Angular

Raúl Novoa

Co-founder at @10labs @fidelizoo @neki_global

JS MEAN Developer / iOS Developer

http://linkedin.com/in/raulnovoa

@raul_novoa

raul@10labs.es

Angular

Temario

- Conceptos básicos de JS
- Typescript
- ¿Como funciona Angular?
- Componentes
- Visión general
- Data binding
- Directivas
- Formularios
- Inyección de dependencias
- HTTP
- Rutas

0.

Herramientas

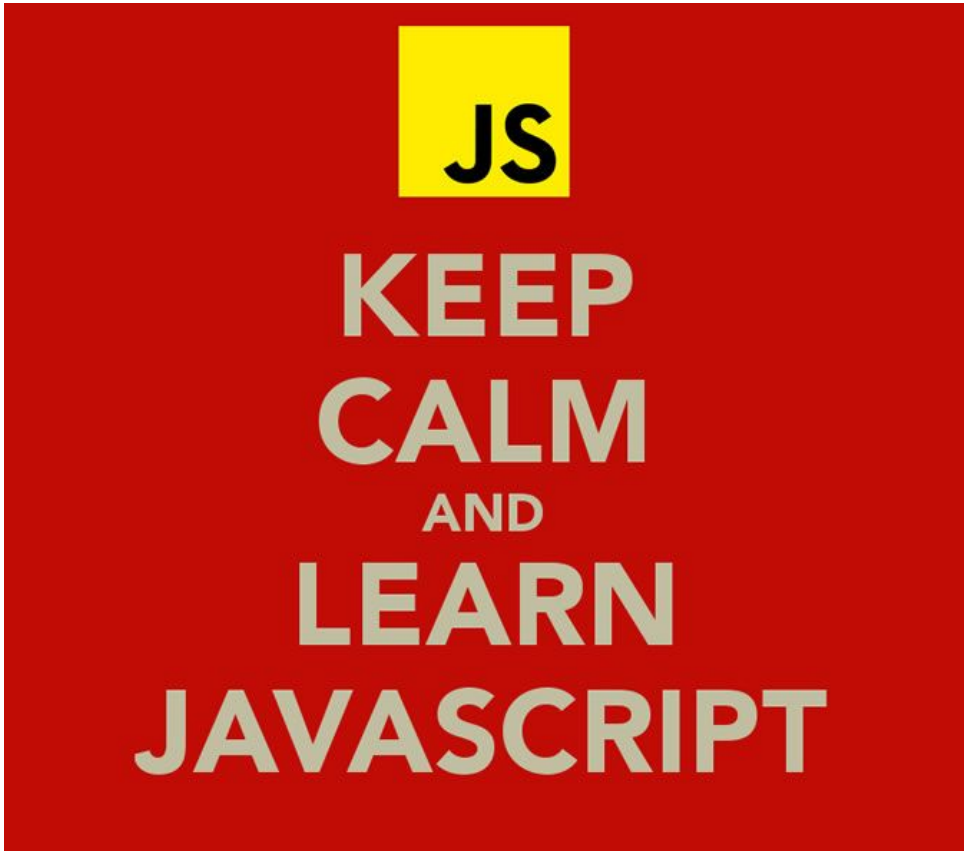
Herramientas

- NodeJS (https://nodejs.org/es/)
- Python 2.7.x (https://www.python.org/downloads/)
- Angular-cli (https://github.com/angular/angular-cli)
- TypeScript (npm install -g typescript)
- Tsun (npm install -g tsun)
- Visual studio code (https://code.visualstudio.com)

ng new practica1

0.1.

Conceptos básicos de Javascript


JS
KEEP
CALM
AND
LEARN
JAVASCRIPT

Tipos

Javascript trabaja con **tipos dinámicos**, una variable puede ser de tipos diferentes según el valor que tenga en cada momento.

Javascript

```
var x;      // undefined
x=10;       // Number
x='Hola';   //String
```

Otros

```
bool x = 'Hola' // error
```

- Numbers
- Strings
- Booleans
- Objects
- Arrays
- Functions
- Undefined
- Null

Tipos Primitivos (undefined, null)

Son tipos de datos que representan un valor simple, es decir que no son objetos.

En JS tenemos seis tipos.

- Undefined
- Null
- Boolean
- Number
- String
- Symbol

Undefined. Representa ausencia de valor. Es el valor que toman las variables declaradas a las que no se ha asignado un valor.

```
var x;      // undefined
```

Null. Representa ausencia de valor. Podemos asignarlo a una variable para representar su ausencia de valor

```
var x = null;      // null
```

Tipos primitivos (number)

Operadores:

- *
- /
- +
- -
- %

Ojo con la precedencia (*,/, %) (+,-) (Izquierda a derecha)

Números especiales:

- Infinity
- -Infinity
- NaN

Se consideran números pero no se comportan como números “normales”

Tipos primitivos (string)

Usados para representar texto. Englobados entre comillas simples o dobles

```
'Lección 1. Javascript' "Lección 1. Javascript"
```

"Preferiblemente" se usan 'single quotes'

HTML: `'<a href="link.html" target="_blank">`

JSON: `'{key:"value"}'`

Carácter de escape:

```
"Ésta es la primera línea\n Y ésta la segunda"
"<a href=\"link.html\" target=\"_blank\">
```

Tipo primitivo (boolean)

Comparadores:

- >,<,>=,<=,==,!=

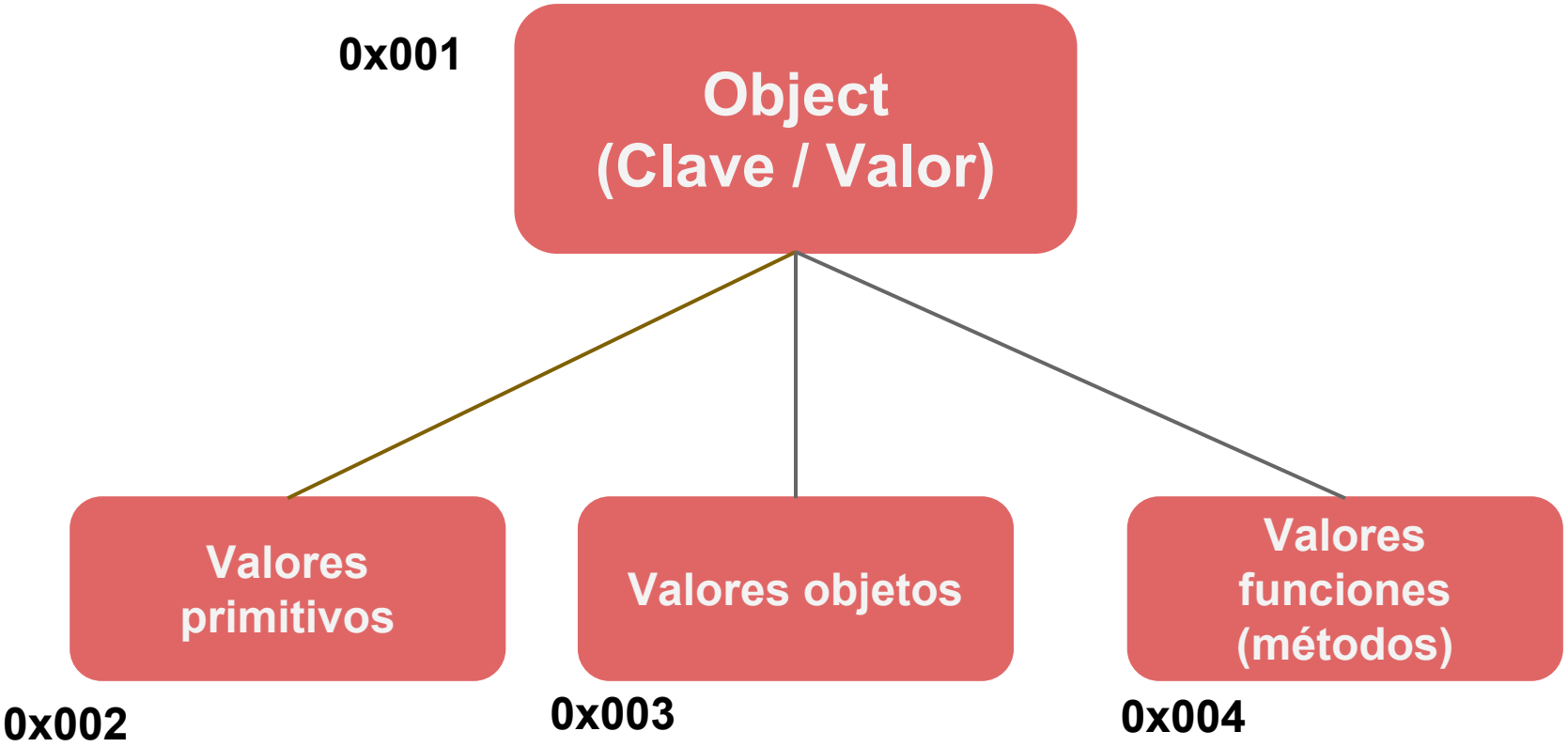
Operadores lógicos:

- &&, ||, !

Operador ternario:

- <expresion> ? <valor si true> : <valor si false>

Objects (I)


0x001
Object
(Clave / Valor)
Valores
primitivos
Valores objetos
Valores
funciones
(métodos)
0x002
0x003
0x004

Objects (II)

- En Javascript “casi todo” son objetos
- Un objeto es un mapeo entre claves y valores
- Las claves son strings y los valores pueden ser cualquier cosa (incluido otro objeto)
- Se escriben entre llaves {} y cada propiedad se escribe en formato clave:valor separado por comas (Object literal)

```
var persona = {nombre:'Luis', apellido:'Ruiz', edad: 50};
```

- Acceso: persona.nombre ó persona['nombre']
- Métodos: un método es una función almacenada como una propiedad objeto.

```
persona.darNombreCompleto()
```

Arrays

- Sirven para almacenar múltiples valores en una única variable

```
var nombreArray = [elem1, elem2, elem3, …]
```

- Pueden instanciarse también a través del constructor Array, pero por simplicidad, legibilidad y rendimiento se desaconseja su uso.

```
var nombreArray = new Array(elem1, elem2,..);
```

- Para acceder a un valor se hace accediendo a través del índice del mismo:

```
var elemento = nombreArray[0];
```

- Son un tipo especial de objeto, con sus propiedades y métodos (.length, .sort(),...)

Functions

- Las funciones en javascript son objetos de primera clase (first-class objects): son un tipo especial de objeto que puede hacer lo mismo que un objeto regular.
  - Son instancias de tipo Object
  - Podemos añadirles propiedades nuevas
  - Podemos almacenar funciones en variables
  - Podemos pasar funciones como parámetros
  - Podemos devolver funciones desde una función
- Son objetos con la capacidad adicional de que pueden ser llamados.
- Son objetos Function con sus propiedades y métodos particulares
- Una función es un bloque de código diseñado para ejecutar una tarea concreta.
- Una función se ejecuta cuando alguien la llama.
- Un método de un objeto es una propiedad función

Undefined values

- null es un tipo en sí mismo (primitivo)
- undefined es un tipo en sí mismo (primitivo)
- Undefined implica ausencia de valor “significativo” (una variable no inicializada)
- Null implica ausencia de valor (variable inicializada con el valor null)
- La diferencia entre null y undefined es considerado por algunas personas como un error en el diseño de Javascript

Conversión automática de tipos (Coerción)

- `console.log(8 * null)` ⇒ `0`
- `console.log('5' - 1)` ⇒ `4`
- `console.log('5' + 1)` ⇒ `'51'`
- `console.log('five' * 2)` ⇒ `NaN`
- `console.log(false == 0)` ⇒ `true`
- `console.log('hola' + undefined)` ⇒ `'holaundefined'`
- `console.log('hola' + null)` ⇒ `'holanull'`

== vs ===

- == compara valores, no importa el tipo
- === compara valores del mismo tipo

- `'5' == 5 , '5' === '5'` ⇒ `true, true`
- `null==undefined, null==0` ⇒ `true, false`
- `null===undefined` ⇒ `false`
- `0==false, ''==false` ⇒ `true, true`
- `0===false, ''===false` ⇒ `false, false`

Scope

- Variables Locales

```
//marcaMoto no disponible

function miMoto() {
    var marcaMoto = 'BMW';
    // marcaMoto disponible
}
```

- Variables Globales

```
var marcaMoto = 'BMW';
//marcaMoto disponible

function miMoto() {
    // marcaMoto disponible
}
```

- El ciclo de vida de una variable comienza cuando se declara
- Las variables locales se eliminan cuando termina la función
- Las variables globales se eliminan al borrar la página
- Los parámetros de una función se comportan como variables locales dentro de la misma.

Strict Mode

- `'use strict'`
- Javascript en modo “estricto”. Disponible desde ECMA 5
- No permite utilizar variables sin declarar
- No permite definir una propiedad más de una vez en el mismo objeto.
- No permite duplicar el nombre de un parámetro en una función.
- No permite escribir en propiedades de solo lectura.
- No permite el uso de palabras reservadas como nombre de variables (eval, arguments,...)
- En definitiva permite escribir código Javascript más “seguro”

Buenas prácticas

- Evitar el uso de variables globales
- Evitar el uso de new ({}, ‘’, [])

```
var v1 = ‘Michael’;
var v2 = new String(‘Michael’);
(v1 === v2) //false, porque v1 es string y v2 es un objeto
```

- Evitar comparaciones ==
- Evitar el uso de eval()

```
<script>
  eval(‘alert(“URL query:’ + unescape(document.location.search) + ‘”);’);
<script>
http://ejemplo.com?v1=10
http://ejemplo.com?hola%22;alert(document.cookie+%22

alert("URL query:hola"); alert(document.cookie+"");
```

Prototypes. ¿Qué son?

- En desarrollo de software siempre intentamos reutilizar el máximo código posible. Una de las principales formas es a través de la herencia, extendiendo las funcionalidades de un objeto con los de otro. En JS la herencia se implementa con un simple mecanismo llamado *prototyping*
- La idea de *prototyping* es sencilla, consiste en indicar un objeto en el que delegar la búsqueda de una propiedad en caso de que el objeto en sí mismo no disponga de ella.
- Es importante destacar que un objeto puede tener un prototipo, y su prototipo además puede tener otro prototipo, formando lo que se conoce como la cadena de prototipos (prototype chain)

Prototypes. ¿Qué son?

```
var x = {a:'a', b: 'b'};
var y = {c:'c', d: 'd'};
console.log(x.a);
console.log(x.c);
Object.setPrototypeOf(x,y);
console.log(x.c);
var z = {e: 'e', f: 'f'};
Object.setPrototypeOf(y,z);
console.log(x.e);
```

Prototypes y construcción de objetos

- La forma más sencilla de construir un objeto es mediante la notación object literal.
  - Ventajas: muy cómodo y sencillo
  - Desventaja: poco propenso a la reutilización de código en caso de querer crear múltiples instancias del mismo tipo de objeto
- JS permite un mecanismo para solventar esto
- Toda función en JS tiene un objeto prototype que es automáticamente asignado como el prototype de todos los objetos creados mediante dicha función (a través del operador new).
- Podemos modificar/enriquecer ese objeto prototype con propiedades y funcionalidades.

Prototypes. ¿Qué son?

```
function Curso() {}
Curso.prototype.numLecciones = function() {
  return 10;
}
var curso1 = Curso();
console.log(curso1);
var curso2 = new Curso();
console.log(curso2);
console.log(curso2.numLecciones);
console.log(curso2.numLecciones());
```

Object Prototypes (III)

- Toda clase JS tiene un prototype
- Objetos nuevos creados via {} o Object() heredan de Object.prototype ({}) (último elemento de la cadena de herencia)
- Para crear una clase reutilizable lo hacemos mediante el Constructror Pattern:

```
function Persona(nombre, apellido, edad) {
    this.nombre = nombre;
    this.apellido = apellido;
    this.edad = edad;
}
```

- Con un constructor, mediante new podemos crear objetos del mismo prototype

```
var miHermano = new Persona('Carlos', 'Pérez', 32);
```

Object Prototypes (IV)

- Podemos añadir propiedades al objeto:

```
miHermano.colorOjos = 'marrón';
```

- Podemos añadir métodos al objeto:

```
miHermano.nombreCompleto = function() {
    return this.nombre + ' ' + this.apellido;
}
```

- Podemos añadir propiedades al prototype.. a través de su constructor

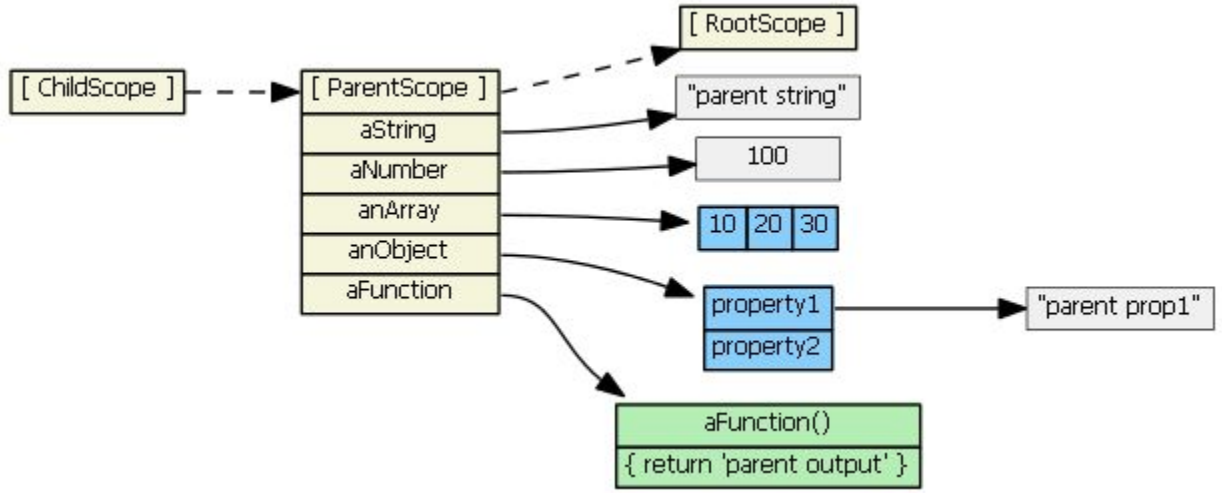
```
function Persona(nombre, apellido, edad, colorOjos) {
    ...
    this.colorOjos = colorOjos;
}
```

- Del mismo modo podemos añadir métodos

```
Persona.prototype.nombreCompleto = function()....
```

JS Prototypal Inheritance (I)

- La herencia en JS es distinta a la herencia clásica utilizada en tecnologías Java o .NET

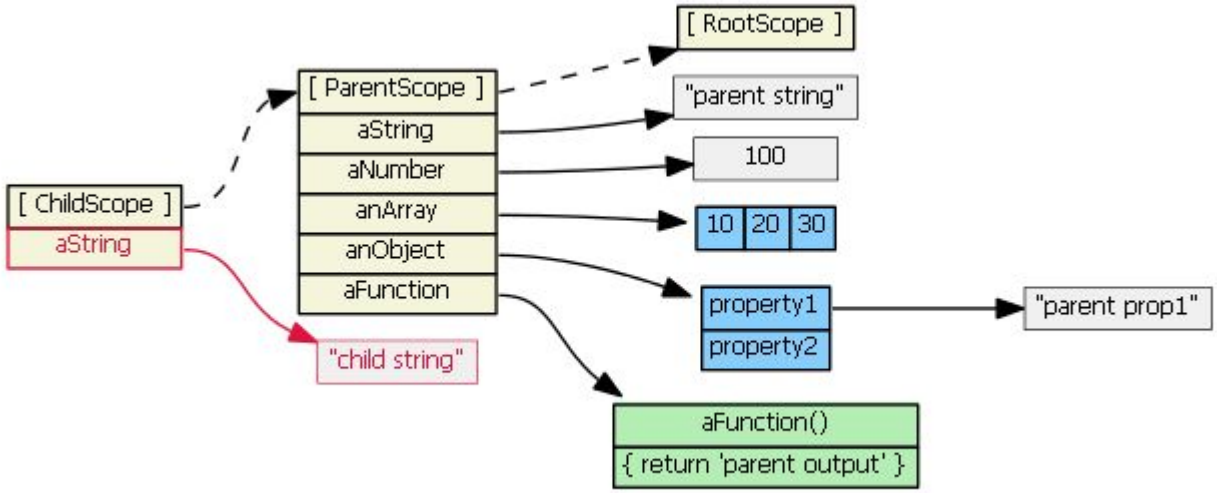



```
childScope.aString === 'parent string'
childScope.anArray[1] === 20
childScope.anObject.property1 === 'parent prop1'
childScope.aFunction() === 'parent output'
```

JS Prototypal Inheritance (II)

- Si ahora hacemos:

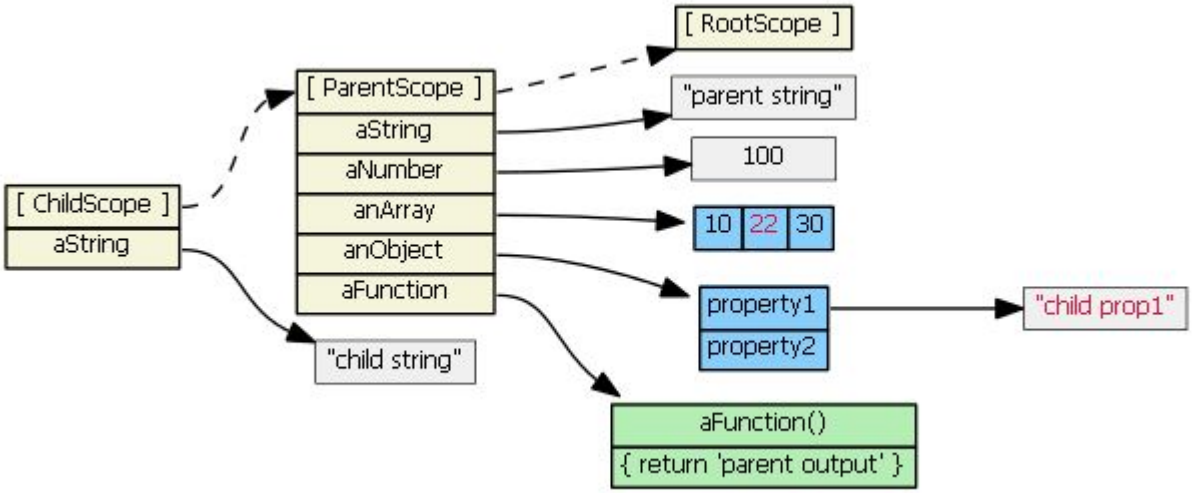
```
childScope.aString = 'child string';
```



JS Prototypal Inheritance (III)

- Si ahora hacemos:

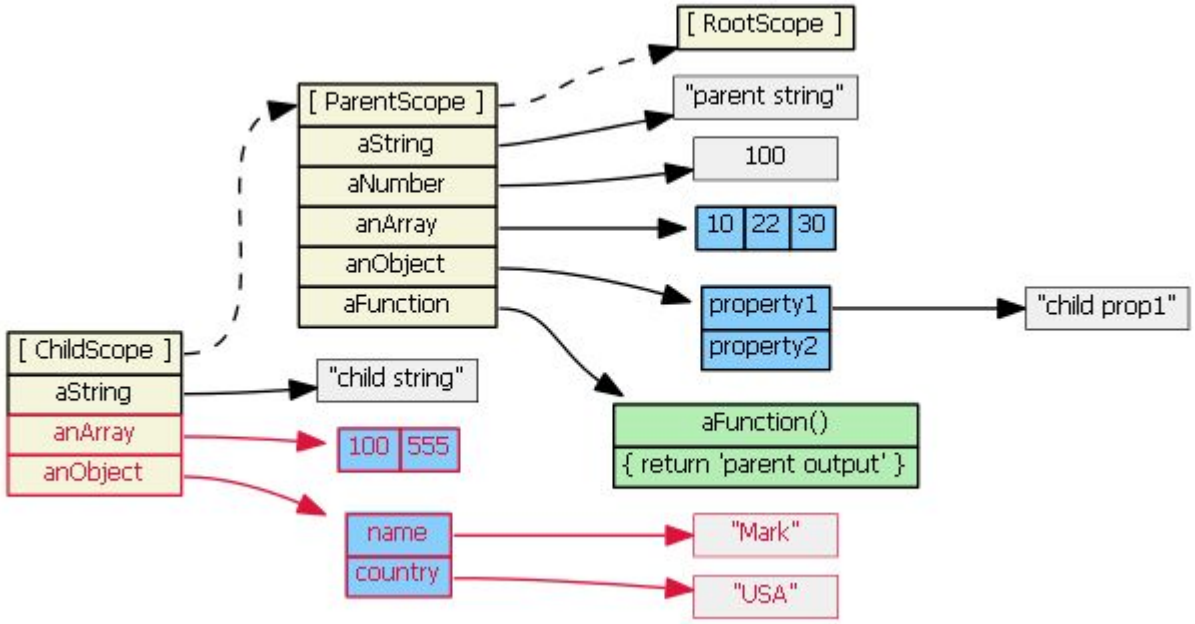
```
childScope.anArray[1] = '22';
childScope.anObject.property1 = 'child prop1';
```



JS Prototypal Inheritance (IV)

- Si ahora hacemos:

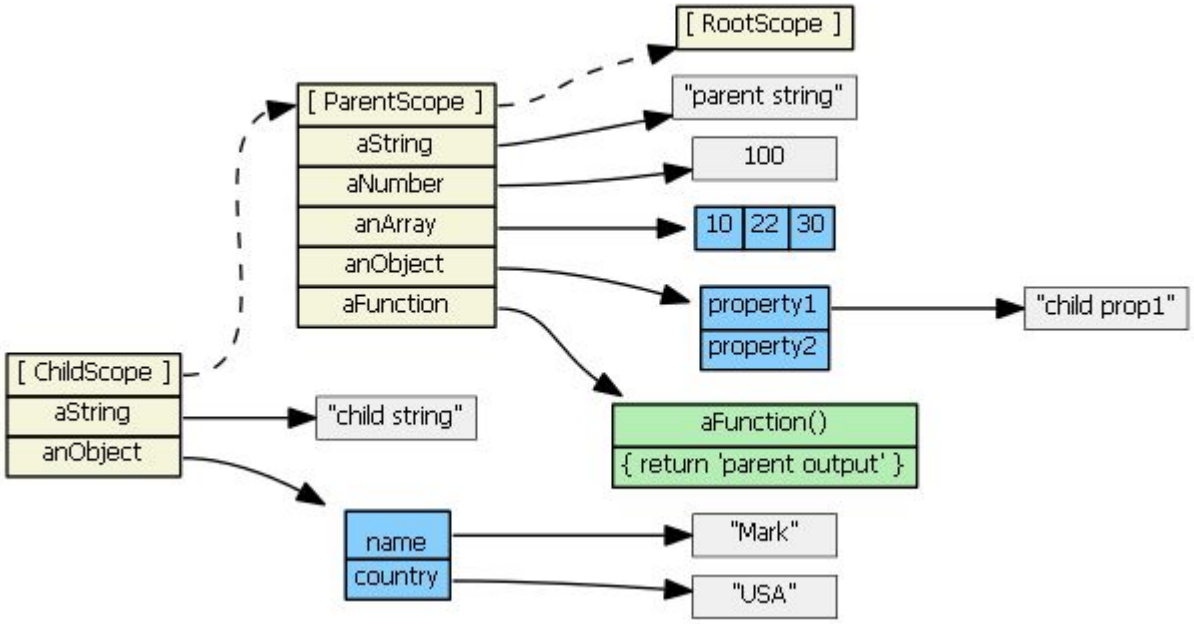
```
childScope.anArray = [100, 555];
childScope.anObject = {name: 'Mark', country: 'USA'}
```



JS Prototypal Inheritance (V)

- Si ahora hacemos:

```
delete childScope.anArray;
childScope.anArray[1] === 22 // true
```



Closures (I)

- Recordatorio: tenemos en JS variables locales y globales

```
function miFuncion() {
     var a = 4;
     return a * a;
}
```

```
var a = 4;
function miFuncion() {
     return a * a;
}
```

- ¿Variables privadas?

```
var contador = 0;
function incrementar() {
     contador += 1;
}
incrementar();
incrementar();
incrementar();
// contador => 3
```

```
function incrementar() {
     var contador = 0;
     contador += 1;
}
incrementar();
incrementar();
incrementar();
// contador => 1
```

Closures (II).

- Funciones anidadas

```
function incrementar() {
    var contador = 0;
    function mas() {
        contador += 1;
    }
    mas();
    return contador;
}
```

Closures (III).

- Closure

```
function incrementador() {
    var contador = 0;
    return function() {
        return contador +=1;
    }
};
var incrementar = incrementador();
incrementar();
incrementar();
incrementar();

//El contador es 3
```

Closures (IV).

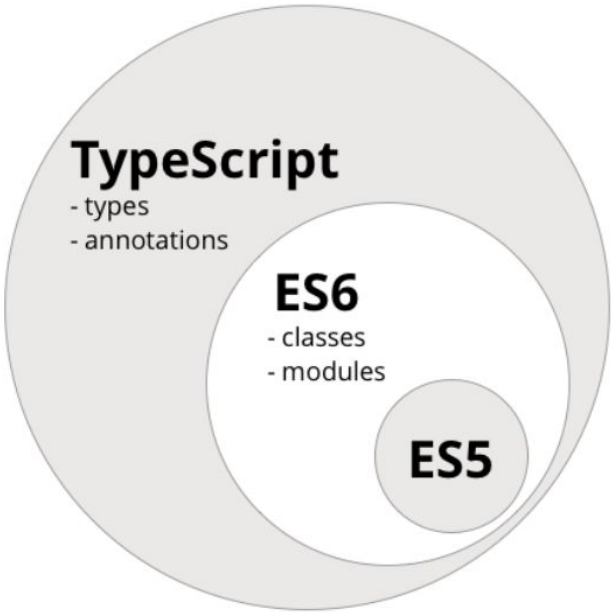
```
function cardID() {
    var cardID = 100;
    return {
        getID: function() {
            return cardID;
        },
        setID: function(newID) {
            cardID = newID;
        }
    }
}
var myCardID = cardID();
myCardID.getID(); //100
myCardID.setID(342);
myCardID.getID(); // 342
```

1.

Typescript

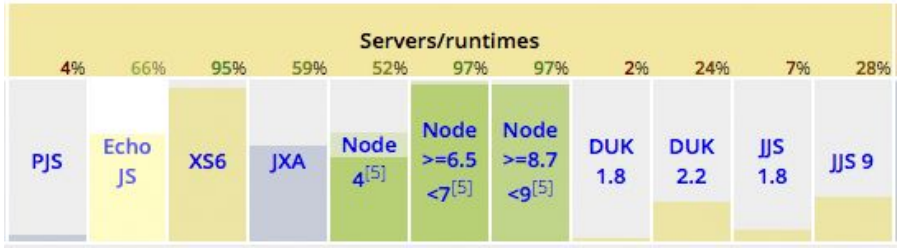
¿Que es TypeScript?

- Angular está construido sobre TypeScript
- TypeScript **no** es un lenguaje nuevo construido desde cero, es un superconjunto de ES6. Si escribimos código ES6 es válido y compatible con TypeScript.



ES6

- Compatibilidad:



ES6, TypeScript

- La compatibilidad de ES6 aunque ya es bastante alta no es suficiente.
- Typescript no es soportado por los navegadores
- La solución son los transpiladores.
- El transpilador de TypeScript transforma nuestro código en código ES5 perfectamente legible por todos los navegadores.
- También hay transpiladores de ES6, como traceur (Google) y babel (JS community)
- TypeScript se ha creado como una colaboración oficial entre Microsoft y Google, lo que garantiza que será ampliamente soportado en el tiempo.
- **NO** es obligatorio usar TypeScript en Angular, podemos escribir código ES5 directamente aunque estaríamos perdiendo características que facilitan el desarrollo.

TypeScript - Características

- Es un lenguaje **tipado**, lo que nos ayudará a **prevenir bugs** y a crear un código **más legible**.
- Se pueden crear **clases** (de forma más sencilla), lo que nos facilita una programación orientada a objetos.
- Dispone de **decoradores** que nos permitirán modificar o anotar una clase o un método.
- **Importación** de módulos
- Utilidades del lenguaje, principalmente las siguientes:
  - Fat arrow functions
  - Template string
- Todo esto transpilando a ES5 permitiendo generar código compatible con la práctica totalidad de los navegadores.

TypeScript - Tipos

- Es la mayor mejora, y de hecho es incluso lo que le da nombre al lenguaje.
- Javascript es un lenguaje de tipado dinámico mientras que TypeScript es fuertemente tipado:
  - Nos ayudará al **escribir** código ya que prevendrá bugs
  - Nos ayudará al **leer** código ya que clarificará las intenciones de los programadores
- Los tipos primitivos son los mismos que en Javascript: string, number, boolean,..

```
var fullName:string = 'Raúl Novoa';
```

- Si tratamos de asignar la variable anterior a un valor de otro tipo obtendremos un error.
- También utilizaremos los tipos como parámetros o valores de retorno de las funciones.

```
function greetText(name:string) : string {
    return 'Hello' + name;
}
```

TypeScript - Tipos (Ejemplos)

```
//primitivos
var fullName: string = 'Luis Pérez';
var age: number = 31;
var married: boolean = true;

//arrays
var jobs: Array<string> = ['IBM', 'Microsoft', 'Google'];
var jobs: string[] = ['Apple', 'Dell', 'HP'];
```

TypeScript - Tipos (Ejemplos)

```
//enums (enumeraciones numéricas)
enum Role {Employee, Manager, Admin};
var role: Role = Role.Employee;

//enums (valores explícitos)
enum Role {Employee=3, Manager, Admin}
var role: Role = Role.Employee;
//La relación es bidireccional
var a = Role.Employee; //3
var b = Role[Role.Employee]; // 'Employee'
```

TypeScript - Tipos (Ejemplos)

```
//vuelta al JS clásico :-)
var something: any = 'i am a string';
something = true;
something = [1, 3, 'hello'];

//void
function setName(name:string): void {
    this.fullName = name;
}

// Podemos usarlo en variables, pero no sirve para nada
var unusable:void;
```

TypeScript - Tipos (Ejemplos)

```
//tuplas
var x: [string, number];
x = ['hello', 10]; // OK
x = [10, 'hello']; // Error
x[0].susbstr(1) // ello
x[1] * 3 // 30
// Los subsiguientes indices tendrían que ser de uno de
// los anteriores string | number
x[2] = 4;
x[3] = 5;
x[4] = ‘adios’;
```

TypeScript - Declaración de variables

- Podemos declarar variables de 3 formas:
  - var
  - let
  - const
- let es similar a var, pero introduce algunas restricciones para evitar errores típicos en la programación con JS
- const es exactamente igual que let salvo que no permite reasignar el valor de una variable, es una constante.
- La diferencia principal entre var y let son sus reglas de ámbito (scope rules). Una variable declarada con var puede accederse en cualquier parte de la función, módulo o ámbito global en el que ha sido declarada (**function-scoping**), mientras que una variable declarada con let su ámbito se reduce al bloque en el que ha sido declarada (**block-scoping**)
- Otra diferencia significativa es que una variable declarada con var podemos definirla las veces que queramos, mientras que con let solamente podemos definirla una vez

TypeScript - var vs let

```
function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        var x = 10;
    }
    return x;
}

f(true);  // returns '10'
f(false); // returns 'undefined'
```

```
function f(shouldInitialize: boolean) {
    if (shouldInitialize) {
        let x = 10;
    }
    return x;
}
```

Cannot find name 'x'.

TypeScript - var vs let

```
function sumMatrix(matrix: number[][]) {
    var sum = 0;
    for (var i = 0; i < matrix.length; i++) {
        var currentRow = matrix[i];
        for (var i = 0; i < currentRow.length; i++) {
            sum += currentRow[i];
        }
    }

    return sum;
}
```

- El for interno puede sobreescribir el valor de la variable de iteración i
- Con let funcionaría correctamente ya que cada variable i sería distinta ya que se han declarado en bloques diferentes

TypeScript - var vs let

```
for (var i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 1000);
}
```

```
for (let i = 0; i < 10; i++) {
    setTimeout(function() { console.log(i); }, 1000);
}
```

TypeScript - Classes

- En ES5 la programación orientada a objetos la conseguimos a través de los prototipos.
- En ES6 finalmente se ha introducido el concepto de clase.
- Para definir una clase utilizaremos la palabra reservada **class**

```
class Person {
    firstName: string;
    lastName: string;
    age: number;
}
```

TypeScript - Classes

- Podemos añadir métodos a nuestras clases

```
class Person {
    firstName: string;
    lastName: string;
    age: number;

    greet() {
        console.log('Hello', this.firstName);
    }
}

let p: Person = new Person();
p.firstName = 'Felipe';
p.greet();  // Hello Felipe
```

TypeScript - Constructores

- El constructor es un método especial que se ejecuta cuando se crea una nueva instancia de la clase, y por tanto donde se realiza la inicialización del objeto.
- El constructor siempre se llama **constructor()** y opcionalmente puede recoger parámetros, pero no se le indica que devuelva ningún tipo ya que lo que va a devolver siempre es el propio objeto creado.
- Si no indicamos constructor es semejante a dejarlo vacío.

```
class Person {
    firstName: string;
    lastName: string;
    age: number;

    constructor() {
    }
}
```

TypeScript - Constructores

- En TypeScript solo podemos tener un constructor por clase.

```
class Person {
    firstName: string;
    lastName: string;
    age: number;

    constructor(firstName:string, lastName:string, age: number) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    greet() {
        console.log('Hello', this.firstName);
    }
}
```

TypeScript - Herencia

- La herencia es la forma de indicar que una clase hereda el comportamiento de una clase padre, de forma que podamos aumentar, sobreescribir o modificar el comportamiento en una nueva clase.
- Para expresar herencia usamos la palabra reservada **extends**

```
class Report {
    data: Array<string>;

    constructor(data: Array<string>) {
        this.data = data;
    }

    run() {
        this.data.forEach(function(line) { console.log(line);});
    }
}
```

TypeScript - Herencia

```
class TabbedReport extends Report {
    headers: Array<string>;

    constructor(headers: string[], values: string[]) {
        super(values);
        this.headers = headers;
    }

    run() {
        console.log(this.headers);
        super.run();
    }
}
let myReport:TabbedReport = new TabbedReport(['C1','C2'],['V1','V2']);
myReport.run();
```

TypeScript - Funciones Fat arrow (=>)

- Es una notación acortada para escribir funciones

```
function sum(a,b) {
    return a + b;
}
```

```
let sum = (a,b) => {
    return a + b;
}
```

- Muy útil al pasar funciones como parámetros:

```
const evens = [2,4,6,8];
let odds = evens.map(function(v) {
    v = v + 1;
});
```

```
const evens = [2,4,6,8];
let odds = evens.map(v => v + 1);
```

TypeScript - Funciones Fat arrow (=>)

- Un detalle importante a tener en cuenta en las funciones fat arrow es que comparten el mismo **this** que el bloque que las contiene. Es muy importante ya que difiere del comportamiento normal de las funciones JS. Generalmente cuando creas una **function** en JS, dicha función tiene su propio **this**

```
let nate = {
    name: "Nate",
    guitars: ["Gibson", "Martin", "Taylor"],
    printGuitars: function() {
        let self = this;
        this.guitars.forEach(function(g) {
            // this.name is undefined so we have to use self.name
            console.log(self.name + " plays a " + g);
        });
    }
}
```

TypeScript - Funciones Fat arrow (=>)

```
let nate = {
    name: "Nate",
    guitars: ["Gibson", "Martin", "Taylor"],
    printGuitars: function() {
        this.guitars.forEach((g) => {
            console.log(this.name + " plays a " + g);
        });
    }
}
```

- ¿Lo comprobamos en playground?

http://www.typescriptlang.org/play/

TypeScript - Template strings

- Se introducen en ES6 y aportan dos grandes mejoras:
  - Podemos usar variables en una cadena sin necesidad de concatenar con el operador +
  - Nos permiten crear cadenas multilínea
- **Importante**: la cadena debemos definirla con **backticks** en vez de con comillas simples o dobles.

```
let firstName = 'Raúl';
let lastName = 'Novoa';
const greeting = `Hello ${firstName} ${lastName}`;
```

```
let template = `
<div>
    <h1>Hello</h1>
    <p>This is a great website</p>
</div>
`;
```

TypeScript - Interfaces

```
interface ClockInterface {
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }

    constructor(){
    }
}
```

TypeScript - Modificadores de acceso

- public, protected, private

```
class Person {
      public name: string;
      protected age: number;
      constructor(name: string, age: number) {
            this.name = name;
            this.age = age;
      }
}
class Employee extends Person {
      private department: string;
      constructor (name: string, age: number, department: string) {
            super(name, age);
            this.department = department;
      }
      public toString() {
            return `Hello my name is ${this.name} and I work in
                  ${this.department}`;
      }
}
```

TypeScript - Getters / Setters

```
class Employee {
      private _fullName: string;
      get fullName(): string {
            return this._fullName;
      }
      set fullName(newName: string): string {
            if (securityCheck()) {
                  this._fullName = newName;
            } else {
                  console.log("Unauthorized");
            }
      }
}

let employee: new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
      console.log(employee.fullName);
}
```

TypeScript - Clases abstractas

```
abstract class Department {
      constructor(public name: string) {
      }
      printName() {
            console.log(`Department name: ${this.name});
      }
      abstract printMeeting();
}
class AccountDepartment extends Department {
      constructor() {
            super('Accounting and auditing');
      }
      printMeeting() {
            console.log('The accounting department meeting each monday at 10am.');
      }
      generateReports() {
            console.log('Generating accounting reports…');
      }
}
```

TypeScript - Iteradores

```
let items = [4, 5, 6];

for (let i in items) {
    console.log(i);  // 0, 1, 2
}

for (let i of items){
    console.log(i);  // 4, 5, 6
}
```

TypeScript - Módulos

```
//PhoneNumberValidator.ts
// Phone number validator
export class PhoneNumberValidator {
    isAcceptable(s: string) {
        return phoneNumberRegExp.test(s);
    }
}
```

```
//other.ts
import { PhoneNumberValidator } from "./PhoneNumberValidator";

let myValidator = new PhoneNumberValidator();
```

Ejercicio

- Crear una clase abstracta Animal
  - Propiedades tipo/nombre del animal, color y número de patas.
  - Método que describa (.desc o .toString) al animal
  - Método abstracto que emita el sonido del animal
- Clase Perro
- Clase Gato
- Clase Serpiente

2.

¿Cómo funciona Angular?

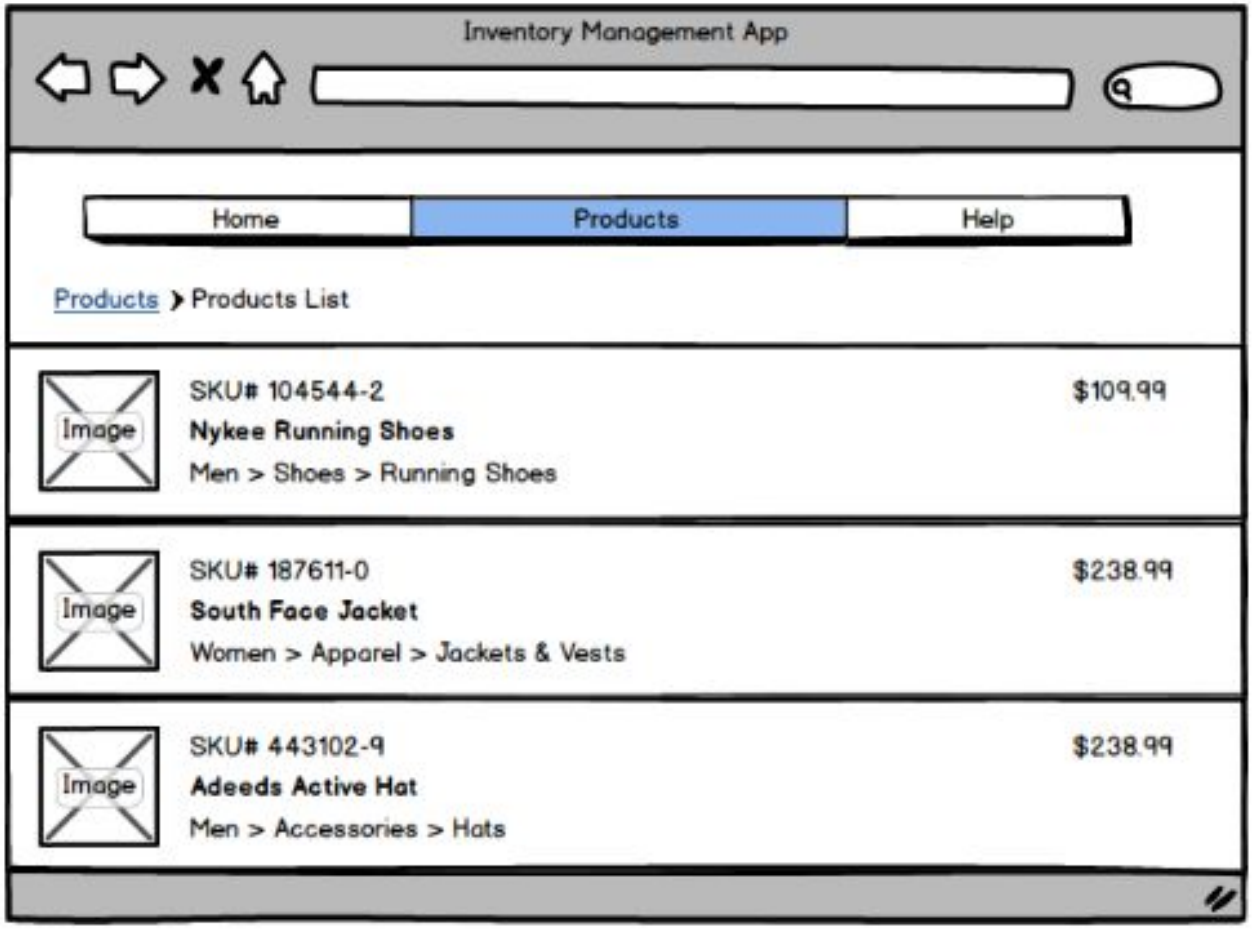
¿Como funciona Angular?

- Vamos a echar un vistazo al funcionamiento general de Angular, sin profundizar en detalle por el momento
- El primer gran concepto es que una aplicación Angular está construida a base de **Componentes**
- ¿Qué es un componente? Una forma de crear nuevas etiquetas en nuestro HTML
- A los que vengáis de AngularJS 1.x es equivalente a las directivas (aunque en Angular también hay directivas). Sin embargo los componentes tienen muchas ventajas sobre las directivas como iremos viendo.

Aplicación

- Una aplicación Angular no es nada más que un árbol de componentes.
- En el nodo raíz del árbol, el primer componente es la aplicación en sí.
- Una de las grandes características de los componentes es que podemos anidarlos. Es decir podemos construir un componente mayores basados en pequeños componentes.
- Al estar estructurados en un árbol, cuando cada componente se renderiza recursivamente renderiza sus componentes hijos

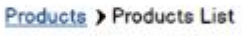
Aplicación - Ejemplo



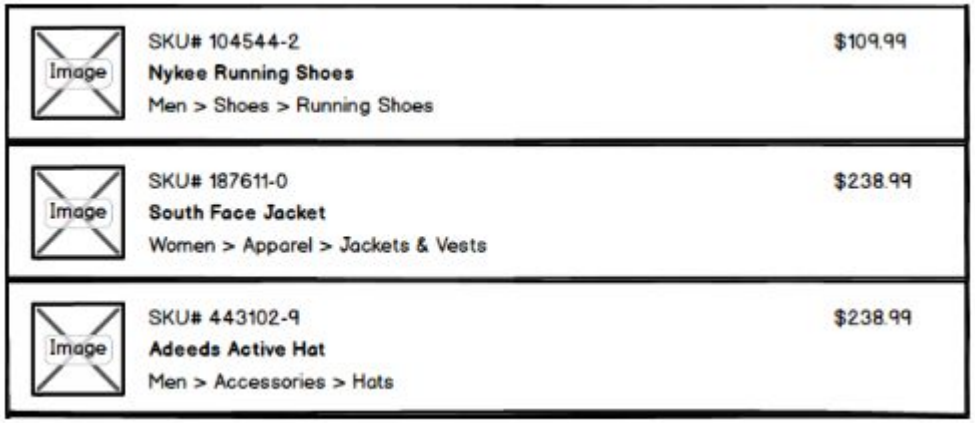
Aplicación - Ejemplo

- Dado este mockup lo primero que tendríamos que hacer es dividirlo en componentes:
  - Componente de navegación

  - Componente camino de migas



  - Componente listado de productos



Aplicación - Ejemplo

- El listado de productos lo podemos descomponer en componentes más pequeños:
  - Componente imagen del producto

  - Componente departamento del producto

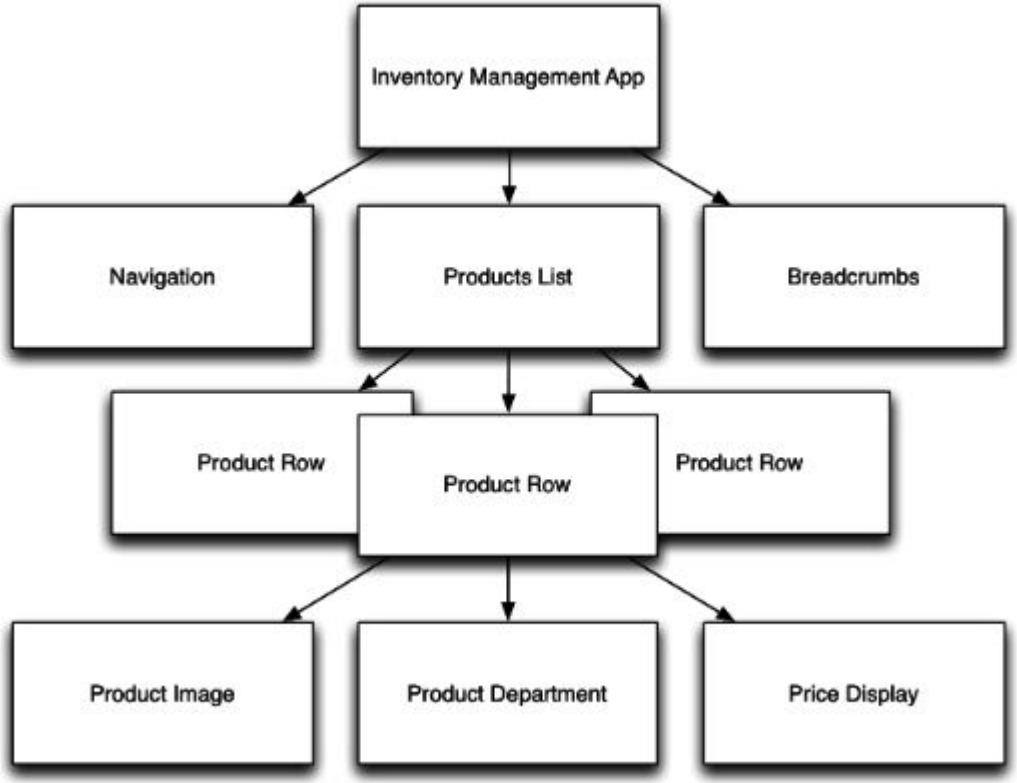
Men > Shoes > Running Shoes

  - Componente precio. Imaginando por ejemplo que el precio se puede personalizar si el usuario está logado para incluir descuentos, gastos de envío, etc

$109.99

Aplicación - Ejemplo

- Haciendo un esquema final quedaría nuestra aplicación así:



Aplicación - Ejemplo

- Una de las conceptos importantes en Angular es que no nos obliga a utilizar un librería de modelo de datos en particular, es flexible para soportar distintos tipos de modelos (y de arquitectura de datos!)
- De momento trabajaremos con clases planas.

```
export class Product {
  constructor(
    public sku: string,
    public name: string,
    public imageUrl: string,
    public department: string[],
    public price: number) {
  }
}
```

- Clase Producto con 5 argumentos
- Los 5 datos son públicos (pueden ser public, private, protected)
- Como vemos no tiene ninguna dependencia de Angular, es un simple objeto JS (ES6)

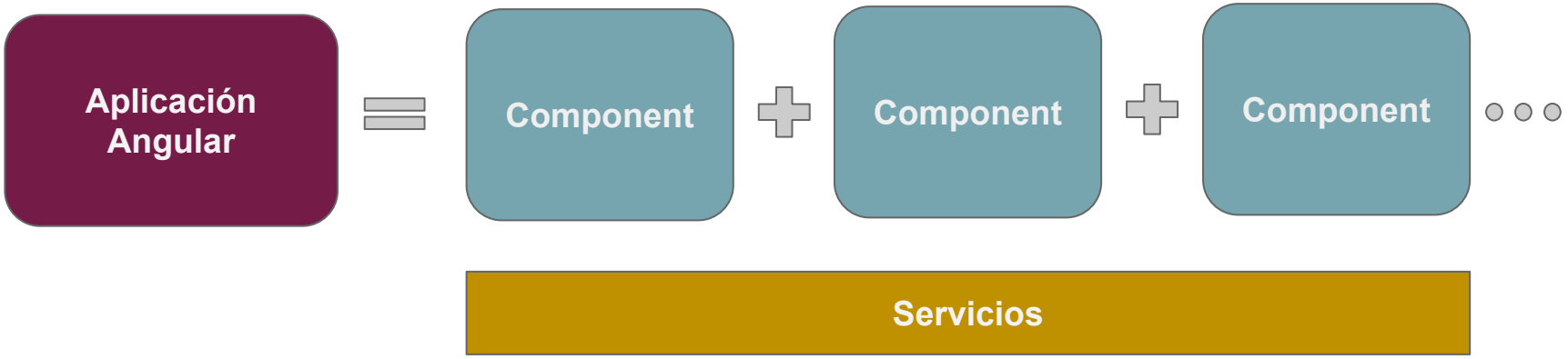
Aplicación - Componente principal

- Una aplicación angular tiene que tener un componente principal (solo uno).

```
<body>
  <app-root>Loading….</app-root>
</body>
```

  - En ese componente principal es donde se renderizará nuestra app.
  - El texto “Loading…” se mostrará durante el arranque de nuestra aplicación. Podemos poner ahí lo que estimemos oportuno (spinner, barra de progreso, ..)
  - En el index.html podemos poner tantos elementos principales como queramos, pero cada uno de ellos será una app angular diferente (se pueden comunicar)

Angular


Aplicación Angular
Component
Component
Component
Servicios

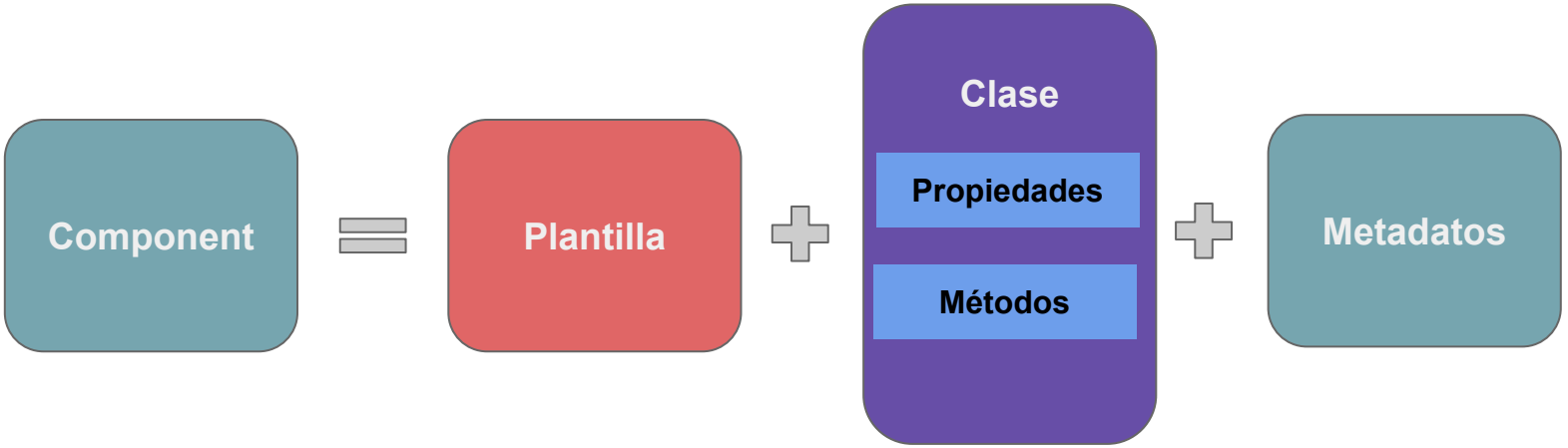
3.

Componentes

Componentes

- Elemento principal de una aplicación Angular
- El “objeto” aplicación en sí es simplemente el componente raíz (diferencia con ng-app)
- Cuando diseñamos nuestra aplicación Angular lo primero es dividir en componentes
- Cada componente tiene 3 partes:
  - Un decorador (metadatos)
  - Una vista (template)
  - Un controlador (clase)

Angular


Component
=
Plantilla
+
Clase
Propiedades
Métodos
+
Metadatos

Hello world!

```
> ng new hello-world
> cd hello-world
> ng serve
```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  // app logic here
  title = 'Hello world!';
}
```

Componentes

- El **decorador** es: @Component. Añade metadatos a la clase que le sigue, en este caso AppComponent
  - selector: le indica a angular el elemento html que lo identifica
  - template/templateUrl: define la **vista**
- El **controlador** lo define una clase, en este caso la clase AppComponent

Decorador - selector

- El decorador @Component sirve para configurar nuestro componente. Sirve para definir como “el mundo exterior” va a interactuar con él
- Hay numerosas opciones que permiten configurar un componente, de momento nos vamos a centrar en las principales.
- Selector: Indica el nombre con el que nos referiremos a nuestro componente desde el HTML. Podemos utilizarlo como un elemento o como un atributo:

```
<app-root></app-root>

<div app-root></div>
```

Decorador - template

- template: nos permitirá indicar el contenido html de nuestro componente

```
@Component({
  selector: 'app-root',
  template: `
    <h1>
        {{title}}
    </h1>
  `
})
```

- templateUrl: nos permitirá indicar el contenido html de nuestro componente referenciando a un archivo

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

Componentes - Modelo de datos

- Ejercicio: añadir un modelo de datos a nuestro proyecto hello-world, instanciarlo en el controlador y visualizarlo en la vista

```
export class Person {
  constructor(
    public fullName: string,
    public lastName: string,
    public age: number,
    public gender: string) {
  }
}
```

Componentes - Inputs y Outputs

- A nuestros componentes le podremos pasar parámetros de entrada (inputs), y podremos gestionar eventos de salida (ouputs).
- Podemos decir que:
  - Los datos suponen un flujo de entrada mediante los input bindings.
  - Los eventos suponen un flujo de salida mediante los ouput bindings.
- Los input bindings se representan con corchetes []
- Los output bindings se representan con paréntesis ()
- Ejemplo:

```
<products-list
    [productList]="products"
    (onProductSelected)="productWasSelected($event)">
  </products-list>
```

- La parte izquierda representa el identificador del input/output, mientras que la parte derecha representa la expresión a evaluar en nuestro componente (Ej: this.products en AppComponent)

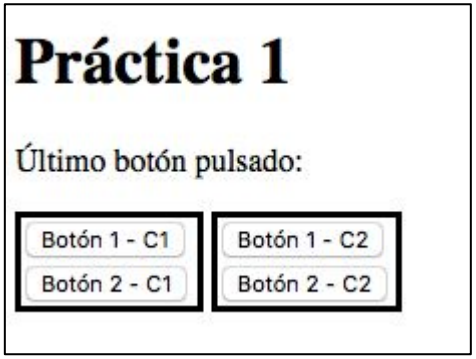
Componentes - Inputs y Outputs

- Echemos un ojo a nuestra aplicación ejemplo de inventario para comprobar los flujos de entrada y de salida

Ejercicio - Practica 1

- Aplicación con 2 componentes
  - Componente aplicación (app-root)
  - Componente con dos botones (p1-selector)
- La aplicación deberá instanciar dos veces el componente p1-selector. El texto de los botones lo indicará el componente app-root (el de los 4), y el componente app-root debe avisar cada vez que alguno de los botones se pulse.

```
> ng new practica1
> cd practica1
> ng g component p1-selector
```



Componentes - Eventos

- Hemos visto como para trabajar con outputs nos basamos en EventEmitter, ¿que son?
- Un EventEmitter es un objeto que mantiene una lista de subscriptores y publica eventos para ellos. Es una ayuda para implementar el patrón del observador (Observer Pattern)

```
let ee = new EventEmitter();
ee.subscribe((name:string) => console.log(`Hello ${name}`));
ee.emit('Luis');

//-> Hello Luis
```

- Cuando adjuntamos un EventEmmiter a un @Output, Angular nos suscribe automáticamente a él.
- Estos eventos nos permiten la comunicación entre componentes, un componente padre se suscribe a los eventos que emite(n) su componente(s) hijo

Componentes - Ciclo de vida

- Los componentes tienen un ciclo de vida gestionado por el framework. Angular los crea, los renderiza, crea y renderiza a sus hijos, comprueba cuando sus datos cambian, y los destruye para finalmente eliminarlos del DOM.
- Angular nos proporciona hooks donde poder interactuar con estos momentos de su ciclo de vida.
  - ngOnChanges: llamado cuando Angular detecta cambios en los datos de entrada. Nos entrega un objeto con los cambios.
  - ngOnInit: llamado después de que Angular vincule vista y el componente. Solo llamado la primera vez.
  - ngOnDestroy: llamado justo antes de destruir el componente. Deberíamos desuscribirnos de los observables y eliminar los listeners para prevenir memory leaks.
  - Hay mas (más “avanzados”)

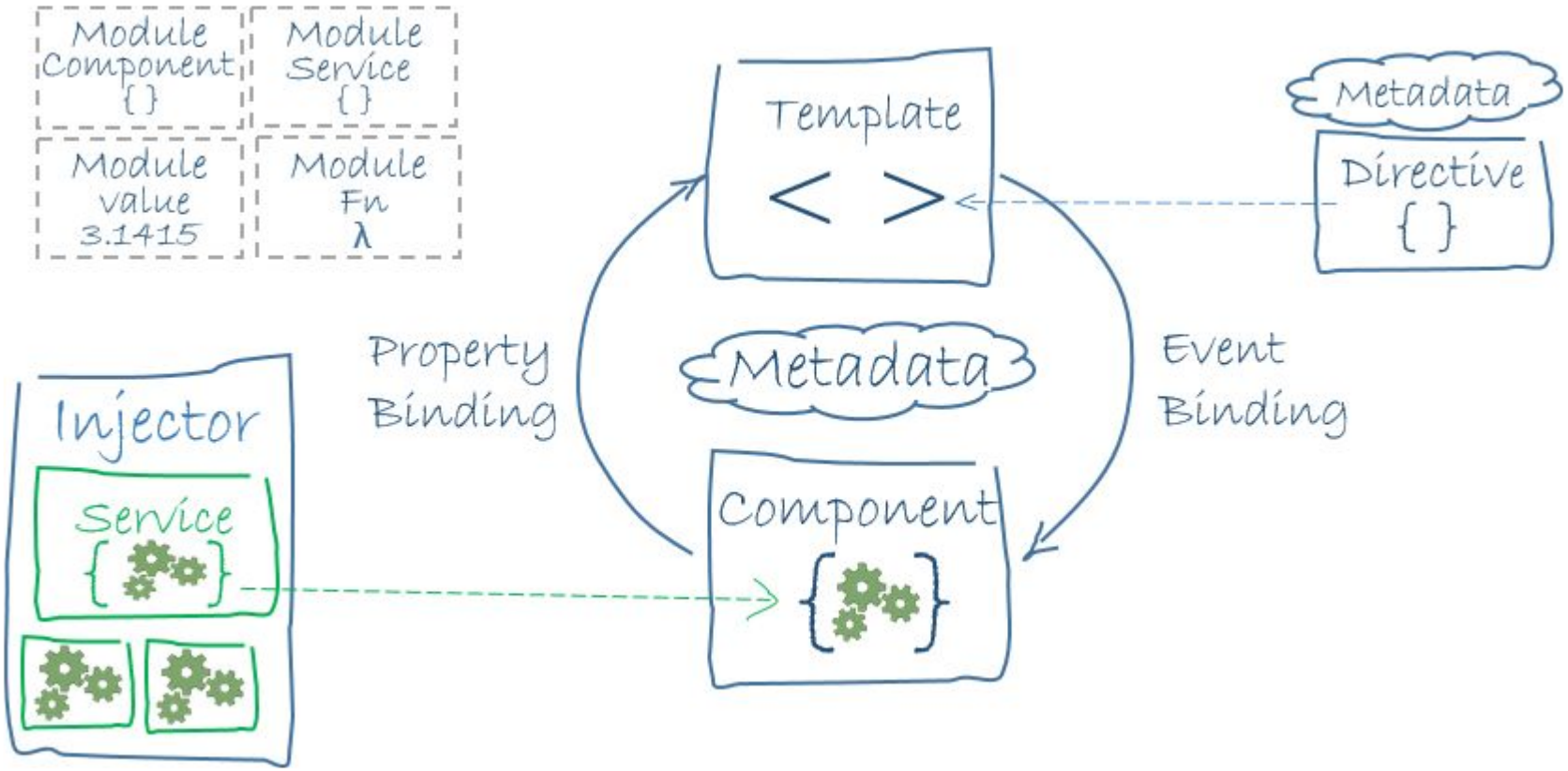
4.

Visión general

Visión general

- Angular 4 (ahora formalmente Angular, a secas) es un framework completo para construir aplicaciones en **cliente** con HTML y Javascript, es decir, con el objetivo de que el peso de la lógica y el renderizado lo lleve el propio navegador, en lugar del servidor.
- A groso modo para crear apps:
  - Componemos plantillas HTML (**templates**) con el markup de Angular
  - Escribimos **componentes** para gestionar esas plantillas y **directivas** que afectan al comportamiento de los componentes
  - Encapsulamos la lógica de la aplicación en **servicios**
  - Definimos un módulo principal que le dice a Angular qué es lo que incluye nuestra app (otros módulos), y cómo compilarlo y lanzarlo (**NgModule)**
- En el siguiente diagrama de arquitectura podemos ver como se relacionan todos estos elementos

Visión general



Módulos

- Las apps de Angular son modulares, gracias a su propio sistema de módulos llamado Angular Modules o NgModules
- Al desarrollar Angular en TypeScript utilizaremos también los módulos de ES6 para gestionar librerías de JS
- De entrada podría parecer que NgModules aporta redundancia sobre los módulos ES6, pero son necesarios para facilitar la inyección de dependencias que veremos más adelante.
- Módulos ES6: exportar / importar

```
//app/app.component.ts
export class AppComponent {
    //aquí la definición del
    //Componente
}
```

```
//app/main.ts
import {AppComponent} from './app.component';
```

Módulos de Angular

- Las librerías principales de Angular (siempre comienzan por @angular) son:
  - @angular/core
  - @angular/platform-browser
  - @angular/router
  - @angular/forms
  - @angular/http
- Para importar clases de estos módulos lo hacemos de la siguiente forma:

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

Módulos de Angular

- Un módulo de Angular es un conjunto de código dedicado a un ámbito concreto de la aplicación, o a una funcionalidad específica y se define mediante una clase decorada con **@NgModule**
- Toda aplicación de Angular tiene al menos un módulo de Angular, su módulo principal
- Decorador @NgModule. Es un decorador que recibe un objeto de metadatos que definen el módulo. Los mas importantes son:
  - declarations: las vistas que pertenecen a tu módulo. Hay 3 tipos: componentes, directivas y pipes
  - exports: conjunto de declaraciones que deben ser accesibles desde otros módulos
  - imports: otros NgModules que utilicemos en componentes de este módulo
  - providers: utilizado por DI. Pone nuestros servicios disponibles para ser inyectados a lo largo de nuestra app
  - bootstrap: define el componente inicial (solo root module)

Módulos de Angular

```
//app/app.module.ts
@NgModule({
  declarations: [
    AppComponent,
    HelloWorldComponent,
    UserItemComponent,
    UserListComponent
], imports: [
   BrowserModule,
   FormsModule,
   HttpModule
 ],
  providers: [SomeProvider],
  bootstrap: [AppComponent]
})
AppModule { }
```

Módulos de Angular - Booting

- Como hemos podido ver Angular nos proporciona un sistema de módulos que nos permite organizar nuestro código. A diferencia de AngularJS 1.x donde las directivas eran globales, en Angular debemos especificar exactamente qué componentes vamos a utilizar en nuestra app.
- Cuando creamos nuevos componentes en Angular, cuando queramos utilizarlos deberemos comprobar:
  - Que está en nuestro mismo módulo
  - Que importamos el módulo que contiene dicho componente
- **IMPORTANTE**: todo componente que escribamos debe ser **declarado** en un NgModule antes de que podamos usarlo en cualquier plantilla

```
//main.ts

import { platformBrowserDynamic } from
@angular/platform-browser-dynamic;

import { AppModule } from './app/app.module';


platformBrowserDynamic.bootstrapModule(AppModule);
```

5.

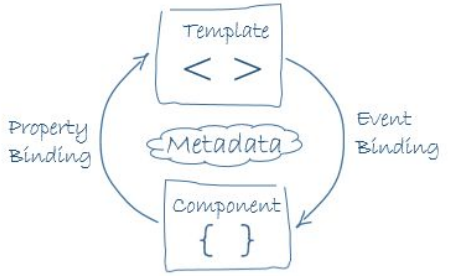
Data binding

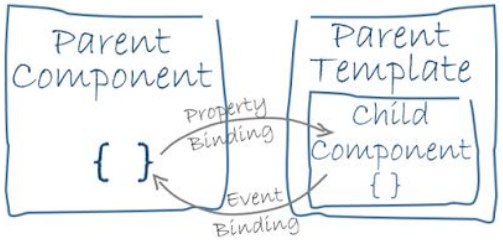
Data binding

- Uno de los principales valores de Angular es que nos abstrae de la lógica pull/push asociada a insertar y actualizar valores en el HTML y convertir las respuestas de usuario (inputs, clicks, etc) en acciones concretas. Escribir toda esa lógica a mano (lo que típicamente se hacía con JQuery) es tedioso y propenso a errores y Angular lo resuelve por nosotros gracias al data binding.
- Angular dispone de 4 formas de data binding:
  - Interpolación: `{{mivariable}}` Angular se encarga de insertar el valor de esa propiedad donde lo hemos definido, es decir, evalúa mivariable y la inserta en el DOM
  - Property binding: `[data]="mySelectedData"` Angular en este caso pasa el objeto mySelectedData del componente padre a la propiedad data del componente hijo. @Input()
  - Event binding: `(click)="selectData(data)"` Le indicamos a angular que se produzca el evento click sobre la etiqueta que lo contiene llame al método selectData, pasando como parámetro el objeto data de ese contexto. Angular mapea los eventos típicos DOM: click, dblclick, contextmenu, keypress, mouseenter, mouseleave, mousemove...

Data binding

- Y la cuarta forma:
  - Two way binding (desde/hacia el dom): combina el event binding y el property binding `<input [(ngModel)]="todo.subject">` El valor de la propiedad fluye a la caja de input, pero los cambios del usuario también fluyen de vuelta al componente.
- Angular procesa los data binding una vez por cada ciclo de eventos, desde la raíz de la aplicación siguiendo el árbol de componentes en orden de profundidad.





Ejercicio

- Hacer una calculadora con dos operandos y operaciones de suma, resta, multiplicaciòn y división.
- Mapear los operandos con two way data binging
- Mapear los eventos con event binding
- Es necesario importar FormsModule:

```
//app.module.ts
import { FormsModule } from @angular/forms;

@NgModule({
    ….
    …
    imports : {
        FormsModule
    },
    …
})
```

6.

Directivas Angular

Directivas

- Angular nos proporciona una serie de directivas con el framework, que son atributos que añadiremos a nuestros elemento HTML para dar dinamismo a nuestras aplicaciones.
- Tendremos dos tipos:
  - Directivas estructurales: nos ayudan a dar forma al HTML, añadiendo o quitando elementos, manipulándolos, etc. (ngIf, ngFor, ngSwitch..)
  - Directivas de atributos: permiten cambiar la apariencia o comportamiento de un elemento (ngStyle, ngClass...)

ngIf

- Se usa cuando queremos visualizar o no un elemento dependiendo de una condición. El elemento se añade o quita completo del DOM. La condición se determina por el resultado de evaluar la expresión que pasamos a la propia directiva. Ejemplos:

```
<div *ngIf="false"></div>
<div *ngIf="a > b"></div>
<div *ngIf="str === 'yes'"></div>
<div *nfIf="myFunc()"></div>
```

- Es el equivalente al ng-if de AngularJS. En este caso no tenemos ng-show ni ng-hide, lo haremos con ngStyle por ejemplo.

ngSwitch

- A veces tenemos que renderizar diferentes elementos según una condición. En esta situación podemos utilizar una combinación de ngIf (que podría resultar compleja según la condición) o utilizar ngSwitch.
- Ejemplo:

```
<div class="container" [ngSwitch]="myVar">
    <div *ngSwitchCase="'A'">Var is A</div>
    <div *ngSwitchCase="'B'">Var is B</div>
    <div *ngSwitchCase="'C'">Var is C</div>
    <div *ngSwitchCase="'B'">Var is B</div>     <!--podemos repetir>
    <div *ngSwithDefault>Var is someting</div>  <!--opcional-->
</div>
```

https://angular.io/docs/ts/latest/guide/structural-directives.html#!#asterisk

ngFor

- Nos permite iterar para repetir un elemento DOM dado

```
this.cities = ['Miami', 'Sao Paulo', 'New York'];

<div class="ui list">
    <div class="item" *ngFor="let city of cities"> {{city}} </div>
</div>
```

- Podemos saber el índice de la iteración también:

```
this.cities = ['Miami', 'Sao Paulo', 'New York'];

<div class="ui list">
    <div class="item" *ngFor="let city of cities; let num=index">
{{num+1}} - {{city}} </div>
</div>
```

ngNonBindable

- La usaremos cuando queramos que angular no procese una sección particular de nuestra página. Imaginemos que queremos mostrar el texto {{ soy un texto entre llaves }} en nuestra página. Al ir entre llaves angular intentará evaluarlo como expresión.

```
<span ngNonBindable>
  Muestro a continuación que {{soy un texto entre llaves}}
</span>
```

ngStyle

- Con la directive ngStyle podemos dar a un elemento DOM propiedades CSS desde una expresión de Angular. La forma mas sencilla de hacerlo es mediante [style.<cssproperty>]=”value”

```
<div [style.background-color]=”’yellow’”>
  Tengo color de fondo amarillo
</div>
```

- Otra posible forma de hacerlo:

```
<div [ngStyle]=”{color: ‘white’, ‘background-color’:’blue’}”>
  Texto blanco sobre fondo azul
</div>
```

ngClass

- Nos permite establecer clases css de forma dinámica en el DOM, mediante la evaluación de una expresión.

```
<div [ngClass]="{bordered: false}">This is never bordered</div>
<div [ngClass]="{bordered: true}">This is always bordered</div>
<div [ngClass]="{bordered: isBordered}">Sometimes bordered</div>
```

7.

Rutas

Rutas

- En un aplicación Angular podríamos llegar a hacer aplicaciones sin necesidad de rutas, haciendo uso de componentes y directivas todos sobre la página principal.
- Evidentemente esto no sería una buena práctica a la hora de generar código mantenible y legible, pero tampoco sería bueno a la hora de permitir a nuestros usuarios guardar enlaces a páginas, a la hora del SEO, etc..
- Por todo esto Angular provee de un mecanismo interno para hacer routing, de forma que en la barra del navegador el usuario ve como cambia la url pese a que en todo caso son urls locales.

Rutas

- Disponemos de 3 componentes principalmente para implementar el Routing de nuestra aplicación:
  - Routes: para definir las rutas que va a implementar nuestra aplicación web.
  - RouterOutlet: permite a Angular saber donde poner el contenido de cada ruta
  - RouterLink: directiva usada para ir a las rutas desde el HTML.
- Las rutas no están implementadas en el módulo @angular/core ya que nos permitirán utilizar módulos de enrutados que no sean el propio de angular.
- El módulo de rutas propio de angular es @angular/router y deberemos importarlo si queremos utilizarlo.

```
import {
    Routes
} from '@angular/router';
```

Routes

- Para definir las rutas de nuestra aplicación web crearemos una configuración Routes y luego haremos un RouterModule.forRoot(Routes).

```
const routes: Routes = [
    {path: '', redirectTo: 'home', pathMatch:'full'},
    {path: 'home', component: HomeComponent },
    {path: 'about', component: AboutComponent },
    {path: 'contact', component: ContactComponent },
    {path: 'contactUs', redirectTo: 'contact'}
];
```

- path especifica la url que la ruta gestionará.
- component es el que componente que se ejecutará en esa ruta
- redirectTo redirecciona una ruta a otra

Routes

- Una vez que tenemos definidas las ruta el siguiente paso es instalarlas. Tenemos que hacer dos cosas para esto:
  - Importar el módulo RouterModule
  - Instalar las rutas usando RouterModule.forRoot(routes) en la sección import de nuestro NgModule.

```
import {RouterModule, Routes } from '@angular/router';
…
const routes: Routes = [......];
@NgModule({
    declarations:[...],
    imports: [...,
        RouterModule.forRoot(routes),
        …
    ],...
})
```

RouterOutlet mediante <router-outlet>

- Cuando cambiamos de ruta solemos querer mantener cierto “layout” externo (cabecera, pie), y simplemente sustituir una zona interior.
- Para indicar a Angular donde queremos ubicar los contenidos de cada ruta usamos la directiva RouterOutlet.
- Para poder usar la directiva RouterOutlet sera imprescindible importar el RouterModule en nuestro NgModule (en todos los que lo vayamos a usar).
- Dentro de las plantillas lo utilizaremos de la siguiente forma:

```
<router-outlet></router-outlet>
```

RouterLink mediante <routerLink>

- Ahora que ya sabemos como se renderizan las rutas nos falta saber como decirle a Angular que navegue a una ruta determinada.
- Para ello utilizaremos la directiva RouterLink

```
<a [routerLink]="['/home']">Home</a>
<a [routerLink]="['/about']">About</a>
<a [routerLink]="['/contact']">Contact Us</a>
```

- En el ejemplo aplicamos la directiva sobre un anchor, pero realmente podemos aplicarlo sobre cualquier etiqueta HTML.
- Parece extraño que el enlace se lo pasemos como una cadena dentro de un array, esto es porque podemos indicarle más información en ese array, como veremos al hablar de rutas anidadas y de parámetros de rutas.

.

Rutas

- Es importante meter en el head del html <base href="/"> para indicar la ruta base de nuestros recursos.
- A veces nuestra aplicación no tendrá acceso a la sección <head> de la aplicación HTML. Por ejemplo cuando seamos una aplicación Angular embebida en un portal como Liferay.
- En estos casos podemos hacerlo programáticamente en la sección providers de nuestro NgModule

```
@NgModule({
…
…
     providers: [...,
          {provide: APP_BASE_URL, useValue: '/'}]
…
})
```

Ejercicio

- Crear una nueva aplicación con angular-cli, llamada practica-rutas
- En el componente principal poner 3 enlaces, a la home, products y faq
- Crear con angular-cli esos 3 componentes
- En la plantilla de cada uno simplemente ponemos un título
- Configurar las 3 rutas en el módulo principal
- Es importante meter en el head del html <base href="/">

Parámetros en las rutas

- En nuestras aplicaciones habitualmente querremos navegar a recursos específicos. Por ejemplo si tenemos una web de noticias con artículos, cada artículo tendrá un identificador que deberemos especificar para cargar el artículo deseado. (Ej. /articles/3)
- Obviamente no vamos a generar una ruta para cada artículo, podemos parametrizarlos, y para ello lo indicaremos de la siguiente forma:

```
/route/:param
/article/:id
```

- Y en la definición de la ruta:

```
const routes: Routes = [
    {path:'article/:id', component: ArticleComponent}
];
```

Parámetros en las rutas

- Para recuperar los parámetros necesitaremos importar ActivedRoute.

```
import {ActivatedRoute} from '@angular/router';
```

- Después, en nuestro componente, inyectaremos ActivatedRoute. (La inyección se hace en el constructor)

```
constructor(private route:ActivatedRoute) {
    route.params.subscribe(myParams => {this.id = myParams['id'];});
}
```

- Si nos fijamos route.params es un observable. Podemos extraer el valor del parámetro usando .subscribe. Dentro del objeto params estará nuestro parámetro con el identificador que le hayamos puesto en la definición de la ruta.

Ejercicio

- Añadir un nuevo componente a nuestro anterior ejercicio y asociarlo a la ruta /product/:id
- En el componente de productos añadir un nuevo enlace a un producto en concreto ficticio, por ejemplo /product/239
- En el nuevo componente, recoger ese parámetro, asociarlo a una variable de nuestro componente y visualizarlo por pantalla.

Rutas

- Más cosas a profundizar sobre las rutas:
  - Rutas anidadas
  - Multiples outlets
  - Resolves
  - Rutas protegidas

Rutas anidadas

- Las rutas anidadas consisten en incluir rutas dentro de otras rutas
- Anidando rutas podemos encapsular la funcionalidad de las rutas padres y hacer que tengan una funcionalidad específica a través de rutas hijas.
- Para anidar rutas lo que debemos hacer es anidar router outlets, de forma que cada zona de nuestra aplicación pueda tener sus propios componentes hijos.
- Es buena práctica definir las rutas dentro de cada módulo.
- Para crear rutas anidadas utilizaremos la propiedad “children” en el objeto de definición de una ruta.

```
{
    path: 'products',
    component: ProductsComponent,
    children: childRoutes
}
```

Rutas anidadas

- Veamos un ejemplo en funcionamiento.

Multiples rutas en outlets con nombre

- También tenemos la posibilidad de definir outlets con un nombre determinado, de forma que podamos configurar las pantallas como distintos bloques.

```
<router-outlet name="list"></router-outlet>
<router-outlet name="detail"></router-outlet>
```

```
{ path: 'speakers', component: SpeakersComponent, children: [
   { path: 'speakersList', component: SpeakersListComponent, outlet: 'list' },
   { path: ':id', component: DetailComponent, outlet: 'detail' }
 ] },
```

9.

Formularios

Formularios

- Los formularios son probablemente una de las partes más utilizadas en una aplicación web
- Angular nos proporciona los siguientes elementos para trabajar con formularios:
  - **FormControls**: encapsula los inputs de nuestros formularios y nos proporciona objetos con los que trabajar
  - **Validators**: nos proporcionan la capacidad para validar inputs de forma sencilla
  - **Observers**: nos permite escuchar cambios sobre nuestro formulario y responder a ellos.

FormControls y FormGroups

- Un FormControl representa un simple campo input. Es la unidad mínima de un formulario
- Encapsulan el valor del campo y nos proporcionan el estado en el que se encuentra:
  - Valid
  - Dirty
  - Error
- Por ejemplo, así es como usaríamos un FormControl en typescript:

```
let nameControl = new FormControl(“..”);
let name = nameControl.value;

nameControl.errors // -> StringMap<string,any> of errors
nameControl.dirty // -> false
nameControl.valid // -> true
...
```

FormControls y FormGroups

- Para crear formularios por tanto crearemos FormControls (y grupos de FormControls) y definiremos lógica sobre ellos.
- Para indicar en el HTML que un input es un FormControl lo haremos mediante:

```
<input type="text" [formControl]="myControl"/>
<input type="text" name="myControl" ngModel/>
```

- De esta forma crearemos un FormControl vinculado al contexto de nuestro formulario.

FormControls y FormGroups

- La gran parte de los formularios lo forman un conjunto de campos, por lo que necesitamos poder gestionar varios FormControls.
- Si queremos comprobar la validez de nuestros formularios, en vez de iterar campo a campo podemos utilizar los FormGroups.

```
let personInfo = new FormGroup({
    firstName: new FormControl("..."),
    lastName: new FormControl("..."),
    zipCode: new FormControl("...")
});
```

- Tanto FormControl como FormGroup extienden de la misma clase AbstractControl, por lo que podemos comprobar el valor y el estado de un FormGroup de la misma forma que lo hacíamos para un FormControl..

FormControls y FormGroups

```
personInfo.value; // -> {
// firstName: "...",
// lastName: "...",
// zipCode: "..."
}

personInfo.errors // -> StringMap<string,any> of errors
personInfo.dirty // -> false
personInfo.valid // -> true
...
```

- Podemos ver como al acceder al value nos entrega un objeto (clave/valor) con todos los campos del FormGroup

Nuestro primer formulario

- El primer paso será cargar el módulo FormsModule.
- FormsModule nos proporciona acceso a:
  - ngModel
  - NgForm
- NgForm es una directiva pero que funciona de forma automática, sin necesidad de que la especifiquemos como atributo de la etiqueta form. Si importamos FormsModule, automáticamente todos los formularios de nuestra aplicación serán “formularios angular”
- Con la notación #v=thing (#f=”ngForm”) estamos creando una variable local dentro de nuestra vista, que en este caso contiene nuestro ngForm (de tipo FormGroup)
- ngModel, al usarlo en los formularios sin atributo, indicamos que queremos que se comporte como one-way binding y que lo asocie a la propiedad indicada en el name del input
- Veamos el ejemplo (demo-form-sku)

FormBuilder

- Ahora vamos a ver un ejemplo de como construir un formulario con FormBuilder. Para usar FormBuilder tendremos que:
  - Importarlo en nuestro módulo
  - Declararlo como una dependencia en nuestro componente (recordad, en el constructor)
- FormBuilder tiene dos funciones:
  - control : nos permite crear un FormControl
  - group : nos permite crear un FormGroup
  - array: nos permite crear un array de FormGroup (anidar)
  - https://scotch.io/tutorials/how-to-build-nested-model-driven-forms-in-angular-2
- Veamos el ejemplo

Validaciones

- Angular nos proporciona validadores para dar feedback a nuestros usuarios mientras rellenan la información de sus formularios.
- Nos los proprociona el módulo Validators, por lo que tendremos que importarlo.
- Ejemplo de validador: Validators.required (obligatoriedad de un campo)
- Para utilizar un validador:
  - Debemos asignarlo a un FormControl
  - Debemos comprobar su estado en la vista y realizar alguna acción de acuerdo con ese estado.
- Para asignar un validador a un FormControl solo debemos pasárselo como argumento.
- Veamos el ejemplo

Validaciones

- Validaciones propias
- Varias validaciones por campo
- Doble data binding
- Observables
- Veamos el ejemplo
- ..
- ..
- Más avanzado:
  - Llamadas a servidor
  - Validadores propios complejos

Ejercicio

- Formulario para dar de alta usuarios:
  - Usuario
  - Contraseña
  - Fecha de nacimiento
  - Email

8.

Inyección de dependencias

Inyección de dependencias

- Según nuestra aplicación va creciendo se hace necesario que partes de ella se comuniquen con otros módulos. Cuando un módulo A requiere de un módulo B para ejecutarse, decimos que B es una dependencia de A
- Una de las formas más comunes de acceder a las dependencias es con un simple import.

```
import {B} from 'B';
B.foo();
```

- En muchos casos será suficiente, pero muchas otras veces no:
  - Cuando queramos sustituir B por MockB durante el testing.
  - Cuando queramos compartir una única instancia de B durante toda la aplicación (Patrón singleton)
  - Cuando queramos que se cree una nueva instancia de B cada vez que se use (Patrón factory)
- Estas necesidades nos las resuelve la inyección de dependencias.

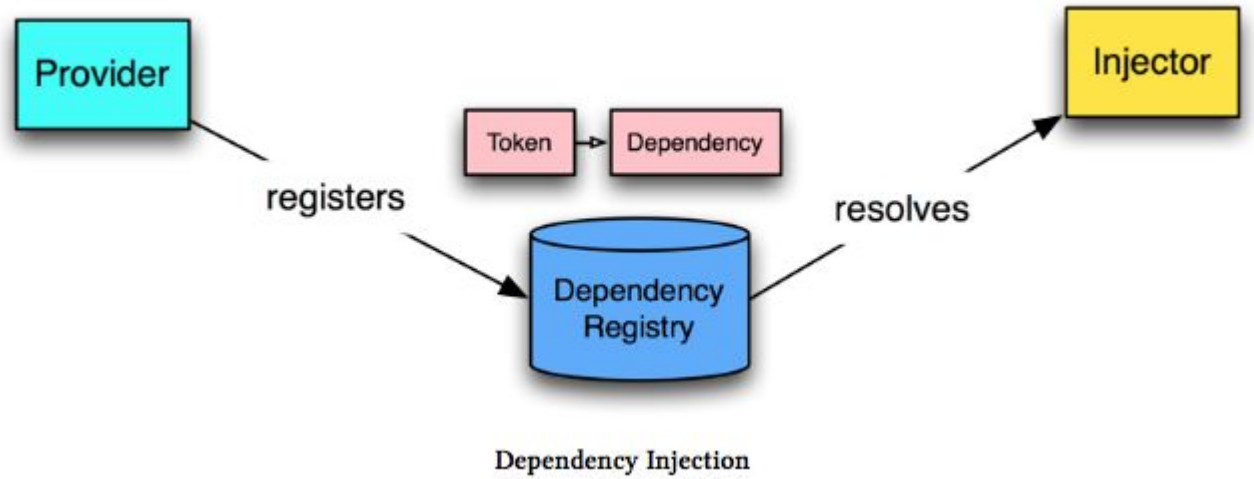
Inyección de dependencias

- El DI es un sistema que se encarga de hacer accesibles “partes” de nuestra aplicación a otras “partes”, permitiéndonos configurar de qué manera va a ocurrir.
- Una manera de ver el “inyector” es como un sustituto del operador new. De esta forma en vez de hacer un new X(), el DI nos entregará esos objetos permitiéndonos configurar cómo se van a crear.

Inyección de dependencias

- Para registrar una dependencia tenemos que vincularla a “algo” que nos permita identificar esa dependencia. Esta identificación se conoce como el “dependency token”. Por ejemplo si queremos registrar la URL de un API podemos usar la cadena ‘API_URL’ como token.
- Si queremos registrar una clase como dependencia, podemos usar la propia clase como token.
- La inyección de dependencias en Angular tiene 3 partes:
  - El **provider**: es el que nos mapea con un token (string o class)
  - El **injector** que es el que resuelve las dependencias y las inyecta donde sea necesario.
  - La **dependencia** que es lo que se está inyectando en sí

Inyección de dependencias



Dependency Injection

- Una manera de verlo es que cuando configuramos el DI especificamos **qué** es lo que se va a inyectar y **cómo** será “creado”

Inyección de dependencias

- Utilizaremos NgModule para registrar que es lo que queremos inyectar: lo indicaremos en los providers
- Marcaremos como @Injectable la clase que queremos utilizar en el DI

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

@Injectable()
export class  UserService {
    …
    …
    …

}
```

```
import {UserService} from
'./user.service';

@NgModule {
    …
    providers: [
        UserService
    ]
    …
}
```

Inyección de dependencias

```
import {Component} from '@angular/core';
Import {UserService} from './user/service';

@Component({
    …
})

export class  UserDemoComponent {
    …
    constructor(private userService: UserService) {
    }
}
```

- No nos tenemos que preocupar de donde sale UserService, el DI se ha ocupado de crearlo y de entregarnos la instancia, en este caso de tipo singleton que es el comportamiento por defecto

Providers

- Para que todo funcione es fundamental declararlo en el providers del NgModule. Es en esta declaración donde indicaremos **cómo** se resuelven las dependencias:
  - Como singleton (defecto)
  - Como value
  - Llamando a una función e inyectando el valor que devuelva dicha función
- Usando una clase. Es el comportamiento por defecto, el que nos entrega un singleton

```
providers: [UserService]
```

- Que es la notación acortada, ya que es el mas común, de:

```
providers: [
    {provide: UserService, useClass: UserService}
]
```

Providers

- Lo que indicamos en el parámetro **provide** sería el token de la dependencia
- Lo que indicamos en **useClass** es cómo y qué inyectar
- También podemos usar valores:

```
providers: [
     {provide: 'API_URL', useValue:'http://my.api.es/v1}
]
```

- En este caso el token sería una cadena, que en cuyo caso la inyectaríamos así:

```
constructor(@Inject('API_URL') apiUrl: string) {
}
```

9.

Servicios

Servicios

- Muy simples de explicar, ya que son una simple clase TypeScript en la que podemos implementar la lógica que necesitemos.
- Los marcaremos con el decorador @Injectable

```
> ng g service data


installing service
  create src\app\data.service.spec.ts
  create src\app\data.service.ts
  WARNING Service is generated but not provided, it must be provided to be used
```

10.

HTTP

HTTP

- Angular viene con su propia librería HTTP que podemos utilizar para llamar a nuestras APIS externas.
- Las llamadas serán asíncronas para no bloquear la interacción con el usuario.
- Tenemos 3 aproximaciones para trabajar de forma asíncrona:
  - Callbacks
  - Promesas
  - Observables
- En Angular la forma aconsejada es con Observables.
- Deberemos importar una librería distinta, ya que angular no nos obligará a trabajar con la suya. (@angular/http)

```
Import {
    HttpModule,
    Http,
    …
} from '@angular/http';
```

HTTP

- Métodos http estándar para hacer las peticiones
  - http.request
  - http.post
  - http.put
  - http.patch
  - http.delete
  - Http.head
- Opciones:
  - method
  - headers
  - body
  - cache
  - ...
- Veamos un ejemplo sencillo

11.

Pipes

Pipes

- Permiten formatear los datos que vamos a mostrar a los usuarios
- Tenemos filtros predefinidos:
  - DatePipe
  - UpperCasePipe
  - LowerCasePipe
  - CurrencyPipe
  - PercentPipe
  - DecimalPipe
  - AsyncPipe
  - ...
- Ejemplo

Pipes

- Podemos crear nuestros propios Pipes
- Para ello tenemos que implementar el interfaz PipeTransform y decorarla con @Pipe
- El método transform recibirá como parámetros el input a transformar y los posibles parámetros del pipe
- Los pipes se pueden componer uno detrás de otro, tomando cada uno la salida del anterior.
- Tendremos que declararlas en nuestro NgModule (igual que los componentes)

```
@Pipe({name: 'exponentialStrength'})

export class ExponentialStrengthPipe implements PipeTransform
{
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

Ejercicio

- Crear una nueva aplicación angular-cli
- Añadir un pipe: ng g pipe ParImpar
- Construir un pipe que dado un campo numérico escriba a su lado si han introducido un valor par o impar

```
@Pipe({name: 'exponentialStrength'})

export class ExponentialStrengthPipe implements PipeTransform
{
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

12.

Arquitectura de datos

Observables

- Los observables son streams de datos.
- Nos podemos suscribir a un stream de datos y ejecutar operaciones que reaccionen a los cambios
- RxJS es la librería más popular en cuanto a streams reactivos en JS y nos aporta operadores muy potentes para componer operaciones sobre streams de datos.
- Usar observables para estructurar nuestros datos se conoce como programación reactiva.
- Angular es extremadamente flexible en cuanto a arquitectura de datos. Una estrategia de datos que funcione para un proyecto no necesariamente tiene que funcionar para otro, y es por eso que Angular no prescribe por defecto ninguna en concreto y nos permite elegir la más adecuada según nuestras necesidades.

Observables / Programación reactiva

- La programación reactiva es una forma de trabajar con streams de datos asíncronos, y los Observables son su estructura principal de datos.
- No es una terminología muy intuitiva, vamos a intentar aclararlo un poco.
- Las promesas emiten un valor simple mientras que los observables son streams que emiten varios valores.
- El código “imperativo” trae datos, mientras que el “reactivo” emite datos.
- Los streams se pueden componer formando un conjunto de operaciones sobre nuestros datos.
- Vamos a ver un ejemplo para hacernos una idea, pero no es la mejor forma de empezar a trabajar en Angular por su complejidad inicial.
- Veamos el ejemplo de youtube también.

13.

Tips

Validadores asíncronos

http://www.carlrippon.com/?p=564

Resolves

- https://angular.io/docs/ts/latest/api/router/index/Resolve-interface.html
- https://angular.io/docs/ts/latest/api/router/index/Event-type-alias.html

```
this.router.events
  .subscribe((event) => {
    if (event instanceof NavigationEnd) {
      console.log('NavigationEnd:', event);
    }
  });
```

Práctica final

- Aplicación con rutas
- Ruta: home
- Ruta: listado de bizis
- Ruta: detalle de bizi
- Ruta: mapa de bizi
- Servicio consulta de API
- ¿Nos atrevemos con un mapa? https://github.com/ng2-ui/map
- ¿Resolve en las rutas?
- ¿Errores en cambios de ruta?

Gracias!

¿Preguntas?

Podéis encontrarme en...

Raúl Novoa

Co-founder at @10labs @fidelizoo @neki_global

JS MEAN Developer / iOS Developer

http://linkedin.com/in/raulnovoa

@raul_novoa

raul@10labs.es