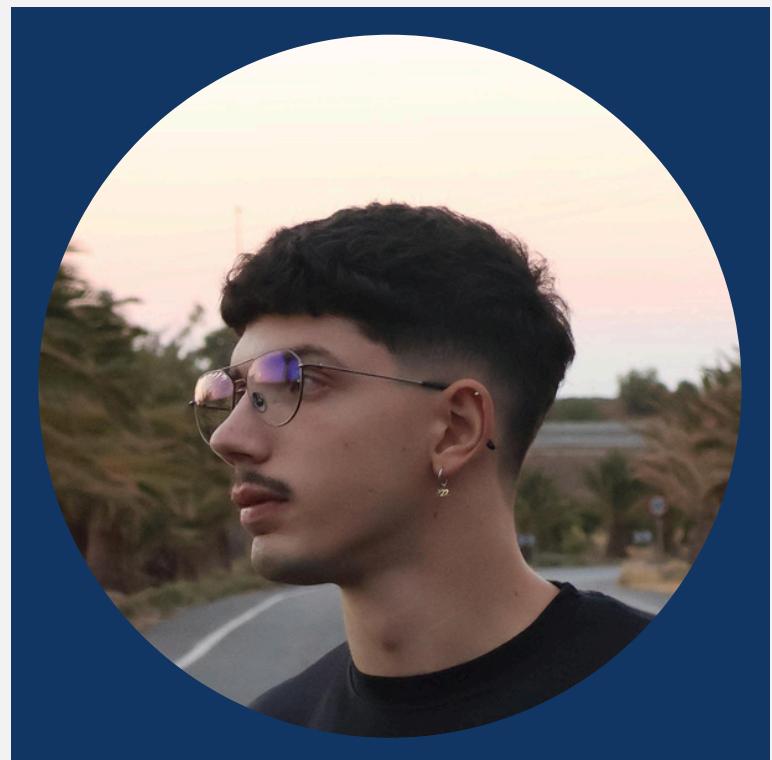


# INTRODUCTION TO TYPESCRIPT



# OUR TEAM



**Cristóbal Jesús Sarmiento Rodríguez**  
[cristobal.jesus.09@ull.edu.es](mailto:cristobal.jesus.09@ull.edu.es)



**Marcos Llinares Montes**  
[marcos.llinares.14@ull.edu.es](mailto:marcos.llinares.14@ull.edu.es)



**Paulo Padilla Domingues**  
[padilla.domingues.37@ull.edu.es](mailto:padilla.domingues.37@ull.edu.es)

# INDEX

01 Introduction

02 Installation

03 Basic  
Concepts

04 Remarkables  
Features

05 OOP

06 Interfaces

07 Modules

08 Conclusion

# INTRODUCTION

# WHAT IS TYPESCRIPT?

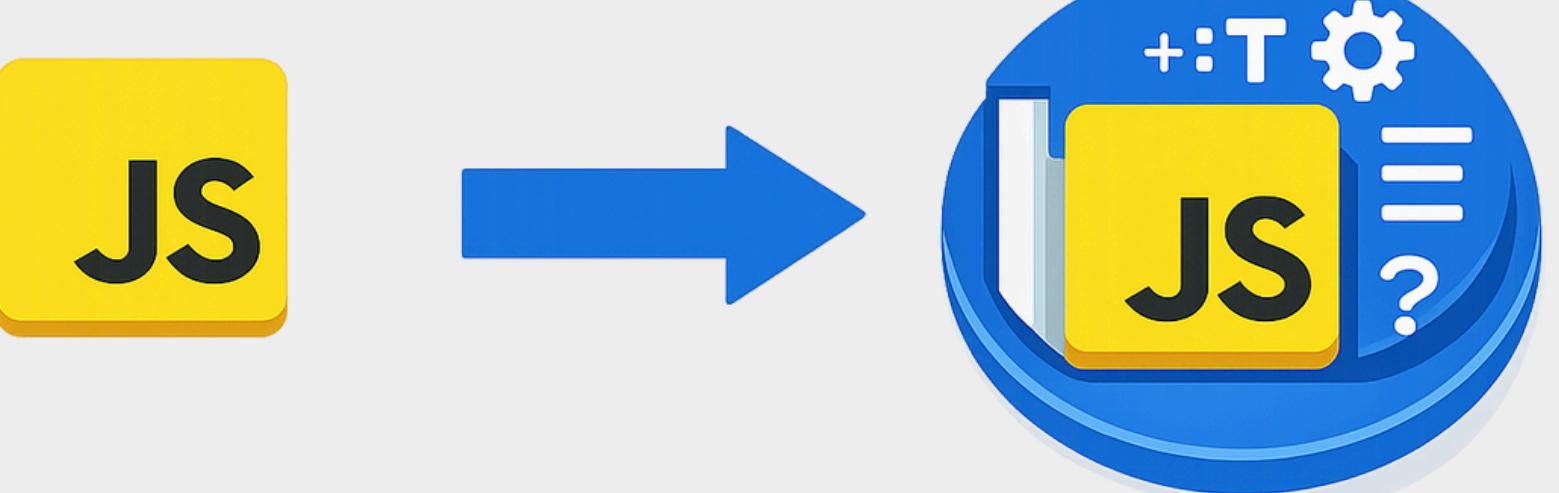
- Superset of JavaScript.
- Open-source.
- High-level programming language.
- Developed by Microsoft.
- Strongly typed.



**Goal:** Layer of static type checking  
allows to catch errors during development (**not in runtime** 💀).

# WHAT IS TYPESCRIPT?

**Compatibility:** Any valid JavaScript code is also valid TypeScript code.

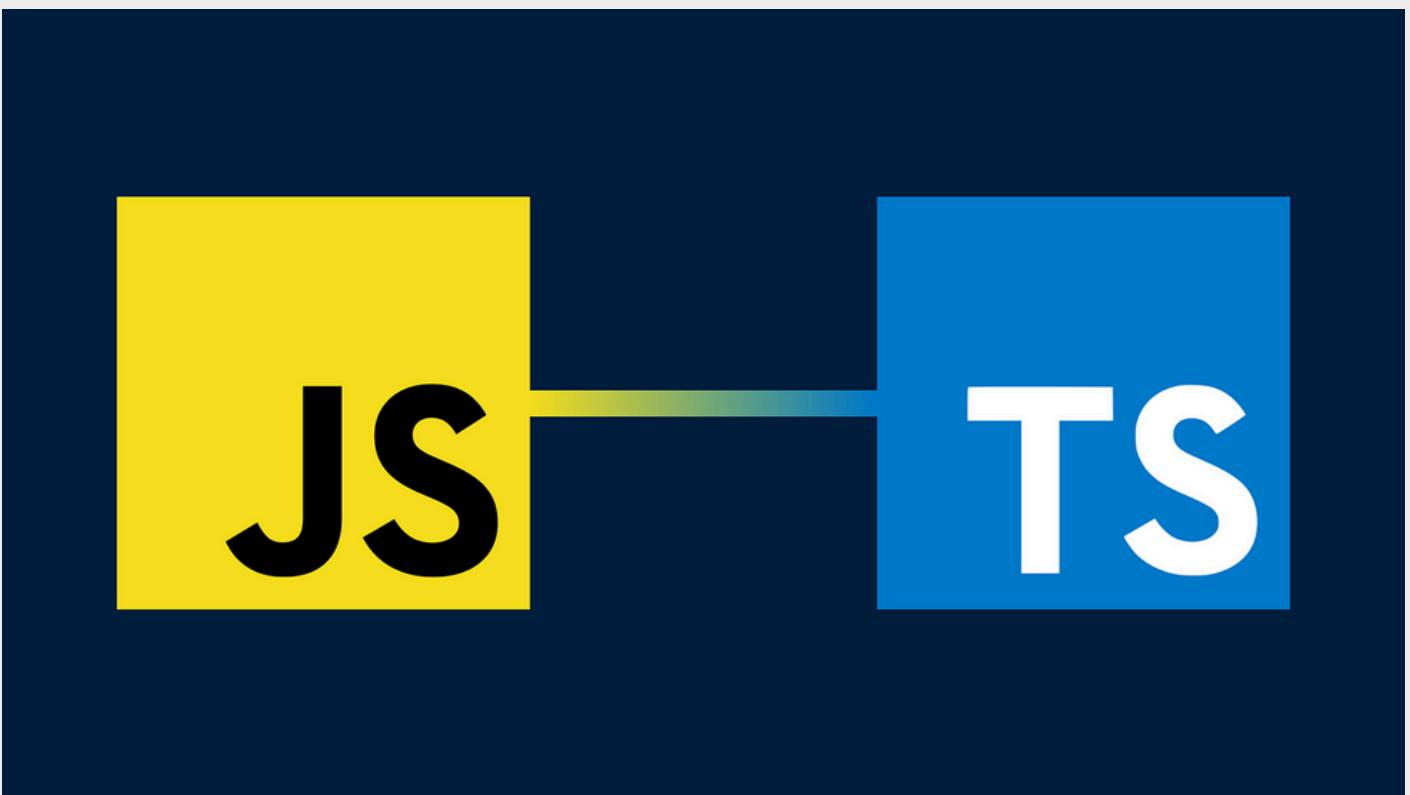


**The "Transpiler" Model:** Browsers and Node.js cannot run TS files. Code must be compiled (transpiled) into JavaScript.

# MAIN DIFFERENCES

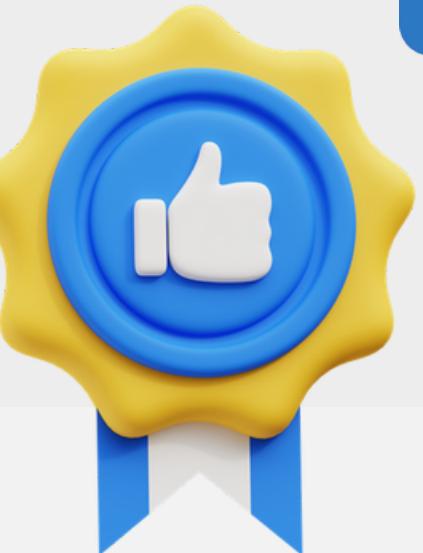
The main differences between this two languages are:

1. Typing.
2. Error Detection.
3. Object-oriented programming.
4. Improved development tools (Features).
5. Compilation Step.



# BENEFITS OF TYPESCRIPT

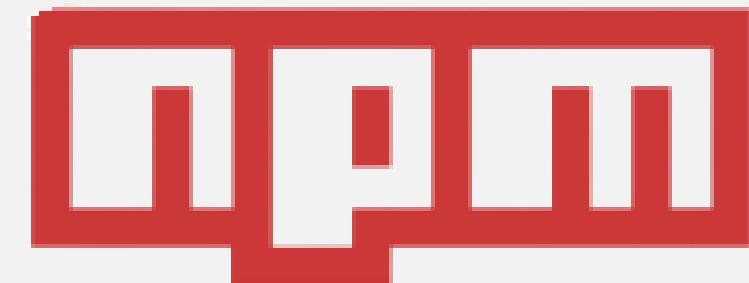
1. Early Error Detection.
2. Self-Documenting Code.
3. Better Scalability & Maintainability.
4. Greater Productivity & Security.
5. Modern Features (latest ECMAScript).
6. Enhanced IDE Support.



# INSTALLATION

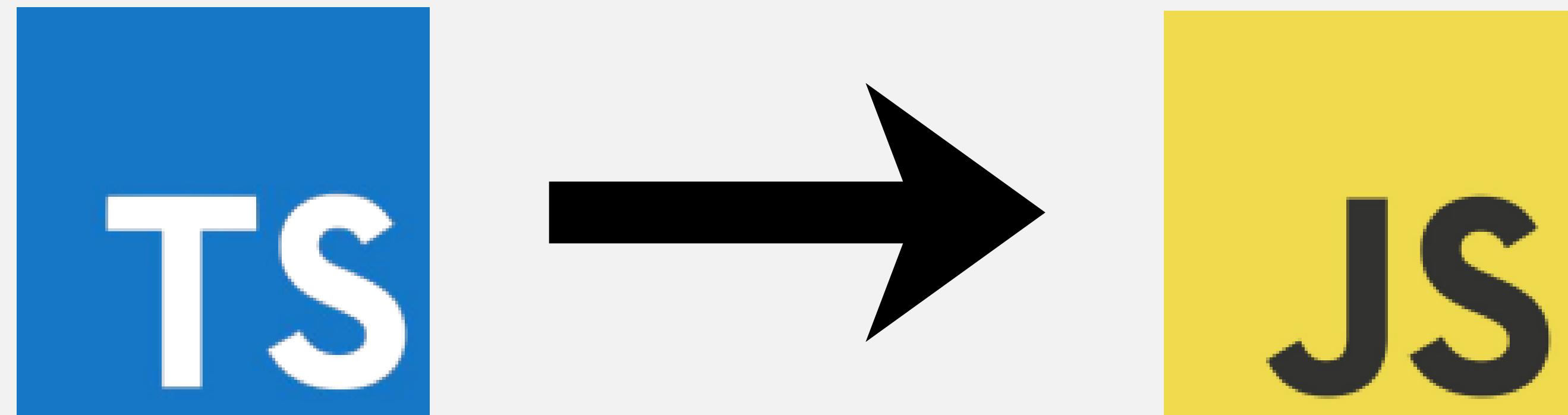
# REQUISITES

1. Necessary to install **Node.js** and a package manager (**npm** or **pnpm**).
2. **Fundamental:** TypeScript compiler.
  - **tsc.**
  - **ts-node.**



# COMPIILATION STRATEGIES

 tsc



Javascript transpiled code is not designed to be read by humans.

# COMPILATION STRATEGIES

 tsc     Installation

```
npm install - g typescript
```

# COMPIILATION STRATEGIES

→ ts-node



# COMPIILATION STRATEGIES

ts-node → Installation

```
npm install -g ts-node
```

# TSCONFIG FILE

- Configuration file for TypeScript.
- Defines how to compile the code.
- We can customize this file with many options.

→ [Example tsconfig.json](#)

→ [tsconfig.json documentation](#)

```
{  
  "display": "Configuration for Exercism TypeScript Exercises",  
  "compilerOptions": {  
    "lib": ["ES2020", "dom"],  
    "module": "Node16",  
    "target": "ES2020",  
    "strict": true,  
    "esModuleInterop": true,  
    "skipLibCheck": true,  
    "forceConsistentCasingInFileNames": true,  
    "allowImportingTsExtensions": true,  
    "noEmit": true,  
    "isolatedModules": true  
},  
  "include": [  
    "*.ts",  
    "*.tsx",  
    ".meta/*.ts",  

```

# BASIC CONCEPTS

# BASIC TYPES

- Number
- String
- Boolean
- Bigint
- Symbol



[Basic Types](#)

# SPECIAL TYPES

- Any
- Unknown
- Void
- Null
- Undefined
- Never

 [Special Types](#)

# COMPOSITE TYPES

- Array
- Tuple
- Object
- Enum

 [Composite Types](#)

# INFERENCE

Compiler automatically assigns the type based on the initial value.

**C++ Similarity:** Equivalent of using the **auto** keyword.

```
● ● ●  
// Explicit Declaration  
let myName: string = 'Cristobal';  
  
// Type Inference  
let myName = 'Cristobal'; // TS knows it is a string  
let myAge = 22;           // TS knows it is a number  
let isStudent = true;     // TS knows it is a boolean
```

¿Funciona con strict: true?

```
let mysteryVariable; // ERROR in strict mode: is a implicit 'any'
```



```
let clearVariable = "Hello"; // OK: TS infers it is a string, not 'any'
```



# CAN WE USE VAR?

var does not respect block scope



```
function blockScopeInCorrect(): void {  
  if (true) {  
    var age = 18;  
  }  
  // Age is still accessible here  
  console.log(age); // 10  
}
```



```
function blockScopeCorrect(): void {  
  if (true) {  
    let age: number = 18;  
  }  
  console.log(age); // Error: Age is not defined  
}
```

# CAN WE USE **VAR**?

var allows variable redeclaration



```
function redeclarationExample(): void {  
    var count = 1;  
    var count = 2; // Allowed  
    console.log(count); // 2  
}
```



```
function redeclarationBlocked(): void {  
    let count: number = 1;  
    let count: number = 2; // Error  
}
```

# CAN WE USE VAR?

var causes unexpected behavior in loops



```
function loopVarExample(): void {
  for (var i = 0; i < 3; ++i) {
    setTimeout(() => {
      console.log(i);
    }, 100);
  }
}
// Output: 3, 3, 3
```



```
function loopLetExample(): void {
  for (let i = 0; i < 3; ++i) {
    setTimeout(() => {
      console.log(i);
    }, 100);
  }
}
// Output: 0, 1, 2
```

# LOOPS: FOR/FOR OF



```
function forExample(): void {  
    for (let i = 0; i < 3; ++i) {  
        console.log(i);  
    }  
}  
forExample();
```



```
function forOfExample(): void {  
    let numbers: number[] = [1, 2, 3];  
  
    for (let value of numbers) {  
        console.log(value);  
    }  
}  
forOfExample();
```

# LOOPS: FOR IN/FOR EACH



```
function forInExample(): void {  
    let user = {name: 'Ana', age: 25};  
  
    for (let key in user) {  
        console.log(key, user[key as keyof typeof user]);  
    }  
}  
forInExample();
```



```
function forEachExample(): void {  
    let numbers: number[] = [1, 2, 3];  
  
    numbers.forEach((value) => {  
        console.log(value);  
    });  
}  
forEachExample();
```

# UNION TYPES

- Union Types allow a variable to have more than one possible type.
- This is safer than using any.



[Union Types](#)

# WHY UNION TYPES INSTEAD OF ANY

- With any, TypeScript stops checking types completely.
- With Union Types, we still have control over which types are allowed.



**Not Any.**

# TYPE ALIASES

- Create our own custom types in TypeScript.
- They help reduce repetition and make the code more readable.



Aliases

# LITERAL TYPES

- We use exact values.
- Restrict a variable to a fixed set of possible values.



Literal Types



# **REMARKABLES FEATURES**





# TYPE ASSERTION

Type assertions tell the compiler how to treat a value when the developer knows its type.



```
function typeAssertionExample(): void {  
    let value: unknown = 'Hello TypeScript';  
  
    let length: number = (value as string).length;  
    console.log(length);  
}  
  
typeAssertionExample();
```



# OPTIONAL CHAINING

```
● ● ●  
function optionalChainingObjectExample(): void {  
  let user:  
    | {  
        name: string;  
        address: {  
            city: string;  
        };  
    }  
    | undefined;  
  
  let city = user?.address.city;  
  
  console.log(city); // undefined, no error  
}  
optionalChainingObjectExample();
```

Optional chaining allows safe access to object properties that may be null or undefined.

# NULLISH COALESING OPERATOR



```
function nullishCoalescingExample(): void {  
  let input: string | null = null;  
  
  let username = input ?? 'Guest';  
  console.log(username);  
}  
nullishCoalescingExample();
```

The nullish coalescing operator provides a default value only when the result is null or undefined.



# SPREAD OPERATOR

```
function spreadOperatorExample(): void {
  let numbers1: number[] = [1, 2];
  let numbers2: number[] = [3, 4];

  let combined = [...numbers1, ...numbers2];
  console.log(combined);

  let user = { name: 'Ana', age: 25 };
  let copy = { ...user };

  console.log(copy);
}

spreadOperatorExample();
```

The spread operator allows copying and merging objects or arrays in a clean and readable way.

# STYLE GUIDE

- Google provides its own TypeScript style guide.
- Shared JavaScript foundations
- TypeScript introduces some specific rules to avoid bad practices.



# GOOGLE GTS



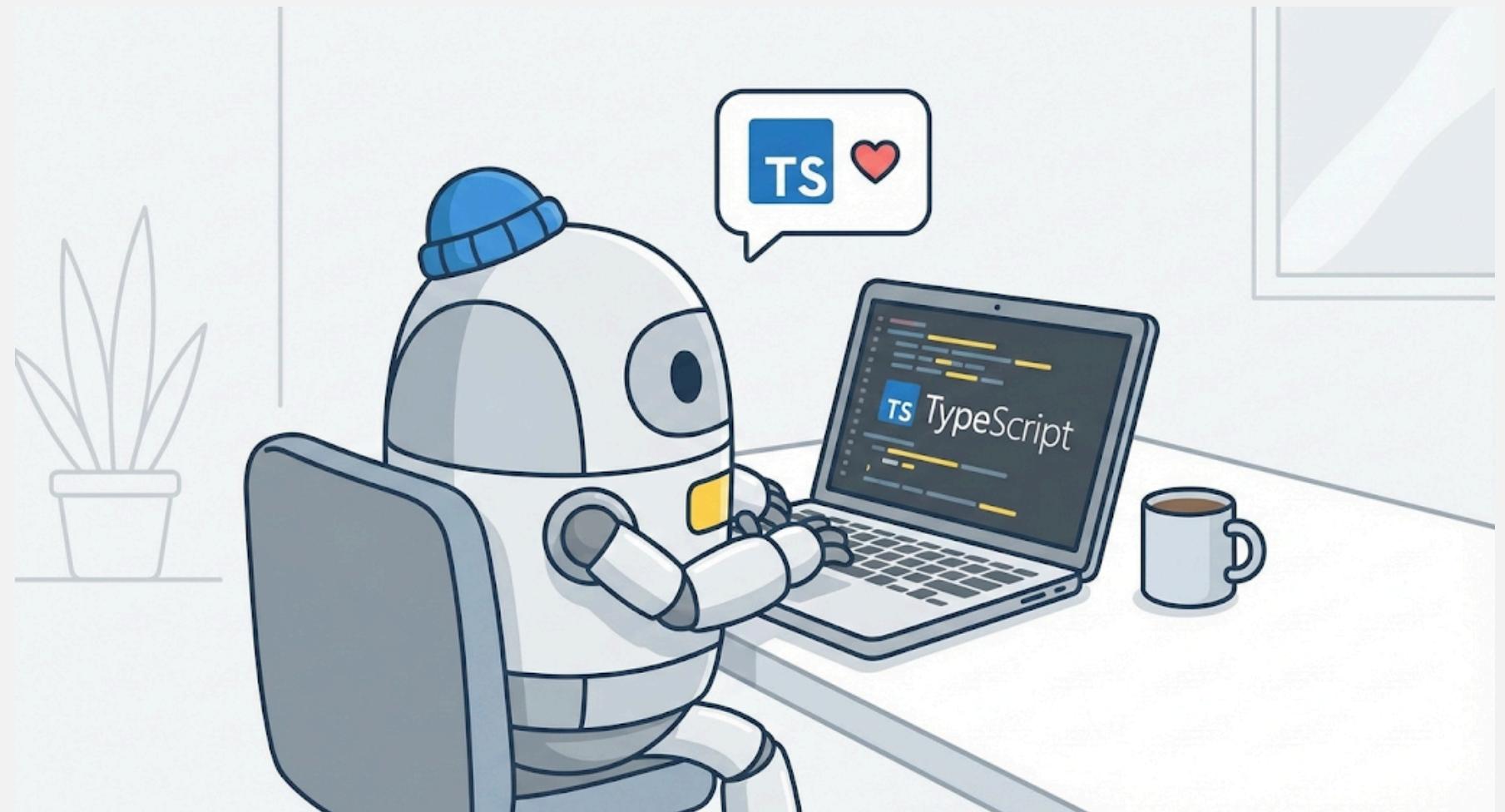
[Google GTS](#)

# OOP

# C++ VS TYPESCRIPT

→ TS

- constructor keyword.
- readonly modifier.
- Single constructor only.



# C++ VS TYPESCRIPT

→ TS

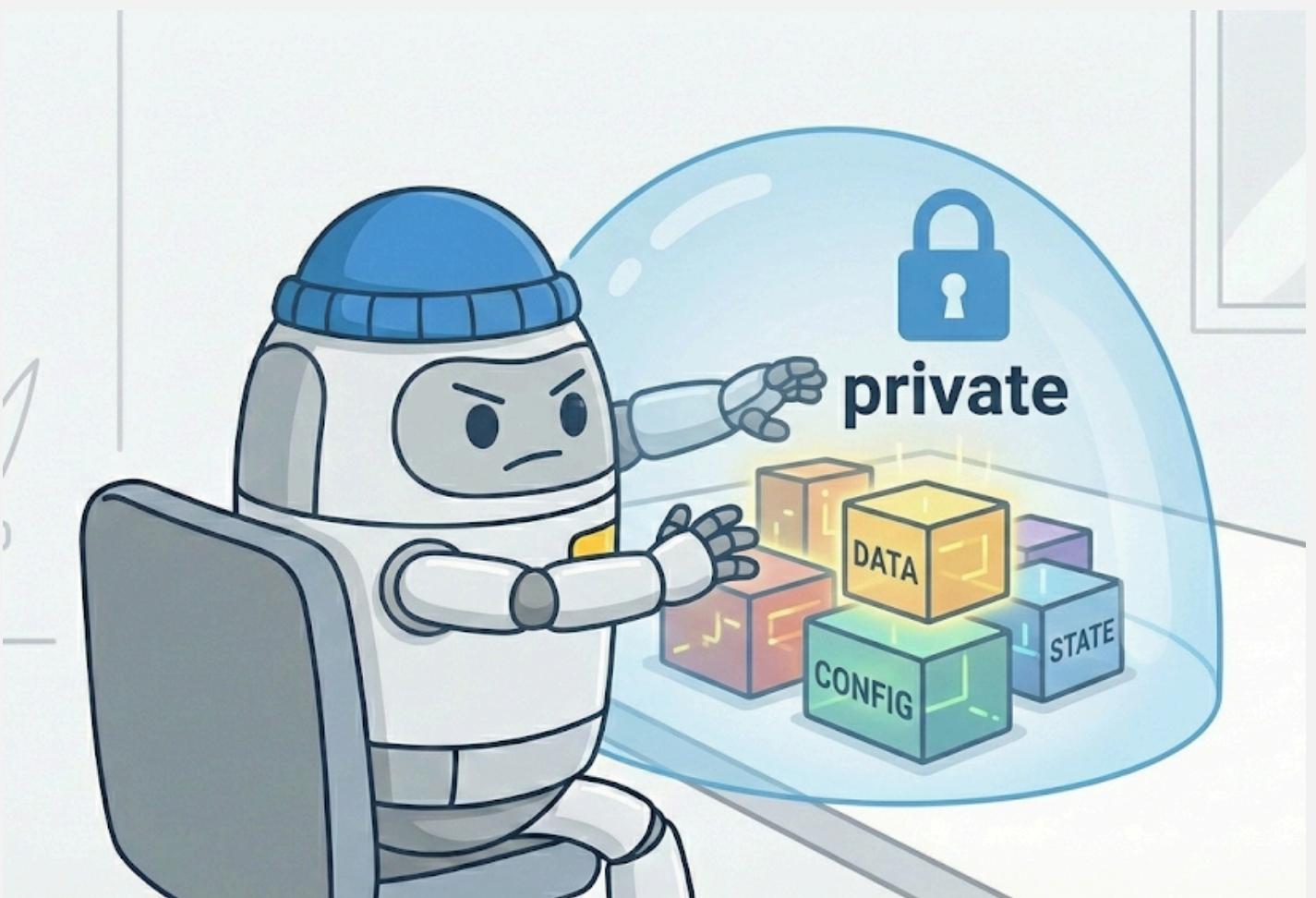
- “**public**” by default.
- “**extends**” for inheritance.
- “**abstract**” classes & methods.
- Properties declared at top.
- No semicolon (;) required.



# C++ VS TYPESCRIPT

→ TS

- Access levels for all members.
- Methods implemented inside class.

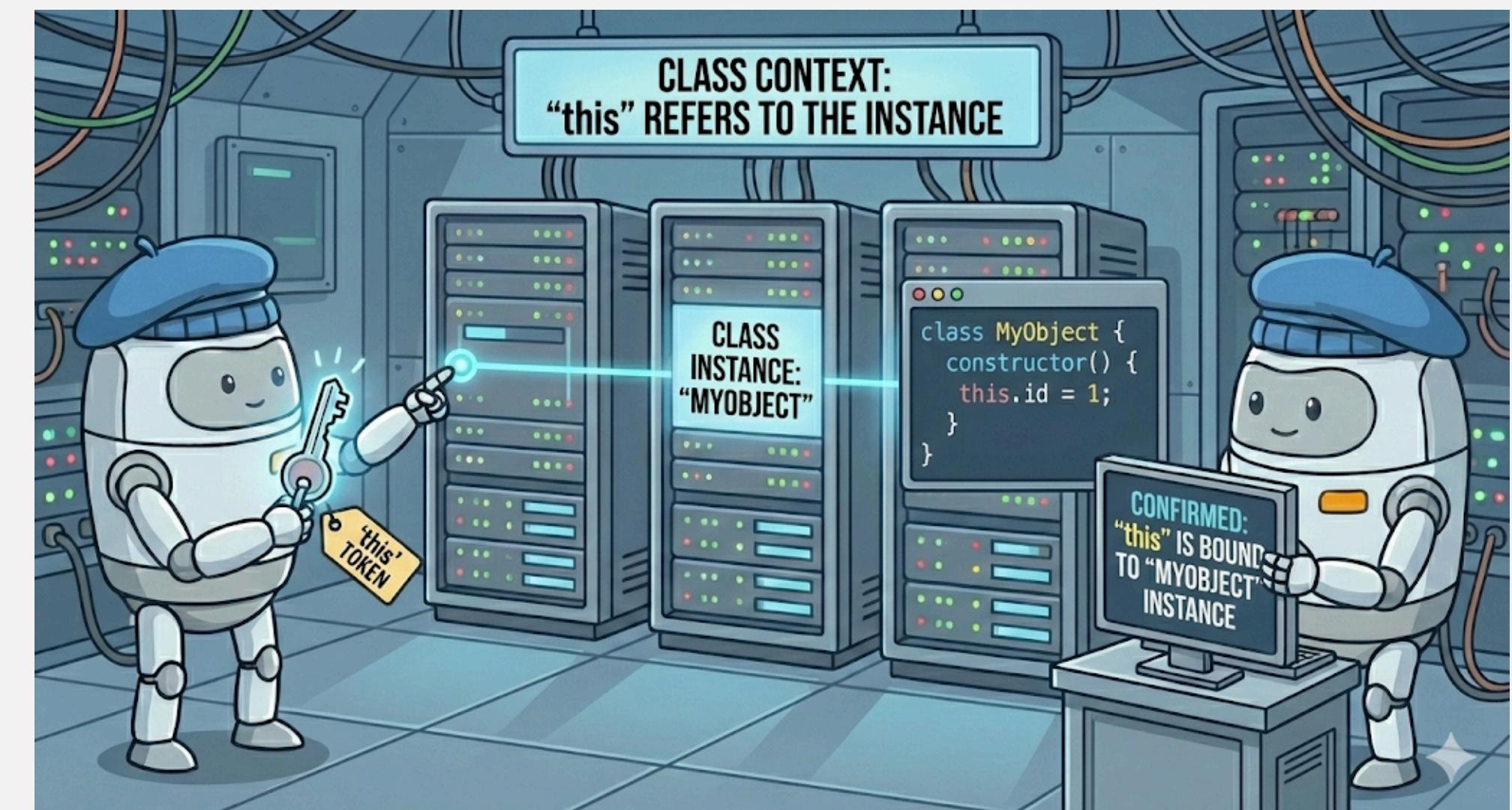


# KEYWORD THIS

- Refers to current object instance.
- Accesses class members (properties & methods).



## Using this



# PARAMETER PROPERTIES

- TypeScript access modifiers in constructor parameters provide a shorthand to automatically declare and initialize class members.



[Concise property initialization example.](#)

# READONLY PROPERTIES

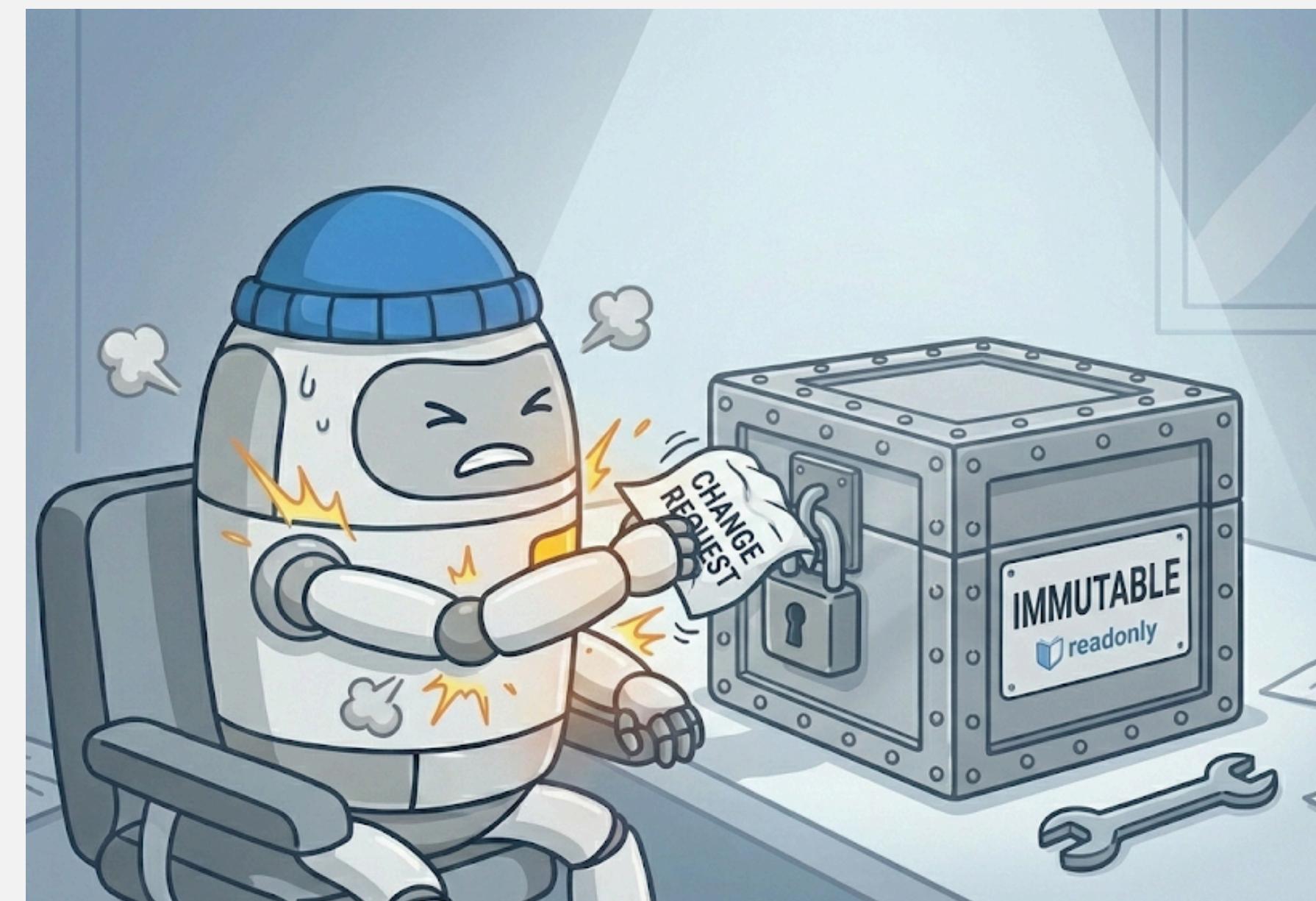
- Readonly properties enforce post-initialization immutability.
- Example [here](#).



# READONLY VS CONST

→ **readonly**

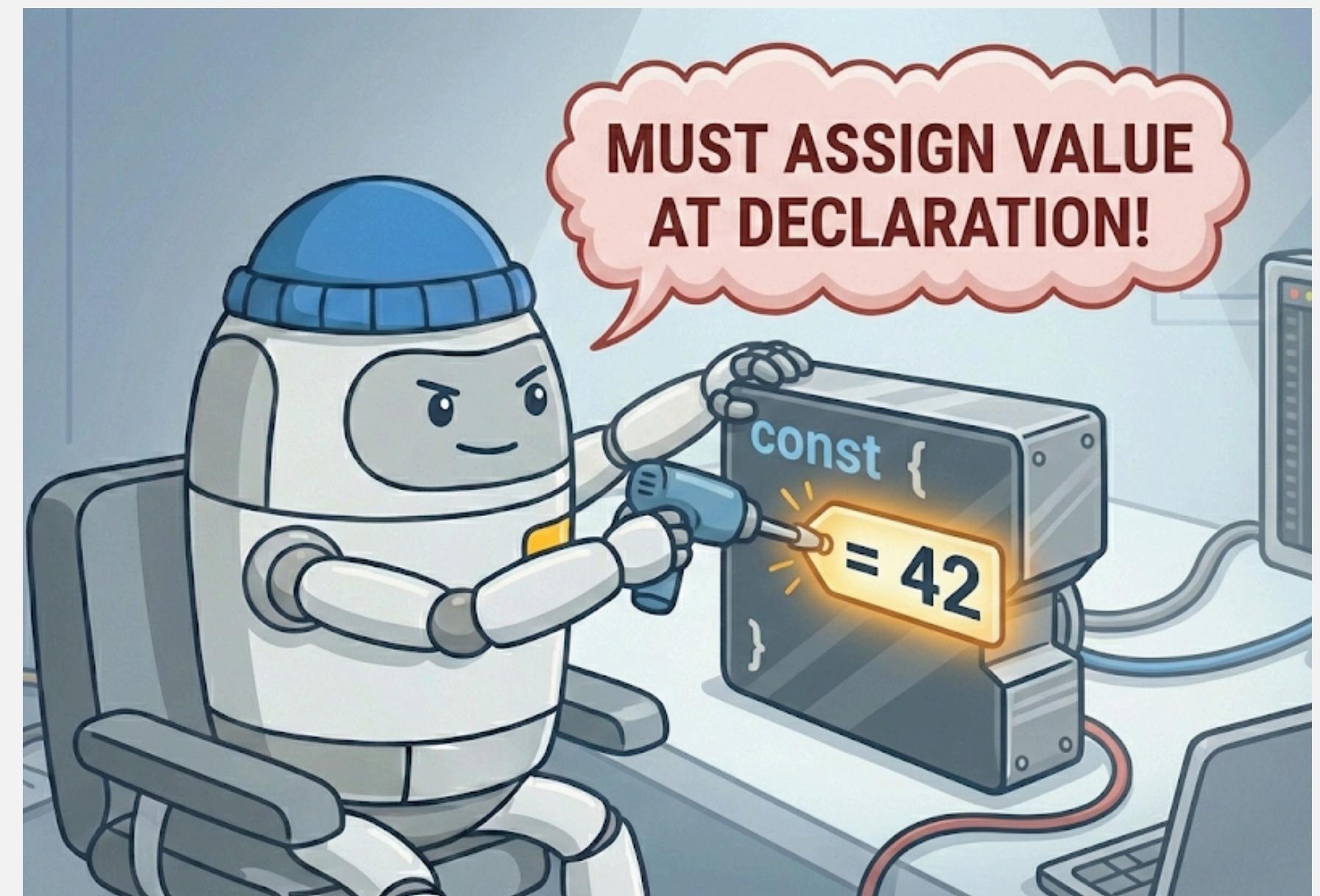
- Values become immutable.
- For class members only.



# READONLY VS CONST

→ **const**

- Immediate initialization.
- For standalone variables.



# CONSTRUCTOR OVERLOADING

- Only a single constructor implementation body.
- Emulating multiple initialization patterns through optional constructor parameters.



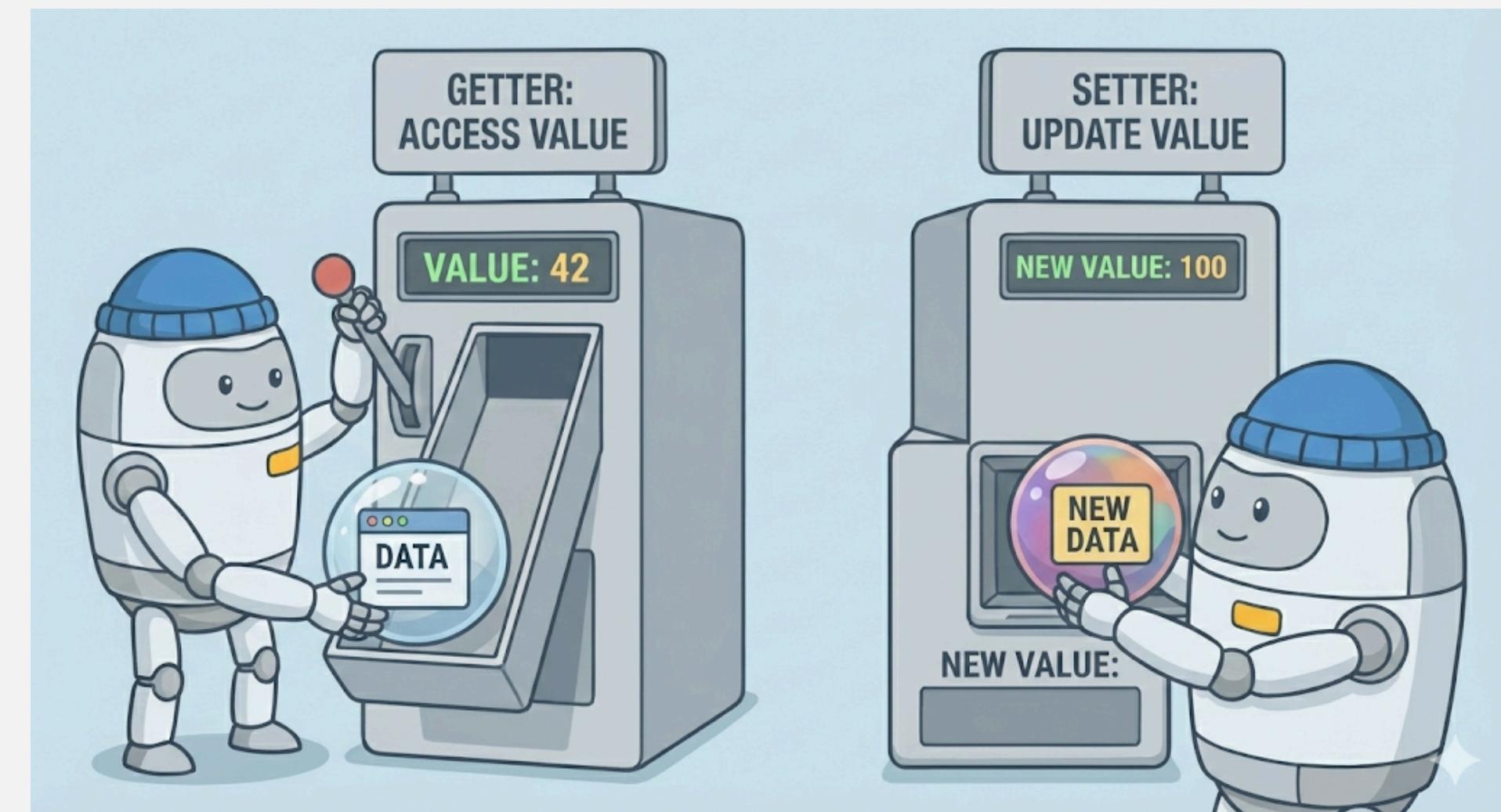
[Constructor-overloading-simulation](#)

# GETTERS AND SETTERS

- Use get / set keywords.
- Implement class accessors.



## Getters and Setters



# STATIC PROPERTY

- Associated with the class itself.
- Accessible across all objects.
- Ideal for global counters.



[Static-property](#)

# CLASS INHERITANCE

- Keyword “**extends**” is essential for establishing basic inheritance hierarchies.
- Multiple inheritance not supported.



[Simple-inheritance](#)

# KEYWORD SUPER()

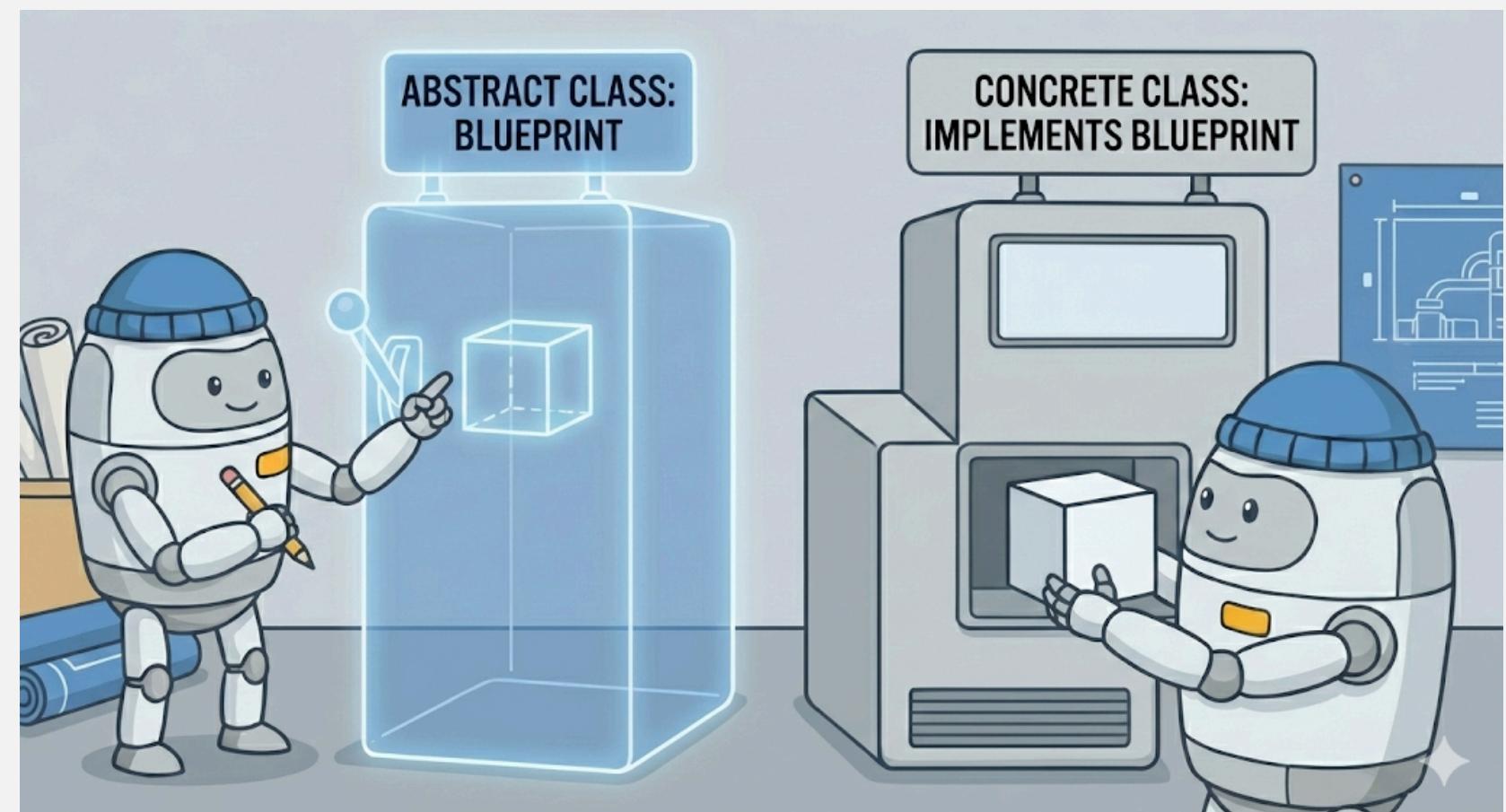
- Invokes base constructor.
- Must call before “**this**”.
- Passes parent parameters.
- Access parent methods (“**super.method()**”).



[Super-usage](#)

# ABSTRACT CLASSES

- Base templates for other classes.
- Cannot be instantiated directly.
- Must be extended by subclasses.
- Enforce core structure across classes.



# ABSTRACT CLASSES

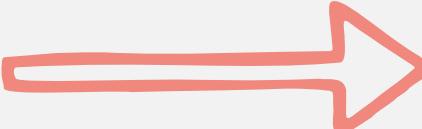
- “**abstract**” keyword required.
- Signatures without implementation.
- Force implementation in subclasses.
- Mix of abstract & concrete members.

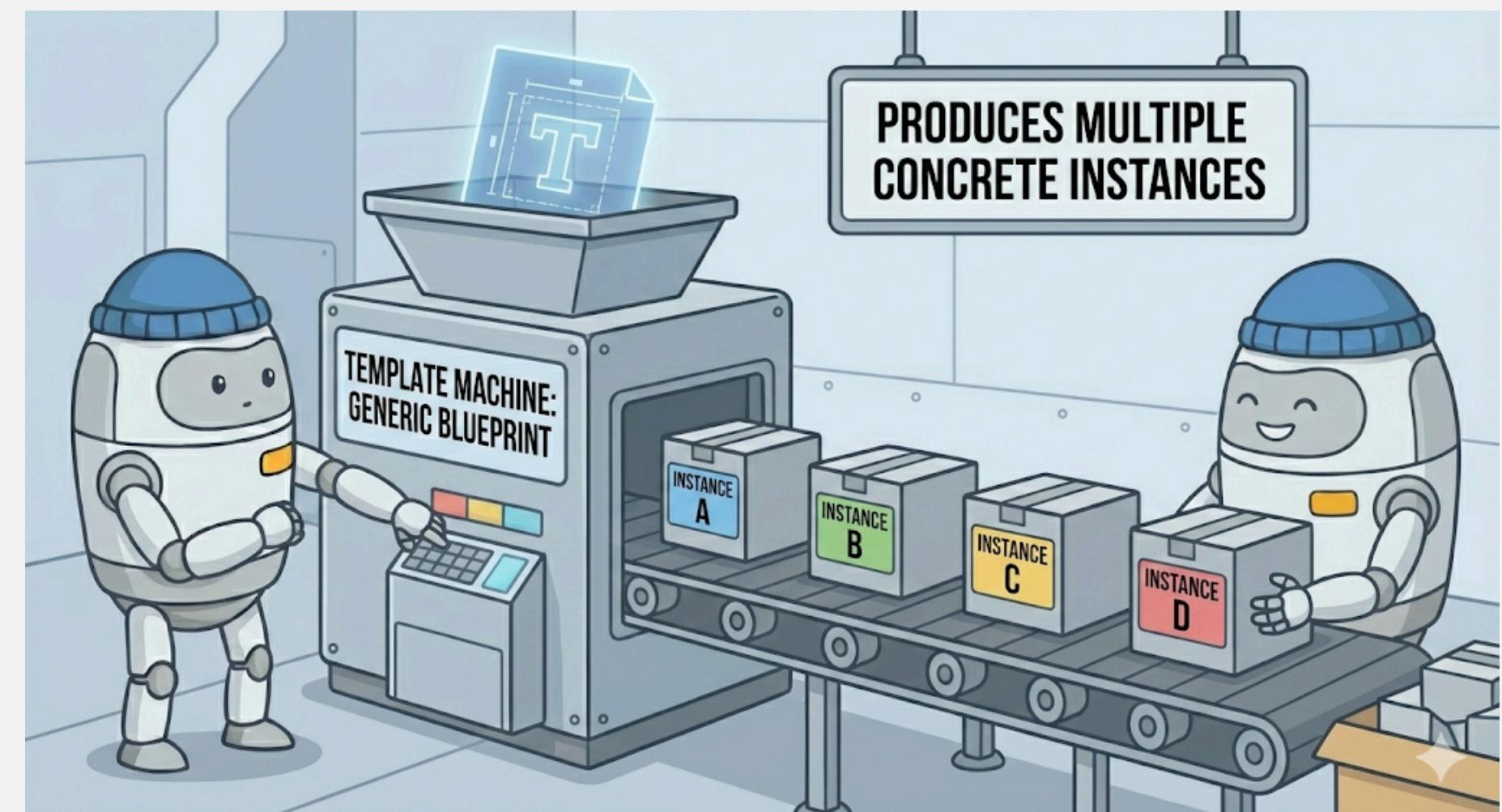


[Abstract-classes](#)

# TEMPLATE IN CLASSES

- Operate with various types.
- Single implementation.
- Flexible & adaptable code.

 **Template-with-classes**



# INTERFACES

# INTERFACES

- Define object shape.
- Enforce properties & methods.
- Implicitly public.



```
interface Task {  
    title: string;  
    priority: number;  
    status: string;  
    assignedTo: string;  
}
```

# INTERFACES

VS

# ABSTRACT CLASSES

- They are completely abstract.
- A class can implement multiple interfaces.
- They act as contracts.
- Everything is public.
- Mix between already implemented methods and abstract methods.
- A class can only inherit /extend a single abstract class.
- They have all available types of visibility.
- They can have a constructor.

# INTERFACE EXTENSION

- **Declaration Merging**: Automatically combines multiple definitions with the same name into one single contract.
- **Interface Inheritance**: Uses the extends keyword to inherit members from other interfaces.



[Interface-extension](#)

# INTERFACE INTERSECTION

- Combines definitions using the & operator.
- Enforces all properties from every intersected type simultaneously.



[Interface-intersection](#)

# EXTENSION VS INTERSECTION



## Extension

- Validates compatibility before merging.
- Errors on type conflicts immediately.
- Recommended approach for scalability.

# EXTENSION VS INTERSECTION

## Intersection

- Combines definitions without name restrictions.
- Merges conflicts, potentially resolving to “never”.
- Risk of unexpected or unusable results.

# MODULES

# MODULES

- Structure complex codebases.
- Partition code into logical units.



# ADVANTAGES

- Prevents name collisions (Encapsulation).
- Facilitates code reuse.



# TYPES OF MODULES

## → CommonJS

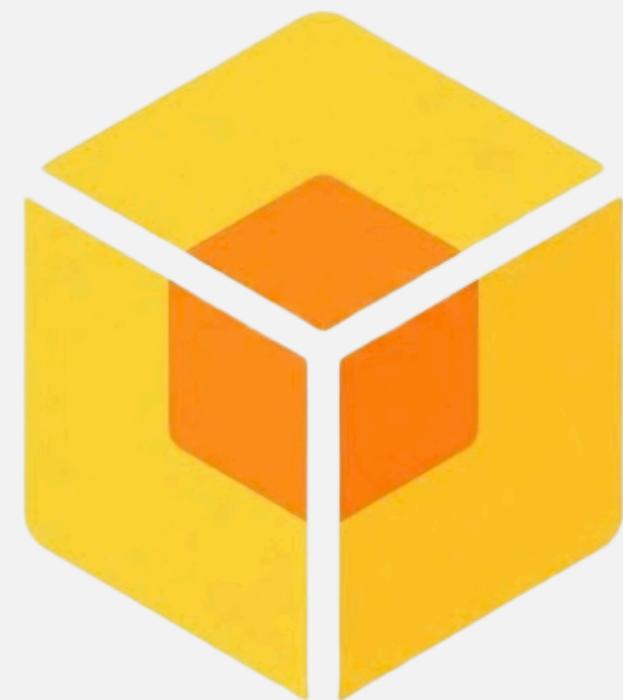
- Legacy ecosystem standard.
- Sequential execution (Synchronous).
- `require()` & `module.exports`



# TYPES OF MODULES

→ **ESModules**

- Modern official specification.
- Non-blocking / Efficient processing.
- **import & export.**



# ESModules

# MODULES ADITIONAL



Enloquent  
Javascript

# CONCLUSION