UCAB UNIVERSIDAD CATÓLICA ANDRÉS BELLO

Tópicos Especiales de Programación

Sistemas de Tipos

El idioma de la computadora es demasiado simple

Cuando escribimos algo como edad = 25
la computadora lo traduce a `00011001`

¿Es el número 25? ¿Es el caracter salto de línea (LF) en la tabla ASCII? ¿Es una instrucción para el procesador?

Es imposible saberlo, por lo tanto tenemos un programa inestable.

Para evitar este caos, los lenguajes de programación nos obligan a ponerle *etiquetas* a nuestros datos.
A estas *etiquetas* las llamamos **tipos**.

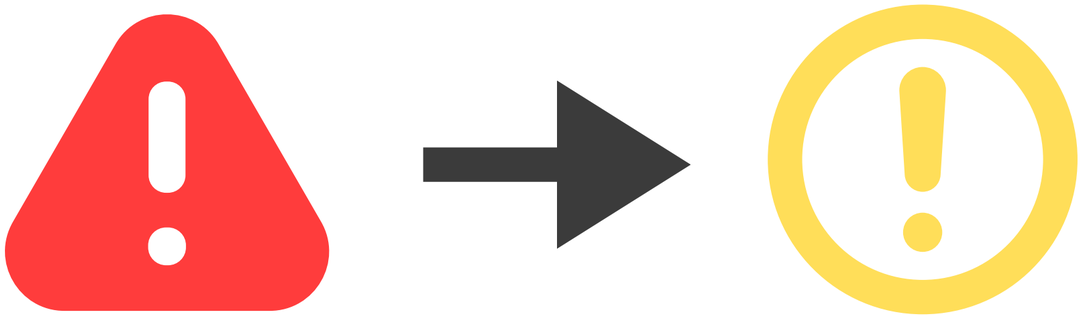
`let edad: number = 25;` dice, trata a esta caja de memoria que llamo **edad** como un **numero**

El tipo le dice a la computadora
¿Qué puedo guardar aquí? ¿Qué puedo hacer con esto? ¿Cómo lo entiendo?

Si la comprobación de tipos falla acabamos con un fallo de compilación (*type checker* en lenguajes interpretados) o con un error en tiempo de ejecución.

Y muchas veces el sistema de tipos es para **proteger a la computadora de nosotros**.

Y por eso ofrece mecanismos para **transformar errores de ejecución en errores de compilación**. Esto podría sonar raro porque convertimos un **error** en otro **error**, pero todo radica en cuál es el tipo de error más peligroso, y este es sin duda un **error de ejecución**.



Correctitud

Podemos tener problemas con el Contrato Implícito

Y lo solucionamos con el Contrato Explícito

```typescript
// ! Error de tipo en tiempo de ejecución
function scriptAt(s: any): number {
  return s.indexOf("Script");
}
console.log(scriptAt("TypeScript")); // Imprime '4'
console.log(scriptAt(42)); // Lanza un error en tiempo de ejecución


// ? Solución: Añadir anotaciones de tipo
function scriptAt(s: string): number {
  return s.indexOf("Script");
}
console.log(scriptAt("TypeScript")); // Imprime '4'
console.log(scriptAt(42)); // Lanza un error en tiempo de compilación
```

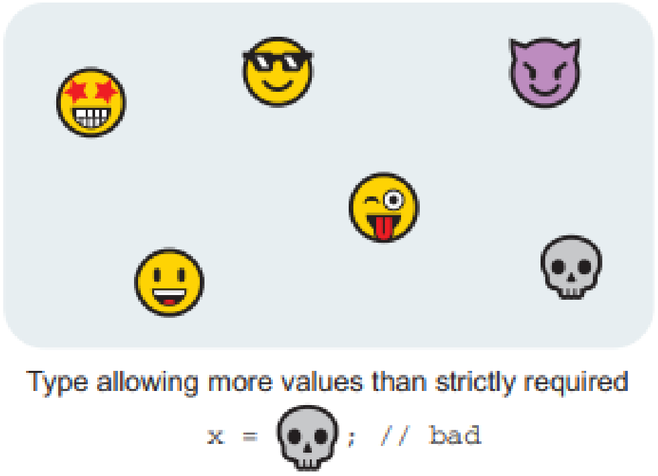
Correctitud

Reducimos el Espacio de Estados

Es el producto cartesiano de los dominios de todas sus variables activas

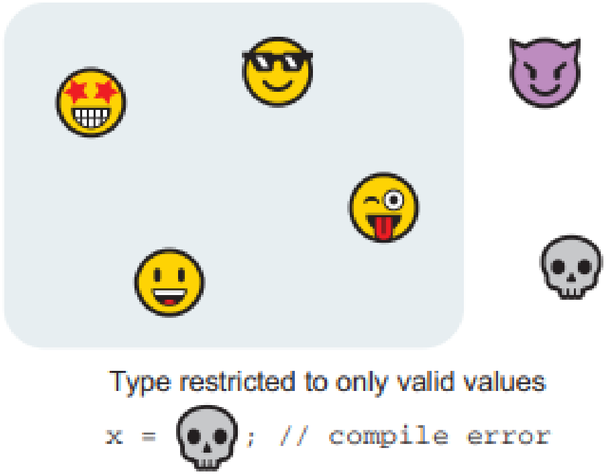
Un tipo como **any** tiene un **dominio casi infinito**, permitiendo un espacio de estados enorme y, por tanto, un gran número de posibles estados inválidos.

Al aplicar un tipo estricto como string, se **restringe drásticamente el dominio** de la variable.



Type allowing more values than strictly required

```
x = 💀; // bad
```



Type restricted to only valid values

```
x = 💀; // compile error
```

Inmutabilidad

Reducimos el espacio de estados de un programa

Simplificamos la concurrencia, evitando que múltiples proceso o hilos puedan modificar un dato.

Aumentamos la predictibilidad, porque las funciones que reciben datos inmutables, solo pueden producir nuevos datos, y no se modifican los originales.

```
1  // ! Problema: Uso de `let` permite reasignación accidental
2  function safeDivide(): number {
3      let divisor = 42;
4
5      // Revisamos que no sea 0... todo bien.
6      if (divisor == 0) throw new Error("¡Peligro!");
7      // Cambio accidental en otra parte del código.
8      divisor = 0;
9      return 100 / divisor; // Error en tiempo de ejecución
10 }
11
12 // ? Solución: Usar `const` para evitar reasignaciones
13 function safeDivide(): number {
14     const divisor = 42;
15
16     // Revisamos que no sea 0... todo bien.
17     if (divisor == 0) throw new Error("¡Peligro!");
18     // Cambio accidental en otra parte del código.
19     divisor = 0; // Error en tiempo de compilación
20     return 100 / divisor;
21 }
```

Encapsulación

Protegemos el estado interno de los objetos.

Se controla **dónde** pueden ocurrir mutaciones necesarias y **bajo qué reglas**.

Un **buen ejemplo** es una **máquina expendedora**

```
1  class SafeDivisor {
2      // ! Problema: `divisor`  puede ser modificado
3      divisor: number = 1;
4      // ? Solución: Hacer `divisor` privado
5      private divisor: number = 1;
6
7      setDivisor(value: number) {
8          if (value == 0) throw new Error("Value should not be 0");
9          this.divisor = value;
10     }
11     divide(x: number): number {
12         return x / this.divisor;
13     }
14 }
15
16 function exploit(): number {
17     let sd = new SafeDivisor();
18     sd.divisor = 0;
19     return sd.divide(42);
20 }
21
```

Componibilidad

Aislamos el Qué del Cómo

El truco es darse cuenta que la condición

```
n < 0 o s.length === 1
```

Es un "trozo de lógica" que podemos pasar como si fuera un dato más.
En JS/TS, las funciones son **ciudadanos de primera clase**, así que podemos hacer exactamente eso.

```typescript
// Función 1: Busca el primer número negativo
function findFirstNegativeNumber(numbers: number[]): number | undefined {
  for (const n of numbers) {
    if (n < 0) { // Condición específica
      return n;
    }
  }
  console.error("No se encontró un número negativo.");
}

// Función 2: Busca el primer string de un solo carácter
function findFirstOneCharacterString(strings: string[]): string | undefined {
  for (const s of strings) {
    if (s.length === 1) { // Condición específica
      return s;
    }
  }
  console.error("No se encontró un string de un carácter.");
}
```

Componibilidad

Aislamos el Algoritmo.

Parametrizamos el Comportamiento.

Facilitamos la Reutilización y Composición.

```
// Una función que sabe CÓMO recorrer, pero no QUÉ buscar.
// Recibe la condición como un parámetro.
function findFirstNumber(
    numbers: number[],
    condicion: (n: number) => boolean // <--
): number | undefined {

    for (const n of numbers) {
        // Ejecuta la condición que nos pasaron
        if (condicion(n)) {
            return n;
        }
    }
    console.error("No se encontró un número que cumpla la condición.");
}

const numeros = [1, 5, -10, 50, -3];

// Queremos el primer negativo. La condición es `n < 0`.
const primerNegativo = findFirstNumber(numeros, (n) => n < 0);
// Queremos el primer número mayor que 40. La condición es `n > 40`.
const mayorQue40 = findFirstNumber(numeros, (n) => n > 40);
// Queremos el primer número par. La condición es `n % 2 === 0`.
const primerPar = findFirstNumber(numeros, (n) => n % 2 === 0);
```

Legibilidad

El código se lee mucho más de lo que se escribe.

Los tipos son la forma más eficaz de hacer que el código se autodocumente, porque a diferencia de un comentario, el compilador los verifica.

Un tipo bien definido es un comentario que el compilador te obliga a mantener actualizado

```
1  // ! Problema: Legibilidad casi nula
2  function add(a: any, b: any) {
3      return a + b;
4  }
5
6  add(2, 3);  // 5
7  add("2", 3);  // "23" (concatenación)
8  add({}, []);  // "[object Object]"
9
10 // ? Solución: Añadir anotaciones de tipo
11 function add(a: number, b: number): number {
12     return a + b;
13 }
14
15 add(2, 3);
```

Sistemas de tipos nominales

- Se basan en el **nombre/declaración explícita** del tipo.
- Dos tipos son compatibles **solo si tienen el mismo nombre** o están explícitamente relacionados (herencia, interfaces implementadas).
- Lenguajes: Java, C#, C++, Rust.

Desventajas:

- Demasiada rigidez.
- Necesitas adaptadores para conectar tipos idénticos
- Más código repetitivo (*boilerplate*).
- Dificulta la reutilización de código

```
// (Java)
class Persona { String nombre; int edad; }
class Usuario { String nombre; int edad; }
// No son compatibles aunque tengan la misma estructura
```

Sistemas de tipos estructurales

- Se basan en la **forma/estructura del tipo**.
- Dos tipos son compatibles si tienen la misma estructura, **independientemente de su nombre**.
- Lenguajes: TypeScript, Go.

Desventajas:

- Duck typing accidental **si camina como pato y habla como pato, es un pato**.
- Errores menos descriptivos.
- Puede crear confusión cuando tipos diferentes tienen la misma estructura por casualidad.
- Menos control explícito sobre las relaciones entre tipos.

```
// (TypeScript)
type Persona = { nombre: string; edad: number }
type Usuario = { nombre: string; edad: number }
// Son compatibles porque tienen la misma estructura
```

Duck Typing

```ts
// Dos tipos con la misma forma pero distinto significado
type Pixel = { x: number; y: number };
type Tile = { x: number; y: number };

function drawAt(px: Pixel) {
    //...
    console.log(`Dibujando en ${px.x}px, ${px.y}px`);
}

const playerInTiles: Tile = { x: 10, y: 5 };

// Compila sin errores (mismo shape), pero es incorrecto en tiempo de ejecución:
// estamos pasando tiles donde se esperaban píxeles.
drawAt(playerInTiles);
```

Por úlltimo, podemos decir que:

Nominal: "¿Tienes el certificado correcto?"
Estructural: "¿Sabes hacer el trabajo?"

Fin