Table of Contents

Introduction

Method sets of a particular type or value are of particular importance in Go, where the method set determines what interfaces a value implements.

The Spec

There are two important clauses in the Go Language Specification about method sets. They are as follows:

Method Sets: A type may have a method set associated with it. The method set of an interface type is its interface. The method set of any other named `type T` consists of all methods with receiver type `T`. The method set of the corresponding pointer type `*T` is the set of all methods with receiver `*T` or `T` (that is, it also contains the method set of `T`). Any other type has an empty method set. In a method set, each method must have a unique name.

Calls: A method call `x.m()` is valid if the method set of (the type of) `x` contains `m` and the argument list can be assigned to the parameter list of `m`. If `x` is addressable and `&x`'s method set contains `m`, `x.m()` is shorthand for `(&x).m()`.

Usage

There are many different cases during which a method set crops up in day-to-day programming. Some of the main ones are when calling methods on variables, calling methods on slice elements, calling methods on map elements, and storing values in interfaces.

Variables

In general, when you have a variable of a type, you can pretty much call whatever you want on it. When you combine the two rules above together, the following is valid:

```
type List []int
```

```
func (l List) Len() int        { return len(l) }
func (l *List) Append(val int) { *l = append(*l, val) }

func main() {
    // A bare value
    var lst List
    lst.Append(1)
    fmt.Printf("%v (len: %d)\n", lst, lst.Len())

    // A pointer value
    plst := new(List)
    plst.Append(2)
    fmt.Printf("%v (len: %d)\n", plst, plst.Len())
}
```

Note that both pointer and value methods can both be called on both pointer and non-pointer values. To understand why, let's examine the method sets of both types, directly from the spec:

```
List
- Len() int

*List
- Len() int
- Append(int)
```

Notice that the method set for `List` does not actually contain `Append(int)` even though you can see from the above program that you can call the method without a problem. This is a result of the second spec section above. It implicitly translates the first line below into the second:

```
lst.Append(1)
(&lst).Append(1)
```

Now that the value before the dot is a `*List`, its method set includes Append, and the call is legal.

To make it easier to remember these rules, it may be helpful to simply consider the pointer- and value-receiver methods separately from the method set. It is legal to call a pointer-valued method on anything that is already a pointer or whose address can be taken (as is the case in the above example). It is legal to call a value method on anything which is a value or whose value can be dereferenced (as is the case with any pointer; this case is specified explicitly in the spec).

Slice Elements

Slice elements are almost identical to variables. Because they are addressable, both pointer- and value-receiver methods can be called on both pointer- and

value-element slices.

Map Elements

Map elements are not addressable. Therefore, the following is an *illegal* operation:

```
lists := map[string]List{}
lists["primes"].Append(7) // cannot be rewritten as (&lists["primes"]).Append(7)
```

However, the following is still valid (and is the far more common case):

```
lists := map[string]*List{}
lists["primes"] = new(List)
lists["primes"].Append(7)
count := lists["primes"].Len() // can be rewritten as (*lists["primes"]).Len()
```

Thus, both pointer- and value-receiver methods can be called on pointer-element maps, but only value-receiver methods can be called on value-element maps. This is the reason that maps with struct elements are almost always made with pointer elements.

Interfaces

The concrete value stored in an interface is not addressable, in the same way that a map element is not addressable. Therefore, when you call a method on an interface, it must either have an identical receiver type or it must be directly discernible from the concrete type: pointer- and value-receiver methods can be called with pointers and values respectively, as you would expect. Value-receiver methods can be called with pointer values because they can be dereferenced first. Pointer-receiver methods cannot be called with values, however, because the value stored inside an interface has no address. When assigning a value to an interface, the compiler ensures that all possible interface methods can actually be called on that value, and thus trying to make an improper assignment will fail on compilation. To extend the earlier example, the following describes what is valid and what is not:

```
type List []int

func (l List) Len() int        { return len(l) }
func (l *List) Append(val int) { *l = append(*l, val) }

type Appender interface {
    Append(int)
}

func CountInto(a Appender, start, end int) {
    for i := start; i <= end; i++ {
        a.Append(i)
```

```
    }
}

type Lener interface {
    Len() int
}

func LongEnough(l Lener) bool {
    return l.Len()*10 > 42
}

func main() {
    // A bare value
    var lst List
    CountInto(lst, 1, 10) // INVALID: Append has a pointer receiver
    if LongEnough(lst) {  // VALID: Identical receiver type
        fmt.Printf(" - lst is long enough")
    }

    // A pointer value
    plst := new(List)
    CountInto(plst, 1, 10) // VALID: Identical receiver type
    if LongEnough(plst) {  // VALID: a *List can be dereferenced for the receiver
        fmt.Printf(" - plst is long enough")
    }
}
```