Implementation of Interfaces

This lesson covers the details about how to test the implementation of interfaces.

WE'LL COVER THE FOLLOWING

- Testing if a value implements an interface
- Using method sets with interfaces

Testing if a value implements an interface

This is a special case of the type assertion: suppose `v` is a value, and we want to test whether it implements the `Stringer` interface. This can be done as follows:

```
type Stringer interface { String() string }
if sv, ok := v.(Stringer); ok {
  fmt.Printf("v implements String(): %s\n", sv.String()); // note: sv, no
t v
}
```

An interface is a kind of *contract*, which the implementing type(s) must fulfill. Interfaces describe the *behavior* of types, specifying what types can do. They completely separate the definition of what an object can do from how it does it, allowing distinct implementations to be represented at different times by the same interface variable, which is what polymorphism essentially is. Writing functions so that they accept an interface variable as a parameter makes them more general.

> **Note:** Use interfaces to make your code more generally applicable.

This is also ubiquitously applied in the code of the standard library. It is impossible to understand how it is built without a good grasp of the interfaceconcept.

Using method sets with interfaces

In Chapter 8, we saw that methods on variables do not distinguish between values or pointers. When storing a value in an interface type, it is slightly more complicated because a concrete value stored in an interface is not addressable. Still, luckily the compiler flags an error on improper use. Consider the following program:

```go
package main
import (
"fmt"
)

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
    // compiler error:
    // cannot use lst (type List) as type Appender in function argument:
    // List does not implement Appender (Append method requires pointer receiver)
    // CountInto(lst, 1, 10)
    if LongEnough(lst) { // VALID: Identical receiver type
        fmt.Printf(" - lst is long enough\n")
    }
    // 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\n") // - plst2 is long enough
    }
}
```

In the above code at **line 6**, we define a type for a slice of *ints* as `List`. At **line 8**, we make a method that returns the *length* of a list of type `List`. Then, at **line 10**, we make another method `Append()` that appends a value `val` in a list with a pointer to that list as receiver.

At **line 12**, we have an interface `Appender` with one method `Append(int)`, implemented by `List` type. Then, at **line 16**, we have a function `CountInto` that takes an `Appender` type object `a` and the `start` and `end` value to append values ranging from `start` to `end` with a *for* loop.

At **line 22**, we have an interface `Lener` with one method `Len()`, implemented by `List` type. Then, at **line 26**, we have a function `longEnough` that takes an `Lener` type object `l` and returns *true* if length of `l` multiplied by **10** is greater than **42**. Otherwise, it returns *false*.

Now, look at the `main`. We make a `List` variable `lst` at **line 32**. At **line 37**, we call the `LongEnough` function on `lst`. In this case, the function returns *false*(**0*10>42**), so control will directly transfer to **line 41**. At **line 41**, we make a `List` variable `plst` using the `new()` function and call the `CountInto` function with `plst` as an `Appender`, `start` as **1** and `end` as **10**. Now, the length of `plst` is **10**. In the next line, we call the `LongEnough` function on `plst`. In this case, the function returns *true*(**10*10>42**), so control will directly transfer to **line 44**, and the message will be printed on screen.

`CountInto` called with the value `lst` gives a compiler error because `CountInto` takes an Appender, and `Append()` is only defined for a pointer. `LongEnough` on value `lst` works because `Len()` is defined on a value. `CountInto` called with the pointer `plst` works because `CountInto` takes an Appender, and `Append()` is defined for a pointer. `LongEnough` on pointer `plst` works because a pointer can be dereferenced for the receiver.

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 methods can be called with pointers.

- Value methods can be called with values.

- 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 be called on that value, so trying to make an improper assignment will fail on the compilation.

Implementing all the methods of an interface is a must. Otherwise, the compilation fails. The next lesson shows the implementation of an interface and demonstrates how to use an interface to sort some data.