

Introduction

This lesson explains how methods and interfaces in Go work using an example

WE'LL COVER THE FOLLOWING ^

- Methods in Go
- Example

Methods in Go

While technically Go isn't an [Object Oriented Programming language](#), types and methods allow for an object-oriented style of programming. The big difference is that Go does not support type inheritance (mechanism of acquiring the features and behaviours of a class by another class) but instead has a concept of [interface](#).

In this chapter, we will focus on Go's use of methods and interfaces.

Note: A frequently asked question is “what is the difference between a function and a method”. A method is a function that has a defined receiver, in OOP terms, a method is a function on an instance of an object.

Go does **not** have classes. However, you can define methods on struct types.

The *method receiver* appears in its own argument list between the `func` keyword and the method name. Here is an example with a `User` struct containing two fields: `FirstName` and `LastName` of string type.

Example

Key:	Value:
GOPATH	/go





```
package main

import (
    "fmt"
)

type User struct {
    FirstName, LastName string
}

func (u User) Greeting() string {
    return fmt.Sprintf("Dear %s %s", u.FirstName, u.LastName)
}

func main() {
    u := User{"Matt", "Aimonetti"}
    fmt.Println(u.Greeting())
}
```



Note how methods are defined outside of the struct, if you have been writing Object Oriented code for a while, you might find that a bit odd at first. The method on the `User` type could be defined anywhere in the package.

Now that we know how methods and interfaces in Go work, in the following lesson, we will look into how the code itself should be organized.