

Phase 1: Introduction to Golang (Sessions 1-8)

Objective: Build a strong foundation in Golang basics.

Session 8: Introduction to Interfaces



Discussion Points

- Understanding interfaces in Go
- Implementing interfaces
- Type assertions and type switches
- Embedding interfaces





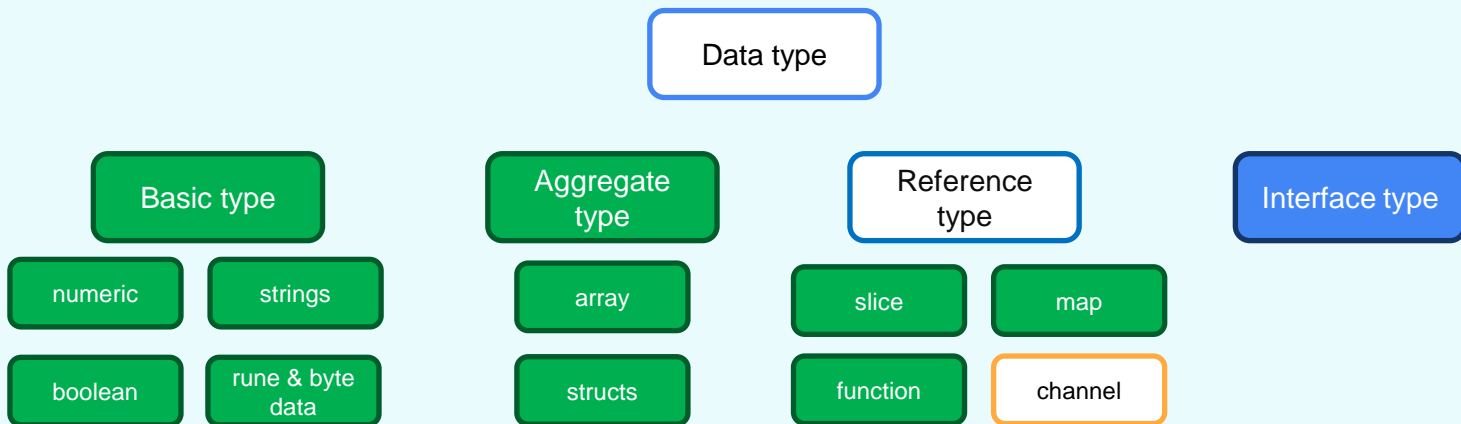
Say to additionally

- Structs Quiz
- Builder pattern using Structs
- Maps using Structs





What is next with Data Types?





Understanding interfaces in Go

What is Interface?

An interface in programming defines a **contract** outlining a set of methods that a *class or type* must implement. It acts as a blueprint, specifying what actions a class *should perform without detailing how*. Interfaces enable a common communication protocol between different types or classes, fostering **abstraction** and modularity in code.



[Gopherfest 2015 | Go Proverbs with Rob Pike \(youtube.com\)](#)



Understanding interfaces in Go

What is Interface?

Interfaces play a crucial role in programming languages, and they serve several important purposes:

- Abstraction
- Polymorphism
- Multiple Inheritance
- Code Reusability
- Testing and Mocking
- API Design





Understanding interfaces in Go

Declaring an Interface?

In Go, interfaces are key to achieving polymorphism and code flexibility. Unlike languages like Java or C#, Go's interfaces are implicit—meaning types implement interfaces simply by defining the required methods, with no explicit "implements" declaration needed. If a type defines the methods that an interface specifies, it automatically implements that interface, reflecting the "**duck typing**" philosophy: *if it behaves like a duck, it's a duck*.

An interface in Go is a set of **method signatures** that a type can implement. It defines the expected behavior of an object rather than declaring its structure. Defining an interface in Go is straightforward, using the type keyword followed by interface and listing the method signatures:

```
type Caller interface {  
    Call(s string)  
    Stop() (bool, error)  
}
```





Understanding interfaces in Go

Naming conventions of Interface?

In Go, interface naming conventions are straightforward and follow effective Go code style guidelines. For interfaces **with a single method**, it is recommended to use the method name combined with the **-er** suffix.

For example, an interface with a *Write* method is named **Writer**, and an interface with a *Read* method is named **Reader**. This convention enhances code readability and aligns with Go's emphasis on simplicity.

Interface names should be concise and clearly indicate their purpose. Avoid overly long names—keep them simple but descriptive enough to convey their intent.

Single-Method Interfaces

```
type Writer interface {  
    Write([]byte) (int, error)  
}  
  
type Reader interface {  
    Read([]byte) (int, error)  
}
```

Multi-Method Interfaces

```
type Shape interface {  
    Area() float64  
    Perimeter() float64  
}  
  
type Caller interface {  
    Call(s string)  
    Stop() (bool, error)  
}
```





Understanding interfaces in Go

Interfaces as Types?

Since an interface is a type like a struct, you can create variables of its type. However, unlike basic data types like integers or strings, interfaces in Go cannot be directly compared using the equality operator (`==`). Instead, comparing interfaces involves checking whether both values are nil or if they have identical underlying types and values.

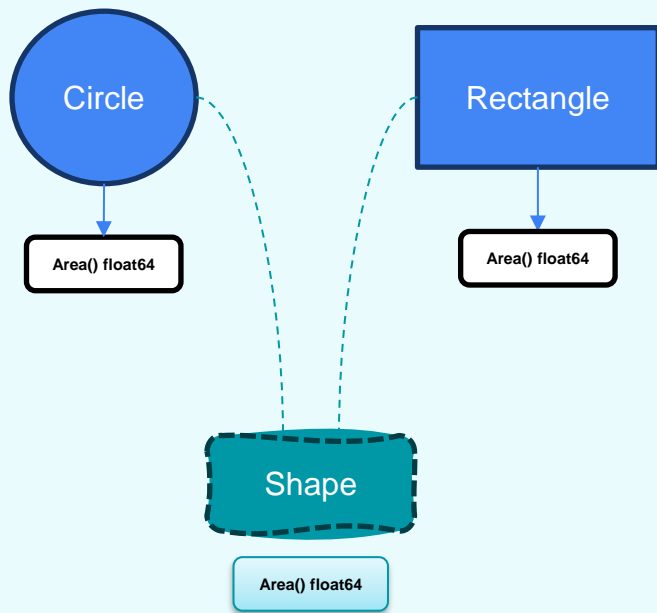
```
type Shape interface {  
    Area() float32  
}  
  
func main() {  
    var s Shape  
    fmt.Println(s)           // nil  
    fmt.Println(s == nil)    // true  
    fmt.Printf("Type is: %T \n", s) // Type is: <nil>  
}
```



Implementing Interfaces

Implementing Interfaces?

In Go, implementing interfaces allows you to define a **contract for behavior**, which multiple types can fulfill by providing their own method implementations. This enables polymorphism, meaning that different types can be treated uniformly if they implement the same interface. This approach fosters code flexibility and reusability by decoupling behavior from specific implementations.





Implementing Interfaces

Empty Interfaces?

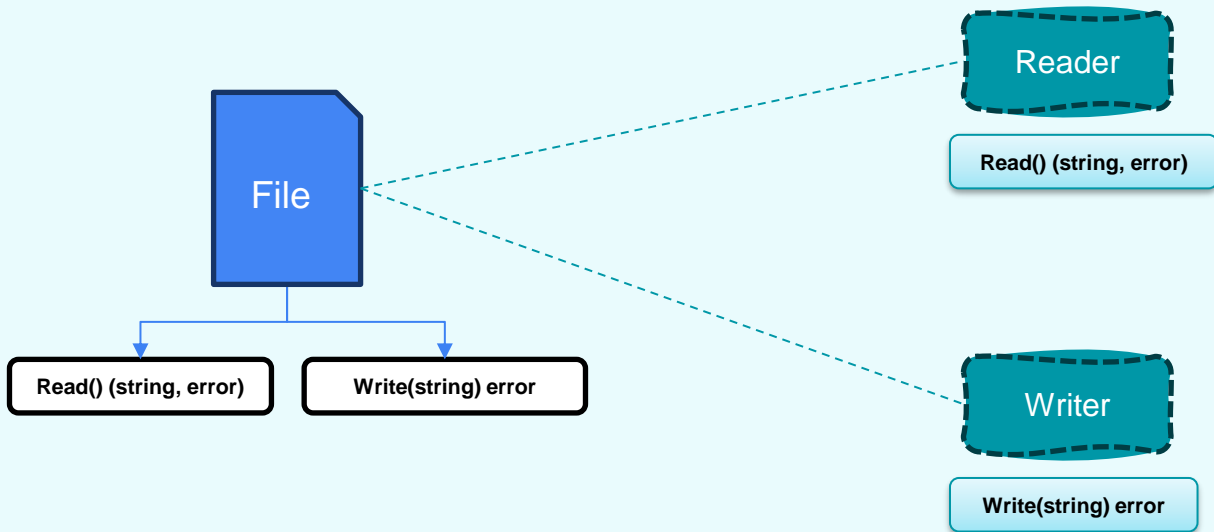
An interface with *zero methods* is called an **empty interface**, represented as **interface{}**.

Since it has no methods, all types implicitly implement it. Starting with Go 1.18, the predeclared type **any** is an alias for the empty interface, meaning **interface{}** can be replaced by **any** for better readability and simplicity in code.

```
func main() {  
    // An empty interface can hold values of any data type.  
    var emptyInterface1 interface{}  
    var emptyInterface2 any  
  
    emptyInterface1 = 1000  
    fmt.Println(emptyInterface1)  
  
    emptyInterface1 = struct{ Name string }{Name: "Ali"}  
    fmt.Println(emptyInterface1)  
  
    emptyInterface2 = "Hi, Gophers"  
    fmt.Println(emptyInterface2)  
  
    emptyInterface2 = map[string]string{"AZ": "Azerbaijan", "TR": "Turkey"}  
    fmt.Println(emptyInterface2)  
  
    emptyInterface2 = nil  
    fmt.Println(emptyInterface2)  
}
```



Multiple Interfaces?





Type assertions and type switches

What is Type assertions?

To determine the underlying dynamic value of an interface in Go, you use the syntax *i.(Type)*, where

- **i** is an interface variable and
- **Type** is the expected concrete type that implements the interface.

Go checks if the dynamic type of *i* matches *Type* and returns the value if possible. This type assertion is commonly used with interface values, especially when dealing with the empty interface (`interface{}`) or when you need to access the specific methods or fields of a concrete type from an interface.

Here's the basic syntax of a type assertion in Go:

```
value, ok := i.(Type)
```





Type assertions and type switches

What is Type assertions?

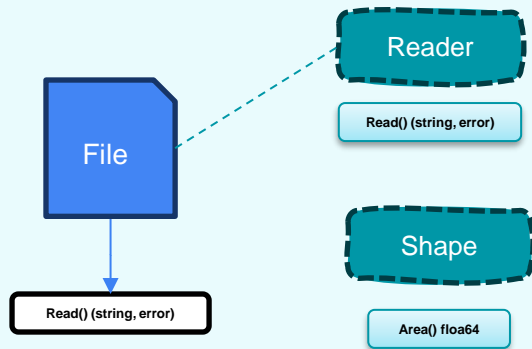
```
type Shape interface {
    Area() float64
}

type Reader interface {
    Read() (string, error)
}

type File struct {
    Name string
}

func (f File) Read() (string, error) {
    return fmt.Sprintf("Content of %s", f.Name), nil
}

func main() {
    var r Reader = File{"session-7.pdf"}
    value1, ok1 := r.(Reader)
    fmt.Printf("dynamic value of File 'r' with value %v implements interface Reader? %v\n", value1, ok1)
    value2, ok2 := r.(Shape)
    fmt.Printf("dynamic value of File 'r' with value %v implements interface Shape? %v\n", value2, ok2)
}
```





Type assertions and type switches

What is Type switch?

A type switch in Go is a **control structure** used to perform actions based on the type of an interface value. It functions similarly to a regular switch statement but is specifically designed to identify and handle different types of an interface value.

Type switches are particularly useful for working with interface values, including empty interfaces (interface{}), when you need to manage different underlying types. The syntax for a type switch:

```
type File struct {
    Name string
}

func FileGenerate(content any) {
    switch v := content.(type) {
    case File:
        fmt.Println("It is a File struct:", v)
    case int:
        fmt.Println("It is an int:", v)
    case string:
        fmt.Println("It is a string:", v)
    default:
        fmt.Println("Unknown type")
    }
}

func main() {
    content1 := "Beginning of my file"
    content2 := File{"myContent.docx"}
    content3 := 15000

    FileGenerate(content1)
    FileGenerate(content2)
    FileGenerate(content3)
}
```





Embedding interfaces

What is Embedding interfaces?

In Go, an interface cannot implement or extend other interfaces directly. However, you can create a new interface by embedding multiple interfaces into one.

This effectively merges the methods of the embedded interfaces, creating a new interface that combines their behaviors

```
type Reader interface {  
    Read() (string, error)  
}  
  
type Writer interface {  
    Write(s string) error  
}  
  
type ReadWriter interface {  
    Reader  
    Writer  
}
```





Interfaces conclusion?

Don't overuse interface{}

Don't design with interfaces, discover them. —Rob Pike

We should create an interface when we need it, not when we foresee that we could need it.



Session Quiz Go?



Join at
slido.com
3167614

Thank you.

Questions?

- Understanding interfaces in Go
- Implementing interfaces
- Type assertions and type switches
- Embedding interfaces

“Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.”

- Albert Einstein

