Structs, Collections and Higher-Order Functions

This lesson is an implementation of the example that covers the concepts of structs, interfaces, and higher-order functions studied so far.

Often, when you have a struct in your application, you also need a collection of (pointers to) objects of that struct, like:

```go
type Any interface{}

type Car struct {
  Model string
  Manufacturer string
  BuildYear int
  // ...
}

type Cars []*Car
```

We can then use the fact that higher-order functions can be arguments to other functions when defining the needed functionality, e.g.:

1. When defining a general `Process()` function, which itself takes a function `f` which operates on every car:

```go
// Process all cars with the given function f:
func (cs Cars) Process(f func(car *Car)) {
  for _, c := range cs {
    f(c)
  }
}
```

2. Building upon this, make *Find-functions* to obtain subsets, and call `Process()` with a closure (so it knows the local slice cars):

```go
// Find all cars matching given criteria.
func (cs Cars) FindAll(f func(car *Car) bool) Cars {
  cars := make([]*Car, 0)
  cs.Process(func(c *Car) {
    if f(c) {
      append(cars,c)
    }
  })
  return cars
}
```

3. And make a *Map-functionality* producing something out of every car object:

```
// Process cars and create new data.
func (cs Cars) Map(f func(car *Car) Any) []Any {
  result := make([]Any, 0)
  ix := 0
  cs.Process(func(c *Car) {
    result[ix] = f(c)
    ix++
  })
  return result
}
```

Now, we can define concrete queries like:

```
allNewBMWs := allCars.FindAll(func(car *Car) bool {
  return (car.Manufacturer == "BMW") && (car.BuildYear > 2010)
})
```

4. We can also return functions based on arguments. Maybe we would like to append cars to collections based on the manufacturers, but those may be varying. So, we define a function to create a special append function as well as a map of collections:

```
func MakeSortedAppender(manufacturers []string) (func(car *Car), map[strin
g]Cars) {
  // Prepare maps of sorted cars.
  sortedCars := make(map[string]Cars)
  for _, m := range manufacturers {
    sortedCars[m] = make([]*Car, 0)
  }

  sortedCars["Default"] = make([]*Car, 0)

  // Prepare appender function:
```

```
  // Prepare appender function:
  appender := func(c *Car) {
    if _, ok := sortedCars[c.Manufacturer]; ok {
      sortedCars[c.Manufacturer] = append(sortedCars[c.Manufacturer], c)
    } else {
      sortedCars["Default"] = append(sortedCars["Default"], c)
    }
  }
  return appender, sortedCars
}
```

We now can use it to sort our cars into individual collections, like in:

```
manufacturers := []string{"Ford", "Aston Martin", "Land Rover", "BMW", "Jaguar"}
sortedAppender, sortedCars := MakeSortedAppender(manufacturers)
allUnsortedCars.Process(sortedAppender)
BMWCount := len(sortedCars["BMW"])
```

Now that you are familiar with a lot of concepts related to interfaces, the next lesson brings you a challenge to solve.