Object Oriented Go - The Basics

Dec 17, 2017 • @IcyApril

I have evangelised Object-Oriented Programming for the bulk of my programming life; when I left school as a teenager I was accepted into an apprenticeship where the primary language was Java, more recently I have written books on Object Oriented Programming in PHP and PHP Design Patterns - largely focusing on refactoring legacy code to a more Object-Oriented design. From this information alone, you'd not be mistaken for thinking of me as a "traditional" OOP programmer.

A few years ago I was given a commercial project to create the first solution to a problem surrounding sensor placement on vehicles; the solution to this problem would later become my masters thesis. With the mathematics squared away, I was tasked with creating an implementation of my proposed solution; due to large processing times and the need for concurrent processing, I chose to do so in the Go (formerly, Golang) Programming Language.

As I started to build in Go, the language looked completely different than any other Object-Oriented language I'd seen before, certainly from the perspective of someone building with Java. Where were the classes? How can interfaces be "implicit"? Despite my initial confusion, I soon came to learn that Go is indeed an Object-Oriented language, and a fine one at that. This blog post explains the basics of writing Object-Oriented code in Go.

Object Oriented Programming typically refers to software where the responsibilities are split into objects; these objects are created from predetermined structures which define how objects are created and what they can do, in Java we refer to these scaffolds as "classes". For example; in a Java application, we can have an object called `Bob` created from a class called `Users`. These objects contain a set of methods (e.g. a user class could outline that a method that defined how a user can send a message to another user). The object exposes methods that allow the object to perform various action, but also allow the object to hide its internal organs.

Objects act as building blocks which allow for applications to be built in a scalable and reusable way. Objects can be put together to build other objects; Design Patterns can provide reusable patterns which allow for how objects can be created, be structured or how they can interact with each other.

Types

Go contains a few different primitive types such as `string` and `float64`. We can use them for type hinting - to define parameters which methods can accept, and indeed return types too. Let's define a fairly simple method:

```
package main

import (
	"fmt"
)

func sum(a int, b int) int {
	return a + b
}
```

At the bottom of this program, if we add the following `main()` method:

```
package main

import (
	"fmt"
)

func sum(a int, b int) int {
	return a + b
}

func main() {
	fmt.Println(sum(1, 2))
}
```

We get the output:

```
3
```

However, if we instead try running this without valid integer values - such that the `main` method looks like this:

```
func main() {
	fmt.Println(sum("hello", "test"))
}
```

We will see an error message like the one below:

```
main.go:12:18: cannot use "hello" (type string) as type int in argument to sum
main.go:12:27: cannot use "test" (type string) as type int in argument to sum
```

Whilst this adds some useful type safety; in order to simplify our code we are also able to create our own types, which are more complex than the scalar types we are using here.

Structs

For those familiar with Object-Oriented programming in other languages, Structs are somewhat like classes. They are a type which defines variables and functions (methods). A `struct` is a type which contains named values.

```
type user struct {
	name  string
	admin bool
}
```

We can initialise a struct as follows, and use `fmt.Println` to dump out values:

```
package main

import (
	"fmt"
)

type user struct {
	name  string
	admin bool
}

func main() {
	paul := user{name: "Paul Smithson", admin: false}
	fmt.Println(paul)
}
```

This provides a rather crude dump of data:

```
{Paul Smithson false}
```

If we want to access named fields from the `main` methods we can simply use:

```
func main() {
	paul := user{name: "Paul Smithson", admin: false}
	fmt.Println(paul.name)
}
```

This will output the specific value from the named field we asked for by resolving `paul.name`, the output is now like this:

```
Paul Smithson
```

Adding Methods to Structs

We can also add methods to a struct, for example, here I've added an `isAdmin` method to the struct which returns true or false depending on a users admin status:

```
type user struct {
	name  string
	admin bool
}

func (u user) IsAdmin() bool {
	return u.admin
}
```

I can also define a method which will return the first name of a user from a struct:

```
func (u user) FirstName() string {
	names := strings.Fields(u.name)
	if len(names) > 0 {
		return names[0]
	}
	return "Unnamed"
}
```

Putting this all together, our program now looks like this:

```
package main

import (
	"fmt"
	"strings"
)

type user struct {
	name  string
	admin bool
}

func (u user) IsAdmin() bool {
	return u.admin
}

func (u user) FirstName() string {
	names := strings.Fields(u.name)
	if len(names) > 0 {
		return names[0]
	}
	return "Unnamed"
}

func main() {
	paul := user{name: "Paul Smithson", admin: false}
	fmt.Printf("First Name: %s\n", paul.FirstName())
	fmt.Printf("Admin? %t\n", paul.IsAdmin())
}
```

This now returns the following output, `false` from the `IsAdmin` method and `Paul` from the `FirstName` method:

```
First Name: Paul
Admin? false
```

We can also accept inputs into the functions we're attaching to our structs, for example, suppose we wanted to generate a dynamic greeting based on whether it was (or wasn't) the morning, we could use the following implementation:

```
func (u user) Greeting(morning bool) string {
	if morning == true {
		return "Good Morning, " + u.FirstName()
	}
	return "Hello, " + u.FirstName()
}
```

As with our other methods, we can run this quite easily - we just parse the arguments in:

```
func main() {
	paul := user{name: "Paul Smithson", admin: false}
	fmt.Println(paul.Greeting(true))
}
```

This returns:

```
Good Morning, Paul
```

You can find a complete demo of this code on The Go Playground; be sure to try changing things around before moving forward.

Interfaces and Polymorphism

Sometimes when building applications, we want to implement the same feature multiple ways. For example; we might want to have a storage layer, but want to implement it both as a JSON file, and an XML file.

Polymorphism allows us to build classes that expose identical public interfaces, but perform differently under the hood. We might have a `user` Struct and a `bot` struct, but for both we want to expose an identical interface for both, such that they can be used interchangeably. For this to happen; we need to use interfaces.

Let's define an interface called `client`, this will eventually be used by the `user` and `bot` structs:

```
type client interface {
	FirstName() string
	IsAdmin() bool
}
```

Like, in the previous section, I've created two structs and defined `FirstName` and `IsAdmin` methods to structs. Note that when we define these structs, we do not have to explicitly mention which interfaces we're implementing. As soon as Go notices a structs with the correct methods, it will detect the interface has been implemented at a compiler level.

```
type user struct {
	name  string
	admin bool
}

func (u user) IsAdmin() bool {
	return u.admin
}

func (u user) FirstName() string {
	names := strings.Fields(u.name)
	if len(names) > 0 {
		return names[0]
	}
	return "Unnamed"
}

type bot struct {
	admin bool
}

func (b bot) IsAdmin() bool {
	return b.admin
}

func (b bot) FirstName() string {
	return "Bot"
}
```

Now, having defined two structs which implement the `client` interface, we can now use type hinting to define methods which can be used by structs which implement that interface; for example:

```

func Hello(c client) string {
	adminStatus := "a normal user"
	if c.IsAdmin() == true {
		adminStatus = "an admin"
	}
	return "Hello, I am " + c.FirstName() + " and I am " + adminStatus + "."
}
```

Using all the structs and methods above, we can define a `main()` method which uses the `Hello` method:

```
func main() {
	paul := user{name: "Paul Smithson", admin: false}
	fmt.Println(Hello(paul))

	automationBot := bot{admin: true}
	fmt.Println(Hello(automationBot))
}
```

This then has the following output:

```
Hello, I am Paul and I am a normal user.
Hello, I am Bot and I am an admin.
```

Suppose we now deleted the `FirstName` method on the `bot` struct, the Go compiler would dynamically detect that we are not implementing the `client` interface and will display an error for us:

```
cannot use automationBot (type bot) as type client in argument to Hello:
bot does not implement client (missing FirstName method)
```

As interfaces are defined implicitly in Go, we are able to define interfaces around structs we don't ordinarily have access to alter (for example, if we pull one in as a third party interface). This also means, we are able to easily mock structs which we don't have access to change.

You can play around with the entire code for this section on The Go Playground.

Composition, and the Lack of Inheritance

When building Object-Oriented applications, a key consideration is how classes interact with each other. We'll need to build objects from other objects, we might need to extend the behaviour of a class. Inheritance and Composition are two ways of achieving this behaviour, Go exclusively uses Composition, but it is still worth understanding the story behind why this decision is important.

Inheritance

Java contains a keyword known as `extends`, this allows for hierarchy to be inserted between between classes. You might have a `HeadTeacher` class, which inherits properties from a `Teacher`. Whilst `HeadTeacher` has all the methods from the `Teacher` class, we can also add in extra methods for managing other teachers (i.e. `fireTeacher()`).

Another example of Inheritance can be vehicles; we can define a `Vehicle` class which acts as an abstract representation of a vehicle, it contains the boilerplate code for starting the ignition, acceleration, etc. The concrete implementations of this abstract `Vehicle` class can exist in other classes; such as `Car`, `Lorry`, `Bus` or `Motorcycle`. These concrete classes will add and override methods and variables in the boilerplate `Vehicle` class depending on the specifics of the implementation. This hierarchy can go even deeper, we can, for example; extend the `Car` class to have a `LuxuryCar` class.

Inheritance defines a "is-a" relationship between classes. A `LuxuryCar` is a `Car`, and a `Car` is a `Vehicle`. This causes a number of architectural difficulties; firstly a change to one method at the top of the hierarchy can have a cascading change down the entire tree, this can lead to unintended side-effects in child classes. The quirks of a parent can cause problems later down the tree.

Secondly, at an engineering level, we rarely consider things to have a "is-a" relationship - we instead tend to consider them having a "has-a" relationship. For example; a `Car` has an `Engine` and 4 `Wheels`.

In the famous Design Patterns book, even recommended "favouring Composition over Inheritance". The founder of the Java language (James Gosling) went even further in condemning Inheritance:

> *I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible.*

– *Java Toolbox* by Allen Holub, JavaWorld

Composition & Dependency Injection

Instead of using Inheritance, Go omitted it and chose to purely use Composition. Composition is how we can build class relationships using a "has-a" relationship. Suppose we are building a `Users` object, it can be composed of a `Permissions` object and a `Person` object.

This is where Composition comes into play, instead of building our objects on a hierarchy, we build objects what functionality they can do (using Dependency Injection).

Here's an example of two structs, you'll notice that the `user` struct embeds an instance of `permissions` into itself. In essence, we're abled to build a struct which contains another:

```
type permissions struct {
	admin bool
	suspended bool
}

type user struct {
	name  string
	access permissions
}
```

When we create an instance of `user`, we're able to define methods which access the fields and methods from the `permissions` object within it:

```
package main

import (
	"fmt"
	"strings"
)

type permissions struct {
	admin bool
	suspended bool
}

type user struct {
	name  string
	access permissions
}

func (u user) FirstName() string {
	names := strings.Fields(u.name)
	if len(names) > 0 {
		return names[0]
	}
	return "Unnamed"
}

func (u user) IsAdmin() bool {
	return u.access.admin
}

func (u user) IsSuspended() bool {
	return u.access.suspended
}

func main() {
	newAdminStatus := permissions{admin: true, suspended: false}
	paul := user{name: "Paul Smithson", access: newAdminStatus}
	fmt.Printf("First Name: %s\n", paul.FirstName())
	fmt.Printf("Admin? %t\n", paul.IsAdmin())
	fmt.Printf("Suspended? %t\n", paul.IsSuspended())
}
```

The above program yields the following output:

```
First Name: Paul
Admin? true
Suspended? false
```

As before, you can modify and play with this code on The Go Platground.

Dependency Injection effectively refers to injecting one object into the constructor when instantiating another, in the next section we'll cover how you can create functions which returns objects, instead of us directly needing to instantiate structs.

Encapsulation

A key consideration in Object-oriented programming is around Encapsulation. Encapsulation is about keeping internal methods hidden away, whilst exposing the interface that we want to expose.

Go does encapsulation at the package, there is no control of visibility *within* a package itself - but you are able to control visibility from one package to another. This is one of the reasons.

Whilst some languages prefix the function name with the visibility (e.g. `private sum`), go does this using the case of the first character of the method name; so for example, `sum` would be private as it has a lowercase `s` but `Sum` would be public because it has an upper-case `S`. Let's put this to the test.

I've defined a simple Go package called `user`; this package defines the `User` struct, and an `IsAdmin` and a `FirstName` method. I've also added a `New` method which effectively acts as a constructor and returns an instance of the object we're after without needing to expose any of the internal fields of the struct.

I've put this package in `go/src/junade.com/icyapril/test/user`; whilst our main method lives in `go/src/junade.com/icyapril/test`, the `user` directory acts as the sub-package:

```
package user

import "strings"

type User struct {
	name  string
	admin bool
}

// IsAdmin - checks if the user is an admin
func (u User) IsAdmin() bool {
	return u.admin
}

// FirstName - get users first name
func (u User) FirstName() string {
	names := strings.Fields(u.name)
	if len(names) > 0 {
		return names[0]
	}
	return "Unnamed"
}

// Greeting - Returns a string with a friendly greeting
func (u User) Greeting(morning bool) string {
	if morning == true {
		return "Good Morning, " + u.FirstName()
	}
	return "Hello, " + u.FirstName()
}

// New user object
func New(userName string, userAdmin bool) User {
	return User{name: userName, admin: userAdmin}
}
```

To use this package, I have a simple main method in `go/src/junade.com/icyapril/test` that uses the `user` package - I use the `user.New` method to get an instance of the `user` struct:

```
package main

import (
	"fmt"

	"junade.com/icyapril/test/user"
)

func main() {
	paul := user.New("Paul Smithson", false)
	fmt.Printf("First Name: %s\n", paul.FirstName())
	fmt.Printf("Admin? %t\n", paul.IsAdmin())
	fmt.Println(paul.Greeting(true))
}
```

The Go method therefore provides the following output:

```
First Name: Paul
Admin? false
Good Morning, Paul
```

If I now change all references for `user.New` to `user.new` and try re-running the program - Go will not let us execute an unexpected method in a different class:

```
./main.go:10:10: cannot refer to unexported name user.new
./main.go:10:10: undefined: user.new
```

Conclusion

Go is undoubtably an Object-Oriented language and you can use Object-Oriented language features to build a applications which are easy to extend (being resilient to the forces of change) and have a far more maintainable codebase.