El concepto de TypeScript union Type es un concepto curioso sobre todo cuando se mira en un primer momento . ¿Qué es un union type en TypeScript? . Se trata de un tipo que unifica varios tipos de golpe. Es decir nosotros en TypeScript podemos tener el siguiente código de TypeScript:

```
let texto:string="hola";
console.log(texto);
let numero:number=5;
console.log(numero);
```

Se trata de un código muy sencillo de entender simplemente son dos tipos de datos y se imprime la información por la consola:



TypeScript Union Type

Podemos construir un código similar usando un unión type. En esta caso el tipo de nuestra variable puede ser number o string veámoslo.

```
</pre>

let tipounido:string| number

tipounido="hola";
console.log(tipounido);
tipounido=5;
```

```
console.log(tipounido);
```

El resultado será idéntico :

```
hola
5
```

¿Esto sirve realmente para algo?. Es una buena pregunta muchas veces la gente nos dirá que es que aporta flexibilidad.Sin embargo esto no queda demasiado claro ya que normalmente necesitamos un numero o una cadena no ambas cosas. Ahora bien lo que esta claro es que si este tipo existe a nivel core del lenguaje tendrá su utilidad. Veamos un ejemplo más practico supongamos que tenemos la clase Factura y la clase Cliente en TypeScript



Veamos su código:

```
class Factura {

  numero:number;
  importe:number;

  pagar() {
    console.log("factura pagada");
  }
```

```
}


class Cliente {

  dni:number;
  nombre:string;

  email() {
    console.log("enviando un correo a "+ this.nombre);
  }


}
```

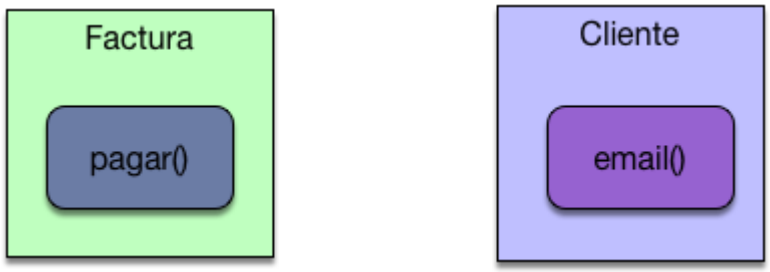
Son dos clases normales y corrientes que podemos usar :

```
let f= new Factura();
f.importe=100;
f.numero=1;
let c= new Cliente();
c.nombre="pedro";
f.pagar();
c.email();
```

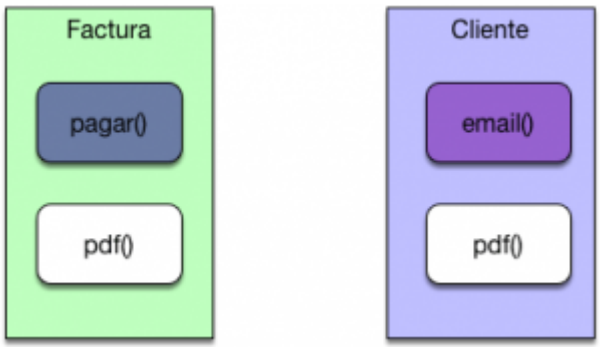
El resultado en la consola:

```
factura pagada
enviando un correo a pedro
```

Hemos invocado un método de cada clase .



Supongamos ahora que queremos generar un pdf por cada uno de los conceptos . No tendríamos problema porque valdría con modificar las clases y añadir un método que genere el pdf algo como esto:




```
class Factura {

  numero:number;
  importe:number;
```

```
  pagar() {
    console.log("factura pagada");
  }

  pdf() {
    console.log("factura"+this.numero+" "+ this.importe);
  }

}


class Cliente {

  dni:number;
  nombre:string;

  email() {
    console.log("enviando un correo a "+ this.nombre);
  }

  pdf() {
    console.log("cliente "+this.dni+" "+ this.nombre);
  }
}
```

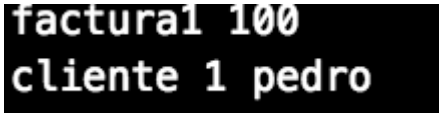
Invocamos estos métodos:

```
let f= new Factura();
f.importe=100;
```

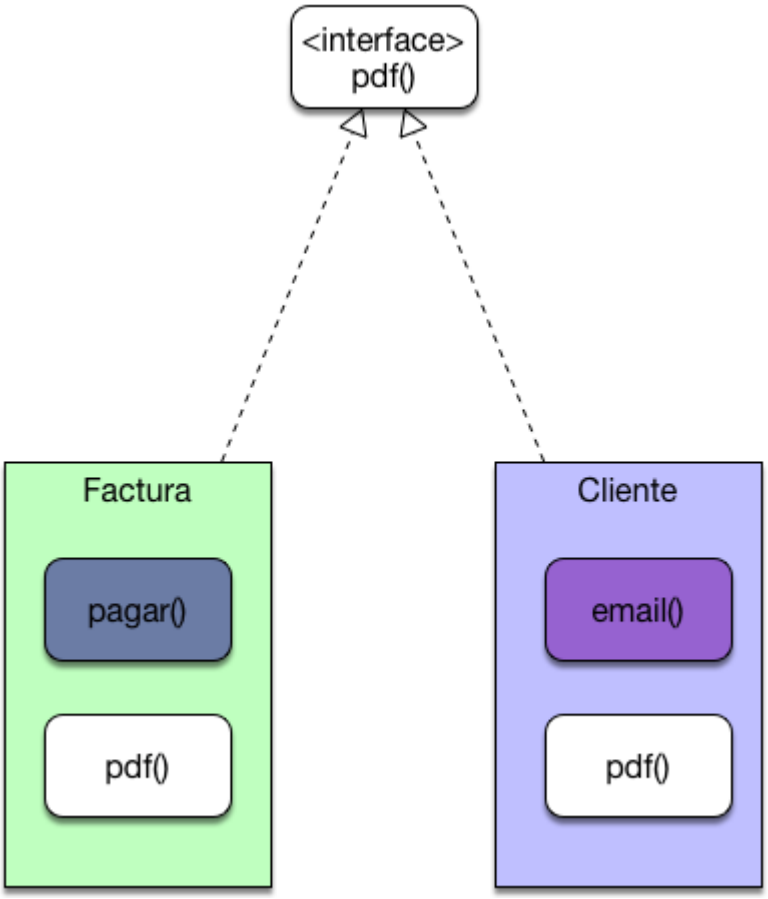
```
f.numero=1;
let c= new Cliente();
c.nombre="pedro";
c.dni=1;
f.pagar();
c.email();

f.pdf();
c.pdf();
```

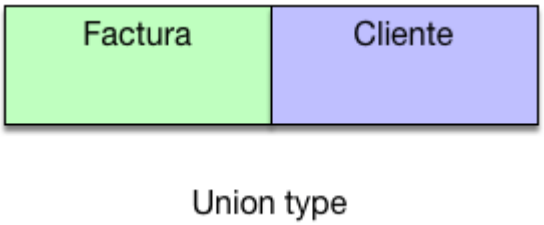
Nos imprimira la info de cada objeto en la consola:



Todo funciona muy bien . El problema viene cuando queremos tener un método que imprima la info de cualquier objeto en consola que disponga del metodo pdf. Uhmmm…. eso ya es más complejo deberiamos definir un interface algo “pdfeable” o “pdf” si nos apoyamos en los conceptos clásicos de Java .



JavaScript y por consonancia TypeScript es mucho más ductil . Nos podemos definir una función que reciba como parámetro un TypeScript Union Type.



Union type

De esta forma nos ahorraremos la declaración del interface:

```
function imprimirObjeto(o : Cliente|Factura):void {

o.pdf();
}

imprimirObjeto(f);
imprimirObjeto(c);
```

El resultado será identico y nos habremos ahorrado definir el interface:

```
factura1 100
cliente 1 pedro
```

TypeScript Type Alias

No solo podemos abordarlo así sino que ademas nos podríamos apoyar en el concepto de Type Alias de TypeScript para simplificarlo aun más y convertir nuestro union type en un alias.

```
type TipoPDF=Cliente| Factura;
function imprimirObjeto(o : TipoPDF):void {

o.pdf();
}

imprimirObjeto(f);
imprimirObjeto(c);
```

Todo más elegante ,aprendamos a usar los TypeScript Type Alias en nuestro código y las cosas nos irán mejor.

Otros artículos relacionados:

1. TypeScript CallBack ,funciones y sintaxis
2. TypeScript interface utilizando Angular DTOs
3. TypeScript Generics y su funcionamiento

Otros artículos externos:

1. Tipos avanzados de TypeScript