

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 practical

0.1.

Conceptos
básicos de
Javascript



**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. 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
```

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

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:

- ▣ `>, <, >=, <=, ==, !=`

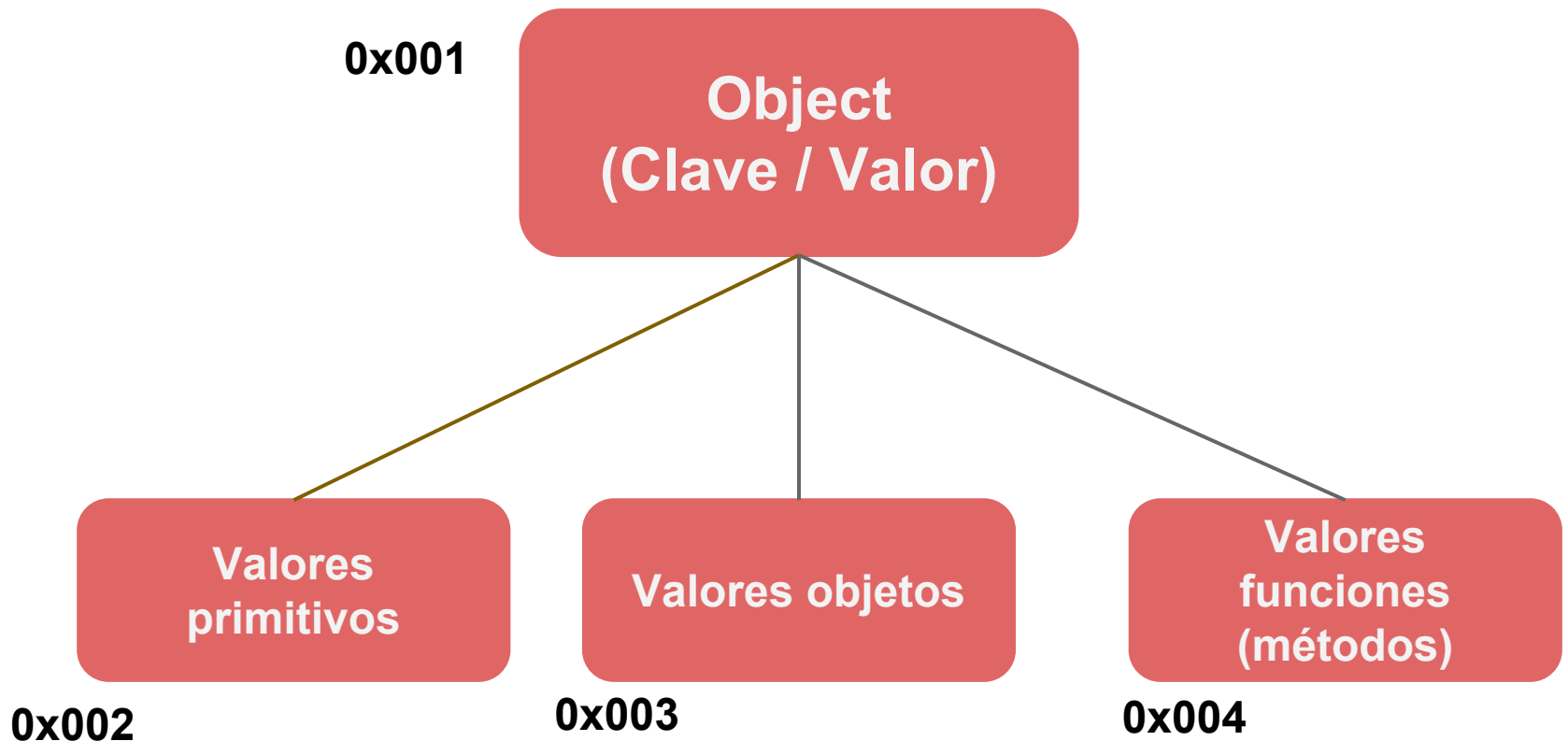
Operadores lógicos:

- ▣ `&&, ||, !`

Operador ternario:

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

Objects (I)



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

== VS ===

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

- ▣ `'5' == 5 , '5' === '5'` $\Rightarrow$ `true, true`
- ▣ `null==undefined, null==0` $\Rightarrow$ `true, false`
- ▣ `null===undefined` $\Rightarrow$ `false`
- ▣ `0==false, ''==false` $\Rightarrow$ `true, true`
- ▣ `0===false, ''===false` $\Rightarrow$ `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 Constructor 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

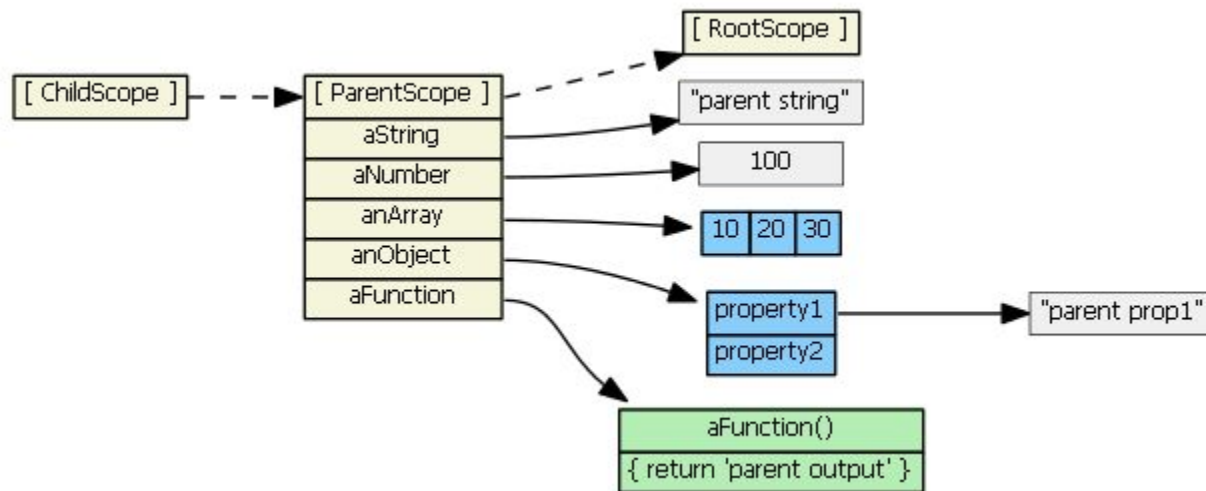
```
function Persona(nombre, apellido, edad, colorOjos) {  
    ...  
    this.colorOjos = colorOjos;  
}
```

- 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

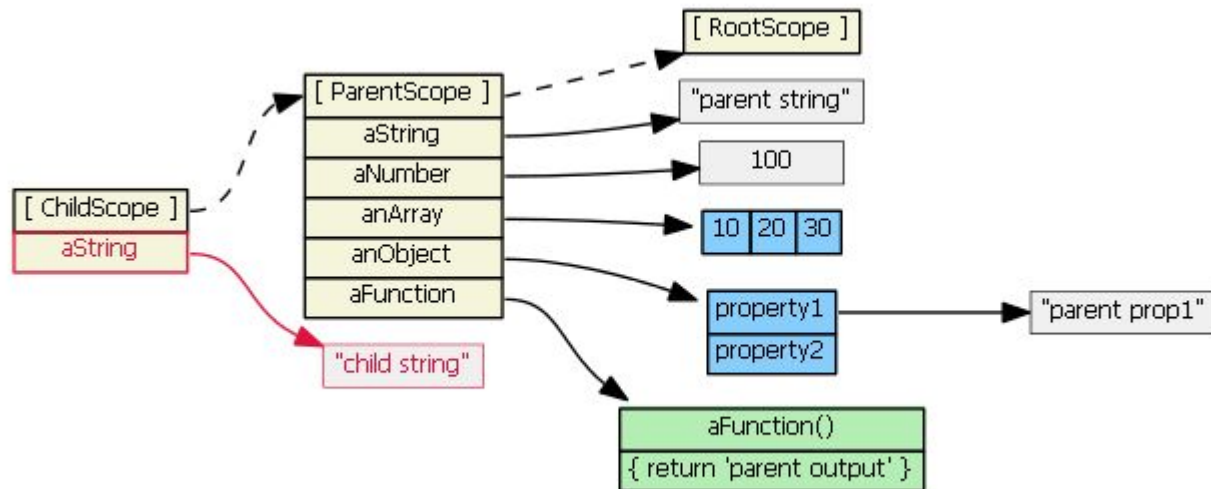


```
childScope.aString === 'parent string'
childScope.anArray[1] === 20
childScope.anObject.property1 === 'parent prop1'
childScope.aFunction() === 'parent output'
```

JS Prototypal Inheritance (II)

- Si ahora hacemos:

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

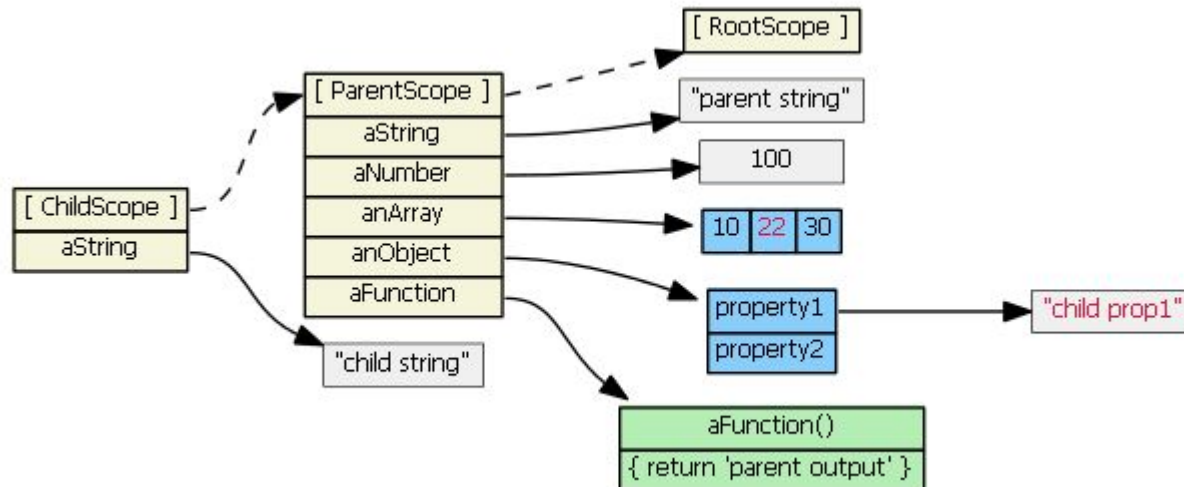


JS Prototypal Inheritance (III)

- Si ahora hacemos:

```
childScope.anArray[1] = '22';
```

```
childScope.anObject.property1 = 'child prop1';
```

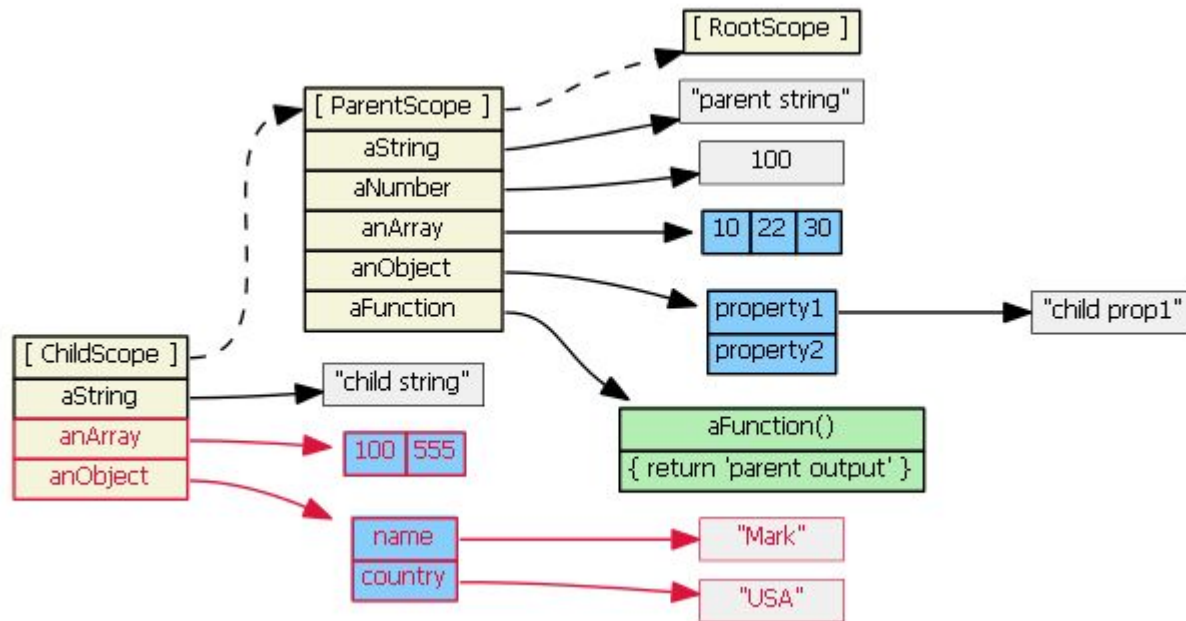


JS Prototypal Inheritance (IV)

- Si ahora hacemos:

```
childScope.anArray = [100, 555];
```

```
childScope.anObject = {name: 'Mark', country: 'USA'}
```

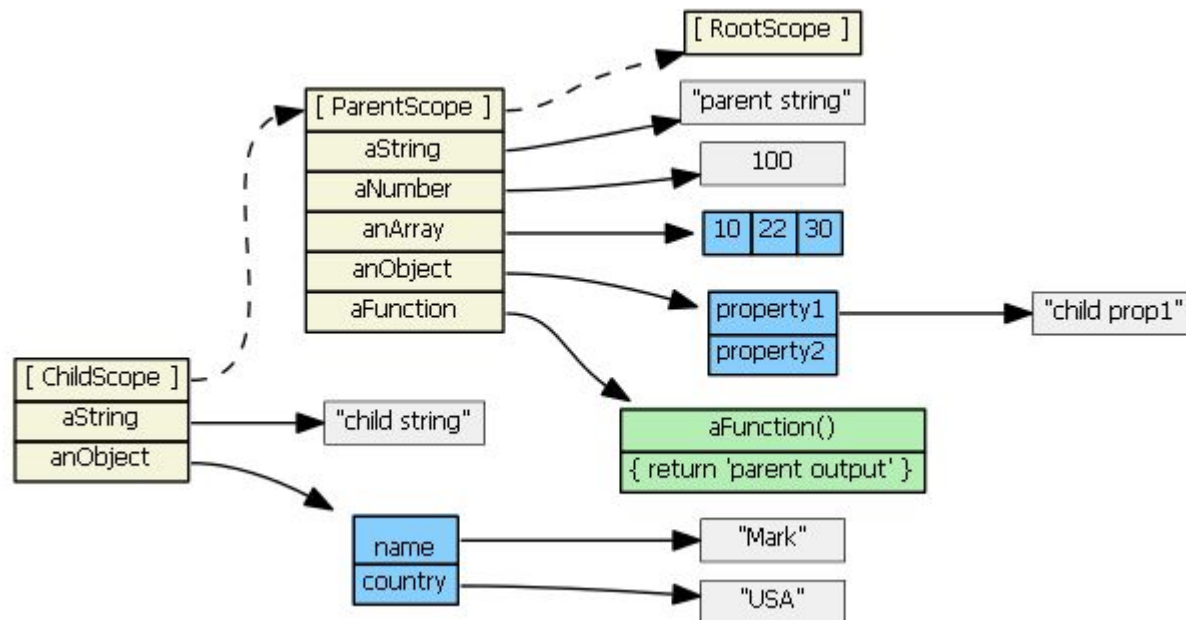


JS Prototypal Inheritance (V)

- 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).

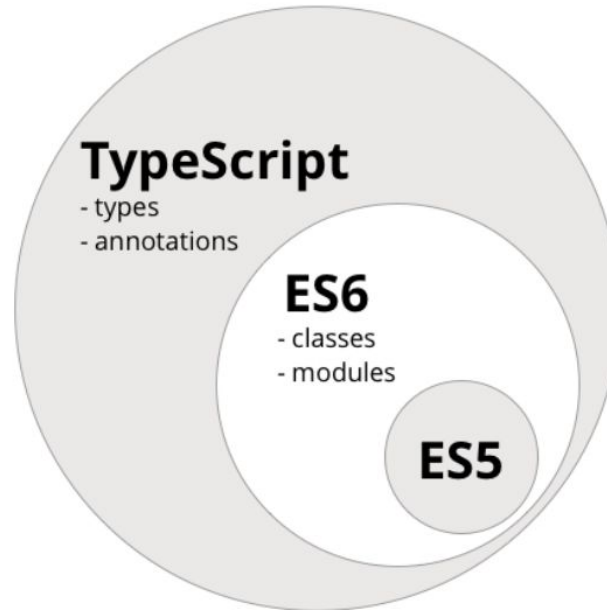
```
function cardID() {  
    var cardID = 100;  
    return {  
        getID: function() {  
            return cardID;  
        },  
        setID: function(newID) {  
            cardID = newID;  
        }  
    }  
}  
  
var myCardID = cardID();  
myCardID.getID(); //100  
myCardID.setID(342);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

1.

Typescript

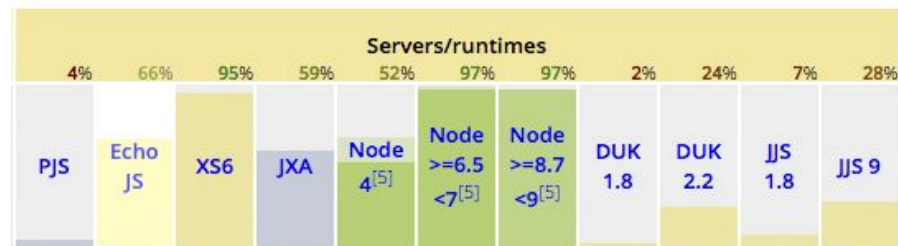
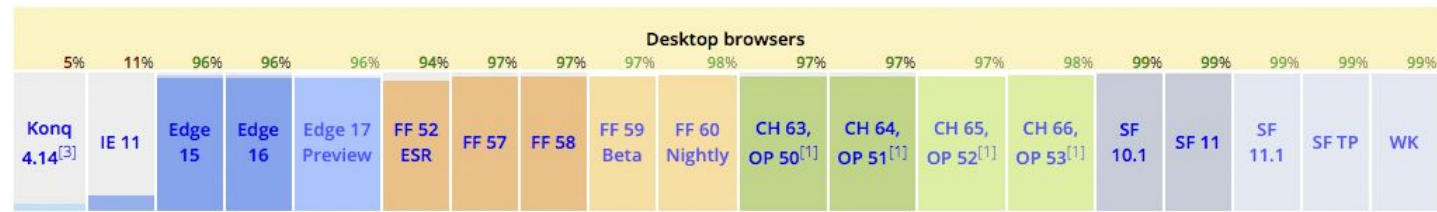
¿Que es TypeScript?

- 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].substr(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 sobrescribir 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, sobrescribir 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.');
```

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?

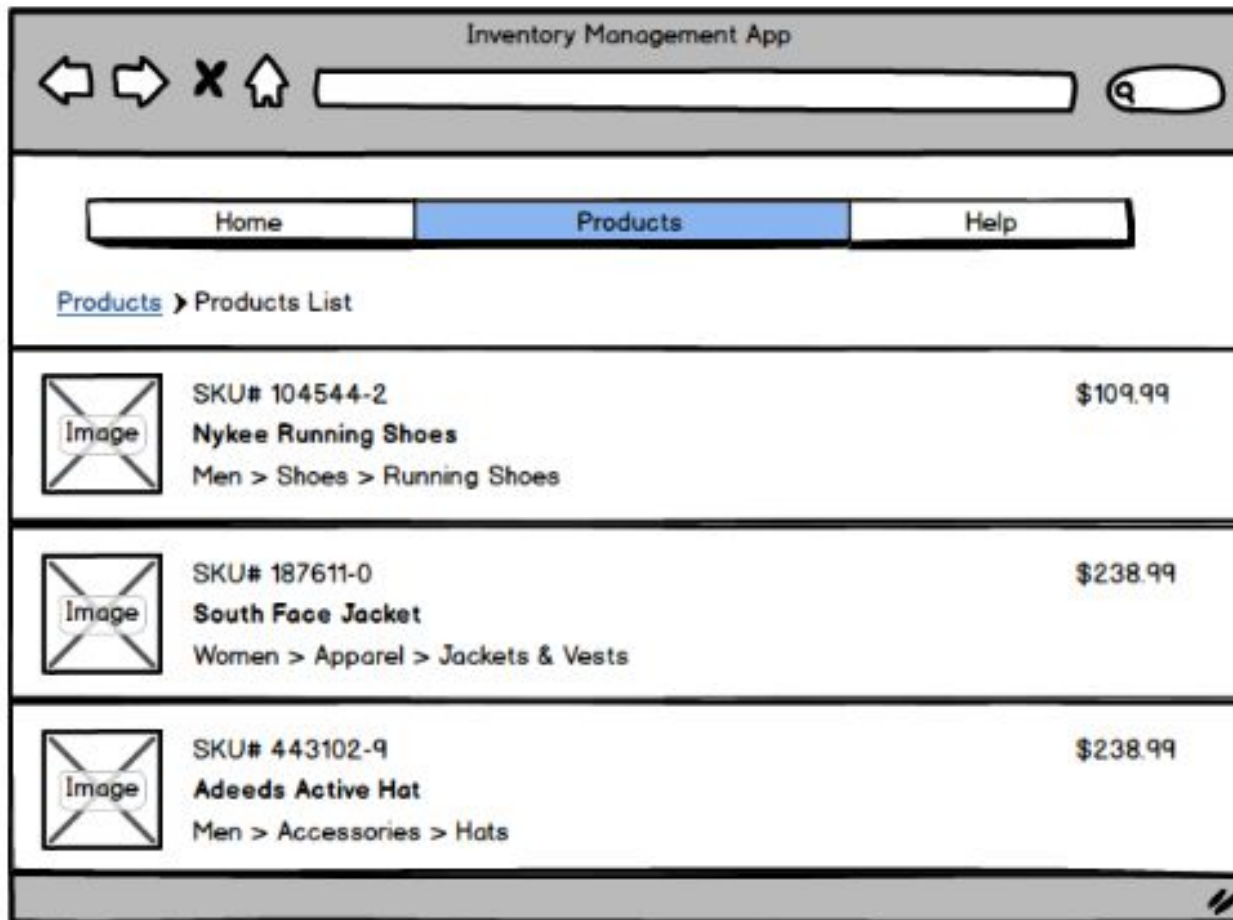
¿Como funciona Angular?

- 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

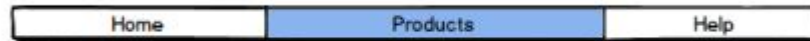
- Una aplicación Angular no es nada más que un árbol de componentes.
- En el nodo raíz del árbol, el primer componente es la aplicación en sí.
- Una de las grandes características de los componentes es que podemos anidarlos. Es decir podemos construir un componente mayores basados en pequeños componentes.
- 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

[Products](#) > Products List

- Componente listado de productos

	SKU# 104544-2 Nykee Running Shoes Men > Shoes > Running Shoes	\$109.99
	SKU# 187611-0 South Face Jacket Women > Apparel > Jackets & Vests	\$238.99
	SKU# 443102-9 Adeeds Active Hat Men > Accessories > Hats	\$238.99

Aplicación - Ejemplo

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



- Componente departamento del producto

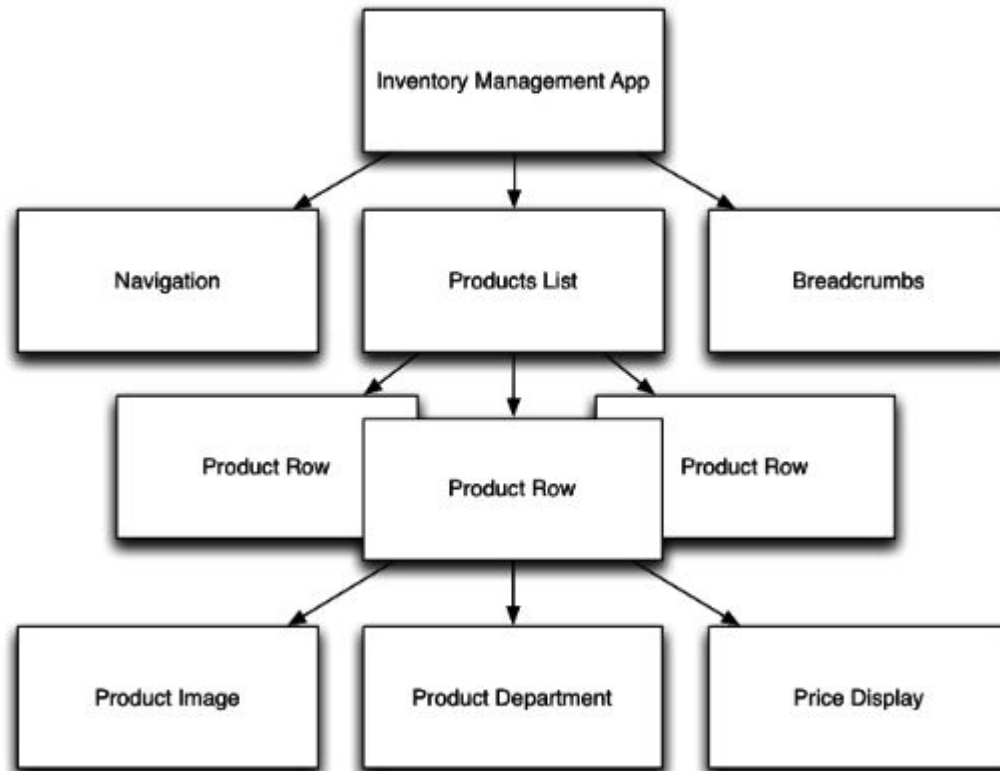
Men > Shoes > Running Shoes

- 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

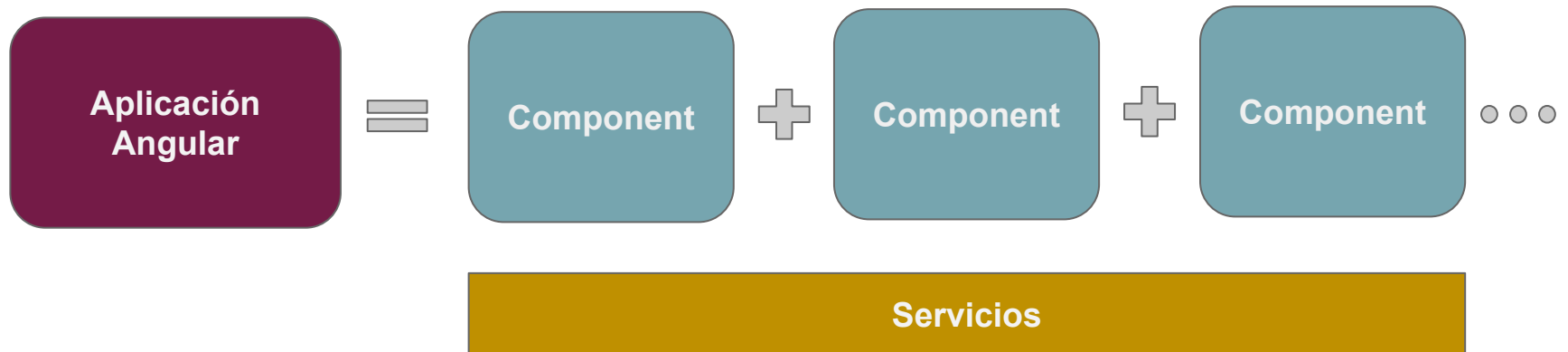


- Una aplicación angular tiene que tener un componente principal (solo uno).

```
<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



3.

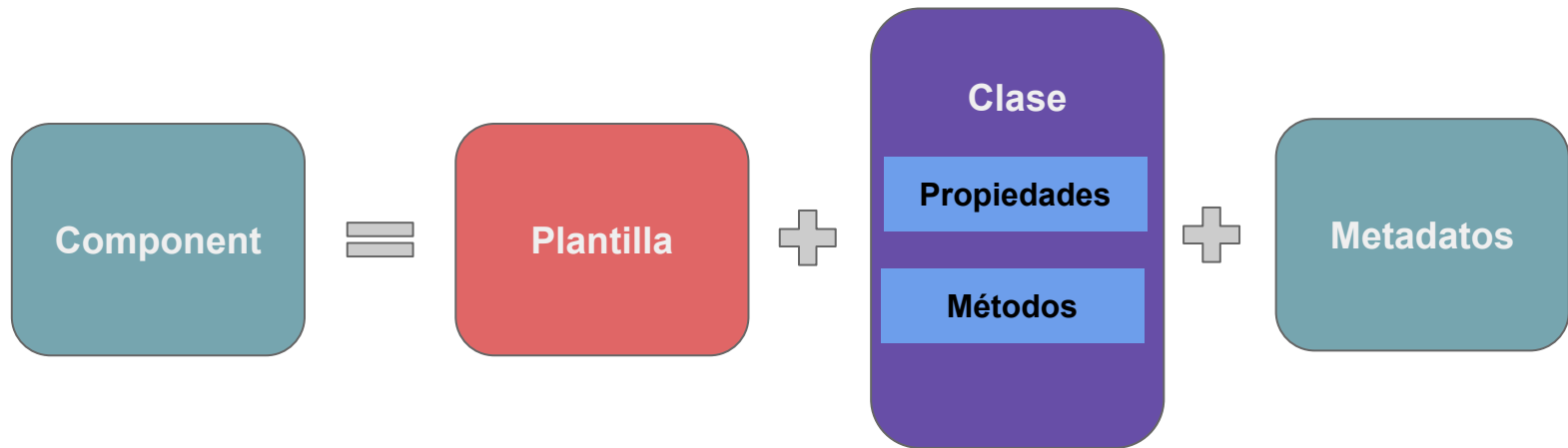
Componentes

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



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 (outputs).
- Podemos decir que:
 - Los datos suponen un flujo de entrada mediante los input bindings.
 - Los eventos suponen un flujo de salida mediante los output 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 practical  
> cd practical  
> ng g component p1-selector
```

Práctica 1

Último botón pulsado:

Botón 1 - C1

Botón 1 - C2

Botón 2 - C1

Botón 2 - C2

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 EventEmmitter 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”)

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

