decrementar en 1 el valor actual.

En el template HTML de la componente padre definir un selectornumerico y definir una variable local para poder llamar luego a un método del selectornumerico para fijar cualquier valor.

La componente selectornumerico debe ser algo similar a esta interfaz:



Desde la línea de comandos de Node.js procedemos a crear el proyecto006:

```
f:\angularya> ng new proyecto006
```

Desde la línea de comandos de Node.js procedemos a crear el proyecto006:

```
f:\angularya\proyecto006> ng generate component selectornumerico
```

Recordemos que al ejecutar este comando se crean 4 archivos y se modifica uno.  
Además dentro de la carpeta 'app' se crea una carpeta llamada 'selectornumerico' y dentro de ella se localizan los cuatro archivos creados.  
El archivo que se modifica es 'app.module.ts' donde podemos comprobar que se importa la componente que acabamos de crear:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { SelectornumericoComponent } from './selectornumerico/selectornumerico.component'

@NgModule({
  declarations: [
    AppComponent,
    SelectornumericoComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

En nuestro tercer paso vamos a implementar la vista de la componente 'selectornumerico' y su modelo. Abrimos el archivo 'selectornumerico.component.ts' y codificamos:

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

@Component({
  selector: 'app-selectornumerico',
  templateUrl: './selectornumerico.component.html',
  styleUrls: ['./selectornumerico.component.css']
})
export class SelectornumericoComponent implements OnInit {
  @Input() minimo: number;
  @Input() maximo: number;
  actual: number;
  constructor() { }

  ngOnInit() {
    this.actual = this.minimo;
  }

  incrementar() {
    if (this.actual<this.maximo)
      this.actual++;
  }

  decrementar() {
    if (this.actual>this.minimo)
      this.actual--;
  }

  fijar(v:number) {
    if (v>=this.minimo && v<=this.maximo)
      this.actual=v;
  }
}
```

En la clase SelectornumericoComponent podemos identificar el método que llamaremos desde el HTML template de la componente AppComponent:

```
fijar(v:number) {  
  if (v>=this.minimo && v<=this.maximo)  
    this.actual=v;  
}
```

Codificamos ahora el archivo 'selectornumerico.component.html':

```
<div class="selector">  
  <button (click)="decrementar()">-</button>  
  <div class="valor">{actual}</div>  
  <button (click)="incrementar()">+</button>  
</div>
```

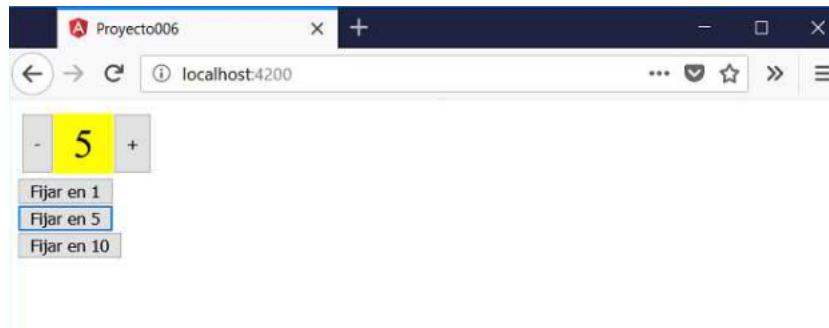
Para definir la hoja de estilo del 'selectornumerico' abrimos el archivo 'selectornumerico.component.css' y codificamos:

```
.selector {  
  display:inline-flex;  
  margin:0.2rem;  
}  
.valor {  
  display:inline-flex;  
  justify-content: center;  
  align-items: center;  
  width: 3rem;  
  height: 3rem;  
  background:#ff0;  
  font-size:2rem;  
}  
  
button {  
  height: 3rem;  
}
```

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de un selectornumerico que define una variable loca y tres botones que llaman a partir de esa variable al método definido dentro del selectornumerico:

```
<div>  
  <app-selectornumerico minimo="1" maximo="10" #selector1></app-selectornumerico>  
  <br>  
  <button (click)="selector1.fijar(1)">Fijar en 1</button><br>  
  <button (click)="selector1.fijar(5)">Fijar en 5</button><br>  
  <button (click)="selector1.fijar(10)">Fijar en 10</button>  
</div>
```

La clase AppComponent la dejamos sin cambios.  
Si ejecutamos ahora el proyecto:



Es decir que mediante dos botones definidos en la componente AppComponent podemos llamar al método fijar de la clase SelectornumeroComponent gracias a que definimos la variable #selector1:

```
<app-selectornumerico minimo="1" maximo="10" #selector1></app-selectornumerico>
<br>
<button (click)="selector1.fijar(1)">Fijar en 1</button><br>
<button (click)="selector1.fijar(5)">Fijar en 5</button><br>
<button (click)="selector1.fijar(10)">Fijar en 10</button>
```

Lo que debe quedar claro que debemos definir una variable en la etiqueta que define la componente y a partir de esta podemos llamar a métodos o inclusive acceder a propiedades de la componente.

## Componentes: llamar a métodos de la componente hija desde la clase padre

En muchas situaciones podemos resolver un problema definiendo una variable local en el template HTML del padre y mediante ésta llamar a sus métodos.



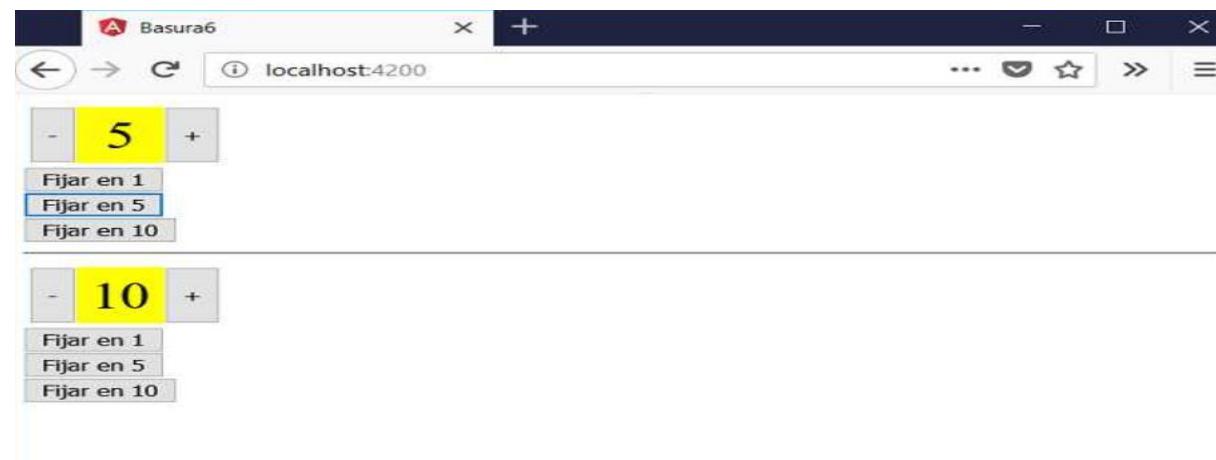
Si necesitamos acceder a la componente hija pero desde la clase (es decir el archivo .ts) tenemos un poco mayor de complejidad que pasaremos a ver en éste concepto.

## Ejercicio

Crear una componente llamada 'selectornumerico' que le pasemos el mínimo y el máximo valor que puede mostrar. Dentro de la componente definir dos botones que puedan incrementar o decrementar en 1 el valor actual (no modificaremos nada la del ejercicio anterior)

En el template HTML de la componente padre definir dos selectores numéricos y tres botones que afectarán a cada selector numérico para fijar los valores 1, 5 y 10.

La aplicación completa se debe mostrar con una interfaz similar a:



Desde la línea de comandos de Node.js procedemos a crear el proyecto007:

Primero descendemos a la carpeta proyecto007 y nuevamente desde la línea de comandos procedemos a crear la componente 'selectornumerico' escribiendo:

En nuestro tercer paso vamos a implementar la vista de la componente 'selectornumerico' y su modelo. Abrimos el archivo 'selectornumerico.component.ts' y codificamos:

```
f:\angularya> ng new proyecto007
```

```
f:\angularya\proyecto007> ng generate component selectornumerico
```

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

@Component({
  selector: 'app-selectornumerico',
  templateUrl: './selectornumerico.component.html',
  styleUrls: ['./selectornumerico.component.css']
})
export class SelectornumericoComponent implements OnInit {
  @Input() minimo: number;
  @Input() maximo: number;
  actual: number;
  constructor() { }

  ngOnInit() {
    this.actual = this.minimo;
  }

  incrementar() {
    if (this.actual<this.maximo)
      this.actual++;
  }

  decrementar() {
    if (this.actual>this.minimo)
      this.actual--;
  }

  fijar(v:number) {
    if (v>=this.minimo && v<=this.maximo)
      this.actual=v;
  }
}
```

En la clase SelectornumericoComponent podemos identificar el método que llamaremos desde la clase AppComponent:

```
fijar(v:number) {  
    if (v>=this.minimo && v<=this.maximo)  
        this.actual=v;  
}
```

Codificamos ahora el archivo 'selectornumerico.component.html':

```
<div class="selector">  
    <button (click)="decrementar()" >-</button>  
    <div class="valor">{{actual}}</div>  
    <button (click)="incrementar()" >+</button>  
</div>
```

Para definir la hoja de estilo del 'selectornumerico' abrimos el archivo 'selectornumerico.component.css' y codificamos:

```
.selector {  
    display:inline-flex;  
    margin:0.2rem;  
}  
.valor {  
    display:inline-flex;  
    justify-content: center;  
    align-items: center;  
    width: 3rem;  
    height: 3rem;  
    background:#ff0;  
    font-size:2rem;  
}  
  
button {  
    height: 3rem;  
}
```

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de dos selectores numéricos que definen una variable local y tres botones para cada uno:

```
<div>
  <app-selectornumerico minimo="1" maximo="10" #selector1></app-selectornumerico>
  <br>
  <button (click)="fijarSelector1(1)">Fijar en 1</button><br>
  <button (click)="fijarSelector1(5)">Fijar en 5</button><br>
  <button (click)="fijarSelector1(10)">Fijar en 10</button>
  <hr>
  <app-selectornumerico minimo="1" maximo="10" #selector2></app-selectornumerico>
  <br>
  <button (click)="fijarSelector2(1)">Fijar en 1</button><br>
  <button (click)="fijarSelector2(5)">Fijar en 5</button><br>
  <button (click)="fijarSelector2(10)">Fijar en 10</button>
</div>
```

La clase AppComponent es la que presenta la mayor cantidad de cambios:

```
import { Component } from '@angular/core';
import { ViewChild } from '@angular/core';
import { SelectornumericoComponent} from './selectornumerico/selectornumerico.component'

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  @ViewChild('selector1') selector1: SelectornumericoComponent;
  @ViewChild('selector2') selector2: SelectornumericoComponent;

  fijarSelector1(valor:number) {
    this.selector1.fijar(valor);
  }

  fijarSelector2(valor:number) {
    this.selector2.fijar(valor);
  }
}
```

Debemos importar la clase SelectornumericoComponent:

```
import { SelectornumericoComponent} from './selectornumerico/selectornumerico.component'
```

Definir una propiedad llamada selector1 y mediante el decorador @ViewChild hacemos referencia a la variable definida en el template HTML que también se llama 'selector1':

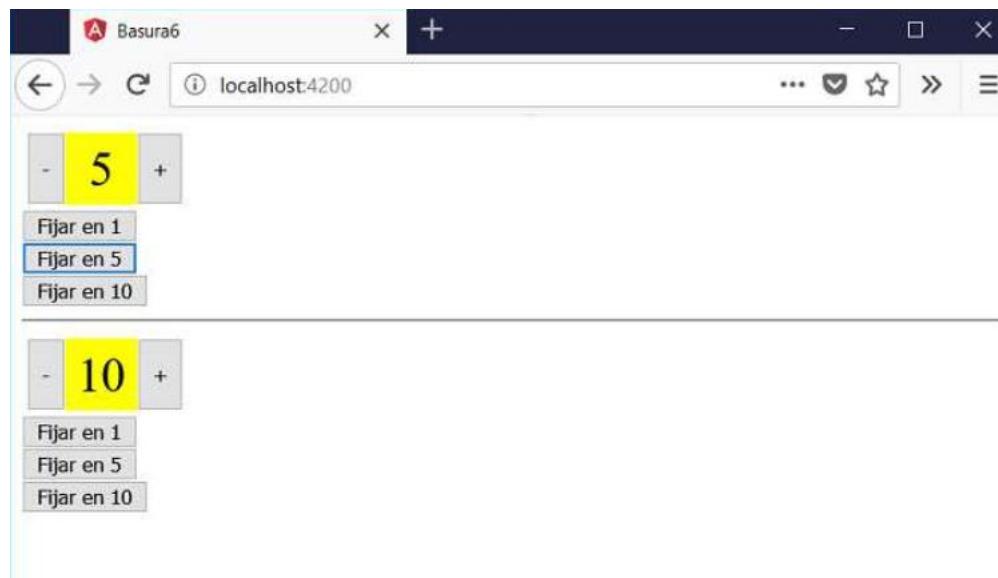
```
@ViewChild('selector1') selector1: SelectornumericoComponent;
```

Mediante las propiedades selector1 y selector2 tenemos las referencias de las dos componentes hijas, luego podemos llamar a sus métodos o acceder a sus propiedades:

```
fijarSelector1(valor:number) {
  this.selector1.fijar(valor);
}

fijarSelector2(valor:number) {
  this.selector2.fijar(valor);
}
```

## Salida



## Componentes: enlace de propiedades (Property Binding)

Vimos que si teníamos que enviar un entero a una propiedad de una componente lo indicamos inmediatamente al valor a enviar:

Hemos visto como enviar datos simples mediante propiedades. Ahora veremos el concepto de Property Binding (enlace de propiedades) muy utilizada si tenemos que enviar una estructura de datos compleja a una componente hija.

```
<app-dado valor="3"></app-dado>
```

En algunas situaciones podemos recuperar el dato del modelo mediante interpolación:

```
<app-dado valor="{{valor}}></app-dado>
```

El problema se presenta si queremos enviar una estructura de datos más compleja como puede ser un arreglo. En estos casos el concepto de enlace de propiedades nos resuelve el problema:

```
<app-listadoarticulos [datos]="articulos"></app-listadoarticulos>
```

Si tenemos la componente app-listadoarticulos y tenemos que enviar el vector 'articulos' disponemos el nombre de la propiedad entre corchetes [datos] Esto corchetes indican a Angular que busque en el modelo la variable 'articulos':

```
articulos = [{codigo:1, descripcion:'papas', precio:10.55},  
             {codigo:2, descripcion:'manzanas', precio:12.10},  
             {codigo:3, descripcion:'melon', precio:52.30},  
             {codigo:4, descripcion:'cebollas', precio:17},  
             {codigo:5, descripcion:'calabaza', precio:20},  
           ];
```

## Ejercicio

En la componente AppComponent definir un arreglo con los datos de 5 artículos. Luego crear una segunda componente llamada 'listadoarticulos' que reciba mediante una propiedad el arreglo y su objetivo sea mostrar en una tabla HTML todos los datos.

Desde la línea de comandos de Node.js procedemos a crear el proyecto008:

```
f:\angularya> ng new proyecto008
```

Primero descendemos a la carpeta proyecto008 y nuevamente desde la línea de comandos procedemos a crear la componente 'listadoarticulos' escribiendo:

```
f:\angularya\proyecto008> ng generate component listadoarticulos
```

En nuestro tercer paso vamos a implementar la vista de la componente 'listadoarticulos' y su modelo. Abrimos el archivo 'listadoarticulos.component.ts' y codificamos:

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

@Component({
  selector: 'app-listadoarticulos',
  templateUrl: './listadoarticulos.component.html',
  styleUrls: ['./listadoarticulos.component.css']
})
export class ListadoarticulosComponent implements OnInit {

  @Input() datos;

  constructor() { }

  ngOnInit() {
  }
}
```

Codificamos ahora el archivo 'listadoarticulos.component.html':

Abrimos ahora el archivo 'app.component.html' y remplazamos su contenido con la definición de una etiqueta app-listadoarticulos con el enlace a la propiedad:

Son fundamentales encerrar entre corchetes el nombre de la propiedad para que angular entienda que hay un enlace y rescate los datos del modelo de la variable 'articulos' y no envíe el string 'articulos'.

La clase AppComponent define la propiedad articulos que es la que se enlaza con la propiedad de la etiqueta app-listadoarticulos:

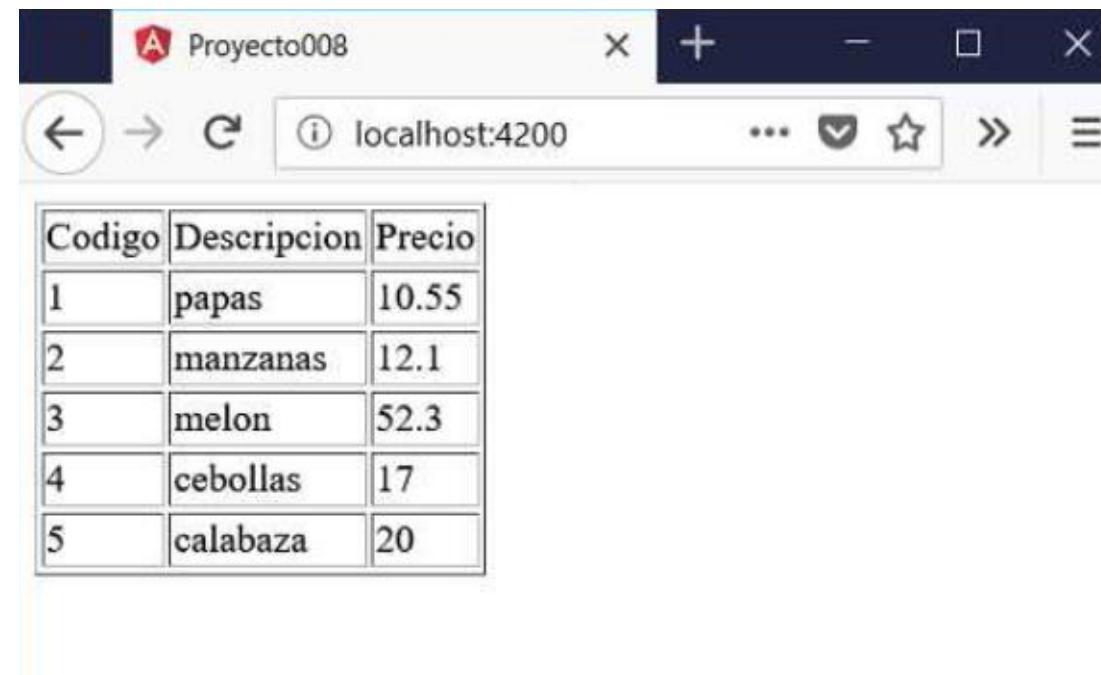
```
<div>
  <table border="1">
    <tr>
      <td>Codigo</td><td>Descripcion</td><td>Precio</td>
    </tr>
    <tr *ngFor="let art of datos">
      <td>{{art.codigo}}</td>
      <td>{{art.descripcion}}</td>
      <td>{{art.precio}}</td>
    </tr>
  </table>
</div>
```

```
<app-listadoarticulos [datos]="articulos"></app-listadoarticulos>
```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  articulos = [{codigo:1, descripcion:'papas', precio:10.55},
               {codigo:2, descripcion:'manzanas', precio:12.10},
               {codigo:3, descripcion:'melon', precio:52.30},
               {codigo:4, descripcion:'cebollas', precio:17},
               {codigo:5, descripcion:'calabaza', precio:20},
             ];
}
```

## Resultado del ejercicio



A screenshot of a web browser window titled "Proyecto008". The address bar shows "localhost:4200". The main content area displays a table with five rows of grocery items:

| Codigo | Descripcion | Precio |
|--------|-------------|--------|
| 1      | papas       | 10.55  |
| 2      | manzanas    | 12.1   |
| 3      | melon       | 52.3   |
| 4      | cebollas    | 17     |
| 5      | calabaza    | 20     |

## Módulos: creación y consumo

Los módulos nos permiten organizar una aplicación compleja en Angular.

Cuando creamos una aplicación con Angular CLI se crea en forma automática el módulo ' AppModule' en el archivo 'app.module.ts'.

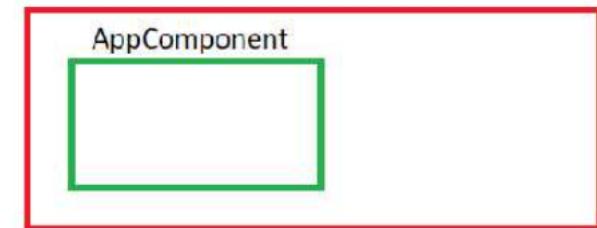
Un módulo tiene por objetivo agrupar un conjunto de componentes relacionadas entre si.

Vamos a ver los pasos para crear un módulo en Angular, agregar una componente en dicho módulo y finalmente consumir dicha componente accediendo a la misma desde otro módulo de la aplicación.

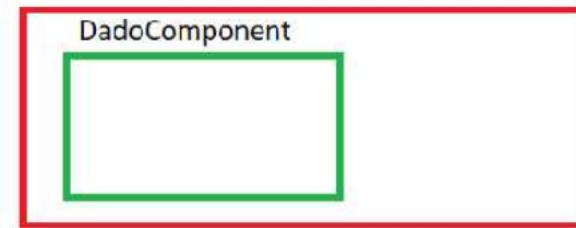
## Ejercicio

Crear un módulo llamado 'elementos' y luego dentro del mismo almacenar una componente llamada 'dado'. Consumir la clase 'dado' desde el módulo principal de la aplicación.

AppModule



ElementosModule



Desde la línea de comandos de Node.js procedemos a crear el proyecto:

```
f:\angularya> ng new proyecto009
```

Primero descendemos a la carpeta proyecto009 y nuevamente desde la línea de comandos procedemos a crear el módulo 'elementos' escribiendo:

```
f:\angularya\proyecto009> ng generate module elementos
```

Al ejecutar este comando se crea una carpeta llamada 'elementos' dentro de la carpeta 'app' y además un archivo llamado 'elementos.module.ts'.

```
Node.js command prompt  
F:\angularya>cd proyecto009  
F:\angularya\proyecto009>ng generate module elementos  
  create src/app/elementos/elementos.module.ts (193 bytes)  
F:\angularya\proyecto009>
```

Se crea el archivo 'elementos.module.ts':

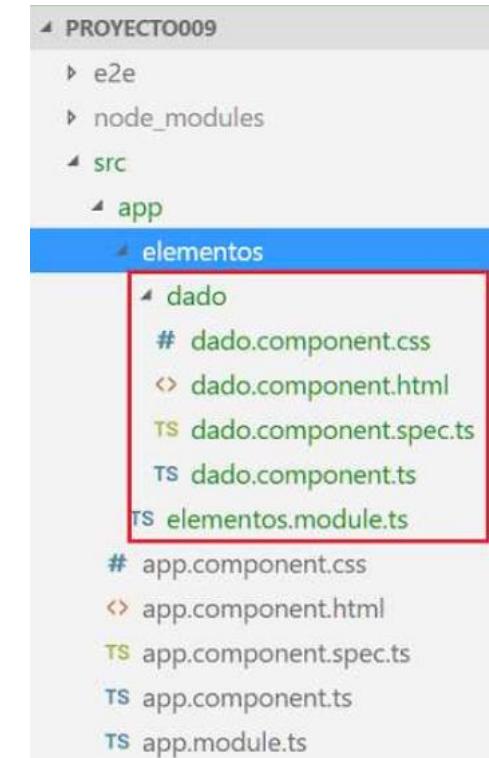
```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class ElementosModule { }
```

En nuestro tercer paso vamos a crear la componente dado almacenando la misma en el módulo elementos (no en el módulo principal):

```
f:\angularaya\proyecto009> ng generate component elementos/dado
```

Como ya sabemos con éste comando se crean los cuatro archivos de la componente y se modifica el archivo 'elementos.module.ts', es importante anteceder al nombre de la componente el módulo donde se debe registrar:



Si abrimos nuevamente el archivo 'elementos.module.ts' podemos ver que la componente 'DadoComponent' se encuentra registrada:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { DadoComponent } from './dado/dado.component';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: [DadoComponent]
})
export class ElementosModule { }
```

Codificamos los archivos de la componente 'dado' como hemos visto en conceptos anteriores de este curso.  
dado.component.ts

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

@Component({
  selector: 'app-dado',
  templateUrl: './dado.component.html',
  styleUrls: ['./dado.component.css']
})
export class DadoComponent implements OnInit {

  @Input() valor: number;

  constructor() { }

  ngOnInit() {
  }

}
```

dado.component.html

```
<div class="forma">  
  {{valor}}  
</div>
```

dado.component.css

```
.forma {  
  width: 5rem;  
  height: 5rem;  
  font-size: 3rem;  
  color:white;  
  background-color: black;  
  border-radius: 1rem;  
  display: inline-flex;  
  justify-content: center;  
  align-items: center;  
  margin:10px;  
}
```

El siguiente paso fundamental para que otros módulos pueden consumir la componente 'dado' es modificar el archivo 'elementos.module.ts' exportando la componente que acabamos de crear:

```
import { NgModule } from '@angular/core';  
import { CommonModule } from '@angular/common';  
import { DadoComponent } from './dado/dado.component';  
  
@NgModule ({  
  imports: [  
    CommonModule  
,  
  exports: [DadoComponent],  
  declarations: [DadoComponent]  
})  
export class ElementosModule { }
```

Ahora abrimos y modificamos el módulo que la va a importar para poder consumirla:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

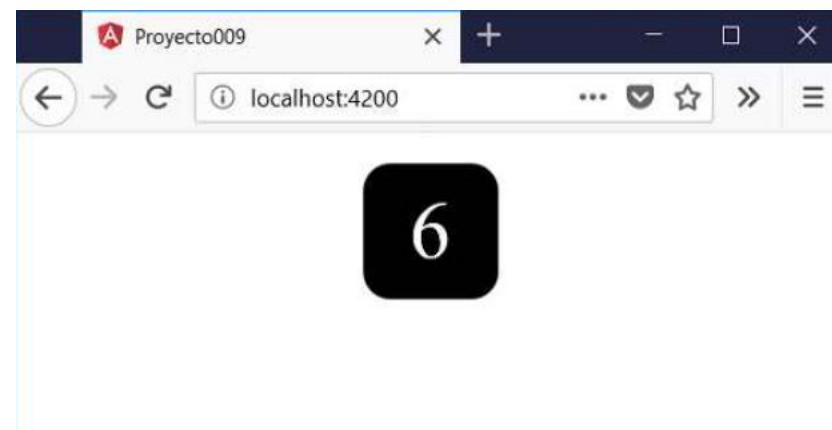
import { ElementosModule } from './elementos/elementos.module'

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    ElementosModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Finalmente en el archivo 'app.component.html' podemos utilizar la clase 'dato' que se encuentra en otro módulo:

```
<div style="text-align:center">
  <app-dado valor="6"></app-dado>
</div>
```

Resultado



## Petición de un archivo JSON a un servidor

Cuando tenemos que hacer peticiones de archivos JSON a un servidor en Angular disponemos de una clase llamada '**HttpClient**' que nos facilita ésta actividad.

Para hacer uso de la **clase 'HttpClient'** debemos importar el **módulo 'HttpClientModule'**.

## Ejercicio

La estructura del archivo JSON es:

```
[  
  {  
    "codigo": 1,  
    "descripcion": "papas",  
    "precio": 12.33  
  },  
  {  
    "codigo": 2,  
    "descripcion": "manzanas",  
    "precio": 54  
  }]
```

Confeccionar una aplicación que recupere una respuesta en JSON de la dirección:

<http://www.codigodata.com/datos.php>

Mostrar en una tabla HTML todos los artículos recuperados.

Desde la línea de comandos de Node.js procedemos a crear el proyecto010:

Como primer paso importaremos el módulo HttpClientModule en el archivo app.module.ts:

```
f:\angularya> ng new proyecto010
```

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

import {HttpClientModule} from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Implementaremos toda la lógica de lectura de datos en la componente por defecto que ha creado Angular CLI, luego en conceptos futuros veremos como distribuir las responsabilidades entre distintas clases.

Importamos la clase HttpClient:

```
import { HttpClient } from '@angular/common/http';
```

Definimos un atributo llamado articulos con valor inicial null:

```
private articulos = null;
```

En Angular podemos definir una propiedad en los parámetros del constructor que se inyecta cuando se crea la componente:

```
constructor(private http: HttpClient) { }
```

Luego la propiedad http que es de la clase HttpClient nos servirá para hacer la petición al servidor.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  private articulos = null;

  constructor(private http: HttpClient) { }

  ngOnInit() {
    this.http.get("http://scratchya.com.ar/vue/datos.php")
      .subscribe(
        result => {
          this.articulos = result;
        },
        error => {
          console.log('problemas');
        }
      );
  }
}
```

En el método onInit que se ejecuta una vez que el template de la componente está creado procedemos a recuperar del servidor los datos llamando al método get de la propiedad http:

```
this.http.get("http://scratchya.com.ar/vue/datos.php")
  .subscribe(
    result => {
      this.articulos = result;
    },
    error => {
      console.log('problemas');
    }
);
```

A partir del objeto que retorna el método get llamamos al método subscribe y le pasamos dos funciones. La primera de esas funciones recibe como parámetro los datos recuperados del servidor.

Falta que veamos como en la vista procedemos a mostrar los datos recuperados 'app.component.html':

Como las peticiones JSON a un servidor pueden demorarse un tiempo mediante la directiva \*ngIf verificamos si la variable articulos tiene un null procedemos a mostrar el contenido de la etiqueta 'ng-template':

```
<div *ngIf="articulos!=null; else espera">
```

Para mostrar los datos de la propiedad 'articulos' lo hacemos como lo hemos visto anteriormente:

```
<div *ngIf="articulos!=null; else espera">
  <table border="1">
    <tr>
      <td>Codigo</td><td>Descripcion</td><td>Precio</td>
    </tr>
    <tr *ngFor="let art of articulos">
      <td>{{art.codigo}}</td>
      <td>{{art.descripcion}}</td>
      <td>{{art.precio}}</td>
    </tr>
  </table>
</div>
<ng-template #espera>Esperando datos...</ng-template>
```

```
<table border="1">
  <tr>
    <td>Codigo</td><td>Descripcion</td><td>Precio</td>
  </tr>
  <tr *ngFor="let art of articulos">
    <td>{{art.codigo}}</td>
    <td>{{art.descripcion}}</td>
    <td>{{art.precio}}</td>
  </tr>
</table>
```

## Resultado



A screenshot of a web browser window titled "Proyecto010". The address bar shows "localhost:4200". The main content area displays a table with three rows of grocery items:

| Codigo | Descripcion | Precio |
|--------|-------------|--------|
| 1      | papas       | 34     |
| 2      | manzanas    | 23.5   |
| 3      | sandia      | 31     |

## Router: definición de rutas

Angular incluye un módulo especial si queremos administrar nuestra aplicación mediante rutas.

Las rutas están dadas por la url, ejemplos de rutas pueden ser:

`http://localhost:4200/clientes`  
`http://localhost:4200/proveedores`  
`http://localhost:4200/contacto`

Luego según la ruta especificada mostramos una vista distinta, generalmente indicando un nombre de componente.

Lo más conveniente es utilizar la aplicación Angular CLI para que nos ayude con la generación del módulo de rutas (Router)

## Ejercicio

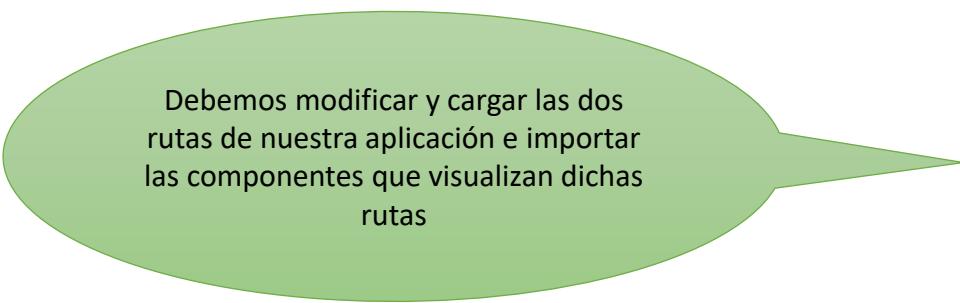
Implementar una aplicación que muestre dos enlaces en la parte superior de la página que acceda el primero al juego de tres dados que habíamos desarrollado en ejercicios anteriores. El segundo enlace debe mostrar mediante otra ruta el nombre del programador, versión y fecha de desarrollo.

Desde la línea de comandos de Node.js procedemos a crear el proyecto011 y solicitaremos que nos cree el archivo de routing:

En la carpeta app se crea el archivo 'app-routing.module.ts':

```
f:\angularya> ng new proyecto011 --routing
```

```
4. import { NgModule } from '@angular/core';
5. import { Routes, RouterModule } from '@angular/router';
6.
7. const routes: Routes = [];
8.
9. @NgModule({
10.   imports: [RouterModule.forRoot(routes)],
11.   exports: [RouterModule]
12. })
13. export class AppRoutingModule { }
```



Debemos modificar y cargar las dos rutas de nuestra aplicación e importar las componentes que visualizan dichas rutas

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { JuegodadosComponent } from './juegodados/juegodados.component';
import { AcercadeComponent } from './acercade/acercade.component';

const routes: Routes = [
  {
    path: 'juegodados',
    component: JuegodadosComponent
  },
  {
    path: 'acercade',
    component: AcercadeComponent
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Por el momento debe mostrar un error cuando iniciamos los nombres de las componentes 'JuegodadosComponent' y 'AcercadeComponent' debido que no las hemos codificado aún.

En la propiedad path estamos indicando que si en el navegador disponemos:

Desde la línea de comandos de Node.js procedemos a crear cada una de las tres componentes que faltan en la aplicación:

`http://localhost:4200/juegodados`

Luego se muestra la componente JuegodadosComponent.

`f:\angularya\proyecto011> ng generate component juegodados`

Codificamos dos archivos:  
juegodados.component.ts

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

@Component({
  selector: 'app-juegodados',
  templateUrl: './juegodados.component.html',
  styleUrls: ['./juegodados.component.css']
})
export class JuegodadosComponent implements OnInit {
  valor1: number;
  valor2: number;
  valor3: number;
  resultado: string;
  constructor() {
    this.valor1 = this.retornarAleatorio();
    this.valor2 = this.retornarAleatorio();
    this.valor3 = this.retornarAleatorio();
  }

  retornarAleatorio() {
    return Math.trunc(Math.random() * 6) + 1;
  }

  tirar() {
    this.valor1 = this.retornarAleatorio();
    this.valor2 = this.retornarAleatorio();
    this.valor3 = this.retornarAleatorio();
    if (this.valor1==this.valor2 && this.valor1==this.valor3)
      this.resultado='Ganó';
    else
      this.resultado='Perdió';
  }

  ngOnInit() {
  }
}
```

juegodados.component.html

```
<app-dado valor="{{valor1}}></app-dado>
<app-dado valor="{{valor2}}></app-dado>
<app-dado valor="{{valor3}}></app-dado>
<hr>
<button (click)="tirar()">Tirar</button>
<hr>
<p>Resultado:{{resultado}}</p>
```

Generamos ahora la componente  
'dado':

```
f:\angularya\proyecto011> ng generate component dado
```

Codificamos sus tres archivos:  
dato.component.ts

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

@Component({
  selector: 'app-dado',
  templateUrl: './dato.component.html',
  styleUrls: ['./dato.component.css']
})
export class DadoComponent implements OnInit {

  @Input() valor: number;

  constructor() { }

  ngOnInit() {
  }
}
```

dado.component.html

```
<div class="forma">  
  {{valor}}  
</div>
```

dado.component.css

```
.forma {  
  width: 5rem;  
  height: 5rem;  
  font-size: 3rem;  
  color: white;  
  background-color: black;  
  border-radius: 1rem;  
  display: inline-flex;  
  justify-content: center;  
  align-items: center;  
  margin: 10px;  
}
```

Finalmente creamos la última componente:

```
f:\angularya\proyecto011> ng generate component acercade
```

Modificamos el archivo  
'acercade.component.html':

```
<h1>Programa: xxxxxxxxx</h1>
<p>Desarrollado:xxxxxxxxxxxxxxxxxxxxxxxx</p>
<p>Fecha: xxxxxxxx</p>
```

Si abrimos ahora el archivo app.module.ts  
tenemos declarados las 4 componentes e  
importado el módulo AppRoutingModule':

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';
import { JuegodadosComponent } from './juegodados/juegodados.component';
import { DadoComponent } from './dado/dado.component';
import { AcercadeComponent } from './acercade/acercade.component';

@NgModule({
  declarations: [
    AppComponent,
    JuegodadosComponent,
    DadoComponent,
    AcercadeComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Nos queda modificar la componente principal 'app.component.html':

```
<div style="text-align:center">
  <a routerLink="/juegodados">Juego de dados</a>
  <a routerLink="/acercade">Acerca de ...</a>
  <div>
    <router-outlet></router-outlet>
  </div>
</div>
```

Mediante la etiqueta 'router-outlet' indicamos el lugar que debe mostrar la componente especificada por la ruta configurada en el archivo 'app-routing.module.ts'

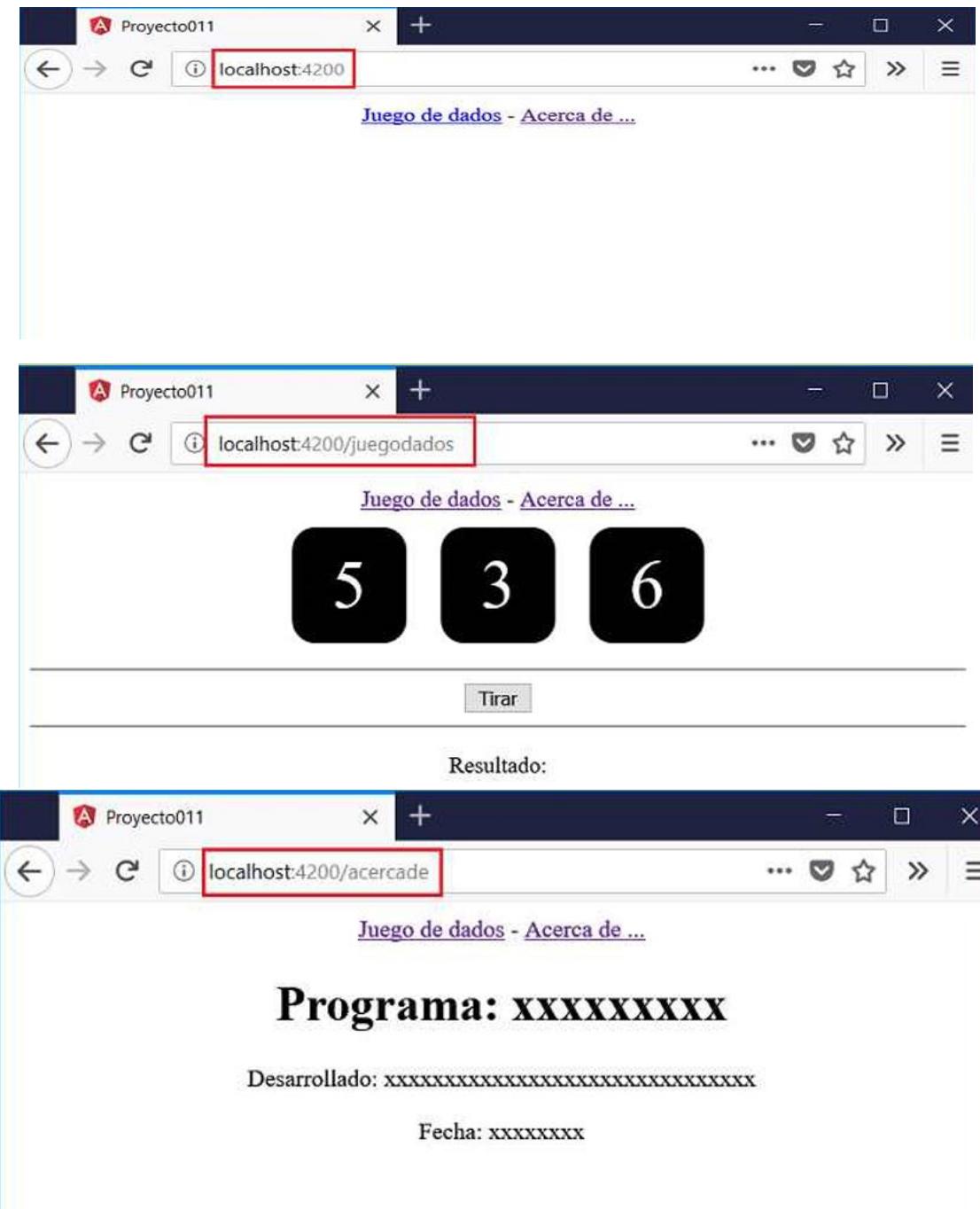
Para cambiar de ruta mediante hipervínculos debemos iniciar la propiedad 'routerLink' asignando la ruta respectiva.

## Resultado

Podemos ver distintos resultados según la ruta indicada:

Si accedemos al primer hipervínculo (tener en cuenta que no se recarga la página, no hay una petición al servidor, sino se resuelve la ruta con aplicación Angular en el navegador):

Finalmente si accedemos a la otra ruta:



## Servicios: concepto y pasos para crearlos

El pilar fundamental cuando creamos una aplicación en Angular es la correcta definición de sus componentes y relaciones entre ellas.

La propuesta del framework de Angular es delegar todas las responsabilidades de acceso a datos (peticiones y envío de datos) y lógica de negocios en otras clases que colaboran con las componentes. Estas clases en Angular se las llama servicios.

Tal es la importancia de los servicios en Angular que la herramienta Angular CLI nos provee la capacidad de crearlos.

Veremos con un ejemplo los pasos que debemos dar para crear un servicio y luego consumirlo desde una componente.

## Ejercicio

Confeccionar una aplicación Angular que muestre una lista de artículos.  
Los artículos almacenarlos en vector localizado en un servicio. Desde la componente acceder al servicio para pedir los artículos a ser mostrados.

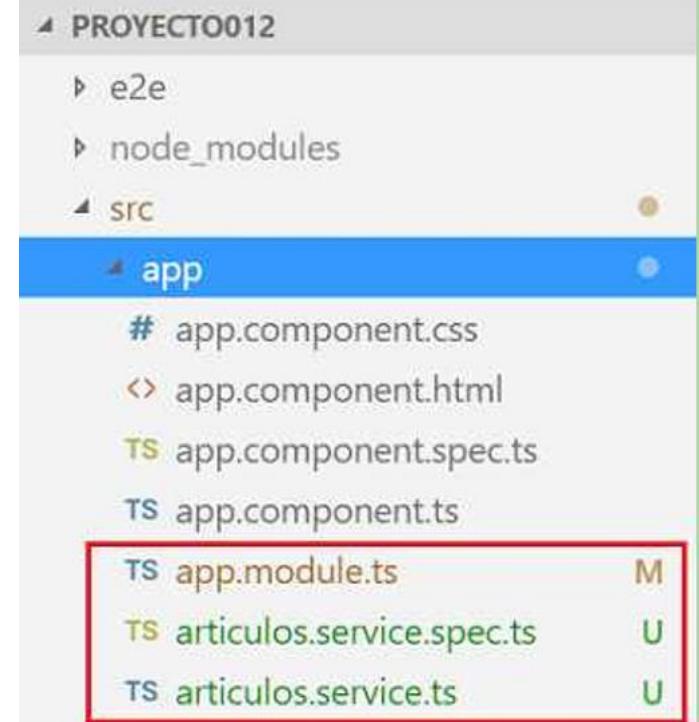
Desde la línea de comandos de Node.js procedemos a crear el proyecto012:

```
f:\angularya> ng new proyecto012
```

Crearemos el servicio que contiene en memoria la lista de artículos (en muchos casos como veremos más adelante el servicio tiene la responsabilidad de recuperar los datos de un servidor web):

```
f:\angularya\proyecto012> ng generate service articulos --module=app
```

Con el comando anterior estamos creando la clase 'ArticulosService' y lo estamos registrando en el módulo 'app' (recordemos que es el módulo que se crea por defecto)  
Se crean dos archivos y se modifica el archivo 'app.module.ts':



El código generado de la clase 'ArticulosService'  
es:

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

@Injectable()
export class ArticulosService {

  constructor() { }

}
```

Lo modificamos por el siguiente código que permita recuperar desde la componente el vector de artículos:

El decorador `@Injectable()` será de suma importancia para poder acceder a esta clase desde la componente.

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

@Injectable()
export class ArticulosService {

  constructor() { }

  retornar() {
    return [
      {
        codigo: 1,
        descripcion: "papas",
        precio: 12.33
      },
      {
        codigo: 2,
        descripcion: "manzanas",
        precio: 54
      },
      {
        codigo: 3,
        descripcion: "sandía",
        precio: 14
      }
    ];
  }
}
```

El archivo 'app.module.ts' se modifica con el siguiente código:

Se agrega en la propiedad 'providers' el nombre del servicio que acabamos de crear: 'ArticulosService'.

Es responsabilidad del framework Angular de crear una instancia de todas las clases que definamos dentro de la propiedad 'providers'

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { ArticulosService } from './articulos.service';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [ArticulosService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ahora veremos como consumimos el servicio desde nuestra componente. Procedemos a modificar la componente que se crea por defecto 'AppComponent' que tiene por responsabilidad mostrar en la página el listado de artículos:

Primero importamos el servicio llamado ArticulosService que se almacena en el archivo 'articulos.service.ts':

```
import { ArticulosService } from './articulos.service';
```

Para injectar el objeto de la clase 'ArticulosService' que crea Angular en forma automática lo hacemos en el parámetro del constructor:

```
constructor(private articulosServicio: ArticulosService) { }
```

```
import { Component, OnInit } from '@angular/core';
import { ArticulosService } from './articulos.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit{
  articulos = null;

  constructor(private articulosServicio: ArticulosService) {}

  ngOnInit() {
    this.articulos=this.articulosServicio.retornar();
  }
}
```

Se almacena en el atributo 'articulosServicio' la referencia del objeto de la clase 'ArticulosService' que crea Angular.

En el método ngOnInit actualizamos la variable 'articulos' con el vector que devuelve el método 'retornar':

```
ngOnInit() {  
  | this.articulos=this.articulosServicio.retornar();  
}
```

Esta asignación dispara la actualización de la página HTML.

Falta que codifiquemos la vista con los datos recuperados:  
app.component.html

```
<table border="1">  
  <tr *ngFor="let articulo of articulos">  
    <td>{{articulo.codigo}}</td>  
    <td>{{articulo.descripcion}}</td>  
    <td>{{articulo.precio}}</td>  
  </tr>  
</table>
```

Si ejecutamos ahora el proyecto012 veremos en el navegador el listado de artículos:

Al principio y con problemas muy sencillos parece que solo agregamos complejidad a nuestra aplicación:

Veremos que esta forma de desacoplar el acceso a datos de las componentes y delegarla en otras clases llamadas servicios facilita el mantenimiento de nuestras aplicaciones.

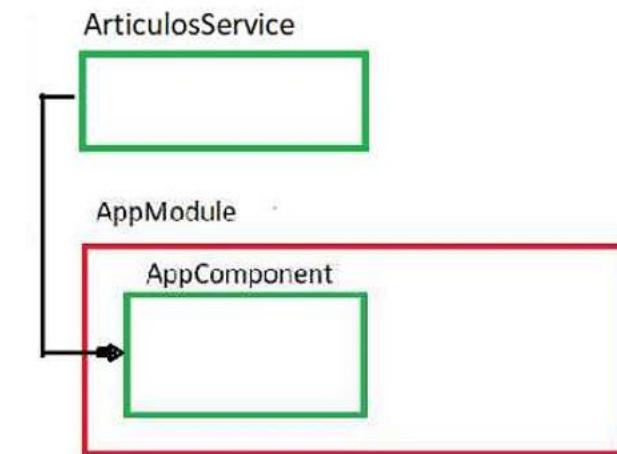
También hay que hacer notar que la forma de consumir dichas clases se hace por medio del patrón de inyección de dependencias.

Símbolo del sistema

Microsoft Windows [Versión 10.0.17134.523]

(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\codig>ng serve --open



## Servicios: recuperación de datos de un servidor web

Hemos dicho en el concepto anterior que en Angular se delega todas las responsabilidades de acceso a datos (peticiones y envío de datos) y lógica de negocios en otras clases que colaboran con las componentes y son llamados servicios.

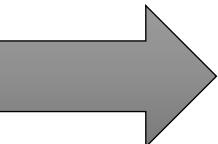
Veremos ahora como recuperar datos de un servidor web implementando dicha actividad en un servicio.

## Ejercicio

Confeccionar una aplicación que recupere una respuesta en JSON de la dirección:

<http://www.codigodata.com/datos.php>

La estructura del archivo JSON es:



```
[  
  {  
    "codigo": 1,  
    "descripcion": "papas",  
    "precio": 12.33  
  },  
  {  
    "codigo": 2,  
    "descripcion": "manzanas",  
    "precio": 54  
  }]  
]
```

Mostrar en una tabla HTML todos los artículos recuperados.

La recuperación de datos se debe hacer en un servicio.

Desde la línea de comandos de Node.js procedemos a crear el proyecto013:

```
f:\angularya> ng new proyecto013
```

Crearemos el servicio que recuperará la lista de artículos

```
f:\angularya\proyecto013> ng generate service articulos --module=app
```

Con el comando anterior estamos creando la clase 'ArticulosService' y lo estamos registrando en el módulo 'app' (recordemos que es el módulo que se crea por defecto)

Se crean dos archivos y se modifica el archivo 'app.module.ts'.

El código generado de la clase 'ArticulosService' es:

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

@Injectable()
export class ArticulosService {

  constructor() { }

}
```

### clase 'ArticulosService.ts'

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable()
export class ArticulosService {

  constructor(private http: HttpClient) { }

  retornar() {
    return this.http.get("http://scratchya.com.ar/vue/datos.php");
  }
}
```

Lo modificamos por el siguiente código que permita recuperar desde un servidor web el archivo JSON:

El archivo 'app.module.ts' se modifica con el siguiente código:

Se agrega en la propiedad 'providers' el nombre del servicio que acabamos de crear: 'ArticulosService'.

El responsabilidad del framework Angular de crear una instancia de todas las clases que definamos dentro de la propiedad 'providers'.

También se importa la clase HttpClientModule.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { ArticulosService } from './articulos.service';
import { HttpClientModule} from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [ArticulosService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

## App.component.ts

Ahora veremos como consumimos el servicio desde nuestra componente. Procedemos a modificar la componente que se crea por defecto 'AppComponent' que tiene por responsabilidad mostrar en la página el listado de artículos:

Primero importamos el servicio llamado ArticulosService que se almacena en el archivo 'articulos.service.ts':

```
import { ArticulosService } from './articulos.service';
```

Para inyectar el objeto de la clase 'ArticulosService' que crea Angular en forma automática lo hacemos en el parámetro del constructor:

```
constructor(private articulosServicio: ArticulosService) {  
}
```

En el método ngOnInit actualizamos la propiedad 'articulos' con el resultado devuelto:

```
ngOnInit() {  
    this.articulosService.retornar()  
        .subscribe( result => this.articulos = result)  
}
```

```
import { Component, OnInit } from '@angular/core';  
import { ArticulosService } from './articulos.service';  
  
@Component({  
    selector: 'app-root',  
    templateUrl: './app.component.html',  
    styleUrls: ['./app.component.css']  
})  
export class AppComponent implements OnInit{  
  
    private articulos = null;  
  
    constructor(private articulosService: ArticulosService) {}  
  
    ngOnInit() {  
        this.articulosService.retornar()  
            .subscribe( result => this.articulos = result)  
    }  
}
```

Se almacena en el atributo 'articulosServicio' la referencia del objeto de la clase 'ArticulosService' que crea Angular.

Esta asignación dispara la actualización de la página HTML.

### app.component.html

```
<div *ngIf="articulos!=null; else espera">
  <table border="1">
    <tr>
      <td>Codigo</td><td>Descripcion</td><td>Precio</td>
    </tr>
    <tr *ngFor="let art of articulos">
      <td>{{art.codigo}}</td>
      <td>{{art.descripcion}}</td>
      <td>{{art.precio}}</td>
    </tr>
  </table>
</div>
<ng-template #espera>Esperando datos...</ng-template>
```

Falta que codifiquemos la vista con los datos recuperados:

Si ejecutamos ahora el proyecto013 veremos en el navegador el listado de artículos, pero ahora recuperados de un servidor y no extraídos de un vector como en el concepto anterior

## Pipes: definición

Las pipes también llamadas tuberías o filtros son funciones que se llaman en una vista (html) y que tienen por objetivo transformar un dato a mostrar para mejorar la experiencia del usuario.

Hay una cantidad reducida de pipes que trae por defecto Angular y podemos hacer uso en cualquier vistas de nuestras componentes. Lo más importante es que podemos crear nuestras propias pipes, lo veremos en el próximo concepto.

La llamada a estas funciones tiene una sintaxis muy distinta a la tradicional y su objetivo es hacer más claro la sintaxis de nuestra vista.

Por ejemplo para mostrar el contenido de una variable toda en mayúsculas en la plantilla utilizamos la siguiente sintaxis:

```
<p>Nombre del cliente:{{ nombre | uppercase}}</p>
```

Considerando que en la componente hemos definido la propiedad:

```
nombre = 'Juan Carlos';
```

La salida en el navegador será:

Nombre del cliente: JUAN CARLOS

Como vemos en la interpolación de la propiedad 'nombre' llamamos a la pipe después del carácter '|':

La pipe 'uppercase' transforma a mayúsculas el dato que se muestra en la página, no se modifica la propiedad 'nombre'.

En algunas pipes podemos pasar parámetros:

```
<p>El saldo es:{{ saldo | currency:'$' }}</p>
```

Podemos aplicar una pipe directamente a un valor indicado en la vista:

Seguido al nombre de la pipe disponemos cada parámetro antecedido por el carácter ':'.  
•

```
<p>{{ 'Hola' | uppercase }}</p>
```

Como se trata de un string debemos indicarlo entre comillas.  
Luego cuando ejecutamos en el navegador tenemos como resultado 'HOLA'.  
•

## Ejercicio pipe angular

Confeccionar una aplicación que defina una serie de propiedades y muestre sus valores aplicando distintas pipes que trae por defecto Angular.

Desde la línea de comandos de Node.js procedemos a crear el proyecto014:

```
f:\angularya> ng new proyecto014
```

Definimos ahora en la componente 'AppComponent' una serie de propiedades:

Hemos definido 5 propiedades de distintos tipos: string, number, vector con los días, vector de artículos con objetos y un objeto de tipo Date.

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  nombre = 'Juan Carlos';
  saldo = 1000.50;
  dias = ['domingo', 'lunes', 'martes', 'miércoles', 'jueves', 'viernes', 'sábado'];
  articulos = [
    {
      codigo: 1,
      descripcion: "papas",
      precio: 12.33
    },
    {
      codigo: 2,
      descripcion: "manzanas",
      precio: 54
    }
  ];
  fechaActual = new Date();
}
```

El archivo de la vista 'app.component.html' queda con la siguiente sintaxis:

```
<p>Nombre del cliente:{{ nombre }}</p>
<p>Nombre del cliente en mayúsculas:{{ nombre | uppercase }}</p>
<p>Nombre del cliente en minúsculas:{{ nombre | lowercase }}</p>
<p>El saldo es:{{ saldo | currency:'$'}}</p>
<p>Días laborables: {{ dias | slice:1:5}}</p>
<p>{{ articulos | json }}</p>
<p>Fecha actual:{{ fechaActual | date:'d/M/y' }}</p>
```

Ya vimos las pipes: uppercase y lowercase:

```
<p>Nombre del cliente en mayúsculas:{{ nombre | uppercase }}</p>
<p>Nombre del cliente en minúsculas:{{ nombre | lowercase }}</p>
```

Para mostrar un importe monetario podemos utilizar la pipe 'currency':

```
<p>El saldo es:{{ saldo | currency:'$'}}</p>
```

Si necesitamos mostrar parte de un vector podemos utilizar la pipe 'slice' e indicar en dos parámetros a partir de que índice recuperar los datos y hasta qué índice:

```
<p>Días laborables: {{ dias | slice:1:5}}</p>
```

Otra pipe útil cuando tenemos que depurar programas es la impresión del contenido de una variable que almacena datos en formato JSON:

```
<p>{{ articulos | json }}</p>
```

Finalmente utilizamos la pipe 'date':

```
<p>Fecha actual:{{ fechaActual | date:'d/M/y' }}</p>
```

## Resultado



## Pipes: creación de pipes personalizadas

Vimos en el concepto anterior que una pipe tiene por objetivo convertir un dato en la vista de la componente (html) con el objetivo que el usuario tenga una mejor experiencia.

Vimos en el concepto anterior que una pipe tiene por objetivo convertir un dato en la vista de la componente (html) con el objetivo que el usuario tenga una mejor experiencia.

Pero lo más importante que tenemos es que Angular nos permite crear nuestras propias pipes que se adapten a resolver problemas de nuestra aplicación.

Veremos los pasos para crear una pipe haciendo uso de la herramienta Angular CLI para facilitar su codificación.

## Ejercicio

Desde la línea de comandos de Node.js procedemos a crear el proyecto015:

```
f:\angularya> ng new proyecto015
```

Para crear una 'pipe' descendemos primero a la carpeta del proyecto que acabamos de crear y mediante angular Cli procedemos a ejecutar :

```
f:\angularya\proyecto015> ng generate pipe letras
```

Luego de esto tenemos creados dos archivos:

```
letras.pipe.ts  
letras.pipe.spec.ts
```

Además se ha modificado el archivo:  
app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { LetrasPipe } from './letras.pipe';

@NgModule({
  declarations: [
    AppComponent,
    LetrasPipe
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

En este último archivo se ha importado y declarado la clase que pasaremos luego a codificar la pipe:

El siguiente paso es codificar la clase 'LetrasPipe' que se encuentra almacenada en el archivo 'letras.pipe.ts':

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'letras'
})
export class LetrasPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    return null;
  }
}
```

El método transform es el que se ejecuta cada vez que llamamos a la pipe 'letras' declarada en el decorador @Pipe.

El método transform recibe un parámetro obligatorio que es 'value' y puede ser de cualquier tipo (number, string, array, object etc.) y un segundo parámetro opcional.

Veamos la lógica como implementar la pipe que transforme un valor entero comprendido entre 1 y 7 a letras:

El parámetro 'value' recibe el valor que debe procesar la pipe. El parámetro 'args' puede no llegar en dicho caso almacena un null.

Si en 'args' llega un null luego ejecuta el último switch de la función retornando en castellano el número almacenado en 'value'

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'letras'
})
export class LetrasPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    if (args != null) {
      if (args=='ingles') {
        switch (value) {
          case 1: return 'one';
          case 2: return 'two';
          case 3: return 'three';
          case 4: return 'four';
          case 5: return 'five';
          case 6: return 'six';
          case 7: return 'seven';
        }
      if (args=='portugues') {
        switch (value) {
          case 1: return 'um';
          case 2: return 'dois';
          case 3: return 'três';
          case 4: return 'quatro';
          case 5: return 'cinco';
          case 6: return 'seis';
          case 7: return 'sete';
        }
      }
      switch (value) {
        case 1: return 'uno';
        case 2: return 'dos';
        case 3: return 'tres';
        case 4: return 'cuatro';
        case 5: return 'cinco';
        case 6: return 'seis';
        case 7: return 'siete';
      }
    }
    return null;
  }
}
```

Luego en la componente definimos por ejemplo un vector con los números del 1 al 7 ('app.component.ts'):

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  vector = [1,2,3,4,5,6,7];
}
```

En el archivo 'app.component.html' es donde hacemos uso de la pipe 'letras' que acabamos de crear:

La primer forma de llamar a la pipe sin parámetro es:

`{{valor | letras}}`

De esta forma tenemos en pantalla los valores de los números en castellano.

En el caso que le pasemos como parámetro alguno de los dos valores 'ingles' o 'portugues' tendremos como resultado la transformación del número a alguno de estos dos idiomas:

`{{valor | letras:'ingles'}}`

```
<h1>Números en castellano</h1>
<ul>
  <li *ngFor="let valor of vector">
    {{valor | letras}}
  </li>
</ul>
<h1>Números en inglés</h1>
<ul>
  <li *ngFor="let valor of vector">
    {{valor | letras:'ingles'}}
  </li>
</ul>
<h1>Números en portugués</h1>
<ul>
  <li *ngFor="let valor of vector">
    {{valor | letras:'portugues'}}
  </li>
</ul>
```

Resultado

The screenshot shows a web browser window titled "Proyecto015" with the URL "localhost:4200". The page content is organized into three main sections, each with a bold title and a bulleted list of numbers from one to seven.

- Números en castellano**
  - uno
  - dos
  - tres
  - cuatro
  - cinco
  - seis
  - siete
- Números en inglés**
  - one
  - two
  - three
  - four
  - five
  - six
  - seven
- Números en portugués**
  - um
  - dois
  - três
  - quatro
  - cinco
  - seis
  - sete

## Otro ejercicio pipe creada por nosotros

Vamos a crear una carpeta donde esten tengamos los pipes dentro de app, llámala pipes

Ahora crea un nuevo fichero, ponle de sufijo .pipe. Por ejemplo, formateaURL.pipe.ts

En nuestro caso, vamos a crear un pipe para formatear un String para que sea valido una URL, simplemente lo pondremos en minúscula y cambiar los espacios por guiones.

Usaremos esta estructura:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: ''
})
export class FormatLinksPipe implements PipeTransform {

  transform(value: String): String {
    }
}
```

Ponle el nombre a la clase que necesites  
y en el **name**: pon como quieras que se  
llame el pipe.

El método transform es lo que realiza el pipe en sí.  
Podria quedarse asi:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'formatLinks'
})
export class FormatLinksPipe implements PipeTransform {

  transform(value: String): String {
    return value.replace(' ', '-').toLowerCase();
  }
}
```

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormatLinksPipe } from './app/pipes/formateaURL.pipe';

import { AppComponent } from './app.component';
import { TestCompComponent } from './components/test-comp/test-comp.component';

@NgModule({
  declarations: [
    AppComponent,
    FormatLinksPipe,
    TestCompComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Ahora tenemos que declararlo en  
app.modules.ts.

Nuestro app.component.ts

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

@Component({
  selector: 'app-test-comp',
  templateUrl: './test-comp.component.html',
  styleUrls: ['./test-comp.component.css']
})
export class TestCompComponent implements OnInit {

  private url: String = "Enlace AL Componente"

  constructor() { }

  ngOnInit() {
  }
}
```

Y nuestro app.component.html agregamos la siguiente interpolación

`{{url | formatLinks}}`

Resultado



## TypeScript: tipado estático

Los lenguajes con tipado estático permiten detectar errores de asignación en tiempo de desarrollo.

Por ejemplo si tenemos la definición de una variable 'number' y luego queremos asignarle un 'string' la herramienta de desarrollo la puede detectar (también si lo compilamos nos detecta un error):

```
let version: number = 2.1;  
version = '2.4';
```

JavaScript es un lenguaje de programación que usa un tipado dinámico, luego la comprobación de tipificación se realiza durante su ejecución en vez de durante la compilación.

## number

Permite almacenar tanto valores enteros como reales.

```
let edad: number = 23;  
let altura: number = 1.92;
```

Si inicializamos la variable inmediatamente podemos dejar que TypeScript infiera el tipo de dato, es decir es lo mismo que escribir:

```
let edad = 23;  
let altura = 1.92;
```

## Tipo de datos básicos soportados por TypeScript.

## string

Permite almacenar una cadena de caracteres:

```
let estudios:string = 'primarios';
```

Podemos almacenar entre las comillas cualquier carácter.

## boolean

Podemos almacenar el valor true o false:

```
let activo: boolean = true;
```

## any

En las situaciones que debemos almacenar un dato en una variable y no sabemos de antemano que tipo se trata, TypeScript incorpora el tipo 'any'.

```
let dato: any;  
dato = 10;  
console.log(dato);  
dato = 'Hola';  
console.log(dato);  
dato = true;  
console.log(dato);  
dato = [1,2,3];  
console.log(dato);
```

## Arrays

Podemos almacenar un conjunto de elementos del mismo tipo mediante vectores o arreglos:

```
let vector: number[] = [1, 4, 2];  
vector.push(33);  
for(let elemento of vector)  
    console.log(elemento);
```

Indicamos luego del tipo de dato los corchetes abiertos y cerrados:

```
let vector: number[]
```

Si queremos definir e inicializar algunas componentes del arreglo luego la sintaxis es:

```
let vector: number[] = [1, 4, 2];
```

## Otra sintaxis

```
let vector: Array<number> = [1, 4, 2];  
vector.push(33);  
for(let elemento of vector)  
    console.log(elemento);
```

## Tipo de datos básicos soportados por TypeScript.

enum

Luego podemos definir una variable de este tipo y almacenar uno de esos cuatro valores:

```
enum Operacion {Suma, Resta, Multiplicacion, Division};

let actual: Operacion = Operacion.Multiplicacion;

switch (actual) {
  case Operacion.Suma: {
    console.log('Operación actual: Suma ');
    break;
  }
  case Operacion.Resta: {
    console.log('Operación actual: Resta ');
    break;
  }
  case Operacion.Multiplicacion: {
    console.log('Operación actual: Multiplicacion ');
    break;
  }
  case Operacion.Division: {
    console.log('Operación actual: Division ');
    break;
  }
}
```

Estamos declarando un nuevo tipo de dato que puede almacenar alguno de los cuatro valores indicados entre llaves:

```
let actual: Operacion = Operacion.Multiplicacion;
```

Cuando queremos saber que valor almacena la variable 'actual' la comparamos por ejemplo con los valores posibles del tipo 'Operacion':

```
switch (actual) {
  case Operacion.Suma: {
    console.log('Operación actual: Suma ');
    break;
  }
  case Operacion.Resta: {
    console.log('Operación actual: Resta ');
    break;
  }
  case Operacion.Multiplicacion: {
    console.log('Operación actual: Multiplicacion ');
    break;
  }
  case Operacion.Division: {
    console.log('Operación actual: Division ');
    break;
  }
}
```

## Parámetros de métodos.

ambién cuando definimos funciones o métodos debemos definir el tipo de dato de los mismos:

```
mayor(valor1: number, valor2: number): number {
  if (valor1 > valor2)
    return valor1;
  else
    return valor2;
}
```

El método mayor recibe dos parámetros de tipo 'number' y retorna un tipo de dato 'number' que lo indicamos luego de los dos puntos.

Si no retorna dato un método podemos utilizar el tipo 'void':

```
export class AppComponent {

  constructor() {
    this.mostrarMensaje('Hola mundo');
  }

  mostrarMensaje(mensaje: string): void {
    alert(mensaje);
  }
}
```

## Variable de tipo unión.

TypeScript permite definir variables que almacenen dos o más tipos de datos. Puede almacenar un único valor en un determinado momento, pero dicho valor puede variar entre los tipos indicados en la definición:

```
let edad: number | string;  
edad=34;  
console.log(edad);  
edad='20 años';  
console.log(edad);
```

Mediante el carácter | separamos los tipos posibles de datos que puede almacenar la variable. La variable 'edad' puede almacenar un valor de tipo 'number' o 'string':

```
| let edad: number | string;
```

## TypeScript: clases

TypeScript incorpora muchas características de la programación orientada a objetos disponibles en lenguajes como Java y C#.

La sintaxis básica de una clase puede ser:

Los atributos se definen fuera de los métodos. Para acceder a los mismos dentro de los métodos debemos anteceder la palabra clave 'this':

```
imprimir() {  
  console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);  
}
```

```
class Persona {  
  nombre: string;  
  edad: number;  
  
  constructor(nombre:string, edad:number) {  
    this.nombre = nombre;  
    this.edad = edad;  
  }  
  
  imprimir() {  
    console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);  
  }  
  
  let persona1: Persona;  
  persona1 = new Persona('Juan', 45);  
  persona1.imprimir();
```

El constructor es el primer método que se ejecuta en forma automática al crear un objeto de la clase 'Persona':

```
constructor(nombre:string, edad:number) {  
  this.nombre = nombre;  
  this.edad = edad;  
}
```

## Modificadores de acceso a propiedades y métodos.

El objeto 'dado1' tiene acceso a todos los atributos y métodos que tienen el modificador 'public' por eso podemos llamar a los métodos 'tirar' e 'imprimir':

```
dado1.tirar();
dado1.imprimir();
```

Se generará un error si queremos acceder al atributo 'valor' o al método 'generar':

```
let dado1=new Dado();
dado1.valor = 6;
dado1.generar();
```

Podemos definir propiedades y métodos privados y públicos antecediendo las palabras claves 'private' y 'public'. Veamos un ejemplo:

```
class Dado {
    private valor: number;

    public tirar() {
        this.generar();
    }

    private generar() {
        this.valor = Math.floor(Math.random() * 6) + 1 ;
    }

    public imprimir() {
        console.log(`Salió el valor ${this.valor}`);
    }
}

let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
```

Si no agregamos modificador de acceso por defecto es 'public', luego tendremos el mismo resultado si utilizamos la siguiente sintaxis para declarar la clase:

```
class Dado {
    private valor: number;

    tirar() {
        this.generar();
    }

    private generar() {
        this.valor = Math.floor(Math.random() * 6) + 1;
    }

    imprimir() {
        console.log(`Salió el valor ${this.valor}`);
    }
}

let dado1=new Dado();
dado1.tirar();
dado1.imprimir();
```

Es decir nos ahorraremos de escribir 'public' antes de las propiedades y métodos que queremos definir con esta característica.

## Definición e inicialización de propiedades en los parámetros del constructor.

Como vemos el constructor tiene un bloque de llaves vacías ya que no tenemos que implementar ningún código en su interior, pero al anteceder el modificador de acceso en la zona de parámetros los mismos pasan a ser propiedades de la clase y no parámetros:

```
constructor(public nombre:string, public edad:number) { }
```

Podemos sin problemas definir propiedades tanto 'public' como 'private'.

La definición de propiedades en la zona de parámetros solo se puede hacer en el constructor de la clase y no está permitido en cualquier otro método.

Esta característica no es común en otros lenguajes orientados a objetos y tiene por objetivo crear clases más breves.

En TypeScript podemos definir algunas propiedades de la clase en la zona de parámetros del constructor, con esto nos evitamos de su declaración fuera de los métodos.

Por ejemplo la clase Persona la podemos codificar con ésta otra sintaxis:

```
class Persona {  
  
    constructor(public nombre:string, public edad:number) { }  
  
    imprimir() {  
        console.log(`Nombre: ${this.nombre} y edad:${this.edad}`);  
    }  
  
}  
  
let personal1: Persona;  
personal1 = new Persona('Juan', 45);  
personal1.imprimir();
```

## Modificador readonly

Una vez que se inicia la propiedad 'codigo' en el constructor su valor no puede cambiar:

```
imprimir() {  
  this.codigo=7; //Error  
  console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);  
}
```

El mismo error se produce si tratamos de cambiar su valor desde fuera de la clase:

```
let articulo1: Articulo;  
articulo1 = new Articulo(1,'papas',12.5);  
articulo1.codigo=7; //Error  
articulo1.imprimir();
```

Podemos utilizar también la sintaxis abreviada de propiedades:

```
class Articulo {  
  
  constructor(readonly codigo:number, public descripcion:string, public precio:number) {}  
  
  imprimir() {  
    console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);  
  }  
}
```

Disponemos además de los modificadores 'private' y 'public' uno llamado 'readonly'. Mediante este modificador el valor de la propiedad solo puede ser cargado en el constructor o al momento de definirlo y luego no puede ser modificado ni desde un método de la clase o fuera de la clase.

Veamos un ejemplo con una propiedad 'readonly':

```
class Articulo {  
  readonly codigo: number;  
  descripcion: string;  
  precio: number;  
  
  constructor(codigo:number, descripcion:string, precio:number) {  
    this.codigo=codigo;  
    this.descripcion=descripcion;  
    this.precio=precio;  
  }  
  
  imprimir() {  
    console.log(`Código:${this.codigo} Descripción:${this.descripcion} Precio:${this.precio}`);  
  }  
  
  let articulo1: Articulo;  
  articulo1 = new Articulo(1,'papas',12.5);  
  articulo1.imprimir();
```

## Propiedades estáticas

Una propiedad estática requiere el modificador 'static' previo a su nombre:

```
static tiradas:number=0;
```

Para acceder a dichas propiedades debemos anteceder el nombre de la clase y no la palabra clave 'this':

```
Dado.tiradas++;
```

Las propiedades estáticas pertenecen a la clase y no a las instancias de la clase. Se las define antecediendo el modificador 'static'.

Con un ejemplo quedará claro este tipo de propiedades:

```
class Dado {  
    private valor: number;  
    static tiradas:number=0;  
    tirar() {  
        this.generar();  
    }  
  
    private generar() {  
        this.valor = Math.floor(Math.random() * 6) + 1 ;  
        Dado.tiradas++;  
    }  
  
    imprimir() {  
        console.log(`Salió el valor ${this.valor}`);  
    }  
}  
  
let dado1=new Dado();  
dado1.tirar();  
dado1.imprimir();  
let dado2=new Dado();  
dado2.tirar();  
dado2.imprimir();  
console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2
```

No importan cuantos objetos de la clase se definan luego todos esos objetos comparten la misma variable estática:

```
let dado1=new Dado();  
dado1.tirar();  
dado1.imprimir();  
let dado2=new Dado();  
dado2.tirar();  
dado2.imprimir();  
console.log(`Cantidad de tiradas de dados:${Dado.tiradas}`); // 2
```

Es por eso que la propiedad 'tiradas' almacena un 2 luego de tirar el primer y segundo dado.  
La propiedad 'valor' es independiente en cada dado pero la propiedad 'tiradas' es compartida por los dos objetos.

## Métodos estáticas

Debemos anteceder la palabra clave static al nombre del método. Cuando llamamos a un método debemos anteceder también el nombre de la clase, no hace falta definir una instancia u objeto de la clase:

```
let x1=Matematica.aleatorio(1,10);
```

Igual que las propiedades estáticas los métodos estáticos se los accede por el nombre de la clase. Este tipo de métodos solo pueden acceder a propiedades estáticas.

```
class Matematica {  
    static mayor(v1:number, v2: number): number {  
        if (v1>v2)  
            return v1;  
        else  
            return v2;  
    }  
  
    static menor(v1:number, v2: number): number {  
        if (v1<v2)  
            return v1;  
        else  
            return v2;  
    }  
  
    static aleatorio(inicio: number, fin: number): number {  
        return Math.floor((Math.random()*(fin+1-inicio))+inicio);  
    }  
}  
  
let x1=Matematica.aleatorio(1,10);  
let x2=Matematica.aleatorio(1,10);  
console.log(`El mayor entre ${x1} y ${x2} es ${Matematica.mayor(x1,x2)}`);  
console.log(`El menor entre ${x1} y ${x2} es ${Matematica.menor(x1,x2)}`);
```

## TypeScript: funciones y métodos

TypeScript aporta varias características que JavaScript no dispone hasta el momento cuando tenemos que plantear funciones y métodos.

### Parámetros tipados y funciones que retornan un valor.

Podemos indicar a cada parámetro el tipo de dato que puede recibir y también el tipo de dato que retorna la función o método en caso que estemos en una clase:

```
function sumar(valor1:number, valor2:number): number {  
    return valor1+valor2;  
}  
  
console.log(sumar(10, 5));
```

La función sumar recibe dos parámetros de tipo number y retorna un valor de tipo number. Luego si llamamos a esta función enviando un valor distinto a number el compilador nos avisará del error:

```
console.log(sumar('juan', 'carlos'));
```

Se genera un error:  
"Argument of type ""juan"" is not assignable to parameter of type 'number'.  
Inclusive editores de texto moderno como Visual Studio Code pueden antes de compilarse avisar del error.

El tipado estático favorece a identificar éste tipo de errores antes de ejecutar la aplicación. Lo mismo cuando una función retorna un dato debemos indicar al final de la misma dicho tipo:

La función sumar retorna un valor de tipo number

Luego si la función retorna un tipo distinto a number se genera un error:

```
function sumar(valor1:number, valor2:number): number {
```

```
function sumar(valor1:number, valor2:number): number {  
    return 'Hola mundo';  
}
```

## Funciones anónimas.

Una función anónima no especifica un nombre. Son semejantes a JavaScript con la salvedad de la definición de tipos para los parámetros:

Como estamos retornando un string se genera el error: Type "Hola mundo" is not assignable to type 'number'

```
const funcSumar = function (valor1:number, valor2:number): number {  
    return valor1 + valor2;  
}  
  
console.log(funcSumar(4, 9));
```

## Parámetros opcionales.

En TypeScript debemos agregar el carácter '?' al nombre del parámetro para indicar que el mismo puede o no llegar un dato:

El tercer parámetro es opcional:

Luego a la función 'sumar' la podemos llamar pasando 2 o 3 valores numéricos:

Si pasamos una cantidad de parámetros distinta a 2 o 3 se genera un error en tiempo de compilación: ' error TS2554: Expected 2-3 arguments, but got 4.'

```
function sumar(valor1:number, valor2:number, valor3?:number):number {  
    if (valor3)  
        return valor1+valor2+valor3;  
    else  
        return valor1+valor2;  
}  
  
console.log(sumar(5,4));  
console.log(sumar(5,4,3));
```

```
function sumar(valor1:number, valor2:number, valor3?:number):number {
```

```
console.log(sumar(5,4));  
console.log(sumar(5,4,3));
```

Los parámetrosopcionales deben ser los últimos parámetros definidos en la función. Puede tener tantos parámetrosopcionales como se necesiten.

## Parámetros por defecto.

Esta característica de TypeScript nos permite asignar un valor por defecto a un parámetro para los casos en que la llamada a la misma no se le envíe.

El tercer parámetro almacena un cero si no se lo pasamos en la llamada:

Puede haber varios valores por defecto, pero deben ser los últimos. Es decir primero indicamos los parámetros que reciben datos en forma obligatoria cuando los llamamos y finalmente indicamos aquellos que tienen valores por defecto.

```
function sumar(valor1:number, valor2:number, valor3:number=0):number {  
    return valor1+valor2+valor3;  
}  
  
console.log(sumar(5,4));  
console.log(sumar(5,4,3));
```

`console.log(sumar(5,4));`

## parámetros Rest.

Otra característica de TypeScript es la posibilidad de pasar una lista indefinida de valores y que los reciba un vector.

El concepto de parámetro Rest se logra antecediendo tres puntos al nombre del parámetro:

El parámetro 'valores' se le anteceden los tres puntos seguidos e indicamos que se trata de un vector de tipo 'number'. Cuando llamamos a la función le pasamos una lista de valores enteros que luego la función los empaqueta en el vector:

La función con un parámetro Rest puede tener otros parámetros pero se deben declarar antes.  
Los parámetros Rest no pueden tener valores por defecto.

```
function sumar(...valores:number[]) {  
    let suma=0;  
    for(let x=0;x<valores.length;x++)  
        suma+=valores[x];  
    return suma;  
}  
  
console.log(sumar(10, 2, 44, 3));  
console.log(sumar(1, 2));  
console.log(sumar());
```

```
console.log(sumar(10, 2, 44, 3));  
console.log(sumar(1, 2));  
console.log(sumar());
```

## operador Spread.

El operador Spread permite descomponer una estructura de datos en elementos individuales. Es la operación inversa de los parámetros Rest. La sintaxis se aplica anteponiendo al nombre de la variable tres puntos:

```
function sumar(valor1:number, valor2:number, valor3:number):number {  
    return valor1+valor2+valor3;  
}  
  
const vec:number[]=[10,20,30];  
const s=sumar(...vec);  
console.log(s);
```

## Funciones callbacks

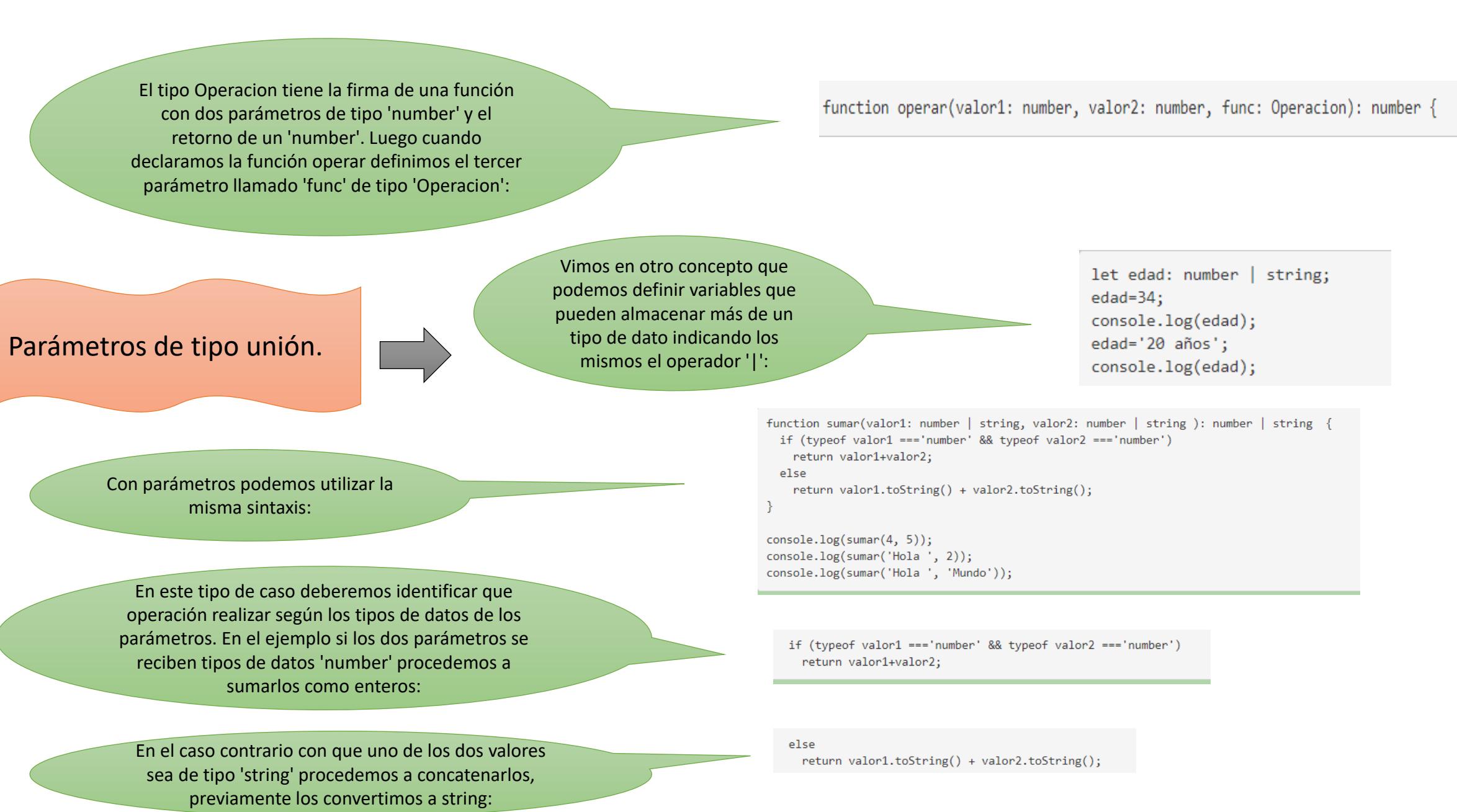
Una función callback es una función que se pasa a otra función como parámetro y dentro de la misma es llamada.

La función operar recibe tres parámetros, los dos primeros son de tipo 'number' y el tercero es de tipo función:

La función que debe recibir debe tener como parámetros dos 'number' y retornar un 'number'. Cuando llamamos a la función además de los dos enteros le debemos pasar una función que reciba dos 'number' y retorne un 'number':

Como podemos observar llamamos a la función 'operar' tres veces y le pasamos funciones que procesan los dos enteros para obtener su suma, resta y multiplicación. Para hacer más claro nuestro código TypeScript mediante la palabra clave type permite crear nuevos tipos y luego reutilizarlos:

```
function operar(valor1: number, valor2: number, func: (x: number, y:number)=>number): number {  
  
    console.log(operar(3, 7, (x: number,y: number): number => {  
        return x+y;  
    }));  
  
    console.log(operar(3, 7, (x: number,y: number): number => {  
        return x-y;  
    }));  
  
    console.log(operar(3, 7, (x: number,y: number): number => {  
        return x*y;  
    }));  
  
}  
  
function operar(valor1: number, valor2: number, func: Operacion): number {  
    return func(valor1, valor2);  
}  
  
type Operacion=(x: number, y:number)=>number;  
  
console.log(operar(3, 7, (x: number,y: number): number => {  
    return x+y;  
}));  
  
console.log(operar(3, 7, (x: number,y: number): number => {  
    return x-y;  
}));  
  
console.log(operar(3, 7, (x: number,y: number): number => {  
    return x*y;  
}));
```



## Acotaciones

Hemos hecho siempre ejemplos con funciones, pero todos estos conceptos se aplican si planteamos métodos dentro de una clase:

```
class Operacion {  
    sumar(...valores:number[]) {  
        let suma=0;  
        for(let x=0;x<valores.length;x++)  
            suma+=valores[x];  
        return suma;  
    }  
}  
  
let op: Operacion;  
op=new Operacion();  
console.log(op.sumar(1,2,3));
```

## TypeScript: herencia

La herencia significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todas los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otros atributos y métodos propios.

La herencia es otra característica fundamental de la programación orientada a objetos y TypeScript lo implementa.

Veamos con un ejemplo la sintaxis que plantea TypeScript para implementar la herencia:

```
class Persona {  
    protected nombre:string;  
    protected edad:number;  
    constructor(nombre:string, edad:number) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    imprimir() {  
        console.log(`Nombre: ${this.nombre}`);  
        console.log(`Edad: ${this.edad}`);  
    }  
}  
  
class Empleado extends Persona {  
    private sueldo:number;  
    constructor(nombre:string, edad:number, sueldo:number) {  
        super(nombre, edad);  
        this.sueldo = sueldo;  
    }  
  
    imprimir() {  
        super.imprimir();  
        console.log(`Sueldo: ${this.sueldo}`);  
    }  
  
    pagaImpuestos() {  
        if (this.sueldo>5000)  
            console.log(`${this.nombre} debe pagar impuestos`);  
        else  
            console.log(`${this.nombre} no debe pagar impuestos`);  
    }  
}  
  
const personal=new Persona('Juan', 44);  
personal.imprimir();  
  
const empleado1=new Empleado('Ana', 22, 7000);  
empleado1.imprimir();  
empleado1.pagaImpuestos();
```

Mediante la palabra clave extends indicamos el nombre de la clase padre. Una clase puede heredar de una sola clase (en este ejemplo 'Persona'):

```
class Empleado extends Persona {  
    private sueldo:number;  
    constructor(nombre:string, edad:number, sueldo:number) {  
        super(nombre, edad);  
        this.sueldo = sueldo;  
    }  
}
```

La subclase Empleado puede acceder a las propiedades de la clase padre si los mismos se definieron en forma public o protected:

```
imprimir() {  
    console.log(`Sueldo: ${this.sueldo} y lo cobra ${this.nombre}`);  
}
```

Con el modificador protected permitimos que la subclase pueda acceder a los atributos de la clase padre pero luego donde definamos un objeto de esta clase no los pueda acceder y permanezcan encapsulados:

```
const empleado1=new Empleado('Ana', 22, 7000);  
empleado1.nombre='facundo'; //error
```

## Clases abstractas

No se pueden definir objetos de una clase abstracta y seguramente será heredada por otras clases de las que si podremos definir objetos.

En algunas situaciones tenemos métodos y propiedades comunes a un conjunto de clases, podemos agrupar dichos métodos y propiedades en una clase abstracta.

Hay una sintaxis especial en TypeScript para indicar que una clase es abstracta.

```
abstract class Operacion {
    public valor1: number;
    public valor2: number;
    public resultado: number=0;

    constructor(v1:number, v2:number) {
        this.valor1=v1;
        this.valor2=v2;
    }

    abstract operar():void;

    imprimir() {
        console.log(`Resultado: ${this.resultado}`);
    }
}

class Suma extends Operacion {
    constructor(v1:number, v2:number) {
        super(v1,v2);
    }

    operar() {
        this.resultado = this.valor1 + this.valor2;
    }
}

class Resta extends Operacion {
    constructor(v1:number, v2:number) {
        super(v1,v2);
    }

    operar() {
        this.resultado = this.valor1 - this.valor2;
    }
}

let sumal: Suma;
sumal=new Suma(10, 4);
sumal.operar();
sumal.imprimir();

let restal: Resta;
restal=new Resta(10, 4);
restal.operar();
restal.imprimir();
```

Problema: Declarar una clase abstracta que represente una Operación. Definir en la misma tres propiedades valor1, valor2 y resultado, y tres métodos: constructor, imprimir y operar (éste último hacerlo abstracto).

Plantear dos clases llamadas Suma y Resta que hereden de la clase Operación e implementen el método abstracto operar.

Mediante la palabra clave `abstract` indicamos que la clase debe definirse como abstracta, luego no se pueden definir objetos de la clase `Operacion`:

```
abstract class Operacion {  
    public valor1: number;  
    public valor2: number;  
    public resultado: number=0;  
  
    constructor(v1:number, v2:number) {  
        this.valor1=v1;  
        this.valor2=v2;  
    }  
  
    abstract operar():void;  
  
    imprimir() {  
        console.log(`Resultado: ${this.resultado}`);  
    }  
}
```

Dentro de la clase abstracta definimos un método abstracto llamado `operar`, esto obliga a todas las clases que heredan de `Operación` implementar el algoritmo de dicho método, sino se genera un error en tiempo de compilación.

La subclase `Suma` al heredar de `Operación` implementa el método `operar`:

```
class Suma extends Operacion {  
    constructor(v1:number, v2:number) {  
        super(v1,v2);  
    }  
  
    operar() {  
        this.resultado = this.valor1 + this.valor2;  
    }  
}
```

Solo podemos definir objetos de las clases `Suma` y `Resta`. Se genera un error si tratamos de crear un objeto de la clase `Operacion`:

```
let suma1: Suma;  
suma1=new Suma(10, 4);  
suma1.operar();  
suma1.imprimir();
```

## TypeScript: interfaces

Una interface declara una serie de métodos y propiedades que deben ser implementados luego por una o más clases.

Las interfaces vienen a suplir la imposibilidad de herencia múltiple.

Por ejemplo podemos tener dos clases que representen un avión y un helicóptero. Luego plantear una interface con un método llamado volar. Las dos clases pueden implementar dicha interface y codificar el método volar (los algoritmos seguramente sean distintos pero el comportamiento de volar es común tanto a un avión como un helicóptero)

La sintaxis en TypeScript para declarar una interface es:

```
interface [nombre de la interface] {  
    [declaración de propiedades]  
    [declaración de métodos]  
}
```

## Ejercicio

Definir una interface llamada Punto que declare un método llamado imprimir. Luego declarar dos clases que la implementen.

```
interface Punto {
    imprimir(): void;
}

class PuntoPlano implements Punto{
    constructor(private x:number, private y:number) {}

    imprimir() {
        console.log(`Punto en el plano: (${this.x},${this.y})`);
    }
}

class PuntoEspacio implements Punto{
    constructor(private x:number, private y:number, private z:number) {}

    imprimir() {
        console.log(`Punto en el espacio: (${this.x},${this.y},${this.z})`);
    }
}

let puntoPlanol: PuntoPlano;
puntoPlanol = new PuntoPlano(10, 4);
puntoPlanol.imprimir();

let puntoEspacio1: PuntoEspacio;
puntoEspacio1 = new PuntoEspacio(20, 50, 60);
puntoEspacio1.imprimir();
```

Para declarar una interface en TypeScript utilizamos la palabra clave `interface` y seguidamente su nombre. Luego entre llaves indicamos todas las cabeceras de métodos y propiedades. En nuestro ejemplo declaramos la interface `Punto` e indicamos que quien la implemente debe definir un método llamado `imprimir` sin parámetros y que no retorna nada:

```
interface Punto {  
    imprimir(): void;  
}
```

Por otro lado declaramos dos clases llamados `PuntoPlano` con dos propiedades y `PuntoEspacio` con tres propiedades, además indicamos que dichas clases implementarán la interface `Punto`:

```
class PuntoPlano implements Punto{  
    constructor(private x:number, private y:number) {}  
  
    imprimir() {  
        console.log(`Punto en el plano: (${this.x},${this.y})`);  
    }  
}  
  
class PuntoEspacio implements Punto{  
    constructor(private x:number, private y:number, private z:number) {}  
  
    imprimir() {  
        console.log(`Punto en el espacio: (${this.x},${this.y},${this.z})`);  
    }  
}
```

La sintaxis para indicar que una clase implementa una interface requiere disponer la palabra clave implements y en forma seguida el o los nombres de interfaces a implementar. Si una clase hereda de otra también puede implementar una o más interfaces.

El método imprimir en cada clase se implementa en forma distinta, en uno se imprimen 3 propiedades y en la otra se imprimen 2 propiedades.

```
let puntoPlano1: PuntoPlano;  
puntoPlano1 = new PuntoPlano(10, 4);  
puntoPlano1.imprimir();  
  
let puntoEspacio1: PuntoEspacio;  
puntoEspacio1 = new PuntoEspacio(20, 50, 60);  
puntoEspacio1.imprimir();
```

Luego definimos un objeto de la clase PuntoPlano y otro de tipo PuntoEspacio:

Si una clase indica que implementa una interfaz y luego no se la codifica, se genera un error en tiempo de compilación informándonos de tal situación (inclusive el editor Visual Studio Code detecta dicho error antes de compilar):

```
prueba.ts(5,7): error TS2420: Class 'PuntoPlano' incorrectly implements interface 'Punto'.  
  Property 'imprimir' is missing in type 'PuntoPlano'.  
prueba.ts(20,13): error TS2339: Property 'imprimir' does not exist on type 'PuntoPlano'.
```

Este error se produce si codificamos la clase sin implementar el método imprimir:

```
class PuntoPlano implements Punto{  
  constructor(private x:number, private y:number) {}  
}
```

```
interface Figura {
    superficie: number;
    perimetro: number;
    calcularSuperficie(): number;
    calcularPerimetro(): number;
}

class Cuadrado implements Figura {
    superficie: number;
    perimetro: number;
    constructor(private lado:number) {
        this.superficie = this.calcularSuperficie();
        this.perimetro = this.calcularPerimetro();
    }

    calcularSuperficie(): number {
        return this.lado * this.lado;
    }

    calcularPerimetro(): number {
        return this.lado * 4;
    }
}

class Rectangulo implements Figura {
    superficie: number;
    perimetro: number;
    constructor(private ladoMayor:number, private ladoMenor:number) {
        this.superficie = this.calcularSuperficie();
        this.perimetro = this.calcularPerimetro();
    }

    calcularSuperficie(): number {
        return this.ladoMayor * this.ladoMenor;
    }

    calcularPerimetro(): number {
        return (this.ladoMayor * 2) + (this.ladoMenor * 2);
    }
}

let cuadrado1: Cuadrado;
cuadrado1 = new Cuadrado(10);
console.log('Perímetro del cuadrado : ' + cuadrado1.calcularPerimetro());
console.log('Superficie del cuadrado : ' + cuadrado1.calcularSuperficie());
let rectangulo1: Rectangulo;
rectangulo1 = new Rectangulo(10, 5);
console.log('Perímetro del rectángulo : ' + rectangulo1.calcularPerimetro());
console.log('Superficie del cuadrado : ' + rectangulo1.calcularSuperficie());
```

## Ejercicio

Se tiene la siguiente interface:

```
interface Figura {
    superficie: number;
    perimetro: number;
    calcularSuperficie(): number;
    calcularPerimetro(): number;
}
```

Declarar dos clases que representen un Cuadrado y un Rectángulo.  
Implementar la interface Figura en ambas clases.

En este problema la interface Figura tiene dos métodos que deben ser implementados por las clases y dos propiedades que también deben definirlos:

```
interface Figura {  
    superficie: number;  
    perimetro: number;  
    calcularSuperficie(): number;  
    calcularPerimetro(): number;  
}
```

La clase Cuadrado tiene una propiedad llamada lado que la recibe el constructor.  
De forma similar la clase Rectangulo implementa la interface Figura:

```
class Rectangulo implements Figura {  
    superficie: number;  
    perimetro: number;  
    constructor(private ladoMayor:number, private ladoMenor:number) {  
        this.superficie = this.calcularSuperficie();  
        this.perimetro = this.calcularPerimetro();  
    }  
  
    calcularSuperficie(): number {  
        return this.ladoMayor * this.ladoMenor;  
    }  
  
    calcularPerimetro(): number {  
        return (this.ladoMayor * 2) + (this.ladoMenor * 2);  
    }  
}
```

La clase Cuadrado indica que implementa la interface Figura, esto hace necesario que se implementen los métodos calcularSuperficie y calcularPerimetro, y las dos propiedades:

```
class Cuadrado implements Figura {  
    superficie: number;  
    perimetro: number;  
    constructor(private lado:number) {  
        this.superficie = this.calcularSuperficie();  
        this.perimetro = this.calcularPerimetro();  
    }  
  
    calcularSuperficie(): number {  
        return this.lado * this.lado;  
    }  
  
    calcularPerimetro(): number {  
        return this.lado * 4;  
    }  
}
```

Finalmente definimos un objeto de la clase Cuadrado y otro de la clase Rectangulo, luego llamamos a los métodos calcularPerimetro y calcularSuperficie para cada objeto:

```
let cuadrado1: Cuadrado;  
cuadrado1 = new Cuadrado(10);  
console.log('Perímetro del cuadrado : ${cuadrado1.calcularPerimetro()}');  
console.log('Superficie del cuadrado : ${cuadrado1.calcularSuperficie()}');  
let rectangulo1: Rectangulo;  
rectangulo1 = new Rectangulo(10, 5);  
console.log('Perímetro del rectángulo : ${rectangulo1.calcularPerimetro()}');  
console.log('Superficie del cuadrado : ${rectangulo1.calcularSuperficie()}');
```

Las interfaces exige que una clase siga las especificaciones de la misma y se implementen algoritmos más robustos. En nuestro ejemplo tanto la clase Rectangulo como Cuadrado tienen una forma similar de trabajar gracias a que implementan la interfaz Figura.

## Parámetros de tipo interface.

```
interface Figura {
    superficie: number;
    perimetro: number;
    calcularSuperficie(): number;
    calcularPerimetro(): number;
}

class Cuadrado implements Figura {
    superficie: number;
    perimetro: number;
    constructor(private lado:number) {
        this.superficie = this.calcularSuperficie();
        this.perimetro = this.calcularPerimetro();
    }

    calcularSuperficie(): number {
        return this.lado * this.lado;
    }

    calcularPerimetro(): number {
        return this.lado * 4;
    }
}

class Rectangulo implements Figura {
    superficie: number;
    perimetro: number;
    constructor(private ladoMayor:number, private ladoMenor:number) {
        this.superficie = this.calcularSuperficie();
        this.perimetro = this.calcularPerimetro();
    }

    calcularSuperficie(): number {
        return this.ladoMayor * this.ladoMenor;
    }

    calcularPerimetro(): number {
        return (this.ladoMayor * 2) + (this.ladoMenor * 2);
    }
}

function imprimir(fig: Figura) {
    console.log(`Perímetro: ${fig.calcularPerimetro()}`);
    console.log(`Superficie: ${fig.calcularSuperficie()}`);
}

let cuadrado1: Cuadrado;
cuadrado1 = new Cuadrado(10);
console.log('Datos del cuadrado');
imprimir(cuadrado1);
let rectangulo1: Rectangulo;
rectangulo1 = new Rectangulo(10, 5);
console.log('Datos del rectángulo');
imprimir(rectangulo1);
```

Un método o función puede recibir como parámetro una interface. Luego le podemos pasar objetos de distintas clases que implementan dicha interface:

La función imprimir recibe como parámetro fig que es de tipo Figura:

```
function imprimir(fig: Figura) {
    console.log(`Perímetro: ${fig.calcularPerimetro()}`);
    console.log(`Superficie: ${fig.calcularSuperficie()}`);
}
```

Podemos luego llamar a la función imprimir pasando tanto objetos de la clase Cuadrado como Rectangulo:

```
imprimir(cuadrado1);

imprimir(rectangulo1);
```

Es importante notar que solo podemos acceder a los métodos y propiedades definidos en la interface y no a propiedades y métodos propios de cada clase.

## Creación de objetos a partir de una interface.

```
interface Punto {  
    x: number;  
    y: number;  
}  
  
let punto1: Punto;  
punto1 = {x:10, y:20};  
console.log(punto1);
```

TypeScript permite crear objetos a partir de una interfaz. La sintaxis para dicha creación es:

No podemos utilizar el operador new para la creación del objeto.  
Podemos definir la variable e inmediatamente iniciarla:

```
let punto1: Punto = {x:10, y:20};
```

## Propiedades opcionales.

```
interface Punto {  
    x: number;  
    y: number;  
    z?: number;  
}  
  
let puntoPlano: Punto = {x:10, y:20};  
console.log(puntoPlano);  
  
let puntoEspacio: Punto = {x:10, y:20, z:70};  
console.log(puntoEspacio);
```

Una interface puede definir propiedades opcionales que luego la clase que la implementa puede o no definirlas. Se utiliza la misma sintaxis de los parámetros opcionales, es decir se le agrega el carácter '?' al final del nombre de la propiedad.

Como vemos el objeto 'puntoPlano' solo implementa las propiedades 'x' e 'y'. Se produce un error en tiempo de compilación si no implementamos todas las propiedades obligatorias, por ejemplo:

Esta línea genera un error ya que solo se define la propiedad 'x' y falta definir la propiedad 'y'.

```
let puntoPlano: Punto = {x:10};
```

## Herencia de interfaces.

TypeScript permite que una interface herede de otra:

```
interface Punto {  
    x: number;  
    y: number;  
}  
  
interface Punto3D extends Punto {  
    z: number;  
}  
  
let punto1: Punto = {x:10, y:20};  
let punto2: Punto3D = {x:23, y:13, z:12};  
console.log(punto1);  
console.log(punto2);
```

## TypeScript: clases genéricas

```
class PilaEnteros
{
    private vec:number[][]=[];
    insertar(x: number) {
        this.vec.push(x);
    }
    extraer() {
        if (this.vec.length>0)
            return this.vec.pop();
        else
            return null;
    }
}

class PilaStrings
{
    private vec:string[][]=[];
    insertar(x: string) {
        this.vec.push(x);
    }
    extraer() {
        if (this.vec.length>0)
            return this.vec.pop();
        else
            return null;
    }
}

let pilal=new PilaEnteros();
pilal.insertar(20);
pilal.insertar(43);
pilal.insertar(1);
console.log(pilal.extraer());

let pila2=new PilaStrings();
pila2.insertar('juan');
pila2.insertar('ana');
pila2.insertar('luis');
console.log(pila2.extraer());
```

TypeScript permite crear clases y funciones que administren distintos tipos de datos. Esta característica no existe en JavaScript

Se utilizan mucho para la administración de colecciones de datos (pilas, colas, listas etc.)

Para entender las ventajas de definir clases genéricas implementaremos los algoritmos para administrar una pila de enteros y una pila de string. Primero lo haremos utilizando clases tradicionales y luego mediante una clase genérica.

Para concentrarnos en la sintaxis plantearemos la pila utilizando un vector donde definiremos los dos métodos fundamentales de insertar y extraer (no haremos ningún tipo de validaciones por simplicidad)

Como podemos analizar hemos planteado dos clases, una para administrar una pila con tipos de dato enteros:

Y por otro lado otra clase para administrar una pila de tipo de dato string:

Para probar estas dos clases definimos un objeto de la clase PilaEnteros e insertamos tres valores y luego extraemos uno:

De forma similar probamos creando un objeto de la clase PilaStrings:

```
class PilaEnteros
{
    private vec:number[][]=[];
    insertar(x: number) {
        this.vec.push(x);
    }
    extraer() {
        if (this.vec.length>0)
            return this.vec.pop();
        else
            return null;
    }
}
```

```
class PilaStrings
{
    private vec:string[][]=[];
    insertar(x: string) {
        this.vec.push(x);
    }
    extraer() {
        if (this.vec.length>0)
            return this.vec.pop();
        else
            return null;
    }
}
```

```
let pila1=new PilaEnteros();
pila1.insertar(20);
pila1.insertar(43);
pila1.insertar(1);
console.log(pila1.extraer()); //se imprime un 1 ya que el Último en entrar es el primero en salir en una pila.
```

```
let pila2=new PilaStrings();
pila2.insertar('juan');
pila2.insertar('ana');
pila2.insertar('luis');
console.log(pila2.extraer()); //se imprime 'luis'
```

Hasta este momento no hemos presentado ninguna novedad con respecto a lo que conocemos. Veamos ahora como podemos resolver este problema pero empleando una clase genérica:

```
class PilaGenerica<T>{
    private vec:T[][]= [];

    insertar(x: T) {
        this.vec.push(x);
    }

    extraer() {
        if (this.vec.length>0)
            return this.vec.pop();
        else
            return null;
    }
}

let pila3:PilaGenerica<number>;
pila3=new PilaGenerica<number>();
pila3.insertar(20);
pila3.insertar(42);
pila3.insertar(1);
console.log(pila3.extraer());

let pila4:PilaGenerica<string>;
pila4=new PilaGenerica<string>();
pila4.insertar('juan');
pila4.insertar('ana');
pila4.insertar('luis');
console.log(pila4.extraer());
```

Como vemos hemos declarado una sola clase llamada PilaGenerica y hemos sustituido en los lugares donde hacíamos referencia a number o string por el tipo 'T' que también tenemos que hacer referencia en donde declaramos la clase:

```
class PilaGenerica<T>{
    private vec:T[]=[];
    insertar(x: T) {
        this.vec.push(x);
    }
    extraer() {
        if (this.vec.length>0)
            return this.vec.pop();
        else
            return null;
    }
}
```

Luego cuando creamos un objeto de la clase PilaGenerica debemos indicar cuando la creamos el tipo de datos que administrará nuestra pila:

```
let pila4:PilaGenerica<string>;
pila4=new PilaGenerica<string>();
```

Podemos crear objetos de la clase PilaGenerica con cualquier tipo de dato primitivo (number, string, boolean etc.) o de otra clase.

Procedemos a codificar otro programa que cree una pila de otra clase creada por nosotros:

```
class PilaGenerica<T>{
    private vec:T[][]= [];

    insertar(x: T) {
        this.vec.push(x);
    }

    extraer() {
        if (this.vec.length>0)
            return this.vec.pop();
        else
            return null;
    }
}

class Persona {
    constructor(public nombre:string, public edad:number) { }
}

let pila5:PilaGenerica<Persona>;
pila5=new PilaGenerica<Persona>();
pila5.insertar(new Persona('pedro', 33));
pila5.insertar(new Persona('maria', 33));
pila5.insertar(new Persona('marcos', 33));
console.log(pila5.extraer());
```

Creamos un objeto de la clase PilaGenerica e indicamos que almacenará objetos de la clase Persona:

```
let pila5:PilaGenerica<Persona>;  
pila5=new PilaGenerica<Persona>();
```

Creamos tres objetos de la clase Persona y los insertamos en la Pila:

```
pila5.insertar(new Persona('pedro', 33));  
pila5.insertar(new Persona('maria', 33));  
pila5.insertar(new Persona('marcos', 33));
```

Si extraemos un elemento de la pila tenemos la última persona ingresada:

```
console.log(pila5.extraer());
```

El planteo de clases genéricas nos reduce tener que crear múltiples clases para administrar distintos tipos de datos.

## Proyecto

Existen muchas tecnologías para procesar los datos que envía y recibe la aplicación Angular, una de las más extendidas es el lenguaje PHP y mediante este acceder a una base de datos MySQL.

Hemos dicho que Angular es un framework para el desarrollo de aplicaciones web de una sola página. Una situación muy común es tener que almacenar en un servidor de internet los datos que se ingresan en la aplicación Angular.

En este concepto dejaremos en forma muy clara todos los pasos que debemos desarrollar tanto en el cliente (aplicación angular) como en el servidor (aplicación PHP y MySQL).

Confeccionar una aplicación que permita administrar una lista de artículos, cada artículo almacena el código, descripción y precio. Se debe poder agregar, borrar y modificar los datos de un artículo almacenados en una base de datos MySQL y accedida con un programa en PHP.

Desde la línea de comandos de Node.js procedemos a crear el proyecto016:

```
f:\angularya> ng new proyecto016
```

Crearemos el servicio 'articulos' para ello utilizamos Angular CLI:

```
f:\angularya\proyecto016> ng generate service articulos --module=app
```

Recordemos que la propuesta del framework de Angular es delegar todas las responsabilidades de acceso a datos (peticiones y envío de datos) y lógica de negocios en otras clases que colaboran con las componentes. Estas clases en Angular se las llama servicios.

Presentaremos primero todos los archivos y luego explicaremos como se relacionan.

Con el comando anterior estamos creando la clase 'ArticulosService' y lo estamos registrando en el módulo 'app' (recordemos que es el módulo que se crea por defecto al crear un proyecto)

El código que debemos guardar en el archivo 'articulos.service.ts' es:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class ArticulosService {

  url='http://scratchya.com.ar/angular/problema016/'; // disponer url de su servidor que

  constructor(private http: HttpClient) { }

  recuperarTodos() {
    return this.http.get(` ${this.url}recuperartodos.php`);
  }

  alta(articulo) {
    return this.http.post(` ${this.url}alta.php`, JSON.stringify(articulo));
  }

  baja(codigo:number) {
    return this.http.get(` ${this.url}baja.php?codigo=${codigo}`);
  }

  seleccionar(codigo:number) {
    return this.http.get(` ${this.url}seleccionar.php?codigo=${codigo}`);
  }

  modificacion(articulo) {
    return this.http.post(` ${this.url}modificacion.php`, JSON.stringify(articulo));
  }
}
```

El archivo 'app.module.ts' se modifica con el siguiente código:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { ArticulosService } from './articulos.service';
import { HttpClientModule} from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [ArticulosService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Nuestra única componente debe implementar el siguiente código en el archivo 'app.component.ts':

```
import { Component, OnInit } from '@angular/core';
import { ArticulosService } from './articulos.service';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  articulos=null;
  ast={
    codigo:null,
    descripcion:null,
    precio:null
  }

  constructor(private articulosServicio: ArticulosService) {}

  ngOnInit() {
    this.recuperarTodos();
  }

  recuperarTodos() {
    this.articulosServicio.recuperarTodos().subscribe(result => this.articulos = result);
  }

  alta() {
    this.articulosServicio.alta(this.ast).subscribe(datos => {
      if (datos['resultado']=='OK') {
        alert(datos['mensaje']);
        this.recuperarTodos();
      }
    });
  }

  baja(codigo) {
    this.articulosServicio.baja(codigo).subscribe(datos => {
      if (datos['resultado']=='OK') {
        alert(datos['mensaje']);
        this.recuperarTodos();
      }
    });
  }

  modificacion() {
    this.articulosServicio.modificacion(this.ast).subscribe(datos => {
      if (datos['resultado']=='OK') {
        alert(datos['mensaje']);
        this.recuperarTodos();
      }
    });
  }

  seleccionar(codigo) {
    this.articulosServicio.seleccionar(codigo).subscribe(result => this.ast = result[0]);
  }

  hayRegistros() {
    return true;
  }
}
```

Y el archivo 'app.component.html'  
con:

```
<div>
  <h1>Administración de artículos</h1>
  <table border="1" *ngIf="hayRegistros(); else sinarticulos">
    <tr>
      <td>Código</td><td>Descripción</td><td>Precio</td><td>Borrar</td><td>Seleccionar</td>
    </tr>
    <tr *ngFor="let art of articulos">
      <td>{{art.codigo}}</td>
      <td>{{art.descripcion}}</td>
      <td>{{art.precio}}</td>
      <td><button (click)="baja(art.codigo)">Borrar?</button></td>
      <td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>
    </tr>
  </table>
  <ng-template #sinarticulos><p>No hay artículos.</p></ng-template>
  <div>
    <p>
      descripción:<input type="text" [(ngModel)]="art.descripcion" />
    </p>
    <p>
      precio:<input type="number" [(ngModel)]="art.precio" />
    </p>
    <p><button (click)="alta()">Agregar</button>
      <button (click)="modificacion()">Modificar</button></p>
    </div>
  </div>
```

odos los archivos presentados son los necesarios en Angular, veamos ahora que tenemos en PHP y MySQL.

Tenemos una serie de archivos PHP que reciben datos en formato JSON y retornan también un JSON.

Primero debemos crear una base de datos en MySQL llamada 'bd1' y crear la siguiente tabla:

```
CREATE TABLE articulos (
    codigo int AUTO_INCREMENT,
    descripcion varchar(50),
    precio float,
    PRIMARY KEY (codigo)
)
```

El archivo 'recuperartodos.php' retorna en formato JSON todos los artículos:

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");
$con=retornarConexion();

$registros=mysqli_query($con,"select codigo, descripcion, precio from articulos");

while ($reg=mysqli_fetch_array($registros))
{
    $vec[]=$reg;
}

$cad=json_encode($vec);
echo $cad;
header('Content-Type: application/json');

?>
```

El archivo 'alta.php' recibe los datos en formato JSON y los almacena en la tabla:

```
<?php
    header('Access-Control-Allow-Origin: *');
    header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

    $json = file_get_contents('php://input');

    $params = json_decode($json);

    require ("conexion.php");
    $con=retornarConexion();

    mysqli_query($con,"insert into articulos(descripcion,precio) values
        ('$params->descripcion','$params->precio')");

    class Result {}

    $response = new Result();
    $response->resultado = 'OK';
    $response->mensaje = 'datos grabados';

    header('Content-Type: application/json');
    echo json_encode($response);
?>
```

El archivo 'baja.php':

```
<?php
    header('Access-Control-Allow-Origin: *');
    header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept")

    require("conexion.php");
    $con=retornarConexion();

    mysqli_query($con,"delete from articulos where codigo=$_GET[codigo]");

    class Result {}

    $response = new Result();
    $response->resultado = 'OK';
    $response->mensaje = 'articulo borrado';

    header('Content-Type: application/json');
    echo json_encode($response);
?>
```

El archivo 'modificacion.php':

```
<?php
    header('Access-Control-Allow-Origin: *');
    header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

    $json = file_get_contents('php://input');

    $params = json_decode($json);

    require("conexion.php");
    $con=retornarConexion();

    mysqli_query($con,"update articulos set descripcion='".$params->descripcion',
                precio=$params->precio
                where codigo=$params->codigo");

    class Result {}

    $response = new Result();
    $response->resultado = 'OK';
    $response->mensaje = 'datos modificados';

    header('Content-Type: application/json');
    echo json_encode($response);
?>
```

El archivo 'seleccionar.php':

```
<?php
    header('Access-Control-Allow-Origin: *');
    header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

    require ("conexion.php");
    $con=retornarConexion();

    $registros=mysqli_query($con,"select codigo, descripcion, precio from articulos where c
    if ($reg=mysqli_fetch_array($registros))
    {
        $vec[]=$reg;
    }

    $cad=json_encode($vec);
    echo $cad;
    header('Content-Type: application/json');

?>
```

El archivo 'conexion.php':

```
<?php
function retornarConexion() {
    $con=mysqli_connect("localhost","root","","bd1");
    return $con;
}
?>
```

El resultado de ejecutar esta aplicación en el navegador, teniendo en funcionamiento el servidor con PHP y MySQL es:

The screenshot shows a web browser window titled "Proyecto016" with the URL "localhost:4200". The main content is a table titled "Administración de artículos" displaying four rows of article data. Below the table are input fields for adding new data, and at the bottom are "Agregar" and "Modificar" buttons.

| Codigo | Descripcion | Precio | Borrar  | Seleccionar |
|--------|-------------|--------|---------|-------------|
| 1      | papas       | 23     | Borrar? | Seleccionar |
| 2      | manzanas    | 42     | Borrar? | Seleccionar |
| 3      | naranjas    | 25     | Borrar? | Seleccionar |
| 4      | lechuga     | 65     | Borrar? | Seleccionar |

descripcion:

precio:  ₡

**Agregar** **Modificar**

## Explicación

Ahora veremos como funciona esta aplicación cliente/servidor implementada con Angular en el cliente y PHP en el servidor.

Recuperación  
de todos los  
registros

Inmediatamente se inicia la aplicación Angular se crea el objeto de la clase 'AppComponent' (nuestra única componente), en esta clase debe llegar al constructor el objeto de la clase 'ArticulosService':

```
constructor(private articulosServicio: ArticulosService) {}
```

La clase ArticulosService está creada en el archivo 'articulos.service.ts':

```
export class ArticulosService {  
    ...  
}
```

En ningún momento creamos un objeto de ésta clase, sino el framework de Angular se encarga de esto. Debemos registrar la clase 'ArticulosService' en el archivo 'app.module.ts':

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

import { ArticulosService } from './articulos.service';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [ArticulosService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Volviendo al archivo app.component.ts en el método ngOnInit procedemos a llamar al método recuperarTodos:

```
ngOnInit() {
  this.recuperarTodos();
}
```

recuperarTodos tiene por objetivo utilizar el 'servicio' que llega al constructor para llamar al método 'recuperarTodos' del servicio propiamente dicho:

```
recuperarTodos() {  
    this.articulosServicio.recuperarTodos().subscribe(result => this.articulos = result);  
}
```

Si vemos ahora el método 'recuperarTodos' de la clase 'ArticulosService', es el que tiene la responsabilidad de hacer una petición al servidor:

```
recuperarTodos() {  
    return this.http.get(`${this.url}recuperartodos.php`);  
}
```

El método 'recuperarTodos' de la clase 'ArticulosService' retorna un objeto de la clase 'HttpClient'.

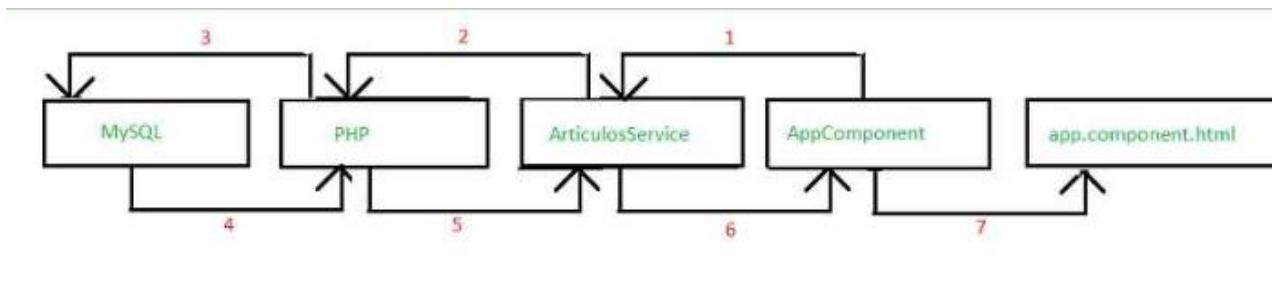
Ahora debemos entender porque podemos llamar al método 'suscribe':

```
recuperarTodos() {  
    this.articulosServicio.recuperarTodos().subscribe(result => this.articulos = result);  
}
```

El método 'suscribe' recibe los resultados y procedemos a asignar a la propiedad 'articulos', con esto, Angular se encarga de actualizar la página con todos los artículos recuperados. El proceso de actualizar la página sucede en el archivo 'app.component.html':

```
<div>
  <h1>Administración de artículos</h1>
  <table border="1" *ngIf="hayRegistros(); else sinarticulos">
    <tr>
      <td>Codigo</td><td>Descripcion</td><td>Precio</td><td>Borrar</td><td>Seleccionar</td>
    </tr>
    <tr *ngFor="let art of articulos">
      <td>{{art.codigo}}</td>
      <td>{{art.descripcion}}</td>
      <td>{{art.precio}}</td>
      <td><button (click)="baja(art.codigo)">Borrar?</button></td>
      <td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>
    </tr>
  </table>
  <ng-template #sinarticulos><p>No hay artículos.</p></ng-template>
  <div>
    <p>
      descripcion:<input type="text" [(ngModel)]="art.descripcion" />
    </p>
    <p>
      precio:<input type="number" [(ngModel)]="art.precio" />
    </p>
    <p><button (click)="alta()">Agregar</button>
      <button (click)="modificacion()">Modificar</button></p>
    </div>
  </div>
```

El flujo de la información lo podemos representar con el siguiente esquema:



Alta

1

La etiqueta button tiene enlazado la llamada al método 'alta':

```
<p><button (click)="alta()">Agregar</button>
```

4

El método 'alta' de la clase 'ArticulosService' hace la llamada al servidor mediante el objeto 'http' de la clase 'HttpClient'. Se utiliza el método 'post' ya que se enviarán datos al servidor:

```
alta(articulo) {
  return this.http.post(`${this.url}alta.php`, JSON.stringify(articulo));
}
```

Veamos ahora los pasos cuando se agrega una fila en la tabla 'articulos'.  
Todo comienza cuando el operador presiona el botón de 'Agregar':



2

El método 'alta' se encuentra codificado en el archivo 'app.component.ts' dentro de la clase 'AppComponent':

```
alta() {
  this.articulosServicio.alta(this.art).subscribe(datos => {
    if (datos['resultado']=='OK') {
      alert(datos['mensaje']);
      this.recuperarTodos();
    }
  });
}
```

3

En este método procedemos a llamar al método 'alta' de la clase 'ArticulosService' y se le pasa como parámetro la propiedad 'art' que almacena la descripción y precio del artículo que el operador acaba de ingresar por teclado.

## Alta

Ahora se ejecuta el programa PHP definido en el archivo 'alta.php' donde procedemos a efectuar el insert en la tabla de MySQL:

También dentro del programa PHP procedemos a retornar en formato JSON que la operación se efectuó en forma correcta.

En el método 'alta' de la clase 'AppComponent' que ya vimos, recibe los datos de la respuesta JSON, mostrando un mensaje si la carga se efectuó correctamente:

```
alta() {
    this.articulosServicio.alta(this.art).subscribe(datos => {
        if (datos['resultado']=='OK') {
            alert(datos['mensaje']);
            this.recuperarTodos();
        }
    });
}
```

La actualización de la página HTML la logramos llamando al método 'recuperarTodos'.

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-with, Content-Type, Accept");

$json = file_get_contents('php://input');

$params = json_decode($json);

require("conexion.php");
$con=retornarConexion();

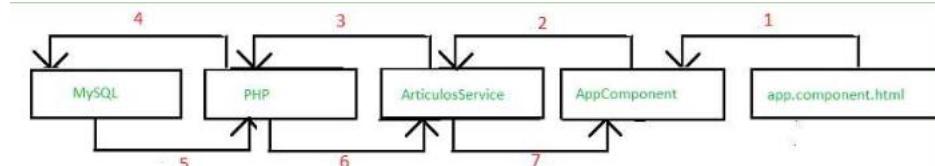
mysqli_query($con,"insert into articulos(descripcion,precio) values
    ('$params->descripcion',$params->precio)");

class Result {}

$response = new Result();
$response->resultado = 'OK';
$response->mensaje = 'datos grabados';

header('Content-Type: application/json');
echo json_encode($response);
?>
```

También llamamos al método 'recuperarTodos' con el objetivo que se actualice la pantalla con los datos actuales de la tabla 'articulos'. El flujo de la información para efectuar el 'alta' de un artículo en la base de datos es::



## Baja

El borrado de un artículo se efectúa cuando el operador presiona el botón con la etiqueta 'Borra?':

Se llama al método 'baja' de la clase 'AppComponent' y se le pasa como parámetro el código de artículo a borrar.

El método baja:

```
baja(codigo) {
    this.articulosServicio.baja(codigo).subscribe(datos => {
        if (datos['resultado']=='OK') {
            alert(datos['mensaje']);
            this.recuperarTodos();
        }
    });
}
```

Ahora en el servidor se ejecuta la aplicación PHP baja.php:

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");
$con=retornarConexion();

mysqli_query($con,"delete from articulos where codigo=$_GET[codigo]");

class Result {}

$response = new Result();
$response->resultado = 'OK';
$response->mensaje = 'articulo borrado';

header('Content-Type: application/json');
echo json_encode($response);
?>
```

```
<td><button (click)="baja(art.codigo)">Borrar</button></td>
```

Llamamos al método 'baja' de la clase 'ArticulosService' y le pasamos como parámetro el código de artículo que queremos borrar.

El método 'baja' de la clase 'ArticulosService' procede a llamar al archivo baja.php y le pasa como parámetro el código de artículo que se debe borrar:

Luego en el método 'baja' de la clase 'AppComponent' podemos mostrar un mensaje si la baja se ejecutó con éxito:

```
baja(codigo) {
    this.articulosServicio.baja(codigo).subscribe(datos => {
        if (datos['resultado']=='OK') {
            alert(datos['mensaje']);
            this.recuperarTodos();
        }
    });
}
```

Como podemos ver para actualizar la pantalla procedemos a llamar al método 'recuperarTodos'.

## Consulta

La consulta o selección se dispara cuando el operador presiona el botón que tiene la etiqueta 'Seleccionar' y tiene por objetivo mostrar en los dos controles 'text' la descripción y precio del artículo:

Al presionar el botón se llama el método 'seleccionar' de la clase 'AppComponent':

Ahora llamamos al método 'seleccionar' de la clase 'ArticulosService':

Recuperamos del servidor llamando a la página 'seleccionar.php' los datos del artículo cuyo código pasamos como parámetro :

En el método 'seleccionar' de la clase AppComponent al ejecutarse el método subscribe almacena en la propiedad 'art' el resultado devuelto por el servidor (con esta asignación se actualizan los dos controles 'input' del HTML):

```
<td><button (click)="seleccionar(art.codigo)">Seleccionar</button></td>
```

```
seleccionar(codigo) {
  this.articulosServicio.seleccionar(codigo).subscribe(result => this.art = result[0]);
}
```

```
seleccionar(codigo:number) {
  return this.http.get(`${this.url}seleccionar.php?codigo=${codigo}`);
}
```

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

require("conexion.php");
$con=retornarConexion();

$registros=mysqli_query($con,"select codigo, descripcion, precio from articulos where codigo=$_GET[codigo]");

if ($reg=mysqli_fetch_array($registros))
{
  $vec[]=$reg;
}

$cad=json_encode($vec);
echo $cad;
header('Content-Type: application/json');

?>
```

```
seleccionar(codigo) {
  this.articulosServicio.seleccionar(codigo).subscribe(result => this.art = result[0]);
}
```

La ejecución del programa en PHP procede a recuperar la fila de la tabla que coincide con el código enviado y lo retorna con formato JSON.

## Modificación

El último algoritmo que implementa nuestra aplicación es la modificación de la descripción y precio de un artículo que seleccionemos primeramente.

Cuando presionamos el botón que tiene la etiqueta 'Modificar' se ejecuta el método 'modificación':

```
<button (click)="modificacion()">Modificar</button></p>
```

El método 'modificación' se implementa en la clase 'AppComponent':

```
modificacion() {
  this.articulosServicio.modificacion(this.art).subscribe(datos => {
    if (datos['resultado']=='OK') {
      alert(datos['mensaje']);
      this.recuperarTodos();
    }
  });
}
```

Lo primero que hacemos en este método es llamar al método 'modificación' de la clase 'ArticulosService' y pasar como dato todos los datos del artículo seleccionado y las posibles modificaciones efectuadas.

El método 'modificacion' de la clase 'ArticulosService':

```
modificacion(articulo) {
  return this.http.post(`${this.url}modificacion.php`, JSON.stringify(articulo));
}
```

Accede al servidor pidiendo el archivo 'modificacion.php' y pasando todos los datos del artículo mediante un 'post'.

## Modificación

El archivo 'modificacion.php' procede a cambiar la descripción y precio del artículo:

```
<?php
header('Access-Control-Allow-Origin: *');
header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept");

$json = file_get_contents('php://input');

$params = json_decode($json);

require("conexion.php");
$con=retornarConexion();

mysqli_query($con,"update articulos set descripcion='".$params->descripcion',
            precio=$params->precio
            where codigo=$params->codigo");

class Result {}

$response = new Result();
$response->resultado = 'OK';
$response->mensaje = 'datos modificados';

header('Content-Type: application/json');
echo json_encode($response);
?>
```

En la clase 'AppComponent' podemos comprobar si el resultado fue 'OK' y actualizar nuevamente la página:

```
modificacion() {
  this.articulosServicio.modificacion(this.art).subscribe(datos => {
    if (datos['resultado']=='OK') {
      alert(datos['mensaje']);
      this.recuperarTodos();
    }
  });
}
```

## Compilación y despliegue de una aplicación Angular en un servidor de Internet

Cuando uno desarrolla una aplicación web utilizando el framework de Angular lo desarrolla y prueba en forma local. Cada cambio que desarrolla lo puede probar en forma local ejecutando el servidor de desarrollo que trae Angular:

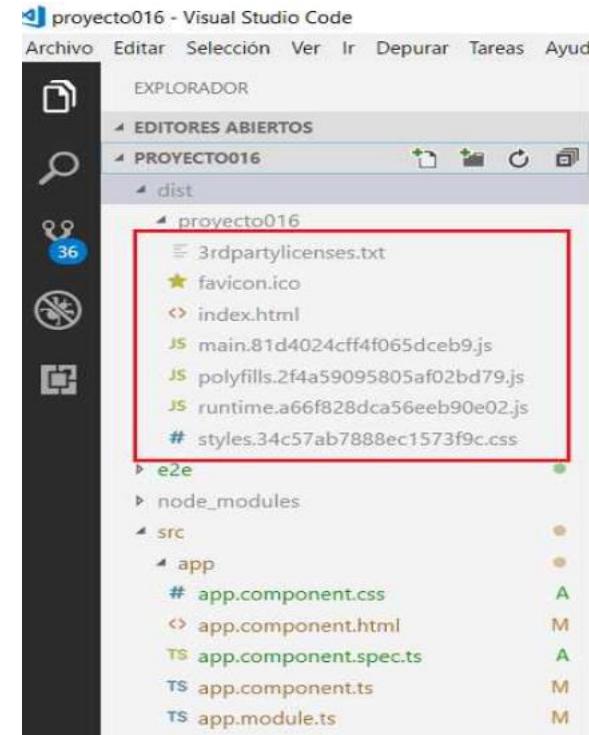
```
f:\angularya\ proyecto016>ng serve -o
```

Una vez finalizada la aplicación debemos subirla a nuestro servidor real en Internet, para esto debemos ejecutar el siguiente comando de Angular CLI:

```
f:\angularya\ proyecto016>ng build --prod
```

## Compilación y despliegue de una aplicación Angular en un servidor de Internet

Luego de este proceso se genera una carpeta llamada 'dist' que contiene todos los archivos que debemos subir a nuestro servidor de Internet:



Los archivos de esta carpeta se deben subir a la carpeta raíz de nuestro servidor de Internet, luego cuando accedemos al dominio de nuestro sitio:

[www.codigodata.com](http://www.codigodata.com)

El servidor responde devolviendo el archivo 'index.html' y este en su interior tiene todas las referencias a archivos \*.js y \*.css

## Compilación y despliegue de una aplicación Angular en un servidor de Internet

Subir una aplicación Angular a una subcarpeta de nuestro servidor.

Si nuestra aplicación Angular no se ejecutará en la raíz de nuestro servidor de Internet, el proceso de compilación es diferente.

Por ejemplo si queremos cargar una aplicación Angular a la carpeta:

<http://www.codigodata.com/angular/>

El proceso de compilación debe ser:

```
f:\angular\ya\proyecto016>ng build --prod --base-href=/angular/
```

Una vez construido el proyecto debemos subir todos los archivos a la carpeta /angular/proyecto016/ y ya podemos probar la aplicación:

## Herramienta Angular CLI

Hemos visto que Angular CLI nos permite entre otras cosas:

Crear una aplicación de Angular (ng new proyecto001)

Ejecutar una aplicación Angular en forma local (ng serve -o)

Crear componentes (ng generate component dado)

Crear módulos (ng generate module elementos)

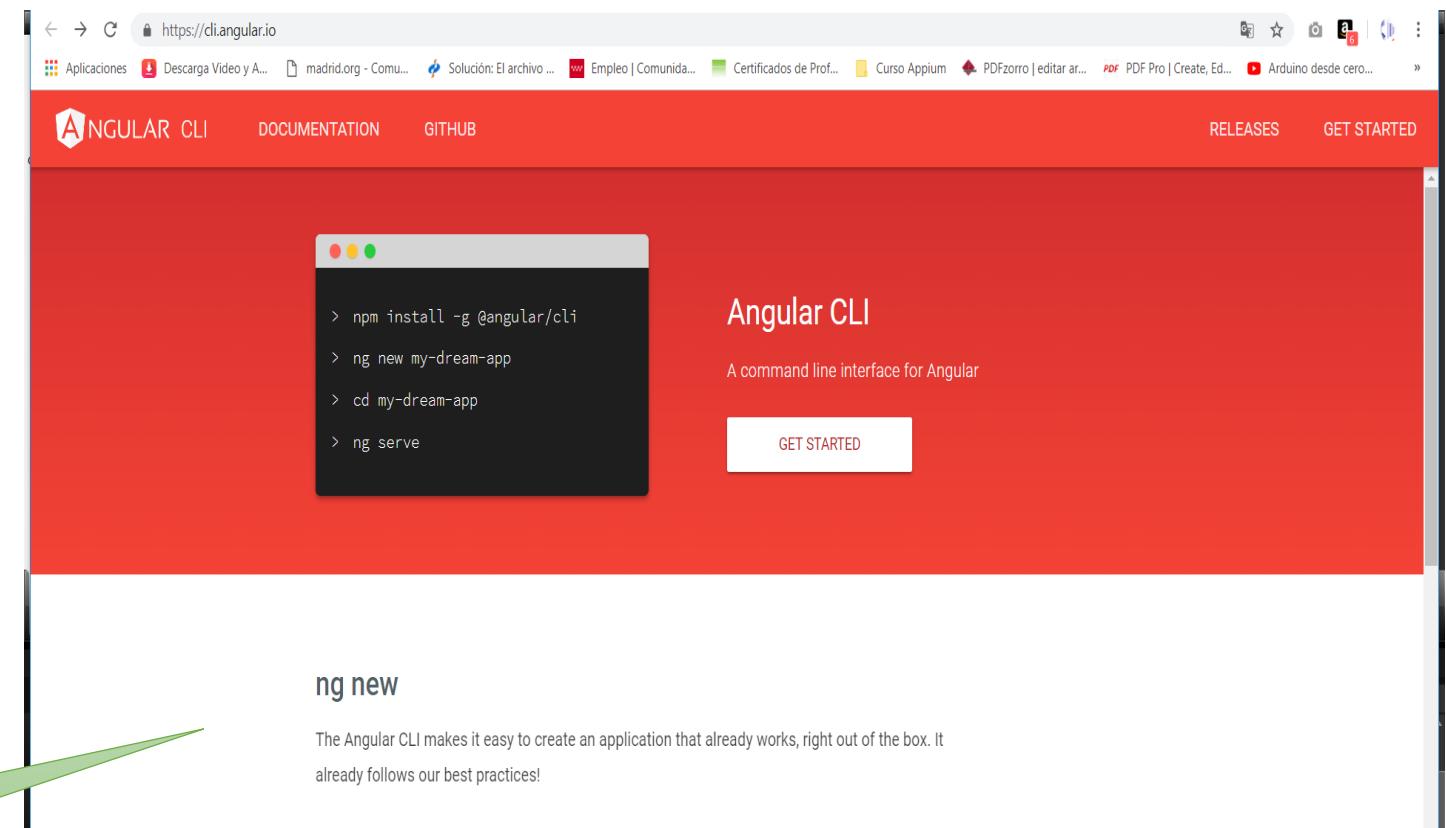
Crear servicios (ng generate service articulos --module=app)

Crear tuberías (ng generate pipe letras)

Desplegar aplicaciones en producción (ng build --prod)

Debe tener siempre a mano la página que contiene la documentación oficial

En las últimas versiones de Angular se le ha dado una responsabilidad fundamental a la herramienta Angular CLI.



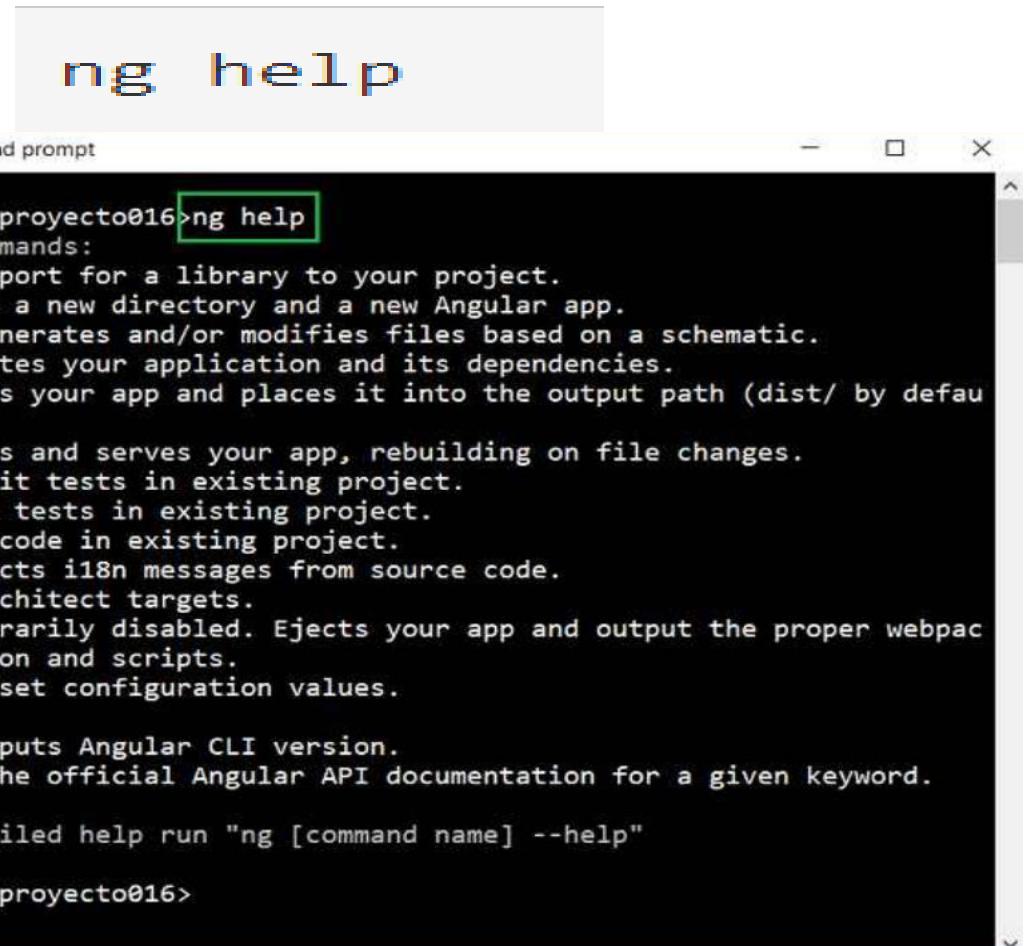
Recordemos que en el primer concepto de este curso lo primero que hicimos luego de instalar Node.js es la instalación de Angular.CLI mediante el comando:

```
npm install -g @angular/cli
```

## Comandos útiles de Angular.CLI

Para conocer todos los comandos disponibles de la herramienta Angular.CLI debemos ejecutar:

Tenemos como resultado muchos de los comandos ya vistos y utilizados en conceptos anteriores:



The screenshot shows a terminal window titled "Node.js command prompt" with the command "ng help" entered. The output lists various Angular CLI commands with their descriptions. Several commands are highlighted with red boxes: "new", "generate", "update", "build", "serve", "test", "e2e", "lint", "i18n", "run", "eject", "config", "help", "version", and "doc". The "help" command is also highlighted.

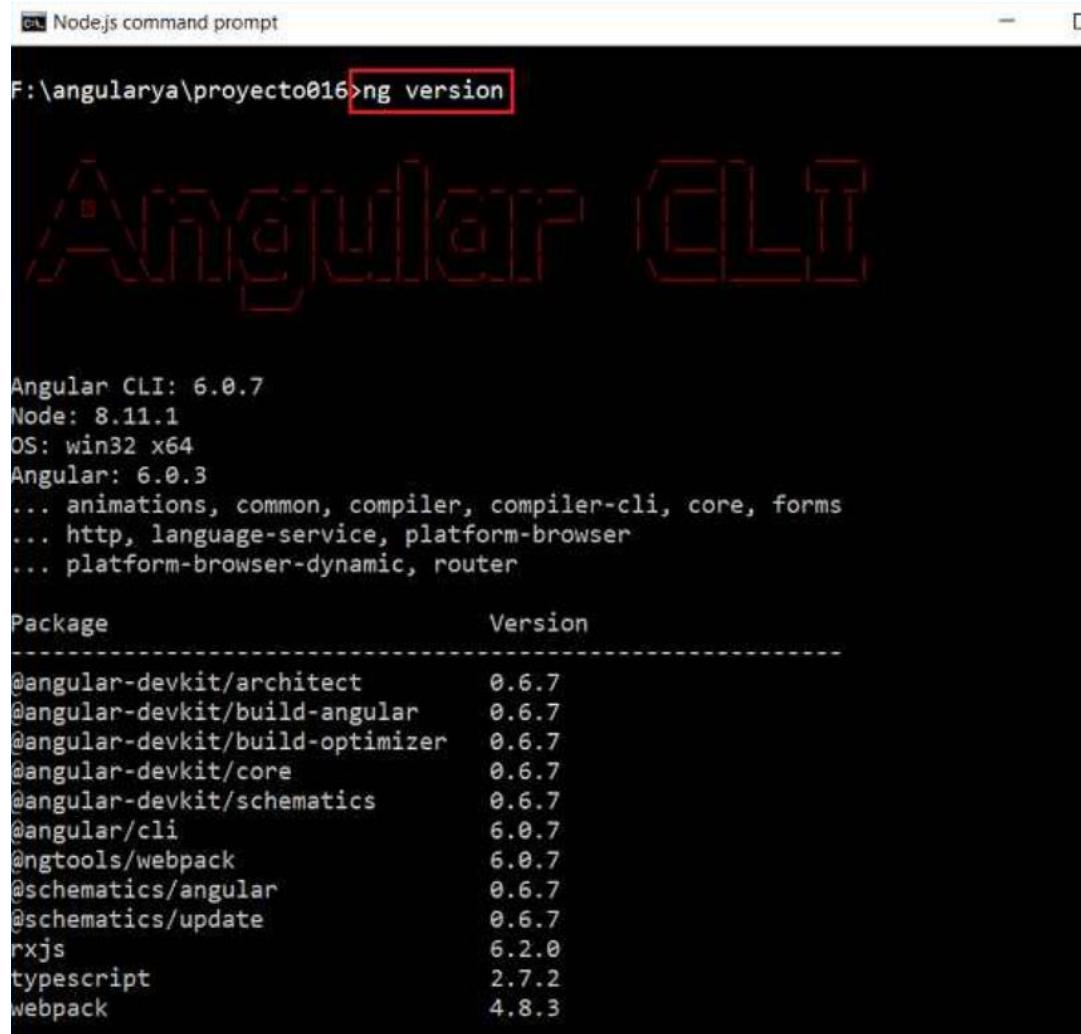
```
ng help
Available Commands:
  add      Add support for a library to your project.
  new      Creates a new directory and a new Angular app.
  generate Generates and/or modifies files based on a schematic.
  update   Updates your application and its dependencies.
  build    Builds your app and places it into the output path (dist/ by default).
  serve    Builds and serves your app, rebuilding on file changes.
  test     Run unit tests in existing project.
  e2e     Run e2e tests in existing project.
  lint     Lints code in existing project.
  i18n    Extracts i18n messages from source code.
  run     Runs Architect targets.
  eject   Temporarily disabled. Ejects your app and output the proper webpack configuration and scripts.
  config  Get/set configuration values.
  help    Help.
  version Outputs Angular CLI version.
  doc     Opens the official Angular API documentation for a given keyword.

For more detailed help run "ng [command name] --help"

F:\angularaya\proyecto016>
```

Si por ejemplo queremos ver la versión de Angular.CLI que tenemos instalada en nuestra computadora debemos ejecutar:

ng version



The screenshot shows a terminal window titled "Nodejs command prompt". The command "F:\angularya\proyecto016>ng version" is entered, with the output displayed below it. The output includes the Angular CLI version (6.0.7), Node version (8.11.1), OS (win32 x64), Angular version (6.0.3), and various dependency versions. A red box highlights the command input.

```
F:\angularya\proyecto016>ng version
Angular CLI: 6.0.7
Node: 8.11.1
OS: win32 x64
Angular: 6.0.3

... animations, common, compiler, compiler-cli, core, forms
... http, language-service, platform-browser
... platform-browser-dynamic, router

Package                           Version
-----@angular-devkit/architect      0.6.7
@angular-devkit/build-angular       0.6.7
@angular-devkit/build-optimizer     0.6.7
@angular-devkit/core                0.6.7
@angular-devkit/schematics          0.6.7
@angular/cli                         6.0.7
@ngtools/webpack                     6.0.7
@schematics/angular                  0.6.7
@schematics/update                   0.6.7
rxjs                                6.2.0
typescript                          2.7.2
webpack                             4.8.3
```

## Herramienta Angular CLI - comando: ng new

Hemos visto que con el comando 'new' de la herramienta Angular CLI se crea un esqueleto de una aplicación Angular:

Si necesitamos administrar rutas en nuestra aplicación deberemos crear la aplicación con la siguiente sintaxis:

```
ng new proyecto017 --routing
```

Se crea el archivo 'app-routing.module.ts' donde debemos configurar las rutas, tema visto anteriormente.

```
@Component({  
  selector: 'ele-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})
```

```
ng new proyecto017
```

El comando new podemos pasar otros parámetros para adaptar la aplicación generada.

Podemos especificar el prefijo a los selectores generados mediante el parámetro --prefix:

```
ng new proyecto017 --prefix ele
```

Luego se crea con prefijo 'ele' en lugar del valor por defecto 'app':  
Si abrimos la componente creada por defecto podemos ver que el prefijo es 'ele':

En lugar de 'app':

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})
```

Luego si creamos una componente dentro del proyecto también se respetará el prefijo definido en la creación del proyecto:

```
@Component({  
  selector: 'ele-dado',  
  templateUrl: './dado.component.html',  
  styleUrls: ['./dado.component.css']  
})
```

La definición de prefijos en las componentes de Angular permiten diferenciar las etiquetas nativas del navegador (ej. 'button') con las etiquetas propias de Angular (ej. ele-button)

En el archivo 'angular.json' se encuentra almacenado el prefijo definido al crearse el proyecto:

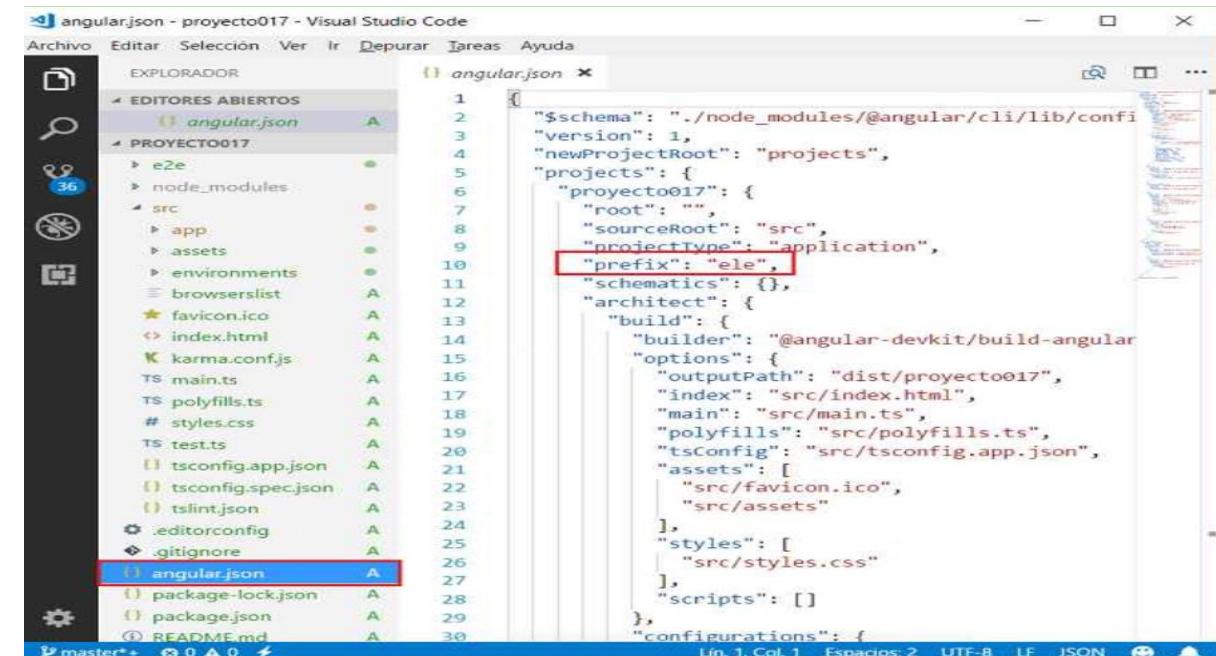
Si queremos escribir menos al crear el proyecto podemos utilizar el alias -p en lugar de --prefix:

```
ng new proyecto017 -p ele
```

ng generate component dado

El selector de la componente dado queda con el siguiente nombre:

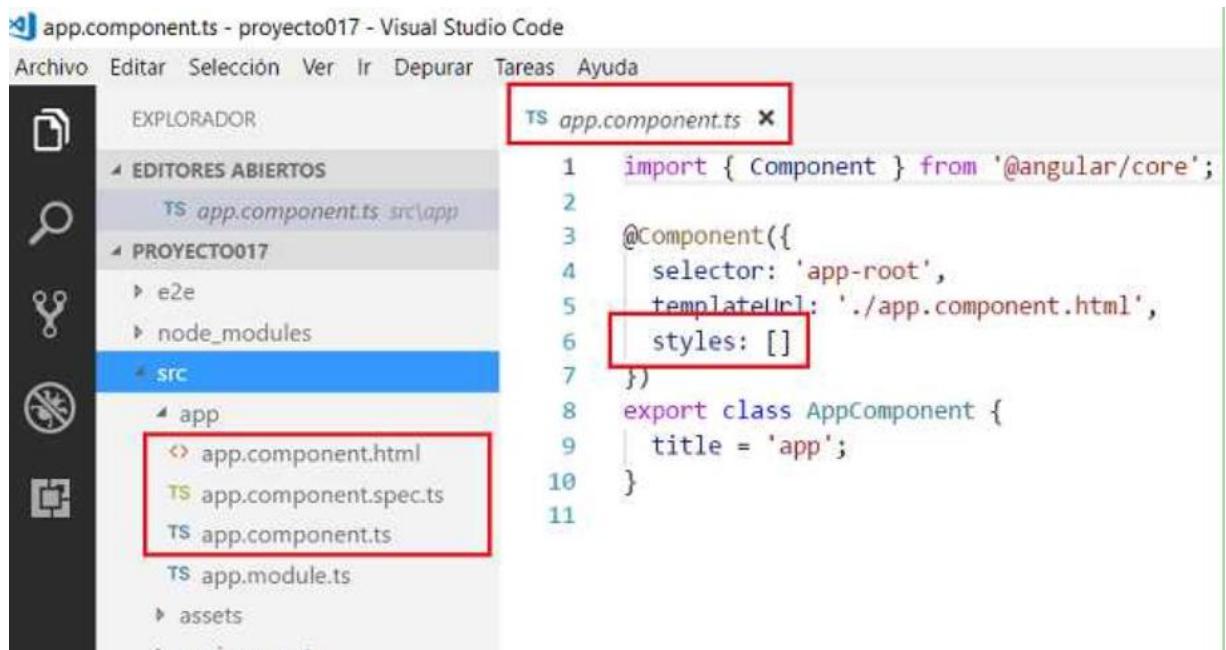
Al crearse un proyecto uno de los archivos fundamentales que almacenan datos de configuración del mismo es 'angular.json' (se encuentra en la carpeta raíz del proyecto)



Mediante el parámetro `--inline-style` (alias: `-s`) podemos evitar que se cree el archivo `*.css` y la definición de los estilos se deba hacer directamente en el archivo `*.ts`:

Con esto no se crearán los archivos `*.css` para las componentes. El archivo `*.ts` de la componente tienen un lugar donde definir los estilos:

```
ng new proyecto017 --inline-style
```



The screenshot shows the Visual Studio Code interface. On the left is the Explorer sidebar displaying the project structure:

- EDITORES ABIERTOS: app.component.ts
- PROYECTO017:
  - e2e
  - node\_modules
  - src
    - app
      - app.component.html
      - app.component.spec.ts
      - app.component.ts
    - assets

The code editor on the right shows the content of `app.component.ts`:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styles: []
7 })
8 export class AppComponent {
9   title = 'app';
10 }
```

Lines 6 and 7 are highlighted with red boxes.

Puede tener sentido si la componente es bastante sencilla y no requiere la definición de una hoja de estilo compleja. Probar modificar los archivos \*.ts y \*.html de la componente del proyecto.

app.component.ts

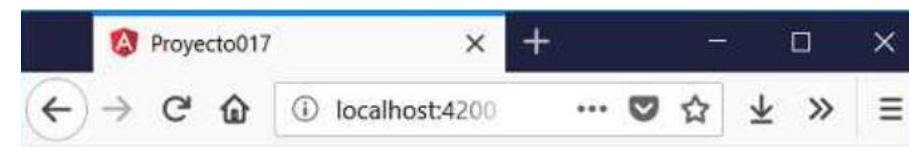
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styles: ['.titulo { color:red; font-size:2rem }',
            '.parrafo {color:black; font-size:1.1rem }']
})
export class AppComponent {
  tit = 'Prueba de inline-style';
}
```

app.component.html

```
<div style="text-align:center">
  <h1 class="titulo">{{tit}}</h1>
  <p class="parrafo">Esto es un párrafo.</p>
</div>
```

Si ejecutamos el proyecto podemos ver que los estilos se están recuperando del archivo \*.ts:



**Prueba de inline-style**

Esto es un párrafo.

Mediante el parámetro `--inline-template` (alias: `-s`) podemos evitar que se cree el archivo `*.html` y la definición del HTML se hace directamente en el archivo `*.ts` (borre primero el proyecto017 antes de crearlo nuevamente con esta nueva configuración):

Luego de creado el proyecto solo se han creado los archivos `*.css` y `*.ts`:

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows the project structure: PROYECTO017, src, app, app.module.ts, app.component.css, app.component.spec.ts, and app.component.ts.
- Editor:** The active file is `app.component.ts`. The code contains an `@Component` decorator with a template section. The template section includes an `<div>` element with `text-align:center` style, an `<h1>` element with `Welcome to {{title}}!`, an `<img>` element with a width of 300 pixels and a data URI source, and an `<ul>` element with three `<li>` items, each containing an `<h2>` element with a link to a blank page.

`ng new proyecto017 --inline-template`

Con este parámetro estamos indicando que no se cree el archivo `*.html` para la componente y la definición del HTML se haga en el mismo archivo `*.js`

Nuevamente decimos que esto tiene sentido si la complejidad de la componente Angular no es grande.

Podemos inclusive evitar que se creen tanto el archivo `*.css` y `*.html` indicando ambos parámetros al crear la componente:

`ng new proyecto017 --inline-style --inline-template`

Si tratamos de crear un proyecto y ya existe uno en la carpeta actual se produce un error al ejecutar:

`ng new proyecto017`

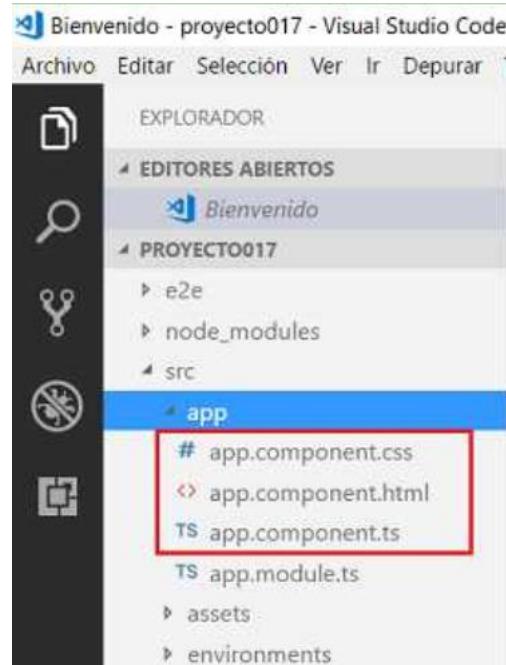
`ng new proyecto017 --force`

Si queremos forzar la creación del proyecto y sobreescribir los archivos actuales debemos añadir el parámetro `--force` (alias `-f`):

Si por algún motivo no queremos que Angular.CLI nos genere el archivo de test debemos pasar el parámetro --skip-tests (alias -s):

```
ng new proyecto017 --skip-tests
```

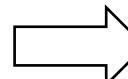
Como podemos comprobar no se ha generado el archivo 'app.component.spec.ts'.



Otro parámetro más que podemos utilizar cuando vamos a crear un proyecto es --dry-run (alias: -d)

Mediante esta opción Angular CLI nos informa que archivos se crearán (no los crea) y a partir de estos datos poder tomar la decisión de crear o no el proyecto:

```
ng new proyecto017 --dry-run
```



```
F:\angular\ya>ng new proyecto017 --dry-run
CREATE proyecto017/angular.json (3440 bytes)
CREATE proyecto017/package.json (1315 bytes)
CREATE proyecto017/README.md (1028 bytes)
CREATE proyecto017/tsconfig.json (384 bytes)
CREATE proyecto017/tslint.json (2805 bytes)
CREATE proyecto017/.editorconfig (245 bytes)
CREATE proyecto017/.gitignore (503 bytes)
CREATE proyecto017/src/environments/environment.prod.ts (51 bytes)
CREATE proyecto017/src/environments/environment.ts (631 bytes)
CREATE proyecto017/src/favicon.ico (5430 bytes)
CREATE proyecto017/src/index.html (298 bytes)
CREATE proyecto017/src/main.ts (370 bytes)
CREATE proyecto017/src/polyfills.ts (3194 bytes)
CREATE proyecto017/src/test.ts (642 bytes)
CREATE proyecto017/src/assets/.gitkeep (0 bytes)
CREATE proyecto017/src/styles.css (80 bytes)
CREATE proyecto017/src/browserslist (375 bytes)
CREATE proyecto017/src/karma.conf.js (964 bytes)
CREATE proyecto017/src/tsconfig.app.json (194 bytes)
CREATE proyecto017/src/tsconfig.spec.json (282 bytes)
CREATE proyecto017/src/tslint.json (314 bytes)
CREATE proyecto017/src/app/app.module.ts (314 bytes)
CREATE proyecto017/src/app/app.component.html (1141 bytes)
CREATE proyecto017/src/app/app.component.spec.ts (986 bytes)
CREATE proyecto017/src/app/app.component.ts (207 bytes)
CREATE proyecto017/src/app/app.component.css (0 bytes)
CREATE proyecto017/e2e/protractor.conf.js (752 bytes)
CREATE proyecto017/e2e/src/app.e2e-spec.ts (299 bytes)
CREATE proyecto017/e2e/src/app.po.ts (208 bytes)
CREATE proyecto017/e2e/tsconfig.e2e.json (213 bytes)

NOTE: Run with "dry run" no changes were made.
```

ng generate component

Se crea una componente 'dato' con los archivos respectivos y la modificación del archivo del módulo:  
Se informa en la consola los archivos creados y modificados:

ng generate component dato

```
Node.js command prompt
F:\angularya\proyecto017>ng generate component dato
CREATE src/app/dado/dado.component.html (23 bytes)
CREATE src/app/dado/dado.component.spec.ts (614 bytes)
CREATE src/app/dado/dado.component.ts (261 bytes)
CREATE src/app/dado/dado.component.css (0 bytes)
UPDATE src/app/app.module.ts (388 bytes)
```

Podemos pasar varias opciones cuando creamos la componente:

ng generate component dato --inline-style

Evita que se cree el archivo dado.component.css

Lo mismo podemos evitar que se cree el archivo HTML: Lo mismo podemos evitar que se cree el archivo HTML:

ng generate component dato --inline-template

Tenemos como resultado la no creación del archivo HTML:

```
:\\angularya\\proyecto017>ng generate component dato --inline-template
CREATE src/app/dado/dado.component.spec.ts (614 bytes)
CREATE src/app/dado/dado.component.ts (275 bytes)
CREATE src/app/dado/dado.component.css (0 bytes)
UPDATE src/app/app.module.ts (388 bytes)
```

Podemos definir el prefijo para la componente mediante la opción **--prefix** (alias: **-p**):

Luego la componente que se crea tiene dicho prefijo:

Si queremos definir el nombre completo para el selector tenemos la opción **--Selector**:

Luego la componente se crea con dicho nombre de selector:

**ng generate component dado --prefix juego**

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

@Component({
  selector: 'juego-dado',
  templateUrl: './dado.component.html',
  styleUrls: ['./dado.component.css']
})
export class DadoComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}
```

**ng generate component dado --selector ju-dado**

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

@Component({
  selector: 'ju-dado',
  templateUrl: './dado.component.html',
  styleUrls: ['./dado.component.css']
})
export class DadoComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }
}
```

Para almacenar la componente en un determinado módulo debemos utilizar la opción --module (alias: -m)

```
ng generate component elementos/dado --module elementos
```

Considerando que ya hemos creado un módulo llamado elementos (ng generate module elementos) estamos creando la componente dado en el modulo elementos.

Para evitar que se cree el archivo 'dado.spec.js' debemos insertar el comando --spec con el valor false:

```
ng generate component dado --spec false
```

Luego no se genera el archivo 'dado.spec.js':

```
F:\angularya\proyecto017>ng generate component dado --spec false
CREATE src/app/dado/dado.component.html (23 bytes)
CREATE src/app/dado/dado.component.ts (261 bytes)
CREATE src/app/dado/dado.component.css (0 bytes)
```

Tengamos en cuenta que todas estas opciones se pueden combinar y ejecutar en forma simultanea, por ejemplo si queremos generar solo el archivo \*.ts de la componente y que no genere el archivo spec, \*.css y \*.html:

```
ng generate component dado --spec false --inline-template --inline-style
```

```
F:\angularya\proyecto017>ng generate component dado --spec false --inline-template --inline-style
CREATE src/app/dado/dado.component.ts (250 bytes)
```

```
F:\angularya\proyecto017>
```

Otras opciones posibles cuando creamos una componente son:

--force (alias: -f) Forzar la sobreescritura de los archivos existentes (se borra la componente anterior que tiene el mismo nombre)

Angular CLI permite ingresar comandos en formato resumido utilizando el primer carácter:

ng g c dado

En lugar de escribir:

ng generate component dado

**ng generate service**

Mediante el comando:

Se crea una clase ArticulosService y se inyecta a nivel de 'root':

**ng generate service articulos**

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

@Injectable({
  providedIn: 'root'
})
export class ArticulosService {

  constructor() { }
}
```

Disponemos de los siguientes opciones en este comando:

--dry-run

--force

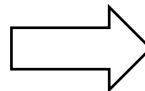
--spec

**ng generate module**

Mediante el comando:

```
ng generate module administracion
```

Se crea la carpeta 'administracion' y dentro de ella el archivo 'administracion.module.ts'.



```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class AdministracionModule { }
```

Disponemos de los siguientes opciones en este comando:

--dry-run

--force

--spec

--routing

**ng generate pipe**

Mediante el comando:

**ng generate pipe letras**

Se genera el archivo 'letras.pipe.ts':

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'letras'
})
export class LetrasPipe implements PipeTransform {

  transform(value: any, args?: any): any {
    return null;
  }
}
```

Disponemos de los siguientes  
opciones en este comando:

--dry-run

--force

--spec

--module

ng generate class

Mediante el comando:

Se crea el archivo 'articulo.ts':

ng generate class articulo

```
export class Articulo {  
}
```

**ng generate interface**

Mediante el comando:

Se crea el archivo 'venta.ts':

**ng generate interface venta**

```
export interface Venta {  
}
```

`ng generate enum`

Mediante el comando:

Se crea el archivo 'operaciones.enum.ts':

`ng generate enum operaciones`

```
export enum Operaciones {  
}
```

## Herramienta Angular CLI - comando: ng serve

Recordemos que debemos ejecutar el comando serve en la carpeta donde se haya nuestra aplicación Angular. Al disponer la opción -o se abre automáticamente el navegador web. La sintaxis larga pero que produce la misma acción es:

ng serve --open

Para conocer todas las opciones disponibles en un comando de Angular CLI debemos acceder a la opción --help:

ng serve -o

Otro comando que hemos utilizado en cada uno de los proyectos que hemos implementado es 'serve' (desde el principio hemos utilizado la opción -o):

### Node.js command prompt

```
:\\angularya\\proyecto017>ng serve --help
Usage:  ng serve <project> [options]

Options:
  --aot
    Build using Ahead of Time compilation.
  --base-href
    Base url for the application being built.
  --browser-target
    Target to serve.
  --common-chunk
    Use a separate bundle containing code used across multiple bundle
  --configuration (-c)
    Specify the configuration to use.
  --deploy-url
    URL where files will be deployed.
  --disable-host-check
    Don't verify connected clients are part of allowed hosts.
  --eval-source-map
    Output in-file eval sourcemaps.
  --hmr
    Enable hot module replacement.
  --hmr-warning
    Show a warning when the --hmr option is enabled.
  --host
    Host to listen on.
  --live-reload
    Whether to reload the page on change, using live-reload.
  --open (-o)
    Opens the url in default browser.
  --optimization
    Defines the optimization level of the build.
  --poll
    Enable and define the file watching poll time period in milliseco
    nds.
  --port
    Port to listen on.
  --prod
    Flag to set configuration to "prod".
  --progress
    Log progress to the console while building.
  --proxy-config
    Proxy configuration file.
  --public-host
    Specify the URL that the browser client will use.
  --serve-path
    The path name where the app will be served.
  --serve-path-default-warning
    Show a warning when deploy-url/base-href use unsupported serve pa
    th values.
  --source-map
    Output sourcemaps.
  --ssl
    Serve using HTTPS.
  --ssl-cert
    SSL certificate to use for serving HTTPS.
  --ssl-key
    SSL key to use for serving HTTPS.
  --vendor-chunk
    Use a separate bundle containing only vendor libraries.
  --watch
    Rebuild on change.
```

## Opción --port

Por defecto el servidor web local que crea Angular CLI se ejecuta en el puerto 4200, si necesitamos que el servidor web se ejecute en otro puerto podemos indicarlo con la opción --port en el momento de iniciarla:

```
ng serve -o --port 4444
```

The screenshot shows a Visual Studio Code interface with the following details:

- Explorador (Explorer):** Shows the project structure: PROYECTO017, e2e, node\_modules, src, app, assets, and files like app.component.css, app.component.html, app.component.ts, app.module.ts.
- Editor:** Displays the content of app.component.html, which includes a placeholder message and an Angular logo.
- Terminal:** Shows the command "F:\angularya\proyecto017>ng serve -o --port 4444" and the response: "Angular Live Development Server is listening on localhost:4444, open your browser on http://localhost:4444/".
- Browser:** Shows the URL "localhost:4444" in the address bar, with the page content "Welcome to app!" and the Angular logo.

Welcome to app!



## Opción --watch (alias: -w)

Por defecto cada vez que modificamos nuestro proyecto y grabamos los cambios el resultado se actualiza en el navegador en forma automática. En algunas situaciones si queremos que no se actualice debemos utilizar la opción 'watch' pasando el valor false:

```
ng serve -o --watch false
```

## Opción --prod

Ejecuta la aplicación con todas las optimizaciones que se hacen cuando se genera el código de producción.

```
ng serve -o --prod
```

## Herramienta Angular CLI - comandos: ng update - ng doc

Si ejecutamos este comando en un proyecto desarrollado con Angular 5 tenemos como resultado:

```
F:\angularaya\proyecto003>ng update --dry-run
Your global Angular CLI version (6.0.0) is greater than your local version (1.7.3). The local Angular CLI version is used.

To disable this warning use "ng config -g cli.warnings.versionMismatch false".
update package.json (1283 bytes)

NOTE: Run with "dry run" no changes were made.

F:\angularaya\proyecto003>
```

Si ejecutamos el comando 'update' sobre un proyecto que coincide con la versión actual:

### ng update

ng update --dry-run

Nos informa que hay que hacer modificaciones en el archivo 'package.json'

Para efectuar los cambios debemos efectivamente ejecutar el comando 'update' sin la opción 'dry-run':

El proyecto de Angular tiene un camino de desarrollo con actualizaciones semestrales. En junio de 2018 la última versión estable es la 6. Angular CLI provee un comando que nos permite actualizar nuestro proyecto a la última versión estable de Angular. Debemos ejecutar el comando 'update' en la carpeta donde se encuentra el proyecto a migrar. Podemos ejecutar este comando primero con la opción 'dry-run' para tener una idea de los cambios que se producirán y si nos conviene en nuestro caso:

```
F:\angularaya\proyecto003>ng update
Your global Angular CLI version (6.0.0) is greater than your local version (1.7.3). The local Angular CLI version is used.

To disable this warning use "ng config -g cli.warnings.versionMismatch false".
update package.json (1283 bytes)

> @angular/cli@6.0.8 postinstall F:\angularaya\proyecto003\node_modules\@angular\cli
> node ./bin/ng-update-message.js

Angular CLI configuration format has been changed, and your existing configuration can be updated automatically by running the following command:
  ng update @angular/cli

npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.3: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

added 26 packages, removed 474 packages, updated 35 packages and moved 29 packages in 144.781s
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.3: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

up to date in 11.223s
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.1.3 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.1.3: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

up to date in 11.057s
```

```
F:\angularaya\proyecto003>ng update
We analyzed your package.json and everything seems to be in order. Good work!
```

ng doc

Este comando abre la documentación oficial de Angular API para una palabra clave que le pasemos:

ng doc ngFor

Luego de esto se abre el navegador que tengamos configurado por defecto y nos muestra:

El comando 'doc' tiene una opción --search (alias: -s), esta nos permite hacer una búsqueda en todo el sitio 'angular.io':

ng doc ngFor -s

Tenemos como resultado:

app.component.html - proyecto017 - Visual Studio Code

Archivo Editar Selección Ver Ir Depurar Tareas Ayuda

EXPLORADOR

EDITORES ABIERTOS app.component.html ... A

PROYECTO017

- e2e
- node\_modules
- src
  - app
    - # app.component.css A
    - app.component.html A
    - TS app.component.spe... A
    - TS app.component.ts A
    - TS app.module.ts A
  - assets

TERMINAL ... 1: cmd + - ☐ □ ^

F:\angularya\proyecto017>ng doc ngFor

F:\angularya\proyecto017>

master+ 0 ▲ 0

Angular - API List https://angular.io/api?query=ngFor

API List

TYPE: All STATUS: All

Search

ngfor

common

app.component.html - proyecto017 - Visual Studio Code

Archivo Editar Selección Ver Ir Depurar Tareas Ayuda

EXPLORADOR

EDITORES ABIERTOS app.component.html ... A

PROYECTO017

- e2e
- node\_modules
- src
  - app
    - # app.component.css A
    - app.component.html A
    - TS app.component.spe... A
    - TS app.component.ts A
    - TS app.module.ts A
  - assets

TERMINAL ... 1: cmd + - ☐ □ ^

F:\angularya\proyecto017>ng doc ngFor

F:\angularya\proyecto017>ng doc ngFor -s

F:\angularya\proyecto017>

master+ 0 ▲ 0

site:angular.io ngFor - Buscar https://www.google.com/search

Google site:angular.io ngFor

Todo Imágenes Vídeos Noticias Maps Más Preferencias

Cerca de 257 resultados (0,17 segundos)

Angular - Displaying Data

<https://angular.io/guide/displaying-data> Traducir esta página

Now use the Angular `ngFor` directive in the template to display each item in the ... In displaying an array, but `ngFor` can repeat items for any ...

## Angular Material

Angular material es un conjunto de componentes visuales que nos permiten desarrollar interfaces de usuario consistentes.

Este conjunto de componentes los desarrolla el mismo equipo de Angular y podemos visitar sus avances en el sitio [material.angular.io/](https://material.angular.io/)

Disponemos de un conjunto de componentes agrupados por categorías:

Layout: card,  
tree, tab etc.

Form Controls: input,  
check box, radio  
button, select etc.

Navigation: menu,  
sidenav, toolbar

Data table: table,  
paginator, sort header

Popups y Modals:  
dialog, snackbar,  
tooltip etc.

Buttons e Indicators:  
button, icon, button  
toggle etc.

## Creación del esqueleto básico de un proyecto con Angular Material.

Crearemos una aplicación que disponga un menú lateral a la izquierda con los nombres de tres países. Mediante el concepto que vimos de Router mostraremos en distintas componentes las informaciones de dichos países.

Crearemos primero el proyecto (es importante la opción 'routing' ya que la aplicación la requiere)

```
ng new proyecto018 --routing
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

```
ng add @angular/material
```

The screenshot shows the Visual Studio Code interface. On the left is the Explorer sidebar with the project structure: PROYECTO018, node\_modules, and src. Under src, there are app, assets, environments, browserslist, favicon.ico, index.html, karma.conf.js, main.ts, polyfills.ts, styles.css, test.ts, and tsconfig.app.json. The app folder contains app.component.css, app.component.html, app.component.spec.ts, and app.module.ts. The terminal at the bottom shows the command `ng add @angular/material` being run, with the output "Installing packages for tooling via npm." and a progress bar.

Se crea una componente llamada 'BarraLateralComponent' con los 4 archivos correspondientes y la modificación del archivo 'app.module.ts':

```
F:\angularya\proyecto018>ng generate @angular/material:material-nav --name barraLateral
CREATE src/app/barra-lateral/barra-lateral.component.html (954 bytes)
CREATE src/app/barra-lateral/barra-lateral.component.spec.ts (748 bytes)
CREATE src/app/barra-lateral/barra-lateral.component.ts (608 bytes)
CREATE src/app/barra-lateral/barra-lateral.component.css (129 bytes)
UPDATE src/app/app.module.ts (902 bytes)
```

Luego de esto ya tenemos en la carpeta node\_modules/angular/material todas las componentes de Angular Material para ser utilizadas en nuestro proyecto.

Crearemos la barra lateral que dispondrá los enlaces:

```
ng generate @angular/material:material-nav --name barraLateral
```

Modificamos ahora el archivo 'app.component.html' donde creamos una etiqueta de la componente que acabamos de crear en el paso anterior:

```
<app-barra-lateral></app-barra-lateral>
```

Si en este momento ejecutamos la aplicación ya podemos ver la barra de navegación lateral:

```
ng serve -o
```



Hasta ahora solo hemos empleado Angular CLI para llegar a este lugar, es decir crear el proyecto y la componente del menú lateral.

Creamos las tres componentes pais1, pais2 y pais3 que tienen por objetivo mostrar datos de paises:

```
ng generate component pais1  
ng generate component pais2  
ng generate component pais3
```

Podemos abrir el archivo 'app.module.ts' y ver que se han importado las tres componentes que acabamos de crear:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { BarraLateralComponent } from './barra-lateral/barra-lateral.component';
import { LayoutModule } from '@angular/cdk/layout';
import { MatToolbarModule, MatButtonModule, MatSidenavModule, MatIconModule, MatListModule } from '@angular/material';
import { Pais1Component } from './pais1/pais1.component';
import { Pais2Component } from './pais2/pais2.component';
import { Pais3Component } from './pais3/pais3.component';

@NgModule({
  declarations: [
    AppComponent,
    BarraLateralComponent,
    Pais1Component,
    Pais2Component,
    Pais3Component
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    LayoutModule,
    MatToolbarModule,
    MatButtonModule,
    MatSidenavModule,
    MatIconModule,
    MatListModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Debemos ahora modificar el archivo 'app-routing.module.ts' con las tres rutas:

Mediante la etiqueta 'router-outlet' indicamos el lugar que debe mostrar la componente especificada por la ruta configurada en el archivo 'app-routing.module.ts', para esto abrimos el archivo 'barra-lateral.component.html' y agregaremos la etiqueta 'router-outlet':

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { Pais1Component } from './pais1/pais1.component';
import { Pais2Component } from './pais2/pais2.component';
import { Pais3Component } from './pais3/pais3.component';

const routes: Routes = [
  {
    path:'pais1',
    component:Pais1Component
  },
  {
    path:'pais2',
    component:Pais2Component
  },
  {
    path:'pais3',
    component:Pais3Component
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

barra-lateral.component.html

```
<mat-sidenav-container class="sidenav-container">
  <mat-sidenav
    #drawer
    class="sidenav"
    fixedInViewport="true"
    [attr.role]="(isHandset$ | async) ? 'dialog' : 'navigation'"
    [mode]="(isHandset$ | async) ? 'over' : 'side'"
    [opened]!"(isHandset$ | async)">
    <mat-toolbar color="primary">Menu</mat-toolbar>
    <mat-nav-list>
      <a mat-list-item routerLink="/pais1">Argentina</a>
      <a mat-list-item routerLink="/pais2">Chile</a>
      <a mat-list-item routerLink="/pais3">Uruguay</a>
    </mat-nav-list>
  </mat-sidenav>
  <mat-sidenav-content>
    <mat-toolbar color="primary">
      <button
        type="button"
        aria-label="Toggle sidenav"
        mat-icon-button
        (click)="drawer.toggle()"
        *ngIf="isHandset$ | async">
        <mat-icon aria-label="Side nav toggle icon">menu</mat-icon>
      </button>
      <span>Datos generales de paises</span>
    </mat-toolbar>
    <!-- Add Content Here -->
    <router-outlet></router-outlet>
  </mat-sidenav-content>
</mat-sidenav-container>
```

Hemos modificado del código generado los hipervínculos a las distintas rutas:

Y también es fundamental agregar la etiqueta 'router-outlet' para indicar donde se muestran las componentes.

Si ejecutamos ahora la aplicación los tres hipervínculos activan la componente respectiva donde se mostrará información referente a cada país:

```
<a mat-list-item routerLink="/pais1">Argentina</a>
<a mat-list-item routerLink="/pais2">Chile</a>
<a mat-list-item routerLink="/pais3">Uruguay</a>
```



Solo nos queda componer los contenidos de las componentes  
'Pais1Component', 'Pais2Component' y 'Pais3Component'.

pais1.component.html

```
<div class="datos">
  <p><strong>Nombre del país:</strong>Argentina</p>
  <p>
    
  </p>
  <h3>Datos generales.</h3>
  <p>Argentina, llamada oficialmente República Argentina,n 12 es un país soberano de América ubicado en el extremo sur y sudeste de dicho subcontinente. Adopta la forma de gobernanza democrática, representativa y federal.</p>
  <p>La Argentina está organizada como un Estado federal descentralizado, integrado desde el Estado nacional y 24 estados autogobernados,11?12? siendo estos sus 23 provincias sumadas a la Ciudad Autónoma de Buenos Aires designada como Capital Federal del país. Cada estado tiene autonomía política, constitución, bandera y cuerpo de seguridad propios. Las 23 provincias mantienen todos los poderes no delegados al Estado nacional y garantizan la autonomía de sus municipios.13?14?</p>
</div>
```

pais2.component.html

```
<div class="datos">
  <p><strong>Nombre del país:</strong>Chile</p>
  <p>
    
  </p>
  <h3>Datos generales.</h3>
  <p>Chile es un país de América ubicado en el extremo sudoeste de América del Sur. Su nombre oficial es República de Chile26? y su capital es la ciudad de Santiago.27 Primer país sudamericano en ingresar a la Organización para la Cooperación y el Desarrollo Económicos, Chile es una de las economías de América Latina que más ha crecido desde mediados de la década de 1980.</p>
  <p>Antes del descubrimiento de América, las tierras situadas al sur del desierto de Atacama se llamaban Chili en la tradición indígena.46? Una vez instalados en Nueva Castilla, Nueva Toledo, los conquistadores españoles siguieron llamando de esa forma a la región del sur, a veces también conocida como «valle de Chile», nombre que se extiende posteriormente a todo el actual país.47?</p>
</div>
```

pais3.component.html

```
<div class="datos">
  <p><strong>Nombre del país:</strong>Uruguay</p>
  <p>
    
  </p>
  <h3>Datos generales.</h3>
  <p>Uruguay, oficialmente República Oriental del Uruguay, es un país de América del Sur situado en la parte oriental del Cono Sur americano. Limita al noreste con Brasil –estado de Río Grande del Sur–, al oeste con Argentina –provincias de Entre Ríos y Corrientes– y tiene costas en el océano Atlántico al sureste y sobre el Río de la Plata hacia el este. Abarca 176.215 km2 y es el segundo país más pequeño de Sudamérica, después de Surinam. Según los datos del último censo del INE en 2011, la población de Uruguay es de 3.290.454 habitantes, por lo que figura en la décima posición entre los países sudamericanos. Oficialmente República Oriental del Uruguay, es un país de América del Sur, situado en la parte oriental del Cono Sur americano. Limita al noreste con Brasil –estado de Río Grande del Sur–, al oeste con Argentina –provincias de Entre Ríos y Corrientes– y tiene costas en el océano Atlántico al sureste y sobre el Río de la Plata hacia el este. Abarca 176.215 km2 y es el segundo país más pequeño de Sudamérica, después de Surinam. Según los datos del último censo del INE en 2011, la población de Uruguay es de 3.290.454 habitantes, por lo que figura en la décima posición entre los países sudamericanos.</p>
</div>
```

Lo nuevo es que hemos agregado las tres imágenes de las banderas en una carpeta llamada `imagenes` que se debe crear dentro de la carpeta '`assets`':

```

```

Si ejecutamos nuevamente la aplicación tenemos como resultado:



## Anotaciones

Agregamos una nueva ruta para el path con un string vacío indicando en la propiedad component cual es la que se debe mostrar:

```
const routes: Routes = [
  {
    path:'',
    component:Pais2Component
  },
]
```

Como podemos ver inmediatamente se inicia la aplicación no aparece la información de ningún país. Si necesitamos que se muestre el país 'Chile' por defecto debemos modificar el archivo 'app-routing.module.ts' con una nueva ruta:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { Pais1Component } from './pais1/pais1.component';
import { Pais2Component } from './pais2/pais2.component';
import { Pais3Component } from './pais3/pais3.component';

const routes: Routes = [
  {
    path:'',
    component:Pais2Component
  },
  {
    path:'pais1',
    component:Pais1Component
  },
  {
    path:'pais2',
    component:Pais2Component
  },
  {
    path:'pais3',
    component:Pais3Component
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

## 36 - Angular Material: botones

Es muy importante cuando trabajamos con Angular Material tener la documentación oficial de esta librería de componentes. En lo referente a botones debemos acceder a [Buttons & Indicators](#)

La etiqueta 'button' de HTML es afectada por esta librería, veremos que pasos debemos dar para utilizar 'button' con Material.

## Problema

Crear un tablero de Ta Te Ti con etiquetas 'button', afectar dichos controles con Material.

Crearemos primero el proyecto

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

Modificamos el archivo 'app.module.ts' para indicar que utilizaremos el módulo 'MatButtonModule':

```
C:\ Símbolo del sistema
Microsoft Windows [Versión 10.0.17134.765]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\codig>ng new proyecto019
```

```
C:\ Símbolo del sistema
Microsoft Windows [Versión 10.0.17134.765]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\codig>ng add @angular/material
```

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { MatButtonModule } from '@angular/material/button';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MatButtonModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Luego en la componente principal implementamos la interfaz visual y la lógica del juego del Ta Te Ti.  
El archivo 'app.component.ts':

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  posiciones=[[['-','-','-'],
               ['|','|','|'],
               ['|','|','|']];
  jugador='0';

  presion(fila:number,columna:number) {
    if (this.posiciones[fila][columna]=='-') {
      this.posiciones[fila][columna]=this.jugador;
      this.verificarGano('X');
      this.verificarGano('0');
      this.cambiarJugador();
    }
  }

  reiniciar() {
    for(let f=0;f<3;f++)
      for(let c=0;c<3;c++)
        this.posiciones[f][c]='-';
  }

  cambiarJugador() {
    if (this.jugador=='0')
      this.jugador='X';
    else
      this.jugador='0';
  }

  verificarGano(ficha: string) {
    if (this.posiciones[0][0]==ficha && this.posiciones[0][1]==ficha && this.posiciones[0][2]==ficha)
      alert('Gano:'+ficha);
    if (this.posiciones[1][0]==ficha && this.posiciones[1][1]==ficha && this.posiciones[1][2]==ficha)
      alert('Gano:'+ficha);
    if (this.posiciones[2][0]==ficha && this.posiciones[2][1]==ficha && this.posiciones[2][2]==ficha)
      alert('Gano:'+ficha);
    if (this.posiciones[0][0]==ficha && this.posiciones[1][0]==ficha && this.posiciones[2][0]==ficha)
      alert('Gano:'+ficha);
    if (this.posiciones[0][1]==ficha && this.posiciones[1][1]==ficha && this.posiciones[2][1]==ficha)
      alert('Gano:'+ficha);
    if (this.posiciones[0][2]==ficha && this.posiciones[1][2]==ficha && this.posiciones[2][2]==ficha)
      alert('Gano:'+ficha);
    if (this.posiciones[0][0]==ficha && this.posiciones[1][1]==ficha && this.posiciones[2][2]==ficha)
      alert('Gano:'+ficha);
    if (this.posiciones[0][2]==ficha && this.posiciones[1][1]==ficha && this.posiciones[2][0]==ficha)
      alert('Gano:'+ficha);
  }
}
```

El template de la componente lo tenemos en el archivo 'app.component.html':

```
<div style="text-align: center">
  <button mat-raised-button (click)="presion(0,0)" class="casilla">{{posiciones[0][0]}}</button>
  <button mat-raised-button (click)="presion(0,1)" class="casilla">{{posiciones[0][1]}}</button>
  <button mat-raised-button (click)="presion(0,2)" class="casilla">{{posiciones[0][2]}}</button>
  <br>
  <button mat-raised-button (click)="presion(1,0)" class="casilla">{{posiciones[1][0]}}</button>
  <button mat-raised-button (click)="presion(1,1)" class="casilla">{{posiciones[1][1]}}</button>
  <button mat-raised-button (click)="presion(1,2)" class="casilla">{{posiciones[1][2]}}</button>
  <br>
  <button mat-raised-button (click)="presion(2,0)" class="casilla">{{posiciones[2][0]}}</button>
  <button mat-raised-button (click)="presion(2,1)" class="casilla">{{posiciones[2][1]}}</button>
  <button mat-raised-button (click)="presion(2,2)" class="casilla">{{posiciones[2][2]}}</button>
</div>
<div style="text-align: center">
  <button mat-button color="primary" (click)="reiniciar()">Reiniciar</button>
</div>
```

El archivo de la hoja de estilo de la componente 'app.component.css':

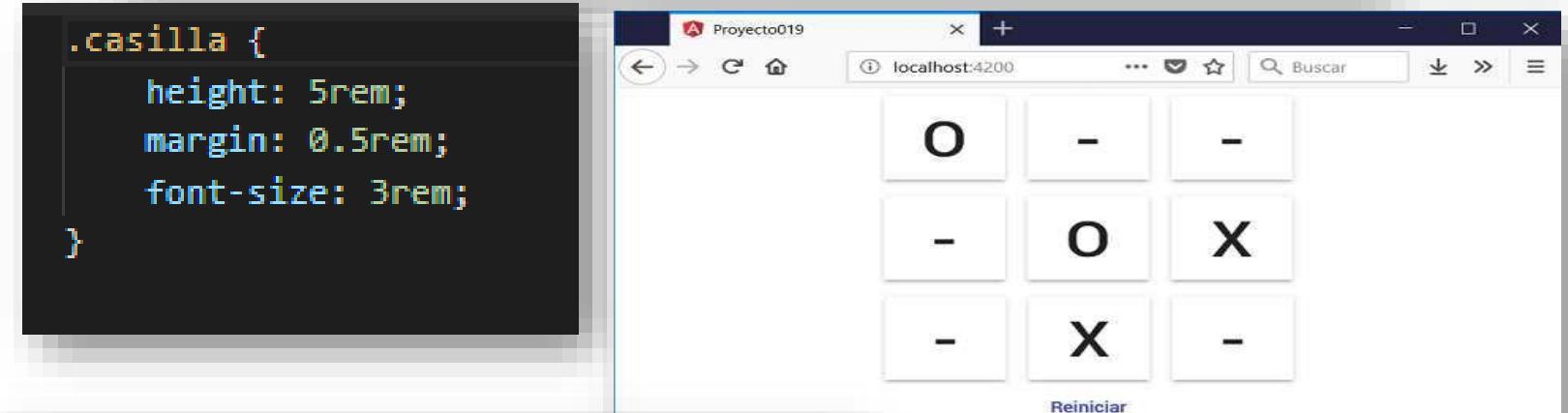
```
.casilla {
  height: 5rem;
  margin: 0.5rem;
  font-size: 3rem;
}
```

Para que la etiqueta HTML sea afectado por Material debemos agregar la propiedad 'mat-raised-button':

Otro estilo de botón se logra mediante la propiedad "mat-button" (no se muestran los bordes del botón):

```
<button mat-raised-button (click)="presion(0,0)" class="casilla">{{posiciones[0][0]}}</button>
```

```
<button mat-button color="primary" (click)="reiniciar()">Reiniciar</button>
```



## Acotaciones

Hay otras variantes de botones en Material:  
mat-button  
mat-raised-button  
mat-stroked-button  
mat-fab (Floating Action Button)  
mat-mini-fab



Hemos utilizado la etiqueta 'button' para definir acciones dentro de nuestra aplicación. Si lo que necesitamos es navegar a otra vista utilizaremos la etiqueta 'a' que perfectamente podemos aplicar las propiedades vistas para la etiqueta 'button'.

## 37 - Angular Material: Extensiones para el editor VS Code

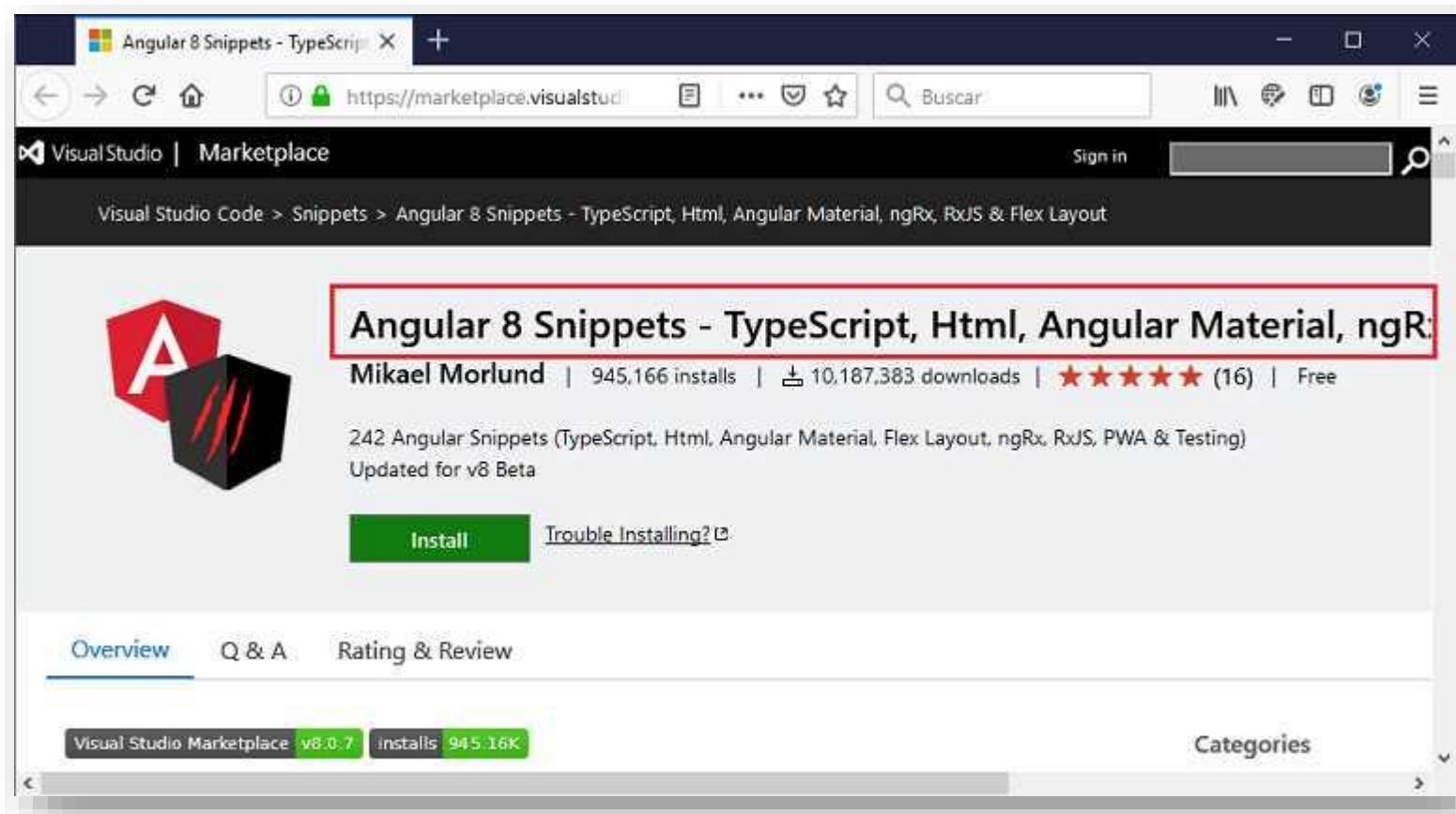
La complejidad actual en el desarrollo de aplicaciones web hace necesario disponer de herramientas que nos faciliten su construcción.

La cantidad de componentes creciente de Angular Material y su variedad de propiedades hacen necesario una gran cantidad de práctica para que lleguemos a dominarla y ser eficientes con el tiempo.

Al utilizar el editor VS Code podemos valernos de las extensiones especialmente desarrolladas para Angular.

La primer extensión que instalaremos con el objetivo de facilitar el desarrollo de aplicaciones que hagan uso de Angular Material es:

[Angular 6 Snippets - TypeScript, Html, Angular Material, ngRx, RxJS & Flex Layout](#):



Los más de 10000000 de descargar nos muestran la cantidad de desarrolladores que utilizan Angular en la construcciones de aplicaciones web.  
Una vez instalada la extensión disponemos de Intellisense para la codificación de las componentes Angular Material:

The screenshot shows an IDE interface with a code editor window titled "app.component.html". The code currently contains:

```
<div style="text-align: center">
  m-
```

A red box highlights the letter 'm-' as the user types. A dropdown menu appears, listing various Angular Material components starting with 'm-'. The item "mat-raised-button" is highlighted with a blue selection bar. To the right of the dropdown, a tooltip provides a brief description of the selected component:

Rectangular Material button w/ elevation. (Angular 8 Snippets - TypeScript, Html, Angular Material, ngRx, RxJS & Flex Layout)

<button mat-raised-button>text</button>

Cuando codificamos la plantilla HTML al escribir 'm-' nos aparece una lista con todas las componentes Angular Material y en segundo cuadro una descripción de la componente seleccionada y el trozo de código a generar.

Una vez confirmado el 'snippet' se agrega donde se encuentra el cursor el trozo de código:

The screenshot shows the same IDE window with the completed code:

```
<button mat-raised-button>text</button>
```

Además si hay que configurar varias partes del trozo de código añadido mediante la tecla 'tabulación' vamos saltando a cada uno de dichos puntos.

## Directivas

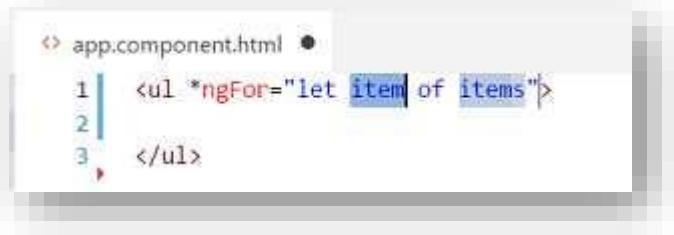
La extensión que acabamos de instalar no solo nos facilita la creación de componentes Material sino que además nos ayuda en la codificación de directivas. Supongamos que tenemos un vector con nombres de sitios webs definidos en el archivo \*.ts:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   sitios: string[] =["http://www.google.com",
10   "http://www.yahoo.com",
11   "http://www.bing.com"];
12 }
```

Luego cuando queremos mostrar los datos del vector podemos utilizar la extensión instalada para generar la directiva ng-for escribiendo 'ng-':



Luego de confirmar el 'snippet' nos muestra un nombre genérico para las variables del for:



```
<ul *ngFor="let item of items">
</ul>
```

Cambiamos cada nombre y presionamos la tecla 'tab':



```
<ul *ngFor="let sitio of sitios">
</ul>
```

Como podemos comprobar es una excelente herramienta para recordarnos los nombres de las directivas disponibles en Angular y generar en forma automática un trozo de código que requiere solo que lo configuremos a nuestras necesidades.

## 38 - Angular Material: formularios (input)

Entre los grupos de componentes fundamentales en Angular Material tenemos los referentes a formularios. Recordemos que la documentación oficial sobre los controles de formulario lo podemos encontrar [aquí](#). Confeccionaremos una pequeña aplicación para internalizar los pasos en el desarrollo con Angular Material y sus formularios.

## Problema

Definir un formulario web que permita cargar dos valores numéricos y al presionar un botón mostrar su suma.

Crearemos primero el proyecto

```
ng new proyecto020
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

```
ng add @angular/material
```

Modificamos el archivo 'app.module.ts' donde debemos importar los módulos de Angular Material que requiere nuestro proyecto:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
6 import { FormsModule } from '@angular/forms';
7
8 import { MatInputModule } from '@angular/material/input';
9 import { MatButtonModule } from '@angular/material/button';
10 @NgModule({
11   declarations: [
12     AppComponent
13   ],
14   imports: [
15     BrowserModule,
16     BrowserAnimationsModule,
17     MatInputModule,
18     MatButtonModule,
19     FormsModule ],
20   providers: [],
21   bootstrap: [AppComponent]
22 })
23 export class AppModule {}
```

Modificamos el archivo 'app.component.ts' con la lógica de nuestra componente que permite sumar dos números ingresados por teclado:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   valor1=0;
10  valor2=0;
11  resultado=0;
12
13  sumar() {
14    this.resultado = this.valor1 + this.valor2;
15  }
16 }
```

Codificamos la interfaz visual en el archivo 'app.component.html':

```
1 <div class="contenedor">
2
3   <mat-form-field>
4     <input matInput [(ngModel)]="valor1" type="number" placeholder="Ingrese primer valor">
5   </mat-form-field>
6
7   <mat-form-field>
8     <input matInput [(ngModel)]="valor2" type="number" placeholder="Ingrese segundo valor">
9   </mat-form-field>
10
11  <button mat-raised-button color="primary" (click)="sumar()">Sumar</button>
12  <p>La suma de los dos valores es:{{resultado}}</p>
13
14 </div>
15
```

El último archivo que debemos codificar es la hoja de estilo de la componente que se encuentra en 'app.component.css':

```
1 .contenedor {
2   display: flex;
3   flex-direction: column;
4   margin: 1rem auto;
5   max-width: 600px;
6 }
7
```

Si ejecutamos ahora la aplicación podemos comprobar cual es la estética de los controles de formulario de Angular Material:



Podemos probar esta aplicación en la web [aquí](#).

Si analizamos primero el HTML debemos agregar la directiva matInput en la etiqueta 'input', además de encerrar entre las marcas 'mat-form-field':

```
<mat-form-field>
  <input matInput [(ngModel)]="valor1" type="number" placeholder="Ingrese primer valor">
</mat-form-field>
```

El contenido de la propiedad placeholder es la que utiliza Angular Material para mostrar en la parte superior.  
Los botones ya lo vimos en conceptos anteriores, recordemos que debemos disponer la directiva mat-raised-button:

```
<button mat-raised-button color="primary" (click)="sumar()>Sumar</button>
```

El otro paso que no debemos olvidar es importar los módulos de Angular Material en el archivo 'app.module.ts'.

## Acotaciones

Hemos utilizado la etiqueta 'input' asignado a la propiedad type el valor 'number', los valores posibles para la propiedad type son:

- color
- date
- datetime-local
- email
- month
- number
- password
- search
- tel
- text
- time
- url
- week

## 39 - Angular Material: formularios - selectores mat-radio-button y mat-radio-group

La etiqueta mat-radio-button provee la misma funcionalidad de la etiqueta nativa de HTML <input type="radio"> pero con las ventajas de estilos y animaciones de Material Design.

Para agrupar un conjunto de mat-radio-button se utiliza la etiqueta mat-radio-group.

Problema

Definir un formulario web que permita cargar dos valores numéricos y mediante 4 'mat-radio-button' permitir indicar si queremos sumar, restar, multiplicar o dividir los valores ingresados. Efectuar la operación al presionar un botón.

Crearemos primero el proyecto

```
ng new proyecto021
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

```
ng add @angular/material
```

Modificamos el archivo 'app.module.ts' donde debemos importar los módulos de Angular Material que requiere nuestro proyecto:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
6
7 import { FormsModule } from '@angular/forms';
8
9 import { MatInputModule } from '@angular/material/input';
10 import { MatButtonModule } from '@angular/material/button';
11 import { MatRadioModule } from '@angular/material/radio';
12 import {MatFormFieldModule} from '@angular/material/form-field';
13
14 @NgModule({
15   declarations: [
16     AppComponent
17   ],
18   imports: [
19     BrowserModule,
20     BrowserAnimationsModule,
21     FormsModule,
22     MatInputModule,
23     MatButtonModule,
24     MatRadioModule,
25     MatFormFieldModule
26   ],
27   providers: [],
28   bootstrap: [AppComponent]
29 })
30 export class AppModule { }
```

Modificamos el archivo 'app.component.ts' con la lógica de nuestra componente que permita operar los dos valores ingresados dependiendo de la selección del mat-radio-button:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   valor1=null;
10  valor2=null;
11  resultado=null;
12
13  operacionSeleccionada: string = 'suma';
14  tipoOperaciones = [
15    'suma',
16    'resta',
17    'multiplicacion',
18    'division',
19  ];
20
21  operar() {
22    switch (this.operacionSeleccionada) {
23      case 'suma' : this.resultado = this.valor1 + this.valor2;
24      | | | break;
25      case 'resta' : this.resultado = this.valor1 - this.valor2;
26      | | | break;
27      case 'multiplicacion' : this.resultado = this.valor1 * this.valor2;
28      | | | | break;
29      case 'division' : this.resultado = this.valor1 / this.valor2;
30      | | | | break;
31    }
32  }
33 }
```

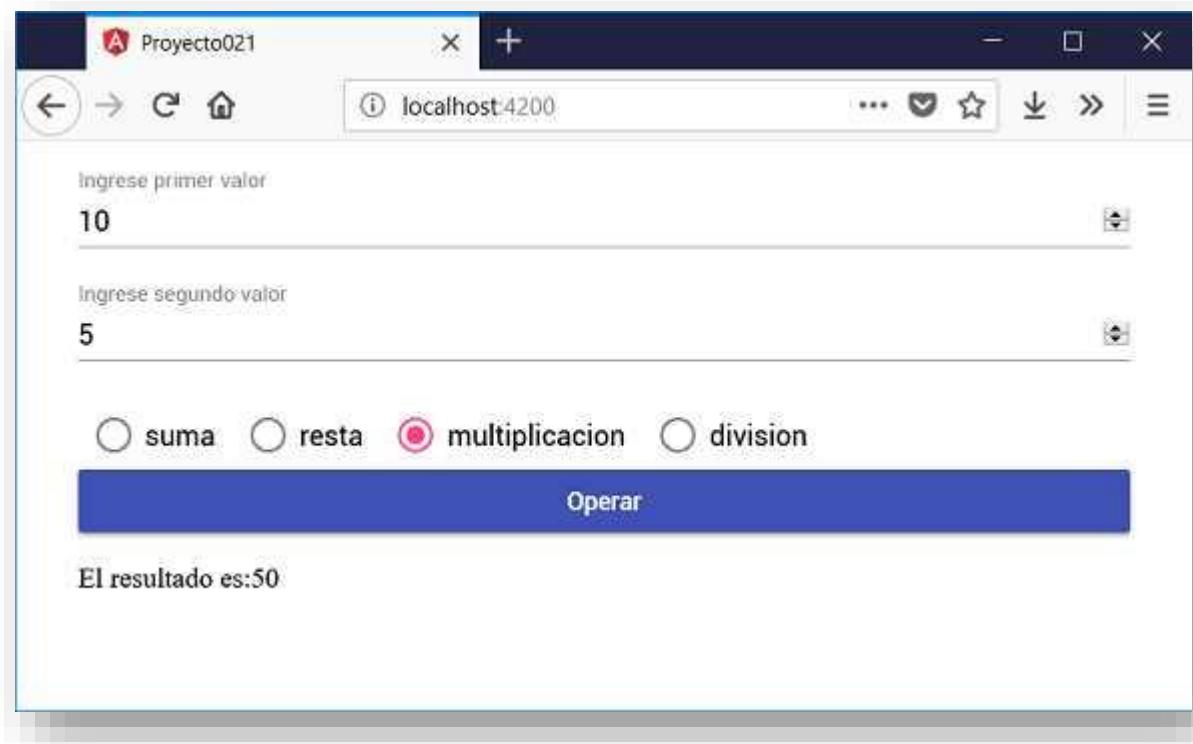
Codificamos la interfaz visual en el archivo 'app.component.html':

```
1 <div class="contenedor">
2
3     <mat-form-field>
4         |   <input matInput [(ngModel)]="valor1" type="number" placeholder="Ingrese primer valor">
5     </mat-form-field>
6
7     <mat-form-field>
8         |   <input matInput [(ngModel)]="valor2" type="number" placeholder="Ingrese segundo valor">
9     </mat-form-field>
10
11    <mat-radio-group name="opciones" [(ngModel)]="operacionSeleccionada">
12        |   <mat-radio-button *ngFor="let op of tipoOperaciones" name="opciones" [value]="op" style="margin: 10px">
13            {{op}}
14        </mat-radio-button>
15    </mat-radio-group>
16
17    <button mat-raised-button color="primary" (click)="operar()">Operar</button>
18    <p>El resultado es:{{resultado}}</p>
19
20
21 </div>
```

El último archivo que debemos codificar es la hoja de estilo de la componente que se encuentra en 'app.component.css':

```
9
10    .contenedor {
11        display: flex;
12        flex-direction: column;
13        margin: 1rem auto;
14        max-width: 600px;
15    }
16
17
```

Si ejecutamos ahora la aplicación podemos comprobar cual es la estética de la componente mat-radio-button:



Podemos probar esta aplicación en la web [aquí](#).

En el archivo 'app.component.html' definimos el selector 'mat-radio-group' y dentro del mismo los cuatro 'mat-radio-button' que los generamos mediante la directiva ngFor por facilidad:

```
<mat-radio-group name="opciones" [(ngModel)]="operacionSeleccionada">
  <mat-radio-button *ngFor="let op of tipoOperaciones" name="opciones" [value]="op" style="margin: 10px">
    {{op}}
  </mat-radio-button>
</mat-radio-group>
```

En el archivo 'app.component.ts' definimos un atributo donde se almacena cual opción se encuentra seleccionada y un vector con todas las opciones de los mat-radio-button:

```
operacionSeleccionada: string = 'suma';
tipoOperaciones = [
  'suma',
  'resta',
  'multiplicacion',
  'division',
];
```

En el archivo 'app.module.ts' importamos todos los módulos necesarios para trabajar con este formulario de Material:

```
import { MatInputModule } from '@angular/material/input';
import { MatButtonModule } from '@angular/material/button';
import { MatRadioModule } from '@angular/material/radio';
import { MatFormFieldModule } from '@angular/material/form-field';
```

## 40 - Angular Material: formularios - selectores mat-checkbox

Otra etiqueta común en un formulario son los mat-checkbox que provee la misma funcionalidad de la etiqueta nativa de HTML <input type="checkbox"> pero con las ventajas de estilos y animaciones de Material Design.

### Problema

Definir un formulario web que permita cargar dos valores numéricos y mediante 4 'mat-checkbox' permitir indicar si queremos sumar, restar, multiplicar y/o dividir los valores ingresados. Efectuar la operación al presionar un botón.

Crearemos primero el proyecto

```
ng new proyecto022
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

```
ng add @angular/material
```

Modificamos el archivo 'app.module.ts' donde debemos importar los módulos de Angular Material que requiere nuestro proyecto:

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppComponent } from './app.component';
5  import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
6
7  import { FormsModule } from '@angular/forms';
8
9  import { MatInputModule } from '@angular/material/input';
10 import { MatButtonModule } from '@angular/material/button';
11 import { MatCheckboxModule } from '@angular/material/checkbox';
12
13 @NgModule({
14   declarations: [
15     AppComponent
16   ],
17   imports: [
18     BrowserModule,
19     BrowserAnimationsModule,
20     FormsModule,
21     MatInputModule,
22     MatButtonModule,
23     MatCheckboxModule
24   ],
25   providers: [],
26   bootstrap: [AppComponent]
27 })
28 export class AppModule {}
```

Como vemos aparece el nuevo módulo MatCheckboxModule.

Modificamos el archivo 'app.component.ts' con la lógica de nuestra componente que permita operar los dos valores ingresados dependiendo de las selecciones de los mat-checkbox:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   valor1=null;
10  valor2=null;
11  resultado=null;
12  opcion1=false;
13  opcion2=false;
14  opcion3=false;
15  opcion4=false;
16
17  operar() {
18    this.resultado='';
19    if (this.opcion1) {
20      let ope = this.valor1 + this.valor2;
21      this.resultado+=`La suma es ${ope} `;
22    }
23    if (this.opcion2) {
24      let ope = this.valor1 - this.valor2;
25      this.resultado+=`La resta es ${ope} `;
26    }
27    if (this.opcion3) {
28      let ope = this.valor1 * this.valor2;
29      this.resultado+=`El producto es ${ope} `;
30    }
31    if (this.opcion4) {
32      let ope = this.valor1 / this.valor2;
33      this.resultado+=`La division es ${ope} `;
34    }
35  }
36 }
```

Codificamos la interfaz visual en el archivo 'app.component.html':

```
1 <div class="contenedor">
2
3   <mat-form-field>
4     <input matInput [(ngModel)]="valor1" type="number" placeholder="Ingrese primer valor">
5   </mat-form-field>
6
7   <mat-form-field>
8     <input matInput [(ngModel)]="valor2" type="number" placeholder="Ingrese segundo valor">
9   </mat-form-field>
10
11  <mat-checkbox [(ngModel)]="opcion1">Sumar</mat-checkbox>
12  <mat-checkbox [(ngModel)]="opcion2">Restar</mat-checkbox>
13  <mat-checkbox [(ngModel)]="opcion3">Multiplicar</mat-checkbox>
14  <mat-checkbox [(ngModel)]="opcion4">Dividir</mat-checkbox>
15
16  <button mat-raised-button color="primary" (click)="operar()">Operar</button>
17  <p>{{resultado}}</p>
18
19 </div>
20
```

El último archivo que debemos codificar es la hoja de estilo de la componente que se encuentra en 'app.component.css':

```
18
19 .contenedor {
20   display: flex;
21   flex-direction: column;
22   margin: 1rem auto;
23   max-width: 600px;
24 }
25
26
```

Si ejecutamos ahora la aplicación podemos comprobar cual es la estética de la componente mat-checkbox:



Podemos probar esta aplicación en la web [aquí](#).

En el archivo 'app.component.html' definimos el selector 'mat-checkbox':

```
<mat-checkbox [(ngModel)]="opcion1">Sumar</mat-checkbox>
<mat-checkbox [(ngModel)]="opcion2">Restar</mat-checkbox>
<mat-checkbox [(ngModel)]="opcion3">Multiplicar</mat-checkbox>
<mat-checkbox [(ngModel)]="opcion4">Dividir</mat-checkbox>
```

En el archivo 'app.component.ts' definimos cuatro atributos donde se almacenan cual de los mat-checkbox se encuentran seleccionados:

```
export class AppComponent {
  valor1=null;
  valor2=null;
  resultado=null;
  opcion1=false;
  opcion2=false;
  opcion3=false;
  opcion4=false;
```

En el archivo 'app.module.ts' importamos todos los módulos necesarios para trabajar con este formulario de Material:

```
import { FormsModule } from '@angular/forms';

import { MatInputModule } from '@angular/material/input';
import { MatButtonModule } from '@angular/material/button';
import { MatCheckboxModule } from '@angular/material/checkbox';
```

## 41 - Angular Material: formularios - selector mat-select

Otra etiqueta común en un formulario son los mat-select que provee la misma funcionalidad de la etiqueta nativa de HTML <select> pero con las ventajas de estilos y animaciones de Material Design.

### Problema

Definir un formulario web que permita cargar dos valores numéricos y mediante una componente 'mat-select' indicar si queremos sumar, restar, multiplicar o dividir los valores ingresados. Efectuar la operación al presionar un botón.

Crearemos primero el proyecto

```
ng new proyecto023
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

```
ng add @angular/material
```

Modificamos el archivo 'app.module.ts' donde debemos importar los módulos de Angular Material que requiere nuestro proyecto:

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3
4  import { AppComponent } from './app.component';
5  import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
6
7  import { FormsModule } from '@angular/forms';
8
9  import { MatInputModule } from '@angular/material/input';
10 import { MatButtonModule } from '@angular/material/button';
11 import { MatFormFieldModule } from '@angular/material/form-field';
12 import { MatSelectModule } from '@angular/material/select';
13
14
15 @NgModule({
16   declarations: [
17     AppComponent
18   ],
19   imports: [
20     BrowserModule,
21     BrowserAnimationsModule,
22     FormsModule,
23     MatInputModule,
24     MatButtonModule,
25     MatFormFieldModule,
26     MatSelectModule
27   ],
28   providers: [],
29   bootstrap: [AppComponent]
30 })
31 export class AppModule {}
```

Como vemos aparece el nuevo módulo MatSelectedModule.

Modificamos el archivo 'app.component.ts' con la lógica de nuestra componente que permita operar los dos valores ingresados dependiendo de la selección del mat-select:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   valor1=null;
10  valor2=null;
11  resultado=null;
12
13  operaciones = [
14    {valor:'suma', muestraValor:'Sumar'},
15    {valor:'resta', muestraValor:'Restar'},
16    {valor:'multiplicacion', muestraValor:'Multiplicar'},
17    {valor:'division', muestraValor:'Dividir'}
18  ];
19
20  seleccionada: string = this.operaciones[0].valor;
21
22  operar() {
23    switch (this.seleccionada) {
24      case 'suma' : this.resultado = this.valor1 + this.valor2;
25        break;
26      case 'resta' : this.resultado = this.valor1 - this.valor2;
27        break;
28      case 'multiplicacion' : this.resultado = this.valor1 * this.valor2;
29        break;
30      case 'division' : this.resultado = this.valor1 / this.valor2;
31        break;
32    }
33  }
34
35 }
```

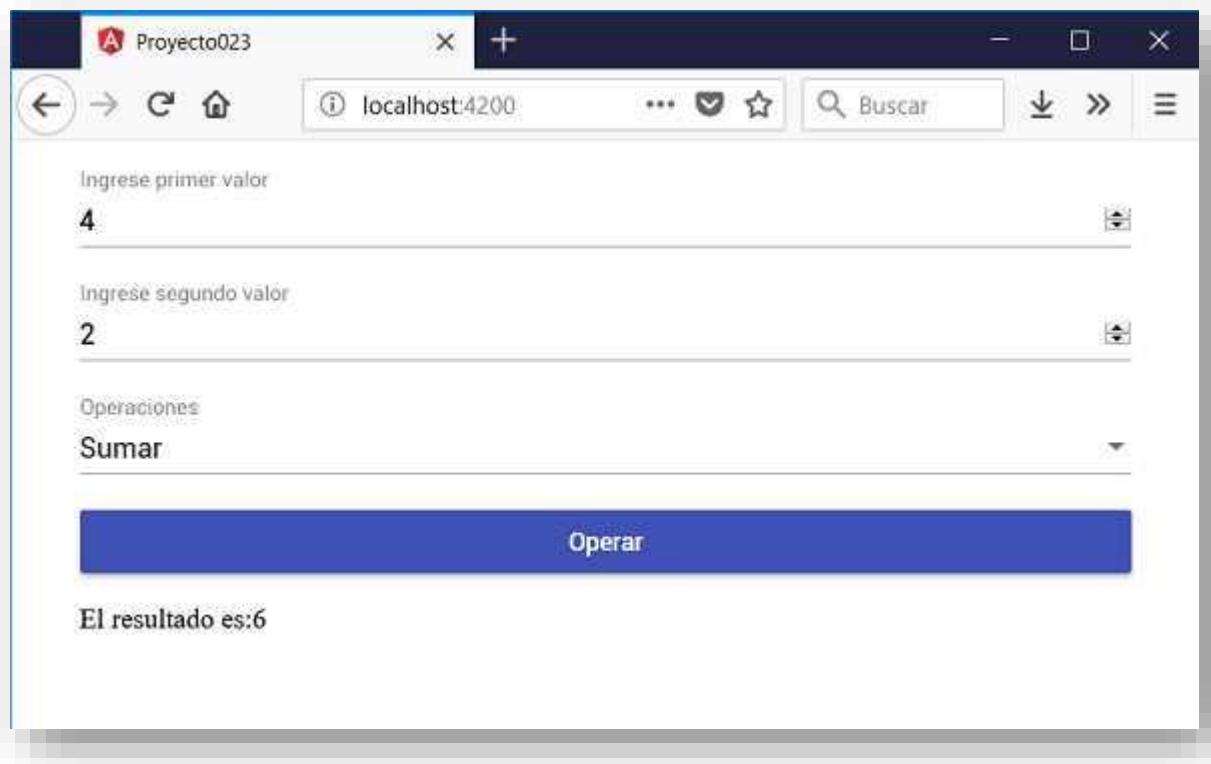
Codificamos la interfaz visual en el archivo 'app.component.html':

```
1 <div class="contenedor">
2   <mat-form-field>
3     <input matInput [(ngModel)]="valor1" type="number" placeholder="Ingrese primer valor">
4   </mat-form-field>
5
6   <mat-form-field>
7     <input matInput [(ngModel)]="valor2" type="number" placeholder="Ingrese segundo valor">
8   </mat-form-field>
9
10  <mat-form-field>
11    <mat-select placeholder="Operaciones" [(ngModel)]="seleccionada" name="operacion">
12      <mat-option *ngFor="let op of operaciones" [value]="op.valor">
13        {{op.muestraValor}}
14      </mat-option>
15    </mat-select>
16  </mat-form-field>
17
18  <button mat-raised-button color="primary" (click)="operar()">Operar</button>
19  <p>El resultado es:{{resultado}}</p>
20
21
22 </div>
23
24
```

El último archivo que debemos codificar es la hoja de estilo de la componente que se encuentra en 'app.component.css':

```
26
27  .contenedor {
28    display: flex;
29    flex-direction: column;
30    margin: 1rem auto;
31    max-width: 600px;
32  }
33
34
```

Si ejecutamos ahora la aplicación podemos comprobar cual es la estética de la componente mat-select:



Podemos probar esta aplicación en la web [aquí](#).

En el archivo 'app.component.html' definimos el selector 'mat-select' contenido dentro de un mat-form-field:

```
<mat-form-field>
  <mat-select placeholder="Operaciones" [(ngModel)]="seleccionada" name="operacion">
    <mat-option *ngFor="let op of operaciones" [value]="op.valor">
      {{op.muestraValor}}
    </mat-option>
  </mat-select>
</mat-form-field>
```

En el archivo 'app.component.ts' definimos un vector que almacena cuatro objetos indicando el valor a mostrar dentro de la componente mat-select y el valor asociado a dicha opción:

```
operaciones = [
  {valor:'suma', muestraValor:'Sumar'},
  {valor:'resta', muestraValor:'Restar'},
  {valor:'multiplicacion', muestraValor:'Multiplicar'},
  {valor:'division', muestraValor:'Dividir'}
];

seleccionada: string = this.operaciones[0].valor;
```

El atributo 'seleccionada' almacena el valor por defecto que debe mostrarse seleccionado dentro del mat-select.

En el archivo 'app.module.ts' importamos todos los módulos necesarios para trabajar con este formulario de Material:

```
import { FormsModule } from '@angular/forms';

import { MatInputModule } from '@angular/material/input';
import { MatButtonModule } from '@angular/material/button';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatSelectModule } from '@angular/material/select';
```

## 42 - Angular Material: formularios - mat-slider

Veremos ahora la componente mat-slider que provee la misma funcionalidad de la etiqueta nativa de HTML <input type="range"> pero con las ventajas de estilos y animaciones de Material Design.

Problema

Definir tres controles mat-slider con valores que varíen entre 1 y 200. Mostrar la suma de los tres slider.

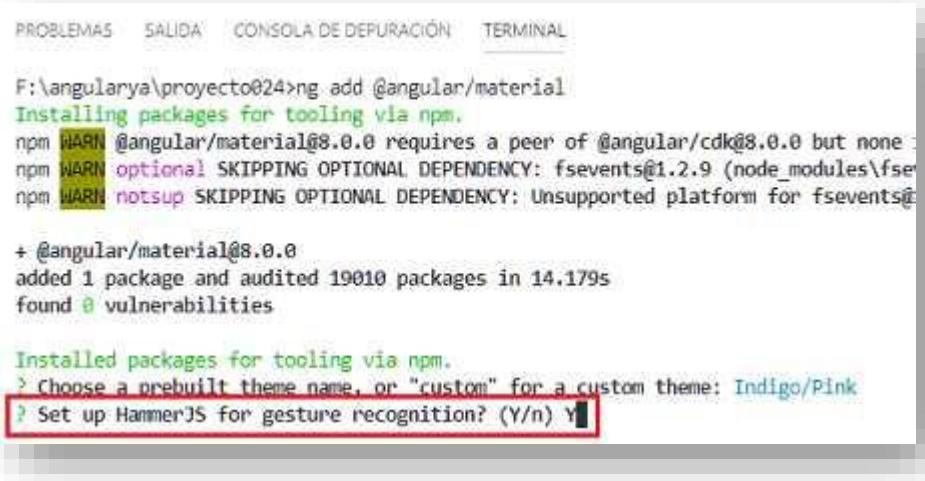
Crearemos primero el proyecto

```
ng new proyecto024
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

```
ng add @angular/material
```

Para utilizar la componente mat-slider necesitamos instalar la librería HammerJS que se nos consulta al ejecutar ng add @angular/material:



```
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

F:\angularaya\proyecto024>ng add @angular/material
Installing packages for tooling via npm.
npm WARN @angular/material@8.0.0 requires a peer of @angular/cdk@8.0.0 but none was found
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@1.2.9 (node_modules\fsevents)
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@1.2.9: wanted {"os": "darwin", "arch": "any"} (current: {"os": "win32", "arch": "x64"})

+ @angular/material@8.0.0
added 1 package and audited 19010 packages in 14.179s
found 0 vulnerabilities

Installed packages for tooling via npm.
> Choose a prebuilt theme name, or "custom" for a custom theme: Indigo/Pink
? Set up HammerJS for gesture recognition? (Y/n) Y
```

Modificamos el archivo 'app.module.ts' donde debemos importar los módulos de Angular Material que requiere nuestro proyecto:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
6
7 import { FormsModule } from '@angular/forms';
8 import { MatSliderModule } from '@angular/material/slider';
9
10
11 @NgModule({
12   declarations: [
13     AppComponent
14   ],
15   imports: [
16     BrowserModule,
17     BrowserAnimationsModule,
18     FormsModule,
19     MatSliderModule
20   ],
21   providers: [],
22   bootstrap: [AppComponent]
23 })
24 export class AppModule {}
```

Como vemos aparece el nuevo módulo MatSliderModule.

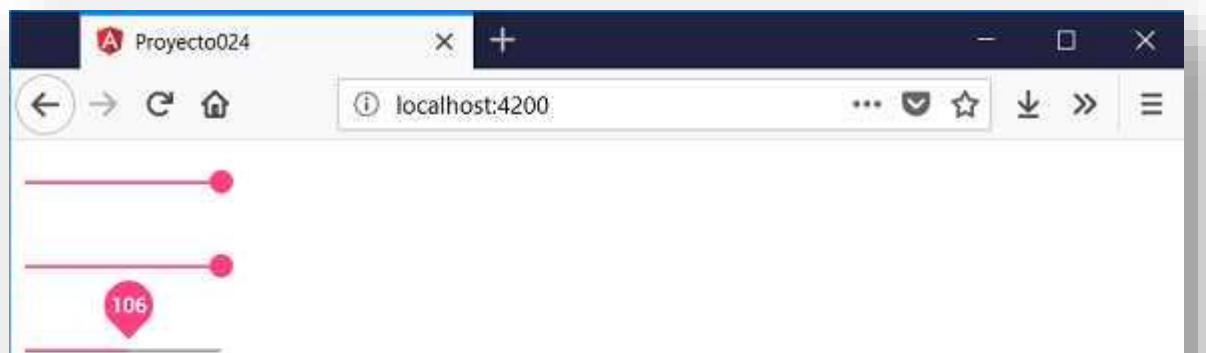
Modificamos el archivo 'app.component.ts' :

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   slider1=0;
10  slider2=0;
11  slider3=0;
12  suma=0;
13
14  cambiar() {
15    this.suma = this.slider1 + this.slider2 + this.slider3;
16  }
17
18 }
```

Codificamos la interfaz visual en el archivo 'app.component.html':

```
1  <mat-slider [(ngModel)]="slider1" thumbLabel (change)="cambiar()" min="1" max="200"></mat-slider><br>
2  <mat-slider [(ngModel)]="slider2" thumbLabel (change)="cambiar()" min="1" max="200"></mat-slider><br>
3  <mat-slider [(ngModel)]="slider3" thumbLabel (change)="cambiar()" min="1" max="200"></mat-slider><br>
4  La suma de los tres slider:{suma}
5
6
```

Si ejecutamos ahora la aplicación podemos comprobar cual es la estética de la componente mat-slider:



La suma de los tres slider:506

Podemos probar esta aplicación en la web [aquí](#).

En el archivo 'app.component.html' definimos los tres selectores 'mat-slider' :

```
<mat-slider [(ngModel)]="slider1" thumbLabel (change)="cambiar()" min="1" max="200"></mat-slider><br>
<mat-slider [(ngModel)]="slider2" thumbLabel (change)="cambiar()" min="1" max="200"></mat-slider><br>
<mat-slider [(ngModel)]="slider3" thumbLabel (change)="cambiar()" min="1" max="200"></mat-slider><br>
```

En el archivo 'app.module.ts' importamos todos los módulos necesarios:

```
import { FormsModule } from '@angular/forms';
import { MatSliderModule } from '@angular/material/slider';
```

Agregamos las tres imágenes de las banderas en una carpeta llamada 'imagenes' que se debe crear dentro de la carpeta 'assets':

```

```

## 43 - Angular Material: formularios - mat-slide-toggle

Otro control de formulario disponible en Angular Material con un objetivo similar a los checkbox de HTML es la componente 'mat-slide-toggle'.

La componente 'mat-slide-toggle' puede estar encendida o apagada. El cambio se produce mediante un clic del mouse o el arrastre (de uso común en un dispositivo táctil)

Problema

Definir un formulario web que permita cargar dos valores numéricos y mediante 4 'mat-slide-toggle' permitir indicar si queremos sumar, restar, multiplicar y/o dividir los valores ingresados. Efectuar la operación al presionar un botón.

Crearemos primero el proyecto

```
ng new proyecto025
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add' (Cargar la librería HammerJS para que funcione correctamente el control mat-slide-toggle):

```
ng add @angular/material
```

Modificamos el archivo 'app.module.ts' donde debemos importar los módulos de Angular Material que requiere nuestro proyecto:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
6
7 import { FormsModule } from '@angular/forms';
8 import { MatInputModule } from '@angular/material/input';
9 import { MatButtonModule } from '@angular/material/button';
10 import { MatSlideToggleModule } from '@angular/material/slide-toggle';
11
12 @NgModule({
13   declarations: [
14     AppComponent
15   ],
16   imports: [
17     BrowserModule,
18     BrowserAnimationsModule,
19     FormsModule,
20     MatInputModule,
21     MatButtonModule,
22     MatSlideToggleModule
23   ],
24   providers: [],
25   bootstrap: [AppComponent]
26 })
27 export class AppModule {}
```

Como vemos aparece el nuevo módulo MatSlideToggleModule.

Modificamos el archivo 'app.component.ts' con la lógica de nuestra componente que permita operar los dos valores ingresados dependiendo de las selecciones de los mat-slide-toggle:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   valor1=null;
10  valor2=null;
11  resultado=null;
12  opcion1=false;
13  opcion2=false;
14  opcion3=false;
15  opcion4=false;
16
17
18  operar() {
19    this.resultado='';
20    if (this.opcion1) {
21      let ope = this.valor1 + this.valor2;
22      this.resultado+=`La suma es ${ope} `;
23    }
24    if (this.opcion2) {
25      let ope = this.valor1 - this.valor2;
26      this.resultado+=`La resta es ${ope} `;
27    }
28    if (this.opcion3) {
29      let ope = this.valor1 * this.valor2;
30      this.resultado+=`El producto es ${ope} `;
31    }
32    if (this.opcion4) {
33      let ope = this.valor1 / this.valor2;
34      this.resultado+=`La division es ${ope} `;
35    }
36  }
37 }
```

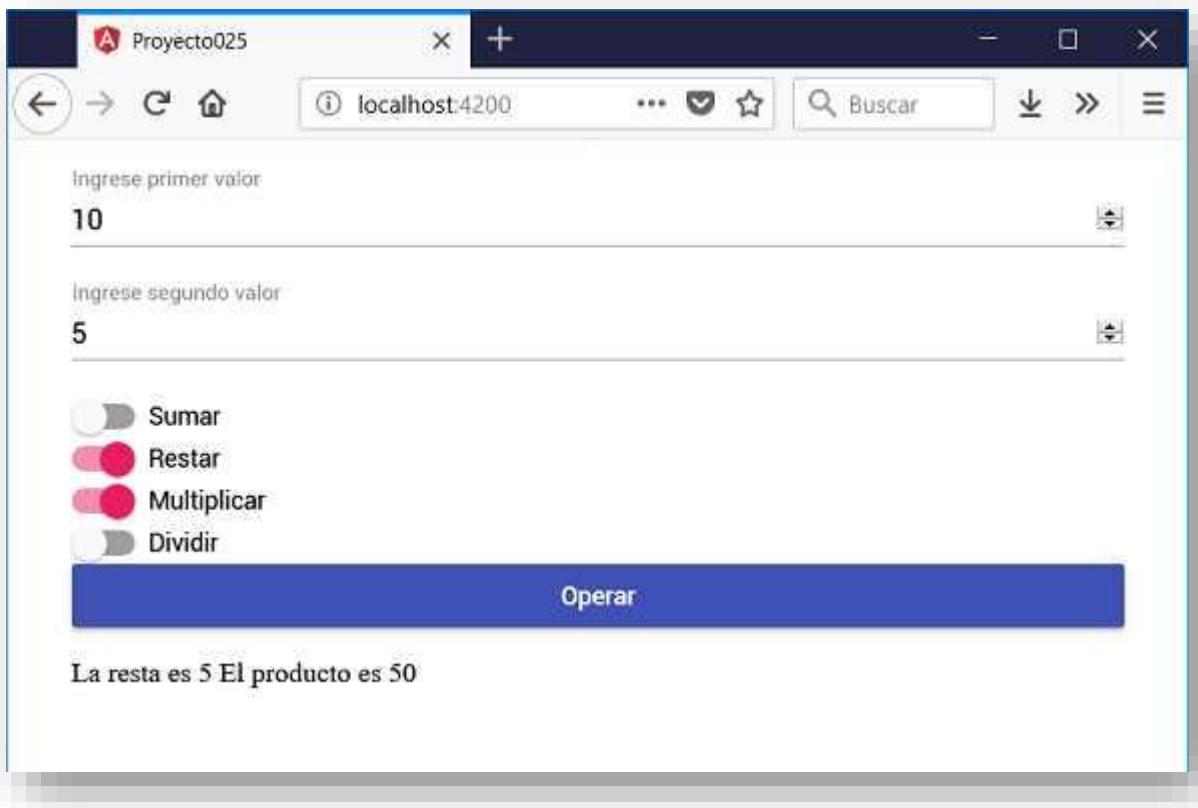
Codificamos la interfaz visual en el archivo 'app.component.html':

```
1 <div class="contenedor">
2   <mat-form-field>
3     <input matInput [(ngModel)]="valor1" type="number" placeholder="Ingrese primer valor">
4   </mat-form-field>
5
6   <mat-form-field>
7     <input matInput [(ngModel)]="valor2" type="number" placeholder="Ingrese segundo valor">
8   </mat-form-field>
9
10  <mat-slide-toggle [(ngModel)]="opcion1">Sumar</mat-slide-toggle>
11  <mat-slide-toggle [(ngModel)]="opcion2">Restar</mat-slide-toggle>
12  <mat-slide-toggle [(ngModel)]="opcion3">Multiplicar</mat-slide-toggle>
13  <mat-slide-toggle [(ngModel)]="opcion4">Dividir</mat-slide-toggle>
14
15  <button mat-raised-button color="primary" (click)="operar()">Operar</button>
16
17  <p>{{resultado}}</p>
18
19
20 </div>
21
```

La hoja de estilo de la componente que se encuentra en 'app.component.css' es:

```
1 .contenedor {
2   display: flex;
3   flex-direction: column;
4   margin: 1rem auto;
5   max-width: 600px;
6 }
7
8
```

Si ejecutamos ahora la aplicación podemos comprobar cual es la estética de la componente mat-slide-toggle:



Podemos probar esta aplicación en la web [aquí](#).

En el archivo 'app.component.html' definimos el selector 'mat-slide-toggle':

```
<mat-slide-toggle [(ngModel)]="opcion1">Sumar</mat-slide-toggle>
<mat-slide-toggle [(ngModel)]="opcion2">Restar</mat-slide-toggle>
<mat-slide-toggle [(ngModel)]="opcion3">Multiplicar</mat-slide-toggle>
<mat-slide-toggle [(ngModel)]="opcion4">Dividir</mat-slide-toggle>
```

En el archivo 'app.component.ts' definimos cuatro atributos donde se almacenan cual de los mat-checkbox se encuentran seleccionados:

```
export class AppComponent {  
  valor1=null;  
  valor2=null;  
  resultado=null;  
  opcion1=false;  
  opcion2=false;  
  opcion3=false;  
  opcion4=false;
```

En el archivo 'app.module.ts' importamos todos los módulos necesarios para trabajar con este formulario de Material:

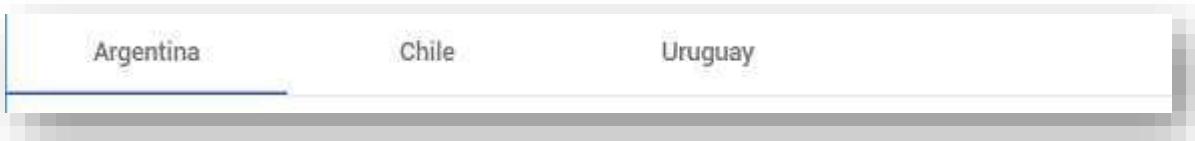
```
import { FormsModule } from '@angular/forms';  
import { MatInputModule } from '@angular/material/input';  
import { MatButtonModule } from '@angular/material/button';  
import { MatSlideToggleModule } from '@angular/material/slide-toggle';
```

Y los añadimos en la propiedad imports del decorador:

```
imports: [  
  BrowserModule,  
  BrowserAnimationsModule,  
  FormsModule,  
  MatInputModule,  
  MatButtonModule,  
  MatSlideToggleModule  
],
```

## 44 - Angular Material: Layout - tabs

Angular material nos permite organizar vistas que se hacen visibles mediante distintas pestañas:  
El título de cada pestaña se muestra en el encabezado y la pestaña activa tiene una barra distinta:



Cuando la lista de etiquetas de pestañas excede el ancho del encabezado, los controles de paginación aparecen para permitir al usuario desplazarse hacia la izquierda y hacia la derecha a través de las etiquetas:



Para el empleo de las pestañas necesitamos las etiquetas mat-tab-group y dentro de esta mat-tab por cada pestaña.

Problema

Mostrar tres pestañas con los datos de tres países.

Crearemos primero el proyecto

```
ng new proyecto026
```

Procedemos a instalar todas las dependencias de Angular Material ayudados por Angular CLI mediante el comando 'add':

```
ng add @angular/material
```

Modificamos el archivo 'app.module.ts' donde debemos importar MatTabsModule:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3
4 import { AppComponent } from './app.component';
5 import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
6
7 import { MatTabsModule } from '@angular/material/tabs';
8
9 @NgModule({
10   declarations: [
11     AppComponent
12   ],
13   imports: [
14     BrowserModule,
15     BrowserAnimationsModule,
16     MatTabsModule
17   ],
18   providers: [],
19   bootstrap: [AppComponent]
20 })
21 export class AppModule {}
```

Codificamos la interfaz visual en el archivo 'app.component.html':

```
1 <mat-tab-group>
2   <mat-tab label="Argentina">
3     <div class="datos">
4       <p>
5         | <strong>Nombre del país:</strong>Argentina
6       </p>
7       <p>
8         | 
9       </p>
10      <h3>Datos generales.</h3>
11      <p>Argentina, llamada oficialmente República Argentina, es un país soberano de América del Sur, ubicado en el extremo
12        sur y suroeste de dicho subcontinente. Adopta la forma de gobierno republicana, democrática, representativa y federal.</p>
13      <p>La Argentina está organizada como un Estado federal descentralizado, integrado desde 1994 por un Estado nacional y
14        24 estados autogobernados, siendo estos sus 23 provincias sumada la ciudad autónoma de Buenos Aires designada
15        como Capital Federal del país. Cada estado tiene autonomía política, constitución, bandera y cuerpo de seguridad
16        propios. Las 23 provincias mantienen todos los poderes no delegados al Estado nacional y garantizan la autonomía
17        de sus municipios.</p>
18      </div>
19    </mat-tab>
20    <mat-tab label="Chile">
21      <div class="datos">
22        <p>
23          | <strong>Nombre del país:</strong>Chile
24        </p>
25        <p>
26          | 
27        </p>
28        <h3>Datos generales.</h3>
29        <p>Chile es un país de América ubicado en el extremo sudoeste de América del Sur. Su nombre oficial es República de Chile
30        y su capital es la ciudad de Santiago. Primer país sudamericano en ingresar a la Organización para la Cooperación
31        y el Desarrollo Económicos, Chile es una de las economías de América Latina que más ha crecido desde mediados de
32        la década de 1980.</p>
33        <p>Antes del descubrimiento de América, las tierras situadas al sur del desierto de Atacama ya se llamaban Chili en la
34        tradición indígena. Una vez instalados en Nueva Castilla y Nueva Toledo, los conquistadores españoles siguieron
35        llamando de esa forma a la región del sur, a veces también conocida como «valle de Chile», nombre que se extendió
36        posteriormente a todo el actual país.</p>
37      </div>
38    </mat-tab>
39    <mat-tab label="Uruguay">
40      <div class="datos">
41        <p>
42          | <strong>Nombre del país:</strong>Uruguay
43        </p>
44        <p>
45          | 
46        </p>
47        <h3>Datos generales.</h3>
48        <p>Uruguay, oficialmente República Oriental del Uruguay, es un país de América del Sur, situado en la parte oriental del
49          Cono Sur Americano. Limita al noreste con Brasil -estado de Río Grande del Sur-, al oeste con Argentina -provincias
50          de Entre Ríos y Corrientes- y tiene costas en el océano Atlántico al sureste y sobre el Río de la Plata hacia el
51          sur. Abarca 176.215 km2 y es el segundo país más pequeño de Sudamérica, después de Surinam.1 Según los datos del
52          último censo del INE en 2011, la población de Uruguay es de 3.290.454 habitantes, por lo que figura en la décima
53          posición entre los países sudamericanos.Uruguay, oficialmente República Oriental del Uruguay, es un país de América
54          del Sur, situado en la parte oriental del Cono Sur Americano. Limita al noreste con Brasil -estado de Río Grande
55          del Sur-, al oeste con Argentina -provincias de Entre Ríos y Corrientes- y tiene costas en el océano Atlántico al
56          sureste y sobre el Río de la Plata hacia el sur. Abarca 176.215 km2 y es el segundo país más pequeño de Sudamérica,
57          después de Surinam.1 Según los datos del último censo del INE en 2011, la población de Uruguay es de 3.290.454 habitantes,
58          por lo que figura en la décima posición entre los países sudamericanos.
59        </p>
60      </div>
61    </mat-tab>
62  </mat-tab-group>
63
```

Agregamos las tres imágenes de las banderas en una carpeta llamada imágenes que se debe crear dentro de la carpeta 'assets':

```

```

Si ejecutamos la aplicación tenemos como resultado:



Podemos probar esta aplicación en la web [aquí](#).

La estructura fundamental del archivo HTML para definir las pestañas es:

```
<mat-tab-group>
  <mat-tab label="Argentina">
    <p>datos</p>
  </mat-tab>
  <mat-tab label="Chile">
    <p>datos</p>
  </mat-tab>
  <mat-tab label="Uruguay">
    <p>datos</p>
  </mat-tab>
</mat-tab-group>
```

El valor almacenado en la propiedad `label` de la etiqueta '`mat-tab`' es lo que se muestra como título de la pestaña.