methods

a method is a function that is declared with a receiver

methods

a method is a function that is declared with a receiver

main.go

```go
package main

import "fmt"

type person struct {
    fname string
    lname string
    age   int
}

func (p person) fullName() string {
    return p.fname + p.lname
}

func main() {
    p1 := person{"James", "Bond", 20}
    fmt.Println(p1.fname)
    fmt.Println(p1.lname)
    fmt.Println(p1.age)
    fmt.Println(p1.fullName())
}
```

receiver

Terminal

```
01_struct $ go run main.go
James
Bond
20
JamesBond
01_struct $
```

main.go

```go
package main

import "fmt"

type person struct {
    fname string
    lname string
    age   int
}

func (p person) fullName() string {
    return p.fname + p.lname
}

func main() {
    p1 := person{"James", "Bond", 20}
    p2 := person{"Miss", "Moneypenny", 18}
    fmt.Println(p1.fullName())
    fmt.Println(p2.fullName())
}
```

receiver

Terminal

```
02_struct $ go run main.go
JamesBond
MissMoneypenny
02_struct $
```

Receivers

- There are two types of receivers:
  - **value**
  - **pointer**

Receivers

- There are two types of receivers:
  - value
  - pointer

```go
type person struct {
    fname string
    lname string
    age   int
}

func (p person) fullName() string {
    fmt.Printf("Inside method:  %p\n", &p)
    return p.fname + p.lname
}

func main() {
    p1 := person{"James", "Bond", 20}
    fmt.Println(p1.fullName())
    fmt.Printf("Inside main: %p\n", &p1)
}
// p1 is the receiver value for the call to fullName
// fullName is operating on a copy of p1
```

operate on a copy of the value used to make the method call

```go
type person struct {
    fname string
    lname stri
    age   int
}

func (p *person) changeAge(newAge int) {
    p.age = newAge
}

func main() {
    p1 := person{"James", "Bond", 20}
    fmt.Println(p1.age)
    p1.changeAge(21)
    fmt.Println(p1.age)
}
```

operate on the actual value used to make the method call

```
03_struct_pointer-receiver
03_struct_pointer-receiver $ go run main.go
20
21
03_struct_pointer-receiver $ _
```

Receivers

- There are two types of receivers:
  - **value**
    - **you don’t need to change the value making the method call**
  - **pointer**
    - **you need to change the value making the method call**

Which one should you use?

a type’s nature

a type’s nature should dictate how you use it

Primitive Types

Pass a Copy

- Golang by default includes several **pre-declared, built-in, primitive** types
  - boolean
  - numeric
  - string

} primitive types
Pass a **copy**, the actual value; not a reference pointer

Reference Types

Pass a Copy

reference types
point to some underlying data structure

- Reference types
  - slice
  - map
  - channel
  - interface
  - function

Header Value

When we declare a reference type, the value that is created is a **header value**. The header value contains a pointer to an underlying data structure. Do not use pointers with reference types. Pass a **copy**; the actual value. The actual value already has a reference pointer to the underlying data structure. When you give a copy of the actual value, that copy also is a pointer to the same underlying data structure. Both the copy, and the original, point to the same underlying data structure.

Struct Types

- Use
  - **value**
    - **if you don’t need to change the value**
    - can also convey semantic meaning
      - eg, Time pkg
        - time is immutable
  - **pointer**
    - **if you need to change the value**
    - typically used with structs

Struct Types

Depends

- Use
  - **value**
    - **if you don’t need to change the value**
    - can also convey semantic meaning
      - eg, pkg time
        - time is immutable

    We could say this has a primitive nature
  - **pointer**
    - **if you need to change the value**
    - typically used with structs
      - eg, *Files in pkg os

    We could say this has a non-primitive nature

a type’s nature

a type’s nature should dictate how you use it

In most cases, **struct** types don't exhibit a primitive nature but a **nonprimitive** one. In these cases, adding or removing something from the value of the type should mutate the value. When this is the case, we want to use a pointer to share the value with the rest of the program that needs it. … [Examples:] … When you think about time, you realize that any given point in time is not something that can change. This is exactly how the standard library implements the Time type. … Since values of type File have a non-primitive nature, they are always shared and never copied.

~William Kennedy

The decision to use a value or pointer receiver should not being based on whether the method is mutating the receiving value. The decision should be based on **the nature of the type**. One exception to this guideline is when you need the flexibility that value type receivers provide when working with interface values. In these cases, you may choose to use a value receiver even though the nature of the type is nonprimitive. It's entirely based on the mechanics behind how interface values call methods for the values stored inside of them.

~William Kennedy

exercise

write a program
that uses a method

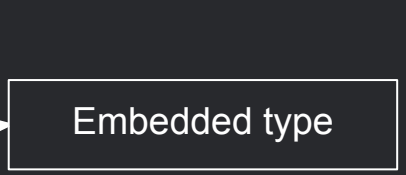
embedded types

“Go’s type system does not support **inheritance**. In Go, **composition** is preferred over inheritance where type **embedding** is the way to implement **composition**. Many pragmatic developers are proponents of using **composition** over inheritance.”

~ this great article

main.go

```go
package main

import "fmt"

type Vehicle struct {
    Seats    int
    MaxSpeed int
    Color    string
}

type Car struct {
    Vehicle
    Wheels int
    Doors  int
}

type Plane struct {
    Vehicle
    Jet bool
}

type Boat struct {
    Vehicle
    Length int
}

func (v Vehicle) Specs() {
    fmt.Printf("Seats %v, max speed %v, color %v\n", v.Seats, v.MaxSpeed, v.Color)
}

func main() {
    prius := Car{Vehicle{6, 120, "white"}, 4, 5}
    prius.Specs()
}
```



Terminal

```
01 $ go run main.go
Seats 6, max speed 120, color white
01 $
```

And we wanted to create a new `Android` struct. We could do this:

```
type Android struct {
  Person Person
  Model string
}
```

not this way

This would work, but we would rather say an Android is a Person, rather than an Android has a Person. Go supports relationships like this by using an embedded type. Also known as anonymous fields, embedded types look like this:

```
type Android struct {
  Person
  Model string
}
```

this way

We use the type (`Person`) and don't give it a name. When defined this way the `Person` struct

http://www.golang-book.com/books/intro/9

exercise

write a program
that uses an embedded type

Review

- methods
  - receivers
- embedded types
  - composition vs. inheritance