Introducción

Introducción a TypeScript

Después de cubrir los principales conceptos de JavaScript, es hora de introducir TypeScript en nuestro "stack" tecnológico. No sólo porque en menos de 5 años se haya colocado como el 4º lenguaje más popular entre programadores, si no porque añade una serie de funcionalidades extra a JavaScript que pueden mejorar significativamente el desarrollo de nuestros proyectos.

Overview

La funcionalidad principal que TypeScript añade con respecto a JavaScript es la **comprobación estática de código**. JavaScript es dinámico, y es por ello que es muy flexible y ágil a la hora de prototipar, el problema es que puedes encontrate con muchos posibles puntos de conflicto:

- Llamada a objetos que no existen

```
objetoSinDefinir.llamadaMetodo();
```

- Parámetros de función de diferentes tipos

```
function iteradorArray(iterador) {
    for (item in iterador) {
        console.log(item);
    }
}

iteradorArray(3);
```

- Atributos que no existen en objetos

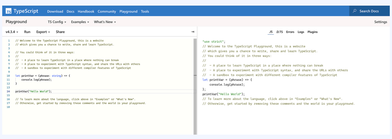
```
let person = {name: "Lucas Fernandez", age:27 };

console.log(person.profession);
```

- Imports mal realizados

```
import { Feature } from "incorrect/path";
```

Al ser JavaScript un lenguaje interpretado, todo el código será ejecutado en su entorno (si es web será en el motor del navegador), y estos errores **no aparecerán hasta el tiempo de ejecución**. Eso supone que muchos errores no aparecerán hasta que sean recogidos por tests, por pruebas reales en navegador o incluso peor, **EN PRODUCCIÓN**. La razón de que TypeScript detecte estos problemas es porque tiene un compilador previo, que transforma el código de TypeScript a JavaScript. Podemos hacer esto mismo simplemente llamando al compilador con `tsc file.js`.



Consola TypeScript Online

TypeScript es un super-set de JavaScript, esto significa que podemos seguir escribiendo código en JavaScript mientras implementamos nuestro programa, permitiendo ser flexibles en el desarrollo mientras vamos teniendo desde el día 0 las ventajas que proporciona TypeScript, como el **auto-completado**, la **auto-importación de módulos** o la **comprobación de tipos**.

Instalación

La instalación de TypeScript es muy sencilla, solo es necesario tener Node instalado en el sistema. Una vez hecho, simplemente hay que ejecutar el siguiente comando:

```
> npm install -g typescript
```

Ahora podemos ejecutar sin ningún problema nuestro código. En este caso, tenemos un fichero llamado `index.ts` que tendremos que compilar a JavaScript, para ello ejecutremos lo siguiente:

```
> tsc index.ts
```

Esto generará un fichero llamado `index.js` que podrá ser utilizado por el html de la página.

Algunas Ventajas frente a JavaScript

Compilación a distintas versiones

Una de las grandes ventajas de TypeScript es que podemos compilar a la versión de JavaScript que elijamos. Todo esto se puede configurar en el fichero tsconfig.json, pero también se puede realizar mediante la línea de comandos.

```
> tsc index.ts # default execution
> tsc --help # display all the info
> tsc index.ts -t ES5 # target TS5 code
> tsc index.ts -t ES6 # target TS6 code
> tsc async.ts # Converts ES6 async await to ES5
```

Esto permite utilizar las últimas características de ES6 que vimos, como **arrow functions, optional chaining y nullish operator** pudiendo luego ejecutarlo en navegadores antiguos.

Tipado estáticos

Al final es la característica estrella de TypeScript es el tipado estático. Esto permite asignar a variables, objetos y funciones de tipos, con los que se comprobará luego en tiempo de compilación si el valor asignado corresponde con la definición inicial **evitando así potenciales errores en ejecución**.

```
let studentName: string;

studentName = 23; // Compiler Error
```

También TypeScript tendrá tipado implícito, esto significa que si no se le añade un tipo pero sí un valor en la declaración, TypeScript inferirá el valor y lo tipará automáticamente.

```
let studentName = "Pepe";

studentName = 23; // Error
```

Interfaces

Ahora hablaremos más en detenimiento de las **interfaces** en el siguiente apartado, pero nos sirve para ejemplificar el siguiente punto. Las interfaces permiten comprobar la forma que los valores asignados a las variables deben de tener. Esto también se puede llamar duck typing o structural subtyping y básicamente permite declarar "contratos" con el resto de tu código o con código externo para seguir un modelo.

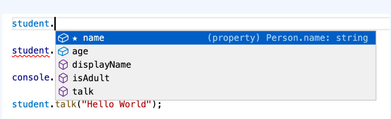
```
interface Person {
    name: string,
    age: number
    displayName(): void;
    talk(phrase: string);
    isAdult(): boolean;
};

interface RestResponse {
    persons: [Person],
    insitutionName: string,
    year: number
}
```

Esto permite por ejemplo tener un modelo bien definido para **comunicación rest**, añadir tipos a objetos o argumentos de funciones para tener un correcto control o incluso tener control sobre el parsing JSON de las respuestas a peticiones web.

Autocompletado

En algunos IDEs como Visusal Studio Code, si descargamos la extensión de TypeScript (o en las versiones más modernas ya viene por defecto), podremos de disfrutar de algunas ventajas muy potentes, como por ejemplo el **autocompletado de objetos o variables** (gracias a la comprobación estática), el **poder auto-importar de otros ficheros automáticamente** o el poder visualizar el tsdoc de un objeto o método comentado.



Flexibilidad en la comprobación de tipos

Hay momentos en los que, bien por compatibilidad con otras librerías, bien por la complejidad de los tipos o por alguna otra razón, se quiere obviar los tipos o directamente escribir el código en JavaScript. Esto es posible de distintas maneras.

```
let studentName: any = "Pepe"; // Be careful, use with precaution

studentName = 23; // It will work great
```

Como se puede ver el tipo **any** permite que la variable actúe de la misma manera que en JavaScript. Esto, unido a que podemos seguir usando la sintaxis de JavaScript, nos permite tener cierta flexibilidad a la hora de desarrollar ciertos componentes que pueden ser difíciles de implementar en primer lugar con TypeScript.

Tipos

Introducción a los tipos de TypeScript

En esta sección vamos a ver como TypeScript gestiona los tipos y como podemos usar una de las características más importantes de este lenguaje.

Type Annotation

La funcionalidad principal de TypeScript es añadir tipos a elementos de JavaScript como variables, funciones, objetos... Para ello, usa la sintaxis `elemento: tipo` para asignar los tipos que hayamos definido. Esto es conocido como **type annotation**. Una vez que un identificador es anotado con un tipo, **solo podrá usarse con ese tipo**. Si se usa ese identificador con un tipo diferente, el compilador de TypeScript lanzará un error.
Si no se declara el tipo desde el principio TypeScript lo inferirá automáticamente, aplicando las mismas características que con los tipos explícitos. Si se quiere obviar esta funcionalidad, se puede asignar el tipo **any** a cualquier identificador para usarlo de la forma en la que estamos habituados en JavaScript.

```
let student: string = [1, 2, 4]; // TypeScript error

let helloWorld: any = "Hello world" // Will compile

function talkFunc(phrase: string) {
    console.log(phrase);
}

let phrase = [1, 2, 3];

//talkFunc(phrase); // TypeScript error
talkFunc("Hello world");
```

Categoría de tipos

Tipos primitivos

JavaScript tiene tres tipos primitivos: `string`, `number` y `boolean`. Cada uno de ellos tiene un tipo en TypeScript:

- `string`: Representa las cadenas de JavaScript, con valores como `"Hola mundo"`.
- `number`: Corresponde a los números de JavaScript. Como recordaréis, JavaScript no tiene diferentes valores como int o float, simplemente tenemos `number`.
- `boolean`: Es para los valores booleanos `true` y `false`.

También existen los tipos `String`, `Number` y `Boolean` **con mayúsculas** se refiere a tipos nativos de TypeScript, es legal su uso pero raramente son necesarios.

```
// string
let message: string = "Hello World";
// number
let sum: number = 2;
// optional
let optional: boolean = true;
```

Any

Como hemos dicho arriba, TypeSCript cuenta con un tipo especial llamado `any`, que se usa como comodín para evitar la comprobación de tipos. Básicamente cualquier elemento al que se le asigna `any` pasa a comportarse como un elemento JavaScript, al que se le puede asignar cualquier valor independientemente del tipo y así saltarse la comprobación del compilador de TypeScript.

```
let anyObject: any = { name: "Lucas" };
anyObject.callEmptyMethod();
anyObject.otherAttribute;
anyObject = "String";
```

Tipos por referencia

Además de los tipos primitivos, TypeScript soporta tipos por referencia como `arrys`, `objetos` y el tipado de `funciones`.

- `arrays`: Estructura ordenada de elementos. En el caso de querer tener una colección de números (`[1, 2, 3]`), podemos usar tanto la sintaxis `number[]` como `Array<number>`.
- `functions`: Veremos más adelante el tipado de las funciones, pero por ahora podremos poner un ejemplo de la sintaxis. Una definición de función completa sería `let func: (firstArg: number, secondArg: number) => number = function(firstArg: number, secondArg: number): number {return firstArg + secondArg };`.
- `objects`: Quitando los primitivos, es el otro tipo de tipo más común en TypeScript. Hablaremos en siguientes secciones, pero por ahora podemos adelantar la sintaxis `let newObj: { x: number, y: number} = { x: 10, y: 4 }`.

```
// array
let numbers: number[] = [1, 2, 4];
// function
let func: (firstArg: number, secondArg: number) => number = function (
  firstArg: number,
  secondArg: number
): number {
  return firstArg + secondArg;
};
// object
let newObj: { x: number; y: number } = { x: 10, y: 4 };
```

Union Types

TypeScript permite construir nuevos tipos en base a los ya existentes mediante una serie de operadores. Ahora que sabemos construir tipos, vamos a ver como podemos **combinarlos** para conseguir nuevas funcionalidades.
Un `union type` está formado por dos o más tipos, representando valores que pueden cualquier otro tipo, cada uno de estos tipos son los union's member.

```
function printPhoneNumber(phoneNumber: number | string) {
  console.log("Your phone number is " + phoneNumber);
}

printPhoneNumber(612389238);
printPhoneNumber("637839489");
// printPhoneNumber({ phone: 613892348}); // Error
```

Type Aliases

Ahora que conocemos los union types, podemos usarlos cuando queramos, pero hay **una característica muy útil** de TypeScript para no tener que ir escribiendo siempre las uniones. Suponemos que queremos un argumento que tenga de tipos `number | string | boolean`. Cada vez que queramos comprobar este tipo único, deberíamos escribir esta cadena, pudiendo llevar a errores si olvidamos algún tipo.
Es por ello que podemos definir alias para identificar estos union types con un nombre característico que le queramos dar.

```
type ID = number | string | boolean

let myID: ID = "23789s";
let myOtherId: ID = 213432423;
```

Podemos crear también type aliases compuestos, de diferentes anotaciones de tipos, siendo así que podamos crear definiciones enteras de un tipo.

```
type Mail = {
  header: string;
  body: string;
  timestamp: number;
}

function printMail(mail: Mail) {
  console.log(`${mail.header} - ${mail.timestamp}`);
  console.log(`----------------------------------`);
  console.log(`${mail.body}`);
}
```

Type Assertions

Hay ocasiones en las que tenemos que hacer una conversión de un tipo, bien porque tenía un valor de **any**, bien porque la librería no tiene tipos definidos o que el elemento es genérico, esto por ejemplo en manipulación del DOM es muy común. Es por ello que existen las **type assertions** que permiten hacer un **cast** del tipo de un elemento.

```
const myCanvas = document.getElementById("container") as HTMLDivElement;
```

Interfaces

Vista general de Interfaces

Overview

Las interfaces son un tipo de construcción que permite definir los tipos en Objetos JavaScript. El compilador de TS no convierte la interfaz en código JavaScript, utiliza las interfaces para la comprobación de tipos.
Puede extenderse para tener más flexibilidad y cuenta con atributos opcionales, atributos de solo lectura o funciones.

```
interface Person {
    name: string;
    age: number;
    id?: string;
    talk:(string)=>void;
}

interface Student extends Person {
    college: string;
    bachelor: string;
}

const student: Student = {
    name: "Lucas",
    age: 25,
    college: "ThreePoints",
    bachelor: "Master Full Stack",
    talk: function(phrase: string) {
        console.log(phrase);
    }
}

student.talk("Hello world");
```

Si habéis estado atentos, os daréis cuenta que las `interfaces` son muy parecidas a las `types aliases` que hemos visto en la sección anterior, y es que podemos declarar un typo de objeto anónimo que luego aplicar a cualquier instancia, teniendo atributos, funciones, opcionales... Con esto nos surge la duda de **cuando escoger tipos y cuando interfaces**. Vamos a responderlo.

Diferencias entre Type Aliases e Interfaces

Hace tiempo, las diferencias entre **type aliases** e **interfaces** eran más pronunciadas, pero estos últimos años TypeScript ha ido incorporando funcionalidades en ambas estructuras haciendo que **sea posible elegir libremente cualquiera de las dos**. Casi todas las funcionalidades de `interfaces` se encuentran en `type`, la diferencia más importante es que **los tipos no pueden volver a abrirse para añadir nuevas propiedades**, mientras que las **interfaces son siempre extensibles**.

```
// Extendiendo una Interface

interface Vehicle {
    name: string
}

interface Car extends Vehicle {
    id: number
}

const opel: Car = { name: "Opel", id: 123423434 };
console.log(opel.name);
console.log(opel.id);

// Extendiendo un Type mediante intersecciones

type Computer = {
    name: string
}

type Mac = Computer & {
    model: string
}

const mac: Mac = { name: "MacBook", model: "Pro" };
console.log(mac.name);
console.log(mac.model);


// Añadir nuevos campos a una Interface

interface Phone {
    brand: string
}

interface Phone {
    model: string
}

const iPhone: Phone = { brand: "iPhone", model: "13" };
console.log(iPhone.brand);
console.log(iPhone.model);

// Añadir nuevos campos a una Type (No se puede)

type Tablet = {
    brand: string
}

// Error: Duplicate identifier "Tablet"
// type Tablet = {
//     model: string
// }
```

Narrowing

Interpolación de tipos en TypeScript

Narrowing es la técnica que tiene TypeScript para comprender un bloque de código especial llamado **type guard** que permite interpolar el valor de un tipo que en principio puede ser ambiguo y así poder implementar múltiples lógicas para diferentes tipos. Existen múltiples **type guards** dependiendo del componentes que vayamos a usar, vamos a ver las más importantes.

Narrowing mediante "TypeOf"

Typeof es un operador de TypeScript que permite devolver el tipo de una variable o propiedad. Se utiliza en el contexto de una expresión para tener diferentes lógicas asociadas a un tipo en nuestro código. Las cadenas que `typeof` puede devolver son las siguientes:

- `string`
- `number`
- `bigint`
- `bolean`
- `symbol`
- `undefined`
- `object`
- `function`

```
let newMessage = "Hello World";
console.log(typeof newMessage); // Will return "string"
```

Como habíamos visto en la sección de tipos, un parámetro puede tener múltiples tipos como `number` o `string` gracias a los Union Types o el uso de `any`. Podemos usar el operador `typeof` para hacer distintas operaciones al parámetro de entrada de la función.

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left s
 */
function padLeft(value: string, padding: any) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${typeof padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

Como veis, al hacer uso del condicional junto al `typeof`, TypeScript infiere automáticamente el valor del parámetro `padding`, haciendo que se comporte como un `number` en el primer condicional y como una `string` en el segundo. Podemos todavía afinar un poco más la función sustituyendo el valor `any` por un atributo con Union Types.

```
/**
 * Takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left side.
 * If 'padding' is a number, then that number of spaces is added to the left s
 */
function padLeftUnion(value: string, padding: string|number) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${typeof padding}'.`);
}

console.log(padLeft("Hello world", "        "));
padLeftUnion("Hello world", boolean); // error as it's not an expected type
```

Narrowing mediante equidad

TypeScript también puede usar comparadores de equidad como `===`, `!==`, `==`, `!=` para el narrowing.

```
function example(x: string | number, y: string | boolean) {
  if (x === y) {
    // X e Y serán string
    x.toUpperCase();
    y.toLowerCase();
  } else {
    // X podrá ser string o number
    console.log(x);
    // Y podrá ser string o boolean
    console.log(y);
  }
}

example("string", true);
```

Narrowing mediante "In"

JavaScript tiene el operador `in` para determinar si una propiedad determinada existe en un objeto. Este operador también se puede utilizar como `type guard` para determinar a través de atributos o métodos el tipo de una variable.

```
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(animal: Fish | Bird) {
  if ("swim" in animal) {
    return animal.swim();
  }

  return animal.fly();
}

function getSmallPet(): Fish | Bird {
  return {
    swim: function () {
      console.log("swimming");
    },
  };
}

let pet = getSmallPet();

move(pet);
```

Narrowing usando "Type Predicates"

De momento hemos usado operadores de JavaScript como Type Guards. Pero podemos conseguir lo mismo con una función construida con TypeScript. Simplemente tendremos que definir una función cuyo tipo de retorno es un type predicate.

```
function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}
```

`pet is Fish` es el type predicate en este ejemplo. Un predicado se basa en el esquema `parameterName is Type`, donde `parameterName` debe ser el nombre de un parámetro de la función. Ahora cada vez que se llame esta función, TypeScript aplicará el narrowing a la variable que pasemos como parámetro a esa función.

```
if (isFish(pet)) {
  pet.swim();
} else {
  pet.fly();
}
```

< Back

Funciones

Funciones en TypeScript

Funciones declarativas

Como ya hemos visto en la **sesión 1**, las funciones son el principal medio para pasar datos en JavaScript. Con TypeScript podremos especificar los tipos de los parámetros de entrada y salida de las funciones.

```
// Named function
function sumNums(first: number, second: number): number {
  return first + second;
}

const sum = sumNums(4, 5);

// will fail const fail = sumNums(3, "four");

// Anonymous function
let sumMore = function (first: number, second: number): number {
    return first + second;
}
const sumAnother = sumMore(3, 9);
```

Como podemos ver, podemos asociar tipos tanto a los parámetros de entrada como en el valor de retorno. Si no se indica, **TypeScript inferirá el valor de retorno en base a el cuerpo de la función**, así que no es completamente necesario en todas las ocasiones. En TypeScript nos podemos encontrar los mismos casos que en JavaScript, como pueden ser **funciones sin retorno**, **parámetros opcionales**, **parámetros por defecto**, **parámetros rest**...

```
function printHelloWorld(): void {
    console.log('Hello World');
}
printHelloWorld();

function sumThreeNums(first: number, second: number, third?: number) {
    return first + second + (third || 0);
}

const sumThree = sumThreeNums(4, 5, 10);

const sumOptional = sumThreeNums(4, 5);

function pow(base: number, exponent: number = 10){
    return base ** exponent;
}

const powNum = sumThreeNums(3, 2);

function sumMultipleNums(...rest: number[]) {
  return rest.reduce((p, c) => p + c, 0);
}

const multipleNums = sumMultipleNums(1, 2, 3, 4, 5, 6);
```

Funciones anónimas

Por otro lado, en **funciones anónimas** TypeScript intentará también asignar tipos a los parámetros de entrada dependiendo del contexto. Esto es llamado tipado contextual ya que usa el contexto de ejecución para inferir los tipos.

```
const names = ["Alice", "Bob", "Eve"];

names.forEach((s) => {
  console.log(s.toUpperCase()); // Will trigger -> Property 'toUppercase' does
});
```

< Back

Objetos

Tipar objetos en TypeScript

Como ya vimos en la sección de JavaScript, los objetos son la **unidad fundamental para agrupar y pasar información**. En TypeScript, representamos esta información mediante el tipo objeto.
Este tipo se puede representar de varias maneras, ya sea como objeto anónimo:

```
function greetAnonymous(person: { name: string; age: number }) {
  return "Hello " + person.name;
}
greetAnonymous({name: "Lucas", age: 28});
```

Pueden ser declarados a través de `interfaces`:

```
interface Person {
  name: string;
  age: number;
}

let person: Person = {name: "Lucas", age: 28};

function greetInterface(person: Person) {
  return "Hello " + person.name;
}
greetInterface(person);
```

O como un `type alias`

```
type PersonType = {
  name: string;
  age: number;
};

let personType: PersonType = {name: "Lucas", age: 28};

function greetType(person: PersonType) {
  return "Hello " + person.name;
}

greetType(personType);
```

En estos tres ejemplos tenemos funciones con objetos como parámetros que contienen la propiedad `name`, un string, y `age`, un número.
Es por ello que podemos utilizar cualquiera de estas estructuras para representar objetos en JavaScript, además, como **vimos en la sección de interfaces**, actualmente en TypeScript puede usarse indistintamente `interfaces` y `type alias` en casi cualquier contexto.

Manipulación de Tipos

Conceptos avanzados de tipos

Genéricos

Overview

Uno de los puntos más importantes dentro de la Ingeniería de Software es crear componentes que sean **robustos** y **reusables**. Es por ello que en esta sección nos centraremos en introducir el concepto de `generics`, que permitirá que nuestro código sea más flexible y reusable. El objetivo principal de los **genéricos** es crear un componente que pueda trabajar con una variedad de tipos en contraposición a funcionar exclusivamente con un sólo tipo, como hemos estado viendo hasta ahora.
Vamos a ver el **genérico** más utilizado y simple en JavaScript: El `array`. En TypeScript un array se declara con el tipo `Array`, seguido del tipo que compondrá esta estructura entre los símbolos `< >`. El caso de que sea genérico es que podemos cambiar el tipo entre los símbolos de menor y mayor y la funcionalidad se mantendrá independientemente del tipo.

```
let nums: Array<number> = [1, 3, 4];
```

Vamos a suponer que para una función, vamos a querer devolver el último elemento de un Array:

```
const lastNumber = (arr: Array<number>) => {
  return arr[arr.length - 1];
}

const lNumber = lastNumber([2, 4, 5]);
```

Pero, ¿qué pasa si quisiéramos pasar un array de `strings` como parámetro?. Pues JavaScript va a lanzar un error. Una solución sería usar `union types`, pero tendríamos que contemplar todas las posibilades que pueda abarcar los tipos de array. Es aquí donde entran los **genéricos**.

```
const lastElement = <T>(arr: Array<T>) => {
  return arr[arr.length - 1];
}

const lastElementNumber = lastElement([2, 4, 5]);
const lastElementString = lastElement<string>(['Hello', 'World', '!']); // / T
```

En este ejemplo podemos ver que le pasamos un tipo `T`. Esta `T` puede ser cualquier identificador, pero por convenio se suele utilizar esa letra mayúscula para representar un tipo genérico.
Si prestamos atención al ejemplo, podemos ver que el tipo de retorno es inferido por TypeScript y es que no es necesario declararlo explícitamente para que TypeScript pueda adivinar el tipo que vamos a devolver. Además de esto, al hacer la llamada de la función, TypeScript inferirá el tipo del genérico en base al parámetro de entrada, como se **puede ver en el primer ejemplo**.

Múltiples genéricos

Podemos tener múltiples genéricos como argumentos en una función, en este caso si queremos tener diferentes argumentos con tipos genéricos. El funcionamiento es el mismo, tener múltiples identificadores separados por comas entre los símbolos de mayor y menor `<X, Y, Z>`, siendo que el nombre del identificador por convenio suele ser una letra mayúscula.

```
const makeString = <X, Y>(x: X, y: Y): string => {
    return `${x} ${y}`;
}

const firstString = makeString(5, "hello");

const secondString = makeString<string, number[]>("Array ->", [1, 2, 3]);
```

Como podéis ver, al igual que en el primer ejemplo, podemos inferir los tipos en la llamada de la función al pasar los argumentos, pero si queremos dejarlos explícitamente marcados podemos hacerlo como en la segunda llamada.

Valor por defecto

También podemos tener valores por defecto en los genéricos de manera **similar a valores por defecto en parámetros de una función**. Básicamente tenemos que tipar alguno de los genéricos dentro de su declaración, haciendo así que si no se indica el valor, por defecto tengan que ser del tipo declarado.

```
const makeStringDefault = <X, Y = number>(x: X, y: Y): string => {
    return `${x} ${y}`;
}

const fistStringValid = makeStringDefault(5, "hello");

const secondStringValid = makeStringDefault<number[]>([1, 2, 3], 4);

// Will Fail -> const secondStringValid = makeStringDefault<number[]>([1, 2, 3
```

Como podemos ver, en el primer ejemplo infiere que el segundo parámetro es un `string`, por lo que convierte el genérico `Y` en un `string`. Por otro lado, si decidimos añadir el tipo explícitamente de uno de los genéricos, el otro quedará cono el tipo number por defecto, por lo que si pasamos un argumento que no sea de ese tipo fallará.

Añadir restricciones

Ahora vamos a suponer que tenemos una función que acepta un **objeto genérico**, pero tenemos como condición que ese objeto **tenga una serie de parámetros obligatorios**. Vamos a ver como podríamos solventarlo con genéricos.

```
const makeFullName = <T extends {firstName: string, lastName: string}>( obj: T
    return  {
        ...obj,
        fullName: obj.firstName + " " + obj.lastName
    };
};

const person1 = makeFullName({firstName: "Lucas", lastName: "Fernandez", age:
const person2 = makeFullName({firstName: "Pedro", lastName: "Ramirez", profess
// Will Fail const personFail = makeFullName({otherName: "Lucas", lastName: "F
```

Con esto ya vemos un caso de uso bastante potente, intentar replicar este mismo comportamiento con JavaScript requeriría comprobaciones en tiempo real en el cuerpo de la función y excepciones o algún mecanismo similar cuando no se cumplen estas restricciones.

Interfaces

Podremos usar también usar genéricos en interfaces cuando queremos múltiples tipos con variaciones de atributos de una forma sencilla.

```
interface Message<T> {
    id: string;
    timestamp: number;
    data: T;
}

type MessageNumber = Message<number>;
let messageNumber: MessageNumber = {id: "as8df98asdf", timestamp: 23429342340,
console.log(messageNumber);

type MessageString = Message<string>;
let messageString: MessageString = {id: "oiausdf989as", timestamp: 30495838989
console.log(messageString);
```

Keyof

El operador `keyof` coge como parámetro un tipo objeto y produce una cadena con la unión de sus tipos, por ejemplo:

```
type Point = { x: number; y: number };
type P = keyof Point; // "x" | "y"
let point: P = "x";
```

Si quisiéramos poner otro valor que no fuese `x` o `y`. TypeScript se quejaría. En principio no parece muy útil, pero este operador combinado con genéricos puede ayudar en ciertos casos de uso:

```
type Staff = {
  name: string;
  salary: number;
};

function getProperty<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}

const developer: Staff = {
  name: "Tobias",
  salary: 100,
};

const nameType = getProperty(developer, "name");
// Compiler error -> const salaryType getProperty(developer, 'pay'); //Cannot
```

Typeof

Ahora vamos a ver como usar `typeof` de forma más avanzada. Si os acordáis, este operador devolvía un string con el tipo de una variable o propiedad.

```
let s = "hello";
let n: typeof s = "world";
console.log(n);
```

Pero si combinamos esta funcionalidad con estructuras más avanzadas como ReturnType, que acepta como parámetro una función y devuelve el tipo de retorno, podemos conseguir cosas como esta:

```
function f() {
    return { x: 10, y: 3 };
}
type PointPredicate = ReturnType<typeof f>;

const pointPredicate: PointPredicate = {x: 10, y: 4};
console.log(pointPredicate);
```

Básicamente podemos conseguir definir tipos con el parámetro de retorno de una función de forma muy sencilla.

Conditionals

Los condicionales son un paso más para permitir la modificación de nuestro código en base a ciertos parámetros de entrada. Vamos a ver un ejemplo.

```
interface Animal {
  live(): void;
}
interface Dog extends Animal {
  woof(): void;
}

type Example1 = Dog extends Animal ? number : string; // En este primer caso s

type Example2 = RegExp extends Animal ? number : string; // Como RegExp no ext
```

Como podemos ver se usa la estructura del operador ternario para decidir si una evaluación se cumple. En el primer caso como el tipo `Dog` extiende del tipo `Animal`, asignaremos el tipo de la izquierda. En el segundo caso al ser la condición falsa se asignará el tipo de la derecha.
Ahora vamos a ver un ejemplo más complejo, supongamos que queremos que una función devuelva un objeto diferente dependiendo del tipo de parámetro de entrada. Con `conditionals` es relativamente sencillo implementar la lógica.

```
interface IdLabel {
    id: number;
    message: string;
    timestamp: number;
}
interface NameLabel {
    name: string;
    message: string;
    timestamp: number;
}

type NameOrId<T extends number | string> = T extends number ? IdLabel : NameLab

function createLabel<T extends number | string>(idOrName: T): NameOrId<T> {
  if (typeof idOrName === "number") {
    return { id: 1, message: "Hello world", timestamp: 234234234 } as NameOrId
  } else {
    return { name: "foo", message: "Hello world", timestamp: 234234234 } as Na
  }
}

let firstLabel = createLabel("typescript");

let secondLabel = createLabel(23);
```

Clases

Definir clases en TypeScript

Overview

TypeScript utiliza la sintaxis moderna de ES6 para soportar **clases**. Tiene una aproximación más parecida a una orientación de objetos real que JavaScript. Además, como habíamos mencionado antes, una de las ventajas de TypeScript es que podemos usar esta característica en cualquier navegador ya que podrá compilarse a versiones anteriores. En esta sección vamos a ver algunas de las propiedades de las clases en TypeScript.

```
class Person {
  name: string;
  constructor(name: string) {
    this.name = name;
  }
  speak(phrase: string = "Hello World") {
    console.log(talk);
  }
}

let person: Person = new Person("Pablo");
```

Palabras reservadas

readonly

Podemos utilizar la palabra reservada `readonly` para indicar que un atributo es sólo de lectura.

```
class Greeter {
  readonly name: string = "world";

  constructor(otherName?: string) {
    if (otherName !== undefined) {
      this.name = otherName;
    }
  }
}

const greeter = new Greeter();

console.log(greeter.name);
// This will fail -> g.name = "also not ok";
```

Si intentamos acceder al valor `name` TypeScript lanzará un error de acceso.

implements

La palabra reservada `implments` permite comprobar si una clase satisface la estructura de una `interface` en concreto.

```
interface Pingable {
  ping(): void;
}

class Sonar implements Pingable {
  ping() {
    console.log("ping!");
  }
}

// class Ball implements Pingable { //Class 'Ball' incorrectly implements inte
//   pong() {
//     console.log("pong!");
//   }
// }
```

extends

La palabra reservada `extends` sirve para implementar herencia en TypeScript. Las clases derivadas heredan todas las propiedades y métodos de la clase base, y como no se pueden definir miembros adicionales. O hacer override de métodos con `super`.

```
class StudentExtend extends Person {
    constructor(name: string) {
        super(name);
    }
    speak(phrase = "And I want to learn") {
        console.log("I'm a student...");
        super.speak(phrase);
    }
}

class Teacher extends Person {
    constructor(name: string) {
        super(name);
    }
    speak(phrase = "And I want to teach") {
        console.log("I'm a teacher...");
        super.speak(phrase);
    }
}

let pepe = new StudentExtend("I'm learning TypeScript");
let juan: Person = new Teacher("I'm teaching new feature");

pepe.speak();
juan.speak("And i love it");
```