UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

Tipos funcionales y Subtipos

Tipos Funcionales

Ya conocemos la **POO**, conocemos los **tipos básicos** y los **nuevos tipos** que podemos crear a partir de ellos.

Ahora nos adentraremos en una nueva característica de los sistemas tipados, la habilitada de **"tipar" funciones.**

Si podemos definir tipos de funciones y usar funciones en los mismos lugares que usamos valores de otros tipos —como variables, argumentos y valores de retorno— podemos simplificar la implementación de varias construcciones comunes y abstraer algoritmos comunes a funciones.

La base

Funciones como ciudadanos de primera clase

TypeScript trata a las funciones como **"ciudadanos de primera clase"**, esto significa que podemos manipular las funciones tal cual como lo hacemos con otros tipos de datos como **string**, **number** y **boolean**

Podemos empezar conociendo las diferencias entre **referenciar** y **ejecutar una función.**

Cuando usamos los paréntesis al llamar a una función estamos ejecutandola.

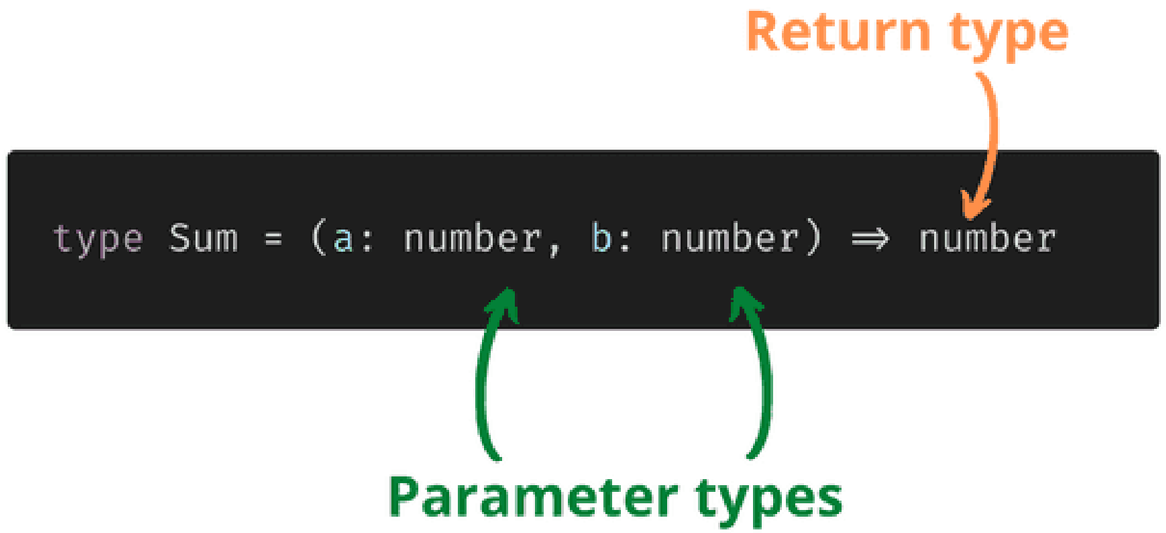
`saludar()`

Cuando **no** usamos los paréntesis al llamar a una función estamos referenciandola.

`saludar`

```
// Lo mas basico
function saludar() {
    console.log("¡Hola mundo!");
}
console.log(saludar); // Muestra la función como objeto -> [Function: saludar]
saludar(); // Ejecuta la función -> ¡Hola mundo!

// Asignando referencias vs resultados
function sumar(a: number, b: number): number {
    return a + b;
}
const referencia = sumar; // Asigna la funcion a una variable
const resultado = sunar(1, 1); // Asigna el resultado (2) a una variable

console.log(referencia); // Muestra la función como objeto -> [Function: suma]
console.log(resultado); // Muestra el resultado -> 2
console.log(referencia(1, 1)) // Ejecuta la función a través de la referencia y
muestra el resultado -> 2
```

¿Como se ve un tipo funcional?



El tipo de una función viene dado por el **tipo de sus argumentos** y su **tipo de retorno**.
Si dos funciones toman los mismos argumentos y retornan el mismo tipo, tienen el mismo tipo.

Al conjunto de argumentos más el tipo de retorno también se le conoce como la firma (signature) de una función.

¿Qué podemos hacer siendo ciudadanos de primera clase?

1. Asignar funciones a variables

```typescript
// Creamos una funcion normal
function saludar(nombre: string): string {
  return `Hola, ${nombre}!`;
}

// El tipo 'saludoFunction' describe cómo debe ser la función.
type saludoFunction = (nombre: string) => string;

// asignamos la funcion a una variable.
const miSaludo: saludoFunction = saludar;

// Ahora podemos usar la variable como si fuera la función original.
console.log(miSaludo("Mundo")); // Salida: Hola, Mundo!

// Incluso podriamos asignarla a una nueva variable
const miSaludoCopia: saludoFunction = saludar;
console.log(miSaludoCopia("Mundo desde Copia")); // Salida: Hola, Mundo desde Copia!
```

miSaludo no contiene el resultado de saludar, sino que **miSaludo** es la función saludar. Esto nos permite, por ejemplo, cambiar dinámicamente qué función se ejecuta.

Ejercicio: Haremos una calculadora.

Tendremos una única variable llamada **operacionActual**. Dependiendo de la función que le asignemos a esta variable, realizará una suma, una resta o una multiplicación.

```
type OperacionMatematica = (a: number, b: number) => number;

// Funciones que podemos usar
function sumar(a: number, b: number): number {
  return a + b;
}
function restar(a: number, b: number): number {
  return a - b;
}
// TAREA: Crea función 'multiplicar'.

// TAREA: Usa solo esta variable para realizar las operaciones.
let operacionActual: OperacionMatematica;

console.log("---Calculadora Dinamica---");
// TAREA: Realiza una suma
// TAREA: Realiza una resta
// TAREA: Realiza una multiplicacion
```

Antes de seguir al punto número 2...

Arrow Functions (=>)

Podemos definir una función flecha de diversas maneras:

```ts
const saludo = (quien: string): string => {
    return `Hola, ${quien}!`;
};

saludo('Estefany'); // => 'Hola, Estefany!'
```

Se parece mucho a la tipificación de una función, no se confundan.

```ts
// Arrow sin bloque y tipo de retorno inferido
const saludo2 = (quien: string) => `Hola, ${quien}!`;

// Arrow sin bloque, tipo de retorno inferido y parámetro sin tipo (any)
const saludo3 = quien => `Hola, ${quien}!`;
```

Ya podemos seguir

2. Pasar Funciones como Argumentos (Callbacks)

Permite que una función tome otra función como "instrucción" sobre qué hacer (se pasa como parámetro).

Supongamos que queremos una función que procese una lista de nombres y los muestre. Podríamos querer mostrarlos en mayúsculas, minúsculas o de alguna otra forma.

En lugar de crear una función para cada caso, creamos una función genérica que acepta otra función como argumento para realizar la transformación.

¿Tienen otra idea de función especifica que podamos crear?

```typescript
type Transformacion = (texto: string) => string;

// Procesa los nombres. Acepta un array y una función de transformación.
function procesarNombres(nombres: string[], transformar: Transformacion): void {
  for (const nombre of nombres) {
    console.log(transformar(nombre));
  }
}

// Ahora definimos las funciones específicas que queremos usar.
// Usa la sintaxis de función anónima.
const aMayusculas: Transformacion = function(texto) {
    return texto.toUpperCase();
}
// Usa la sintaxis de función flecha.
const aMinusculas: Transformacion = (texto) => texto.toLowerCase();

const listaDeNombres = ["Ana", "Juan", "Pedro"];
procesarNombres(listaDeNombres, aMayusculas); // Salida: ANA JUAN PEDRO
procesarNombres(listaDeNombres, aMinusculas); // Salida: ana juan pedro
```

Ejercicio: Retrocedamos un poco

¿Cómo haríamos esto si las funciones no fueran tratadas como ciudadanos de primera clase?

Ejercicio: Retrocedamos un poco

¿Cómo haríamos esto si las funciones no fueran tratadas como ciudadanos de primera clase?

```
function procesarNombresAMayusculas(nombres: string[]): void {
  for (const nombre of nombres) {
    console.log(nombre.toUpperCase());
  }
}
function procesarNombresAMinusculas(nombres: string[]): void {
  for (const nombre of nombres) {
    console.log(nombre.toLowerCase());
  }
}

procesarNombresAMayusculas(listaDeNombres); // Salida: ANA JUAN PEDRO
procesarNombresAMinusculas(listaDeNombres); // Salida: ana juan pedro
```

3. Retornar Funciones desde otras Funciones (Closures)

Una función también puede "fabricar" y devolver otra función. Podemos crear funciones preconfiguradas.

Quieres crear funciones que multipliquen por un número específico, podríamos tener duplicar, triplicar, etc.

```typescript
// Esta función no devuelve un número, devuelve otra funcion!
function crearMultiplicador(factor: number): (numero: number) => number {
  // La función que devolvemos "recuerda" el valor de 'factor'.
  // Esto se conoce como un 'closure'.
  return function(numero: number): number {
    return numero * factor;
  };
}

// Creamos funciones específicas usando nuestra "fábrica" de funciones.
const duplicar = crearMultiplicador(2);
const triplicar = crearMultiplicador(3);

// Ahora usamos las nuevas funciones que hemos creado.
console.log(duplicar(5));   // Imprime: 10
console.log(triplicar(5));  // Imprime: 15
console.log(duplicar(10));  // Imprime: 20
```

Internamente, si usamos esta "fábrica" de la siguiente forma:

const duplicar = crearMultiplicador(2)

es como si estuviéramos devolviendo:

```
function(numero: number): number {
    return numero * 2
}
```

```
// Esta función no devuelve un número, devuelve otra funcion!
function crearMultiplicador(factor: number): (numero: number) => number {
  // La función que devolvemos "recuerda" el valor de 'factor'.
  // Esto se conoce como un 'closure'.
  return function(numero: number): number {
    return numero * factor;
  };
}

// Creamos funciones específicas usando nuestra "fábrica" de funciones.
const duplicar = crearMultiplicador(2);
const triplicar = crearMultiplicador(3);

// Ahora usamos las nuevas funciones que hemos creado.
console.log(duplicar(5));   // Imprime: 10
console.log(triplicar(5));  // Imprime: 15
console.log(duplicar(10));  // Imprime: 20
```

No podemos olvidar que estamos definiendo un tipo funcional como retorno de la función ***crearMultiplicador***

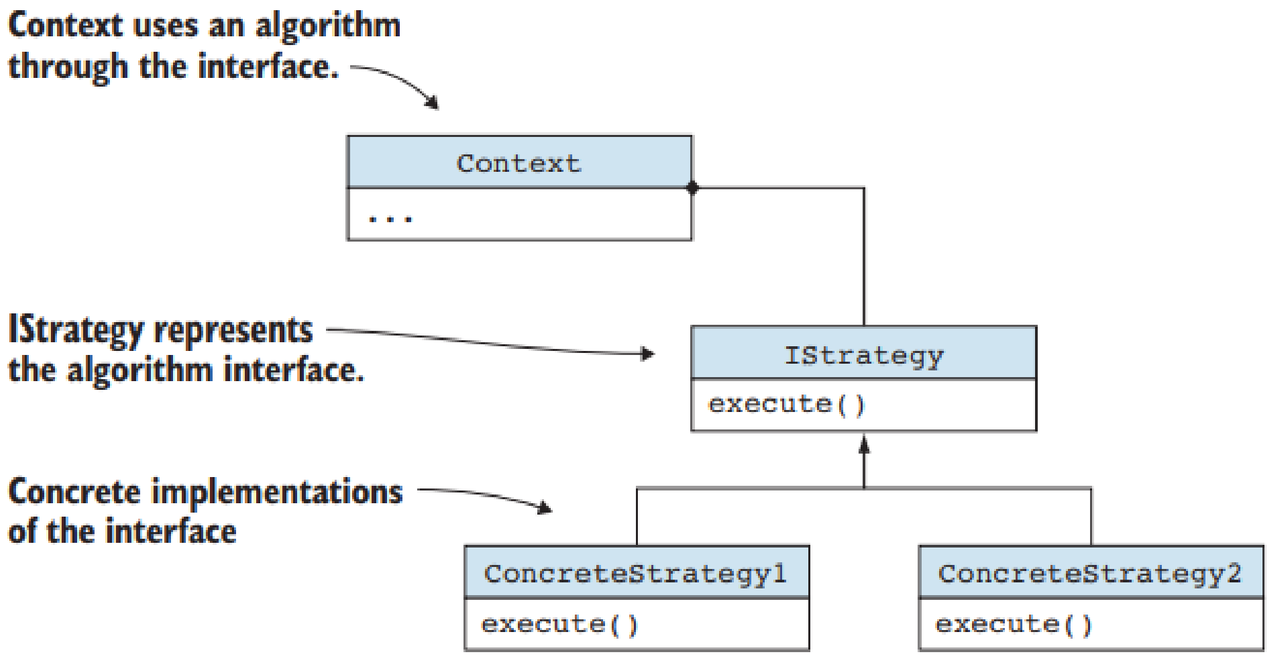
```
function crearMultiplicador(factor: number): (numero: number) => number {
  return function(numero: number): number {
    return numero * factor;
  };
}
```

Tipo de Retorno

Patron Strategy ¿Otra vez?

Supongamos que tenemos un **auto-lavado** con dos tipos de servicios, **lavado standard** y **lavado premium** (cuesta mas).

Les suena a **Strategy** ¿verdad?



Este código funciona, pero es **innecesariamente verboso**. Hemos introducido una interfaz y dos tipos que la implementan, cada uno proporcionando un único método **wash()**.

Estos tipos no son realmente importantes; la **parte valiosa** de nuestro código es la **lógica de lavado**.

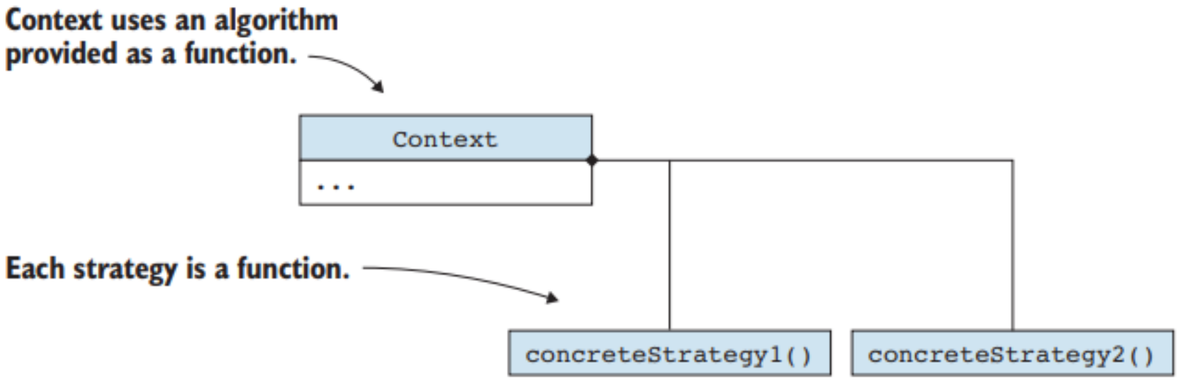
Este código es solo una función, por lo que podemos simplificarlo mucho si pasamos de interfaces y clases a un tipo funcional y dos implementaciones concretas.

```
class Car {
    /* ... */
}

interface IWashingStrategy {
    wash(car: Car): void;
}

class StandardWash implements IWashingStrategy {
    public wash(car: Car): void {
        /* Perform standard wash */
    }
}

class PremiumWash implements IWashingStrategy {
    public wash(car: Car): void {
        /* Perform premium wash */
    }
}

class CarWash {
    public service(car: Car, premium: boolean): void {
        let washingStrategy: IWashingStrategy;

        if (premium) {
        washingStrategy = new PremiumWash();
        } else {
        washingStrategy = new StandardWash();
        }
        washingStrategy.wash(car);
    }
}
```

Podemos definir **WashingStrategy** como un tipo que representa una función que recibe un **Car** como argumento y devuelve **void** (no devuelve nada).

Luego podemos implementar los dos tipos de lavados como dos funciones **standardWash()** y **premiumWash()**, ambas tomando un **Car** y devolviendo **void**.

El **CarWash** puede seleccionar una de ellas para aplicarla a un auto determinado.



En esta implementación tenemos menos partes.
Pero las dos logran el mismo objetivo.

```typescript
class Car {
    /* Represents a car */
}

type WashingStrategy = (car: Car) => void;

function standardWash(car: Car): void {
    /* Perform standard wash */
}

function premiumWash(car: Car): void {
    /* Perform premium wash */
}

class CarWash {
    public service(car: Car, premium: boolean): void {
        let washingStrategy: WashingStrategy;

        if (premium) {
            washingStrategy = premiumWash;
        } else {
            washingStrategy = standardWash;
        }
        washingStrategy(car);
    }
}
```

¿Entienden por que esto funciona?

Es importante tener en cuenta que el patrón es el mismo, todavía estamos encapsulando una familia de algoritmos y seleccionando en tiempo de ejecución cuál usar. La diferencia está en la implementación, que las capacidades modernas nos permiten expresar más fácilmente.

Estamos reemplazando una interfaz y dos clases concretas con una declaración de tipo y dos funciones.

En la mayoría de los casos, la implementación más sucinta es suficiente. Podríamos necesitar reconsiderar la implementación con interfaz y clases cuando los algoritmos no son representables como funciones simples. A veces, necesitamos múltiples funciones o necesitamos rastrear algún estado, en cuyo caso la primera implementación sería más adecuada, ya que agrupa las piezas relacionadas de una estrategia bajo un tipo común.

Preguntas

1. ¿Cuál es el tipo de una función isEven() que toma un número como argumento y devuelve true si el número es par y false en caso contrario?

a) [number, boolean]

b) (x: number) => boolean

c) (x: number, isEven: boolean)

d) {x: number, isEven: boolean}

2. ¿Cuál es el tipo de una función check() que toma un número y una función del mismo tipo que isEven() como argumentos, y devuelve el resultado de aplicar la función dada al valor dado?

a) (x: number, func: number) => boolean

b) (x: number) => (x: number) => boolean

c) (x: number, func: (x: number) => boolean) => boolean

d) (x: number, func: (x: number) => boolean) => void

Preguntas

1. ¿Cuál es el tipo de una función isEven() que toma un número como argumento y devuelve true si el número es par y false en caso contrario?

a) [number, boolean]

b) (x: number) => boolean

c) (x: number, isEven: boolean)

d) {x: number, isEven: boolean}

2. ¿Cuál es el tipo de una función check() que toma un número y una función del mismo tipo que isEven() como argumentos, y devuelve el resultado de aplicar la función dada al valor dado?

a) (x: number, func: number) => boolean

b) (x: number) => (x: number) => boolean

c) (x: number, func: (x: number) => boolean) => boolean

d) (x: number, func: (x: number) => boolean) => void

Preguntas

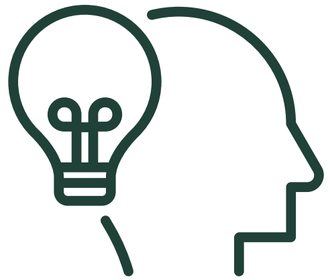
1. ¿Cuál es el tipo de una función isEven() que toma un número como argumento y devuelve true si el número es par y false en caso contrario?

b) (x: number) => boolean

2. ¿Cuál es el tipo de una función check() que toma un número y una función del mismo tipo que isEven() como argumentos, y devuelve el resultado de aplicar la función dada al valor dado?

c) (x: number, func: (x: number) => boolean) => boolean

3. Implementa la función check() , la función isEven() y una función adicional con la misma firma que isEven()



1. ¿Cuál es el tipo de una función isEven() que toma un número como argumento y devuelve true si el número es par y false en caso contrario?

b) (x: number) => boolean

2. ¿Cuál es el tipo de una función check() que toma un número y una función del mismo tipo que isEven() como argumentos, y devuelve el resultado de aplicar la función dada al valor dado?

c) (x: number, func: (x: number) => boolean) => boolean

3. Implementa la función check() , la función isEven() y una función adicional con la misma firma que isEven()

```
function check(x: number, func: (x: number) => boolean): boolean {
    return func(x);
}

// Función isEven del ejemplo
function isEven(x: number): boolean {
    return x % 2 === 0;
}

// Función adicional, vemos que es flexibilidad
function isPositive(x: number): boolean {
    return x > 0;
}
```

```
// Usando la función check
console.log(check(4, isEven));      // true
console.log(check(5, isEven));      // false
console.log(check(10, isPositive)); // true
console.log(check(-3, isPositive)); // false
```

Ejercicio

Convierte el ejercicio de **Duck en un Strategy Funcional**.

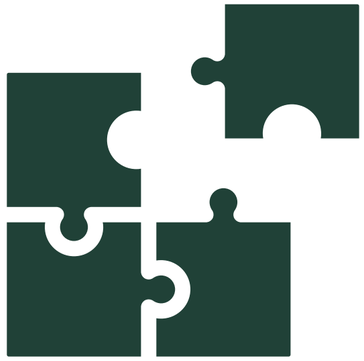
Agrega un nuevo comportamiento "**comer**" el cual debe recibir **siempre** alguna comida (solo existen 3 tipos de comida: **grano, pan, semilla**).

Implementa 3 algoritmos de comer, sabiendo que:
- Algunos patos pueden comer todo tipo de comida.
- El pato mallard es alérgico, por lo tanto solo puede comer granos.
- Existe patos de goma que no pueden comer.

El comportamiento comer también debe recibir una "cantidad", esta debe ser **opcional**.

Planteamiento inicial en M7

Ejercicio



Resolución en M7

Acá existe un detalle, y es que como TypeScript usa un **sistema de tipos estructural**, podríamos hacer algo como:

```
mallardDuck.setFlyBehavior(muteQuack)
```

Es decir, podemos usar un comportamiento de quack en donde se espera un comportamiento de vuelo, esto es porque tienen la misma estructura/firma.

Lazy Values

La **evaluación perezosa** o **evaluación tardía** es una técnica de optimización que consiste en **retrasar un cálculo costoso** hasta el momento exacto en que se necesita su resultado, porque no siempre será necesario realizar este cálculo.

En lugar de calcular un valor inmediatamente (lo que se conoce como evaluación "ansiosa" o "eager"), envolvemos ese cálculo en una función. Luego, pasamos esa función en lugar del valor. La función solo se ejecutará si el resultado es **realmente necesario**.

Esto evita desperdiciar recursos en operaciones pesadas que quizás nunca se lleguen a utilizar.

Ejemplo

```
class Bike {
    // Logica de Bicicleta
}
class Car {
    // Logica de Carro
}

function isItRaining(): boolean {
    return Math.random() < 0.5; // Simula lluvia aleatoria
}

function chooseMyRide(bike: Bike, car: Car): Bike | Car {
    if (isItRaining()) {
        return car;
    } else {
        return bike;
    }
}

chooseMyRide(new Bike(), new Car());
```

Esto se conoce como **eager evaluation** o **evaluación ansiosa**.

Para llamar a **chooseMyRide()**, necesitamos suministrar un objeto **Car**, por lo que ya estamos pagando el costo de construir un **Car**. Si el clima es bueno y decido usar mi bicicleta, la instancia del **Car** fue creada para nada.

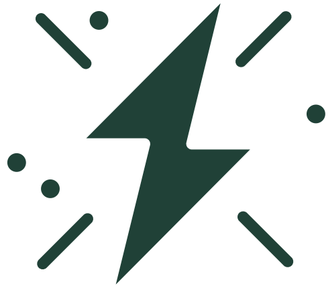
Hagamos esto de una forma Lazy.

Ejemplo

```
class Bike { /* Logica de Bicicleta */ }
class Car { /* Logica de Carro */ }

function isItRaining(): boolean {
    return Math.random() < 0.5; // Simula lluvia aleatoria
}

function chooseMyRide(bike: Bike, car: () => Car): Bike | Car {
    if (isItRaining()) {
        return car();
    } else {
        return bike;
    }
}

chooseMyRide(new Bike(), () => new Car());
```

El carro se crea única y exclusivamente si se va a usar.

Por lo tanto no desperdiciamos recursos ni tiempo.



Funciones de orden superior

Funciones que aceptan o retornan otras funciones.

A una función "normal", es decir, una que solo trabaja con argumentos y valores de retorno que no son funciones (como números o strings), se le conoce como **función de primer orden**.

Cuando una función **toma una función de primer orden** como **argumento** o la devuelve como **resultado**, la llamamos **función de segundo orden**. Podríamos continuar esta lógica para definir funciones de "tercer orden" y así sucesivamente, pero en la práctica, agrupamos a todas las funciones que operan con otras funciones bajo un solo término: **funciones de orden superior** (higher-order functions).

Acabamos de hacer esto en el último ejemplo:

```
function chooseMyRide(bike: Bike, car: () => Car): Bike | Car {
```

Varios algoritmos útiles pueden ser implementados como funciones de orden superior, siendo los más fundamentales **map()**, **filter()** y **reduce()**.

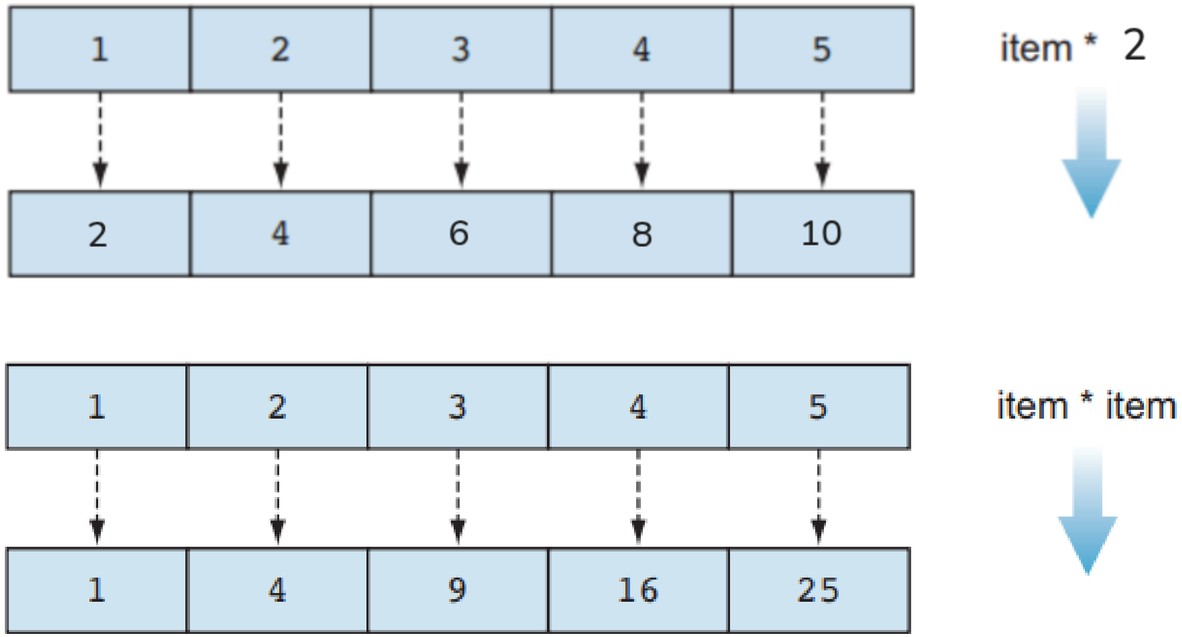
Función map()

La idea detrás de la función **map()** es que a partir de una colección/arreglo de valores, se aplica una función a cada uno de ellos y se devuelve una nueva colección con los resultados.
Este es un tipo de operación que surge constantemente en la práctica, por lo que tiene mucho sentido crear una abstracción para reducir la duplicación de código.

Veamos dos escenarios como ejemplo. En el primero, tenemos un arreglo de números y queremos multiplicar por dos cada uno. En el segundo, queremos el cuadrado de cada uno.

Podríamos resolver ambos casos usando un bucle for, pero nos daremos cuenta de algo...

```ts
let numbers: number[] = [1, 2, 3, 4, 5];

let doubled: number[] = [];
// Multiplicar por 2 cada numero
for (const n of numbers) {
    doubled.push(n * 2);
}

let squared: number[] = [];
// Cuadrado de cada numero
for (const n of numbers) {
    squared.push(n * n);
}
```



Aunque multiplicar por dos y sacar el cuadrado son operaciones distintas, la **estructura subyacente del proceso es idéntica**:

- Se toma un arreglo de entrada.
- se aplica una función a cada uno de sus elementos.
- Se genera un nuevo arreglo con los resultados.

Ejercicio

Pueden crear una función map() que pueda encapsular este comportamiento? (solo recibe números y retorna números)

Ejercicio

Pueden crear una función map() que pueda encapsular este comportamiento? (solo recibe números y retorna números).

```
function mapNumbers(items: number[], func: (n: number) => number): number[] {
    const result: number[] = [];
    for (const item of items) {
        result.push(func(item));
    }
    return result;
}

const numbers: number[] = [1, 2, 3, 4, 5];
const doubled = mapNumbers(numbers, n => n * 2);
const squared = mapNumbers(numbers, n => n * n);

console.log('numbers:', numbers); // [1, 2, 3, 4, 5]
console.log('doubled:', doubled); // [2, 4, 6, 8, 10]
console.log('squared:', squared); // [1, 4, 9, 16, 25]
```

Ejercicio

Incluso podemos usar **tipos genéricos**.

```
function map<T, U>(items: T[], func: (item: T) => U): U[] {
    let result: U[] = [];
    for (const item of items) {
        result.push(func(item));
    }
    return result;
}

let numbers: number[] = [1, 2, 3, 4, 5];
let squares: number[] = map(numbers, (item) => item * item);
let strings: string[] = ["apple", "orange", "peach"];
let lengths: number[] = map(strings, (item) => item.length);
```

La función **map()** encapsula el proceso de aplicar una función (que le pasamos como argumento) a todos los elementos de un arreglo (que también le pasamos como argumento).

Nosotros solo tenemos que proporcionarle un arreglo de elementos y una función, y **map()** se encarga de devolvernos el nuevo arreglo con los resultados de dicha aplicación.

Más adelante, veremos cómo podemos llevar esta idea un paso más allá para que funcione con cualquier tipo de estructura de datos, no únicamente con arreglos. Pero incluso con la implementación actual, ya tenemos una excelente abstracción para aplicar funciones a colecciones de elementos, lo que nos permite reutilizarla en una gran variedad de situaciones.

*Algo similar pasa con **filter()** y **reduce()***

Fin