Sorting with the Sorter Interface

This lesson provides a detailed code and explanation of an interface designed in a separate package and implemented in the main package.

Sorting the data

An excellent example comes from the Go standard library itself, namely the package `sort` (look up the documentation of this package here).

To sort a collection of numbers or strings, you only need the number of elements `Len()`, a way to compare items `i` and `j` with `Less(i, j)` and a method to swap items with indexes `i` and `j` with `Swap(i, j)`. The Sort-function in sort has an algorithm that only uses these methods on a collection data (to implement it we use here a bubble sort, but any sort-algorithm could be used):

```go
func Sort(data Sorter) {
  for pass:=1; pass < data.Len(); pass++ {
    for i:=0; i < data.Len() - pass; i++ {
      if data.Less(i+1, i) {
        data.Swap(i, i+1)
      }
    }
  }
}
```

`Sort` can accept a general parameter of an interface type `Interface`, which declares these methods:

```go
type Interface interface {
  Len() int
  Less(i, j int) bool


  Swap(i, j int)
}
```

The type *int* in `Interface` does not mean that the collected data must contain ints, `i` and `j` are integer indices, and the length is also an integer. Note that a type implementing the methods of `Interface` can be a collection of data of any type.

Now, if we want to be able to sort an array of ints, all we must do is to define a type and implement the methods of `Interface`. For example, for a type `IntSlice` the package defines:

```
type IntSlice []int
func (p IntSlice) Len() int { return len(p) }
func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }
func (p IntSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }
```

Here, is the code to call the sort-functionality in a concrete case:

```
data := []int{74, 59, 238, -784, 9845, 959, 905, 0, 0, 42, 7586, -5467984,
 7586}
a := sort.IntSlice(data) //conversion to type IntSlice from package sort
sort.Sort(a)
```

Environment Variables

Key:	Value:
GOROOT	/usr/local/go
GOPATH	//root/usr/local/go/src
PATH	//root/usr/local/go/src/bin:/usr/local/go...

```
package mysort

type Interface interface {
    Len() int
    Less(i, j int) bool
    Swap(i, j int)
}

func Sort(data Interface) {
    for pass:=1; pass < data.Len(); pass++ {
        for i:=0; i < data.Len() - pass; i++ {
            if data.Less(i+1, i) {
                data.Swap(i, i+1)
            }
        }

    }
}

func IsSorted(data Interface) bool {
    n := data.Len()
    for i := n - 1; i > 0; i-- {
        if data.Less(i, i-1) {
            return false
        }
    }
    return true
}

// Convenience types for common cases
type IntSlice []int

func (p IntSlice) Len() int { return len(p) }

func (p IntSlice) Less(i, j int) bool { return p[i] < p[j] }

func (p IntSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

type StringSlice []string

func (p StringSlice) Len() int { return len(p) }


func (p StringSlice) Less(i, j int) bool { return p[i] < p[j] }

func (p StringSlice) Swap(i, j int) { p[i], p[j] = p[j], p[i] }

// Convenience wrappers for common cases
func SortInts(a []int) { Sort(IntSlice(a)) }

func SortStrings(a []string) { Sort(StringSlice(a)) }

func IntsAreSorted(a []int) bool { return IsSorted(IntSlice(a)) }

func StringsAreSorted(a []string) bool { return IsSorted(StringSlice(a)) }
```

In the file **mysort.go**:

- We make an interface `Interface` with three methods `Len()`, `Less()` and `Swap()`, as discussed above. At **line 9**, we have a function header of `Sort` that takes `Interface` as a parameter, and sorts the data of a collection using the `Len()` , `Less()` and `Swap()` functions.
- Then, at **line 19**, we have a function header of `IsSorted` that takes `Interface` as a parameter and returns *true* if the data is sorted. Otherwise, false. At **line 30**, we define a type `IntSlice` for []int.
- From **line 32** to **line 36**, we implement the methods `Len()`, `Less()` and

`Swap()` for slice of ints. At **line 38**, we define a type `StringSlice` for

[]string.

- From **line 40** to **line 45**, we implement the methods `Len()`, `Less()` and `Swap()` for slices of strings. Next, we make wrappers of two functions `Sort()` and `IsSorted()` each for `IntSlice` and `StringSlice` from **line 48** to **line 54**.

Now, look at the **main.go** file. Before studying `main`, let's see three basic functions, `ints()`, `strings()`, and `days()`:

- Look at the header of function `ints()` at **line 9**. We make a slice of integers `data` and convert it to type `IntSlice` by importing: `mysort.IntSlice()` at **line 11**, and saving it in `a`. Then, we call the `Sort` function on `a`. At **line 13**, we check whether `a` is sorted or not by calling `IsSorted` function on `a`. If `a` is not sorted, control will transfer to **line 16**, and the sorted slice will be printed. If, sorted then **line 14** will be executed.

- Now, look at the header of function `strings()` at **line 20**. We make a slice of strings `data` and convert it to type `StringSlice` by importing it as: `mysort.StringSlice()` at **line 22**, and saving it in `a`. Then, we call the `Sort` function on `a`. At **line 24**, we check whether `a` is sorted or not by calling the `IsSorted` function on `a`. If `a` is not sorted, control will transfer to **line 27**, and the sorted slice will be printed. If sorted, then **line 25** will be executed.

- Now, before studying the `days()` function, look at the definition of `day` type struct at **line 31**. It has *three* fields: one `num` and integer field and two strings fields `shortName` and `longName`. Then, at **line 37**, we make another struct of type `dayArray` with one field `data` with a type of pointer to slice of type `day`. We implement `Len()`, `Less()` and `Swap()` for `dayArray` type (from **line41** to **line 43**), just like we did for `IntSlice` and `StringSlice` in **mysort.go**.

- Now, look at the header of function `days()` at **line 46**. At **line 47**, we make a `day` type variable and assign **0** to `num`, **SUN** to `shortName` and **Sunday** to `longName`. Then, we make further *six* `day` type variables each for a day (from **line 48** to **line 53**). Next, we make a slice of these *seven* `day` type variables and convert it into `dayArray` type and store it in `a`. At **line 56**,

we check whether `a` is sorted or not by calling the `IsSorted` function on `a`. If `a` is not sorted, control will transfer to **line 60**, and the sorted slice will be printed using `longName`. If sorted, then **line 58** will be executed.

Now, look at `main`. We are just calling `ints()`, `strings()` and `days()` function at **line 67**, **line 68** and **line 69**, respectively. At the end, sorted arrays are shown in output on the screen.

> **Remarks:** The `panic("fail")` is a way to stop the program in a situation that could not occur in common circumstances; we could also have printed a message and then used `os.Exit(1)`. This code can also be run with the `sort` package from Go's standard library. Replace `"./mysort"` with `"sort"` in main.go, and replace all the calls to `mysort.` with `sort.`

This example has given us a better insight into the significance and use of interfaces. For sorting primitive types, we know that we don't have to write this code ourselves. The standard library provides for this. The `Interface` type can be used in implementing sorting for other kinds of data. In fact, that is precisely what we did in the example above, using this for ints and strings, but also for a user-defined type `dayArray`, to sort an internal array of strings.

Interfaces make it possible to write generic programs such as the one in this lesson that can sort data of any type. There is a challenge in the next lesson for you to solve.