**Typescript**

**Class**

A class is a blueprint or template used to create objects. It defines the properties (data) and methods (functions) that the created objects will have.

In TypeScript, classes help you organize code using object-oriented programming principles like encapsulation, inheritance, and abstraction

**Object**

An **object** is a concrete instance of a class. It contains real values for the properties defined in the class and has access to all its methods.

When you use the new keyword with a class, you create a distinct object with its own state and behavior derived from that class.

**Constructor**

A **constructor** is a special method in a class that is automatically executed when an object is instantiated. It is primarily used to initialize the object’s properties and set up any required configuration at creation time.

**What is Typescript**

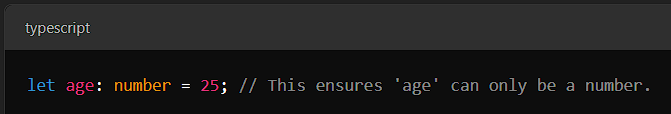
### **TypeScript is a statically typed superset of JavaScript** that allows developers to define and enforce types (like number, string, array, object, etc.) at compile time, making large-scale applications **more robust, readable, and maintainable**

### **Simple Explanation:**

 **JavaScript** is flexible but doesn’t catch type errors until the code runs.

**TypeScript** adds an extra **layer of safety** by checking types **before the code runs**, during development.

. For example:



If you try to assign a string (like "hello") to age, it will show an error during coding itself.

**Better Development Experience**: It provides features like autocompletion, code navigation, and debugging, which make coding faster and easier.

### **Why use TypeScript?**

1. **Error Prevention**: Catches mistakes early during coding, saving debugging time.
2. **Better Team Collaboration**: Other developers can easily understand your code due to clear type definitions.
3. **Works with JavaScript**: It is fully compatible with JavaScript, so you can gradually switch to TypeScript in your project.

### **Disadvantages of TypeScript:**

1. **Learning Curve**:
   * If you’re new to JavaScript, learning TypeScript can feel overwhelming because you need to understand both JavaScript and the additional TypeScript features like types, interfaces, and decorators.
2. **Longer Development Time**:
   * Writing TypeScript code often takes more time than plain JavaScript because you have to define types, interfaces, and more.
   * Small projects may not benefit much, and it can feel like extra work.
3. **Compilation Required**:
   * TypeScript code cannot run directly in browsers. It needs to be converted (or "compiled") to JavaScript first, which adds an extra step.
4. **Tooling and Setup**:
   * Setting up TypeScript in a project requires more configuration compared to plain JavaScript. For example, you need tools like tsc (TypeScript compiler) or bundlers like Webpack.
5. **Potential Overhead for Small Projects**:
   * If the project is small and simple, using TypeScript might not be worth it because its features are more beneficial for large-scale applications.
6. **Community Ecosystem Lag**:
   * Some JavaScript libraries and frameworks may not fully support TypeScript or may require additional type definitions (like @types packages) to work correctly.
7. **Debugging Can Be Tricky**:
   * When debugging in the browser, you usually debug the compiled JavaScript code, not the original TypeScript code, which can make it harder to trace issues.
8. **Complexity in Team Adoption**:
   * If a team is used to JavaScript, switching to TypeScript can slow them down initially, as everyone needs to learn and follow the new typing rules.

**Statically typed language**

A **statically typed language** is a programming language where the type of a variable (like number, string, or boolean) is **fixed at compile time**. This means you must declare the type of data when declaring. the variable will hold the type when you write the code, and the compiler checks the type before the program .

### **Key Features of Statically Typed Languages:**

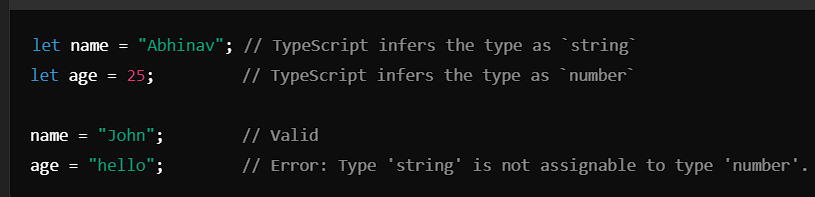
1. **Type Checking at Compile Time**:
   * Errors related to type mismatches are caught during coding or compilation, not while running the program.
2. **Prevention of Type-Related Bugs**:
   * Because types are checked early, you avoid common runtime errors like passing the wrong type of data to a function or using a variable inappropriately.
3. **Better Documentation**:
   * The types themselves act as documentation, making it easier to understand the code and the data it handles.
4. **Performance**:
   * Since the type information is known at compile time, some statically typed languages can optimize the code better, improving performance.

### **Implicit vs. Explicit Types in TypeScript**

TypeScript allows you to use **types** in two ways: **implicitly** (TypeScript guesses the type) or **explicitly** (you specify the type). Let’s break it down with simple examples.

### **Implicit Types**

When **automatically infers the type** of a variable based on the value assigned to it, it’s called **implicit typing**. You don’t need to explicitly declare the type.

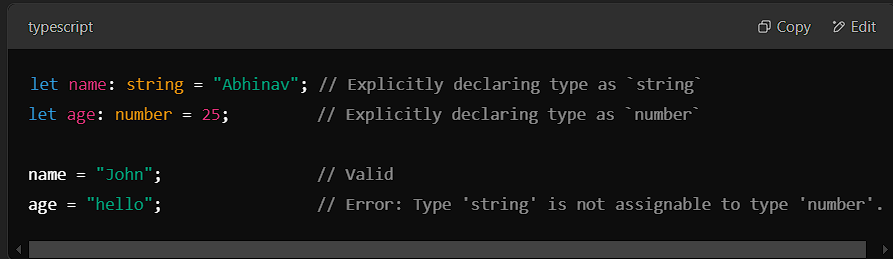


#### **How It Works:**

* TypeScript looks at the assigned value and assumes its type.
* **Benefit**: Saves time and makes the code cleaner.
* **Drawback**: If you're not careful, implicit typing might cause unexpected issues when dealing with complex scenarios.

### **Explicit Types**

When you **manually declare the type** of a variable, it’s called **explicit typing**. You specify what the variable should have.



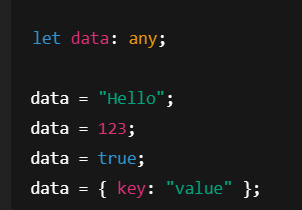
#### **How It Works:**

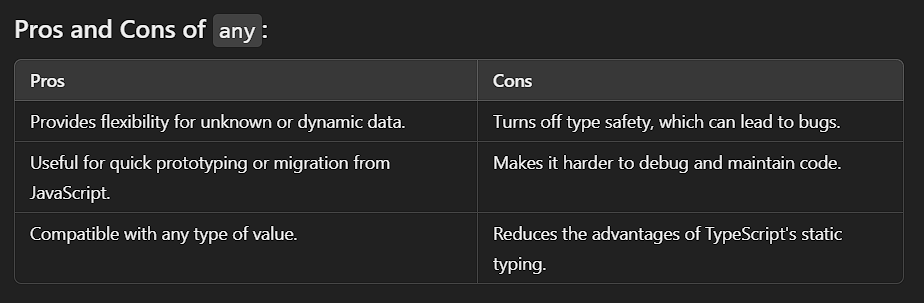
* You declare the explicitly using a colon (:) followed by the type.
* **Benefit**: More control and clarity, especially for larger projects or when collaborating in teams.
* **Drawback**: Adds extra typing effort.

**Any type**

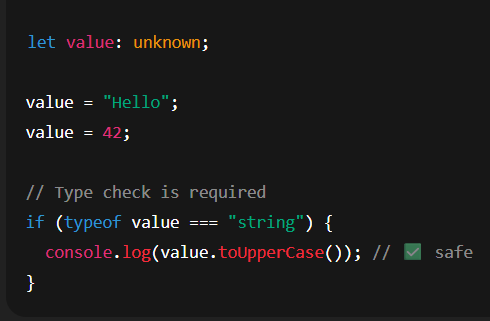
In TypeScript, the any type is a special type that can represent **any** , similar to JavaScript's dynamic typing. It essentially turns off type checking for that variable, allowing you to assign values of any type to it.

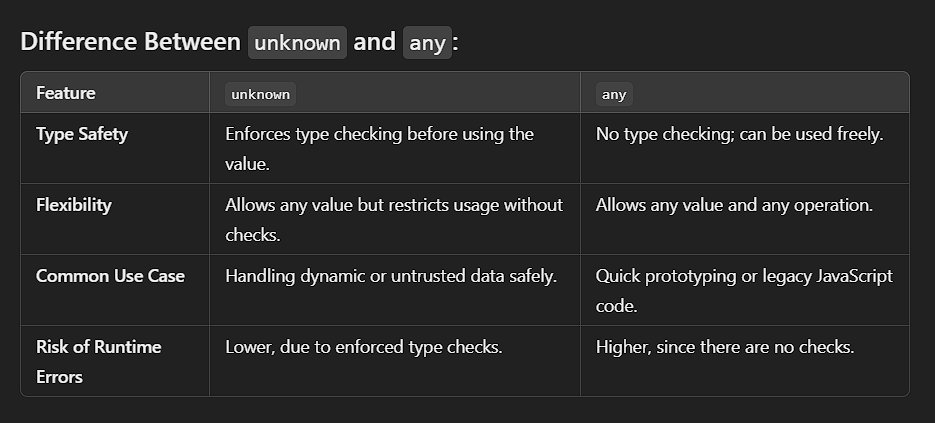
**Avoid it** unless absolutely necessary because it disables type checking.





**Unknown Type**  
  
  
  
In TypeScript, the unknown type is similar to any because it can hold any value, but it is **safer and more restrictive**. While any disables type-checking completely, unknown requires you to explicitly check the type of a value before performing operations on it.

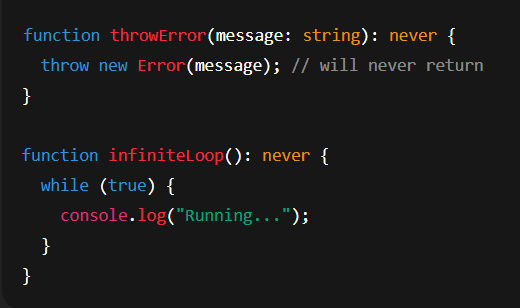


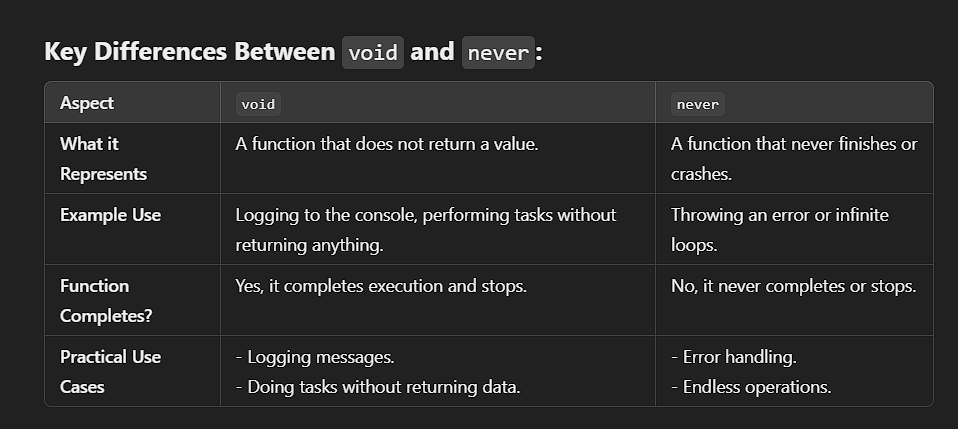


### **never Type**

The never type represents a value that **never occurs**. It’s used to indicate situations where:

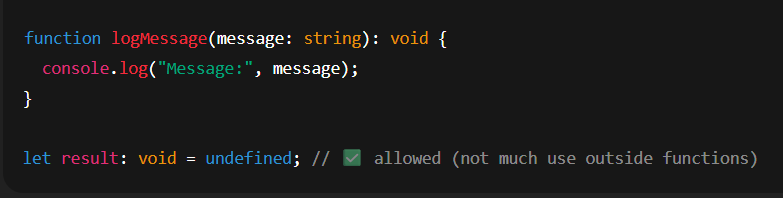
1. A function **never returns** (e.g., it throws an error or goes into an infinite loop).
2. A type is **impossible to assign** (e.g., a variable that can't have any valid value).





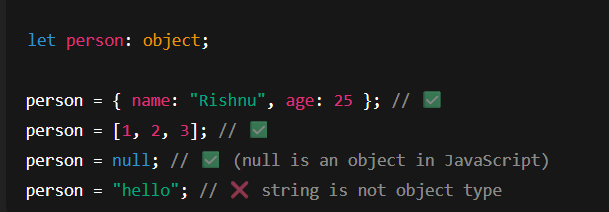
**VOID**

Used for functions that **do not return a value**.



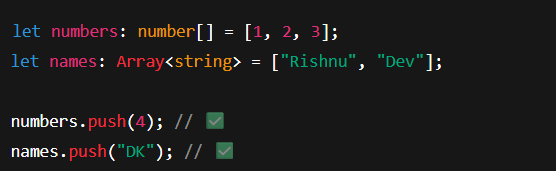
**Object**

Represents any **non-primitive** value (not number, string, boolean, etc).



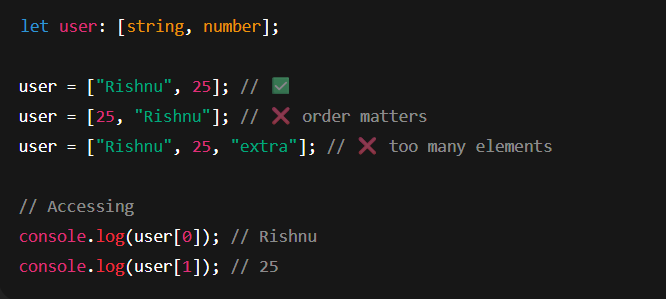
**ARRAY**

Use type[] or Array<type>.



TUPLE

Fixed-length arrays with **specific types at each position**

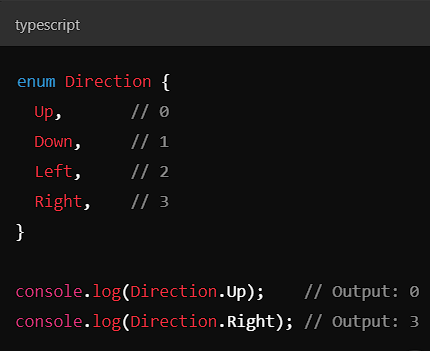


**Enum Type**

In TypeScript, **enum** is a special data type used to define a set of named constants. Enums make your code more readable and maintainable by giving meaningful names to values.

### **Types of Enums in TypeScript:**

1. **Numeric Enums**Numeric enums have assigned numbers starting from 0 by default. You can also manually specify values.

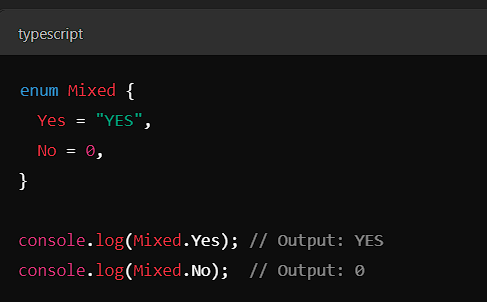


**2.String Enums**

String enums let you assign string values to constants, making them more descriptive.



1. **Heterogeneous Enums**Enums can contain both numeric and string values, but this is less common.

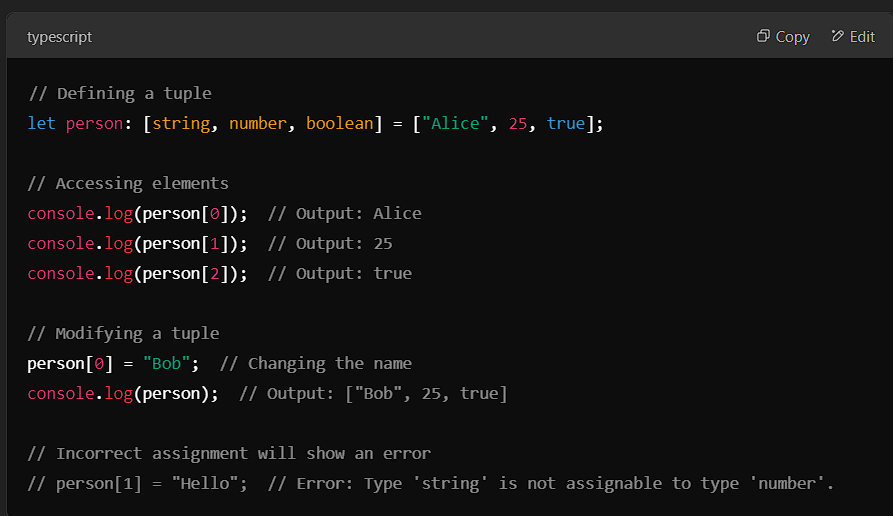


**Tuple**

In TypeScript, a **tuple** is a fixed-size array where each element can have a different type. It allows you to group values of different types together into a single entity, maintaining the order of the elements.

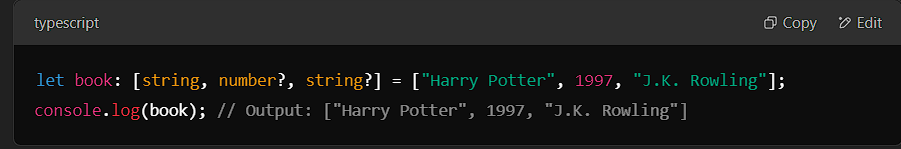
### **Key points about Tuples in TypeScript:**

1. **Fixed Length**: The number of elements in a tuple is fixed, and TypeScript checks that the tuple matches the defined type.
2. **Different Types**: Each element of a tuple can have a different type (e.g., string, number, boolean).
3. **Index-based Access**: You can access the tuple elements using their index.



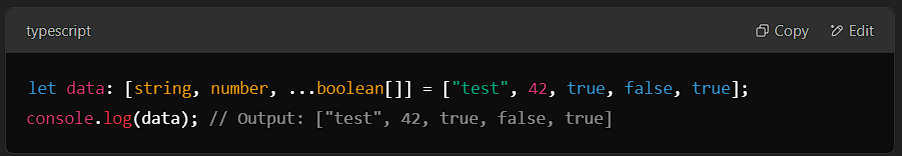
### **Tuple with Optional Elements:**

You can also define optional elements in a tuple using the ? symbol.



### **Tuple with Rest Elements:**

You can define tuples with a fixed number of elements followed by a rest parameter to represent any number of remaining elements.



### **Objects in TypeScript:**

An **object** is a collection of key-value pairs where each key (property) is a string (or symbol), and each value can be any valid type (string, number, boolean, array, or even another object).

### **Explanation of Your Points:**

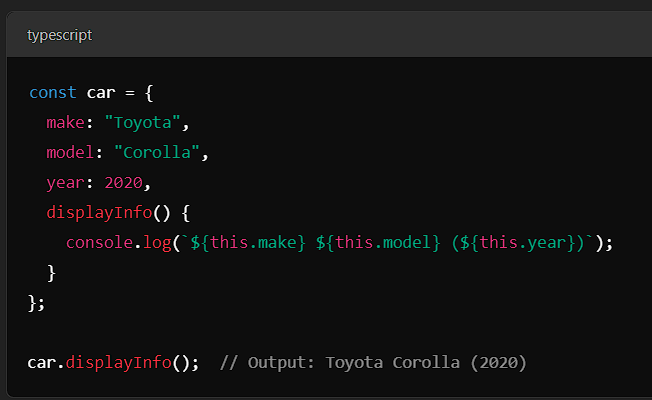
#### **a. Readyone (Possibly 'Readonly'):**

* **Readonly**: In TypeScript, you can create **readonly** objects or properties, meaning once set, they cannot be modified.
* **Readonly** is a modifier you can use to make a property or an entire object immutable.



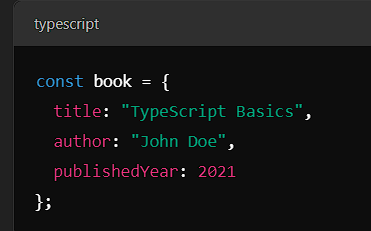
#### **b. Method:**

* **Method** in an object refers to a function defined as a property of the object. Methods are used to define behaviors associated with that object.



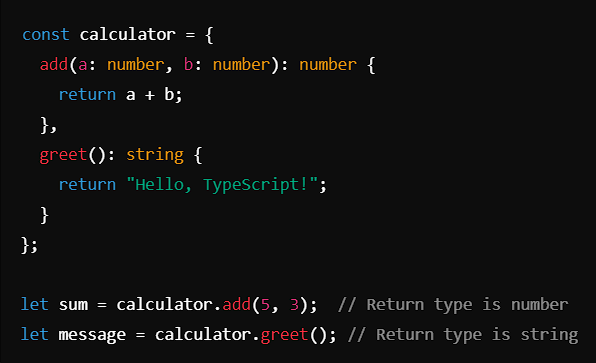
#### **c. Specific Values:**

* This refers to assigning specific values to properties of an object. Each key-value pair represents a property and its associated value.



#### **d. Return Type:**

* The **return type** refers to the type of value that a method or function will return. In TypeScript, you can define the return type explicitly for methods or functions.

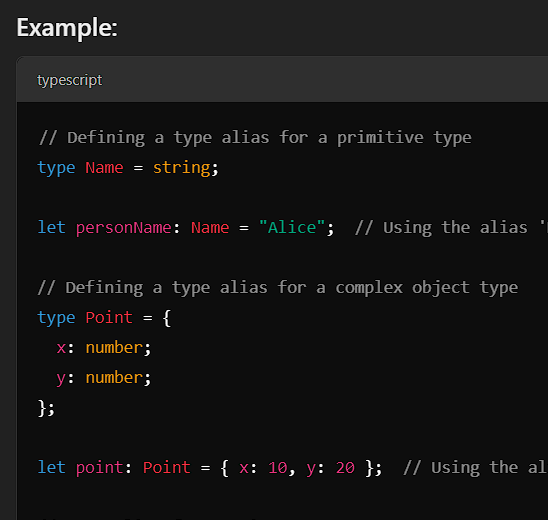


### **Summary of Key Concepts in TypeScript Objects:**

1. **Readonly**: Used to make properties of objects immutable.
2. **Methods**: Functions defined as properties of an object.
3. **Specific Values**: Values assigned to object properties.
4. **Return Type**: The type of value a method or function will return.

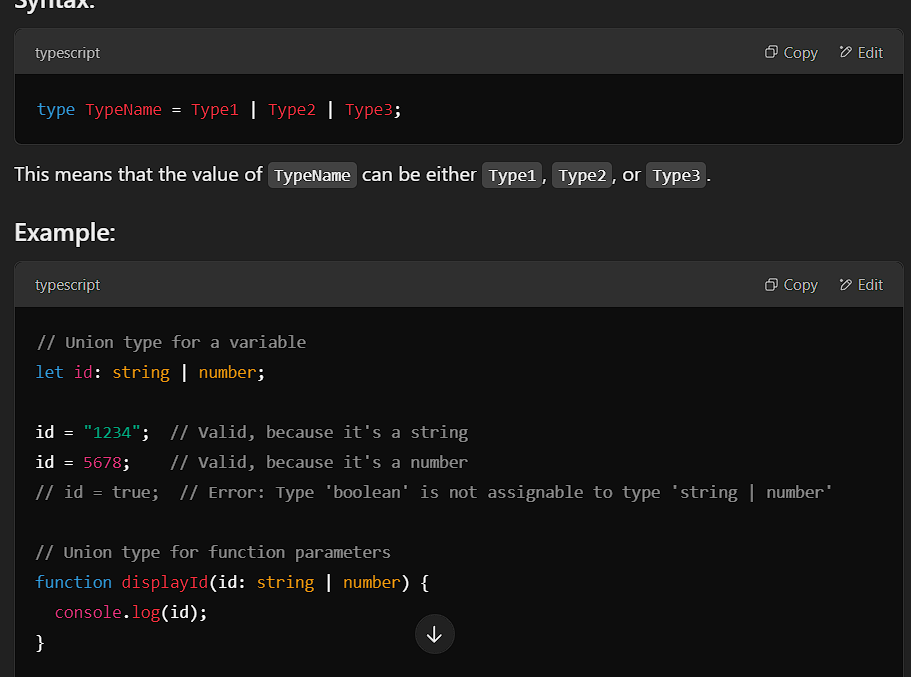
**Type alias**

In TypeScript, a **type alias** allows you to create a custom name for a type. It enables you to simplify and make your code more readable by giving a meaningful name to a type, which could be a primitive, a union, an intersection, or even a complex object type



**Union type**

A **union type** in TypeScript allows you to define a variable or parameter that can hold multiple types. It means the value can be one of several specified types, providing flexibility while still maintaining type safety. The union type is represented using the | (pipe) symbol.



**Type intersection**

In TypeScript, **intersection types** allow you to combine multiple types into one. The resulting type will have all the properties and methods from each of the types involved in the intersection. This is represented by the & (ampersand) operator.

An **intersection type** combines several types into one, meaning a value of this type must conform to all the combined types.



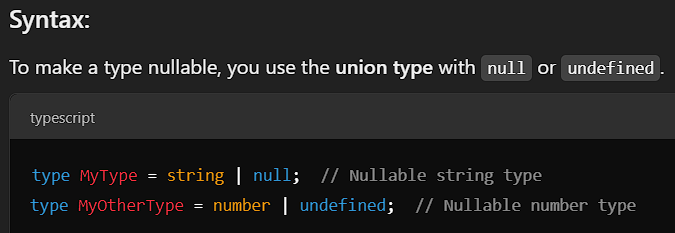
**Literal types**

In TypeScript, **literal types** allow you to specify the exact value that a variable can hold, rather than just its type. This can be helpful when you want a variable or a function parameter to have a very specific value or a set of possible values.



**Nullalbe type**

In TypeScript, a **nullable type** allows a variable to hold a value of a specified type or null or undefined. This is useful when you want to indicate that a variable can either have a meaningful value or no value at all (i.e., it can be null or undefined).

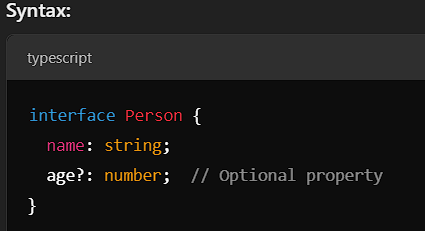


**Option property, element, call**

In TypeScript, **optional properties**, **optional elements**, and **optional calls** are features that allow you to define properties, elements, or functions that may or may not be present. These features provide flexibility when working with types and interfaces, especially when some data might be missing or undefined.

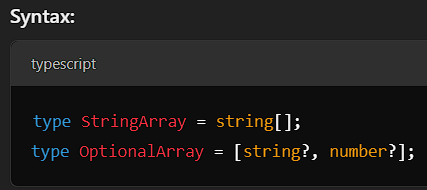
### **1. Optional Properties:**

Optional properties in TypeScript allow an object to have properties that may or may not be present. This is done by appending a ? to the property name in an interface or type.



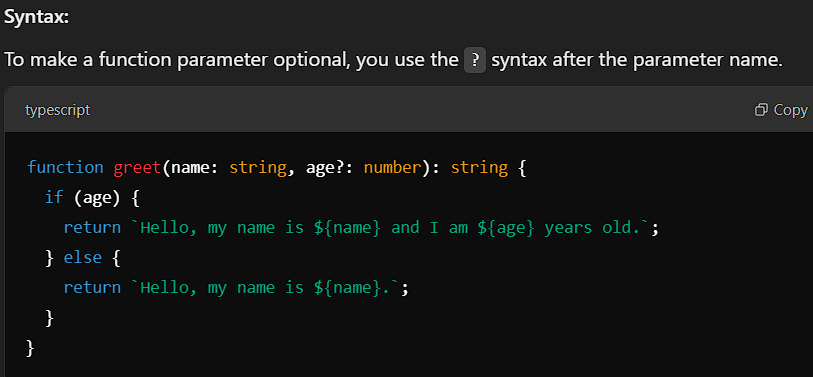
### **2. Optional Elements (in Arrays):**

Optional elements in arrays allow some array elements to be optional. This is particularly useful when you want to create arrays where some elements may be undefined.



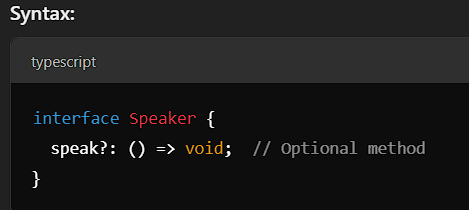
### **3. Optional Function Arguments (Optional Parameters):**

Function arguments can also be optional. This is helpful when you want to define a function that doesn't necessarily require all its arguments to be provided.



### **4. Optional Method Calls:**

In TypeScript, methods can also be optional in interfaces or types. This is particularly useful when defining an object that may or may not have a certain method.

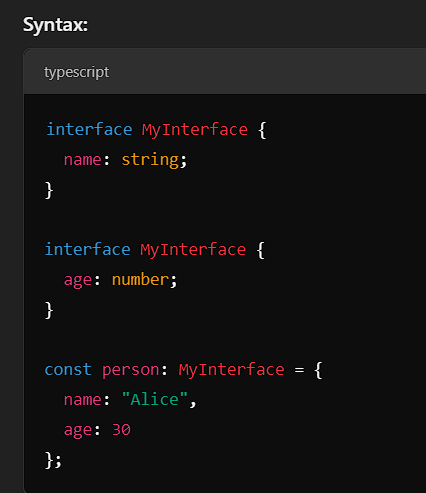


**Interface**

In TypeScript, **interfaces** are used to define the structure of an object. Interfaces are powerful and provide great flexibility in defining and enforcing types. Two of the advanced features of interfaces are **reopening an interface** and **interface inheritance**.

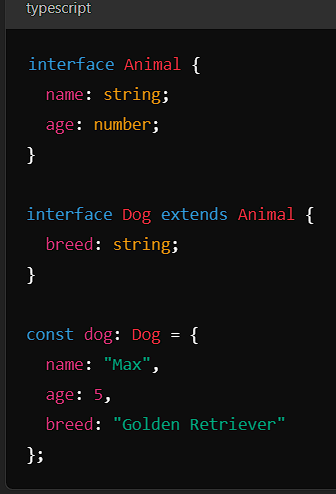
### **a. Reopening an Interface:**

In TypeScript, you can **reopen an interface** to extend or add new properties. This means that you can declare the same interface multiple times, and TypeScript will automatically merge them together. This is useful for extending existing interfaces or adding additional properties to interfaces declared in third-party libraries.



### **b. Interface Inheritance:**

In TypeScript, interfaces can inherit from other interfaces. This means one interface can extend another interface, inheriting its properties and methods. This is useful for creating a hierarchy of interfaces or adding more specific properties to a base interface.



**Modifiers in Classes (Access Modifiers)**:

Access modifiers are used to control the visibility of class properties and methods. The three main modifiers are:

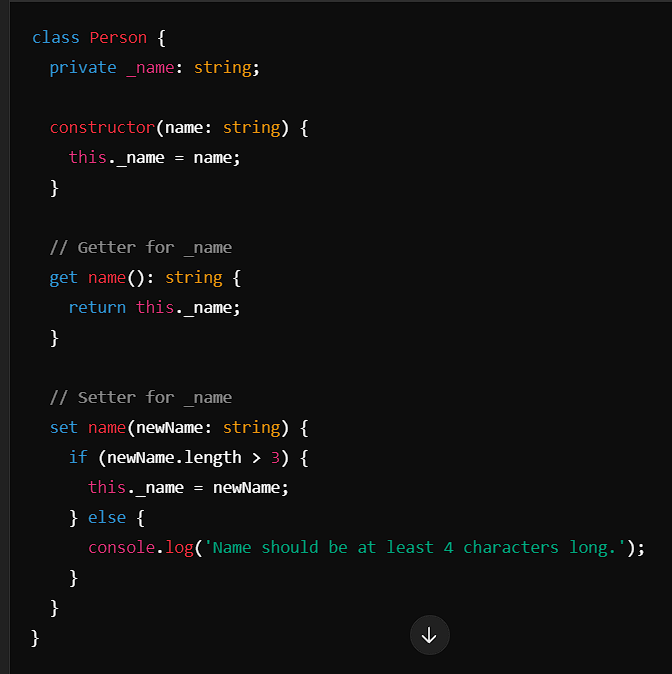
1. **public**: The property or method is accessible from anywhere.
2. **private**: The property or method is accessible only within the class.
3. **protected**: The property or method is accessible within the class and its subclasses.



### **b. Getters and Setters:**

In TypeScript, **getters** and **setters** are special methods used to get and set the values of private or protected properties. This allows you to control how the properties are accessed or modified.

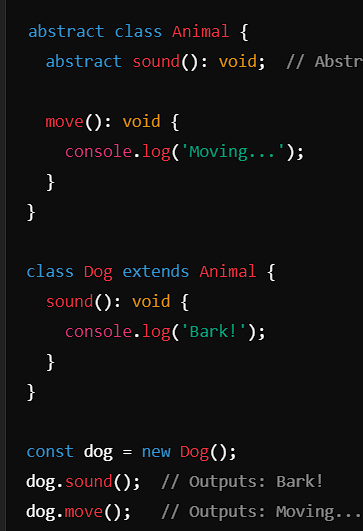
* **Getter**: A method used to access the value of a property.
* **Setter**: A method used to set the value of a property.



### **c. Abstract Class:**

An **abstract class** is a class that cannot be instantiated directly. it can contain abstract methods (which don't have implementation) and concrete methods (with implementation). Abstract classes are often used as base classes to be extended by other classes.

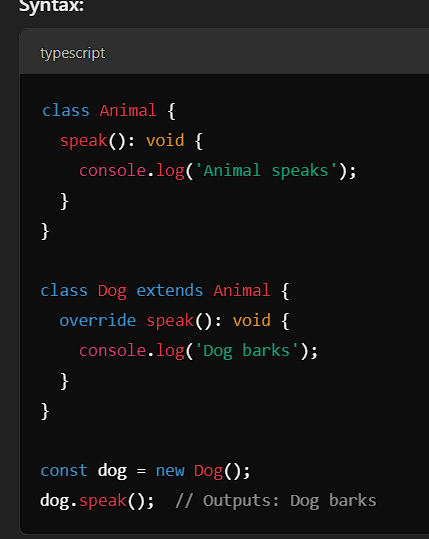
* **Abstract methods**: These are methods that must be implemented in derived classes.
* **Concrete methods**: These methods can have an implementation and can be inherited by derived classes.



### **d. Overriding Methods:**

Method **overriding** occurs when a subclass provides a specific implementation for a method that is already defined in its parent class.

In TypeScript, you can override methods in subclasses by using the same method name as in the parent class. The override keyword can be used (in TypeScript 4.3 and later) to explicitly mark the method as an override, providing better type safety.



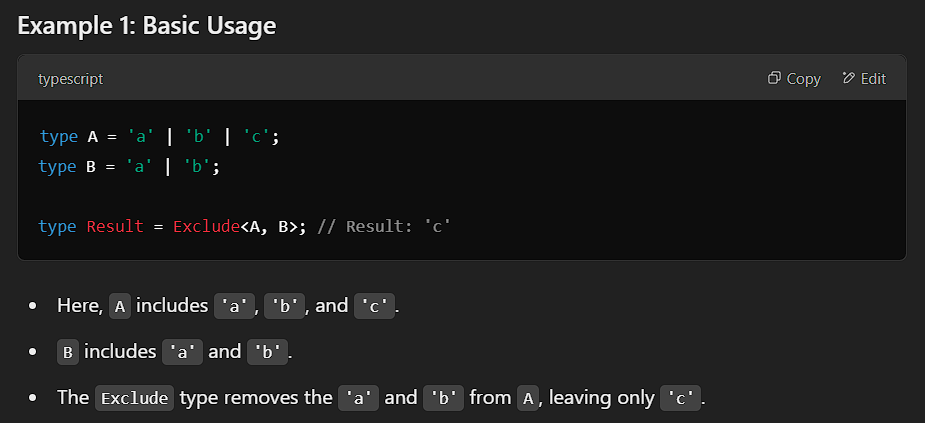
### **e. Difference between Class and Abstract Class:**

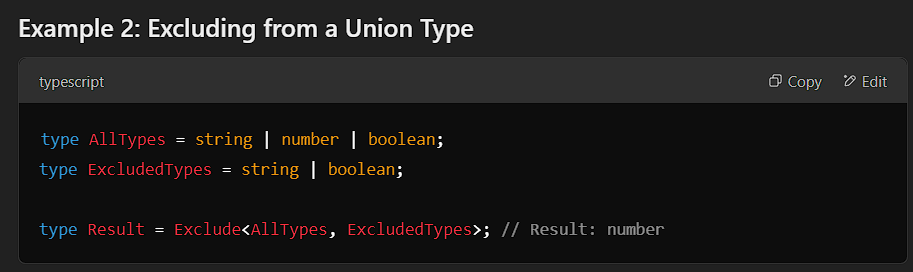
The main difference between a **class** and an **abstract class** lies in their ability to be instantiated and how they handle methods.

1. **Class**:
   * A normal class can be instantiated directly using the new keyword.
   * It can have both abstract and non-abstract methods, but it must have concrete implementations for all methods.
   * You cannot define an abstract method in a regular class.
2. **Abstract Class**:
   * An abstract class cannot be instantiated directly.
   * It can contain abstract methods (without implementation) and concrete methods (with implementation).
   * Subclasses must implement all abstract methods.
   * It is meant to be extended by other classes.

* **Exclude<T, U>**

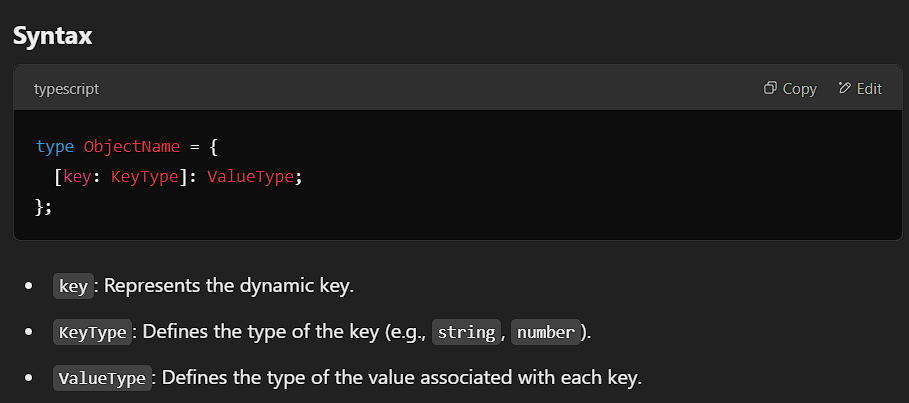
In TypeScript, Exclude<T, U> is a utility type that constructs a type by **excluding** from T all properties that are assignable to U. It’s part of the set of built-in utility types provided by TypeScript.

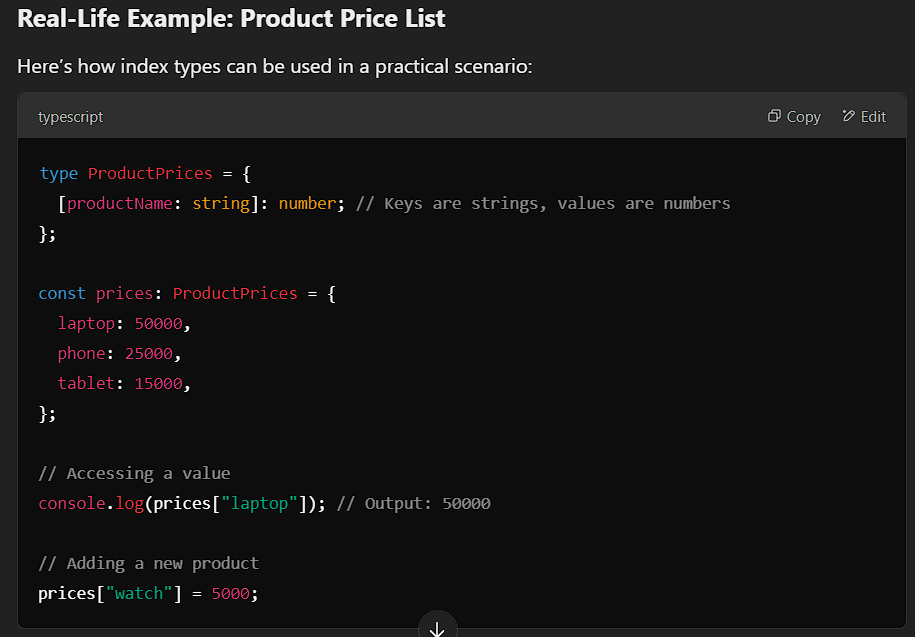




**index types**

**Index types** allow you to work with **dynamic keys and values** of objects — like accessing properties by string names and keeping types safe.  
They are **extremely useful** when you don’t know all the property names in advance.





### **What is a Decorator?**

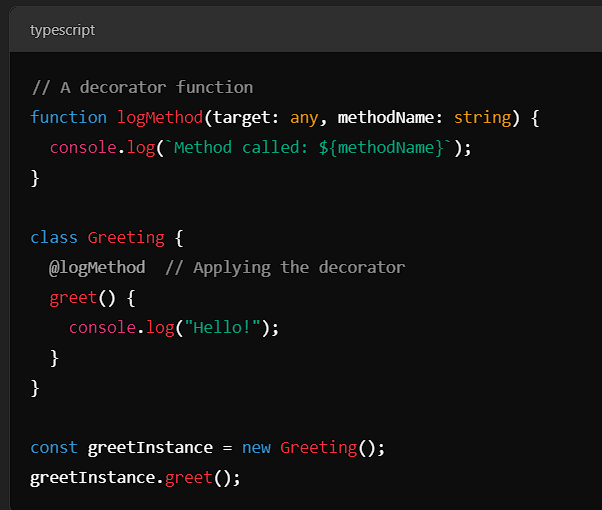
A **decorator** is a special type of **function** that allows you to **add extra features** or **modify** the behavior of a method, property, or class without changing the actual code inside.

### **How does it work?**

* You use a **decorator** by putting the @ symbol **above** a class method or property.
* The decorator function can add features like **logging**, **validation**, or **timing** to your code.

### **Why Use a Decorator?**

1. **Add Extra Features**: You can add things like **logging**, **validation**, or **security checks** to your code.
2. **Reusability**: Decorators help you reuse the same feature across different parts of your code without rewriting it.
3. **Keep Code Clean**: Decorators allow you to keep your code organized and separate the extra features from the main logic.



Types of Decorators

## 1. Class Decorator

### Use it to modify or observe the **class itself**.

## 2. Property Decorator

### Used to modify or track **class properties** (not methods).

## 3. Method Decorator

### Used to intercept or modify class **methods**.

## 4. Accessor Decorator

### Used to decorate **getters/setters**.

## 5. Parameter Decorator

### Used to decorate **constructor or method parameters**.

## Decorator Evaluation Order

When multiple decorators are applied to a **class**, **method**, or **property**, they are evaluated and called in a specific order.

### **Evaluation Order**

**Decorator Expressions** are **evaluated top to bottom**.

**Decorator Functions** are **called bottom to top** (i.e., applied in reverse).

## Decorator Composition

## When multiple decorators are used on the same declaration, they form a **composition chain**, where:

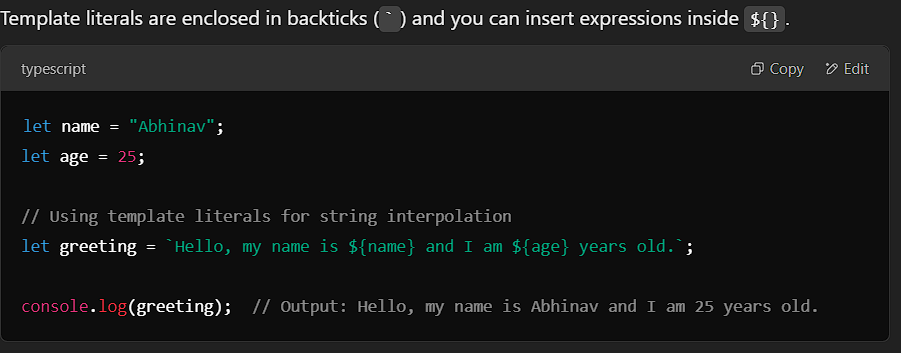
The **last decorator applied** is the **first to execute**.

The decorators wrap each other **like nested functions**.

## Decorator Factory

A **decorator factory** is a function that **returns the actual decorator** function. It allows you to pass arguments into the decorator.

**string interpolation**



### **SOLID Principles**

**SOLID** is a set of five design principles in object-oriented programming that help create **maintainable**, **flexible**, and **scalable** software.

1. **Single Responsibility Principle (SRP)**:
   * A class should have **one reason to change**.
   * It should only have one job or responsibility.
2. **Open/Closed Principle (OCP)**:
   * A class should be **open for extension** but **closed for modification**.
   * New behavior should be added by extending classes, not by modifying existing ones.
3. **Liskov Substitution Principle (LSP)**:
   * A **child class must behave like the parent** — without causing errors, crashes, or unexpected behavior.
   * Derived classes should behave in ways that do not break the functionality of the base class.
4. **Interface Segregation Principle (ISP)**:
   * Don’t force a class to implement interfaces it doesn't use.
   * It's better to have **many specific interfaces** than one general-purpose interface.
5. **Dependency Inversion Principle (DIP)**:
   * High-level modules should depend on **abstractions**, not on low-level modules.
   * Both high-level and low-level modules should depend on the same abstraction (e.g., interfaces).

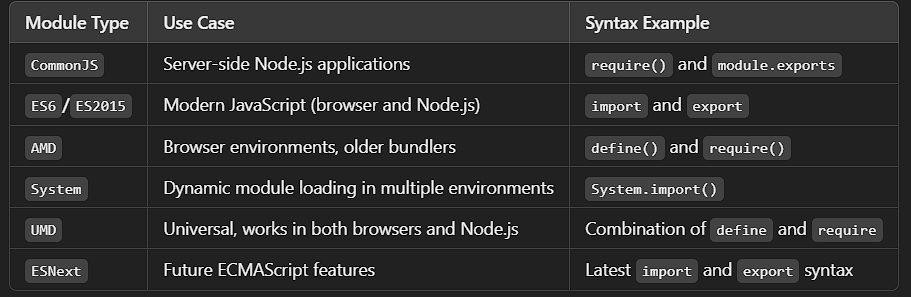
**Transpiler**

A **transpiler** (short for **source-to-source compiler**) is a tool that converts **source code written in one programming language** into **source code of another language**

### **TypeScript as a Transpiler**

* **TypeScript** is a **transpiler** because it converts TypeScript (.ts) code into JavaScript (.js).
* TypeScript does not output machine code (like a regular compiler does); instead, it outputs source code (JavaScript) that can run in any JavaScript environment (like browsers or Node.js).

**Common Module Types**



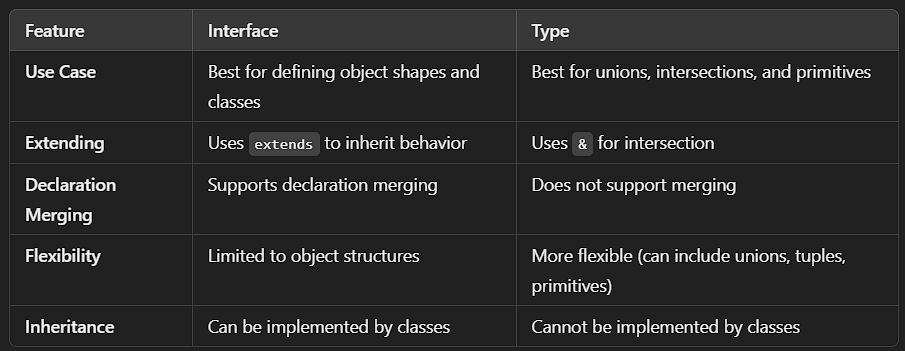
**type vs interface**

In TypeScript, both **type** and **interface** are used to define the structure of objects, but they have key differences:

* **interface** is mainly used to define **object shapes** and is best for working with **classes**, allowing for **extension** (extends) and **declaration merging**. It’s ideal when you want to define object structures and work with inheritance-like behavior.
* **type** is more **flexible** and can define **primitives**, **unions**, **intersections**, and **tuples**, making it suitable for complex type compositions. It can't merge declarations, unlike **interface**.

### **Key Points:**

* **Declaration Merging**: **interface** supports merging multiple declarations, while **type** does not.
* **Extending**: **interface** uses extends for inheritance, while **type** uses & for intersection.
* **Use Case**: Use **interface** for object shapes and **type** for more complex and flexible types (unions, intersections, etc.).



### **Mixin**

### **What is a Mixin?**

A **mixin** is a programming concept used to add reusable features or functionalities to classes without relying on traditional inheritance. In TypeScript, mixins let you combine behaviors from multiple sources, making it easier to create flexible and modular code.

### **Why Use Mixins?**

* To **reuse code** across multiple classes.
* To **avoid deep inheritance chains**.
* To **combine multiple behaviors** in a single class.

### **What is a Singleton?**

A **singleton** is a design pattern that ensures a class has only **one instance** throughout the application and provides a global access point to that instance.

### **Key Features:**

1. **Single Instance**: Only one object of the class is created.
2. **Global Access**: The instance is shared across the entire application.
3. **Controlled Creation**: The instance is created lazily or eagerly, depending on the implementation.

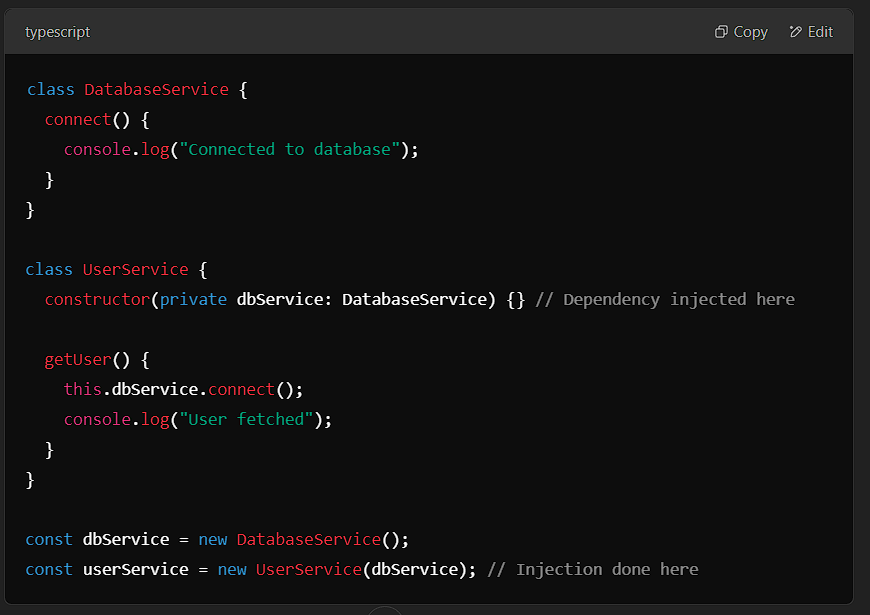
**Dependency Injection**

**Dependency Injection is a design pattern where a class receives its dependencies from the outside, rather than creating them inside.**.

**"Don't create your own tools — ask for them to be given to you."**

### **Key Points:**

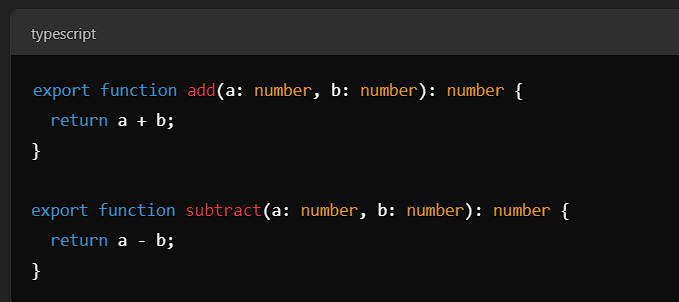
1. **Loose Coupling**: Classes don't depend on how their dependencies are created, just on what they do.
2. **Flexibility**: Dependencies can be easily changed or replaced without modifying the class.
3. **Testability**: Easier to test as you can inject mock dependencies during testing.



**Module in TypeScript:**

A **module** in TypeScript is a way of organizing and encapsulating code. It allows you to divide your code into separate files, and each file can export specific parts of its code (like variables, functions, classes, etc.) that can then be imported into other files. This helps to manage the code more effectively, especially in large applications.

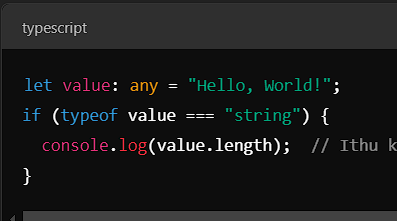
TypeScript modules use **export** and **import** to expose and use the code from different files.



**Type Guards in TypeScript**

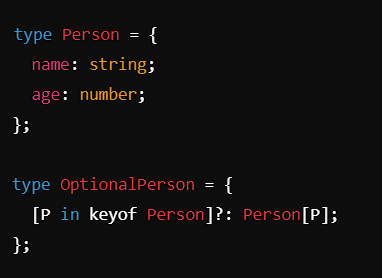
Type guards in TypeScript are a mechanism that helps to check the type of a variable dynamically. When working with variables of type any or multiple possible types, type guards help to narrow down the type to a more specific one.

**typeof** is a built-in type guard in TypeScript



**Mapped Types in TypeScript**

“Mapped types in TypeScript are a powerful feature that allow you to create new types by transforming the properties of existing types. For example, you can make all properties optional, readonly, or even rename them using mapped types.



**utility types**

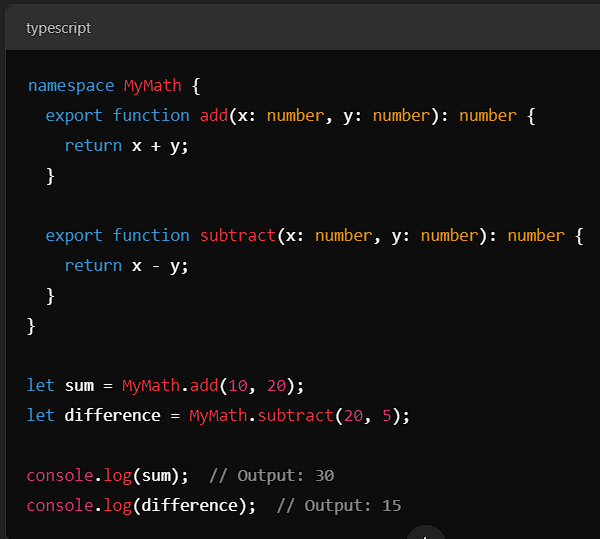
**utility types** are built-in types provided to facilitate common type transformations. They allow you to create new types based on existing ones—making your code more flexible, readable, and DRY (Don’t Repeat Yourself).



**Namespaces in TypeScript**

A **Namespace** is a way to group related code — **variables**, **functions**, **interfaces**, **classes**, etc. — **under a single name** to avoid name conflicts.

They were especially useful before ES modules were standard. However, in modern TypeScript applications, it's generally better to use the import and export syntax for modularity and maintainability.



**Utility types**

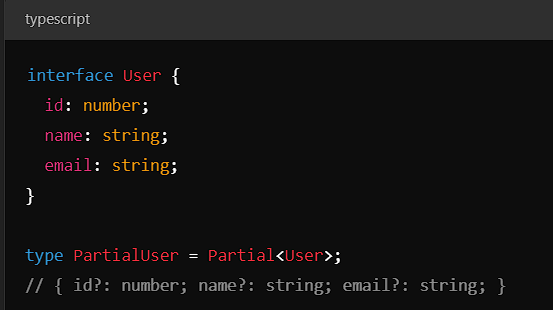
Utility types in TypeScript are built-in generic types that help with common type transformations and manipulations. These types are very useful for creating new types based on existing types. Some of the most popular utility types are:

**Omit Utility in TypeScript**

**Omit** is a utility type in TypeScript that helps to remove specific properties from an existing type. The syntax is **Omit<Type, Keys>**, where Type is the original type and Keys is the property or properties you want to exclude.

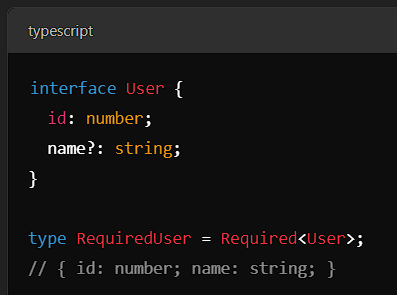
### **. Partial<Type>**

Creates a new type where all properties of Type are optional.



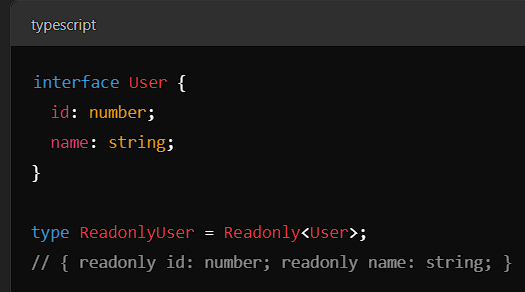
### **Required<Type>**

Creates a type where all properties of Type are required (non-optional).



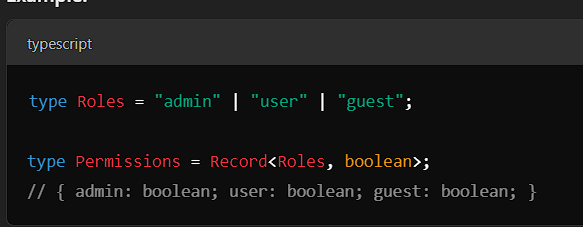
### **Readonly<Type>**

Creates a type where all properties of Type are read-only.



### **Record<Keys, Type>**

Creates an object type with keys of Keys and values of Type.



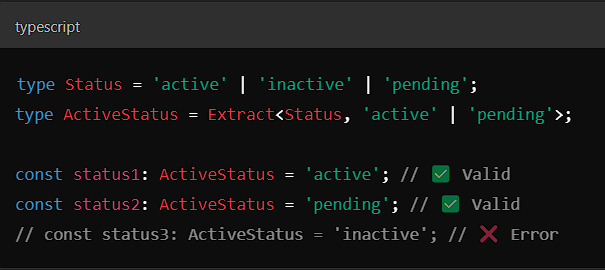
**Pick<Type, Keys> in TypeScript**

The **Pick** utility type in TypeScript helps create a new type by selecting specific properties from an existing type. This is useful when you want to work with a subset of an existing type.



**Extract<T, U> in TypeScript**

The **Extract<T, U>** utility type in TypeScript creates a new type by selecting only the types in **T** that are assignable to **U**. It is often used for filtering types in unions or interfaces.



**Type Annotation**

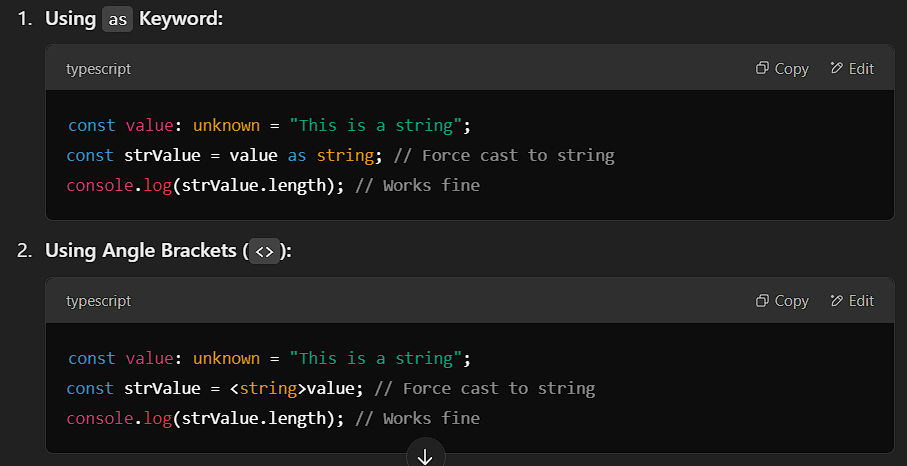
You **explicitly tell TypeScript** the type of a variable, function, or parameter.

**Type Inference**

TypeScript **automatically infers** the type from the value assigned.

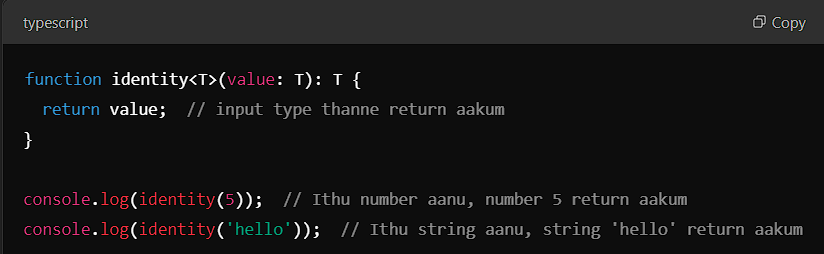
### **Force Casting in TypeScript**

Force casting is a way to explicitly tell the TypeScript compiler to treat a value as a specific type, even if the compiler cannot automatically verify it. This is typically done using **type assertions**. While it can be powerful, you should use force casting carefully, as it bypasses TypeScript's type-checking system and may lead to runtime errors if used incorrectly.



**Generics**

Generics are a feature in programming that allow you to write flexible and reusable code by using placeholders for types. Instead of defining specific types (like number or string), you use a placeholder (e.g., T) that gets replaced by an actual type when the function or class is used. This makes your code more reusable, as it can handle different types while maintaining type safety.



## OOP: Object-Oriented Programming

OOP is a programming paradigm based on the concept of **"objects"**, which represent real-world entities with **properties** and **behaviors**.  
The 4 key **pillars** of OOP are:

## Encapsulation – **Hiding internal data**

**It wrap data (properties) and methods (functions) inside a class. You hide internal details and expose only what is necessary.** Use private, protected, and public to control access.

## Abstraction – **Hide complex logic**

**Goal:** Show **only essential features** and hide unnecessary details.  
Achieved using **abstract classes** or **interfaces**.

## Inheritance – **Reusing code**

**One class (child) inherits properties and methods from another (parent). Promotes code reuse.**

## Polymorphism – **Many forms**

**A method can behave differently depending on the object that calls it. Mainly used with method overriding and interfaces.**

**Compile-time Polymorphism** (Function Overloading)

Same function name behaves differently based on input types.

**Run-time Polymorphism** (Method Overriding)

Subclasses override the behaviour of base class methods.

Types of Inheritance:

|  |
| --- |
| **Single** |

|  |  |
| --- | --- |
| One class inherits from one base class | |
| **Multilevel** |

|  |  |
| --- | --- |
| A class inherits from a class that itself inherits another class | |
| **Hierarchical** |

|  |  |
| --- | --- |
| Multiple classes inherit from a single parent class | |
| **Multiple** |

|  |  |
| --- | --- |
| One class inherits from multiple classes | |
| **Hybrid** |

|  |
| --- |
| Combination of more than one type of inheritance |