TypeScript

Equipo de trabajo

Evian Concepción Peña
alu0101395548@ull.edu.es

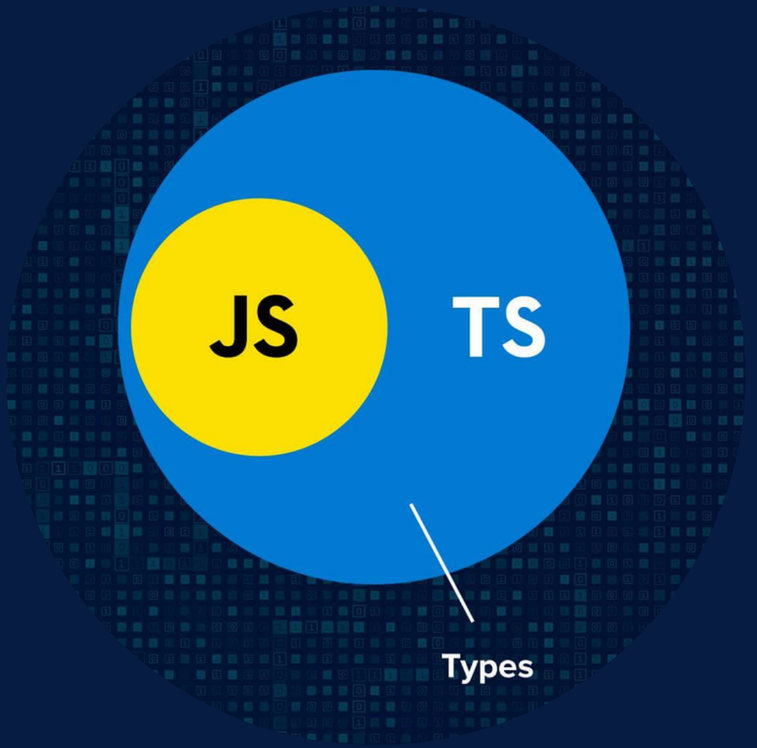
Saúl Sosa Díaz
alu0101404141@ull.edu.es

Index

1 - What is TypeScript?
2 - Basic installation and configuration.
3 - Basic concepts.
4 - Advanced concepts.
5 - Bibliografy.

What is TypeScript?

- Is a superset of JavaScript
- Adds static typing
- Implements objects and classes
- Extends JavaScript syntax
- Is translated to JavaScript
- ".ts" / ".tsx" extension type
- Multiparadigm

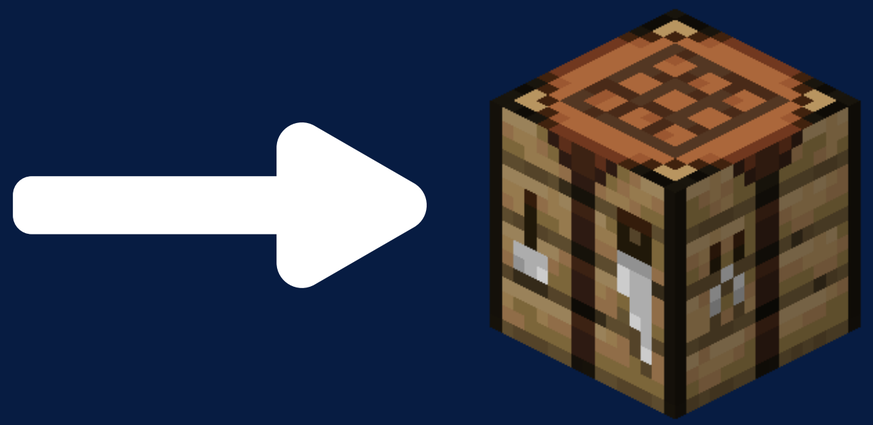


How TypeScript works?

How TypeScript works?

Source code
TS

Compiler



Source code JS

Installation

1- Installing Node.js and npm

Node version must be greater than **14.17**

2- Install a TypeScript compiler

Compiler

First choice

Use TypeScript to generate a JavaScript source file that is then interpreted with Node.js

Installation

```
npm install -g typescript
```

```
tsc -v
```

Compiler

Second choice

Use ts-node

- It compiles and runs with a single command.

Installation

```
npm init -y # Iniciar un nuevo proyecto de Node.js
npm install -g -ts-node
```

```
ts-node -v
```

Start guide

1. Create Node.js project

```
npm init -y
```

```
usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAI$ npm init -y
Wrote to /home/usuario/Presentation_PAI/package.json:

{
  "name": "presentation_pai",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

2. Create TypeScript compilation configuration

```
tsc --init
```

```
usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAI$ tsc --init

Created a new tsconfig.json with:

  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true


You can learn more at https://aka.ms/tsconfig _
```

TSconfig.json

- Project Configuration
- Code rules
- Improve Code Quality
- Customization

```
{
  "compilerOptions": {
    "target": "es2016",
    "strict": true,
    "noImplicitAny": true,
    "outDir": "./dist",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "skipLibCheck": true
  }
}
```

Hello world

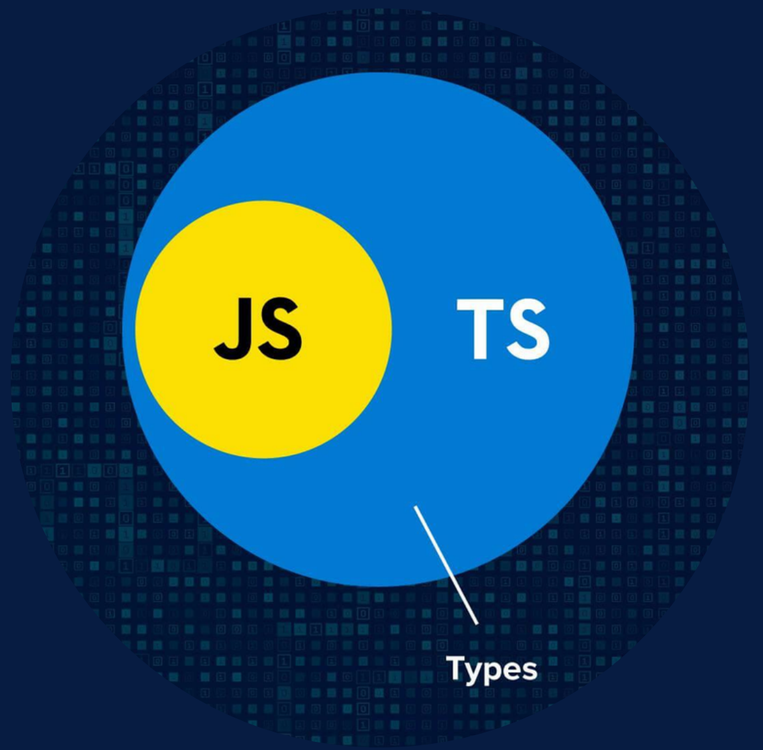
```
1  function helloWorld(): void {
2    console.log("Hello world!");
3  }
4
5  helloWorld();
```

```
usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAI/src/primerosPasos$ tsc ./hello-world.ts && node ./hello-world.js
Hello world!
```

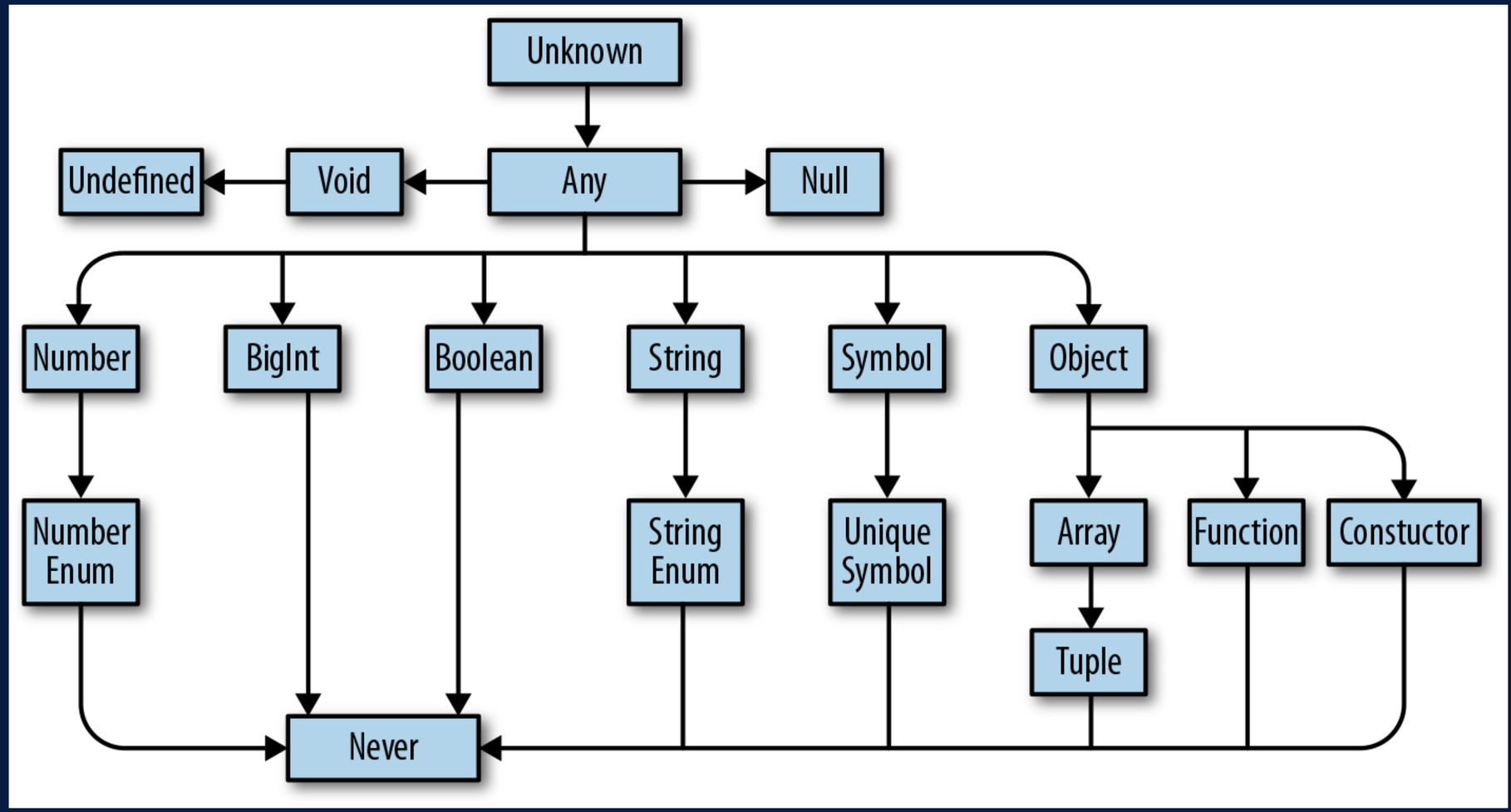
```
usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAI/src/primerosPasos$ ts-node ./hello-world.ts
Hello world!
```

Basic Concepts

- Data types
- Variable declaration
- User inputs
- Conditionals
- Loops
- Operators
- Fuctions
- Export and import
- Read/Write from a file
- More reserved words



Data types


Unknown
Undefined
Void
Any
Null
Number
BigInt
Boolean
String
Symbol
Object
Number Enum
String Enum
Unique Symbol
Array
Function
Constuctor
Tuple
Never

Variable declaration

Global variable

- Not recommended, actually forbidden for us

```
var randomVariable: number = 10;
```

Constants

- Use if no modification needed

```
const randomVariable: number = 10;
```

Scope variable

- Recommended

```
let randomVariable: number = 10;
let randomVariable2: number | string = 10;
```

Variable declaration

Arrays

- [] type recommended / Array<type> not recommended

```
let randomVariable: number[] = [1, 2, 3];
```

Strings

- Preferable (‘ ’) over (“ “)

```
let randomVariable: string = 'You passed CYA';
```

Create objects

- always use new, don´t worry, TS uses garbage recollection

```
let randomVariable: Array<number> = new Array<number>(1, 2, 3);
```

User inputs

Terminal

- user input starts at argv[2]

⚠️ npm i --save-dev @types/node

Execution time

- we need to load the readline module

```ts
const entry: string[] = process.argv;
console.log(entry);
// entry[0] path to ts-node
// entry[1] path to the file
// entry[2] begining of arguments

// load de required module
const readLine = require('readline');
const readUserInput = readLine.createInterface({
  input: process.stdin,
  output: process.stdout
});

readUserInput.question('Write somenting: ', (entry: string) => {
  console.log(`Ha ingresado: ${entry}`);
  // Puedes guardar la entrada en una variable o
  // en algún lugar según tus necesidades.
  readUserInput.close();
});
```

Conditionals

switch

- Efficient and easy to read
- Remember to set a break per case
- It´s a good practice to put a default case

```
let randomVariable = 40;
const randomVariable2: number = 50;
switch (randomVariable) {
    case 10:
        // CODE
        break;
    case 20:
        // CODE
        break;
    case randomVariable2:
        // CODE
        break;
    default:
        // GOOD PRACTICE
        break;
}
```

Conditionals

if

- if it is too simple you don´t need braces

else

- try to avoid, you might use ternary operator

else if

- if you got many of them probably you need a switch

```
let randomVariable = 40;
const randomVariable2: number = 50;

if (randomVariable > randomVariable2) {

} else if (randomVariable < randomVariable2) {

} else {

}
```

Loops

Classic for

- Customizable

for of

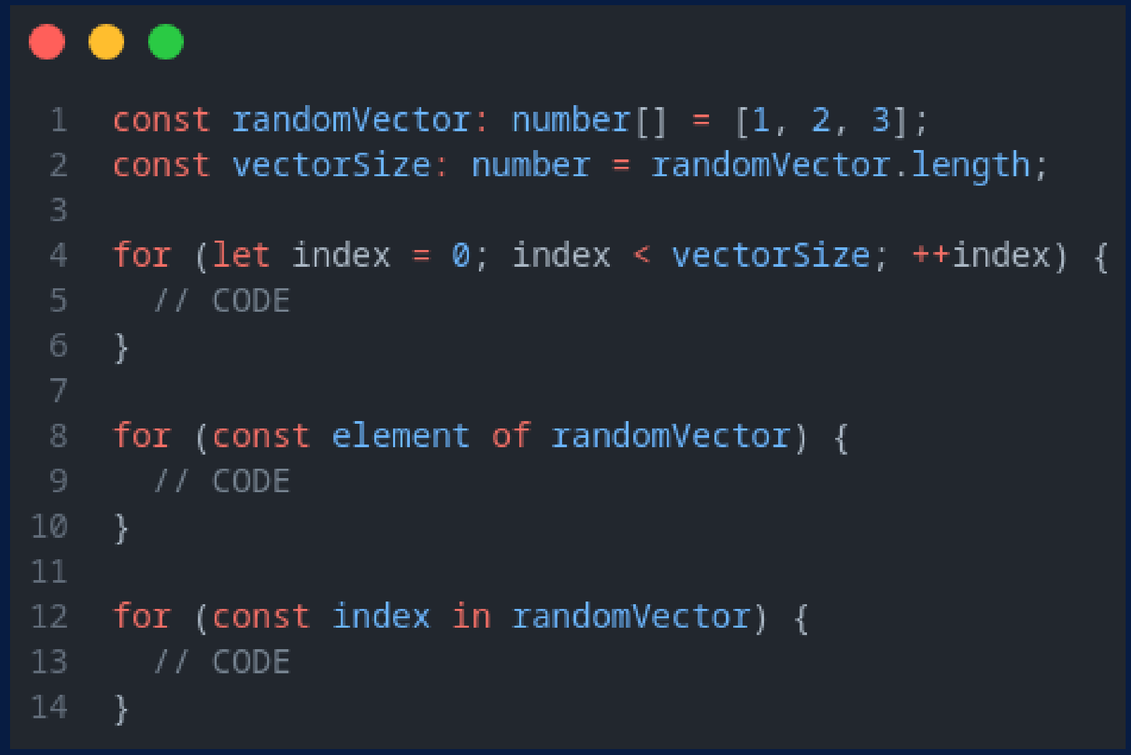
- Safe iteration

for in

- Use for index

for each

- Based in functional programming



```ts
const randomVector: number[] = [1, 2, 3];
const vectorSize: number = randomVector.length;

for (let index = 0; index < vectorSize; ++index) {
  // CODE
}

for (const element of randomVector) {
  // CODE
}

for (const index in randomVector) {
  // CODE
}
```

Loops

while

- Be careful with infinite loops

do while

- executed at least once

continue

- skips to the next iteration

break

- finish the loop

```
let condition: boolean = false;
let condition2: boolean = false;
while (condition) {
  if (condition2) {
    continue;
  }
}

do {
  if (condition2) {
    break;
  }
} while (condition);
```

Functions

Basic

- good practice: remember to put the type it will return

Arrow

- use of “->”
- can be stored in variables
- if not needed preferible use basic functions, more redeable

```
1  function randomFunction(randomParameter: number): number {
2    return randomParameter * randomParameter;
3  }
4
5  randomFunction(10);
6
7  let randomFunction2 = (value1: number, value2: number): number => {
8    return value1 + value2;
9  }
10
11 randomFunction2(10, 10);
```

Export and Import

export

- doesn´t export the whole file

```
export class dummy {
  constructor() {}
  public hello(): void { console.log("Hello"); }
}

export function hello(): void { console.log("Hello"); }
```

import

- if only one thing needed avoid using “*”

```
// import { dummy } from "./a";
// import { hello } from "./a";
import * as testing from './exporting';
import { dummy, hello } from './exporting';

let temp: dummy = new dummy();
temp.hello();
hello();
testing.hello();
```

circular dependency

- if you reached to this error, consider redesigning your code

Read/Write from a file

Read

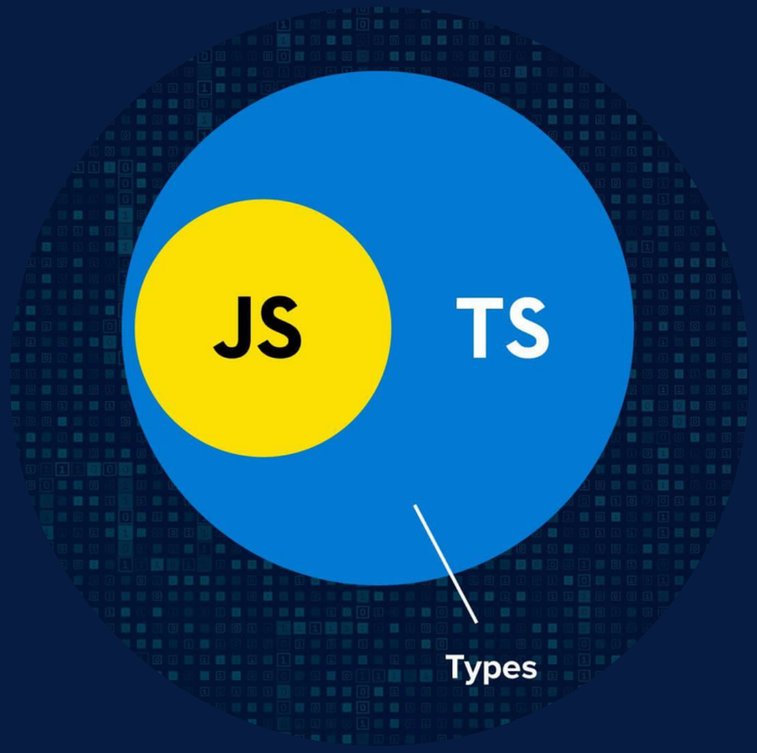
```ts
import * as fs from 'fs';
import * as readline from 'readline';

const filePath: string = 'data.txt';

// read the whole file
fs.readFile(filePath, 'utf8', (error, data) => {
  if (error) {
    console.error('Error reading file: ${err}');
  } else {
    console.log('\nFile content:\n', data);
  }
});

// read line by line
const readLine = readline.createInterface({
  input: fs.createReadStream(filePath),
  crlfDelay: Infinity,
});

readLine.on('line', (line) => {
  console.log('Line:', line);
});

readLine.on('close', () => {
  console.log('File reading complete.');
})
```

Write

```ts
import * as fs from 'fs';

const filePath: string = 'data.txt';

const dataToWrite: string = 'NEW LINE';

fs.writeFile(filePath, dataToWrite, { flag : 'a' }, (error) => {
  if (error) {
    console.error('Error reading file: ${err}');
  } else {
    console.log('Write operation complete.');
  }
});
```

Advanced concepts

- Classes
- Inheritance
- User Defined Types
- Interfaces
- Generic Types
- Narrowing



What is a class?

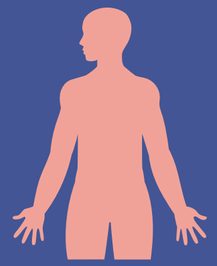
Template defining objects

```typescript
class Car {
  // Properties of the class should be declared at the top
  private brand: string;
  private model: string;
  private year: number;

  // Constructor
  constructor(brand: string, model: string, year: number) {
    this.brand = brand;
    this.model = model;
    this.year = year;
  }

  // Method to start the car
  public startEngine(): void {
    console.log(`Starting the engine of the ${this.brand} ${this.model}...`);
  }

}
```

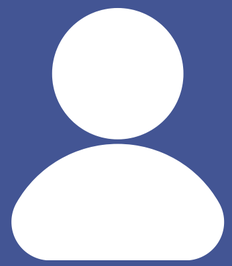
There is no ; at the end of the class definition

Relations between objects

Composition



Aggregation



Association

Inheritance

The magic word is **extend**

⚠️ Multiple inheritance does not exist

- Public
- Protected
- Private

```ts
class Animal {
    protected name: string;

    constructor(name: string) {
      this.name = name;
    }

    move(distanceInMeters: number): void {
      console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
  }
```

```ts
// Derived class
class Dog extends Animal {
  private numberOfBarks: number;

  constructor(name: string) {
    super(name);
    this.numberOfBarks = 0;
  }

  bark(): void {
    console.log('Woof! Woof!');
    this.numberOfBarks += 1;
  }

  showBarks(): void {
    console.log(`${this.name} has barked ${this.numberOfBarks} times.`);
  }
}
```

User Defined Types

```
function calculator(firstOperand: number, secondOperand: number, operation: string): number {
  switch (operation) {
    case 'sum':
      return firstOperand + secondOperand;
    case 'subtract':
      return firstOperand - secondOperand;
    case 'multiply':
      return firstOperand * secondOperand;
    default:
      throw new Error("Operation not supported");
  }
}
```

User Defined Types

```
1  function calculator(firstOperand: number, secondOperand: number, operation: string): number {
2    switch (operation) {
3      case 'sum':
4        return firstOperand + secondOperand;
5      case 'subtract':
6        return firstOperand - secondOperand;
7      case 'multiply':
8        return firstOperand * secondOperand;
9      default:
10       throw new Error("Operation not supported");
11   }
12 }
```

❌ Lack of control in the code

User Defined Types

```
function calculator(firstOperand: number, secondOperand: number, operation: 'sum' | 'subtract' | 'multiply'): number {
  switch (operation) {
    case 'sum':
      return firstOperand + secondOperand;
    case 'subtract':
      return firstOperand - secondOperand;
    case 'multiply':
      return firstOperand * secondOperand;
    default:
      //This line should never be reached if there is no user input.
      throw new Error("Operation not supported");
  }
}
console.log(calculator(2,3,'a'))    Argument of type '"a"' is not assignable to parameter of type '"sum" | "subtract" | "multiply"'.
```

User Defined Types

```
function calculator(firstOperand: number, secondOperand: number, operation: 'sum' | 'subtract' | 'multiply'): number {
  switch (operation) {
    case 'sum':
      return firstOperand + secondOperand;
    case 'subtract':
      return firstOperand - secondOperand;
    case 'multiply':
      return firstOperand * secondOperand;
    default:
      //This line should never be reached if there is no user input.
      throw new Error("Operation not supported");
  }
}
console.log(calculator(2,3,'a'))    Argument of type '"a"' is not assignable to parameter of type '"sum" | "subtract" | "multiply"'.
```

Not scalable or maintainable

User Defined Types

```ts
type Operation = 'sum' | 'subtract' | 'multiply'

function calculator(firstOperand: number, secondOperand: number, operation: Operation ): number {
  switch (operation) {
    case 'sum':
      return firstOperand + secondOperand;
    case 'subtract':
      return firstOperand - secondOperand;
    case 'multiply':
      return firstOperand * secondOperand;
    default:
      //This line should never be reached if there is no user input.
      throw new Error("Operation not supported");
  }
}
```

Code reuse and facilitates refactoring

Interfaces

It is a contract.

Its obligations and annexes must be complied with.

```typescript
interface Book { // No code was implemented in an interface
  title: string;
  author: string;
  yearPublished: number;
  // Method to display information about the book
  displayDetails(): string;
}

class Novel implements Book {
  title: string;
  author: string;
  yearPublished: number;

  constructor(title: string, author: string, yearPublished: number) {
    this.title = title;
    this.author = author;
    this.yearPublished = yearPublished;
  }

  displayDetails(): string {
    return `Novel: "${this.title}" by ${this.author}, published in ${this.yearPublished}.`;
  }
}

```

Interfaces

Another example

```ts
interface Algorithm { // No code was implemented in an interface
  name: string;
  run(): string;
}
```

```ts
class QuickSort implements Algorithm {
  name: string = "QuickSort";
  run(): string {
    return `QuickSort trace.`;
  }
}
```

```ts
class BubbleSort implements Algorithm {
  name: string = "BubbleSort";
  run(): string {
    return `BubbleSort trace.`;
  }
}
```

Generic Types

```
class Pair<T, U> {
    private first: T;
    private second: U;

    constructor(first: T, second: U) {
        this.first = first;
        this.second = second;
    }

    public getFirst(): T {
        return this.first;
    }

    public getSecond(): U {
        return this.second;
    }

    public setFirst(first: T): void {
        this.first = first;
    }

    public setSecond(second: U): void {
        this.second = second;
    }
}
```

- Reuse code

```
// Creating a Pair instance with two numbers
let point = new Pair<number, number>(5, 10);
```

```
// Creating a Pair instance with a string and a number
let labeledValue = new Pair<string, number>("Age", 30);
```

```
// Creating a Pair instance with an object and an array
let studentScores = new Pair<{ name: string }, number[]>({ name: "Alicia" }, [85, 92, 88]);
```

Narrowing

- It is used to inform the compiler about the most specific type that a variable can have at a given point in the code.

```
function padLeft(padding: number | string, input: string) {
  return " ".repeat(padding) + input;     Argument of type 'string
}
```

```
1  function padLeft(padding: number | string, input: string) {
2    if (typeof padding === "number") {
3      return " ".repeat(padding) + input;
4    }
5    return padding + input;
6  }
```

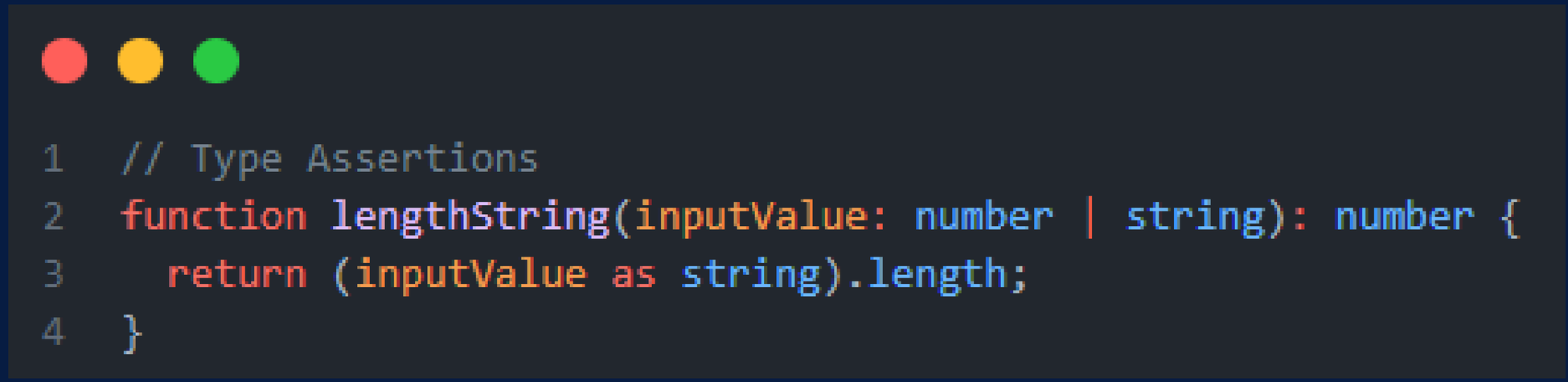
Narrowing

- **Type Guards:** Use type checks such as typeof, instanceof.

```typescript
// Type Guards
function printFormattedValue(inputValue: number | string): void {
    if (typeof inputValue === 'string') {
        // Here TypeScript knows that 'inputValue' is a 'string'
        console.log(inputValue.toUpperCase()); // This works because it's a 'string'
    } else {
        // Here TypeScript knows that 'inputValue' is a 'number'
        console.log(inputValue.toFixed(2)); // This works because it's a 'number'
    }
}
```

Narrowing

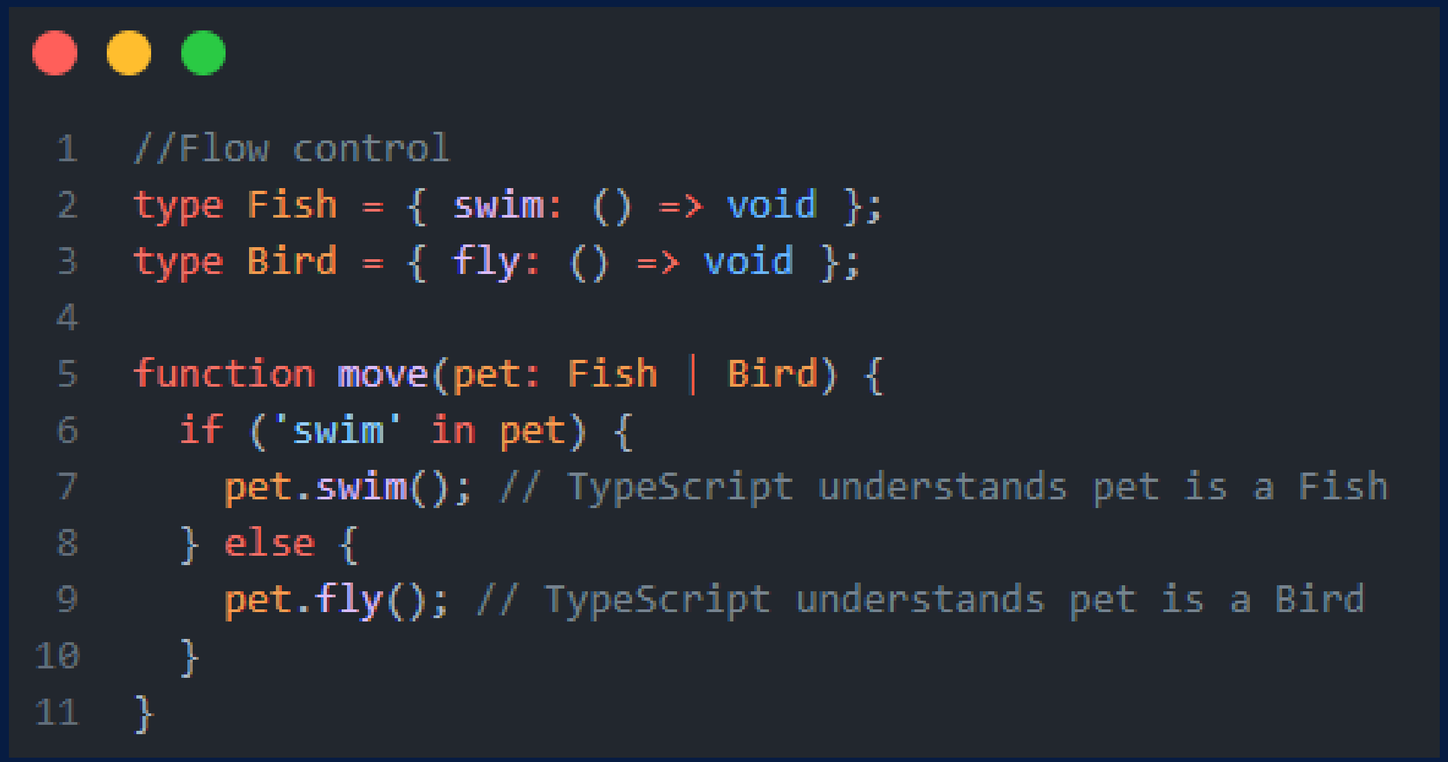
- **Type Assertions:** Explicitly assert that a variable is of a specific type.



```
// Type Assertions
function lengthString(inputValue: number | string): number {
  return (inputValue as string).length;
}
```

Narrowing

- **Flow Control:** Flow control is used to narrow down the type based on the logic of the program.



```ts
//Flow control
type Fish = { swim: () => void };
type Bird = { fly: () => void };

function move(pet: Fish | Bird) {
  if ('swim' in pet) {
    pet.swim(); // TypeScript understands pet is a Fish
  } else {
    pet.fly(); // TypeScript understands pet is a Bird
  }
}
```

Narrowing

- **Type Predicates:** Used in functions to explicitly indicate the return type under certain conditions.

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

function movePet(pet: Fish | Bird) {
  if (isFish(pet)) {
      pet.swim(); // Here, pet is treated as a Fish
  } else {
      pet.fly(); // Here, pet is treated as a Bird
  }
}
```

Bibliography

- TutorialsTeacher. (n.d.). Learn TypeScript. Retrieved January 25, 2024, from https://www.tutorialsteacher.com/typescript

- TutorialsTeacher. (n.d.). TypeScript - Classes. Retrieved January 25, 2024, from https://www.tutorialsteacher.com/typescript/typescript-class

- hdeleon.net. (7 dic 2020). ¿Para qué sirven las Interfaces en Typescript? [Video]. YouTube. https://www.youtube.com/watch?v=F9TR4EX5kQg

- minudev. (25 mar 2022). TypeScript - Tutorial desde CERO en Español 🏆 [Video]. YouTube. https://www.youtube.com/watch?v=xtp_DuPxo9Q

- TypeScript. (n.d.). Narrowing. In TypeScript Handbook. Retrieved January 25, 2024, from https://www.typescriptlang.org/docs/handbook/2/narrowing.html

- TypeScript. (n.d.). TypeScript Documentation. Retrieved January 25, 2024, from https://www.typescriptlang.org/docs/

- Google. (n.d.). TypeScript Style Guide. Retrieved January 25, 2024, from https://google.github.io/styleguide/tsguide.html