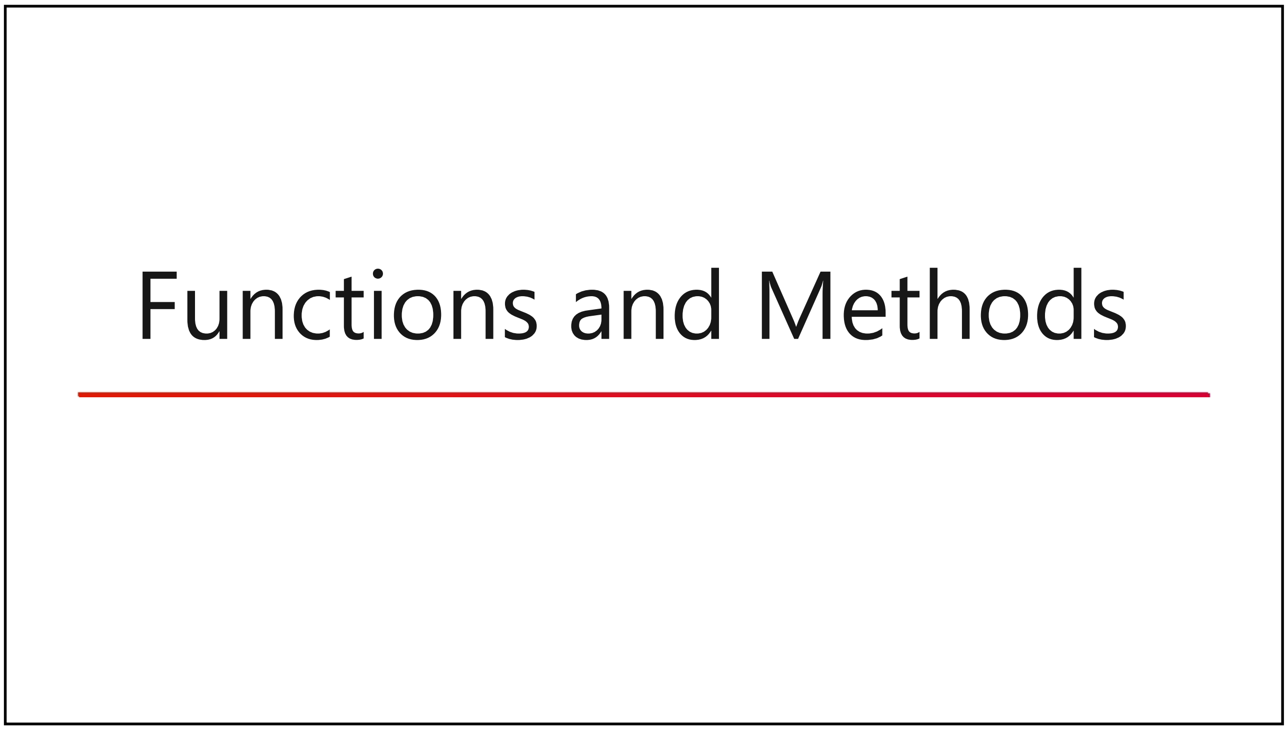
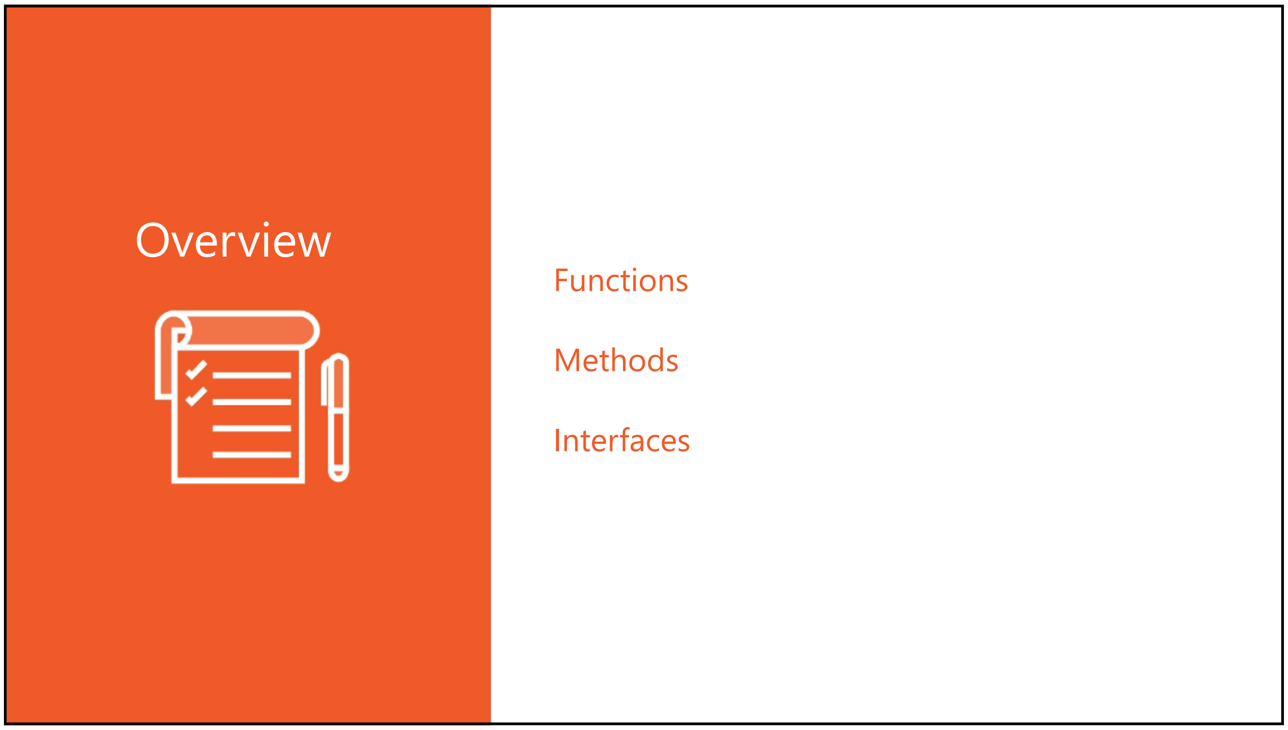


Functions and Methods



Overview

Functions

Methods

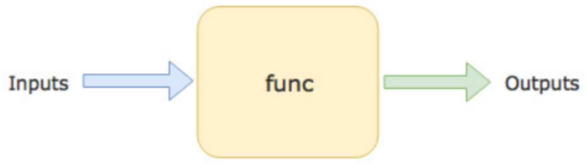
Interfaces

Creating Functions and Methods

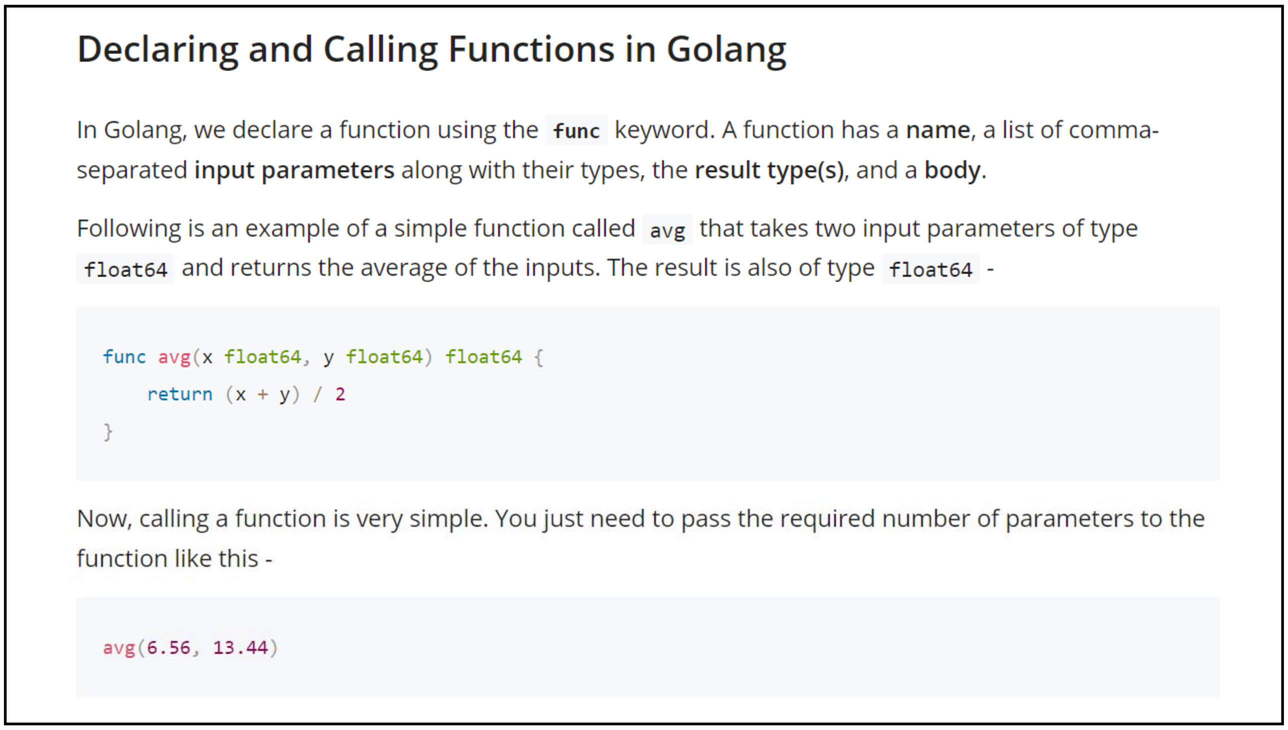
FUNCTIONS

Introduction to Functions

A function is a block of code that takes some input(s), does some processing on the input(s) and produces some output(s).



Functions help you divide your program into small reusable pieces of code. They improve the readability, maintainability, and testability of your program.



Declaring and Calling Functions in Golang

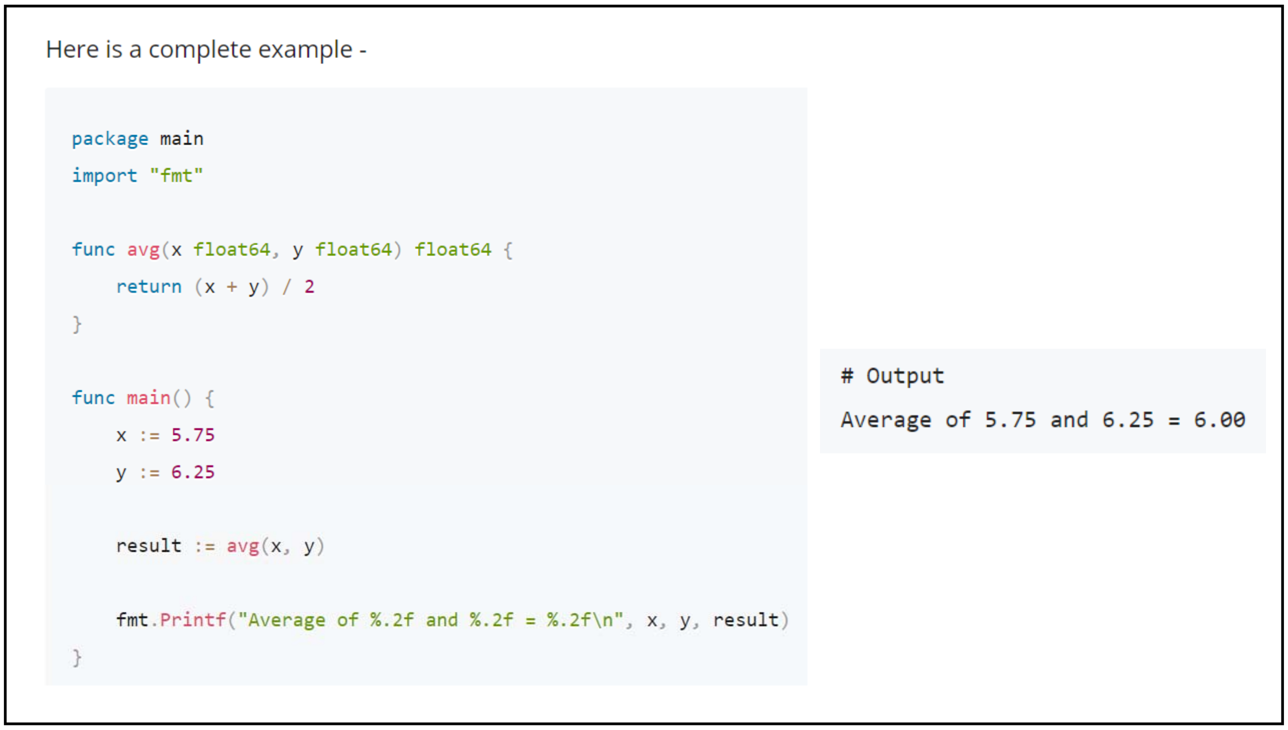
In Golang, we declare a function using the `func` keyword. A function has a **name**, a list of comma-separated **input parameters** along with their types, the **result type(s)**, and a **body**.

Following is an example of a simple function called `avg` that takes two input parameters of type `float64` and returns the average of the inputs. The result is also of type `float64` -

```go
func avg(x float64, y float64) float64 {
    return (x + y) / 2
}
```

Now, calling a function is very simple. You just need to pass the required number of parameters to the function like this -

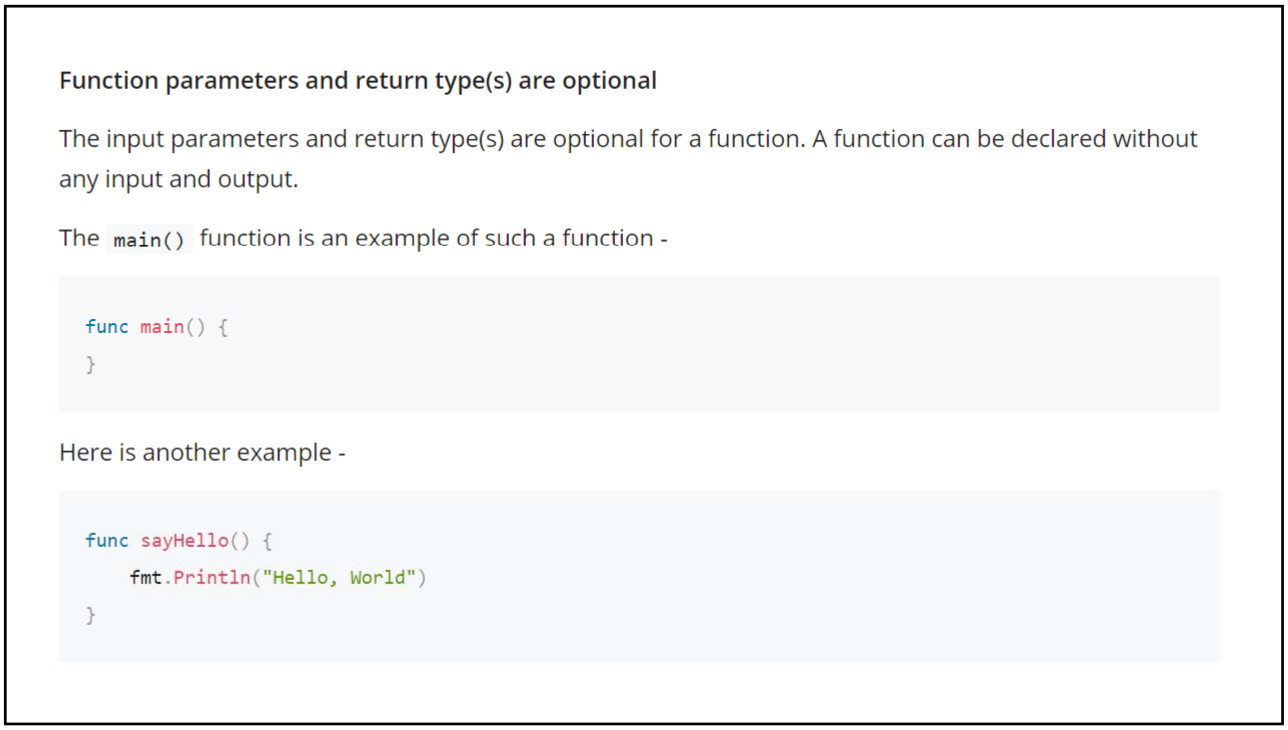
```go
avg(6.56, 13.44)
```



Here is a complete example -

```go
package main
import "fmt"

func avg(x float64, y float64) float64 {
    return (x + y) / 2
}

func main() {
    x := 5.75
    y := 6.25

    result := avg(x, y)

    fmt.Printf("Average of %.2f and %.2f = %.2f\n", x, y, result)
}
```

```
# Output
Average of 5.75 and 6.25 = 6.00
```



Function parameters and return type(s) are optional

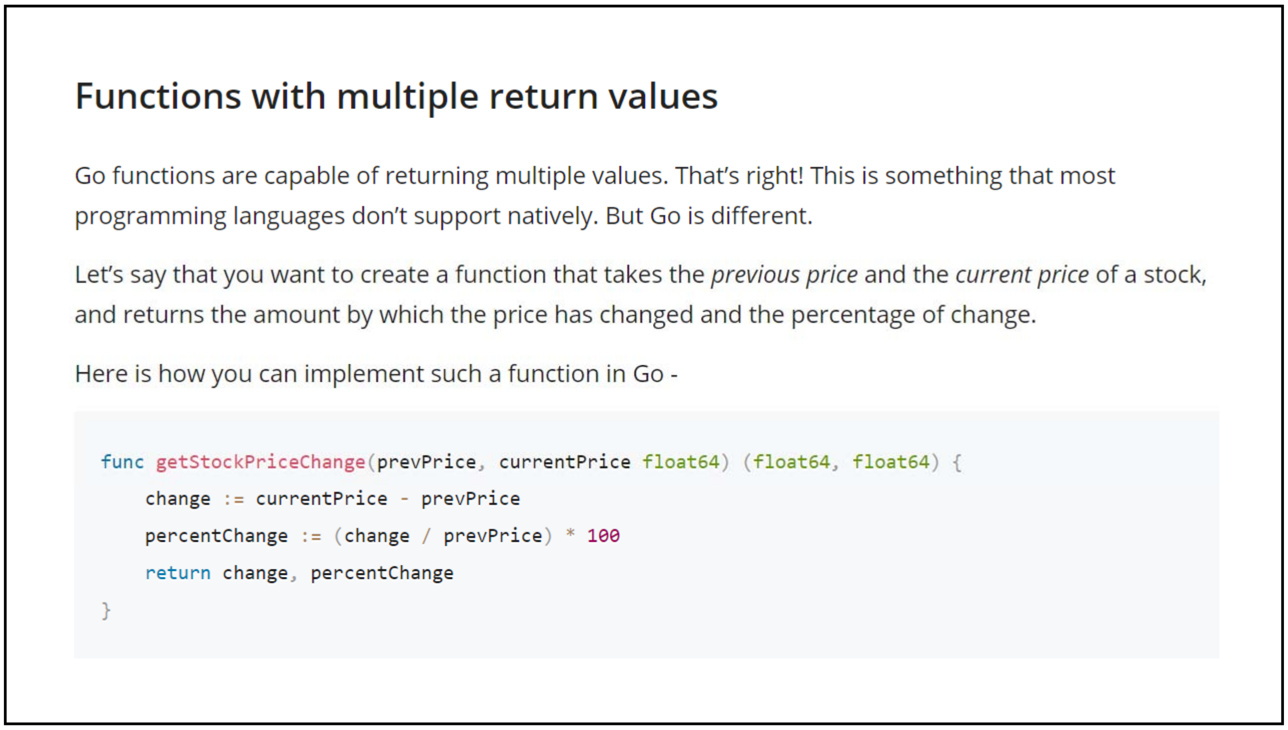
The input parameters and return type(s) are optional for a function. A function can be declared without any input and output.

The `main()` function is an example of such a function -

```go
func main() {
}
```

Here is another example -

```go
func sayHello() {
    fmt.Println("Hello, World")
}
```



Functions with multiple return values

Go functions are capable of returning multiple values. That's right! This is something that most programming languages don't support natively. But Go is different.

Let's say that you want to create a function that takes the *previous price* and the *current price* of a stock, and returns the amount by which the price has changed and the percentage of change.

Here is how you can implement such a function in Go -

```go
func getStockPriceChange(prevPrice, currentPrice float64) (float64, float64) {
    change := currentPrice - prevPrice
    percentChange := (change / prevPrice) * 100
    return change, percentChange
}
```

Functions with named return values

The return values of a function in Golang may be named. Named return values behave as if you defined them at the top of the function.

Let’s rewrite the `getStockPriceChange` function that we saw in the previous section with named return values -

```go
// Function with named return values
func getNamedStockPriceChange(prevPrice, currentPrice float64) (change, percentChange float64) {
    change = currentPrice - prevPrice
    percentChange = (change / prevPrice) * 100
    return change, percentChange
}
```

Named return values allow you to use the so-called **Naked return** (a `return` statement without any argument). When you specify a `return` statement without any argument, it returns the named return values by default. So you can write the above function like this as well -

```go
// Function with named return values and naked return
func getNamedStockPriceChange(prevPrice, currentPrice float64) (change, percentChange float64) {
    change = currentPrice - prevPrice
    percentChange = (change / prevPrice) * 100
    return
}
```

Creating Functions and Methods

METHODS

Introduction to Methods

Technically, Go is not an object-oriented programming language. It doesn't have classes, objects, and inheritance.

However, Go has types. And, you can define **methods** on types. This allows for an object-oriented style of programming in Go.

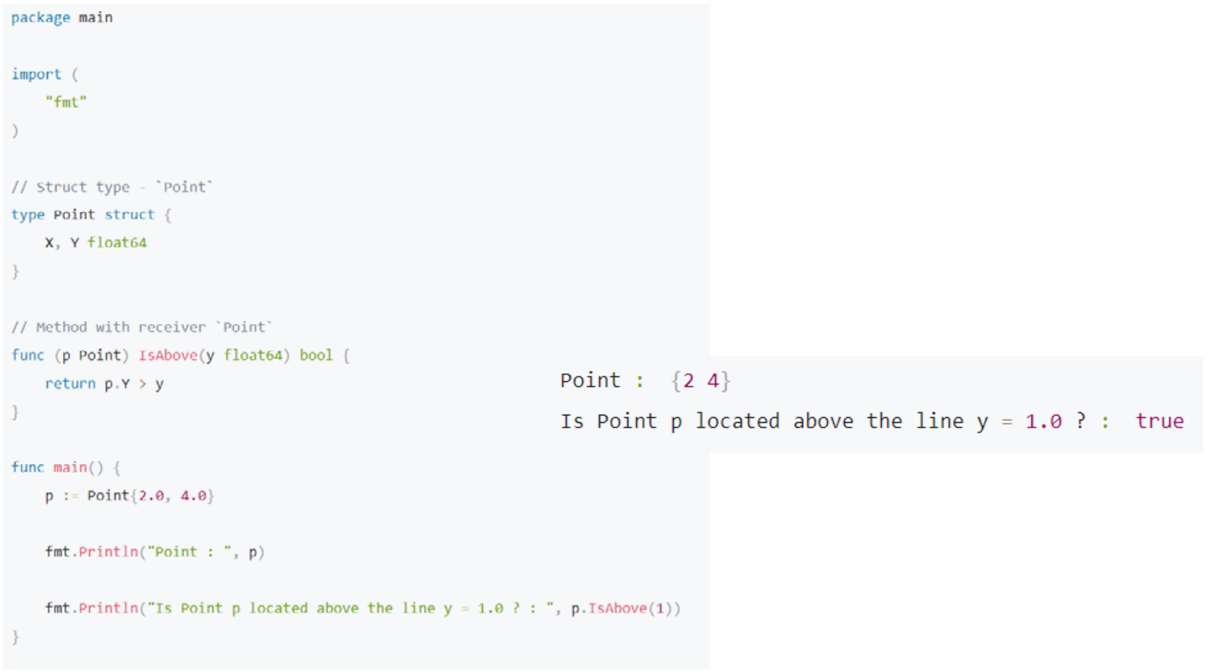
Go Methods

A method is nothing but a function with a special *receiver* argument.

The *receiver* argument has a name and a type. It appears between the `func` keyword and the method name -

```
func (receiver Type) MethodName(parameterList) (returnTypes) {
}
```

The receiver can be either a struct type or a non-struct type.



```
package main

import (
    "fmt"
)

// Struct type - `Point`
type Point struct {
    X, Y float64
}

// Method with receiver `Point`
func (p Point) IsAbove(y float64) bool {
    return p.Y > y
}

func main() {
    p := Point{2.0, 4.0}

    fmt.Println("Point : ", p)

    fmt.Println("Is Point p located above the line y = 1.0 ? : ", p.IsAbove(1))
}
```

```
Point :  {2 4}
Is Point p located above the line y = 1.0 ? :  true
```

Creating Functions and Methods

INTERFACES

Introduction to Interfaces

An interface in Go is a **type** defined using a set of method signatures. The interface defines the behavior for similar type of objects.

For example, Here is an interface that defines the behavior for Geometrical shapes:

```
// Go Interface - `Shape`
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

An interface is declared using the **type** keyword, followed by the name of the interface and the keyword `interface`. Then, we specify a set of method signatures inside curly braces.

Implementing an interface in Go

To implement an interface, you just need to implement all the methods declared in the interface.

Go Interfaces are implemented implicitly

Unlike other languages like Java, you don't need to *explicitly* specify that a type implements an interface using something like an `implements` keyword. You just implement all the methods declared in the interface and you're done.

Here are two Struct types that implement the `Shape` interface:

```go
// Struct type `Rectangle` - implements the `Shape` interface by implementing all its methods.
type Rectangle struct {
    Length, Width float64
}

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

func (r Rectangle) Perimeter() float64 {
    return 2 * (r.Length + r.Width)
}
```

```go
// Struct type `Circle` - implements the `Shape` interface by implementing all its methods.
type Circle struct {
    Radius float64
}

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

func (c Circle) Perimeter() float64 {
    return 2 * math.Pi * c.Radius
}

func (c Circle) Diameter() float64 {
    return 2 * c.Radius
}
```

Using an interface type with concrete values

An interface in itself is not that useful unless we use it with a concrete type that implements all its methods.

Let's see how an interface can be used with concrete values.

- ***An interface type can hold any value that implements all its methods***
- ***Using Interface types as arguments to functions***
- ***Using Interface types as fields***

- *An interface type can hold any value that implements all its methods*

```go
package main

import (
    "fmt"
)

func main() {
    var s Shape = Circle{5.0}
    fmt.Printf("Shape Type = %T, Shape Value = %v\n", s, s)
    fmt.Printf("Area = %f, Perimeter = %f\n\n", s.Area(), s.Perimeter())

    s = Rectangle{4.0, 6.0}
    fmt.Printf("Shape Type = %T, Shape Value = %v\n", s, s)
    fmt.Printf("Area = %f, Perimeter = %f\n", s.Area(), s.Perimeter())
}
```

```
# Output
Shape Type = main.Circle, Shape Value = {5}
Area = 78.539816, Perimeter = 31.415927

Shape Type = main.Rectangle, Shape Value = {4 6}
Area = 24.000000, Perimeter = 20.000000
```

- *Using Interface types as arguments to functions*

```go
package main

import (
    "fmt"
)

// Generic function to calculate the total area of multiple shapes of different types
func CalculateTotalArea(shapes ...Shape) float64 {
    totalArea := 0.0
    for _, s := range shapes {
        totalArea += s.Area()
    }
    return totalArea
}

func main() {
    totalArea := CalculateTotalArea(Circle{2}, Rectangle{4, 5}, Circle{10})
    fmt.Println("Total area = ", totalArea)
}
```

```
# Output
Total area =  346.7256359733385
```

- *Using Interface types as fields*

```go
package main

import (
    "fmt"
)

// Interface types can also be used as fields
type MyDrawing struct {
    shapes  []Shape
    bgColor string
    fgColor string
}

func (drawing MyDrawing) Area() float64 {
    totalArea := 0.0
    for _, s := range drawing.shapes {
        totalArea += s.Area()
    }
    return totalArea
}
```

```go
func main() {
    drawing := MyDrawing{
        shapes: []Shape{
            Circle{2},
            Rectangle{3, 5},
            Rectangle{4, 7},
        },
        bgColor: "red",
        fgColor: "white",
    }

    fmt.Println("Drawing", drawing)
    fmt.Println("Drawing Area = ", drawing.Area())
}
```

```
# Output
Drawing {[{2} {3 5} {4 7}] red white}
Drawing Area = 55.56637061435917
```

Summary

Functions

Methods

Interfaces