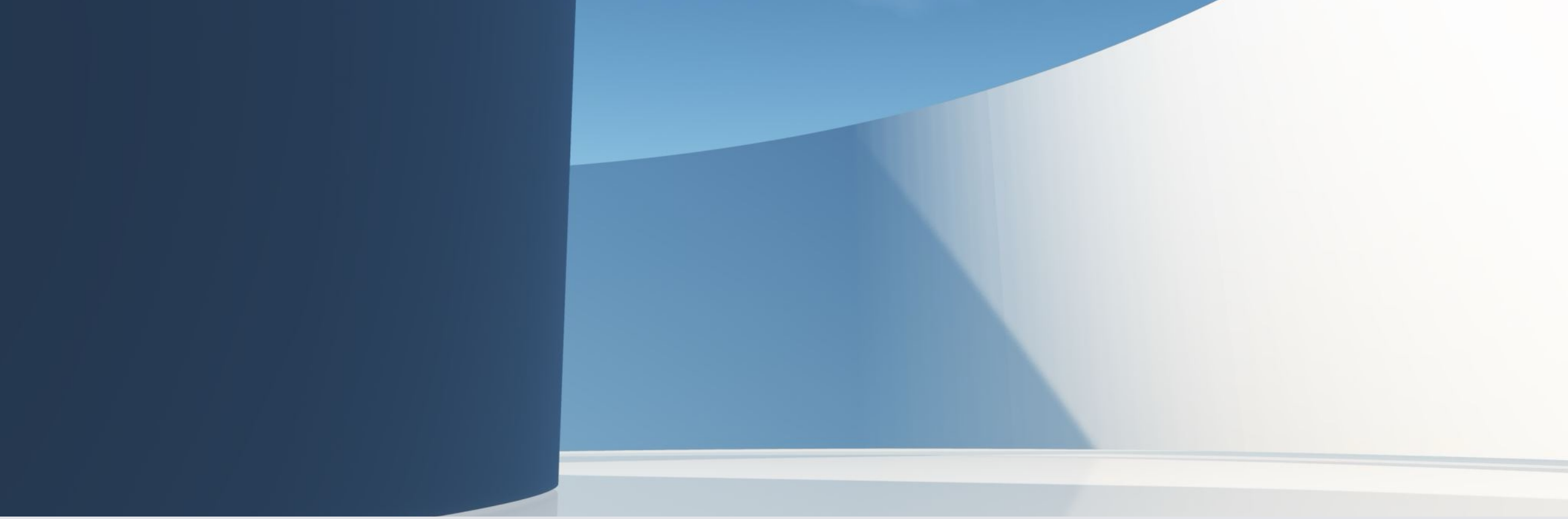


Typescript Generics & Type Transformations

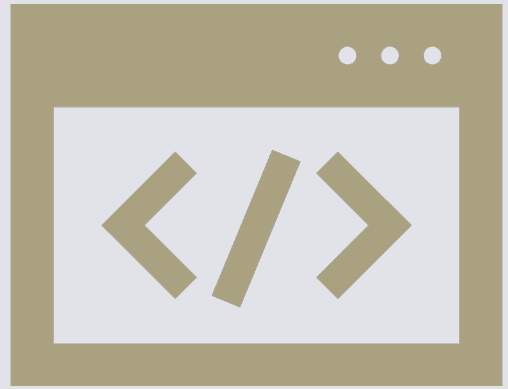
JS Sessions

Ankara - TEKMER

2 Aralık 2023

Typescript

- TypeScript is a strongly typed programming language that builds on JavaScript, giving you better tooling at any scale.
- Variables and function parameters in TypeScript can have explicit types declared
- TypeScript can infer types where they are not explicitly declared.
- TypeScript checks that the values assigned to variables or passed to functions match their declared types during compilation.



Nominal Type System (Java – C++ – Swift)

- Nominal typing is a concept in programming languages where the compatibility and equivalence of data types are determined by their names and explicit declarations.
- For instance, in a language with nominal typing, two classes with identical fields and methods are considered different types.
- Creates runtime type-safety (will discuss more later)

```
// Pseudo code: nominal system
class Foo { method(input: string) { /* ... */ } }
class Bar { method(input: string) { /* ... */ } }

let foo: Foo = new Bar(); // Error!
```

Structural Type System (TS – Go – PHP)

- Focuses on the shape that values have. For example, two types are considered the same if they have the same shape (properties and methods).
- TypeScript's structural type system was designed based on how JavaScript code is typically written.
- Allows for more flexible and often more reusable code.

```ts
interface Pet {
  name: string;
}

class Dog {
  name: string;
}

let pet: Pet;
// OK, because of structural typing
pet = new Dog();
```

Why do we need types?

- Static typing allows catching errors at compile time rather than at runtime. This is particularly beneficial for catching type-related errors and is a significant advantage when working with complex or large-scale codebases.
- The static type system provided by TypeScript enables better tooling support
- TypeScript's type system makes the code more predictable and easier to debug.
- In team environments, TypeScript's type annotations and compile-time checks can make it easier for developers to understand and work on each other's code

KÜMELER

KAZANIM:Bu konu 6. sınıf konusu olup bir kümeyi modelleri ile belirler, farklı temsil biçimleri ile gösterir.

Set Theory

- We can define a set either by enumerating the elements it contains:

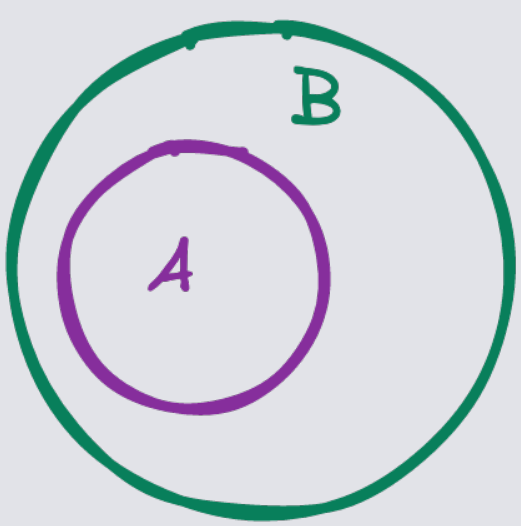
$$\{ a, b, c \}$$

- Or by stating a rule to determine which elements belong in the set:

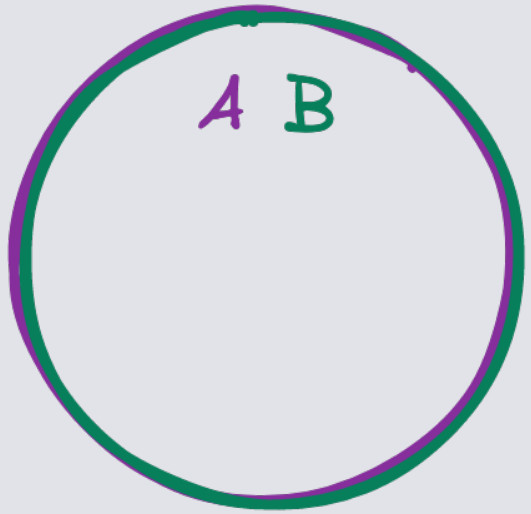
$$\{ x \in S \mid P(x) \}$$

- We can also describe relation between types, e.g., subset – superset

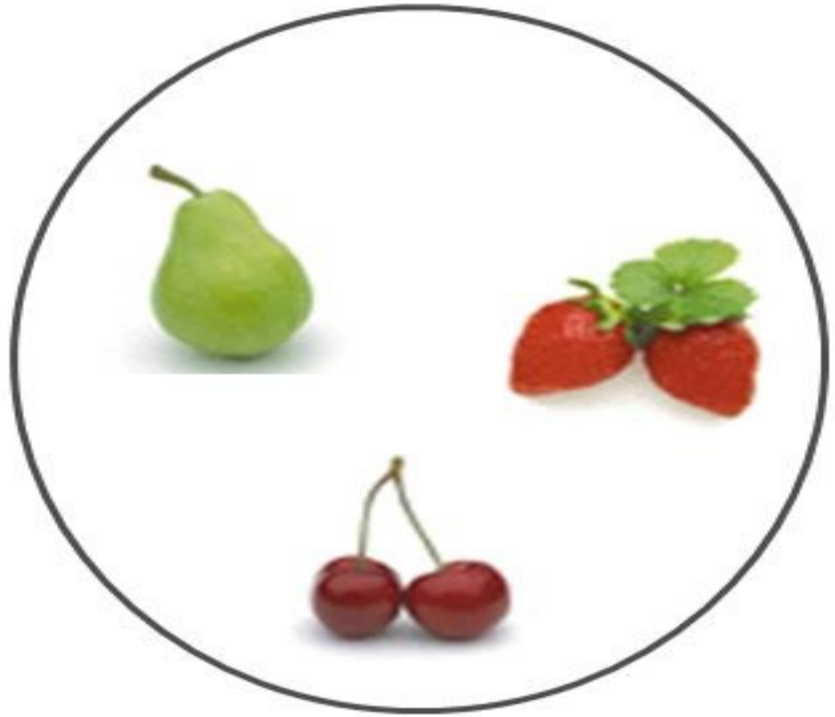
$$\{ \text{a, b} \} \supset \{ a, b, c \}$$



(proper) subset



equality

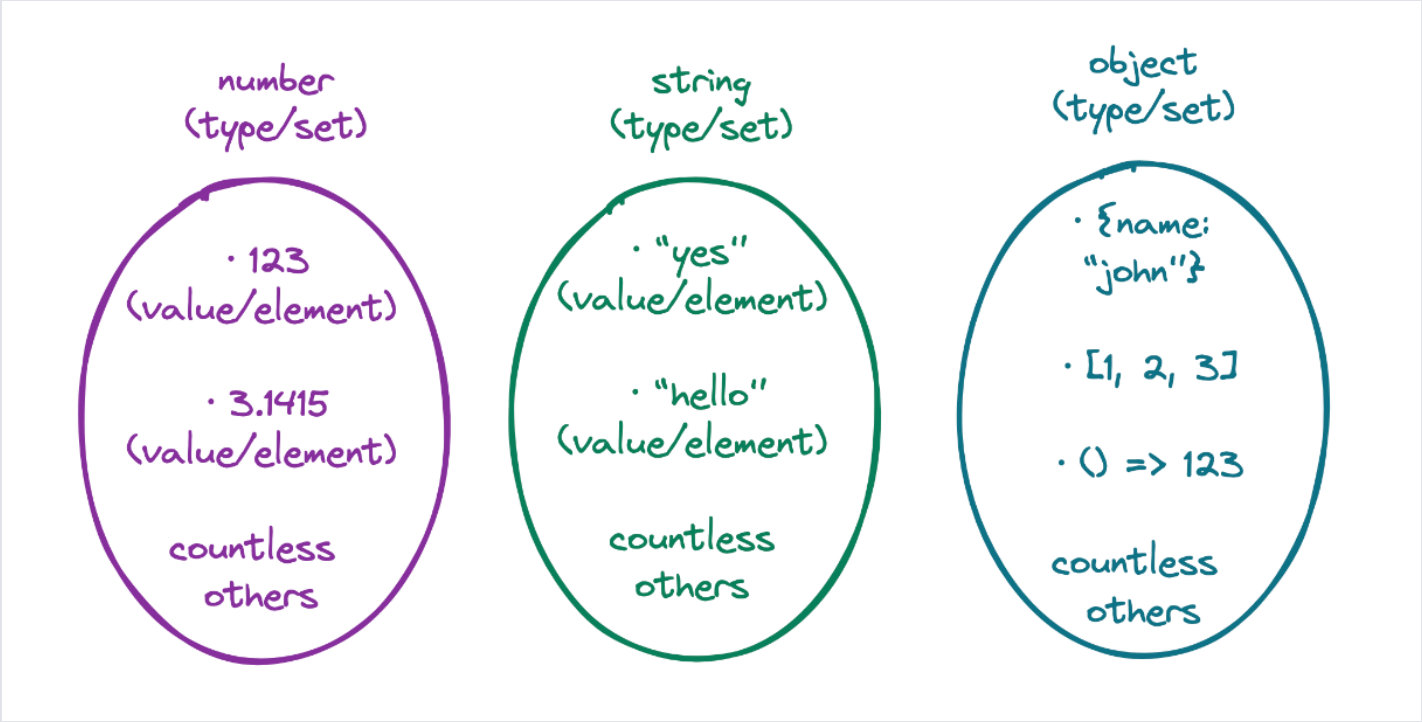


Elemanları; armut,çilek ve vişneden oluşan

“MEYVELER KÜMESİ”

Types as sets

- Set theory offers a mental model for reasoning about types in TypeScript.
- Through the lens of set theory, we can view a type as a set of possible values, i.e. every value of a type can be thought of as an element in a set.
- Note: not all types-as-sets are infinite.



01_types-as-sets.ts

Special types-> never – unknown – any

- **unknown** is a type to which all variables can be assigned, representing a value that could be anything, but requires type checking before usage.
- **never** is a type used for values that never occur, such as the return type of functions that always throw an error or never complete their execution.
- **any** essentially turns off TypeScript's type checking. When you declare a variable as any, you can assign anything to it and access any properties or methods on it without causing compilation errors.

02_special-types.ts

03_never.ts

`extends` keyword

`extends` in a conditional type is TypeScript's equivalent for:

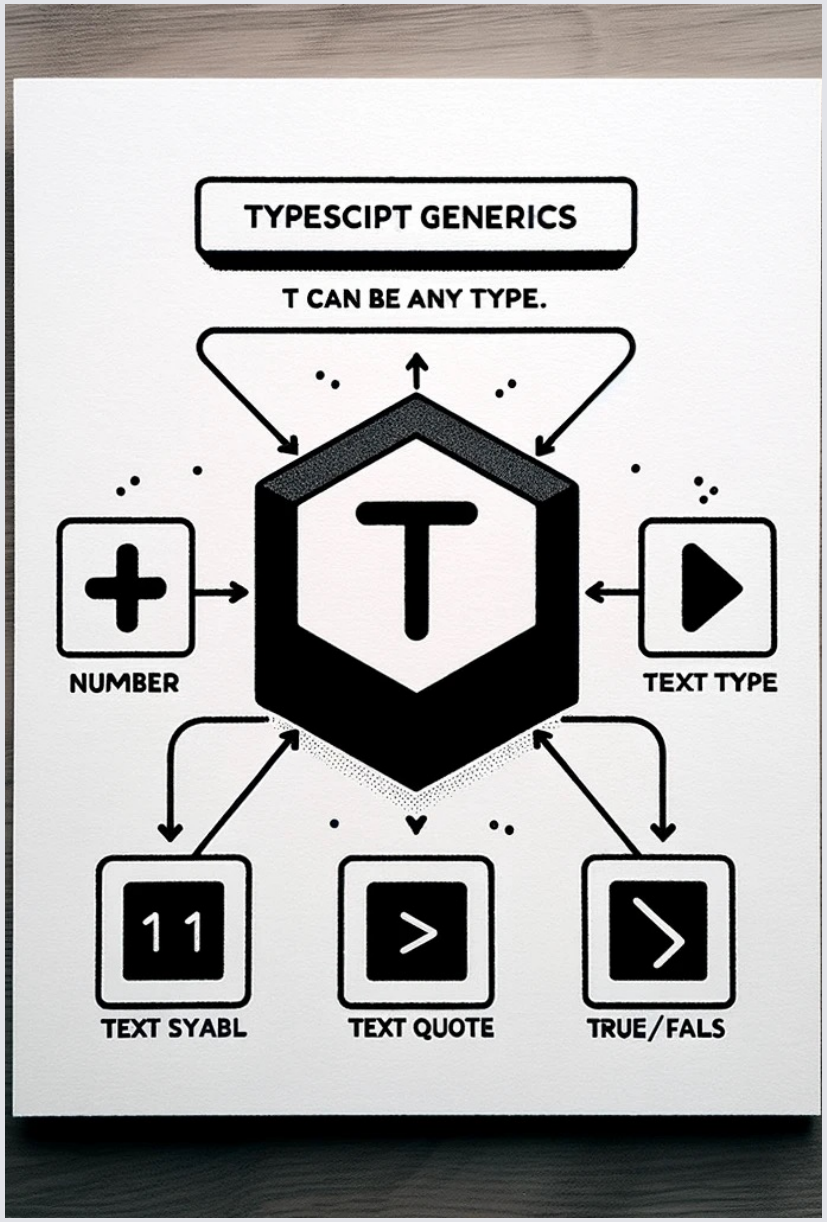
- proper subset, i.e. every element of A is in B, and B has extra elements.
- subset, i.e. every element of A is in B, and B has no extra elements.

```ts
// both true, string literal ⊂ string
type W = 'a' extends string ? true : false;
type X = 'a' | 'b' extends string ? true : false;

// true, string literal ⊆ same string literal
type Y = 'a' extends 'a' ? true : false;

// true, string ⊆ string
type Z = string extends string ? true : false;
```

04_extends.ts

Generics

- Generics in TypeScript allow you to write reusable, flexible code components that work with a variety of types instead of a single type. When you use generics, you create components that can work over a range of types rather than having to duplicate code for each type.
- This approach ensures that your code can be more abstract and capable of handling different types without losing the benefits of type checking.



05_generics-start.ts

Distributive Conditional Types

- A distributive conditional type is a feature that applies a conditional type to each member of a union type individually.
- When you have a conditional type that is checked against a union type, TypeScript distributes the conditional over each member of the union.
- If T is a union type, say A | B | C, the conditional type is applied to each member of the union as if you wrote:

```
`T extends U ? X : Y` is equal to
`(A extends U ? X : Y) | (B extends U ? X : Y) | (C extends U ? X : Y)`
```

08_distributive-conditional-types.ts

Type Transformations

- Type transformations in TypeScript are operations that allow developers to modify or create new types from existing ones.
- This powerful feature of the TypeScript type system enables you to create complex and reusable type definitions
  - keyof / typeof
  - indexed access
  - template literals
  - infer keyword

09_type-transformations-start.ts

keyof operator

- The keyof operator in TypeScript is used to produce a string or numeric literal union of the known, public property names of a type.
- When applied to an interface or type, keyof will return a type that represents all the keys of that interface or type as a union

10_mapped-types-keyof.ts

typeof operator

- typeof operator is used in type contexts to query the type of a variable or property.
- It's different from the JavaScript typeof operator, which is used in runtime expressions to determine the type of a JavaScript value.
- TypeScript's typeof is used at compile time to extract the type information that the TypeScript compiler has about a variable

11_typeof.ts

Indexed accesses

- Indexed access types, also known as lookup types, in TypeScript allow you to access the type of a property within another type..
- This is similar to how you might access the value of a property from an object at runtime, but with types at compile time.
- Indexed access types are particularly useful when you need to reuse the type of a property from an object type.

12_indexed-access.ts

Template literals

- These types use the same syntax as template literals in JavaScript, but are used in type positions to construct new string types by concatenating other types
- These types use the same syntax as template literals in JavaScript but are used in type positions to construct new string types by concatenating other types.

13_template-literal.ts

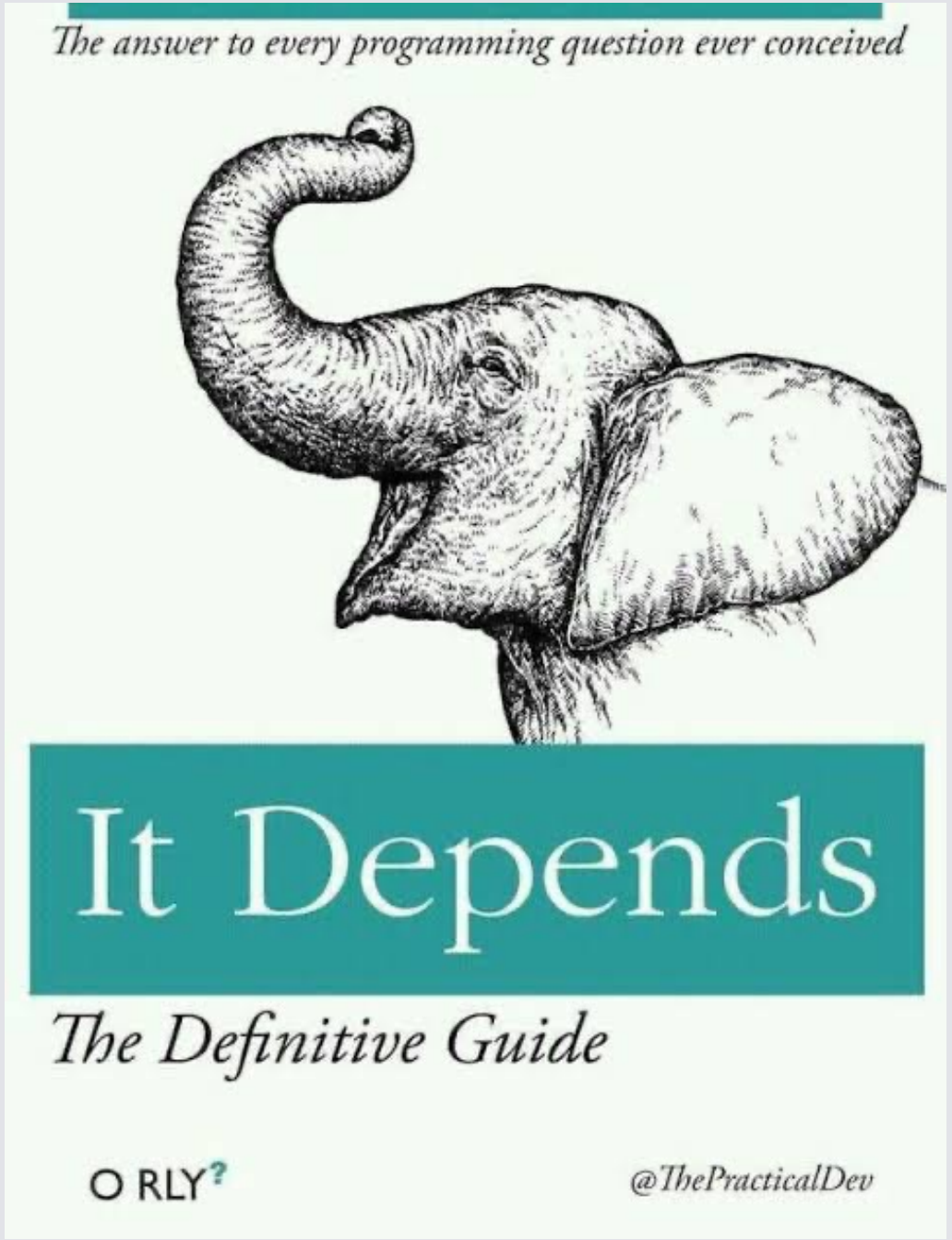
infer keyword

- The infer keyword compliments conditional types and cannot be used outside an extends clause. Infer allows us to define a variable within our constraint to be referenced or returned.
- The infer keyword is a powerful tool that allows us to unwrap and store types while working with third-party TypeScript code. In this article, we explained various aspects of writing robust conditional types using the never keyword, extends keyword, unions, and function signatures.

14_infer.ts

Q & A

oguz@kazkayasi.dev



Sources

- https://ivov.dev/notes/typescript-and-set-theory
- https://www.typescriptlang.org/docs/