UD 9
Frameworks

DESARROLLO WEB EN ENTORNO CLIENTE

Técnico de Grado Superior Desarrollo de Aplicaciones Web

2024-25

IES Velázquez
Sevilla

J. Mario Rodríguez
jrodper183e@g.educaand.es

Contenidos

- Introducción y contexto
- Evolución de los frameworks en cliente
- Tipos de aplicaciones web
- ¿Por qué utilizar frameworks?
- Características comunes de los framewoks
- Otros frameworks y conceptos relacionados

- Prueba de concepto: el Framework Angular.
  - Introducción a Angular 19.
  - Angular CLI.
  - Typescript.
  - Directivas.
  - Data Binding.
  - Componentes.
  - Módulos / Standalone.

Introducción

Los frameworks de desarrollo en cliente han surgido para optimizar la creación de aplicaciones web modernas.

A lo largo del tiempo, la complejidad de las aplicaciones web ha aumentado.
Se hizo necesario mejorar la eficiencia, la mantenibilidad y la escalabilidad del código.

Vale, ¿y qué es un framework?

Un framework es un conjunto de herramientas, bibliotecas y convenciones que establecen una estructura y pretenden facilitar el desarrollo.

Se diferencian de las bibliotecas en que imponen una forma específica de trabajar, mientras que una biblioteca es *simplemente* un conjunto de funciones reutilizables.

Algunos frameworks relevantes en cliente

1995 JavaScript

- 2006 - Aparece jQuery, facilitando la manipulación del DOM y el uso de AJAX.
- 2010 - **AngularJS** (Google)
  primer framework que introduce el concepto de MVVM
  (con permiso de Backbone.js y Knockout.js)
- 2013 - **React** (Facebook)
  revoluciona con su enfoque basado en componentes y Virtual DOM
- 2014 - **Vue.js**
  aparece como una alternativa ligera y flexible a Angular y React
- 2015 - **Angular** (nueva versión de AngularJS)
  adopta TypeScript y una arquitectura más robusta
- 2016 en adelante
  otros enfoques como JAMStack (Astro, Next.js, Svelte, etc.)
  optimizan el rendimiento y la carga diferida

Tipos de aplicaciones: SPA vs. MPA

SPA (Single Page Application)

Carga una sola página HTML y actualiza dinámicamente el contenido con JS.

Ventajas

Experiencia fluida, menor consumo de ancho de banda.

Inconvenientes

SEO complicado, mayor carga inicial.

Ejemplos

Gmail, Facebook…

MPA (Multi Page Application)

Cada interacción carga una nueva página desde el servidor.

Ventajas

Mejor SEO, simplicidad en seguridad y caché.

Inconvenientes

Experiencia menos fluida, más peticiones al servidor.

Ejemplos

Wikipedia, Amazon…

¿Por qué utilizar frameworks?

Beneficios

- Facilitan el desarrollo estructurado.
- Mejoran la reutilización de código.
- Optimizan el rendimiento y la seguridad.
- Integran herramientas de prueba y depuración.
- Puede haber una comunidad y ecosistema que aporte ayuda, documentación, plugines, etc.

Inconvenientes

- Mayor curva de aprendizaje.
- Posible sobrecarga de código innecesario.
- Dependencia de actualizaciones y comunidad.

¿Cuándo usar un framework en cliente y cuándo usar VanillaJS?

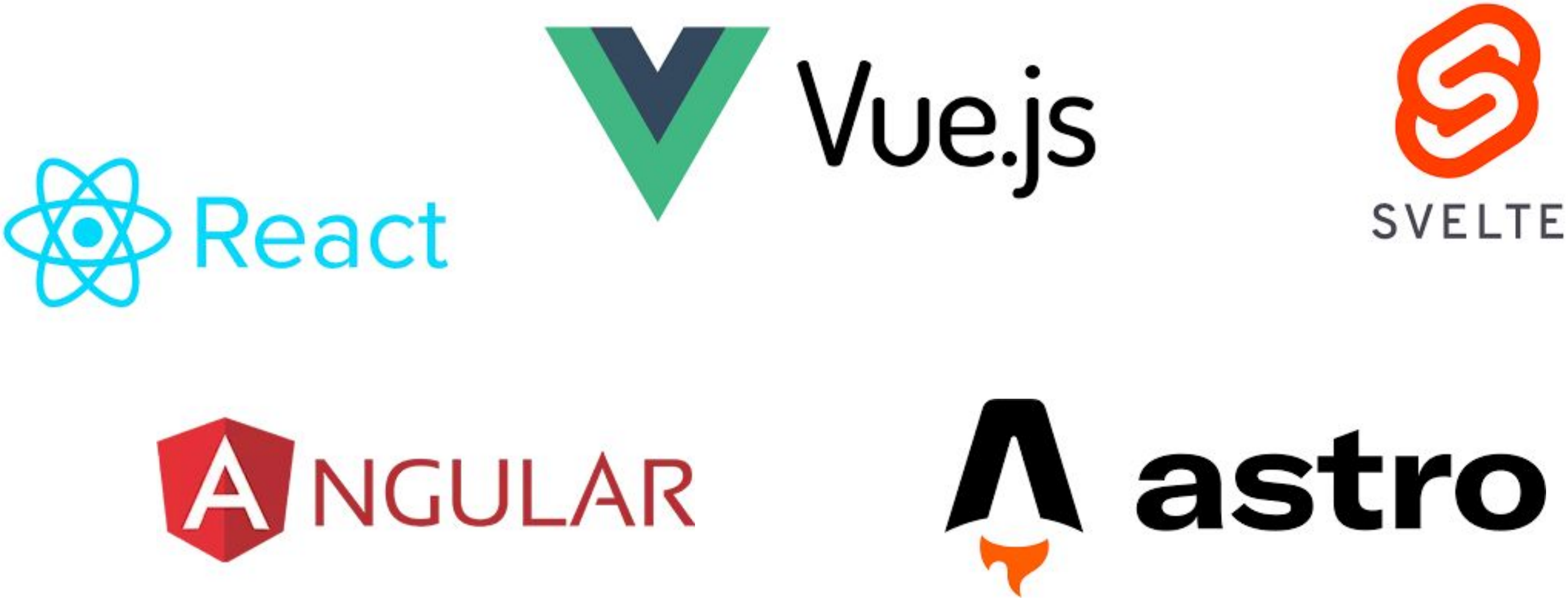
Características comunes en los frameworks

- **Interpolación de valores**
  permite insertar datos dinámicos en la vista de manera sencilla.
  React usa JSX ({valor}), Vue usa {{ valor }} y Angular usa {{ valor }}
- **Gestión del estado**
  mecanismos para gestionar el estado de la aplicación.
  React utiliza hooks como useState y librerías como Redux o Zustand; Angular utiliza servicios y NgRx; Vue tiene Vuex o Pinia.
- **Enrutamiento**
  manejan la navegación entre vistas "a su manera".
  React usa React Router, Angular usa Angular Router, y Vue usa Vue Router
- **Renderizado**
  optimización de la actualización de la interfaz de usuario.
  React usa Virtual DOM, Svelte compila directamente a JavaScript y Angular usa Change Detection

Características comunes en los frameworks

- **Componentes**
  todos los frameworks modernos fomentan la reutilización de código mediante componentes. Los componentes encapsulan la lógica y la interfaz; esto facilita la escalabilidad.
- **Manejo de eventos**
  los gestionan de manera declarativa.
  React usa onClick, Vue tiene @click, y Angular usa (click)
- **Binding bidireccional**
  Algunos frameworks como Angular y Vue permiten la sincronización automática entre la vista y el modelo de datos
  v-model en Vue, [(ngModel)] en Angular
- Soporte para **SSR** (Server-Side Rendering) y **SSG** (Static Site Generation)
  frameworks como Next.js y Nuxt.js permiten generar páginas estáticas o renderizadas en el servidor.
  Esto es muy útil para mejorar el SEO y el rendimiento.

Frameworks más usados actualmente



Otros frameworks

SOLID
ember
PREACT
The React Alternative
Inferno
Mithril
PHASER

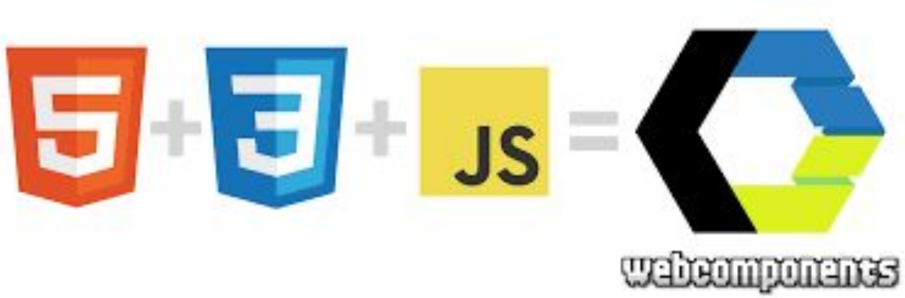
11ty
Gatsby
jekyll
HUGO
A Fast & Modern Static Website Engine

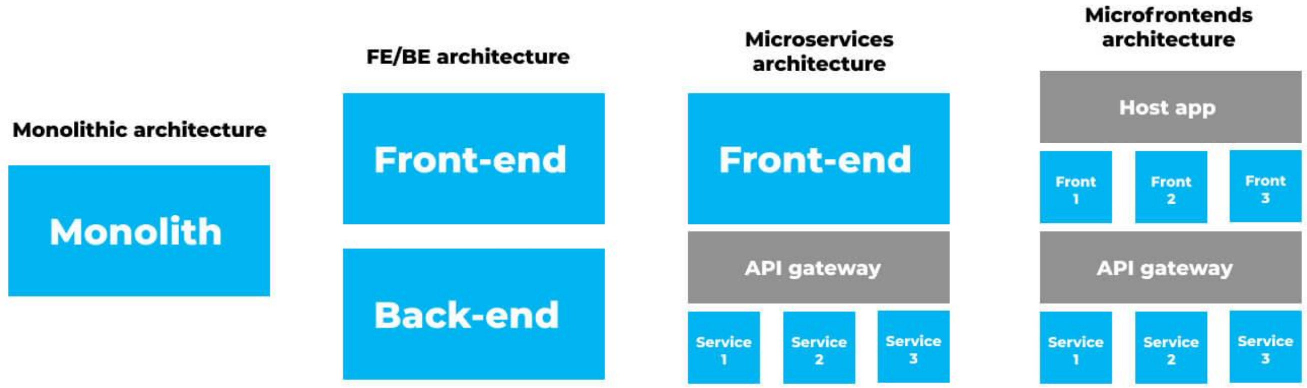
Otros frameworks

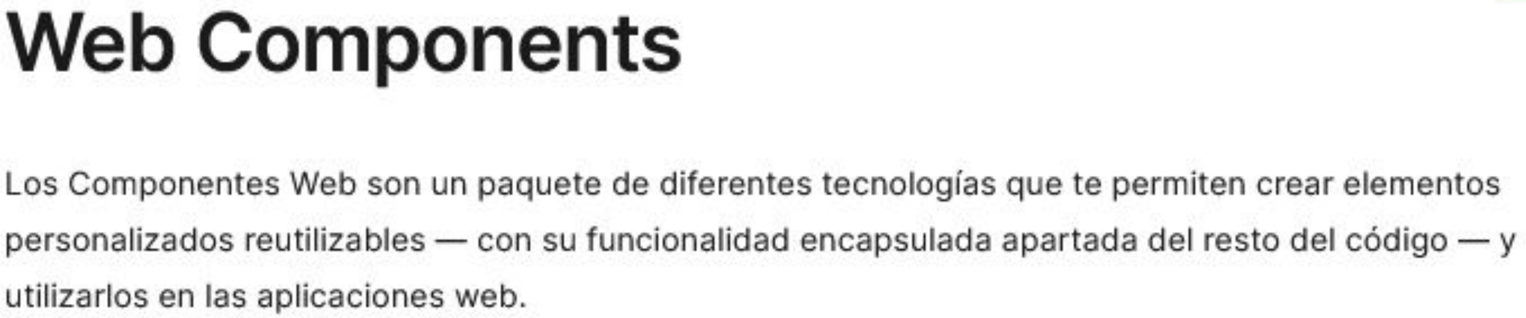
Y qué son los Micro Frontends?

Y qué son los WebComponents?

una arquitectura que permite dividir una aplicación web en partes más pequeñas e independientes, llamadas microfrontends. Cada microfrontend puede ser desarrollado, probado y desplegado por un equipo diferente, utilizando tecnologías y frameworks distintos.







Web Components

Los Componentes Web son un paquete de diferentes tecnologías que te permiten crear elementos personalizados reutilizables — con su funcionalidad encapsulada apartada del resto del código — y utilizarlos en las aplicaciones web.

Prueba de concepto: ANGULAR

Contexto

La primera versión de Angular se llamaba AngularJS.

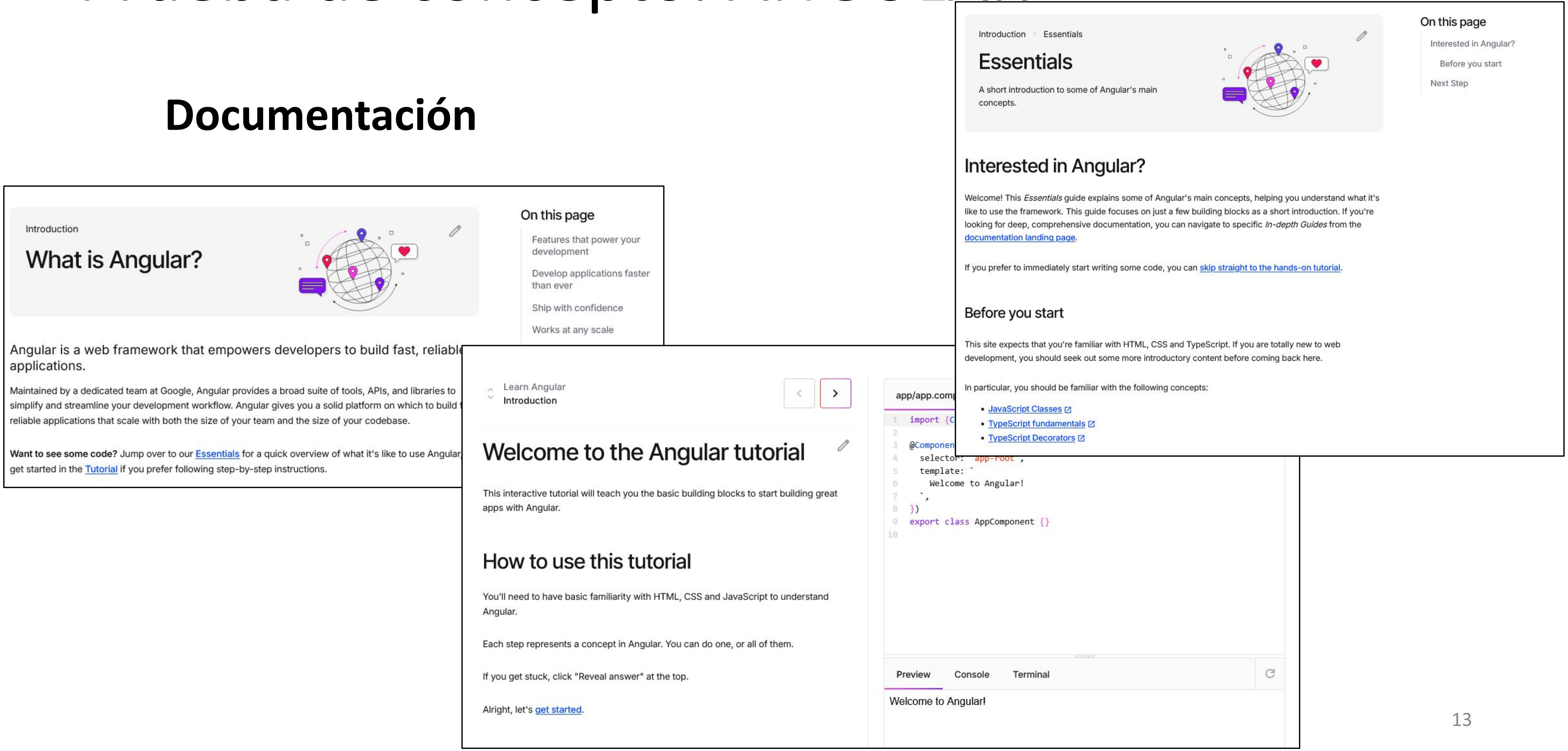
En 2016, Google lanzó Angular 2, una reescritura completa del framework. A partir de ahí se conoce simplemente como Angular.

a versión actual de Angular es la 19.

Se lanza una nueva versión cada seis meses, aproximadamente.

Prueba de concepto: ANGULAR

Documentación



Introduction

What is Angular?

On this page

- Features that power your development
- Develop applications faster than ever
- Ship with confidence
- Works at any scale

Angular is a web framework that empowers developers to build fast, reliable applications.

Maintained by a dedicated team at Google, Angular provides a broad suite of tools, APIs, and libraries to simplify and streamline your development workflow. Angular gives you a solid platform on which to build f reliable applications that scale with both the size of your team and the size of your codebase.

Want to see some code? Jump over to our Essentials for a quick overview of what it's like to use Angular, get started in the Tutorial if you prefer following step-by-step instructions.

Learn Angular
Introduction

Welcome to the Angular tutorial

This interactive tutorial will teach you the basic building blocks to start building great apps with Angular.

How to use this tutorial

You'll need to have basic familiarity with HTML, CSS and JavaScript to understand Angular.

Each step represents a concept in Angular. You can do one, or all of them.

If you get stuck, click "Reveal answer" at the top.

Alright, let's get started.

app/app.comp

```
import {C

@Componen
  selector: 'app-root',
  template: `
    Welcome to Angular!
  `,
})
export class AppComponent {}
```

Welcome to Angular!

Introduction › Essentials

Essentials

A short introduction to some of Angular's main concepts.

On this page

- Interested in Angular?
  - Before you start
- Next Step

Interested in Angular?

Welcome! This *Essentials* guide explains some of Angular's main concepts, helping you understand what it's like to use the framework. This guide focuses on just a few building blocks as a short introduction. If you're looking for deep, comprehensive documentation, you can navigate to specific *In-depth Guides* from the documentation landing page.

If you prefer to immediately start writing some code, you can skip straight to the hands-on tutorial.

Before you start

This site expects that you're familiar with HTML, CSS and TypeScript. If you are totally new to web development, you should seek out some more introductory content before coming back here.

In particular, you should be familiar with the following concepts:

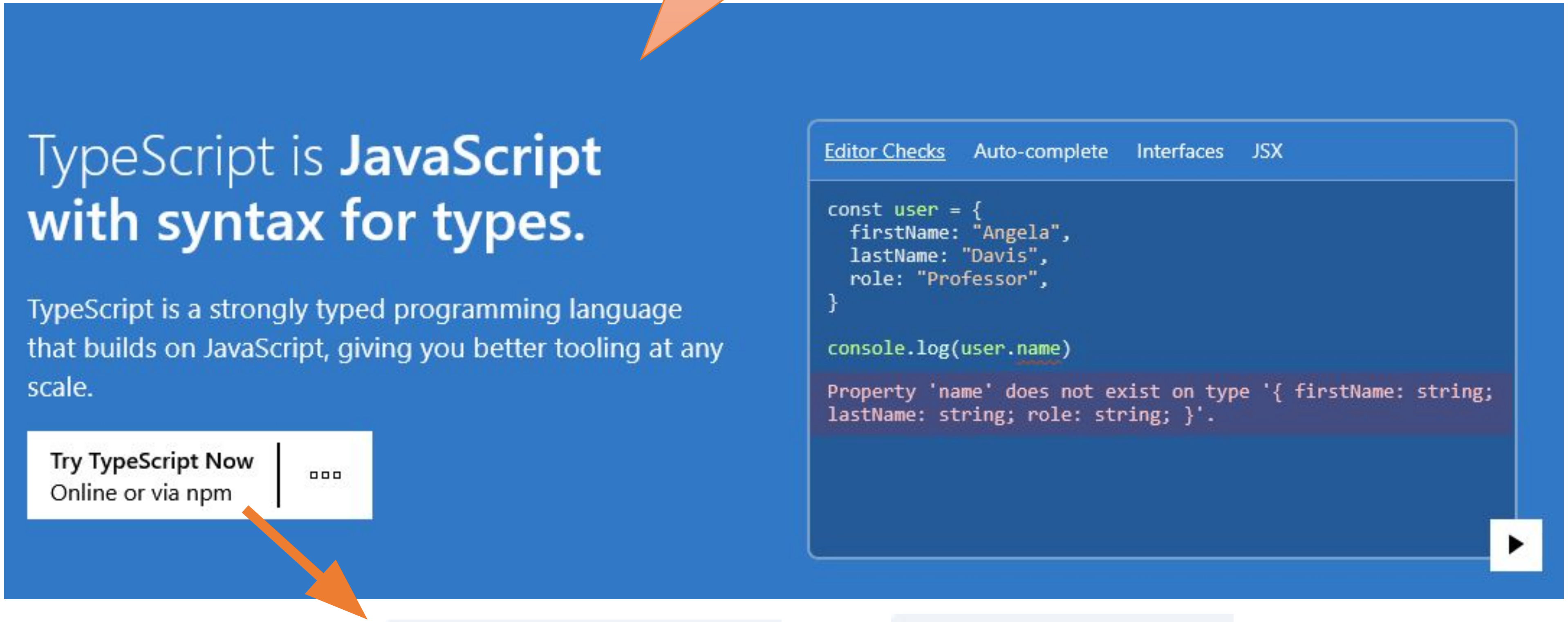
- JavaScript Classes
- TypeScript fundamentals
- TypeScript Decorators

Prueba de concepto: ANGULAR

Hace posible la detección temprana de errores, Mejora la legibilidad, el mantenimiento del código y, en general, la experiencia de desarrollo

¿Por qué?

TS TypeScript

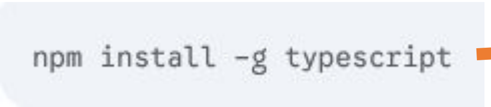


TypeScript is **JavaScript with syntax for types.**

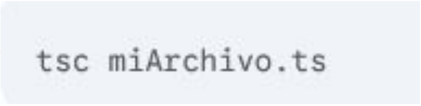
TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.

```
const user = {
  firstName: "Angela",
  lastName: "Davis",
  role: "Professor",
}

console.log(user.name)
```

```
Property 'name' does not exist on type '{ firstName: string;
lastName: string; role: string; }'.
```



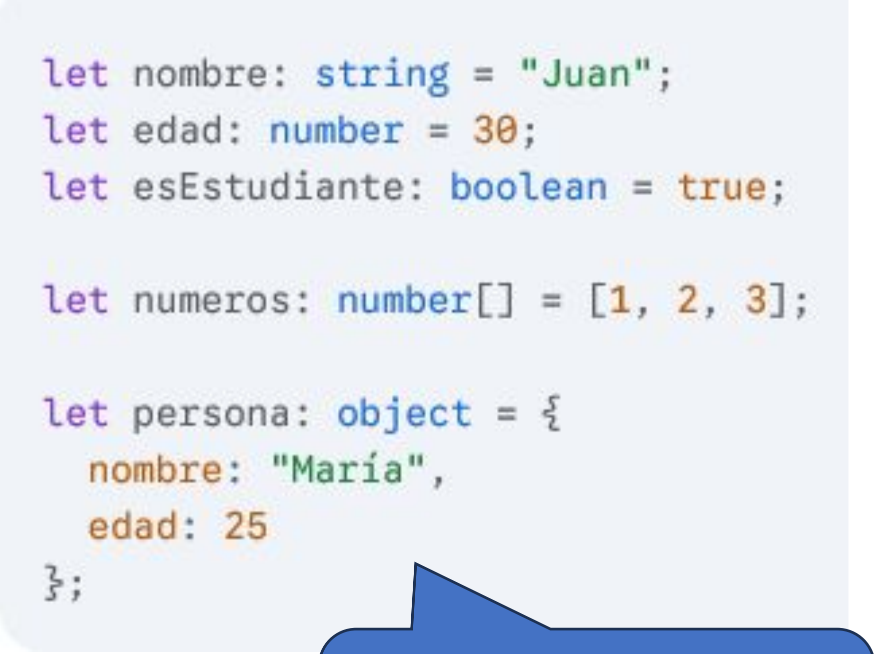
```
npm install -g typescript
```



```
tsc miArchivo.ts
```

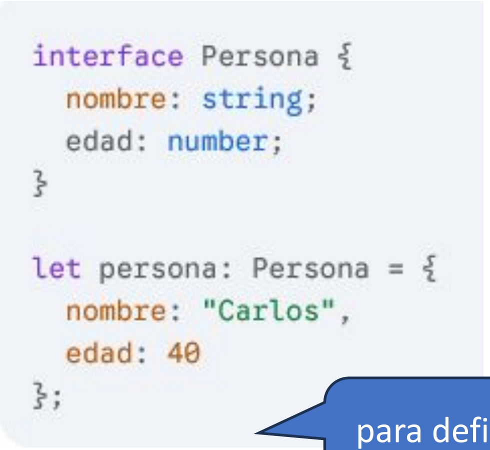
Prueba de concepto: ANGULAR

TS TypeScript



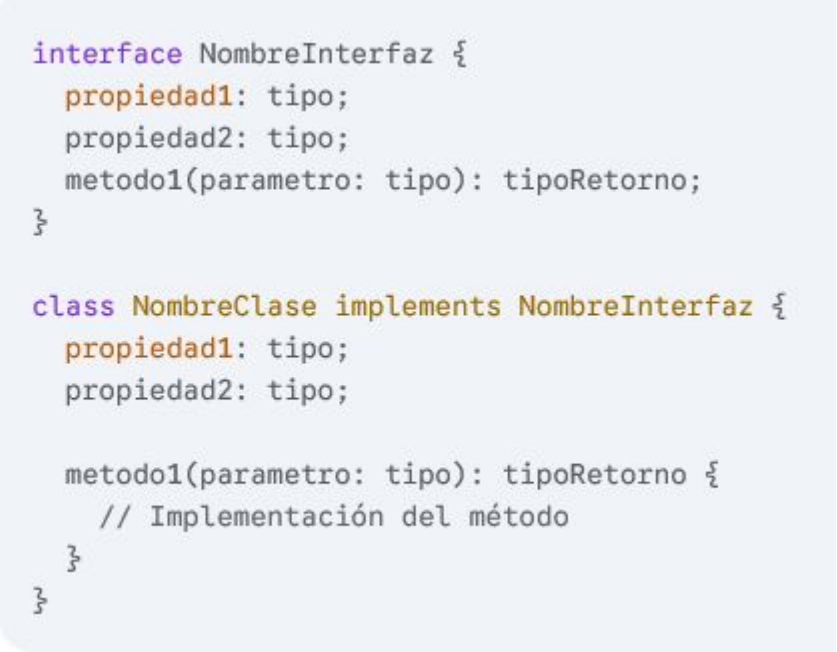
```
let nombre: string = "Juan";
let edad: number = 30;
let esEstudiante: boolean = true;

let numeros: number[] = [1, 2, 3];

let persona: object = {
  nombre: "María",
  edad: 25
};
```

tipos básicos
(+ tipo especial any)
(string | number) unknown



```
interface Persona {
  nombre: string;
  edad: number;
}

let persona: Persona = {
  nombre: "Carlos",
  edad: 40
};
```

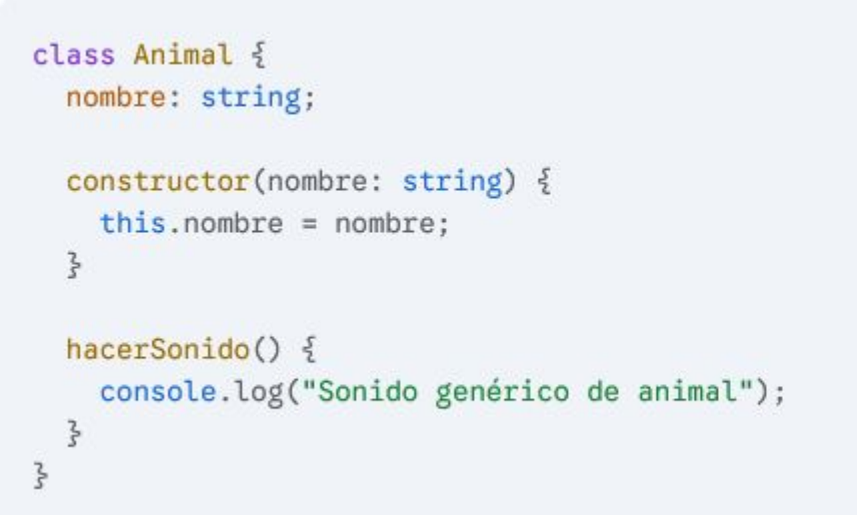
para definir la forma de los objetos



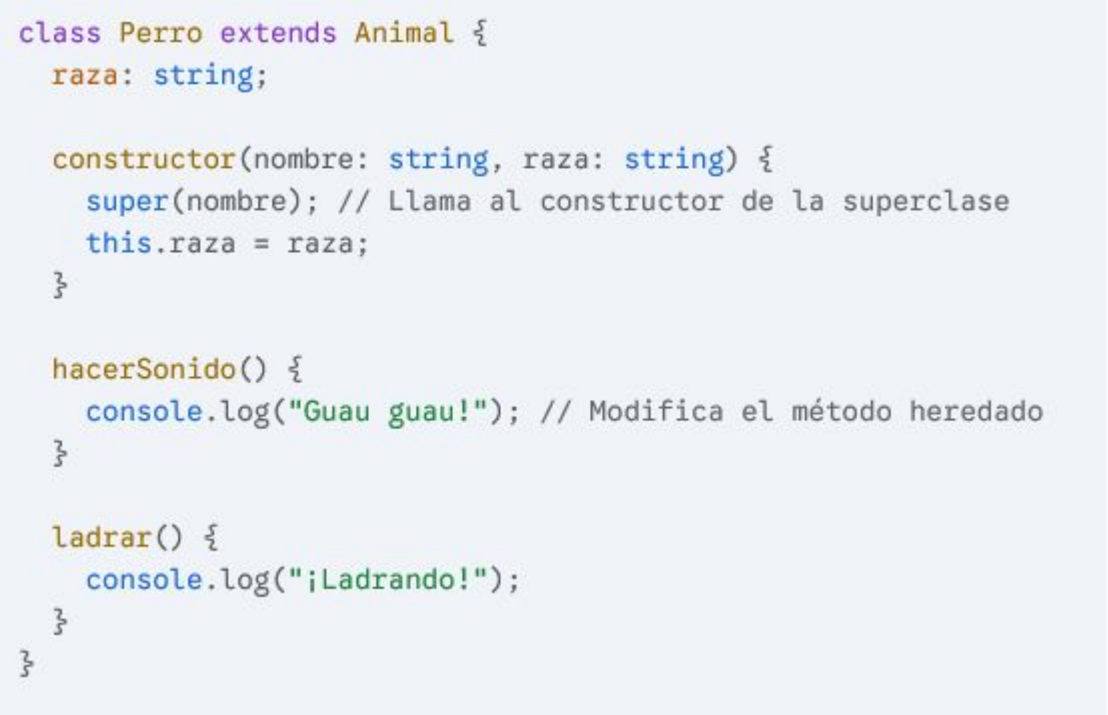
```
interface NombreInterfaz {
  propiedad1: tipo;
  propiedad2: tipo;
  metodo1(parametro: tipo): tipoRetorno;
}

class NombreClase implements NombreInterfaz {
  propiedad1: tipo;
  propiedad2: tipo;

  metodo1(parametro: tipo): tipoRetorno {
    // Implementación del método
  }
}
```

Prueba de concepto: ANGULAR

TS TypeScript



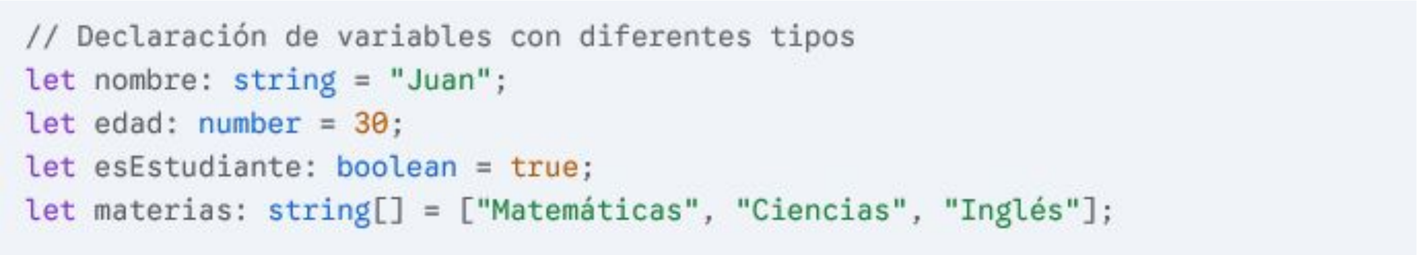
```
class Animal {
  nombre: string;

  constructor(nombre: string) {
    this.nombre = nombre;
  }

  hacerSonido() {
    console.log("Sonido genérico de animal");
  }
}
```



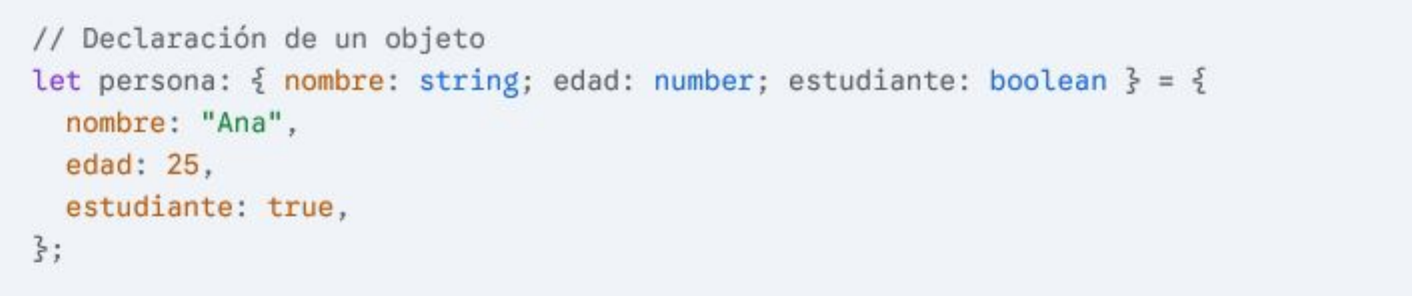
```
class Perro extends Animal {
  raza: string;

  constructor(nombre: string, raza: string) {
    super(nombre); // Llama al constructor de la superclase
    this.raza = raza;
  }

  hacerSonido() {
    console.log("Guau guau!"); // Modifica el método heredado
  }

  ladrar() {
    console.log("¡Ladrando!");
  }
}
```

Prueba de concepto: ANGULAR

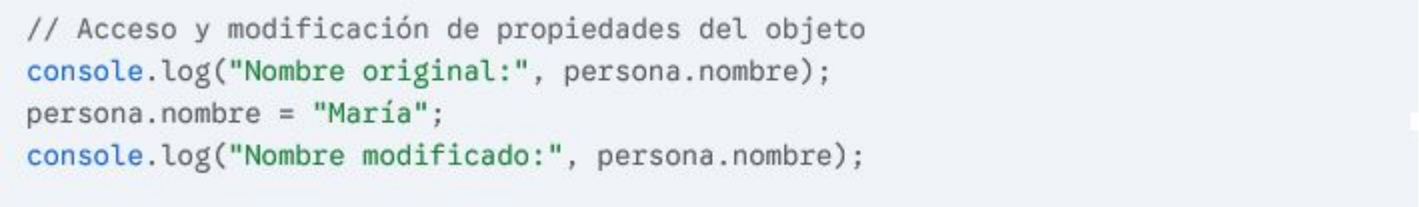
TypeScript



```
// Declaración de variables con diferentes tipos
let nombre: string = "Juan";
let edad: number = 30;
let esEstudiante: boolean = true;
let materias: string[] = ["Matemáticas", "Ciencias", "Inglés"];
```



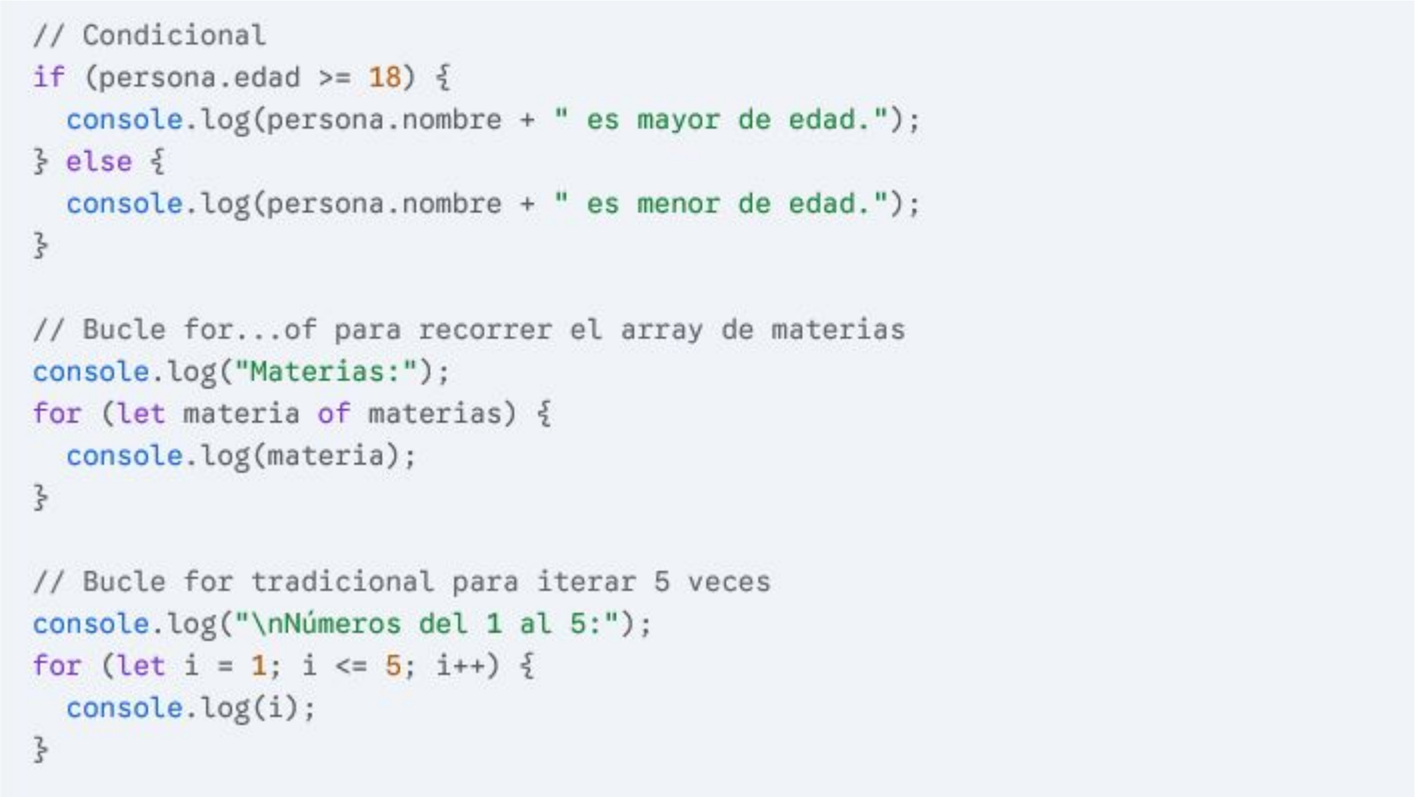
```
// Declaración de un objeto
let persona: { nombre: string; edad: number; estudiante: boolean } = {
  nombre: "Ana",
  edad: 25,
  estudiante: true,
};
```



```
// Acceso y modificación de propiedades del objeto
console.log("Nombre original:", persona.nombre);
persona.nombre = "María";
console.log("Nombre modificado:", persona.nombre);
```

Prueba de concepto: ANGULAR

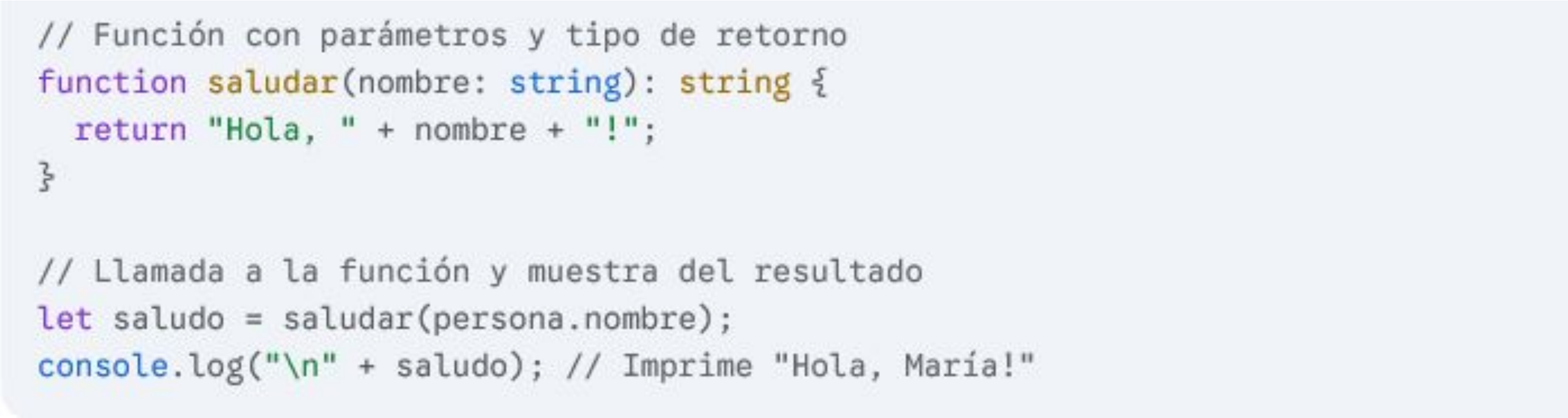
TypeScript



```
// Condicional
if (persona.edad >= 18) {
  console.log(persona.nombre + " es mayor de edad.");
} else {
  console.log(persona.nombre + " es menor de edad.");
}

// Bucle for...of para recorrer el array de materias
console.log("Materias:");
for (let materia of materias) {
  console.log(materia);
}

// Bucle for tradicional para iterar 5 veces
console.log("\nNúmeros del 1 al 5:");
for (let i = 1; i <= 5; i++) {
  console.log(i);
}
```

Prueba de concepto: ANGULAR

TS TypeScript



```
// Función con parámetros y tipo de retorno
function saludar(nombre: string): string {
  return "Hola, " + nombre + "!";
}

// Llamada a la función y muestra del resultado
let saludo = saludar(persona.nombre);
console.log("\n" + saludo); // Imprime "Hola, María!"
```

TypeScript
Cheat Sheet

Control Flow Analysis

Key points

CFA nearly always takes a union and reduces the number of types inside the union based on logic in your code.

Most of the time CFA works inside natural JavaScript boolean logic, but there are ways to define your own functions which affect how TypeScript narrows types.

If Statements

Most narrowing comes from expressions inside if statements, where different type operators narrow inside the new scope

typeof (for primitives)

```
const input = getUserInput()
input // string | number

if (typeof input === "string") {
   input // string
}
```

"property" in object (for objects)

```
const input = getUserInput()
input // string | { error: ... }

if ("error" in input) {
   input // { error: ... }
}
```

instanceof (for classes)

```
const input = getUserInput()
input // number | number[]

if (input instanceof Array) {
   input // number[]
}
```

type-guard functions (for anything)

```
const input = getUserInput()
input // number | number[]

if (Array.isArray(input)) {
   input // number[]
}
```

Expressions

Narrowing also occurs on the same line as code, when doing boolean operations

```
const input = getUserInput()
input // string | number

const inputLength =
   (typeof input === "string" && input.length) || input
                                  // input: string
```

Discriminated Unions

```
type Responses =
   | { status: 200, data: any }
   | { status: 301, to: string }
   | { status: 400, error: Error }
```

All members of the union have the same property name, CFA can discriminate on that.

Usage

```
const response = getResponse()
response // Responses

switch(response.status) {
  case 200: return response.data
  case 301: return redirect(response.to)
  case 400: return response.error
}
```

Assignment

Narrowing types using 'as const'

Subfields in objects are treated as though they can be mutated, and during assignment the type will be 'widened' to a non-literal version. The prefix 'as const' locks all types to their literal versions.

```
const data1 = {                typeof data1 = {
   name: "Zagreus"    »           name: string
}                              }

const data2 = {                typeof data2 = {
   name: "Zagreus"    »           name: "Zagreus"
} as const                     }
```

Tracks through related variables

```
const response = getResponse()
const isSuccessResponse
    = res instanceof SuccessResponse

if (isSuccessResponse)
  res.data // SuccessResponse
```

Re-assignment updates types

```
let data: string | number = ...
data // string | number
data = "Hello"
data // string
```

Type Guards

A function with a return type describing the CFA change for a new scope when it is true.

```
function isErrorResponse(obj: Response): obj is APIErrorResponse {
   return obj instanceof APIErrorResponse
}
```

Return type position describes what the assertion is

Usage

```
const response = getResponse()
response // Response | APIErrorResponse

if (isErrorResponse(response)) {
   response // APIErrorResponse
}
```

Assertion Functions

A function describing CFA changes affecting the current scope, because it throws instead of returning false.

```
function assertResponse(obj: any): asserts obj is SuccessResponse {
  if (!(obj instanceof SuccessResponse)) {
    throw new Error("Not a success!")
  }
}
```

Usage

```
const res = getResponse():
res // SuccessResponse | ErrorResponse

assertResponse(res)

res // SuccessResponse
```

Assertion functions change the *current* scope or throw

TypeScript
Cheat Sheet

Interface

Key points

Used to describe the shape of objects, and can be extended by others.

Almost everything in JavaScript is an object and `interface` is built to match their runtime behavior.

Built-in Type Primitives

```
boolean, string, number,
undefined, null, any,
unknown, never, void,
bigint, symbol
```

Common Built-in JS Objects

```
Date, Error, Array, Map,
Set, Regexp, Promise
```

Type Literals

Object:
`{ field: string }`
Function:
`(arg: number) => string`
Arrays:
`string[] or Array<string>`
Tuple:
`[string, number]`

Avoid

Object, String, Number, Boolean

Common Syntax

```
interface JSONResponse extends Response, HTTPAble {
  version: number;

  /** In bytes */
  payloadSize: number;

  outOfStock?: boolean;

  update: (retryTimes: number) => void;
  update(retryTimes: number): void;

  (): JSONResponse

  new(s: string): JSONResponse;

  [key: string]: number;

  readonly body: string;
}
```

Optionally take properties from existing interface or type

JSDoc comment attached to show in editors

This property might not be on the object

These are two ways to describe a property which is a function

You can call this object via () - (functions in JS are objects which can be called)

You can use new on the object this interface describes

Any property not described already is assumed to exist, and all properties must be numbers

Tells TypeScript that a property can not be changed

Generics

Declare a type which can change in your interface

```
interface APICall<Response> {
  data: Response
}
```

Type parameter

Used here

Usage

```
const api: APICall<ArtworkCall> = ...
api.data // Artwork
```

You can constrain what types are accepted into the generic parameter via the extends keyword.

```
interface APICall<Response extends { status: number }> {
  data: Response
}

const api: APICall<ArtworkCall> = ...
api.data.status
```

Sets a constraint on the type which means only types with a 'status' property can be used

Overloads

A callable interface can have multiple definitions for different sets of parameters

```
interface Expect {
    (matcher: boolean): string
    (matcher: string): boolean;
}
```

Get & Set

Objects can have custom getters or setters

```
interface Ruler {
    get size(): number
    set size(value: number | string);
}
```

Usage

```
const r: Ruler = ...
r.size = 12
r.size = "36"
```

Extension via merging

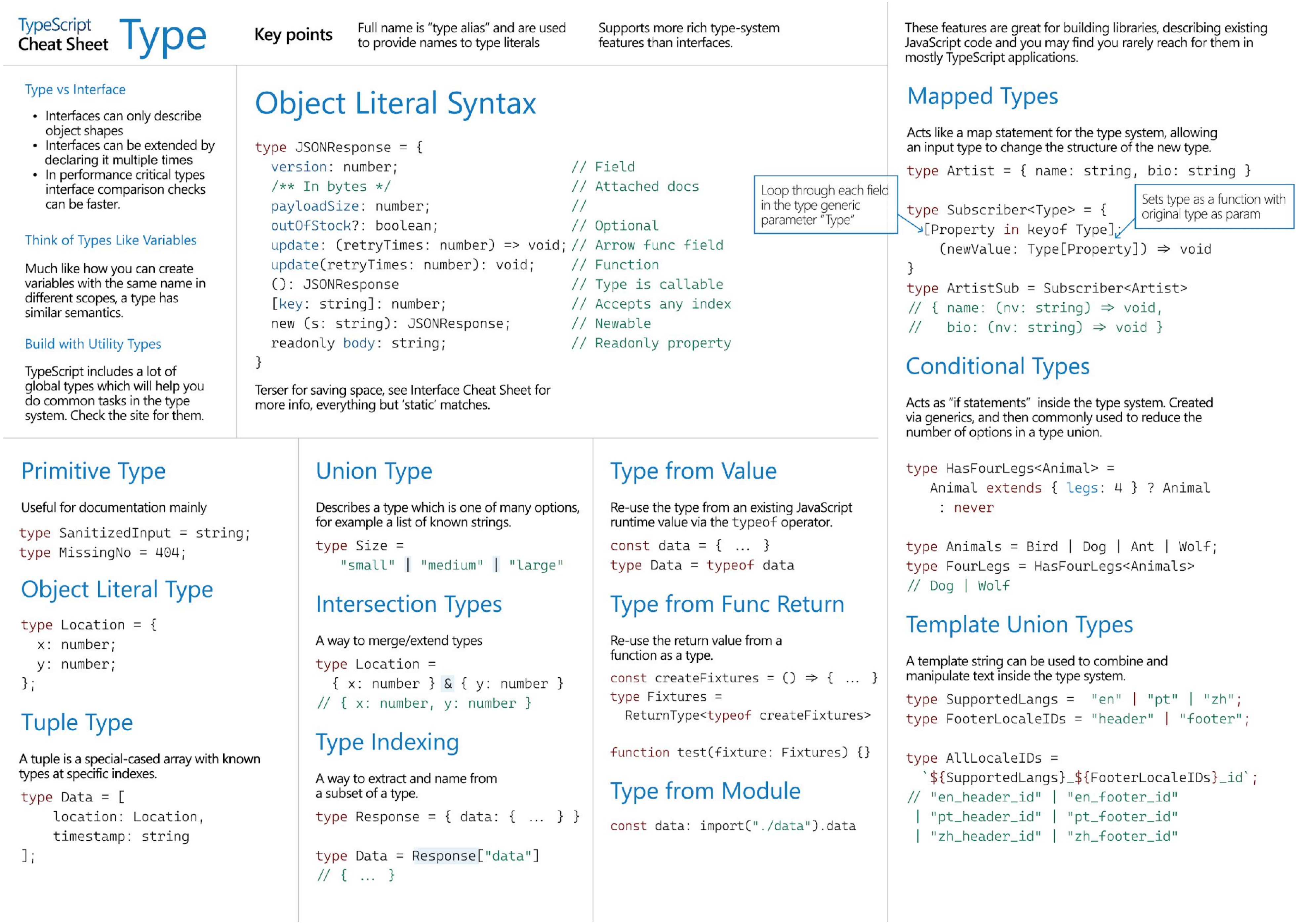
Interfaces are merged, so multiple declarations will add new fields to the type definition.

```
interface APICall {
  data: Response
}

interface APICall {
  error?: Error
}
```

Class conformance

You can ensure a class conforms to an interface via `implements`:

```
interface Syncable { sync(): void }
class Account implements Syncable { ... }
```



TypeScript
Cheat Sheet **Type**

Key points

Full name is "type alias" and are used to provide names to type literals

Supports more rich type-system features than interfaces.

These features are great for building libraries, describing existing JavaScript code and you may find you rarely reach for them in mostly TypeScript applications.

Type vs Interface

- Interfaces can only describe object shapes
- Interfaces can be extended by declaring it multiple times
- In performance critical types interface comparison checks can be faster.

Think of Types Like Variables

Much like how you can create variables with the same name in different scopes, a type has similar semantics.

Build with Utility Types

TypeScript includes a lot of global types which will help you do common tasks in the type system. Check the site for them.

Object Literal Syntax

```
type JSONResponse = {
  version: number;                        // Field
  /** In bytes */                         // Attached docs
  payloadSize: number;                    //
  outOfStock?: boolean;                   // Optional
  update: (retryTimes: number) => void; // Arrow func field
  update(retryTimes: number): void;      // Function
  (): JSONResponse                        // Type is callable
  [key: string]: number;                  // Accepts any index
  new (s: string): JSONResponse;          // Newable
  readonly body: string;                  // Readonly property
}
```

Terser for saving space, see Interface Cheat Sheet for more info, everything but 'static' matches.

Mapped Types

Acts like a map statement for the type system, allowing an input type to change the structure of the new type.

```
type Artist = { name: string, bio: string }

type Subscriber<Type> = {
  [Property in keyof Type]:
    (newValue: Type[Property]) ⇒ void
}
type ArtistSub = Subscriber<Artist>
// { name: (nv: string) ⇒ void,
//   bio: (nv: string) ⇒ void }
```

Loop through each field in the type generic parameter "Type"

Sets type as a function with original type as param

Conditional Types

Acts as "if statements" inside the type system. Created via generics, and then commonly used to reduce the number of options in a type union.

```
type HasFourLegs<Animal> =
   Animal extends { legs: 4 } ? Animal
     : never

type Animals = Bird | Dog | Ant | Wolf;
type FourLegs = HasFourLegs<Animals>
// Dog | Wolf
```

Primitive Type

Useful for documentation mainly

```
type SanitizedInput = string;
type MissingNo = 404;
```

Object Literal Type

```
type Location = {
  x: number;
  y: number;
};
```

Tuple Type

A tuple is a special-cased array with known types at specific indexes.

```
type Data = [
    location: Location,
    timestamp: string
];
```

Union Type

Describes a type which is one of many options, for example a list of known strings.

```
type Size =
   "small" | "medium" | "large"
```

Intersection Types

A way to merge/extend types

```
type Location =
  { x: number } & { y: number }
// { x: number, y: number }
```

Type Indexing

A way to extract and name from a subset of a type.

```
type Response = { data: { … } }

type Data = Response["data"]
// { … }
```

Type from Value

Re-use the type from an existing JavaScript runtime value via the `typeof` operator.

```
const data = { … }
type Data = typeof data
```

Type from Func Return

Re-use the return value from a function as a type.

```
const createFixtures = () ⇒ { … }
type Fixtures =
  ReturnType<typeof createFixtures>

function test(fixture: Fixtures) {}
```

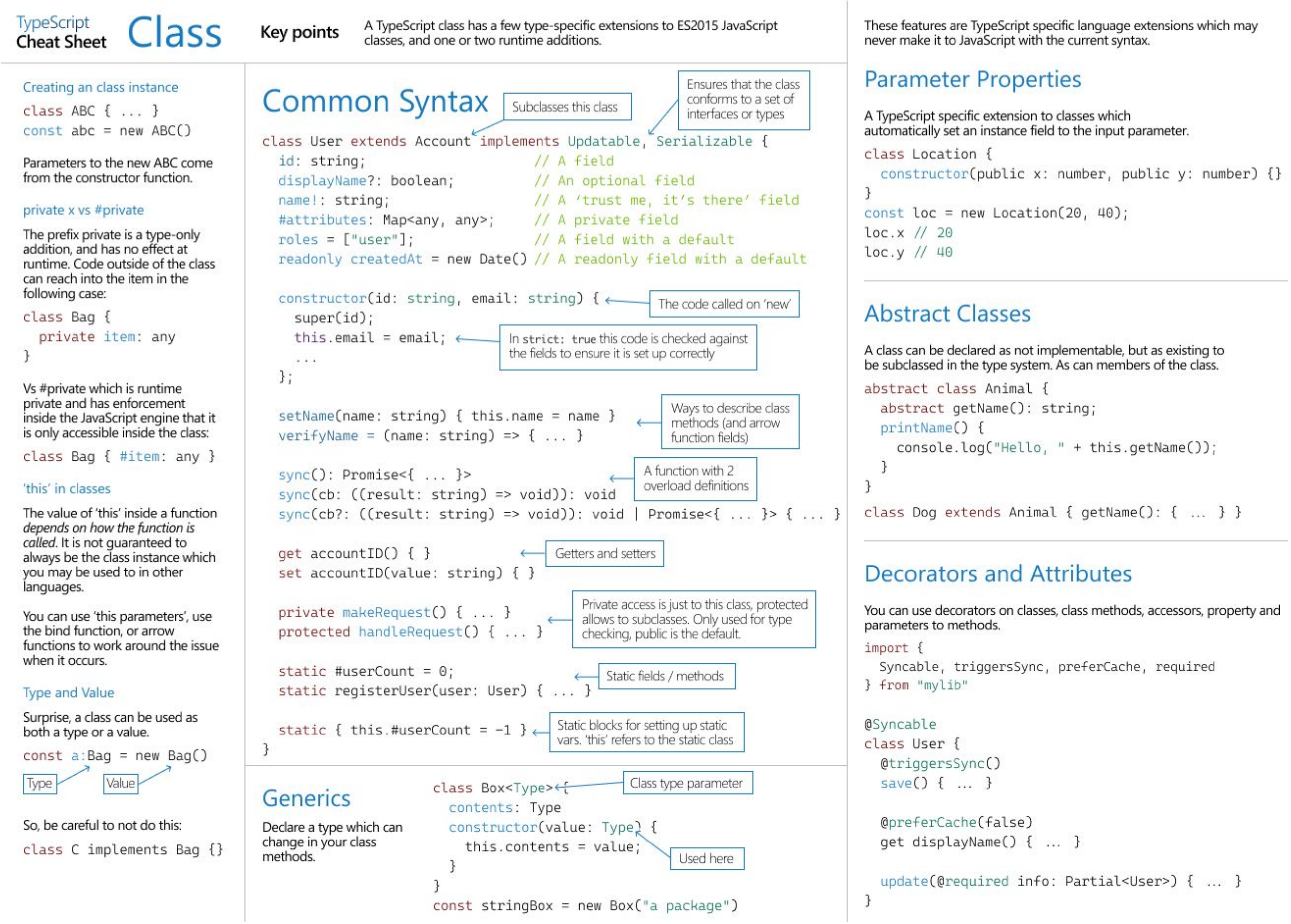
Type from Module

```
const data: import("./data").data
```

Template Union Types

A template string can be used to combine and manipulate text inside the type system.

```
type SupportedLangs =  "en" | "pt" | "zh";
type FooterLocaleIDs = "header" | "footer";

type AllLocaleIDs =
  `${SupportedLangs}_${FooterLocaleIDs}_id`;
// "en_header_id" | "en_footer_id"
 | "pt_header_id" | "pt_footer_id"
 | "zh_header_id" | "zh_footer_id"
```



TypeScript Cheat Sheet

Class

Key points

A TypeScript class has a few type-specific extensions to ES2015 JavaScript classes, and one or two runtime additions.

Creating an class instance

```
class ABC { ... }
const abc = new ABC()
```

Parameters to the new ABC come from the constructor function.

private x vs #private

The prefix private is a type-only addition, and has no effect at runtime. Code outside of the class can reach into the item in the following case:

```
class Bag {
  private item: any
}
```

Vs #private which is runtime private and has enforcement inside the JavaScript engine that it is only accessible inside the class:

```
class Bag { #item: any }
```

'this' in classes

The value of 'this' inside a function *depends on how the function is called.* It is not guaranteed to always be the class instance which you may be used to in other languages.

You can use 'this parameters', use the bind function, or arrow functions to work around the issue when it occurs.

Type and Value

Surprise, a class can be used as both a type or a value.

```
const a:Bag = new Bag()
```

Type

Value

So, be careful to not do this:

```
class C implements Bag {}
```

Common Syntax

Subclasses this class

Ensures that the class conforms to a set of interfaces or types

```
class User extends Account implements Updatable, Serializable {
  id: string;                     // A field
  displayName?: boolean;          // An optional field
  name!: string;                  // A 'trust me, it's there' field
  #attributes: Map<any, any>;     // A private field
  roles = ["user"];               // A field with a default
  readonly createdAt = new Date() // A readonly field with a default

  constructor(id: string, email: string) {
    super(id);
    this.email = email;
    ...
  };

  setName(name: string) { this.name = name }
  verifyName = (name: string) => { ... }

  sync(): Promise<{ ... }>
  sync(cb: ((result: string) => void)): void
  sync(cb?: ((result: string) => void)): void | Promise<{ ... }> { ... }

  get accountID() { }
  set accountID(value: string) { }

  private makeRequest() { ... }
  protected handleRequest() { ... }

  static #userCount = 0;
  static registerUser(user: User) { ... }

  static { this.#userCount = -1 }
}
```

The code called on 'new'

In `strict: true` this code is checked against the fields to ensure it is set up correctly

Ways to describe class methods (and arrow function fields)

A function with 2 overload definitions

Getters and setters

Private access is just to this class, protected allows to subclasses. Only used for type checking, public is the default.

Static fields / methods

Static blocks for setting up static vars. 'this' refers to the static class

Generics

Declare a type which can change in your class methods.

```
class Box<Type> {
  contents: Type
  constructor(value: Type) {
    this.contents = value;
  }
}
const stringBox = new Box("a package")
```

Class type parameter

Used here

These features are TypeScript specific language extensions which may never make it to JavaScript with the current syntax.

Parameter Properties

A TypeScript specific extension to classes which automatically set an instance field to the input parameter.

```
class Location {
  constructor(public x: number, public y: number) {}
}
const loc = new Location(20, 40);
loc.x // 20
loc.y // 40
```

Abstract Classes

A class can be declared as not implementable, but as existing to be subclassed in the type system. As can members of the class.

```
abstract class Animal {
  abstract getName(): string;
  printName() {
    console.log("Hello, " + this.getName());
  }
}
class Dog extends Animal { getName(): { ... } }
```

Decorators and Attributes

You can use decorators on classes, class methods, accessors, property and parameters to methods.

```
import {
  Syncable, triggersSync, preferCache, required
} from "mylib"

@Syncable
class User {
  @triggersSync()
  save() { ... }

  @preferCache(false)
  get displayName() { ... }

  update(@required info: Partial<User>) { ... }
}
```

Prueba de concepto: ANGULAR