Receivers and Methods as Pointers and Values

In this lesson, you'll study the difference between the value and the pointer type in Go to receive a result or call a method.

WE'LL COVER THE FOLLOWING

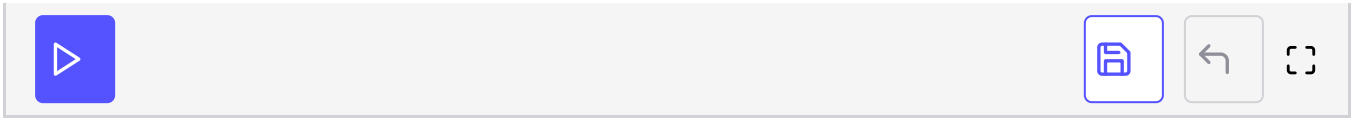
Pointer or value as a receiver

The `recv` is most often a pointer to the *receiver*-type for performance reasons (because we don't make a copy of the value, as would be the case with call by value); this is especially true when the receiver type is a struct. Define the method on a pointer type if you need the method to modify the data the receiver points to. Otherwise, it is often cleaner to define the method on a normal value type.

This is illustrated in the following example :

```
package main
import (
"fmt"
)

type B struct {
  thing int
}

func (b *B) change() { b.thing = 1 }

func (b B) write() string { return fmt.Sprint(b) }

func main() {
  var b1 B // b1 is a value
  b1.change()
  fmt.Println(b1.write())
  b2 := new(B) // b2 is a pointer
```

```
  b2.change()
  fmt.Println(b2.write())
}
```



Pointer and Value

In the above code, at **line 6**, we make a struct of type `B` with one integer field `thing`. Look at the header of `change()` method at **line 10**: `func (b *B) change()`. The part `(b *B)` shows that only the pointer to `B` type object can call this method. This method is changing its internal field `thing` by assigning the value of **1** to it. Now, look at the header of the `write()` method at **line 12**: `func (b B) write() string`. The part `(b B)` shows that only the `B` type object can call this method. This method is returning its internal field `thing` after converting it to type *string*.

> **Note:** The function `Sprint()` from `fmt` package returns the string without printing them on the console.

Now, look at `main`. We make a variable `b1` of type `B` using `var` keyword at **line 15**. In the next line, we call `change()` method on `b1`. After returning from this method, `thing` of `b1` will hold value **1**. At **line 17**, we are printing the result from `write()` method called on `b1`. It will print `thing` from `b1`, which will output **1** on the screen. We make a variable `b2` of type `B` using the `new()` function at **line 18**. In the next line, we call the `change()` method on `b2`. After returning from this method, `thing` of `b2` will hold a value of **1**. At **line 20**, we are printing the result from the `write()` method called on `b2`, which will print `thing` from `b2` that will output **1** on the screen.

Notice, in `main()`, that Go does plumbing work for us; we do not have to figure out whether to call the methods on a pointer or not, Go does that for us. The variable `b1` is a value, and `b2` is a pointer. However, the method calls work just fine. Try to make `write()` change its receiver value `b`. You will see that it compiles fine, but the original `b` is not changed. We see that a method does not require a pointer as a receiver as in the following example, where we only need the values of Point3 to compute something:

```
type Point3 struct { x, y, z float }
```

```
// A method on Point3:

func (p Point3) Abs() float {
  return math.Sqrt(p.x*p.x + p.y*p.y + p.z*p.z)
}
```

However, this is a bit expensive because `Point3` will always be passed to the method by value and copied, but it is valid in Go. In this case, the method can also be invoked on a pointer to the type (there is automatic dereferencing). Suppose `p3` is defined as a pointer:

```
p3 := &Point3{3, 4, 5}
```

Then, you can write `p3.Abs()` instead of `(*p3).Abs()`.

Calling methods on values and pointers

There can be methods attached to the type, and other methods attach a pointer to the type. However, this does not matter: *if for a type `T` a method `Meth()` exists on `*T` and `t` is a variable of type `T`, then `t.Meth()` is automatically translated to `(&t).Meth()`. Pointer and value methods can both be called on the pointer or non-pointer values*. This is illustrated in the following program:

```
package main
import (
"fmt"
)

type List []int

func (l List) Len() int { return len(l) }

func (l *List) Append(val int) { *l = append(*l, val) }

func main() {
  // A bare value
  var lst List
  lst.Append(1)
  fmt.Printf("%v (len: %d)\n", lst, lst.Len()) // [1] (len: 1)
  // A pointer value
  plst := new(List)
  plst.Append(2)
  fmt.Printf("%v (len: %d)\n", plst, lst.Len()) // &[2] (len: 1)
}
```

Calling Methods

In the above program, at **line 6**, we declare a new type `List` on the basis of the type of slice on integers(`[]int`). Look at the header of the method `Len()` at **line 8**: `func (l List) Len() int` . The part `(l List)` shows that the method can be called by the object of type `List` only. The method returns the length of the object `l` . Now, look at the header of the method `Append()` at **line 10**: `func (l *List) Append(val int)` . The part `(l *List)` shows that the method can be called by the pointer to the object of type `List` only. The method appends the `val` value at the end of the list `l` .

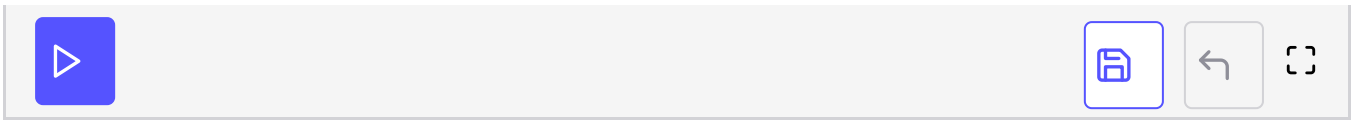
Now, look at `main` , we make a variable `lst` of type `List` using `var` keyword at **line 14**. In the next line, we call `Append(1)` method on `lst` . At **line 16**, we are printing `lst` and length of `lst` by calling `Len()` on `lst` . We make a variable `plst` of type `List` using `new()` function at **line 18**. In the next line, we call `Append(2)` method on `plst` . At **line 20**, we are printing `plst` and the length of `plst` by calling `Len()` on `plst` .

Methods as values and parameters

Methods are just like functions in that they can be used as values, and passed to other functions as parameters. This is shown in the following program:

```
package main
import "fmt"

type T struct {
  a int
}

func (t T) print(message string) {
  fmt.Println(message, t.a)
}

func (T) hello(message string) {
  fmt.Println("Hello!", message)
}

func callMethod(t T, method func(T, string)) {
  method(t, "A message")
}

func main() {
  t1 := T{10}
  t2 := T{20}
  var f func(T, string) = T.print
  callMethod(t1, f)
  callMethod(t2, f)
```

```
  callMethod(t1, T.hello)
}
```



Passing Methods

In the above program, at **line 4**, we make a struct of type `T` with a single integer field `a`. Look at the header of the `print` method at **line 8** as: `func (t T) print(message string)`. The part `(t T)` means that this method can only be called by an object of type `T`. This method is printing `message` sent as a parameter, and the internal field `a` of `t`. Look at the header of the `print` method at **line 12** as: `func (T) hello(message string)`. The part `(T)` means that this method can be directly called with type `T` as: `T.hello("anyString")`. This method is printing **Hello!**, and then prints the `message` parameter. Look at the header of `callMethod` function at **line 16** as: `func callMethod(t T, method func(T, string))`. This function takes `t` as a parameter and a function `method`. That function method is called with `T` and a string **"A message"** as parameters.

Now, look at `main`. We made a `T` type variable `t1` using struct-literal giving `a` value of **10**, at **line 21**. Similarly, in the next line, we make a `T` type variable `t2` using struct-literal giving `a` value of **20**. At **line 23**, we make a variable `f` equal to the `print` method of `T`.

Now, at **line 24**, we are calling the `callMethod` function with `t1` and `f` as parameters. Here, the `method` in `callMethod` is equal to the `print()` method of type `T`. So, from **line 17**, `print()` will be called for `t1` with `message` as **"A message"**. So, **A message 10** will be printed on the screen. Similarly, in the next line, we are calling the `callMethod` function with `t2` and `f` as parameters. Here, the `method` in `callMethod` is equal to `print()` method of type `T`. So, from **line 17**, `print()` will be called for `t2` with `message` as **"A message"**. So, **A message 20** will be printed on the screen.

Now at **line 26**, we are calling the `callMethod` function with `t1` and `T.hello` as parameters. Here the `method` in `callMethod` is equal to the `hello()` method of type `T`. So, from **line 17**, `hello()` will be called for `t1` with the `message` as **A message**. So, **Hello! A message** will be printed on the screen.

That's it about values and pointers, in the next lesson you'll study the un-exported concept in Go.