Introduction to Methods

So far in this course, you have studied and used functions. This lesson brings a new concept of Go similar to functions but slightly different, i.e., methods.

WE'LL COVER THE FOLLOWING

What is a method?

Structs look like a simple form of classes, so an OO programmer might ask, where are the methods of the class? Again, Go has a concept with the same name and roughly the same meaning. A Go **method** is a function that acts on a *variable of a certain type*, called the *receiver*. Therefore, a method is a special kind of function.

> **Note**: A method acting on a variable in Go is similar to the object of a class calling its function in other OO languages, using a `.` selector, e.g., `object.function()`.

The receiver type can be (almost) *anything*, not only a struct type. Any type can have methods, even a function type or alias types for int, bool, string, or array. However, the receiver cannot be an interface type (see Chapter 9) since an interface is an abstract definition and a method is the implementation. Trying to do so generates the compiler error: invalid receiver type.

Lastly, a method cannot be a pointer type, but it can be a pointer to any of the allowed types. The combination of a (struct) type and its methods is the Go equivalent of a class in OO. One important difference is that the code for the type and the methods binding to it are not grouped together. They can exist in different source files; the only requirement is that they have to be in the same package.

The collection of all the methods on a given type `T` (or `*T`) is called the *method set* of `T` (or `*T`).

Methods are functions, so again, there is no *method overloading*, which means for a given type, there is only one method with a given name. However, based on the receiver type, there is overloading. A method with the same name can exist on two or more different receiver types, e.g., this is allowed in the same package:

```
func (a *denseMatrix) Add(b Matrix) Matrix
func (a *sparseMatrix) Add(b Matrix) Matrix
```

An alias of a certain type can't redefine the methods defined on that type because an alias is the same as the original type. However, a new type based on a type can redefine these methods.

This is illustrated in the following example:

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package main
import "fmt"

type S struct {
  a int
}

type SType S // New type
type SAlias = S // Alias
type IntType int // New type
type IntAlias = int // Alias

func (recv S) print() { // function for type defined on type S
  fmt.Printf("%t: %[1]v\n", recv)
}
func (recv SType) print() { // function for type defined on the basis of S
  fmt.Printf("%t: %[1]v\n", recv)
}

// func (recv SAlias) print() { // <-- error: S.print redeclared in this block previous decla
// fmt.Printf("%t: %[1]v\n", recv)
```

```
// fmt.Printf("%t: %[1]v\n", recv)
// }

func (recv IntType) print() { // function for type defined on type on the basis of int
  fmt.Printf("%t: %[1]v\n", recv)
}

// func (recv IntAlias) print() { // <-- error: cannot define new methods on non-local type i
// fmt.Printf("%t: %[1]v\n", recv)
// }

func main(){
  a := S{10}
  a.print() // calling function from line 13
  b := SType{20}
  b.print() // calling function from line 16
  c := SAlias{30}
  c.print() // calling function from line 13
  d := IntType(40)
  d.print() // calling function from line 24
  // e := IntAlias(50) <-- error: e.print undefined (type int has no field or method print)
  // e.print()
}
```

Click the **RUN** button and wait for the terminal to start. Type `go run main.go` and press ENTER. In case you make any changes to the file, you have to press **RUN** again.

In the above code, at **line 4**, we declare a struct of type `S` with one *int* field `a`. Then, at **line 8**, we declare a new type similar to `S` with the name `SType`. In the next line, we alias the type `s` as `SAlias`. Similarly, at **line 10**, we declare a similar type to *int* as `IntType`. In the next line, we alias the type *int* as `IntAlias`.

Let's study the methods involved in our program:

- Look at the header of `print()` method at **line 13**: `func (recv S) print()`. The part `recv S` means that the object of type `S` can call this method. This method is printing the value assigned to the field of the calling object.

- Look at the header of the `print()` method at **line 16**: `func (recv SType) print()`. We redefine the `print()` method for `SType`, as this type is defined based on `S`. So, redefining the method is allowed. The part `recv SType` means that the object of type `SType` can call this method. This method is also printing the value assigned to the field of the calling object.

- See the commented **line 20**. It is the header for the `print()` method but

- See the commented **line 20**. It is the header for the `print()` method but also for the object of type `SAlias`. This will give an error because for alias, the same method of the base class can't be redefined.

- Look at the header of `print()` method at **line 24**: `func (recv IntType) print()` . The part `recv IntType` means that the object of type `IntType` can call this method. This method is printing the value assigned to the field of the calling object.

- See the commented **line 28**. It is the function header for the `print()` method but also for the object of type `IntAlias`. This will give an error because for alias, the same method of the base class can't be redefined.

Let's see `main` now. At **line 33**, we make an `S` type object `a` using struct-literal and assign its field with a value of **10**. In the next line, we call `print()` method on `a`, due to which method at **line 13** will get control.

Similarly, at **line 35**, we made an `SType` type object `b` using struct-literal and assign its field with a value of **20**. In the next line, we call `print()` method on `b`. Due to which, method at **line 16** will gain control.

At **line 37**, we make an `SAlias` type object `c` using struct-literal, and assign its field with a value of **30**. In the next line, we call `print()` method on `c`. Due to which, the method at **line 13** will gain control (as `SAlias` is the alias for type `S`).

At **line 39**, we make an `IntType` type object `d` using struct-literal, and assign its field with a value of **40**. In the next line, we call `print()` method on `d`. Due to which, the method at **line 24** will gain control.

See the commented **line 41**, where we made an `IntAlias` type object `e` using struct-literal and assign its field with a value of **50**. In the next line, we are calling `print()` method. It will generate an error because `IntType` is an alias for *int*, where *int* type doesn't have any method called `print()`. So, it can't find any method `print()`.

The general format of a method is:

```
func (recv receiver_type) methodName(parameter_list) (return_value_list) {
 ... }
```

The receiver is specified in ( ) before the method name after the `func` keyword. If `recv` is the receiver value and `Method1` the method name, then the call or invocation of the method follows the traditional object.method selector notation: `recv.Method1()`. In this expression, if `recv` is a pointer, then it is automatically dereferenced. If the method does not need to use the value `recv`, you can discard it by substituting a `_`, as in:

```
func (_ receiver_type) methodName(parameter_list) (return_value_list) {
... }
```

(or you could also remove it entirely).

Here is a simple example of methods on a struct:

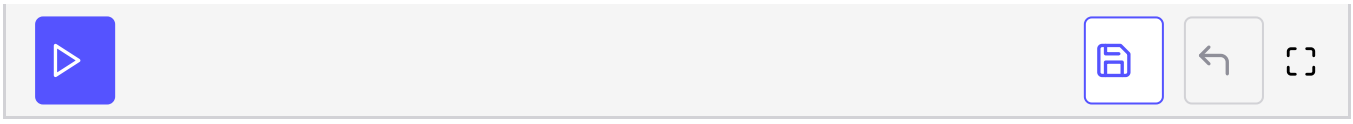
```
package main
import "fmt"

type TwoInts struct {
  a int
  b int
}

func main() {
  two1 := new(TwoInts)
  two1.a = 12
  two1.b = 10
  fmt.Printf("The sum is: %d\n", two1.AddThem()) // calling method
  fmt.Printf("Add them to the param: %d\n", two1.AddToParam(20)) // calling method
  two2 := TwoInts{3, 4}
  fmt.Printf("The sum is: %d\n", two2.AddThem()) // calling method
}

func (tn *TwoInts) AddThem() int { // can be called by pointer to TwoInt type var.
  return tn.a + tn.b
}

func (tn *TwoInts) AddToParam(param int) int { // can be called by pointer to TwoInt type var
  return tn.a + tn.b + param
}
```



Methods on a Struct

In the above code, at **line 4**, we make a struct `TwoInts` with two integer fields `a` and `b`. Before going in `main`, let's see two basic methods: `AddThem()` and `AddtoParam`. Look at the header of method `AddThem` at **line 19**: `func (tn *TwoInts) AddThem() int`. The part `(tn *TwoInts)` means that the pointer to

the variable of type `TwoInts` can call this method. It takes nothing as a parameter and returns an *int* value. It adds both fields of the variable to which `tn` is pointing using the selector operator (see **line 20**), and return the sum. Similarly, at **line 23**, look at the header of the method `AddtoParam`: `func (tn *TwoInts) AddToParam(param int) int` . The part `(tn *TwoInts)` means that pointer to the variable of type `TwoInts` can call this method. It takes an integer `param` as a parameter, and returns an *int* value. It adds both fields of the variable to which `tn` is pointing using the selector operator and then adds `param` also (see **line 24**) and returns the sum.

Now, look at `main`. At **line 10**, we are making a `TwoInts` type variable `two1`. In the next two lines, we are assigning the values to the fields of `two1`. At **line 13**, we are calling the method `AddThem` on `two1` as: `two1.AddThem`. It will return **22** as `a` of `two1` is **12** (see **line 11**), and `b` of `two1` is **10** (see **line 12**). Similarly, in the next line, we are calling the method `AddToParam` on `two1` as: `two1.AddToParam(20)`. It will return **42** (10+12+20). At **line 15**, we are making a `TwoInts` type variable `two2` using struct-literal, giving `a` of `two2` the value of **3** and `b` of `two2` the value of **4**. At **line 16**, we are calling method `AddThem` on `two2` as: `two2.AddThem`. It will return 7 (3+4).

A method and the type on which it acts must be defined in the same package; that's why you cannot define methods on type *int*, *float*, or the like. Trying to define a method on an int type gives the compiler error: `cannot define new methods on non-local type int`.

For example, if you want to define the following method on `time.Time`:

```
func (t time.Time) first3Chars() string {
return time.LocalTime().String()[0:3]
}
```

You get the same error for a type defined in another, thus also non-local package. However, there is a way around this: you can define an alias for that type (int, float, ...), and then define a method for that alias. Or, embed the type as an unknown type in a new struct, like in the following example. Of course, this method is only valid for the alias type.

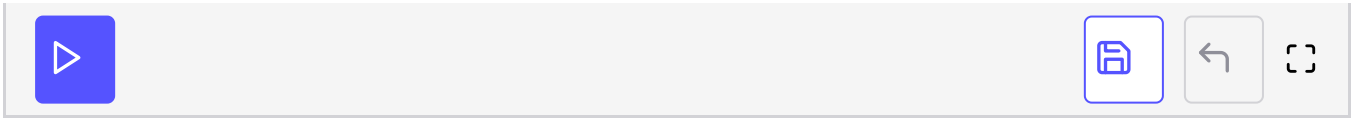
```
package main
import (
"fmt"
```

```
"time"
)


type myTime struct {
  time.Time //anonymous field
}

func (t myTime) first3Chars() string {
  return t.String()[0:3]
}

func main() {
  m := myTime{time.Now()}
  //calling existing String method on anonymous Time field
  fmt.Println("Full time now:", m.String())
  fmt.Println("First 3 chars:", m.first3Chars()) //calling myTime.first3Chars
}
```



Methods on time

In the above code, we importe package `time` at **line 4** to use its methods. At **line** 7, we make a struct of type `myTime` and create an anonymous field of type `time.Time` in it. Look at the header of method `first3Chars()` at **line 11**: `func (t myTime) first3Chars() string` . The part `(t myTime)` means that the variable of type `myTime` can call this method. It takes nothing as a parameter and returns a *string* value. It converts the time stored in the field of `t` to string and returns the first three characters (see **line 12**).

Now, look at `main` . At **line 16**, we are creating a variable of type `myTime` `m` using struct-literal, and setting its anonymous field of type `time.Time` to the *present* time. At **line 18**, we are printing the string value for `m` . It means that the present time that was stored at **line 16** will be printed in the form of a string. In the next line, we call the method `first3Chars` on `m` as: `m.first3Chars()` , which will print only the first three characters of the present time from **line 16** after returning from the method.

Now, that you know what are methods and how to use them, let's study the difference between methods and functions.