Learning Objectives

Learners will be able to...

- **associate methods with a type**
- **write getter and setter methods**
- **embed attributes and methods from other types**

info

Make Sure You Know

Python or other modern programming language.

OOP in Go

There are no classes in Go, however this does not prevent us from using Object-Oriented Programming (OOP) approaches when writing code.

For any named type, we can define methods or functions associated with types. You can declare a method for both structures and non-structural types.

A method in Go is a function with a **receiving argument**:

```
func (r receivingType) methodName()
```

where the receiving type is between the `func` keyword and the method name. The receiving type signifies that the method is defined for the receiving type.

The method must be defined in the same package where the receiver type is defined. When we try to define a method for a type that is defined in another package, we will receive an error: `cannot define new methods on non-local type`.

Here is an example:

```
type Book struct {
    Title string
    ISBN  int
}

func (b Book) title() string {
    return b.Title
}
```

In this case, the `title` method is defined for the `Book` type named `b`.

If a type is specified as the receiver, then the method uses a copy of the original object. Thus, we cannot change the original object, unlike the case when we have a pointer receiver in a method. We will talk about this case further.

Getter and Setter

In Go, the case (i.e. lower vs UPPER case) of the first letter of a field or function is used to distinguish private from public. This means if we want to hide direct access to a field, we can write them with a lower letter and use a getter and setter to access them.

> xdiscipline
>
> ## ## Style: Getter Names
>
> Go style preferneces brevity - so the `get` prefix for getter functions is often excluded.
>
> For example, using `Title` and not `GetTitle` in the example below.

Consider the following:

```
type Book struct {
    title string
    isbn  int
}

func (b *Book) Title() string {
    return b.title
}

func (b *Book) SetTitle(title string) {
    b.title = title
}
```

The `title` and `isbn` fields are private and can only be accessed through `Title()` and `SetTitle()` methods.

Embedding

As an alternative to the inheritance mechanism in traditional object-oriented programming languages, Go has an embedding mechanism.

You embed the parent struct in the child struct by declaring an anonymous field, such as:

```
// parent
type User struct {
    firstName string
    lastName string
}

//child
type Teacher struct {
    User // embedding
    academicDegree string
}

// or
type Teacher struct {
    *User // embedding
    academicDegree string
}
```

Take a look at the example on the left. A Teacher object is created which can use both User and Teacher methods:

```
teacher := newTeacher("Jack", "Lee", "PhD")
fmt.Println(teacher.FirstName())
fmt.Println(teacher.AcademicDegree())
```

While Teacher inherits User, sub-typing is not allowed. That is, Teacher methods cannot be used by User objects. For example, the following throws an error:

```
user := User{firstName: "Jack", lastName: "Lee"}
fmt.Println(user.lastName())
```

You can see that in the constructor, an anonymous `User` object is created:

```
func newTeacher(firstName, lastName, academicDegree string)
Teacher {
    user := User{firstName: firstName, lastName: lastName}
    return Teacher{User: user, academicDegree: academicDegree}
}
```

If we try to change the `User` field in the `Teacher` structure to non-anonymous we get an error:

```
type Teacher struct {
    User User
    academicDegree string
}
```

Methods of the parent type can be overridden by the child type.

For example:

```
func (t *Teacher) FirstName(prefix string) string {
    return fmt.Sprintf("%s %s", prefix, t.firstName)
}

func main() {
    teacher := newTeacher("Jack", "Lee", "PhD")
    fmt.Println(teacher.FirstName("Dr."))
    fmt.Println(teacher.AcademicDegree())
}
```