Higher-Order Functions

This lesson discusses how to use functions as parameters and values.

WE'LL COVER THE FOLLOWING

Function used as a value

Functions can be used as values just like any other value in Go. In the following code, `f1` is assigned a value, the function `inc1`:

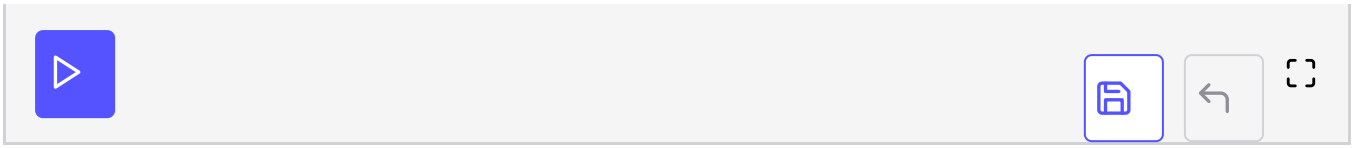
```
func inc1(x int) int { return x+1 }
f1 := inc1(2) // f1 := func (x int) int { return x+1 }
```

Function used as a parameter

Functions can be used as parameters in another function. The passed function can then be called within the body of that function; that is why it is commonly called a *callback*. To illustrate, here is a simple example:

```
package main
import (
    "fmt"
)

func main() {
    callback(1, Add) // function passed as a parameter
}

func Add(a, b int) {
    fmt.Printf("The sum of %d and %d is: %d\n", a, b, a + b)
}

func callback(y int, f func(int, int)) {
    f(y, 2) // this becomes Add(1, 2)
```

```
}
```



Functions as a Parameter

To understand the code, look at **line 10** at the function header of the function `Add`. It takes two parameters `a` and `b` and prints the *sum* of the parameters. In the `main`, we are just calling the `callback` function as: `callback(1, Add)`. Here, we have two parameters: the first is `1` used as *int* and the second is `Add` function passed as a parameter. You can verify it through the header of the `callback` function: `func callback(y int, f func(int, int))` at **line 14**. At **line 15**, we call the function `f` from the header of `callback`, and treat it like the `Add` function. So `y` in `callback` is `a` in `Add`, and `2` is `b` in `Add`.

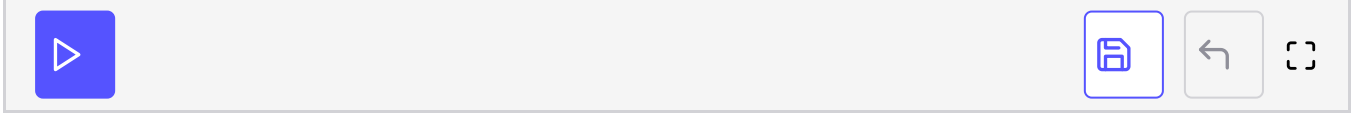
A good example of the use of a function as a parameter is the `strings.IndexFunc()` function. It has the signature:

```
func IndexFunc(s string, f func(c int) bool) int
```

and returns the *index* into `s` of the first Unicode character for which `f(c)` is *true*, or **-1** if none will do.

For example, `strings.IndexFunc(line, unicode.IsSpace)` will return the index of the 1st whitespace character in line.

```
package main
import "fmt"
import "strings"
import "unicode"

func main(){
  s := "Hello! Let's run Go lang"
  //finding index of first space from s string
  fmt.Print(strings.IndexFunc(s, unicode.IsSpace))
}
```

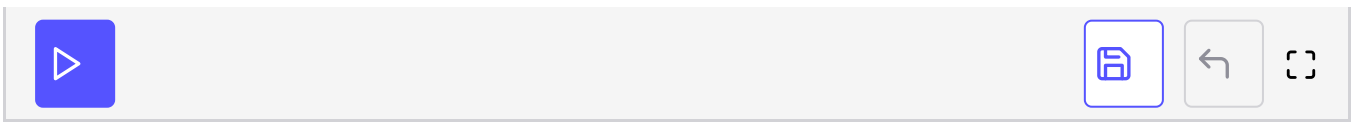


Index of First Whitespace

Function used as a filter

In the following program, we have a *filter* function which takes a *slice* of integers and a function that takes an integer and returns a bool. Slices are a key data type in Go, giving a more powerful interface to sequences than arrays. You'll study them in detail in Chapter 5. For a brief introduction of slices, you can visit A Tour of Go.

```
package main
import "fmt"

type flt func(int) bool
    // isOdd takes an int slice and returns a bool set to true if the
    // int parameter is odd, or false if not.
    // isOdd is of type func(int) bool which is what flt is declared to be.

func isOdd(n int) bool {
        if n % 2 == 0 {
            return false
        }
        return true
    }

    // Same comment for isEven
func isEven(n int) bool {
    if n % 2 == 0 {
        return true
    }
    return false
}

func filter(sl[] int, f flt)[] int {
    var res[] int
    for _, val := range sl {
        if f(val) {
            res = append(res, val)
        }
    }
    return res
}

func main() {
    slice := [] int {1, 2, 3, 4, 5, 7}
    fmt.Println("slice = ", slice)
    odd := filter(slice, isOdd)
    fmt.Println("Odd elements of slice are: ", odd)
    even := filter(slice, isEven)
    fmt.Println("Even elements of slice are: ", even)
}
```



Function Filter

The program has two basic functions. The first function, `isOdd` takes `n` as a parameter and returns a boolean value (see its header at **line 9**). If `n` is odd, it will return **true**. Otherwise, it will return **false**. Similarly, the second function, `isEven` takes `n` as a parameter and returns a boolean value (see its header at **line 17**). If `n` is even, it will return **true**. Otherwise, it will return **false**.

At **line 4**, we are aliasing a type. A function that takes a single *integer* as a parameter and returns a single *boolean* value is given a type `flt`. Now, moving towards a major part of the program: the `filter` function. See its header at **line 24**. It takes a slice of integers (that are to be judged as even or odd) as a first parameter, and function `f` of type `flt` (either `isEven` or `isOdd`). The function `filter` returns a slice of integers `res`, for which the `f` returns **true**.

Let's see the `main` function now. At **line 35**, we declare a slice of integers named `slice`. Then at **line37**, we call the `filter` function with `slice` as the first parameter and `isOdd` as the second parameter and store the result in `odd` slice. Similarly, at **line39**, we call the `filter` function with `slice` as the first parameter and `isEven` as the second parameter and store the result in the `even` slice. Printing `odd` and `even` slices at **line 38** and **line 40**, respectively, verifies the result. The same principle can be applied to construct filters for any type.

This is how you use a function as a value or a parameter. The next lesson brings a challenge for you to solve.