TypeScript

Team

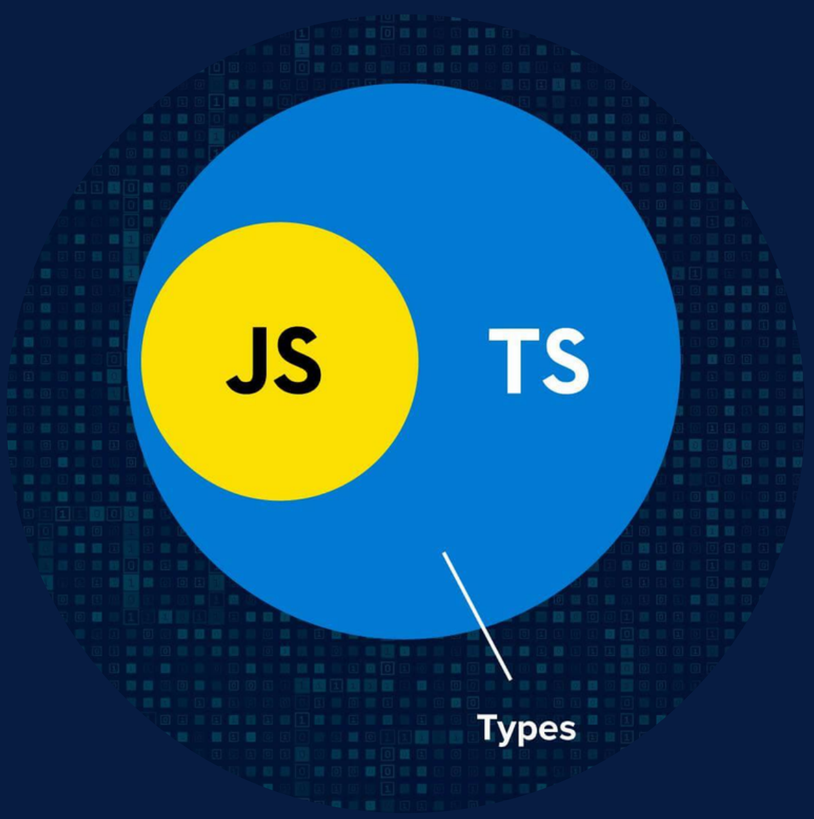
Evian Concepción Peña
alu0101395548@ull.edu.es

Saúl Sosa Díaz
alu0101404141@ull.edu.es

Index

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"
}
```

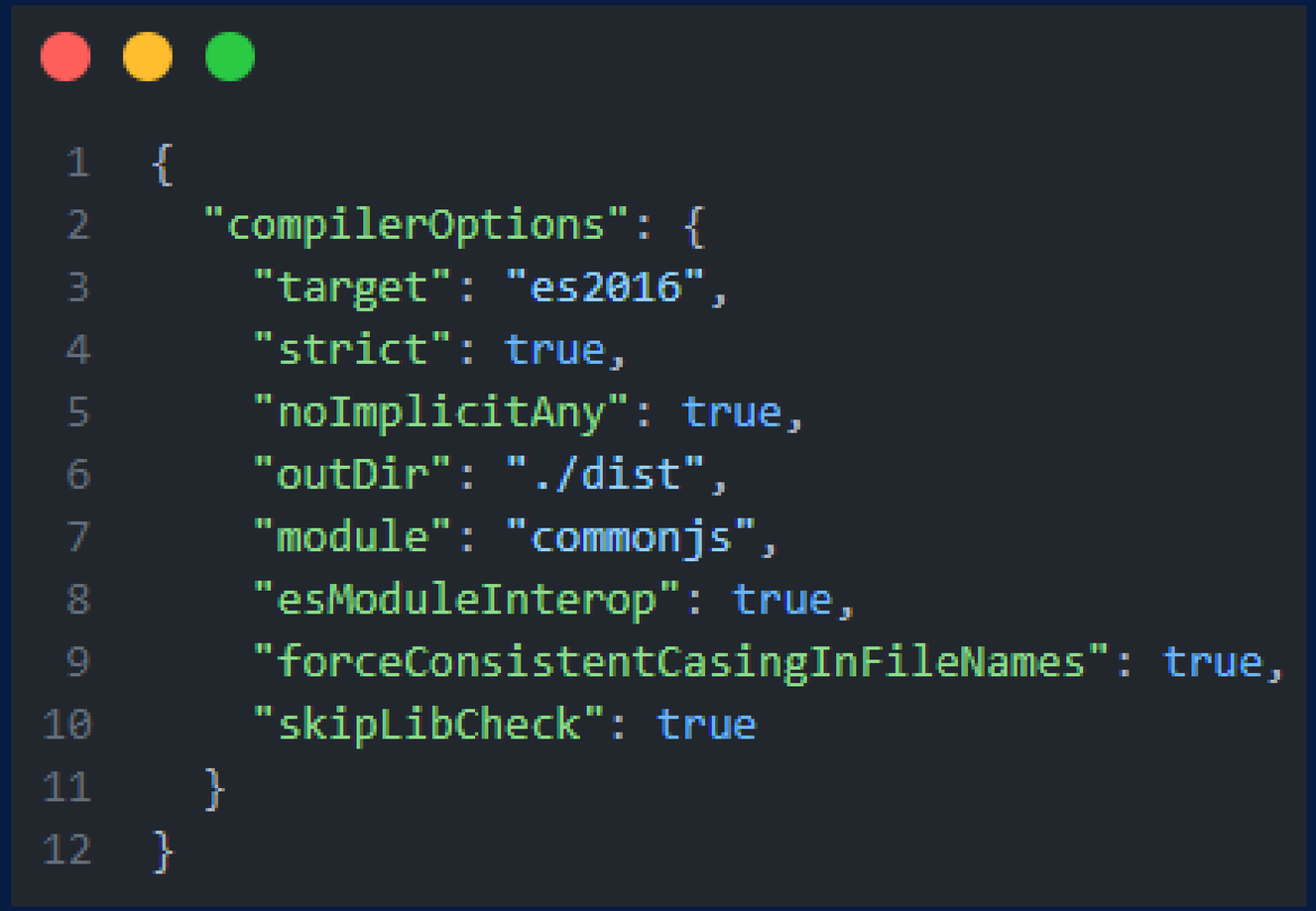
2. Create TypeScript compilation configuration

```
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
  }
}
```

A tipic TSconfig.json

```
1
2  {
3      "compilerOptions": {
4        "module": "system",                          // Specify module code generation method: 'none', 'commonjs', 'amd', 'system'....
5        "noImplicitAny": true,                       // Raise error on expressions and declarations with an implied 'any' type.
6        "removeComments": true,                      // Do not emit comments to output.
7        "preserveConstEnums": true,                  // Do not erase const enum declarations in generated code.
8        "outFile": "../../built/local/tsc.js",       // Concatenate and emit output to single file.
9        "sourceMap": true,                           // Generate corresponding '.map' file.
10       "target": "es5",                             // Specify ECMAScript target version: 'ES3' (default), 'ES5', 'ES6', etc.
11       "strict": true,                              // Enable all strict type-checking options.
12       "forceConsistentCasingInFileNames": true,    // Disallow inconsistently-cased references to the same file.
13       "skipLibCheck": true                        // Skip type checking of declaration files.
14     },
15     "include": ["src/**/*"],                       // Include all files under the 'src' folder.
16     "exclude": ["**/*.spec.ts"]                    // Exclude files ending with .spec.ts
17 }
```

you can see all the options at https://www.typescriptlang.org/tsconfig

Hello world

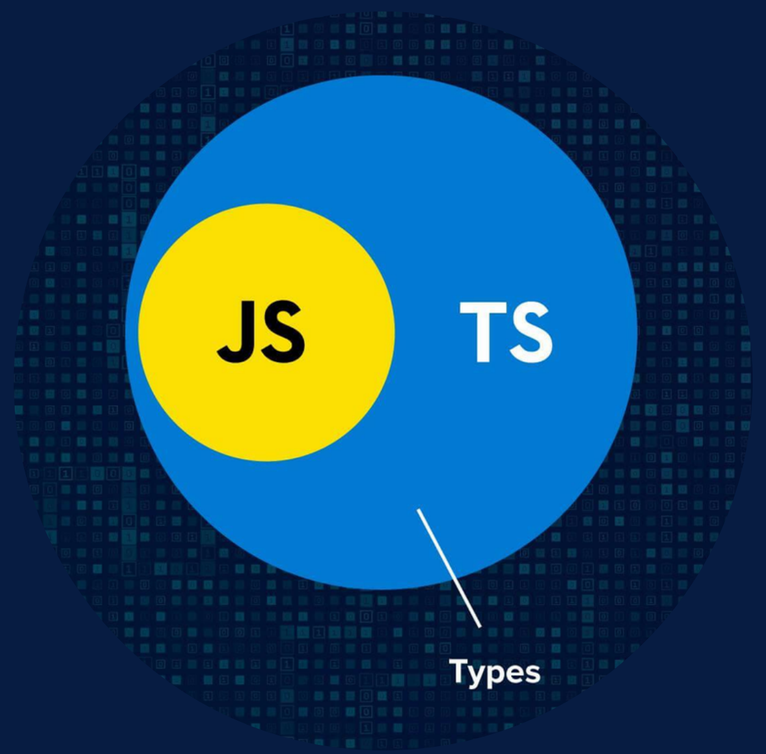
```ts
function helloWorld(): void {
  console.log("Hello world!");
}

helloWorld();
```

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

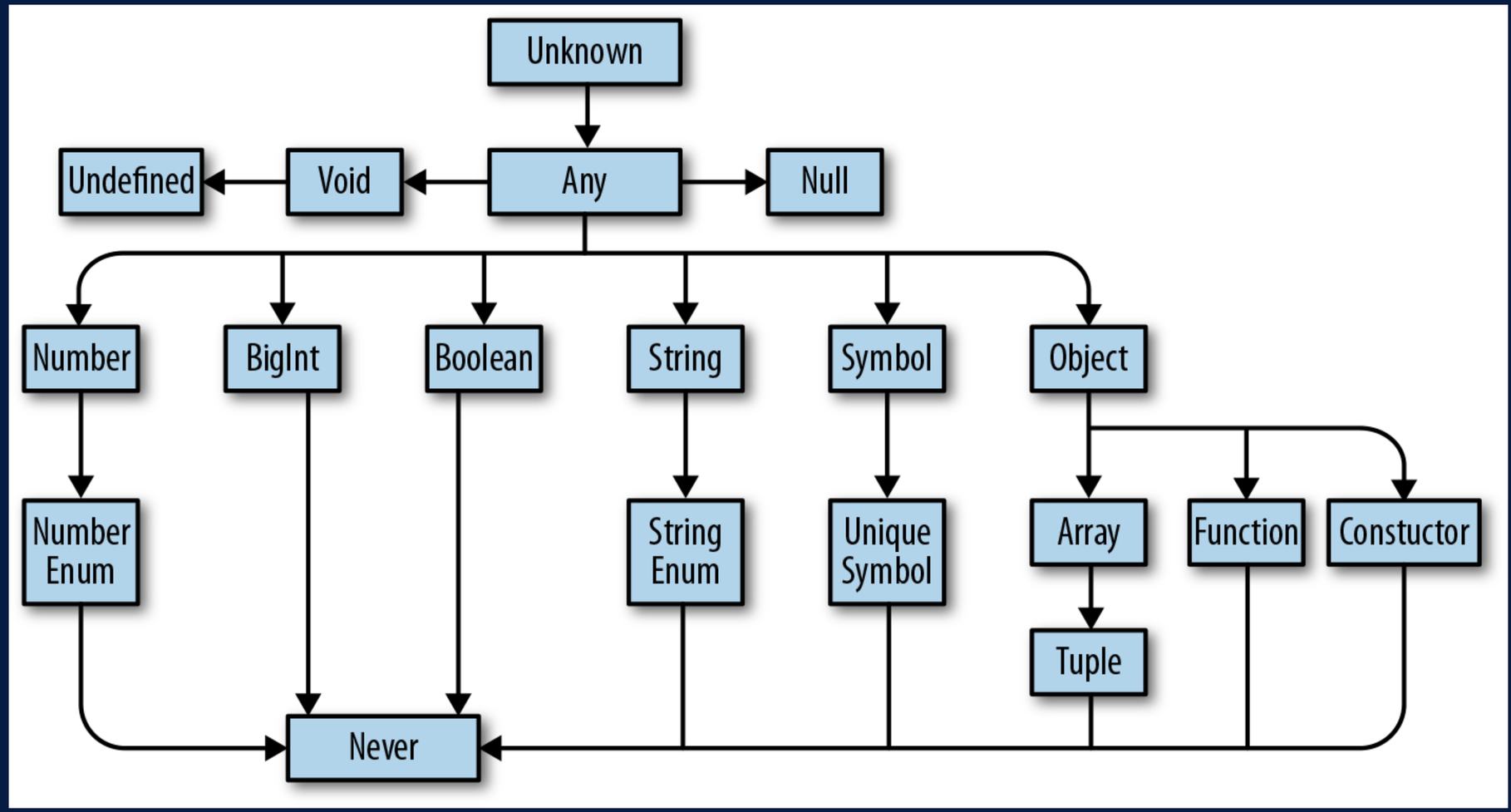
```
usuario@Ubuntu-18-PAI-SSD:~/Presentation_PAI/src/primerosPasos$ ts-node ./hello-world.ts
Hello world!
```

Basic Concepts

- Data types
- Variable declaration
- User inputs
- Conditionals
- Loops
- Operators
- Functions
- Read/Write from a file
- And many more

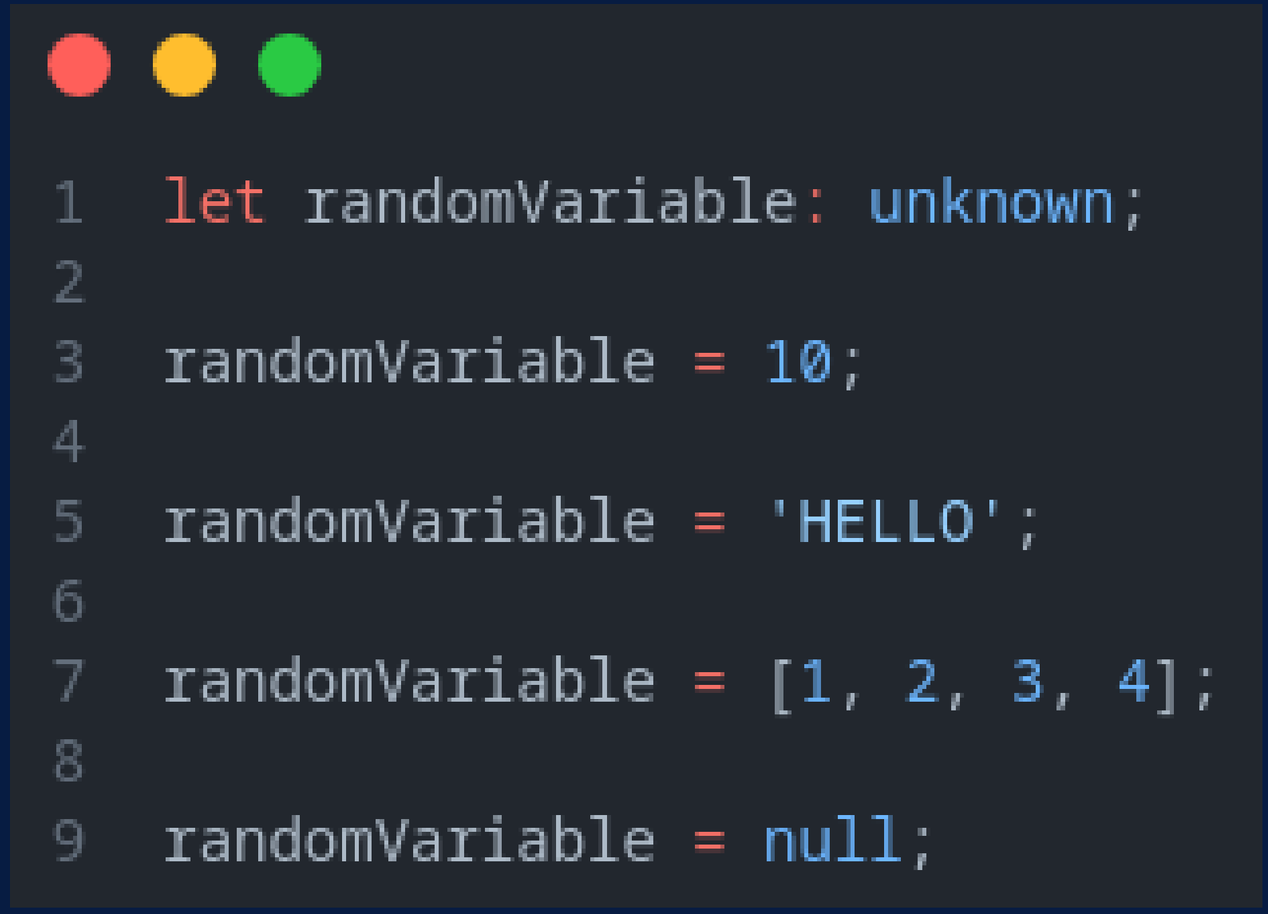


Data types


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

unkown

- Forbidden for us.
- All types are assignable.
- NOT a good practice.
- Leads to bugs.
- Basically going back to JavaScript



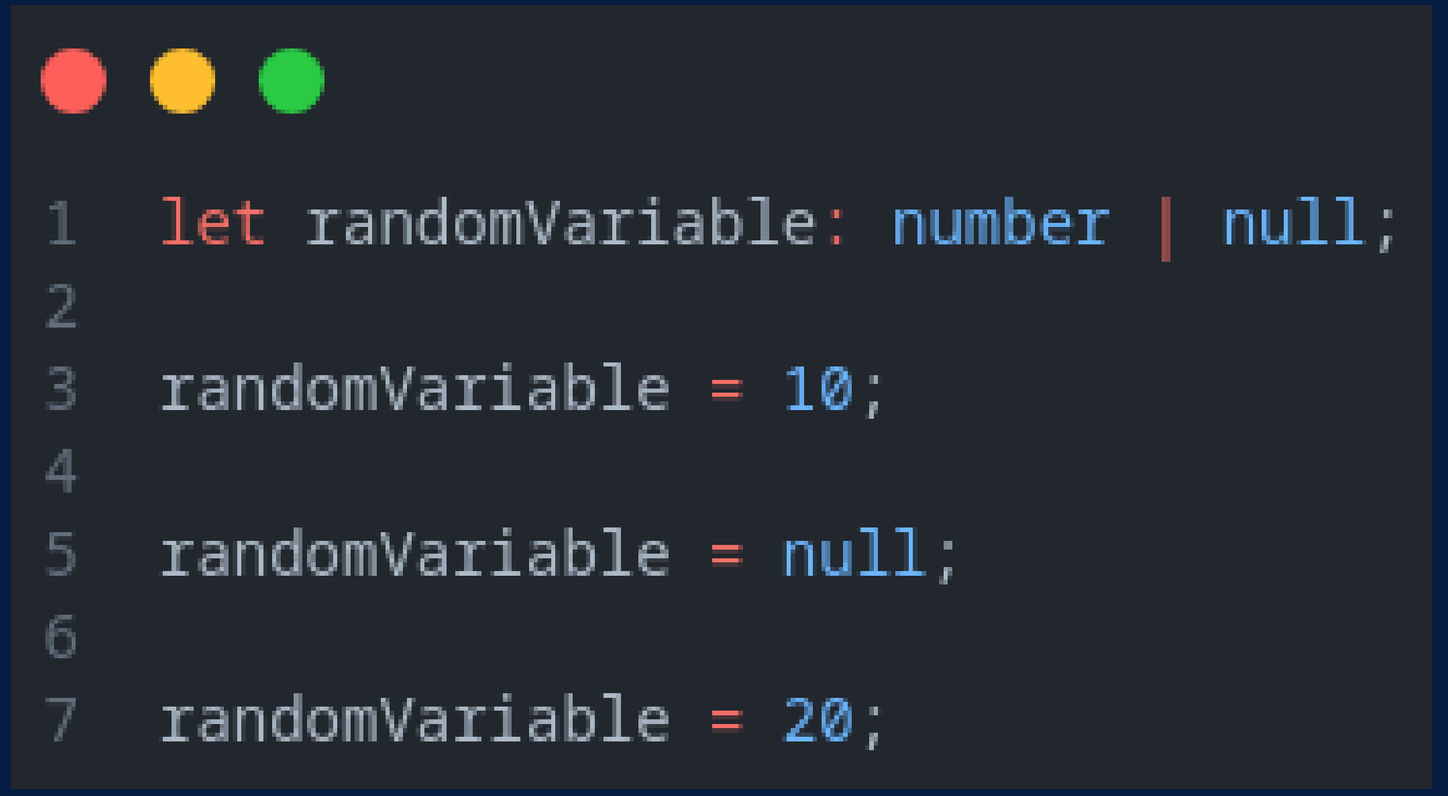
any

- Forbidden for us.
- Deactivates type restrictions.
- NOT a good practice.
- Leads to bugs.
- Basically going back to JavaScript.

```
let randomVariable: any;

randomVariable = 10;

randomVariable = 'HELLO';

randomVariable = [1, 2, 3, 4];

randomVariable = null;
```

null

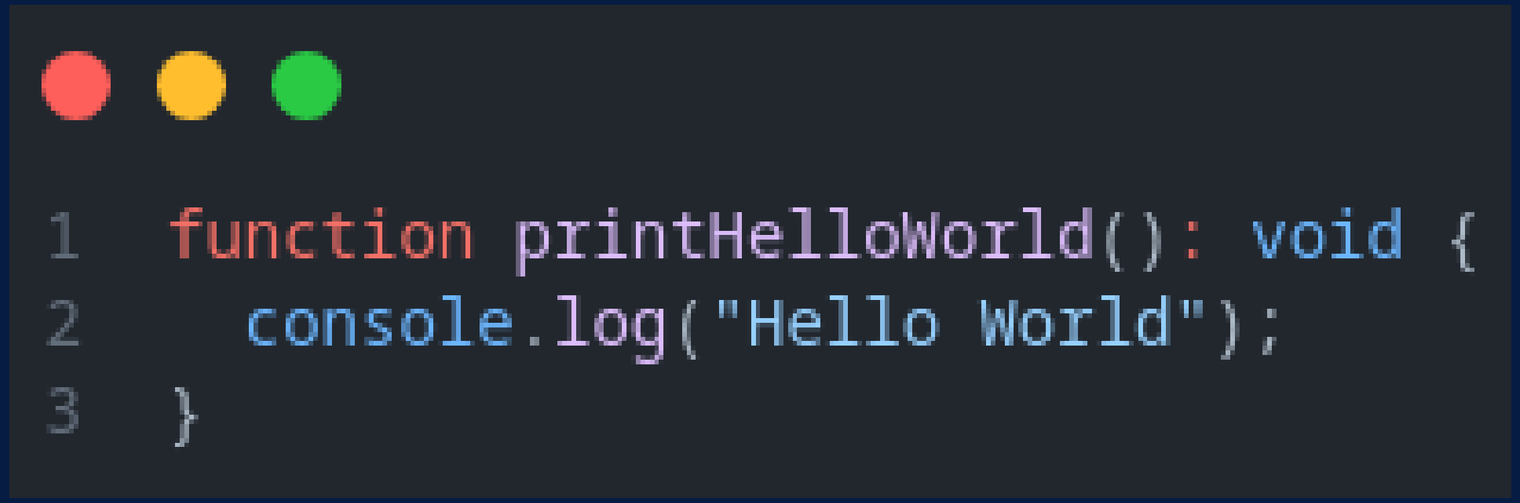
- Useful for data structures.
- Intentional absence of a value.



```
let randomVariable: number | null;

randomVariable = 10;

randomVariable = null;

randomVariable = 20;
```

void

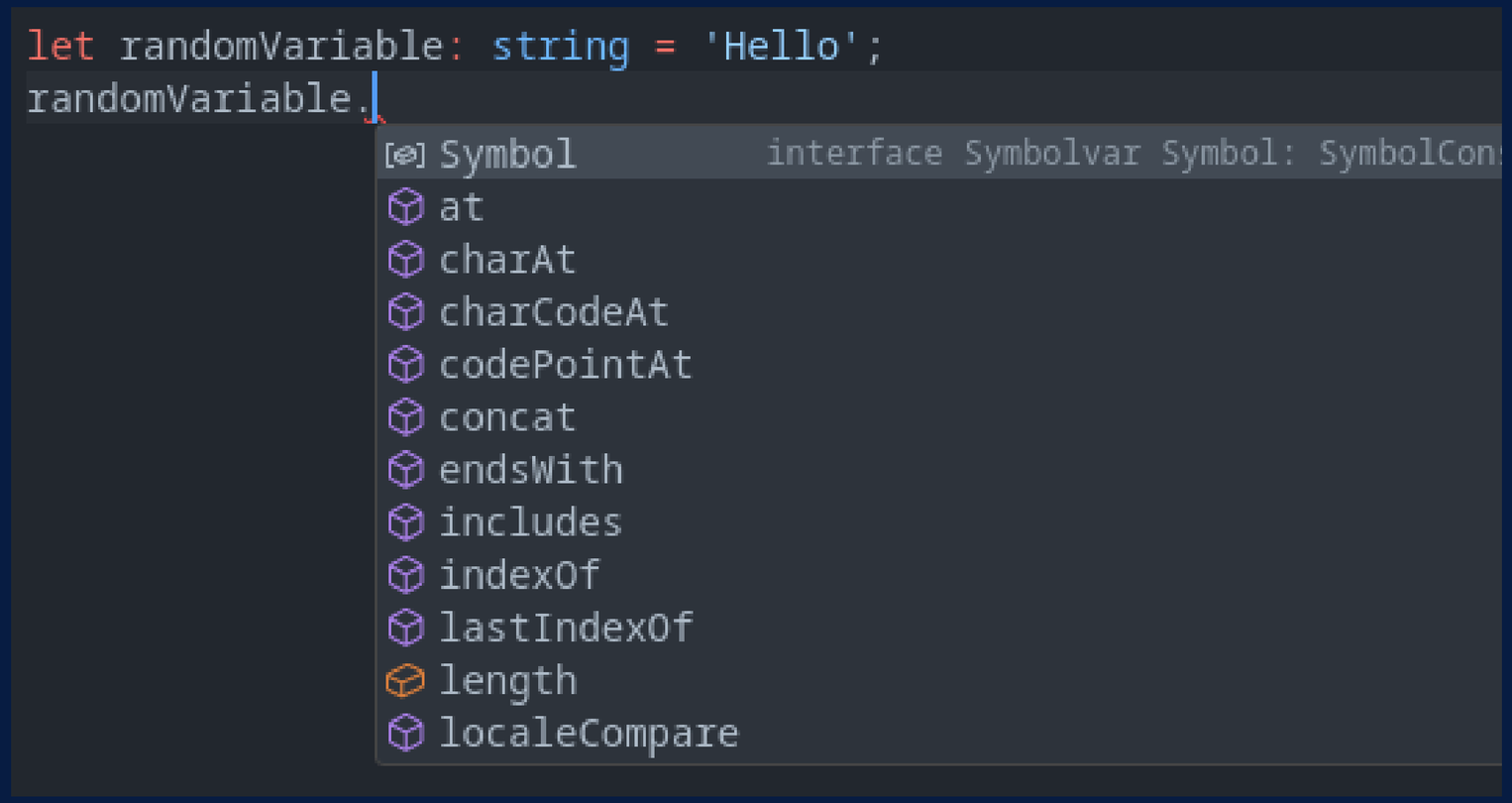
- You can use void as the return type in functions that do not return any explicit value.



```
function printHelloWorld(): void {
  console.log("Hello World");
}
```

built-in methods

- Don´t reinvent the wheel.
- They are useful and efficient.
- Reduces coding time.



enum

- Set of constants.
- They are useful when you got related constants.

```
enum Direction {
  NORTH,
  SOUTH,
  EAST,
  WEST
}

let direction: Direction = Direction.NORTH; // 0
console.log(Direction[direction]); // NORTH
```

symbol

- They are unique.
- Useful as properties to mark an unique object for example.

```
let symbol: symbol = Symbol('optional');
let symbol2: symbol = Symbol('optional');
if (symbol === symbol2) {
  console.log('NEVER WILL BE TRUE');
}
```

object

- Avoid if possible.
- Any class can be assigned to the object type but not primitive types.

```
let object1: object = new Set<number>();

object1 = new Map<number, number>();

object1 = 10;
```

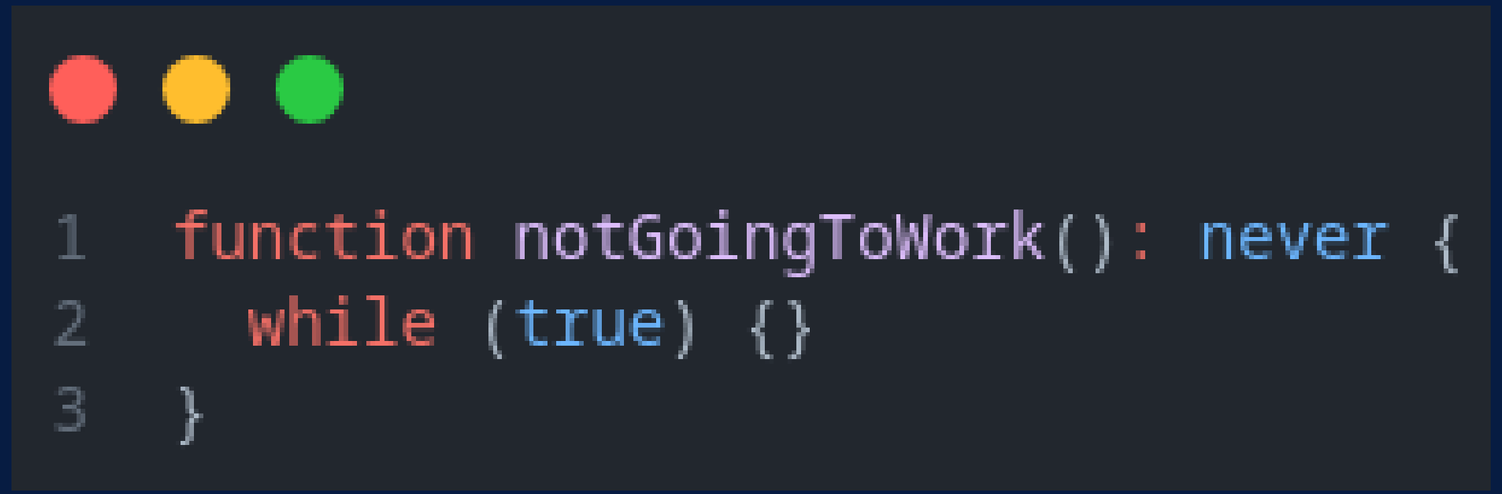
tuple

- Avoid modifying the length.
- The implementation does not makes sense.

```ts
const myTuple: [number, string] = [10, 'Hello'];

console.log(myTuple[0]);
console.log(myTuple[1]);

myTuple[0] = 20;
myTuple.push(100);

console.log(myTuple);

myTuple[2] = 200;
```

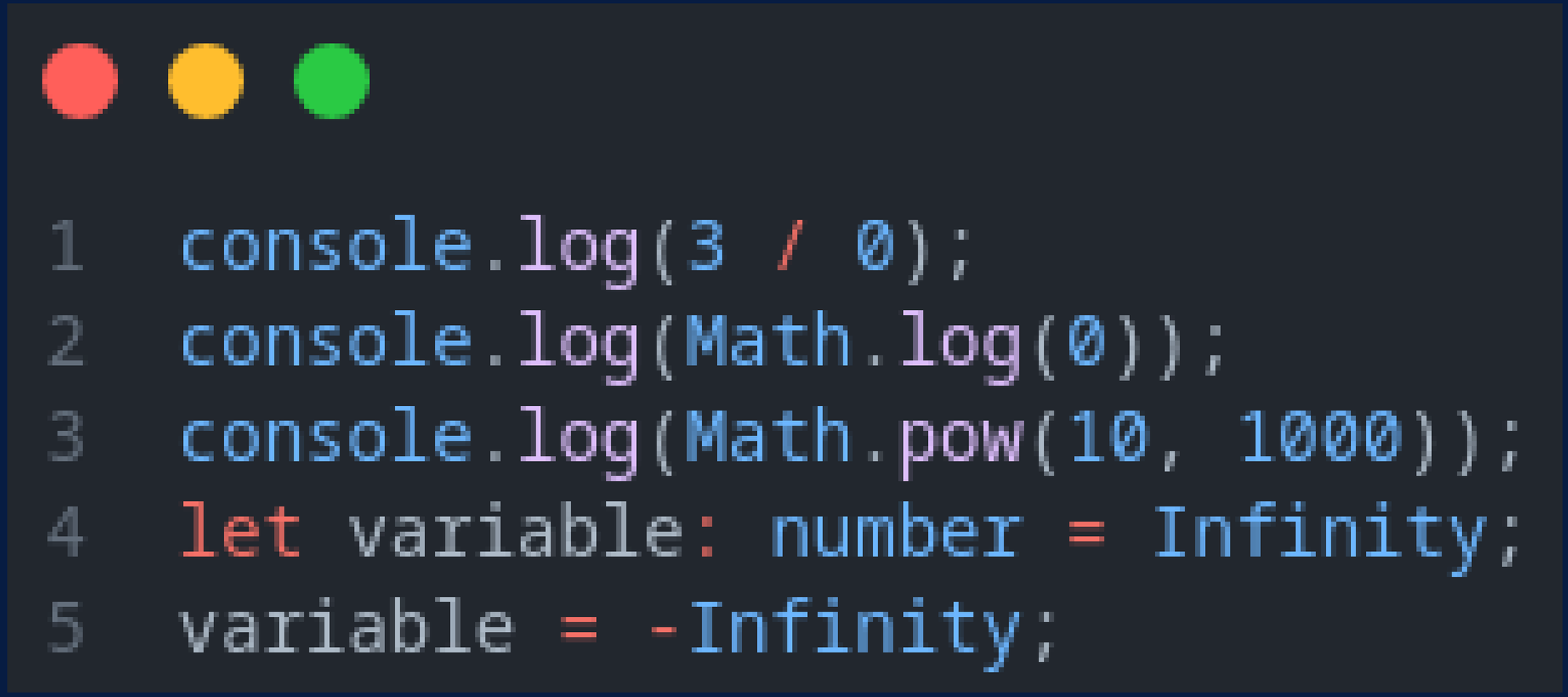
never

- Used when you want to specify that something isn´t fully implemented.



```
1  function notGoingToWork(): never {
2    while (true) {}
3  }
```

Infinity



```
1   console.log(3 / 0);
2   console.log(Math.log(0));
3   console.log(Math.pow(10, 1000));
4   let variable: number = Infinity;
5   variable = -Infinity;
```

Variable declaration

- Global variables are forbidden for us.
- Use constants if no modification needed.
- Use scope variables always

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

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

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

Variable declaration

- Arrays, always use type[]
- Strings, always use ‘ ’
- TS has garbage recollector

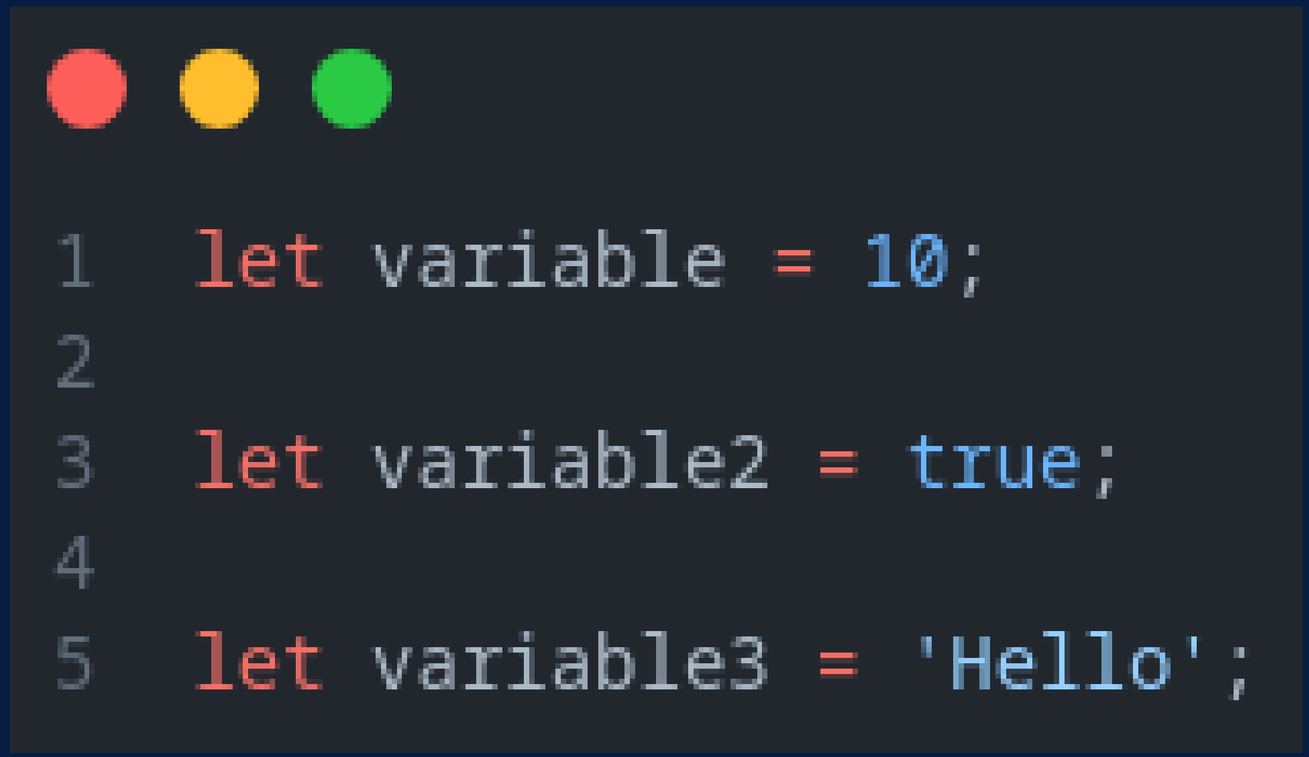
```
1  let randomVariable: number[] = [1, 2, 3];
```

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

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

Variable declaration

- Good practice: Leave out type annotations for trivially inferred types: variables or parameters initialized to a string, number, boolean, RegExp literal or new expression



Regular expressions

```
1  let regex1: RegExp = new RegExp('pattern');
2  let regex2: RegExp = /pattern/;
3  let regex3 = /pattern/;
4  let myString: string = 'HELLO pattern BYE';
5
6  console.log(regex1.test(myString));
```

User inputs

- Terminal, user's arguments start at argv[2].
- For execution time we import readline module.

```
1  const ARGUMENTS: string[] = process.argv;
```

```
1  import * as readLine from 'readline';
2  const readUserInput = readLine.createInterface({
3    input: process.stdin,
4    output: process.stdout
5  });
6
7  readUserInput.question('Write somenting: ', (entry: string) => {
8    console.log(`You wrote: ${entry}`);
9    readUserInput.close();
10 });
```

⚠️ npm i --save-dev @types/node

Conditionals

```
1  const CONDITION: boolean = false;
2  const CONDITION2: boolean = false;
3  if (CONDITION) {
4
5  } else if (CONDITION2) {
6
7  } else {
8
9  }
```

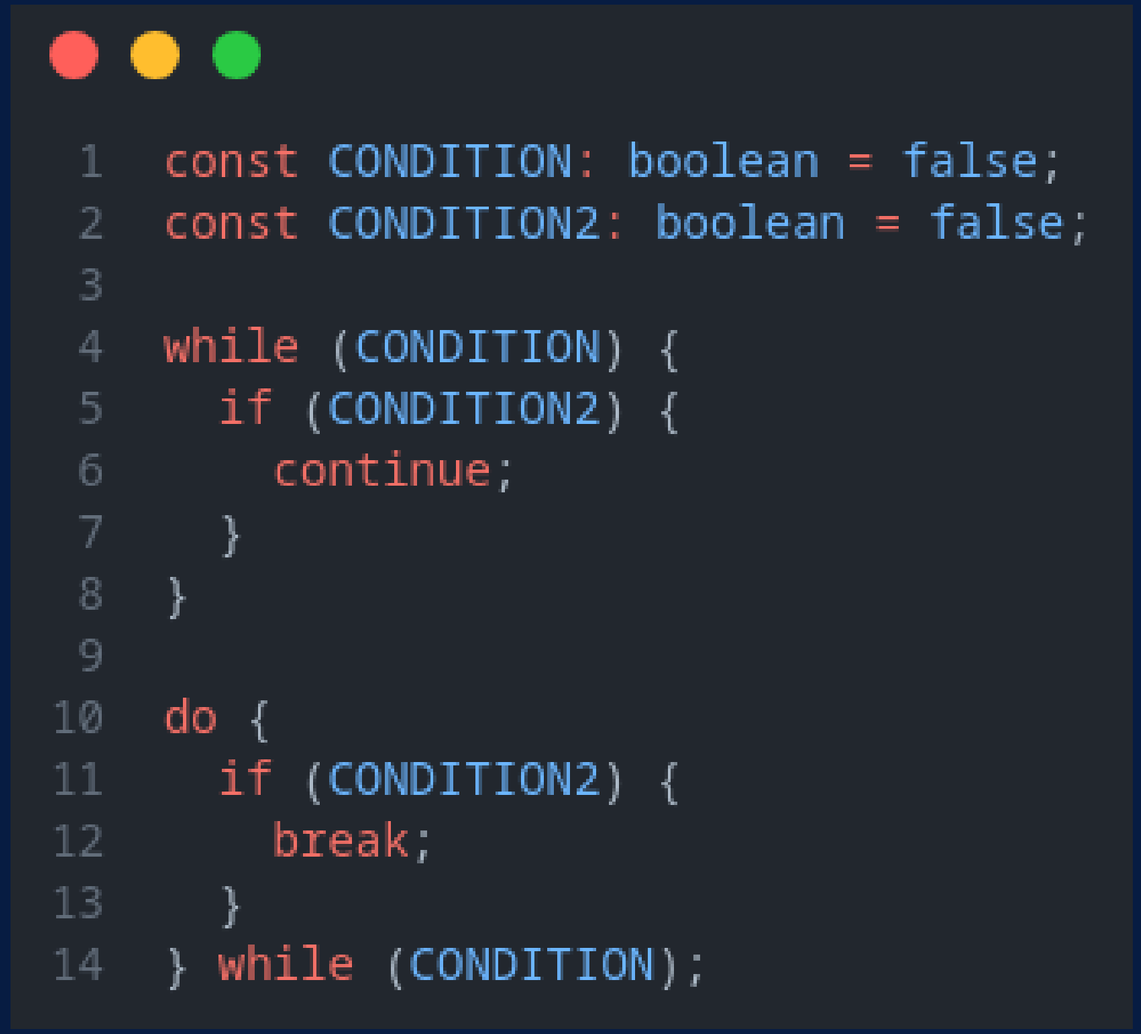
```
1  let variable: number = 10;
2  switch (variable) {
3    case 10:
4
5      break;
6    default:
7      break;
8  }
```

Loops

- Classic for.
- For of.
- For in.
- For each.

```
1  const VECTOR: number[] = [1, 2, 3];
2  const VECTORSIZE: number = VECTOR.length;
3
4  for (let index = 0; index < VECTORSIZE; ++index) {
5
6  }
7
8  for (const iterator of VECTOR) {
9
10 }
11
12 for (const index in VECTOR) {
13
14 }
15
16 VECTOR.forEach(element => {
17
18 });
```

Loops



```
1  const CONDITION: boolean = false;
2  const CONDITION2: boolean = false;
3
4  while (CONDITION) {
5    if (CONDITION2) {
6      continue;
7    }
8  }
9
10 do {
11   if (CONDITION2) {
12     break;
13   }
14 } while (CONDITION);
```

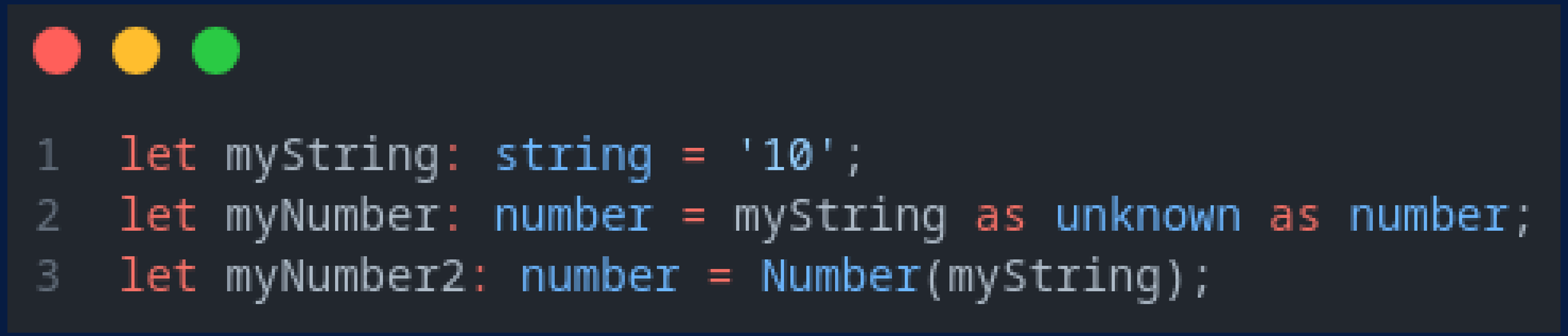
Operator “in”

- Elegant.
- Looks like pseudo code.
- Easier to undertand.
- Can be used for user´s defined classes.

```
1  let vector: number[] = [1, 2, 3];
2  let randomNumber: number = 2;
3  if (randomNumber in vector) {
4    console.log('USE MEEEE!!!');
5  }
```

Operator “as”

- **Used to convert types.**
- **Used in polymorphism.**



```
let myString: string = '10';
let myNumber: number = myString as unknown as number;
let myNumber2: number = Number(myString);
```

Operator “?.”

- Let´s you access to a property that might be null or undefined.
- Avoid if possible.

```
let person = {
  name: 'Jorge',
  dog: {
    name: 'Firulais'
  }
};

let dogNameSize = person.dog?.name?.length;
console.log(dogNameSize);
```

Operator “!.”

- Indicates to the compiler that you are sure that the expression won´t be null or undefined.
- Avoid if possible.

```
let person = {
  name: 'Jorge',
  dog: {
    name: 'Firulais'
  }
};

let dogNameSize = person.dog!.name!.length;
console.log(dogNameSize);
```

Functions

- Preferable function declarations over arrow functions.
- Arrow may be used, when an explicit type annotation is required.

```
1  function squareRoot(value: number): number {
2    return Math.sqrt(value);
3  }
4
5  interface AdditionFuction {
6    (value1: number, value2: number): number;
7  }
8
9  const addition: AdditionFuction = (value1, value2) => {
10   return value1 + value2
11 };
```

Optional parameters

- Optinal parameters and should be at the end.

```
function randomFunction(name: string, greetings?: string): void {
  if (greetings) {
    console.log(name + ', ' + greetings);
  } else {
    console.log(name);
  }
}

randomFunction('Juan');
randomFunction('Ana', 'Hello');
```

Function overloading

- You can´t have multiple functions with the same name neither with different parameters and returns.

```ts
/*
        ERROR!!!!!!

  function overloading(value: number): void {
    console.log(value);
  }

  function overloading(value: string): number {
    return Number(value);
  }
*/

function overloading(value: number | string): void | number {
  if (typeof value === 'number') {
    console.log(value);
  } else if (typeof value === 'string') {
    return Number(value);
  }
}

overloading(42);
overloading('42');
```

Rest parameters

- the numbers of parameters that a function will receive is not known or can vary.

```
function concatenateStrings(separator: string,
                            ...strings: string[]): string {
  return strings.join(separator);
}

// Example usage
let result = concatenateStrings(', ', 'apple', 'orange', 'banana');
console.log(result);  // Output: "apple, orange, banana"
```

Read/Write from a file

- Read line by line

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

const FILEPATH: string = 'data.txt';

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.');
});
```

Read/Write from a file

Write

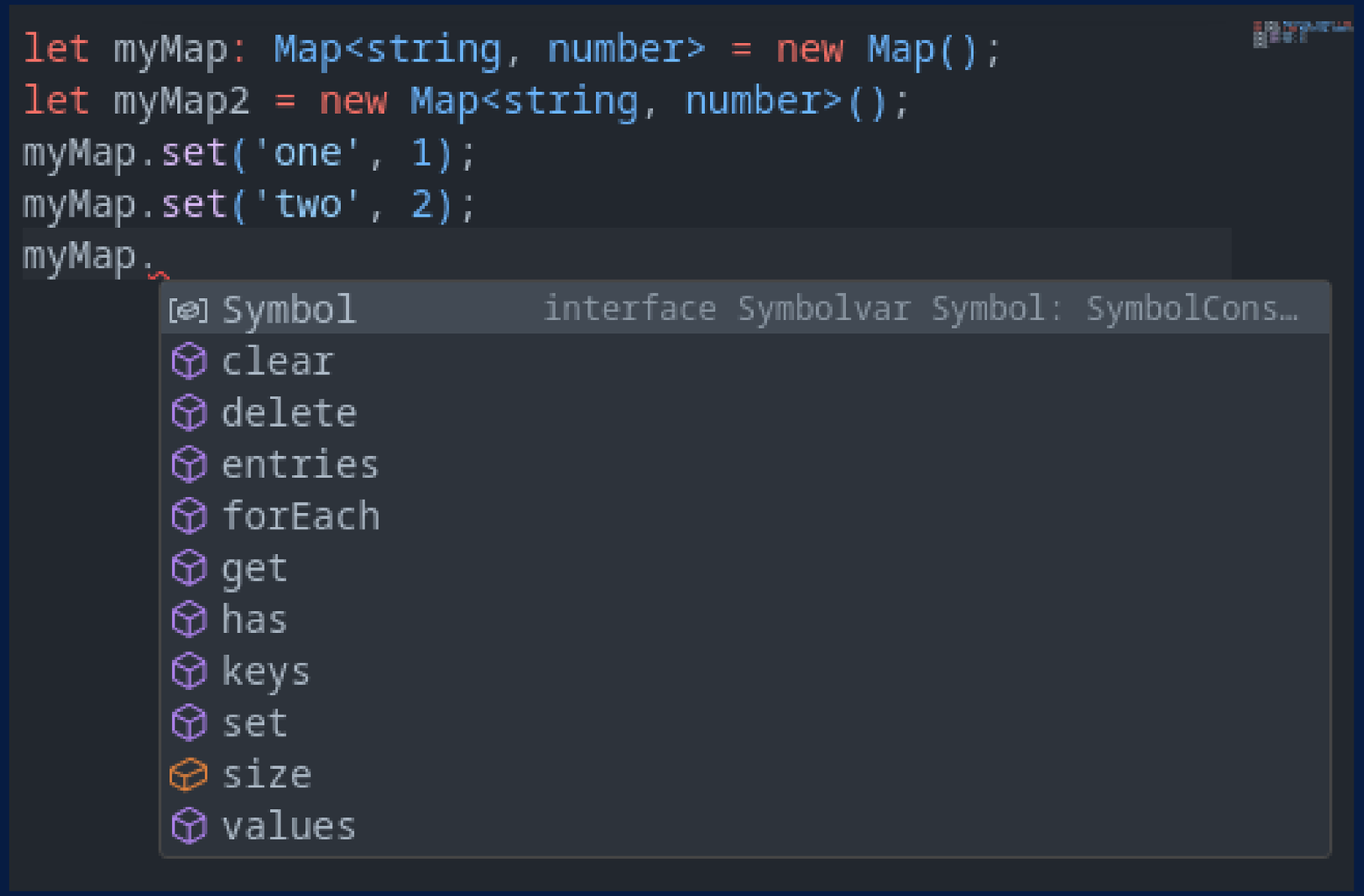
```
import * as fs from 'fs';

const FILEPATH: string = 'data.txt';
const LINE: string = 'NEW LINE';

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

Error Handling

```ts
function divide(value1: number, value2: number): number {
  const ZERO: number = 0;
  try {
    if (value2 === ZERO) throw new Error('Cannot divide by 0');
    return value1 / value2;
  } catch (error) {
    console.error('Error:', error);
    return Infinity;
  }
}

console.log(divide(10, 0));
```

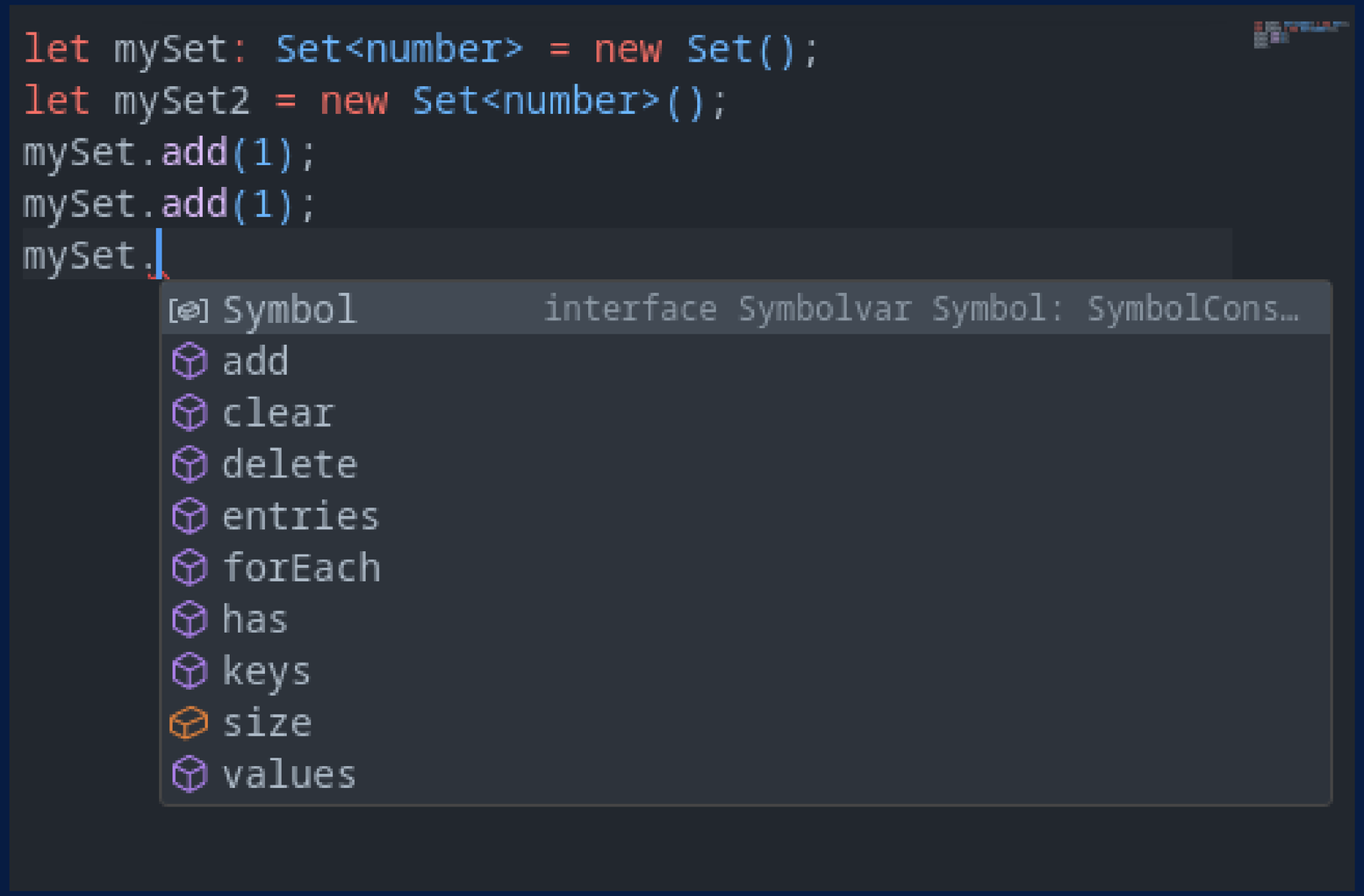
Built-in Data Structures

Map



Built-in Data Structures

Set

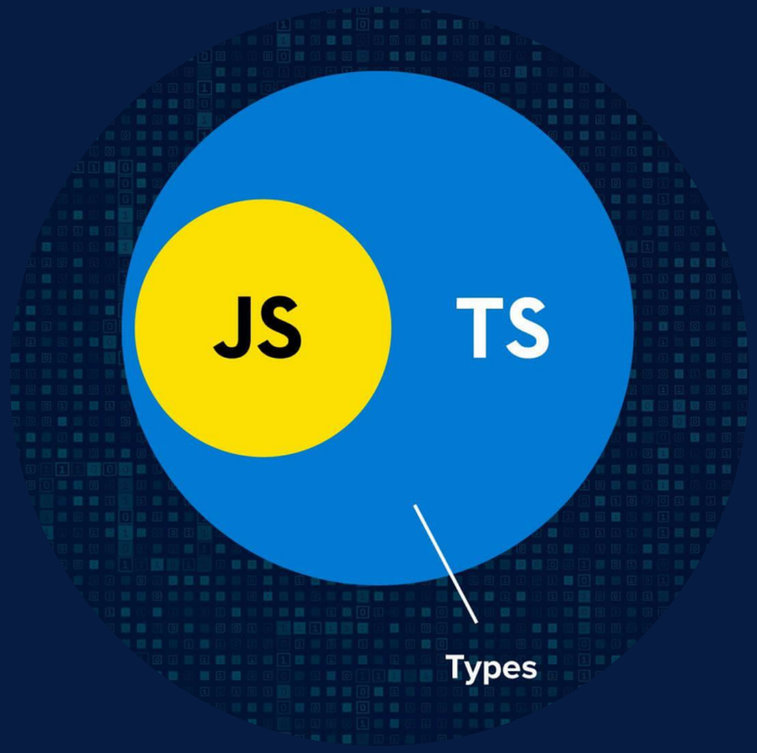


Built-in Data Structures

- Stack and Qeueu can be implemented with arrays.
- Hash tables are implemented with maps.

Advanced concepts

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



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    // Constructor
7    constructor(brand: string, model: string, year: number) {
8      this.brand = brand;
9      this.model = model;
10     this.year = year;
11   }
12   // Method to start the car
13   startEngine(): void {
14     console.log(`Starting the engine of the ${this.brand} ${this.model}...`);
15   }
16 }
```

There is no ; at the end of the class definition

Parameter Properties

TypeScript offers special syntax for turning a constructor parameter into a class property with the same name and value.

```
1  class Car {
2    // Constructor
3    constructor(private brand: string, private model: string, private year: number) {}
4    // Method to start the car
5    startEngine(): void {
6      console.log(`Starting the engine of the ${this.brand} ${this.model}...`);
7    }
8  }
```

Read only properties

It cannot be changed!

```
class Car {
  readonly carPlate: string = "6753ABC";
  ...
}
```

```
myCar.carPlate = "2341C";    Cannot assign to 'carPlate' because it is a read-only
```

Read only

- When we assign a value we cannot modify it
- Is used for properties within the members of a class.

Const

- We need to assign a value when we declare it.
- Is used for variables.

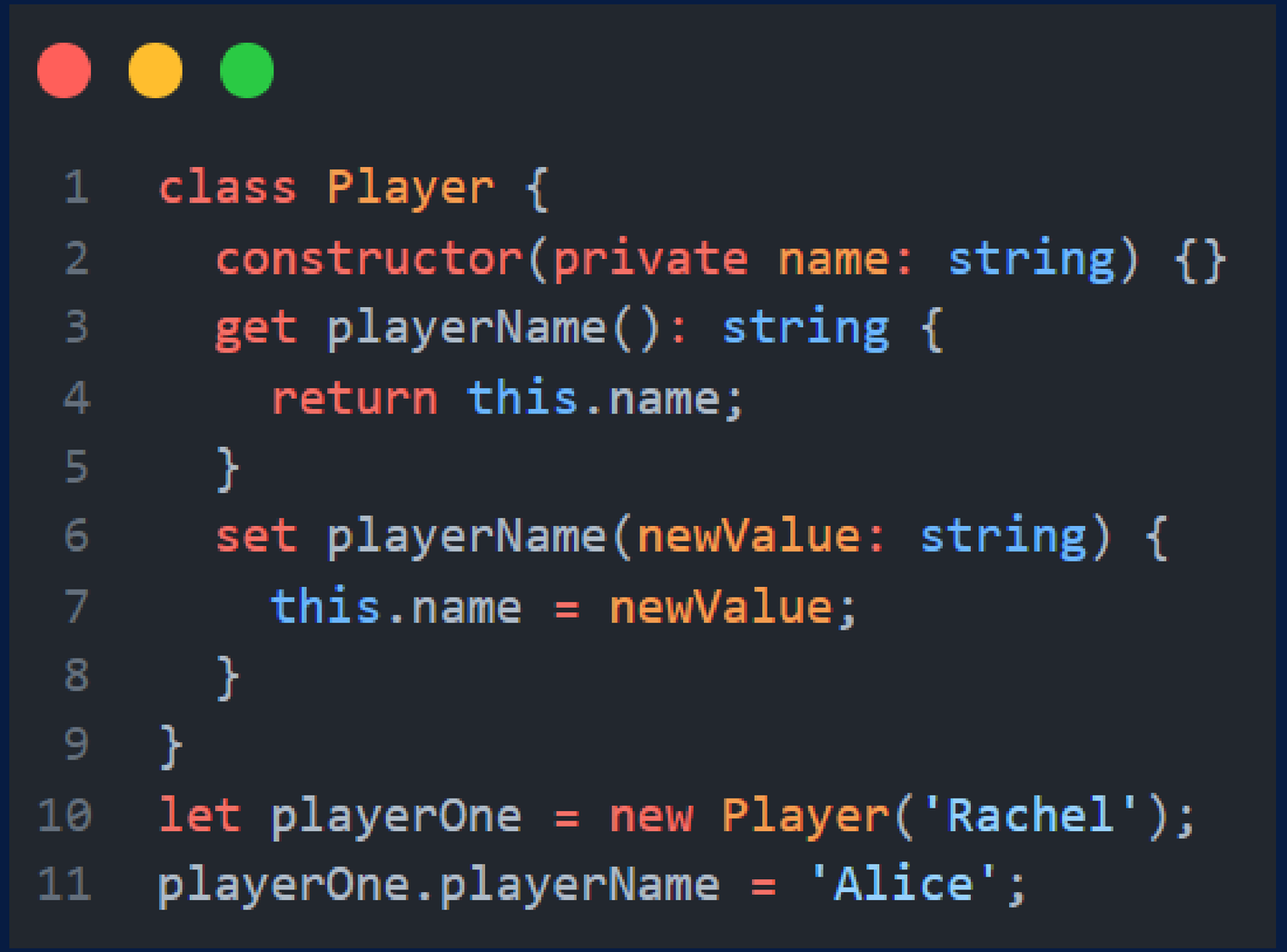
Overloading constructors

You can only implement one constructor

```
1  class Vehicle {
2    private type: string;
3    private wheels: number;
4    constructor(type: string);
5    constructor(type: string, wheels: number);
6    constructor(type: string, wheels?: number) {
7      this.type = type;
8      this.wheels = wheels ?? 4; // Default to 4 wheels if not specified
9    }
10 }
```

It only serves for clarity and to explicitly state the expectations of how an object of that class can be constructed.

Getters and setters



```
class Player {
  constructor(private name: string) {}
  get playerName(): string {
    return this.name;
  }
  set playerName(newValue: string) {
    this.name = newValue;
  }
}
let playerOne = new Player('Rachel');
playerOne.playerName = 'Alice';
```

Getters and setters

“At least one accessor for a property must be non-trivial: ***do not define pass-through accessors only for the purpose of hiding a property****. Instead,* ***make the property public*** *(or consider* ***making it readonly*** *rather than just defining a getter…”*

Static members

```
1  class Calculator {
2    static add(a: number, b: number): number {
3      return a + b;
4    }
5  }
6  const SUM = Calculator.add(5, 10);
```

Belong to the class itself, and not to instances of that class.

Static members

Also works with attributes

```
1  class Animal {
2    public static numberOfAnimals: number = 0;
3    constructor() {
4      Animal.numberOfAnimals += 1;
5    }
6  }
```

It is useful when you need to share common value for all instances

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.`);
  }
}
```

Abstract classes

```
abstract class Shape {
  constructor(public name: string) {}
  abstract calculateArea(): number;  // Abstract methods
  abstract calculatePerimeter(): number;
  display(): void {
      console.log(`Shape: ${this.name}`);
  }
}
class Circle extends Shape {
  constructor(public radius: number) {
    super("Circle");
  }
  calculateArea(): number { // Implementatios of the abstract methods
    return Math.PI * this.radius * this.radius;
  }
  calculatePerimeter(): number {
    return 2 * Math.PI * this.radius;
  }
}
```

Interfaces

It is a contract.

Its obligations and annexes must be complied with.

```ts
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

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

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

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

Interfaces

An important aspect

All is public!

Abstract class

- Mix of implemented methods and abstract methods.
- A class can only extend an abstract class.
- They can have a constructor.
- They have all types of visibility

Interface

- Totally abstract.
- A class can implement multiple interfaces.
- They only serve as a type contract.
- All is public.

Generic Types

Reuse code

```ts
class Pair<T, U> {
  private first: T;
  private second: U;
  constructor(first: T, second: U) {
    this.first = first;
    this.second = second;
  }
}
```

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

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

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

Adding constraints to the types

```
interface Account {
  readonly name: string,
  password: string
}
function deleteSocialMedia<T extends Account>(socialMedia: T): Boolean {
  //implementation
  return true;
}
class FacebookAccount implements Account {
  constructor(public readonly name: string, public password: string) {}
}
let facebookAccount = new FacebookAccount("Saul", "1234");
deleteSocialMedia(facebookAccount); // Ok
```

Adding constraints to the types

```
class Person {
  constructor(public readonly name: string) {}
}

let person = new Person("Saul");
deleteSocialMedia(person); // Compilation error
```

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

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  }
```

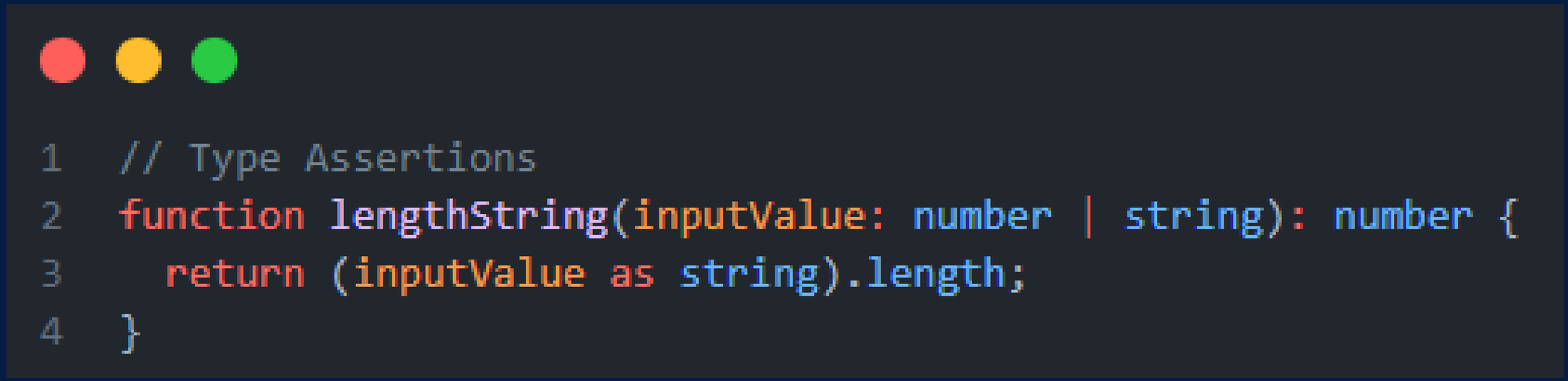
Narrowing

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

```ts
// 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

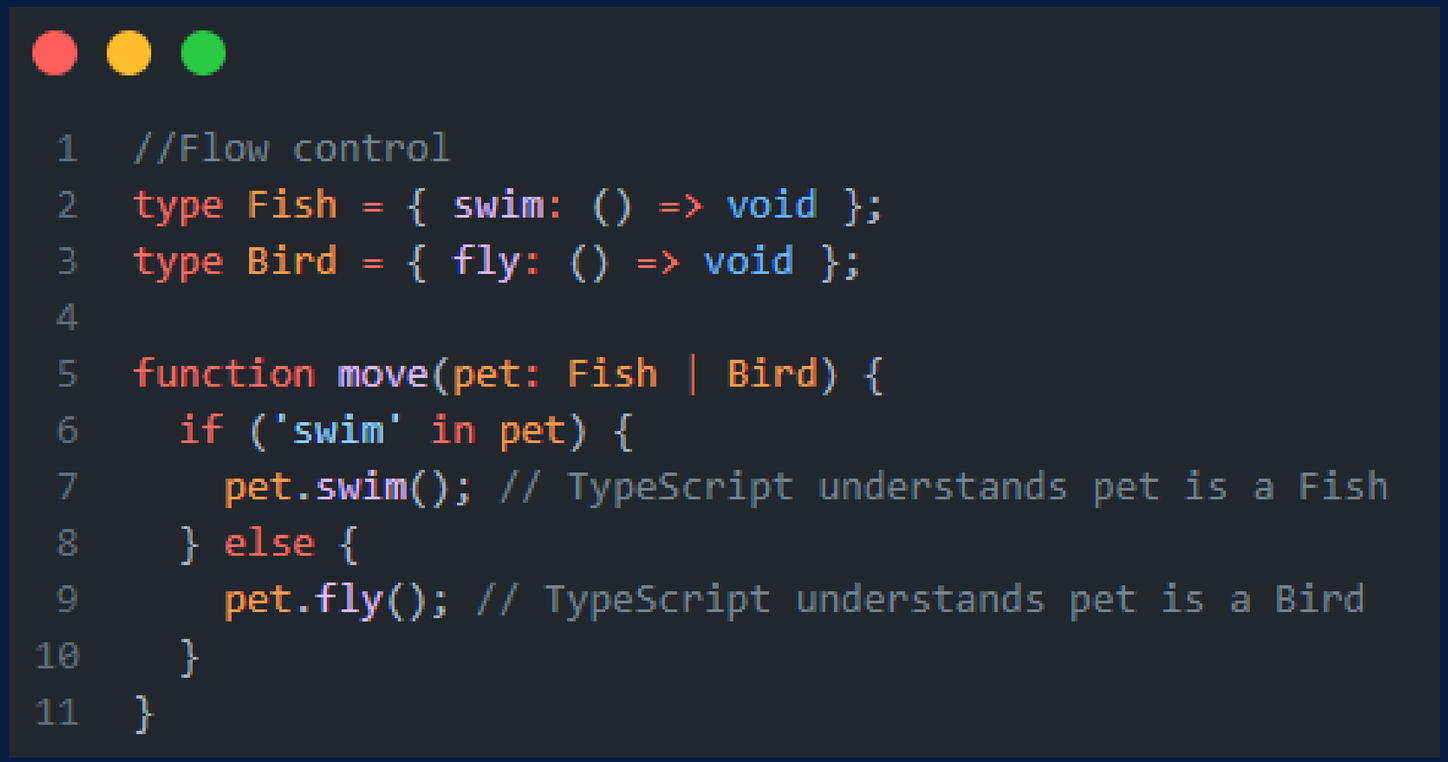
- **Type Assertions:** Explicitly assert that a variable is of a specific type.



```
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.



```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
  }
}
```

Modules

Are a way to organize and encapsulate code

- Split the code into separate files.
- They can be used to create namespaces.

Types of modules

CommonJS

- Used in **Node.js** applications
- Use **require** to import.
- Use **module.exports** or **exports** to export .
- Module loading is synchronous.

ES Modules

- **Standard** for modules in modern frontend application.
- Use **import** to import.
- Use **exports** to export .
- Module loading is dynamic.

Export and Import

- Exports and import what you need.
- Do not use “require”.
- Be careful with circular dependencies.

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

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

```
// 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();
```

NameSpaces

```
1  namespace Football {
2    export function result(): void {
3      console.log('BARCELONA 2 - 8 BAYERN');
4    }
5  }
6
7  Football.result();
```

NameSpaces

*“TypeScript supports two methods to organize code: namespaces and modules, but **namespaces** are **disallowed**.”*

More reserved words

delete

```
let vector: number[] = [1, 2, 3, 4];
delete vector[0];
console.log(vector); // [ <1 empty item>, 2, 3, 4 ]
console.log(vector[0]); // undefined
```

More reserved words

finally always gets executed.

```
function divide(value1: number, value2: number): number {
  const ZERO = 0;
  try {
    if (value2 === ZERO) {
      throw new Error('CANNOT DEVIDE BY 0');
    }
    return value1 / value2;
  } catch (error) {
    console.error("Error:", error);
    return Infinity;
  } finally {
    console.log('FINALLY WILL ALWAYS EXECUTE');
  }
}

console.log(divide(10, 2));
console.log(divide(5, 0));
```

More reserved words

yield

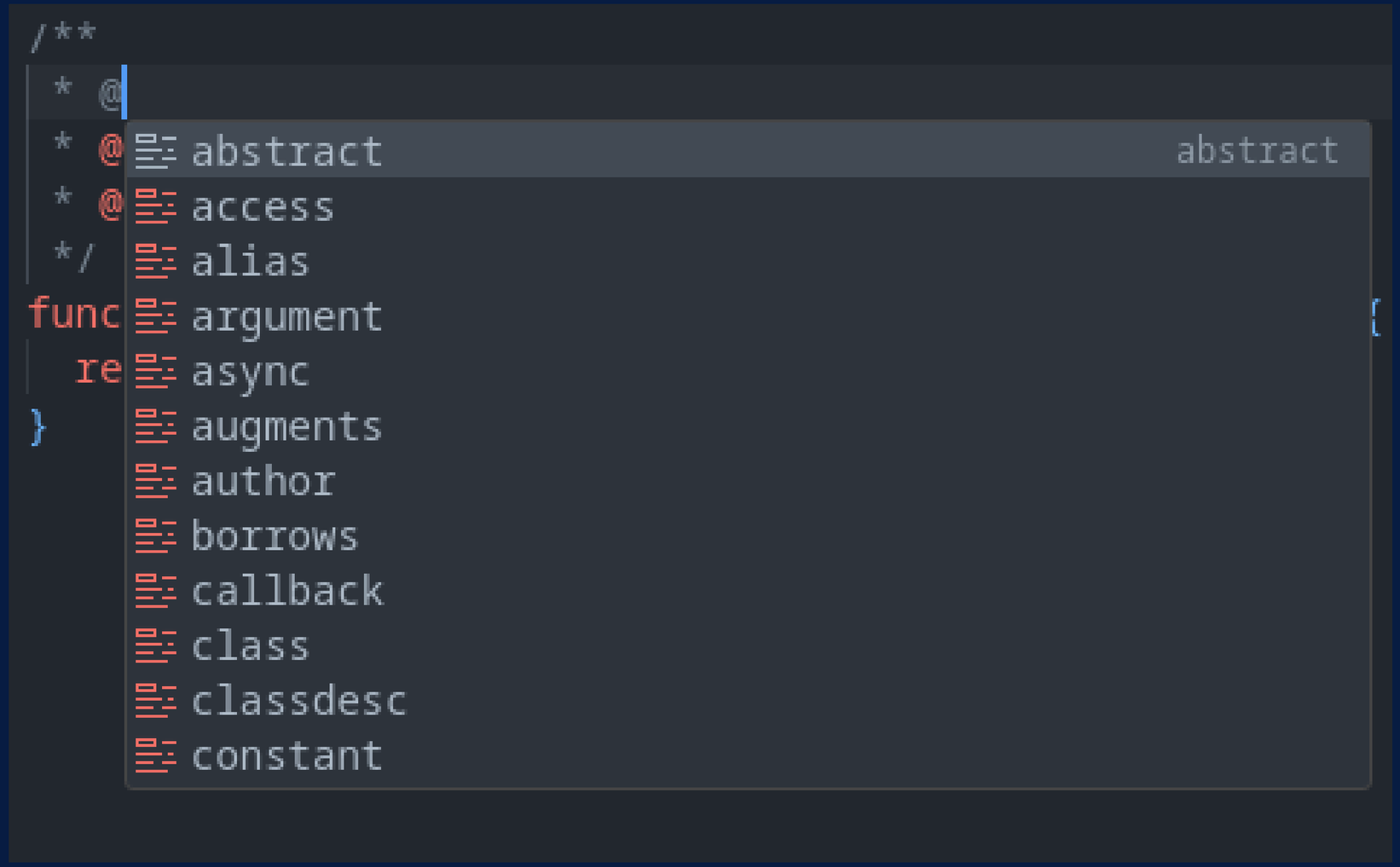
```
1  function* numberGenerator(index: number) {
2    while (index < 5) {
3      yield index;
4      ++index;
5    }
6  }
7
8  let iterator = numberGenerator(0);
9  console.log(iterator.next().value);
10 console.log(iterator.next().value);
```

More reserved words

“with” is just part of JS and not TS.

```
1   let vector: number[] = [1, 2, 3];
2
3   with (vector) {
4     toString();
5   }
```

JSDoc comments



JSDoc comments

- Do not add redundant information!
- Document every property and method!
- Avoid types of parameters and return, the code already tells you!

File Structure

WE WILL ALWAYS USE THIS FORMAT:

1. **Copyright information, if present.**
2. **JSDoc with @fileoverview, if present.**
3. **Imports, if present.**
4. **The file’s implementation.**

Recommendations

Don´t forget

PRACTISE MAKES PERFECT!

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

Bibliography

- 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
- Scaler Topics. (n.d.). TypeScript ReadOnly. Retrieved January 28, 2024, from https://www.scaler.com/topics/typescript/typescript-readonly/
- TypeScript. (n.d.). TSConfig Reference. Retrieved January 28, 2024, from https://www.typescriptlang.org/tsconfig
- Manz (n.d.). CommonJS vs ES Modules. Lenguaje JS. Retrieved January 28, 2024, from https://lenguajejs.com/automatizadores/introduccion/commonjs-vs-es-modules/