

# Curso CFP

## CFP

# Programador full-stack

*Manejo de Errores*  
*Repaso General de POO*

# Agenda

- Un sistema en funcionamiento
  - Expectativa vs. Realidad
  - Necesidad de *estandarizar*
- Manejo de Errores
- Errores en TypeScript
- Ejemplos de Errores predefinidos
- Posibilidad de definir errores
- Repaso de la Semana
  - Interfaces
  - Clase Abstracta
  - Patrones de Diseño
  - Errores
- Recomendaciones Generales
- Ejercicios

# Expectativa vs. Realidad

- Normalmente cuando estamos programando, pensamos en el funcionamiento del sistema según la funcionalidad que debería proveer
- Sin embargo, también es muy común no prestar el tiempo necesario a contemplar todos los casos posibles en nuestras clases
- La realidad es que el código es utilizado más veces de forma incorrecta que las que uno se imagina
- Por lo tanto es muy importante tener en cuenta posibles casos donde usen nuestras clases de una manera diferente a la originalmente pensada

# Necesidad de *estandarizar*


- Estableciendo una forma común de resolver los problemas, es más fácil que el grupo e incluso uno mismo entienda la forma en que resuelve las cosas → y por ende lo resuelva más rápido
- Hasta el momento, para validar parámetros, cada uno lo hacía de la manera en que se le ocurría
  - Retornar valores inválidos, imprimir mensajes, etc.
- Existen formas específicas de hacer el manejo de errores, provistas por los lenguajes

# Manejo de Errores

- En TypeScript se pueden lanzar errores cuando se detecta que algún valor no cumple con alguna condición determinada
- Junto con cada error se puede incluir un mensaje de error que nosotros querremos
- De esta manera, cada vez que por ejemplo queramos hacer *validación de parámetros*, en caso de presentarse valores inválidos, generamos un *error*

# Errores en TypeScript (1)


```
function sumarValores(valores: number[]): number {  
  if (valores.length == 0)  
    throw new Error('No se pueden sumar los valores de un arreglo vacío')  
  
  let sumaTotal = 0;  
  
  for (let i = 0; i < valores.length; i++)  
    sumaTotal += valores[i];  
  
  return sumaTotal;  
}
```



**Cuando se detecta algo que consideramos  
inválido → lanzamos el error**

```
let valores = [];  
let resultado: number;
```

```
try {  
  resultado = sumarValores(valores);  
} catch(error) {  
  console.log('Ocurrió un error esperado, seteando resultado en 0');  
  resultado = 0;  
}
```



**Con los bloques  
try/catch capturamos  
un error**

```
console.log(resultado);
```

# Errores en TypeScript (2)

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores> tsc auto.ts; node auto.js
C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores\auto.js:3
    throw new Error('No se pueden sumar los valores de un arreglo vacío');
    ^
```

```
Error: No se pueden sumar los valores de un arreglo vacío
    at sumarValores (C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores\auto.js:3:15)
    at Object.<anonymous> (C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores\auto.js:11:13)
    at Module._compile (internal/modules/cjs/loader.js:799:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:810:10)
    at Module.load (internal/modules/cjs/loader.js:666:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:606:12)
    at Function.Module._load (internal/modules/cjs/loader.js:598:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:862:12)
    at internal/main/run_main_module.js:21:11
```



**Cuando ejecutamos código sin envolverlo en un try/catch**

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores> tsc auto.ts; node auto.js
Ocurrió un error esperado, seteando resultado en 0
0
```



**Cuando lo ejecutamos envolviéndolo, es decir contemplando el caso en que se de el error**

# Posibilidad de Definir Errores

```
class CustomError extends Error {  
    public constructor(message?: string) {  
        super(message);  
  
        this.name = CustomError.name;  
    }  
}
```

**Definimos nuestros propios errores  
como si fuesen una clase más**

```
PS C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores> tsc auto.ts; node auto.js  
C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores\auto.js:25  
    throw new CustomError('No se pueden sumar los valores de un arreglo vacío');  
    ^
```

```
CustomError: No se pueden sumar los valores de un arreglo vacío  
    at new CustomError (C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores\auto.js:17:28)  
    at sumarValores (C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores\auto.js:25:15)  
    at Object.<anonymous> (C:\Users\Francisco\Documents\CFP\3. POO\Ejercicios\poo\errores\auto.js:33:13)  
    at Module._compile (internal/modules/cjs/loader.js:799:30)  
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:810:10)  
    at Module.load (internal/modules/cjs/loader.js:666:32)  
    at tryModuleLoad (internal/modules/cjs/loader.js:606:12)  
    at Function.Module._load (internal/modules/cjs/loader.js:598:3)  
    at Function.Module.runMain (internal/modules/cjs/loader.js:862:12)  
    at internal/main/run_main_module.js:21:11
```

**El uso de errores propios nos puede  
ayudar a ver por donde viene un error**



# Demostración en Clase

- Captura de errores → try/catch
- Definición de Errores
- Forzar errores

# Curso CFP

## CFP Programador full-stack

*Repaso de la Semana*

# Repaso - Interfaces

- Consiste en un planteo de lo que una clase debería hacer
- Los métodos se especifican pero no se implementan
- No se ponen modificadores de acceso
  - Todo es *public*
- Recomendación → si bien se puede, no poner variables internas en las interfaces
  - Porque se pondría todo como público
  - Las clases que implementen la interfaz expondrían al exterior su estado → se rompe el encapsulamiento

# Repaso - Clase Abstracta

- Existe un “punto intermedio” entre las clases y las interfaces
  - Ni todo definido (interfaces), ni todo implementado (clases)
- Las *clases abstractas* son clases normales, pero que permiten tener métodos *sin implementar*
- La contra que tienen, es que no se pueden instanciar → o sea no se pueden usar
  - Tiene que haber otra clase que las extienda e implemente el método que les falta implementar
- Una clase no se puede usar si no tiene *todos* sus métodos implementados

# Repaso - Patrones de Diseño

- Los problemas conocidos en programación tienen soluciones conocidas
- Las soluciones conocidas se llaman *patrones de diseño*
- Se tratan de formas de programar que facilitan la resolución de un problema
- Existen muchos patrones de diseño
  - Vamos a ver los patrones que usan como solución, los mecanismos de programación orientada a objetos

# Repaso - Errores

- En TypeScript se pueden lanzar errores cuando se detecta que algún valor no cumple con alguna condición determinada
- Junto con cada error se puede incluir un mensaje de error que nosotros querremos
- De esta manera, cada vez que por ejemplo queramos hacer *validación de parámetros*, en caso de presentarse valores inválidos, generamos un *error*

# Recomendaciones Generales

- Recordar hacer un planteo de la solución antes de pasar a la implementación
  - El uso de interfaces es muy recomendable para pensar de forma más pulida la funcionalidad que debería tener una clase
- Una clase tiene que tener una sola responsabilidad
- Los patrones de diseño si bien son soluciones conocidas a problemas conocidos, no son una *navaja suiza*
  - No forzarlos, saber justificar el por qué de su uso
- Usar manejo de errores para contemplar casos de datos inválidos

**Curso CFP**

**CFP**  
**Programador**  
**full-stack**

***Ejercicios***



# Ejercicios - En Clase

Tomar tres ejercicios a elección que hayan hecho antes, y modificar la validación de los datos para hacerla empleando errores

# Ejercicios - Fuera de Clase

## Ejercicio 1

Agregar soporte de errores a la base de datos básica de la clase 1

## Ejercicio 2

A partir del soporte de errores de la base de datos, agregar a dicho soporte manejo de *errores propios*