Lab3

Structs, Methods & Interfaces

Introduction

A **struct** in Go lets you group related data together. You can use it to make your own types. **Methods** are functions you can attach to these types to give them actions. **Interfaces** let you describe what a type can do, so different types can be used in the same way.Often, you'll use structs, methods, and interfaces together for powerful and flexible design.

In this lab, you'll build a **Geometry Calculator** where you'll learn how structs organize data, how methods add behavior, and how interfaces enable flexible, reusable code.

What You'll Learn

Throughout this lab, you'll progressively build a geometry calculator:

Stage	Topic	What You'll Do
1	How Structs Work	Understand struct syntax, field access, and type safety
2	How Methods Work	Learn method syntax, receivers, and type-specific behavior
3	How Interfaces Work	Understand implicit implementation and polymorphism
4	Building Shapes	Implement Rectangle, Circle, and Triangle with Area methods

Learning Approach: Each stage includes:

- A **concept section** with explanations and diagrams
- **Syntax examples** to understand the patterns
- **Challenge sections** with pseudocode to try first

- **Complete code** at the end for reference

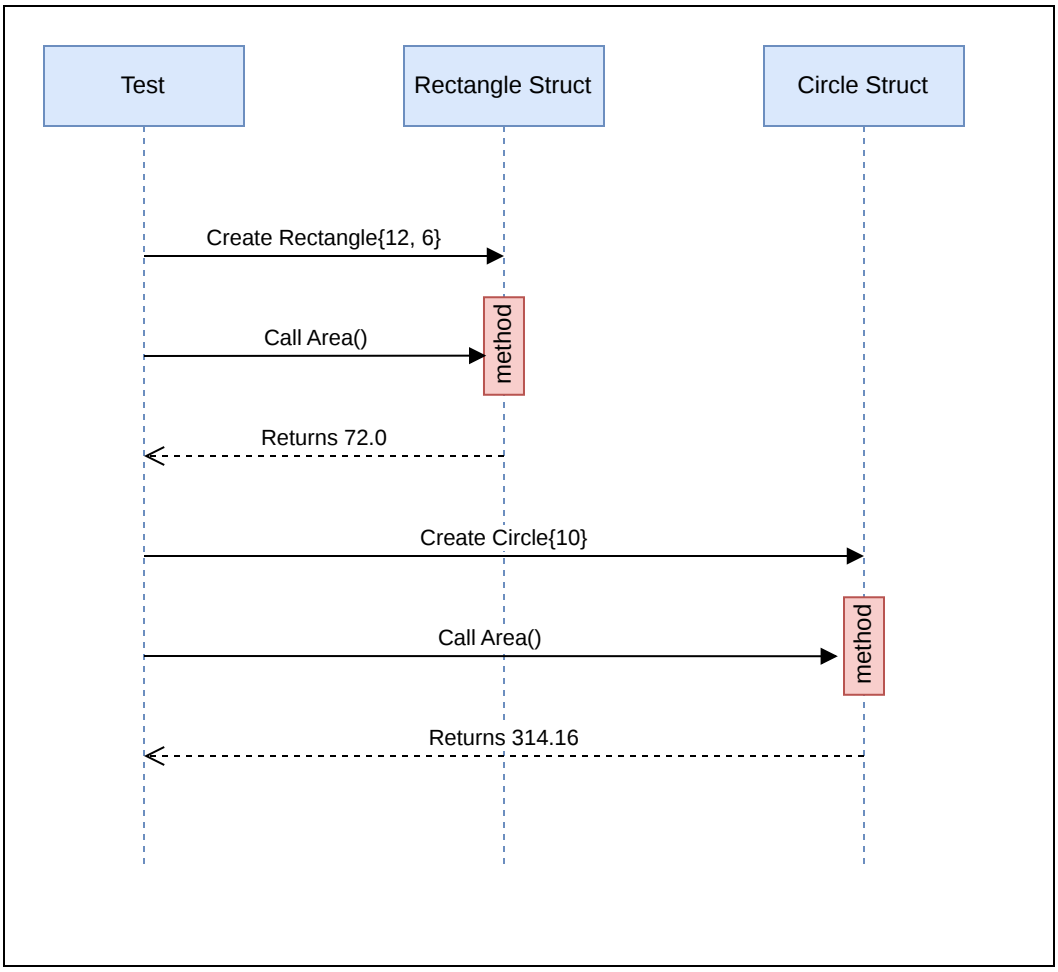
Project Setup

```
mkdir shapes
cd shapes
go mod init github.com/yourusername/shapes
```

Stage 1: How Structs Work

Before writing code, let's understand the key concepts.

1.1 What is a Struct?



A struct in Go is a composite data type that groups together variables (called fields) under a single name. Each field has a name and a type. Structs let you model real-world entities by combining related data.

For example, a `Rectangle` has a width and height. Instead of passing two separate variables everywhere, you can group them into one struct:

```
type Rectanglestruct{
   Widthfloat64
   Heightfloat64
}
```

The line `type Rectangle struct` creates a brand new data type called `Rectangle`. Inside the curly braces `{}`, you specify its fields. Every `Rectangle` will have a `Width` and

a `Height`, both of which are `float64` numbers.

1.2 Creating Struct Instances

There are several ways to create a struct instance:

```
// Method 1: Field order (not recommended)
rect1:= Rectangle{10.0,5.0}

// Method 2: Named fields (recommended)
rect2:= Rectangle{Width:10.0, Height:5.0}

// Method 3: Zero value then assign
var rect3 Rectangle
rect3.Width=10.0
rect3.Height=5.0

// Method 4: Using new (returns pointer)
rect4:=new(Rectangle)
rect4.Width=10.0
rect4.Height=5.0
```

1.3 Accessing Struct Fields

Access struct fields using dot notation:

```
rect:= Rectangle{Width:12.0, Height:6.0}

// Read fields
rect.Width// → 12
rect.Height// → 6

// Modify fields
rect.Width=20.0
```

```
// Use fields in calculations
area:= rect.Width* rect.Height
```

1.4 Structs as Function Parameters

You can pass structs to functions:

```
// Function that takes a Rectangle
funcCalculateArea(r Rectangle)float64{
return r.Width* r.Height
}

funcCalculatePerimeter(r Rectangle)float64{
return2*(r.Width+ r.Height)
}

// Usage
rect:= Rectangle{Width:10.0, Height:5.0}
area:=CalculateArea(rect)// → 50.0
perimeter:=CalculatePerimeter(rect)// → 30.0
```

1.5 Struct Copy Behavior (Value Type)

Structs are **value types** - copying a struct creates an independent copy:

```
original:= Rectangle{Width:10.0, Height:5.0}

// Copy the struct (creates independent copy)
copy:= original

// Modify the copy
copy.Width=100.0
```

```
// Original is unchanged!
original.Width// → 10 (unchanged)
copy.Width// → 100
```

This is different from maps (which are reference types). Each struct copy is independent.

Learning Outcomes :

- Structs group related fields into a single type
- Create structs with `Type{Field: value}` syntax
- Access fields with dot notation: `struct.Field`
- Structs are value types - copies are independent
- Use named fields for clarity

Stage 2: How Methods Work

Methods let us attach functions to a specific type.

2.1 What is a Method?

A method is a function with a special **receiver** argument. The receiver connects the method to a type, allowing you to call it using dot notation:

```
// Function (standalone)
funcCalculateArea(r Rectangle)float64{
return r.Width* r.Height
}
area:=CalculateArea(rect)// Called as function

// Method (attached to type)
func(r Rectangle)Area()float64{
return r.Width* r.Height
```

```
}
area:= rect.Area()// Called on the instance
```

In Go, the syntax `func (r Rectangle) Area()` defines a method called `Area` that is attached to the `Rectangle` type. Here, `r` is the receiver variable, which works similarly to `this` or `self` in other programming languages. It represents the specific instance the method is called on. By convention, the receiver variable is usually a short, lowercase version of the type name (for example, `r` for Rectangle, `c` for Circle).

2.2 Defining Methods

```
type Rectanglestruct{
    Widthfloat64
    Heightfloat64
}

// Method: Area belongs to Rectangle
func(r Rectangle)Area()float64{
return r.Width* r.Height
}

// Method: Perimeter belongs to Rectangle
func(r Rectangle)Perimeter()float64{
return2*(r.Width+ r.Height)
}

// Usage - call methods with dot notation
rect:= Rectangle{Width:12.0, Height:6.0}
rect.Area()// → 72.0
rect.Perimeter()// → 36.0
```

2.3 Multiple Types, Same Method Name

Unlike functions, methods can have the same name on different types. This is because methods are namespaced to their receiver type:

```
type Rectanglestruct{
    Widthfloat64
    Heightfloat64
}

type Circlestruct{
    Radiusfloat64
}

// Both types have Area() - no conflict!
func(r Rectangle)Area()float64{
return r.Width* r.Height
}

func(c Circle)Area()float64{
return math.Pi* c.Radius* c.Radius
}

// Each type uses its own Area method
rect:= Rectangle{Width:12.0, Height:6.0}
circle:= Circle{Radius:10.0}

rect.Area()// → 72.0
circle.Area()// → 314.16
```

This is a key advantage of methods - each type can have its own implementation of a method with the same name.

2.4 Methods vs Functions Comparison

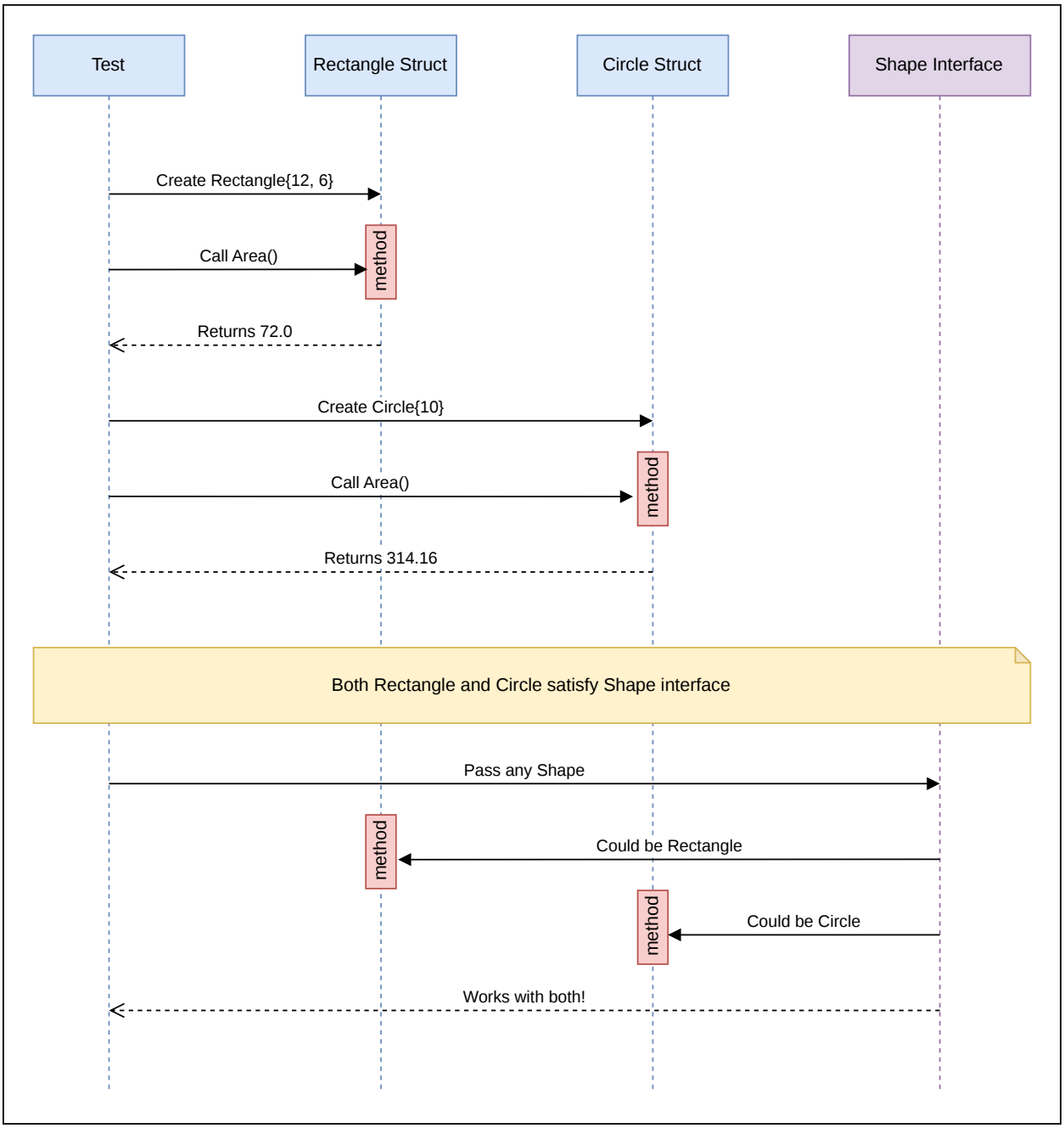
Aspect	Function	Method
Syntax	`func Name(param Type)`	`func (r Type) Name()`
Call	`Name(value)`	`value.Name()`
Naming	Must be unique	Can repeat across types
Purpose	Standalone operations	Behavior attached to types

Learning Outcomes :

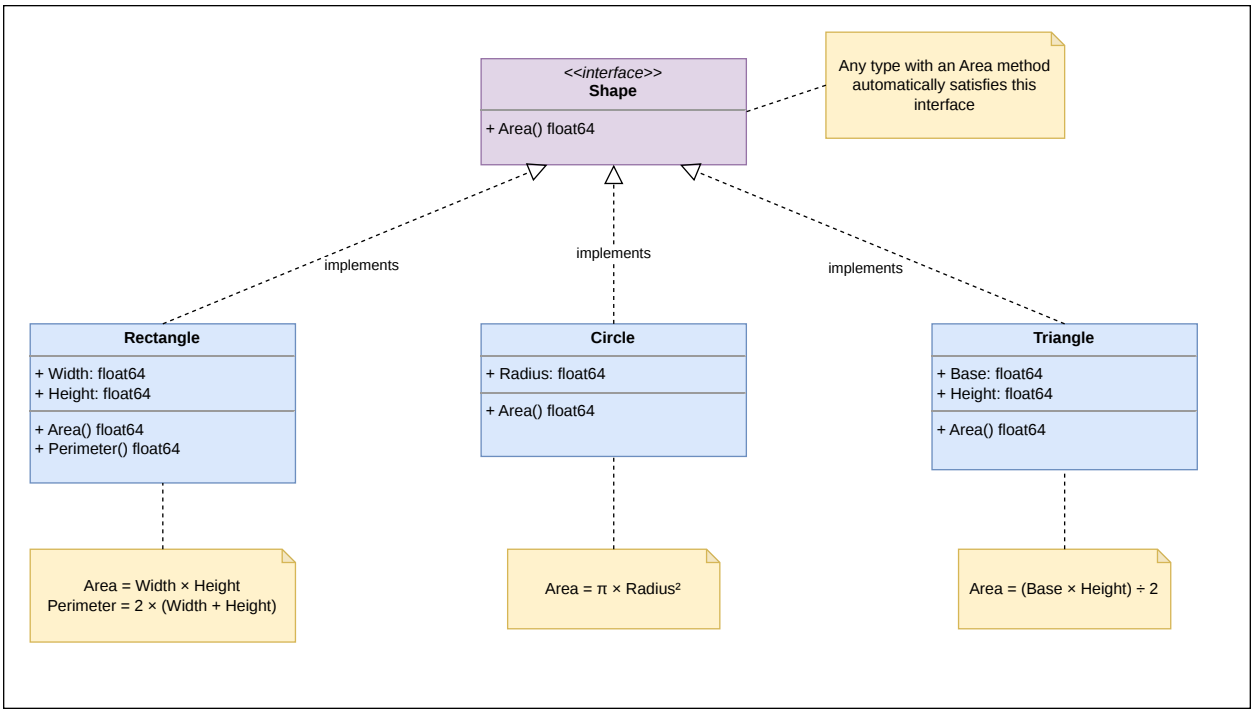
- Methods are functions with a receiver argument
- Syntax: `func (r Type) MethodName() returnType`
- Call methods with dot notation: `instance.Method()`
- Different types can have methods with the same name
- Use receiver's first letter by convention (`r` for Rectangle)

Stage 3: How Interfaces Work

Interfaces are one of Go's most powerful features. They enable polymorphism means writing code that works with multiple types.



3.1 What is an Interface?



An interface defines a set of method signatures. Any type that implements all those methods automatically satisfies the interface meaning no explicit declaration needed.

```
// Interface definition
type Shapeinterface{
Area()float64
}
```

This says: "Any type that has an `Area() float64` method is a Shape."

Both `Rectangle` and `Circle` have `Area() float64` methods, so they both automatically implement `Shape`:

```
type Rectanglestruct{ Width, Heightfloat64}
func(r Rectangle)Area()float64{return r.Width* r.Height}

type Circlestruct{ Radiusfloat64}
```

```
func(c Circle)Area()float64{return math.Pi* c.Radius* c.Radius}

// Both Rectangle and Circle are Shapes!
```

3.2 Using Interfaces

Interfaces let you write functions that work with any type implementing the interface:

```
// Interface: any type with Area() is a Shape
type Shapeinterface{
Area()float64
}

// Types that implement Area() satisfy Shape automatically
type Rectanglestruct{ Width, Heightfloat64}
func(r Rectangle)Area()float64{return r.Width* r.Height}

type Circlestruct{ Radiusfloat64}
func(c Circle)Area()float64{return math.Pi* c.Radius* c.Radius}

// This function works with ANY Shape!
funcPrintArea(s Shape){
   fmt.Printf("Area: %.2f\n", s.Area())
}

// Both work because both are Shapes
PrintArea(Rectangle{12.0,6.0})// → Area: 72.00
PrintArea(Circle{10.0})// → Area: 314.16
```

3.3 Implicit Implementation

Go interfaces are implemented **implicitly**. There's no `implements` keyword like in Java or C#. If a type has the required methods, it **satisfies the interface**

automatically:

```
type Speakerinterface{
Speak()string
}

type Dogstruct{ Namestring}
type Catstruct{ Namestring}

// Dog implements Speaker (implicitly - no keyword needed)
func(d Dog)Speak()string{
return d.Name+" says: Bark!"
}

// Cat implements Speaker (implicitly)
func(c Cat)Speak()string{
return c.Name+" says: Meow!"
}

funcMakeSpeak(s Speaker){
   fmt.Println(s.Speak())
}

// Both work - they have Speak() method
MakeSpeak(Dog{Name:"Tommy"})// → Tommy says: Bark!
MakeSpeak(Cat{Name:"Bilai"})// → Bilai says: Meow!
```

3.4 Interfaces with Multiple Methods

Interfaces can require multiple methods:

```
type Shapeinterface{
Area()float64
```

```
Perimeter()float64
}
```

A type must implement **all** methods to satisfy the interface:

```
type Rectanglestruct{
    Width, Heightfloat64
}

// Rectangle implements both methods - it's a Shape
func(r Rectangle)Area()float64{
return r.Width* r.Height
}

func(r Rectangle)Perimeter()float64{
return2*(r.Width+ r.Height)
}
```

3.5 The Empty Interface

The empty interface `interface{}` has no methods, so every type satisfies it:

```
funcPrintAnything(vinterface{}){
    fmt.Printf("Value: %v, Type: %T\n", v, v)
}

// Works with any type!
PrintAnything(42)// → Value: 42, Type: int
PrintAnything("hello")// → Value: hello, Type: string
PrintAnything(Rectangle{10,5})// → Value: {10 5}, Type: main.Rectangle
```

Learning Outcomes :

- Interfaces define method signatures that types must implement
- Implementation is implicit - no `implements` keyword
- If a type has the methods, it satisfies the interface
- Interfaces enable polymorphism - one function, many types
- The empty interface `interface{}` accepts any type

Stage 4: Building the Geometry Calculator

Now let's apply these concepts by building a complete geometry calculator.

4.0 Challenge Yourself First!

Before looking at the solutions, copy this pseudocode and try to implement it yourself:

```
package main

import(
"fmt"
"math"
)

// TODO: Define Shape interface
// - Method: Area() float64

// TODO: Define Rectangle struct
// - Fields: Width (float64), Height (float64)

// TODO: Define Circle struct
// - Fields: Radius (float64)

// TODO: Define Triangle struct
// - Fields: Base (float64), Height (float64)
```

```
// TODO: Implement Area() for Rectangle
// - Formula: Width * Height

// TODO: Implement Area() for Circle
// - Formula: math.Pi * Radius * Radius

// TODO: Implement Area() for Triangle
// - Formula: (Base * Height) / 2

// TODO: Implement PrintShapeInfo function
// - Input: shape (Shape)
// - Output: prints the area

funcmain(){
// Test your implementation here!
}
```

Give it a try! Once you've attempted it, the complete implementation is explained below.

4.1 Defining the Shape Interface

First, define the contract that all shapes must follow:

```
type Shapeinterface{
Area()float64
}
```

4.2 Implementing Rectangle

```
type Rectanglestruct{
    Widthfloat64
```

```
    Heightfloat64
}

func(r Rectangle)Area()float64{
return r.Width* r.Height
}
```

Try it out:

```
funcmain(){
    rect:= Rectangle{Width:12.0, Height:6.0}
    fmt.Printf("Rectangle: %+v\n", rect)
    fmt.Printf("Area: %.2f\n", rect.Area())
}
```

Output:

```
Rectangle: {Width:12 Height:6}
Area: 72.00
```

4.3 Implementing Circle

```
type Circlestruct{
    Radiusfloat64
}

func(c Circle)Area()float64{
return math.Pi* c.Radius* c.Radius
}
```

Try it out:

```
funcmain(){
    circle:= Circle{Radius:10.0}
    fmt.Printf("Circle: %+v\n", circle)
    fmt.Printf("Area: %.2f\n", circle.Area())
}
```

Output:

```
Circle: {Radius:10}
Area: 314.16
```

4.4 Implementing Triangle

```
type Trianglestruct{
    Basefloat64
    Heightfloat64
}

func(t Triangle)Area()float64{
return(t.Base* t.Height)/2
}
```

Try it out:

```
funcmain(){
    triangle:= Triangle{Base:12.0, Height:6.0}
    fmt.Printf("Triangle: %+v\n", triangle)
    fmt.Printf("Area: %.2f\n", triangle.Area())
}
```

Output:

```
Triangle: {Base:12 Height:6}
Area: 36.00
```

4.5 Using Polymorphism

Now we can write code that works with any shape:

```
funcPrintShapeInfo(s Shape){
   fmt.Printf("Area: %.2f\n", s.Area())
}

funcmain(){
   shapes:=[]Shape{
      Rectangle{Width:12, Height:6},
      Circle{Radius:10},
      Triangle{Base:12, Height:6},
}

for_, shape:=range shapes{
PrintShapeInfo(shape)
}
}
```

Output:

```
Area: 72.00
Area: 314.16
Area: 36.00
```

4.6 Complete Geometry Calculator Code

Here's the full implementation:

```
package main

import(
"fmt"
"math"
)

// Shape interface
type Shapeinterface{
Area()float64
}

// Rectangle type
type Rectanglestruct{
    Widthfloat64
    Heightfloat64
}

func(r Rectangle)Area()float64{
return r.Width* r.Height
}

// Circle type
type Circlestruct{
    Radiusfloat64
}

func(c Circle)Area()float64{
return math.Pi* c.Radius* c.Radius
}
```

```
// Triangle type
type Trianglestruct{
    Basefloat64
    Heightfloat64
}

func(t Triangle)Area()float64{
return(t.Base* t.Height)/2
}

// PrintShapeInfo works with any Shape
funcPrintShapeInfo(namestring, s Shape){
    fmt.Printf("%s Area: %.2f\n", name, s.Area())
}

funcmain(){
    fmt.Println("=== GEOMETRY CALCULATOR ===\n")

// Create shapes
    rect:= Rectangle{Width:12.0, Height:6.0}
    circle:= Circle{Radius:10.0}
    triangle:= Triangle{Base:12.0, Height:6.0}

// Print individual shapes
    fmt.Println("1. Individual Shapes:")
PrintShapeInfo("Rectangle", rect)
PrintShapeInfo("Circle", circle)
PrintShapeInfo("Triangle", triangle)

// Use slice of shapes
    fmt.Println("\n2. All Shapes' Area in a Sum:")
    shapes:=[]Shape{rect, circle, triangle}

    totalArea:=0.0
for_, shape:=range shapes{
        totalArea+= shape.Area()
```

```
}
    fmt.Printf("Total Area: %.2f\n", totalArea)
}
```

Output:

```
=== GEOMETRY CALCULATOR ===

1. Individual Shapes:
Rectangle Area: 72.00
Circle Area: 314.16
Triangle Area: 36.00

2. All Shapes' Area in a Sum:
Total Area: 422.16
```

Learning Outcomes :

- Define interfaces to establish contracts
- Implement methods on each type to satisfy the interface
- Use interface types in functions for polymorphism
- Store different types in slices using interface types
- Calculate aggregate values across different shape types

Conclusion

In this lab, you learned:

- **Structs** - Creating custom types to group related data with fields
- **Field Access** - Using dot notation to read and modify struct fields
- **Value Semantics** - Structs are copied when assigned (unlike maps)
- **Methods** - Attaching behavior to types using receiver functions

- **Same Method Names** - Different types can have methods with identical names
- **Interfaces** - Defining contracts for polymorphic code
- **Implicit Implementation** - Types satisfy interfaces automatically
- **Polymorphism** - One function working with many types through interfaces

These foundations will support your journey into writing effective and idiomatic Go code.