Multiple Inheritance

This lesson describes what multiple inheritance is and how Go achieves this via structs.

WE'LL COVER THE FOLLOWING

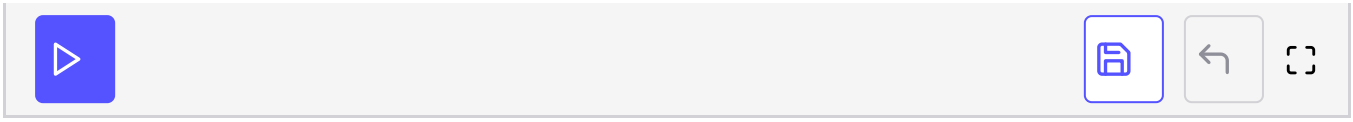
- Universal methods and method naming
- Comparisons between Go types and methods and other OO languages

Multiple inheritance is the ability for a type to obtain the behaviors of more than one parent class. In classical OO languages, it is usually not implemented (exceptions are C++ and Python), because, in class-based hierarchies, it introduces additional complexities for the compiler. But in Go, multiple inheritance can be implemented simply by embedding all the necessary 'parent' types in the type under construction.

Look at the following implementation:

```
package main
import "fmt"

type Camera struct { }

func (c *Camera) TakeAPicture() string {  // method of Camera
  return "Click"
}

type Phone struct { }

func (p *Phone ) Call() string {  // method of Phone
  return "Ring Ring"
}

// multiple inheritance
type SmartPhone struct {    // can use methods of both structs
  Camera
  Phone
}

func main() {
  cp := new(SmartPhone)
  fmt.Println("Our new SmartPhone exhibits multiple behaviors ...")
```

```
  fmt.Println("It exhibits behavior of a Camera: ", cp.TakeAPicture())
  fmt.Println("It works like a Phone too: ", cp.Call())
}
```



Multiple Inheritance

In the above code, at **line 4**, we make a struct of type `Camera` with no fields. At **line 6**, there is a method that can be called only by a pointer that points to a `Camera` object. The method `TakeAPicture` returns a string **Click**. At **line 10**, we make a struct of type `Phone` with no fields. At **line 12**, there is a method that can be called only by a pointer pointing to a `Phone` object. The method `Call` just returns a string **Ring Ring**.

Now, we have a struct `SmartPhone` at **line 17**. It can implement the functionalities of both `Camera` and `Phone`. It means we need multiple inheritance now. So, we made two anonymous fields in `SmartPhone`: `Camera` and `Phone`. Now, look at `main`. We make a `SmartPhone` type pointer variable `cp` using the `new()` function. Now, this `cp` object can call methods of `Camera` and `Phone` as well. See **line 25**. We are calling the method `TakeAPicture`, which can be called by the `Camera` object. Since `SmartPhone` inherits `Camera`, `cp.TakeAPicture()` works fine. It will print **Click** on the screen. In the next line, we are calling the method `Call()`, which can be called by the `Phone` object. Since `SmartPhone` inherits `Phone`, `cp.Call()` works fine. So it will print **Ring Ring** on the screen.

Universal methods and method naming

In programming, a number of basic operations appear over and over again like opening, closing, reading, writing, sorting, and so on. Moreover, they have a general meaning to them: opening can be applied to a file, a network connection, a database connection, and so on. The implementation details are very different in each case, but the general idea is the same. In Go, this is applied extensively in the standard library through the use of interfaces (see Chapter 9), wherein such generic methods get canonical names like `Open()`, `Read()`, `Write()`, and so on. If you want to write idiomatic Go, you should follow this convention, giving your methods the same names and signatures as those ‘canonical’ methods where appropriate. This makes Go software more consistent and readable. For example, if you need a convert-to-

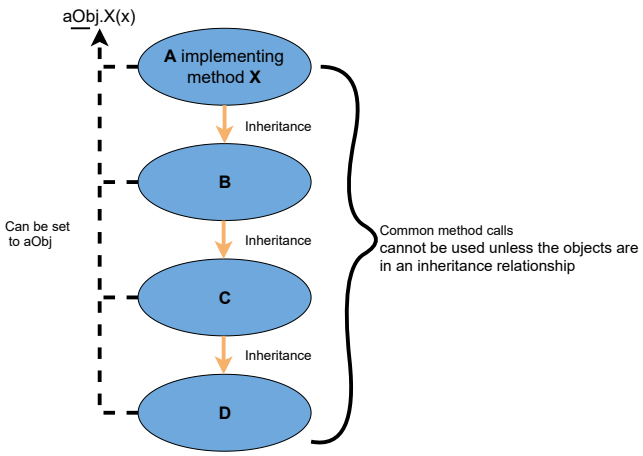
string method, name it `String()` and not `ToString()`.

Comparisons between Go types and methods and other OO languages

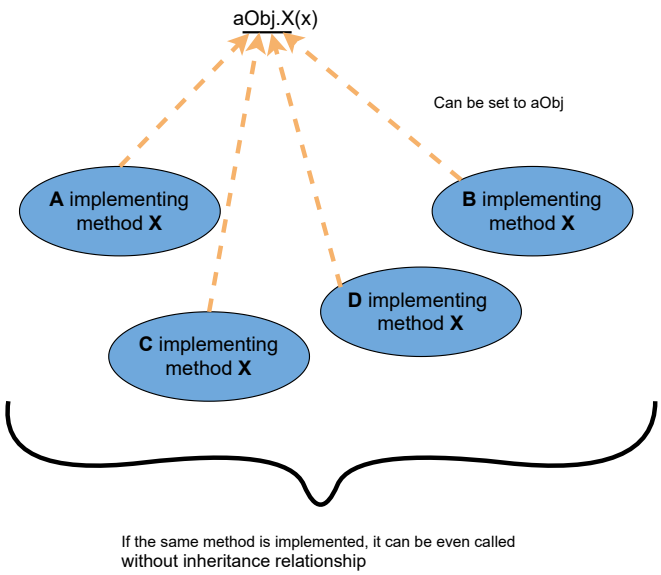
Methods in OO languages like C++, Java, C#, or Ruby are defined in the context of classes and inheritance: when a method is invoked on an object, the runtime sees whether its class or if any of its superclasses have a definition for that method. Otherwise, the result is an exception.

In Go, such an inheritance hierarchy is not needed at all. If the method is defined for that type, it can be invoked, independent of whether or not the method exists for other types. In that sense, there is greater flexibility.

This is nicely illustrated in the following schema:



C++ Example



Go

Go *does not require an explicit class definition* like Java, C++, C#, etc do. Instead, a *class* is implicitly defined by providing a set of methods that operate on a common type. This type may be a struct or any other user-defined type.

For example, say we would like to define our Integer type to add some possible methods for working with integers, like a to-string-conversion. In Go, we would define this as:

```
type Integer int
```

```
func (i *Integer) String() string {


  return strconv.Itoa(i)
}
```

In Java or C#, the type and the `func` would be placed together in a class `Integer`; in Ruby, you could write the method on the basic type int.

In Go, types are classes (data and associated methods). Go doesn't know inheritance like class-oriented OO languages. Inheritance has two main benefits, *code reuse* and *polymorphism*.

Code reuse in Go is achieved through composition and delegation, and polymorphism is achieved through the use of interfaces: it implements what is sometimes called *component programming*. Many developers say that Go's interfaces provide a more powerful and yet simpler polymorphic behavior than class inheritance.

If you need more OO capabilities, take a look at the `goop` package ("Go Object-Oriented Programming") from *Scott Pakin* (click here). It provides Go with JavaScript-style objects (prototype-based objects) but supports multiple inheritance and type-dependent dispatch so you can probably implement most of your favorite constructs from other programming languages.

Now you are familiar with the use of arrays, in the next lesson, you have to write a program to solve a problem.