



# TypeScript

Matías Ramos



# JAVA

- ▶ Imperativo
- ▶ Funcional (lambda)
- ▶ Recolector de basura
- ▶ Tipado estático
- ▶ Orientado a Objetos (con clases)
- ▶ Modularidad con paquetes.

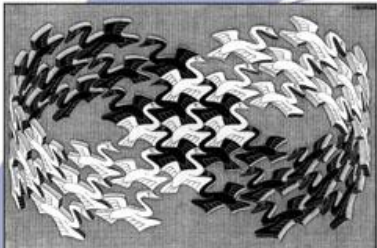


# ORIENTACIÓN A OBJETOS

## Design Patterns

Elements of Reusable  
Object-Oriented Software

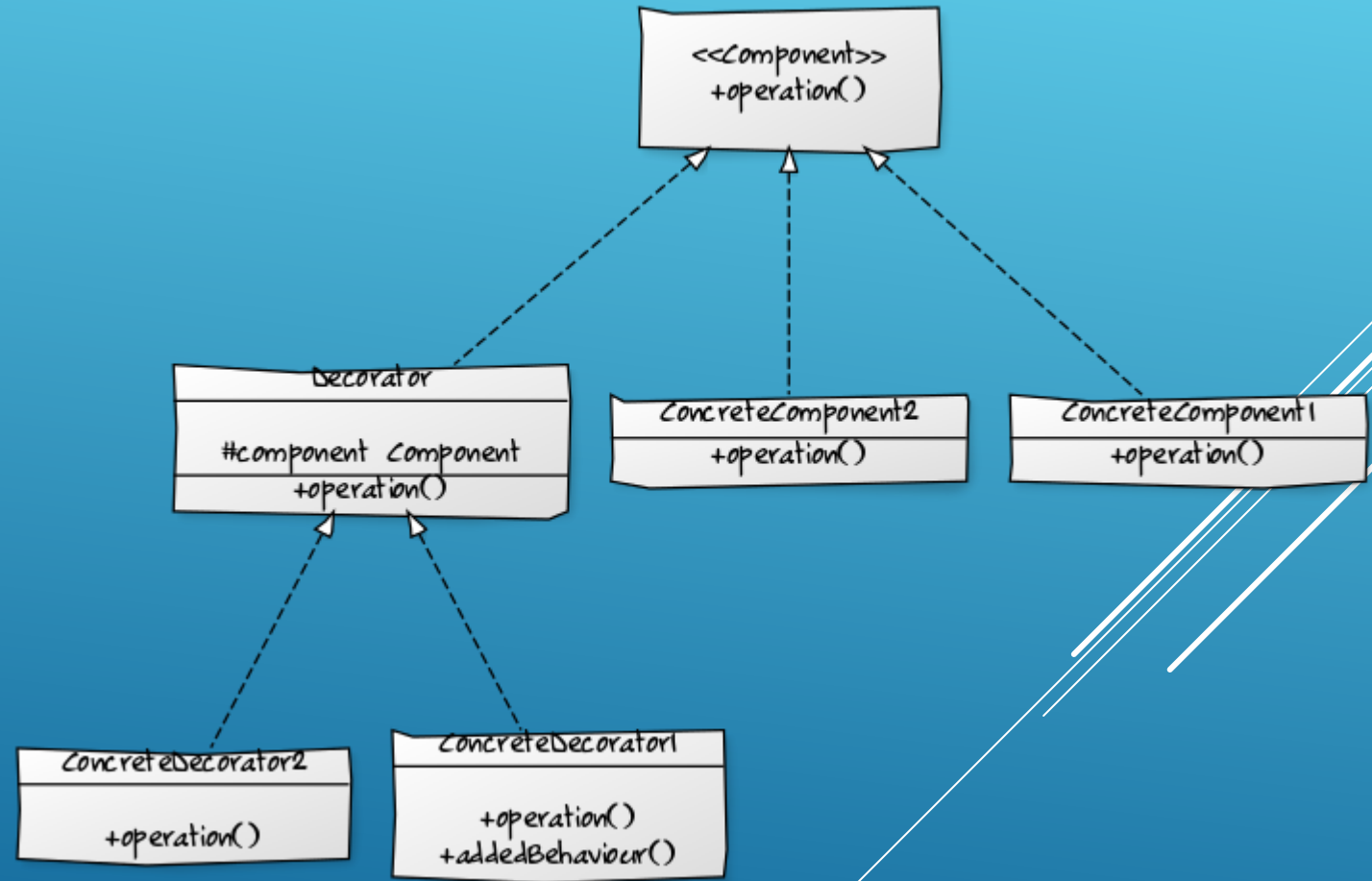
Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Cover art © 1994 M.C. Escher / Gordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES









# TIPADOS ESTÁTICOS

```
public class MainTest {  
  
    public static void main(String[] args) {  
        miBoolean = true;  
    }  
}
```

  miBoolean cannot be resolved to a variable

4 quick fixes available:

-  [Create local variable 'miBoolean'](#)
-  [Create field 'miBoolean'](#)
-  [Create parameter 'miBoolean'](#)
-  [Remove assignment](#)



# TIPADOS ESTÁTICOS

```
public class Foo {  
    public void foo(){
```

eq

```
}
```

```
}
```

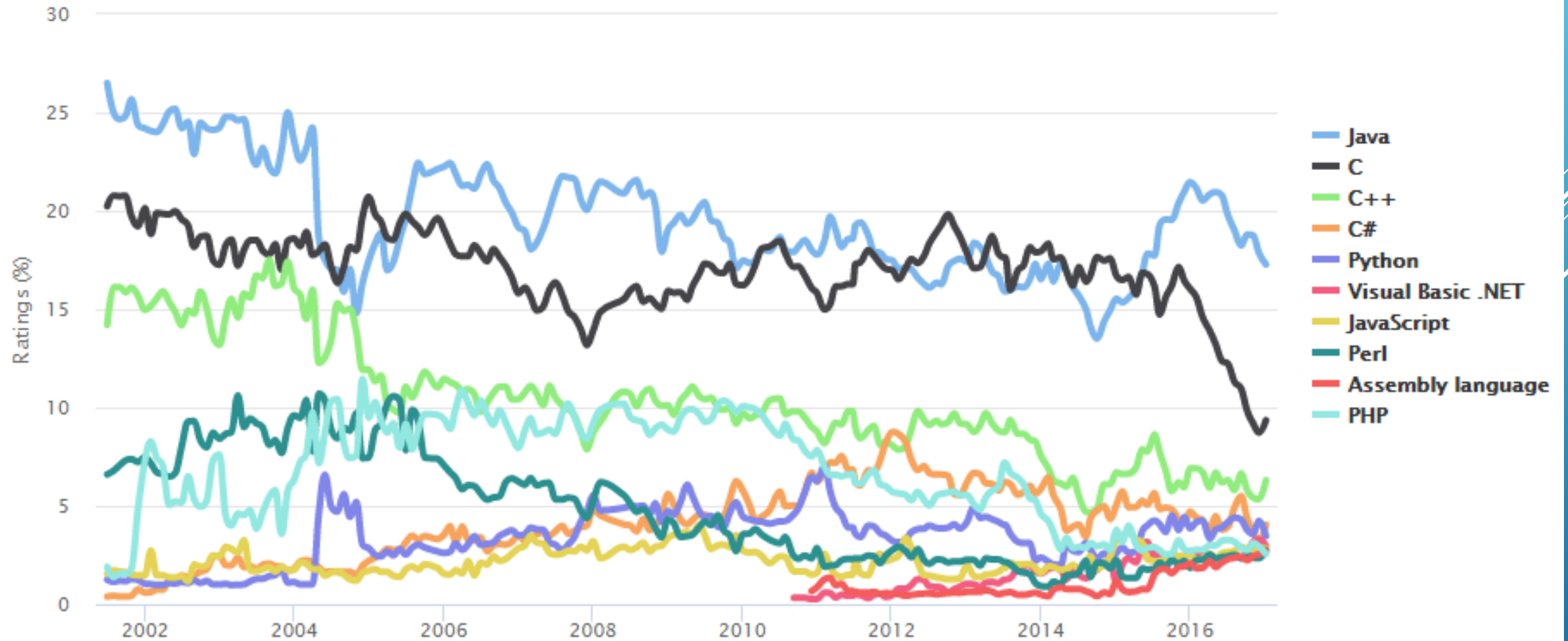
- equals(Object obj) : boolean - Object
- ⓘ ExtendedRequest - javax.naming.ldap
- Ⓒ EventQueue - java.awt



# TIOBE

## TIOBE Programming Community Index

Source: [www.tiobe.com](http://www.tiobe.com)



# JS (ES5)

## CARACTERÍSTICAS

- ▶ Imperativo
- ▶ Funcional
- ▶ Recolector de basura
- ▶ **Tipado dinámicos**
- ▶ Orientado a Objetos (**con prototipos**)
- ▶ Sin modularidad.

- ▶ El compilador no te ayuda
  - ▶ Hay que ejecutar los test (si se tienen)
- ▶ El IDE tampoco te ayuda
  - ▶ No se puede refactorizar de forma automática.
  - ▶ El auto completado es muy limitado.
  - ▶ No se puede navegar a la implementación.



- ▶ Existen tres formas diferentes de implementar “clases”
  - ▶ Prototipos “a mano”
  - ▶ Simulación de clases con librerías
  - ▶ Patrón modulo usando clousuers



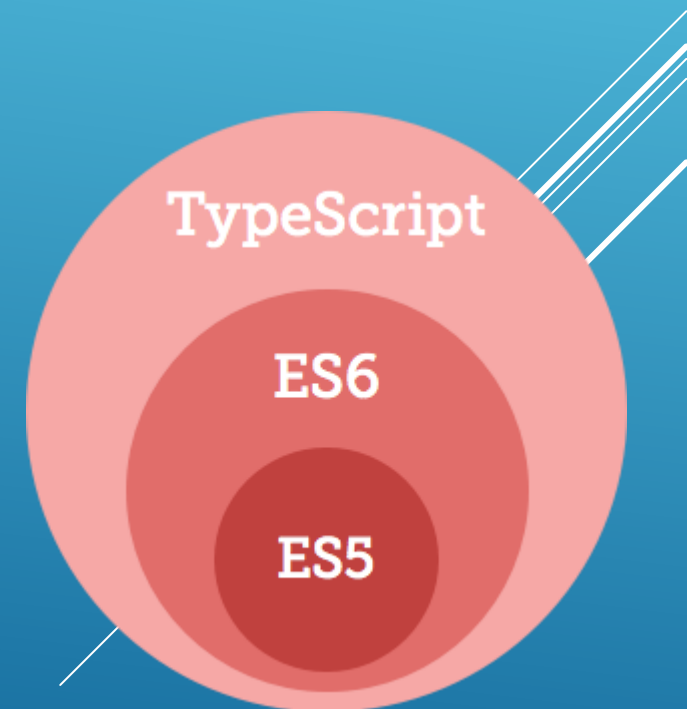
# ORIENTADO A OBJETOS

CON PROTOTIPOS

- ▶ La herencia no es limpia.
- ▶ Los patrones de diseño OO no se pueden aplicar directamente.

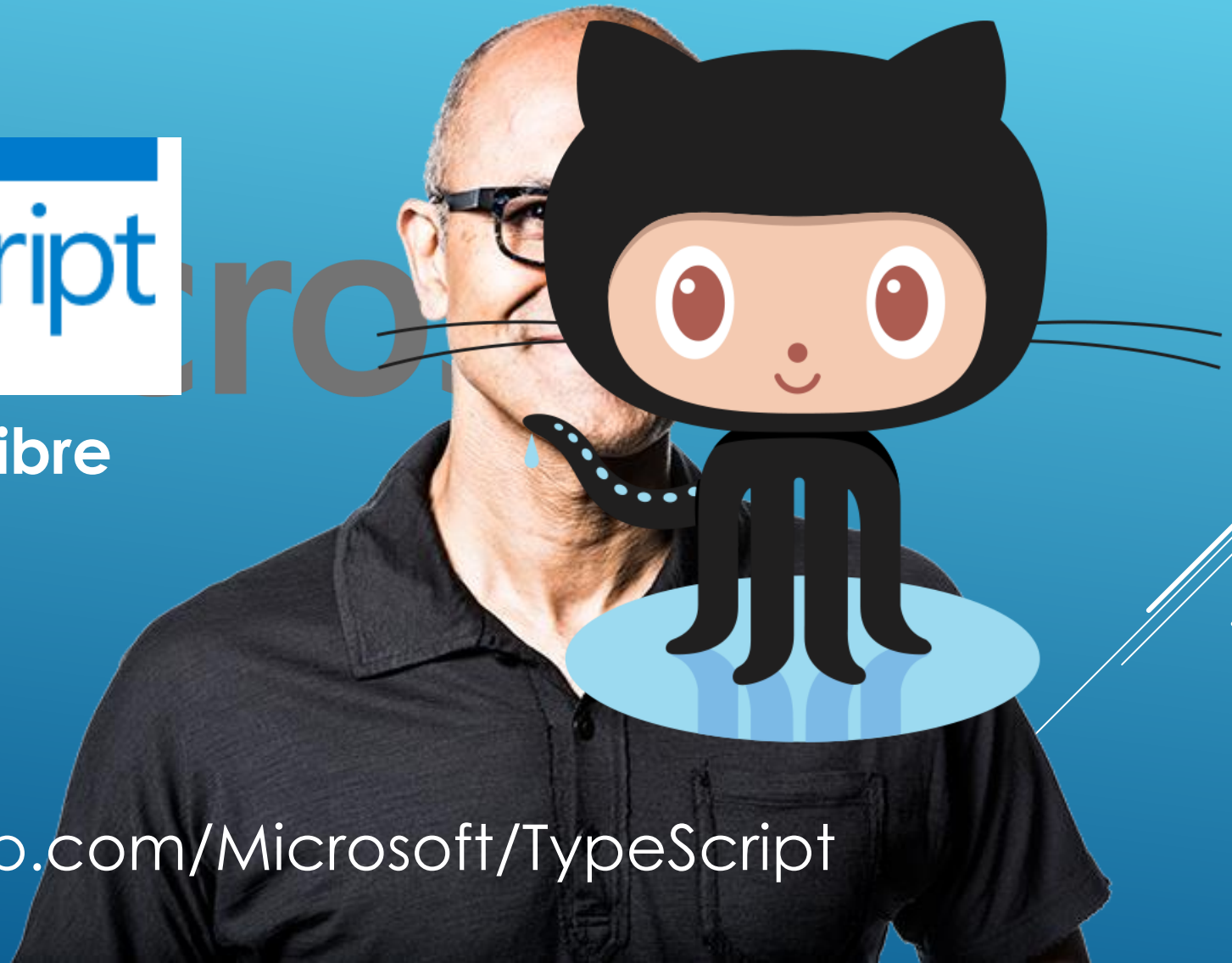
# ¿QUÉ ES TYPESCRIPT?

- ▶ TypeScript es un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft.
- ▶ Es un superset estricto de JavaScript, y añade, de manera opcional, tipado estáticos y herramientas que ayudan a desarrollar a través de metodologías de programación orientada a objetos basado en clases.
- ▶ puede ser utilizado para desarrollar aplicaciones en JavaScript para el lado del cliente o en el servidor a través de Node.js.



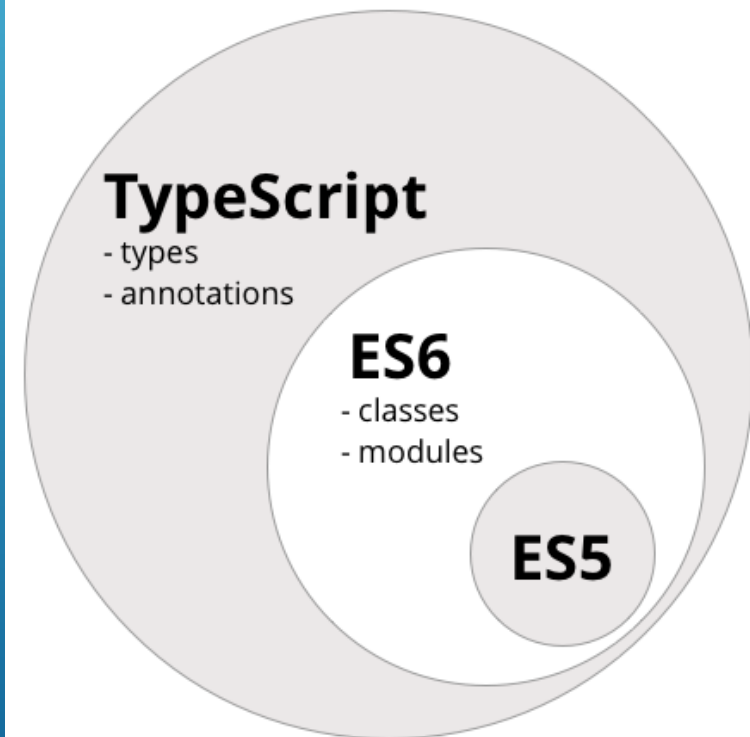
TS

SATYA NADELLA - CEO



<https://github.com/Microsoft/TypeScript>

- ▶ Añade tipos estáticos a JavaScript ES6
  - ▶ Inferencias de tipos
  - ▶ Tipos opcionales
- ▶ El compilador genera código JavaScript ES5 (Navegadores actuales)
- ▶ Orientado a Objetos con clases. (No como ES5)
- ▶ Anotaciones (ES7)



## CASE JAVA VS CLASE TS

```
1 class Empleado{
2
3     private String nombre;
4     private double salario;
5
6     public Empleado(String nombre,
7         double salario){
8         this.nombre = nombre;
9         this.salario = salario;
10    }
11
12    public String getNombre(){
13        return nombre;
14    }
15
16    public String toString(){
17        return "Nombre:" + nombre +
18            ", Salario: " + salario;
19    }
20 }
```

```
1 export class Empleado{
2
3     private nombre:string;
4     private salario:number;
5
6     constructor(nombre:string,
7         salario:number){
8         this.nombre = nombre;
9         this.salario = salario;
10    }
11
12    getNombre(){
13        return this.nombre;
14    }
15
16    toString(){
17        return "Nombre: " + this.nombre +
18            ", Salario: " + this.salario;
19    }
20 }
```

```
1 export class Empleado{
2
3     private nombre:string;
4     private salario:number;
5
6     constructor(nombre:string,
7         salario:number){
8         this.nombre = nombre;
9         this.salario = salario;
10    }
11
12    getNombre(){
13        return this.nombre;
14    }
15
16    toString(){
17        return "Nombre: "+this.nombre+
18            ", Salario: "+this.salario;
19    }
20 }
```

```
1 "use strict";
2 var Empleado = (function () {
3     function Empleado() {
4         this.nombre = "nombre";
5         this.salario = 123;
6     }
7     Empleado.prototype.getNombre = function () {
8         console.log("asdasdasd");
9         return this.nombre;
10    };
11    Empleado.prototype.toString = function () {
12        return "Nombre: " + this.nombre +
13            ", Salario: " + this.salario;
14    };
15    return Empleado;
16 })();
17 exports.Empleado = Empleado;
```

TS

# TIPOS BÁSICOS



Object

void

boolean

int long  
short ...

String

Type[]



TS

any

void

boolean

number

string

Type[]



# TIPOS BÁSICOS

- ▶ Los tipos básicos que maneja Typescript son booleans, number, string, Any y Void.

```
1 var isDone: boolean = false;  
2 var height: number = 6;  
3 var name: string = "bob";  
4 var list: number[] = [1, 2, 3];  
5 var notSure: any = 4;  
6 function warnUser(): void {  
7   alert("This is my warning message");  
8 }
```

- ▶ tipo Any es un tipo dinámico, se utiliza principalmente cuando no queremos declarar el tipo o también se usa en arrays que contienen distintos tipos.
- ▶ void se utiliza principalmente para declarar el tipo de funciones que no devuelven nada, cómo en el ejemplo de arriba.

DEMO



# TS

## CLASSES E INTERFACES

```
1 interface Animal {  
2   name : string;  
3   makeSound();  
4 }  
5  
6 class Dog implements Animal {  
7   name:string;  
8  
9   constructor(name:string) {  
10    this.name = name;  
11  }  
12  makeSound() {  
13    return "guau!";  
14  }  
15 }  
16  
17 function sayHi(animal:Animal) {  
18   console.log("hi " + animal.name);  
19 }  
20  
21 sayHi(new Dog("Timmy"))  
22  
23
```

# TS

## INSTALAR

- ▶ El primer requisito para utilizar TypeScript es instalar Node.js. La forma mas conveniente de instalación es a través de npm, el package manager que viene por default con Node.js.

```
npm install -g typescript
```

# ¿CÓMO COMPILAR?

- ▶ TypeScript sencillamente produce código en plain JavaScript. Podemos compilar el código de dos maneras, a través del terminal o usando algún IDE que automáticamente compile el código.
- ▶ Desde el terminal podemos compilar de forma cavernícola con tan solo ejecutar el siguiente código:


```
tsc --sourcemap archivo.ts
```

- ▶ Los genéricos son muy útiles para hacer código mas reusable, uno de los claros ejemplos son las listas/Arrays, en el siguiente extracto podemos observar como podemos definir el tipo del Array.

```
1 var animals:Array<Animal> = [new Dog('Timmy'), new Dog('Michael'), new Dog('Dwight')];  
2 animals.forEach(sayHi);
```

Esto puede ayudar sobre todo a hacer librerías y piezas de código más reutilizables.

# MÁS CARACTERÍSTICAS

- ▶ Funciones lambda (llamadas arrow function).
  - ▶ Módulos (exportar e importar elementos)
  - ▶ Anotaciones
  - ▶ Programación pseudo-síncrona con `async` / `await`
- 
- Several thin, parallel white lines of varying lengths and orientations are positioned in the bottom right corner of the slide, creating a modern, abstract graphic element.

DEMO





# ARROW FUNTION

- ▶ Sintaxis () => {}
- ▶ Sin lanbda

```
1    var foo = function (x, y) {  
2        return x + y;  
3    };
```

- ▶ Con lanbda

```
1    var foo = (x: number, y: number) => x+y;  
2
```

- ▶ Este quizás sea uno de los puntos más necesarios a la hora de conseguir una mejor arquitectura en las aplicaciones Javascript.
- ▶ Typescript lo resuelve usando una sintaxis parecida a la que veremos en Javascript cuando el estándar ES6 se implemente en los navegadores.
- ▶ Typescript tiene dos tipos de módulos internos y externos.

► Módulo global

```
//----- a.ts -----  
var a = 123;
```

```
//----- b.ts -----  
var b = a; // Permitido
```

► Modulo de archivo (módulos externos)

```
//----- a.ts -----  
export var a = 123;
```

```
//----- b.ts -----  
var b = a; // NO permitido - no encontraria el valor de a
```

- Modulo de archivo (módulos externos)

```
//----- a.ts -----  
export var a = 123;
```

```
//----- b.ts -----  
import {a} from "./a";  
var b = a; // Permitido, ahora si encontraria el valor
```

- Modulo

```
//----- a.ts -----  
// Se puede realizar un módulo sin emplear declare  
declare module "a" {  
    export var a:number; /*sample*/  
}
```

```
//----- b.ts -----  
import var a = require("./a");  
var b = a;
```

# MÓDULOS VS NAMESPACE

- ▶ Modulo
  - ▶ estará normalmente dentro de un archivo
- ▶ Namespace
  - ▶ puede ser un conjunto de archivos, permitiéndonos así englobar una serie de clases(archivos) bajo un mismo namespace.
  - ▶ es considerado un módulo interno.

```
//----- namespace.ts -----  
// Módulos internos TypeScript  
namespace MySpace {  
    export class MyClass {  
        public static myProperty: number = 1;  
    }  
}
```

```
//----- importacionImport.ts -----  
import {MySpace} from 'namespace.ts'; // Cuidado  
console.log(MySpace.MyClass.MyProperty);
```

```
//----- importacionReference.ts -----  
/// <reference path="namespace.ts" />  
console.log(MySpace.MyClass.MyProperty);
```

# ASYNC / AWAIT

- ▶ Las async no pueden ser utilizados en es5 deben ser usados en es6 o superior.
- ▶ Await se utiliza para parar la ejecución del código hasta que la función termine correctamente.
- ▶ Si la función falla, generará un error de manera sincrónica que podremos atrapar mediante un try catch.

```
async function foo() {  
  try {  
    var val = await getMeAPromise();  
    console.log(val);  
  }  
  catch(err) {  
    console.log('Error: ', err.message);  
  }  
}
```

- ▶ Si la función termina entonces devolverá un valor
- ▶ Si la función falla devolverá un error que podremos capturar

# ASYNC / AWAIT

- ▶ Esto convierte drásticamente la programación asíncrona tan fácil como la programación síncrona. ya que cumple 3 requisitos indispensables:
  - ▶ Capacidad de pausar la función en tiempo de ejecución
  - ▶ Capacidad de pasarle valores a funciones
  - ▶ Capacidad de lanzar excepciones en caso de fallo

TS

IDE

