Go Generics

Stop Copy-Pasting Your Code

```
func Index[T comparable](s []T, x T)
```

Part 3: Go Basics Series

1. The "Copy-Paste" Problem

Before Generics, if you wanted to add up numbers for Integers AND Floats, you had to write the function twice.

```
// Bad! Logic is duplicated just for different types.
func SumInts(m map[string]int64) int64 { ... }

func SumFloats(m map[string]float64) float64 { ... }
```

This is boring and error-prone. The other option was using `interface{}`, but that is unsafe and slow.

2. The Solution: Type Parameters

Go 1.18 introduced Generics. Think of it like a variable, but for **Types** instead of values.

We use `[T any]` as a placeholder. When you run the code, Go fills in "T" with the real type (like `int` or `string`).

```
// T is a placeholder. It can be ANY type.
func PrintSlice[T any](s []T) {
    for _, v := range s {
        fmt.Println(v)
    }
}
```

3. The "Comparable" Rule

You cannot use == on just "any" type (e.g., you can't compare two functions). So, the compiler stops you.

If you need to check equality, you must use the built-in **comparable** constraint.

```
func Index[T comparable](s []T, x T) int {
    for i, v := range s {
        if v == x { // This ONLY works because of 'comparable'
            return i
        }
    }
    return -1
}
```

4. Defining Your Own Rules

What if you want to use math (+)? "Any" type doesn't support addition (you can't add strings to bools!).

You can create a custom constraint (Interface) to list exactly which types are allowed.

```
// 1. Create the rule: "Only Ints or Floats allowed"
type Number interface {
    int64 | float64
}

// 2. Apply the rule. Now we can use + safely!
func Add[T Number](a, b T) T {
    return a + b
}
```

5. Requiring a Method (Stringer)

Sometimes you don't care about the *data* (int/float), you only care about the *behavior* (methods).

Here, we accept **any** type, as long as it has a `.String()` method.

```
// The Contract: "You must have this method"
type Stringer interface {
    String() string
}

// The Function: Accepts User, Product, or Order types!
func PrintMe[T Stringer](item T) {
    fmt.Println(item.String())
}
```

6. Generic Data Structures

You can create Data Structures that hold any type. Perfect for Lists, Stacks, and Queues.

```
type List[T any] struct {
    val  T
    next *List[T]
}

func main() {
    // A list of Integers
    head := List[int]{val: 1, next: &List[int]{val: 2}}

    // A list of Strings
    words := List[string]{val: "Go"}
}
```

7. Summary

- **Generics** allow us to write logic once and use it for many types.
- We use **Type Parameters** `[T any]` as placeholders.
- **Constraints** act as gatekeepers. They tell the compiler what operations (like `==` or `+`) are allowed.
- **Interfaces** are the secret sauce. You can use them to define Constraints (like `Stringer` or `Number`).