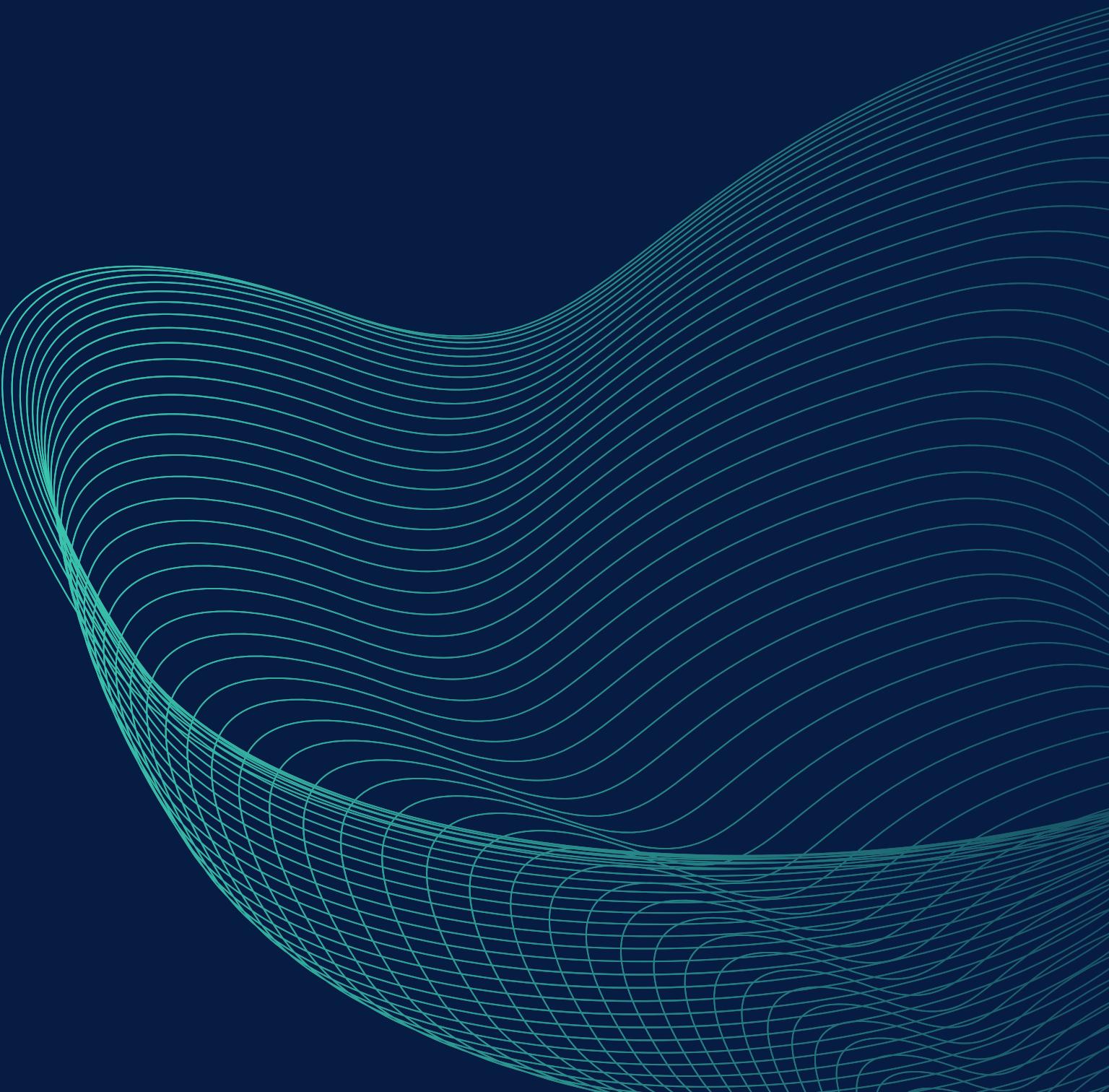


TypeScript



Equipo de trabajo



Evian Concepción Peña
alu0101395548@ull.edu.es



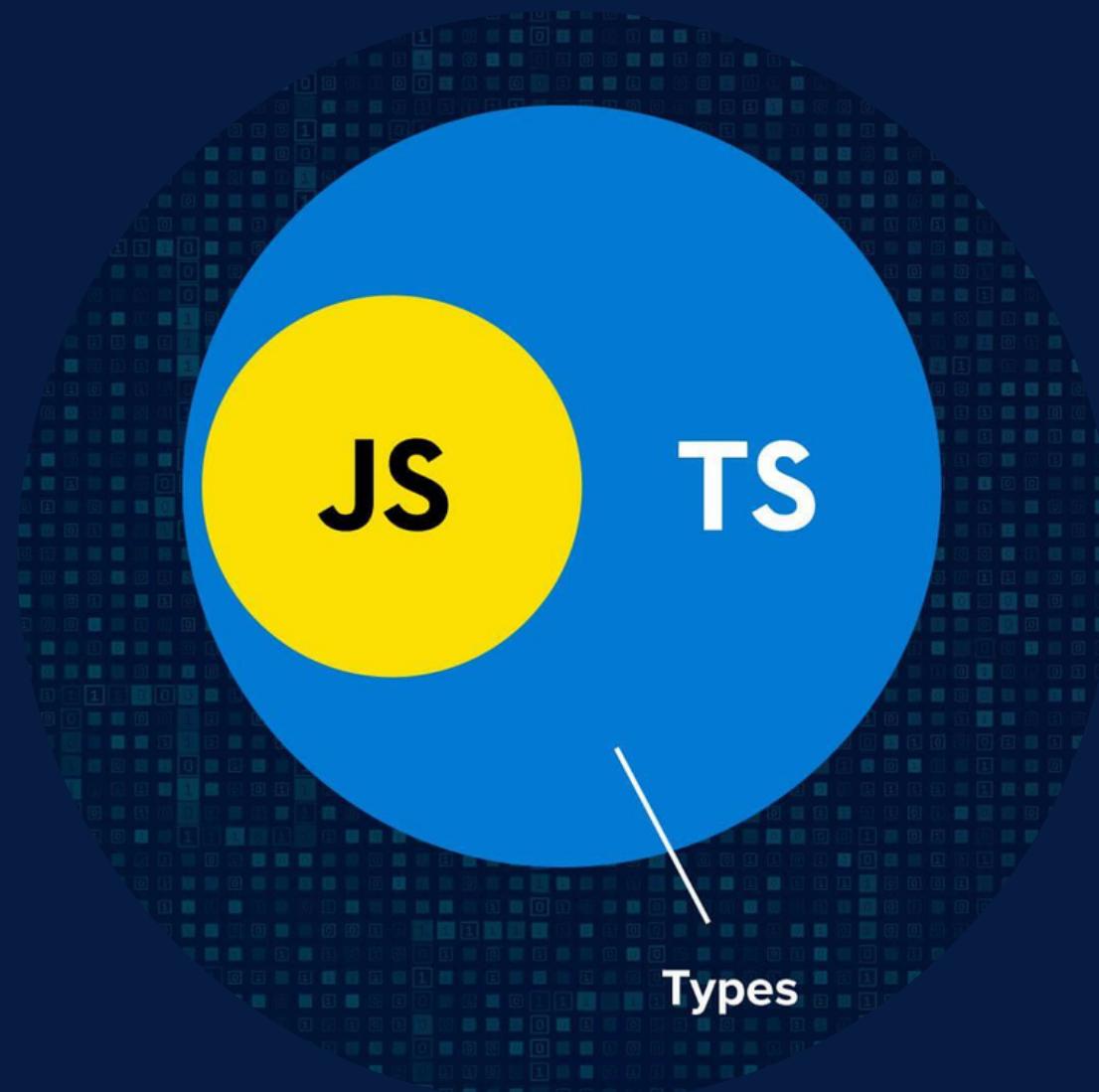
Saúl Sosa Díaz
alu0101404141@ull.edu.es

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

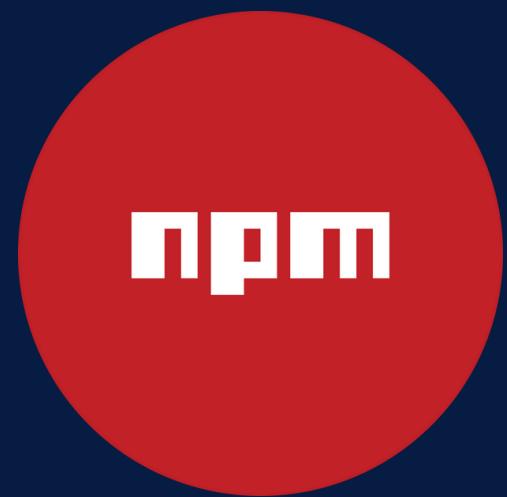


Source code JS



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



```
1 npm install -g typescript
```



```
1 tsc -v
```

Compiler

Second choice



Use ts-node

- It compiles and runs with a single command.



Installation



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



```
1 ts-node -v
```

Start guide

1. Create Node.js project



```
1 npm init -y
```

2. Create TypeScript compilation configuration



```
1 tsc --init
```

```
● usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAIS$ npm init -y
Wrote to /home/usuario/Presentation_PAIS/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"
}
```

```
● usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAIS$ 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

```
1  {  
2    "compilerOptions": {  
3      "target": "es2016",  
4      "strict": true,  
5      "noImplicitAny": true,  
6      "outDir": "./dist",  
7      "module": "commonjs",  
8      "esModuleInterop": true,  
9      "forceConsistentCasingInFileNames": true,  
10     "skipLibCheck": true  
11   }  
12 }
```

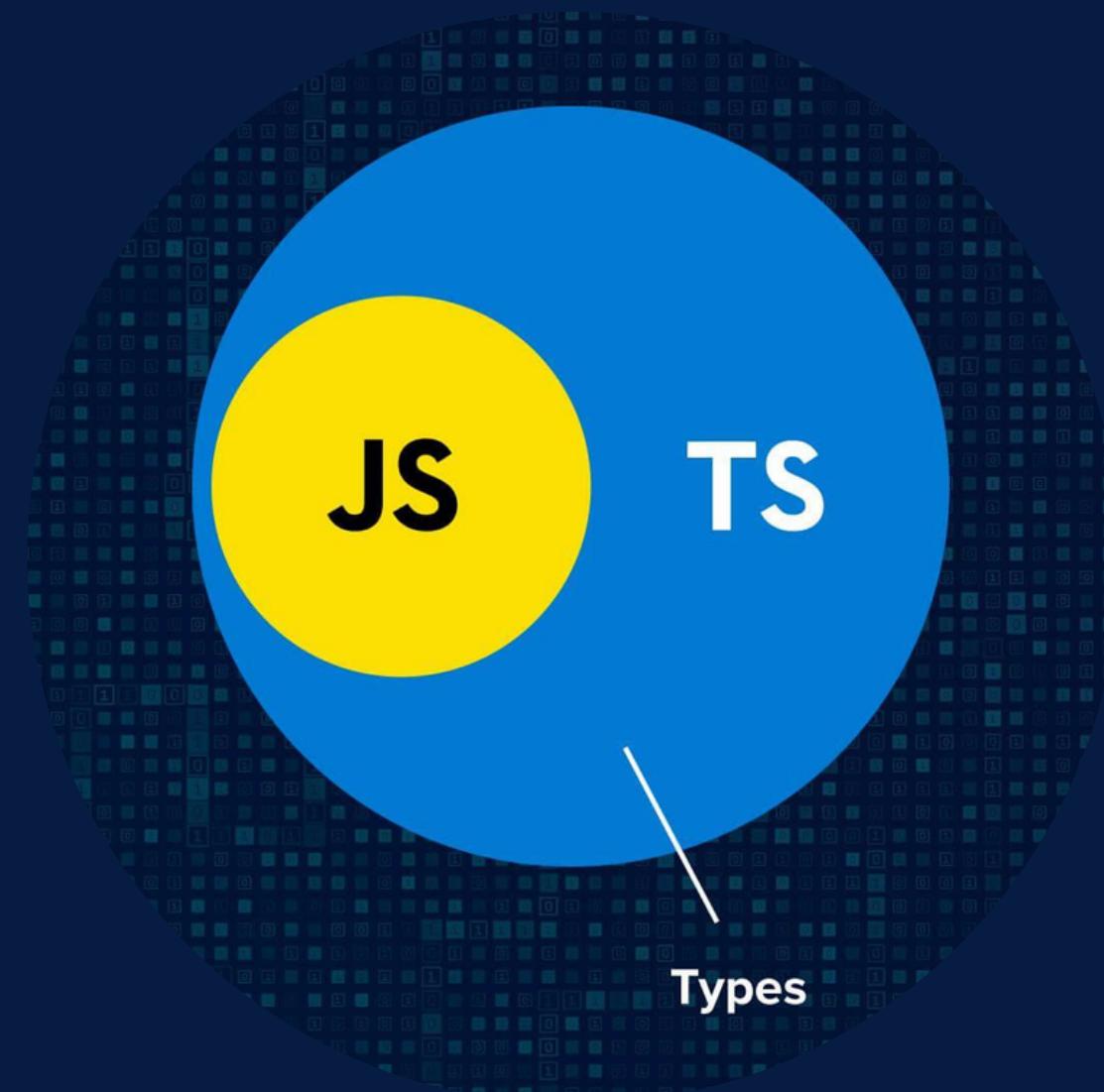
Hello world

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

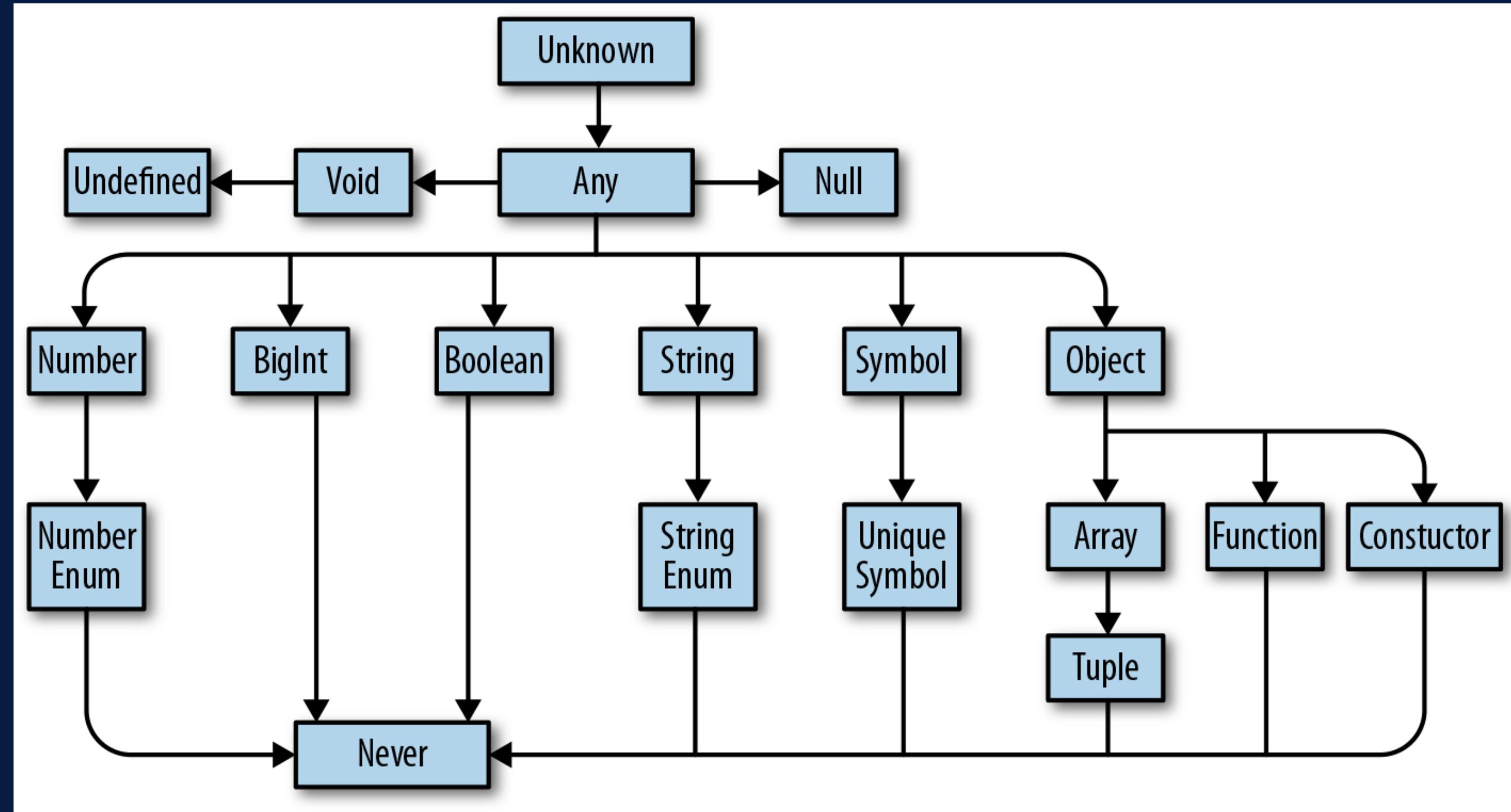
- usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAI/src/primerosPasos\$ tsc ./hello-world.ts && node ./hello-world.js
Hello world!
- usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAI/src/primerosPasos\$ ts-node ./hello-world.ts
Hello world!

Basic Concepts

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



Data types



Variable declaration

Global variable

- Not recommended, actually forbidden for us

Constants

- Use if no modification needed

Scope variable

- Recommended



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



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



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

Variable declaration

Arrays

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



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

Strings

- Preferable (' ') over ("")



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

Create objects

- always use new, don't worry, TS uses garbage collection



```
1 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



```
1 const entry: string[] = process.argv;
2 console.log(entry);
3 // entry[0] path to ts-node
4 // entry[1] path to the file
5 // entry[2] begining of arguments
6
7 // load de required module
8 const readLine = require('readline');
9 const readUserInput = readLine.createInterface({
10   input: process.stdin,
11   output: process.stdout
12 });
13
14 readUserInput.question('Write something: ', (entry: string) => {
15   console.log(`Ha ingresado: ${entry}`);
16   // Puedes guardar la entrada en una variable o
17   // en algún lugar según tus necesidades.
18   readUserInput.close();
19 });
```

Conditionals

switch

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



```
1 let randomVariable = 40;
2 const randomVariable2: number = 50;
3 switch (randomVariable) {
4     case 10:
5         // CODE
6         break;
7     case 20:
8         // CODE
9         break;
10    case randomVariable2:
11        // CODE
12        break;
13    default:
14        // GOOD PRACTICE
15        break;
16 }
```

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



```
1 let randomVariable = 40;
2 const randomVariable2: number = 50;
3
4 if (randomVariable > randomVariable2) {
5
6 } else if (randomVariable < randomVariable2) {
7
8 } else {
9
10 }
```

Loops

Classic for

- Customizable

for of

- Safe iteration

for in

- Use for index

for each

- Based in functional programming



```
1  const randomVector: number[] = [1, 2, 3];
2  const vectorSize: number = randomVector.length;
3
4  for (let index = 0; index < vectorSize; ++index) {
5      // CODE
6  }
7
8  for (const element of randomVector) {
9      // CODE
10 }
11
12 for (const index in randomVector) {
13     // CODE
14 }
```

Loops

while

- Be careful with infinite loops

do while

- executed at least once

continue

- skips to the next iteration

break

- finish the loop



```
1 let condition: boolean = false;
2 let condition2: boolean = false;
3 while (condition) {
4     if (condition2) {
5         continue;
6     }
7 }
8
9 do {
10     if (condition2) {
11         break;
12     }
13 } while (condition);
```

Operators

Arithmetic

- +
- -
- *
- /
- %
- ++
- --

Assignation

- =
- +=
- -=
- /=
- *=

Operators

Logic

- `&&`
- `||`
- `!`

Concatenation

- `+`

Search

- `in`
simple and elegant

Analyze types

- `instanceof`
- `typeof`

Operators

Relational

- >
- <
- >=
- <=
- ==
- ===
- !=
- !==

Ternary

- ?

Bitwise

- &
- |
- ^
- ~
- <<
- >>
- >>>

Functions

Basic

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

Arrow

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



```
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

import

- if only one thing needed avoid using “*”

circular dependency

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

```
● ● ●  
1 export class dummy {  
2   constructor() {}  
3   public hello(): void { console.log("Hello"); }  
4 }  
5  
6 export function hello(): void { console.log("Hello"); }
```

```
● ● ●  
1 // import { dummy } from './a';  
2 // import { hello } from './a';  
3 import * as testing from './exporting';  
4 import { dummy, hello } from './exporting';  
5  
6 let temp: dummy = new dummy();  
7 temp.hello();  
8 hello();  
9 testing.hello();
```

Read/Write from a file

Read

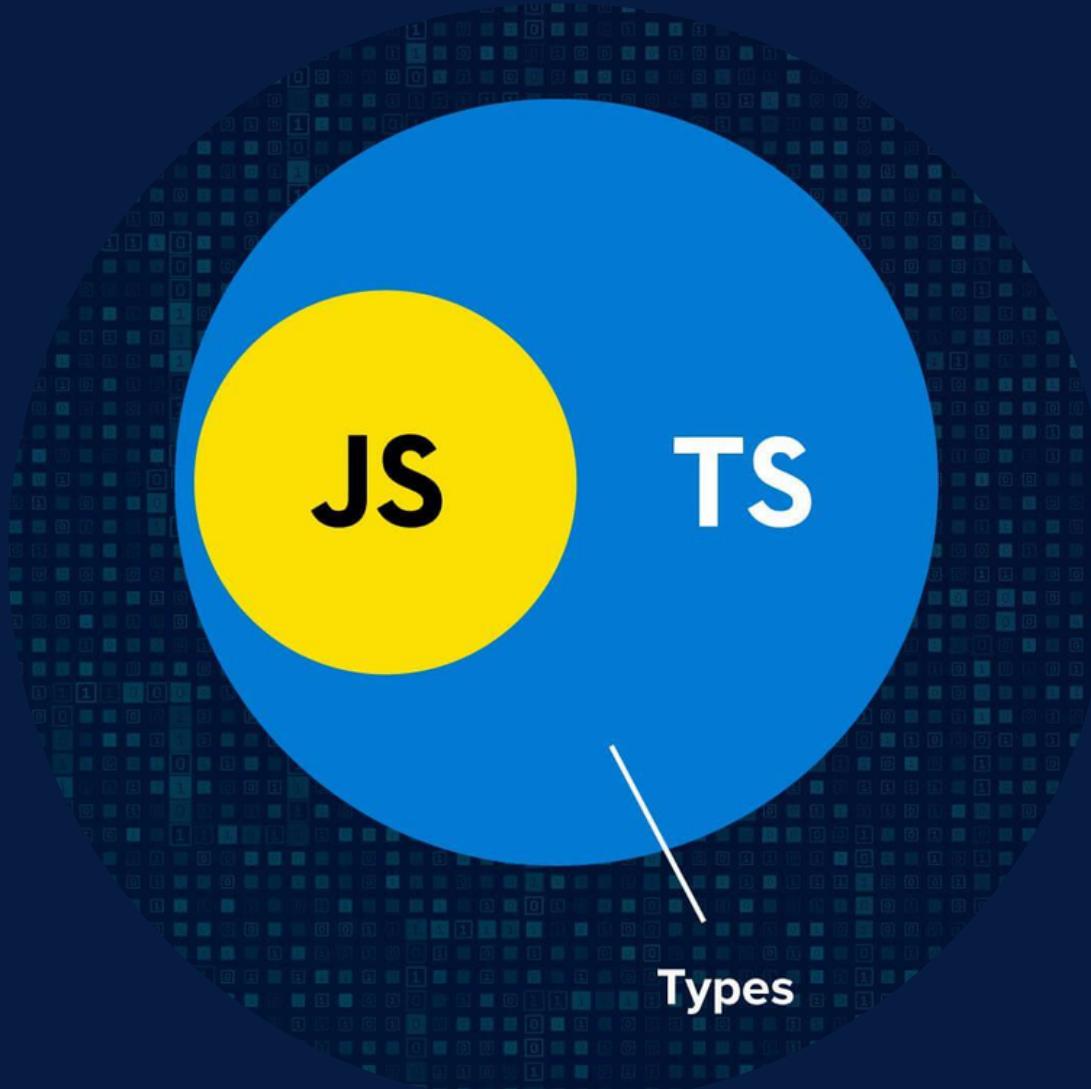
```
● ● ●  
1 import * as fs from 'fs';  
2 import * as readline from 'readline';  
3  
4 const filePath: string = 'data.txt';  
5  
6 // read the whole file  
7 fs.readFile(filePath, 'utf8', (error, data) => {  
8   if (error) {  
9     console.error('Error reading file: ${err}');  
10  } else {  
11    console.log(`\nFile content:\n`, data);  
12  }  
13});  
14 // read line by line  
15 const readLine = readline.createInterface({  
16  input: fs.createReadStream(filePath),  
17  crlfDelay: Infinity,  
18});  
19  
20 readLine.on('line', (line) => {  
21  console.log('Line:', line);  
22});  
23  
24 readLine.on('close', () => {  
25  console.log('File reading complete.');  
26});  
27})
```

Write

```
● ● ●  
1 import * as fs from 'fs';  
2  
3 const filePath: string = 'data.txt';  
4  
5 const dataToWrite: string = 'NEW LINE';  
6  
7 fs.writeFile(filePath, dataToWrite, { flag : 'a' }, (error) => {  
8   if (error) {  
9     console.error('Error reading file: ${err}');  
10  } else {  
11    console.log('Write operation complete.');  
12  }  
13});
```

Advanced concepts

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



What is a class?

Template defining objects



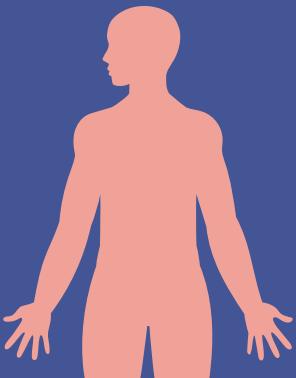
```
1  class Car {  
2      // Properties of the class should be declared at the top  
3      private brand: string;  
4      private model: string;  
5      private year: number;  
6  
7      // Constructor  
8      constructor(brand: string, model: string, year: number) {  
9          this.brand = brand;  
10         this.model = model;  
11         this.year = year;  
12     }  
13  
14     // Method to start the car  
15     public startEngine(): void {  
16         console.log(`Starting the engine of the ${this.brand} ${this.model}...`);  
17     }  
18  
19 }
```



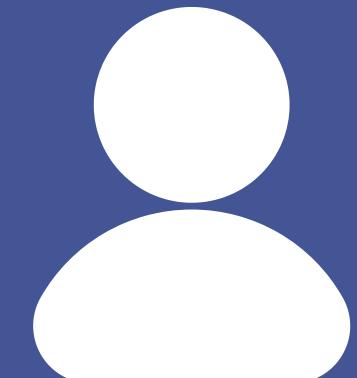
There is no ; at the end of the class definition

Relations between objects

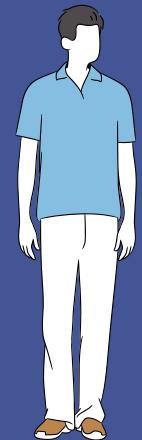
Composition



Aggregation



Association



Inheritance

The magic word is **extend**



Multiple inheritance does not exist

- Public
- Protected
- Private

```
● ● ●  
1 class Animal {  
2     protected name: string;  
3  
4     constructor(name: string) {  
5         this.name = name;  
6     }  
7  
8     move(distanceInMeters: number): void {  
9         console.log(`${this.name} moved ${distanceInMeters}m.`);  
10    }  
11}
```

```
● ● ●  
1 // Derived class  
2 class Dog extends Animal {  
3     private numberOfBarks: number;  
4  
5     constructor(name: string) {  
6         super(name);  
7         this.numberOfBarks = 0;  
8     }  
9  
10    bark(): void {  
11        console.log('Woof! Woof!');  
12        this.numberOfBarks += 1;  
13    }  
14  
15    showBarks(): void {  
16        console.log(`${this.name} has barked ${this.numberOfBarks} times.`);  
17    }  
18}
```

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    }
```

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



```
1 type Operation = 'sum' | 'subtract' | 'multiply'  
2  
3 function calculator(firstOperand: number, secondOperand: number, operation: Operation ): number {  
4     switch (operation) {  
5         case 'sum':  
6             return firstOperand + secondOperand;  
7         case 'subtract':  
8             return firstOperand - secondOperand;  
9         case 'multiply':  
10            return firstOperand * secondOperand;  
11        default:  
12            //This line should never be reached if there is no user input.  
13            throw new Error("Operation not supported");  
14    }  
15 }
```



Code reuse and facilitates refactoring

Interfaces

It is a contract.

Its obligations and annexes must be complied with.



```
1 interface Book { // No code was implemented in an interface
2   title: string;
3   author: string;
4   yearPublished: number;
5   // Method to display information about the book
6   displayDetails(): string;
7 }
8
9 class Novel implements Book {
10   title: string;
11   author: string;
12   yearPublished: number;
13
14   constructor(title: string, author: string, yearPublished: number) {
15     this.title = title;
16     this.author = author;
17     this.yearPublished = yearPublished;
18   }
19
20   displayDetails(): string {
21     return `Novel: "${this.title}" by ${this.author}, published in ${this.yearPublished}.`;
22   }
23 }
24
```

Interfaces

Another example



```
1 interface Algorithm { // No code was implemented in an interface
2   name: string;
3   run(): string;
4 }
```



```
1 class QuickSort implements Algorithm {
2   name: string = "QuickSort";
3   run(): string {
4     return `QuickSort trace.`;
5   }
6 }
```



```
1 class BubbleSort implements Algorithm {
2   name: string = "BubbleSort";
3   run(): string {
4     return `BubbleSort trace.`;
5   }
6 }
```

Generic Types



```
1 class Pair<T, U> {
2     private first: T;
3     private second: U;
4
5     constructor(first: T, second: U) {
6         this.first = first;
7         this.second = second;
8     }
9
10    public getFirst(): T {
11        return this.first;
12    }
13
14    public getSecond(): U {
15        return this.second;
16    }
17
18    public setFirst(first: T): void {
19        this.first = first;
20    }
21
22    public setSecond(second: U): void {
23        this.second = second;
24    }
25 }
```

- Reuse code



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



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



```
1 // Creating a Pair instance with an object and an array
2 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`.



```
1 // Type Guards
2 function printFormattedValue(inputValue: number | string): void {
3     if (typeof inputValue === 'string') {
4         // Here TypeScript knows that 'inputValue' is a 'string'
5         console.log(inputValue.toUpperCase()); // This works because it's a 'string'
6     } else {
7         // Here TypeScript knows that 'inputValue' is a 'number'
8         console.log(inputValue.toFixed(2)); // This works because it's a 'number'
9     }
10 }
```

Narrowing

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

```
● ● ●  
1 // Type Assertions  
2 function lengthString(inputValue: number | string): number {  
3   return (inputValue as string).length;  
4 }
```

Narrowing

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



```
1 //Flow control
2 type Fish = { swim: () => void };
3 type Bird = { fly: () => void };
4
5 function move(pet: Fish | Bird) {
6     if ('swim' in pet) {
7         pet.swim(); // TypeScript understands pet is a Fish
8     } else {
9         pet.fly(); // TypeScript understands pet is a Bird
10    }
11 }
```

Narrowing

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



```
1 // Type predicates
2 function isFish(pet: Fish | Bird): pet is Fish {
3     return (pet as Fish).swim !== undefined;
4 }
5
6 function movePet(pet: Fish | Bird) {
7     if (isFish(pet)) {
8         pet.swim(); // Here, pet is treated as a Fish
9     } else {
10        pet.fly(); // Here, pet is treated as a Bird
11    }
12 }
```

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>