Lecture Overview

Methods and Interfaces

- **Methods review**
- **Interfaces**
- **Example: using the sort interface**
- **Anonymous fields in structs**
- **Generic printing using the empty interface**

Maps

- **Creating a map**
- **Accessing elements of a map**
- **Missing keys**
- **Deleting elements of a map**
- **Limitations on keys**
- **Processing a map with a for-loop**
- **Sample program**

Methods and Interfaces

One of the novel features of Go is its use and implementation of interfaces. They provide many of the same benefits of object-oriented programming, but without explicit classes as in C++ or Java.

Methods

Recall that a **method** is a special kind of Go function. Consider this code:

```
type Rectangle struct {
   width, height float64
}

// area() is a method
func (r Rectangle) area() float64 {
    return r.width * r.height
}

// perimeter() is a method
func (r Rectangle) perimeter() float64 {
    return 2 * (r.width + r.height)
}
```

```
// main is a function
func main() {
   r := Rectangle{width:5, height:3}
   fmt.Printf("dimensions of r: width=%d,height=%d\n",
              r.width, r.height)
   fmt.Printf("      area of r: %.2f\n", r.area())
   fmt.Printf(" perimeter of r: %.2f\n",
              r.perimeter())
}
```

We know `area()` and `perimeter()` are **methods** because they have a special parameter written in brackets before their name. Essentially, this lets us pass in one value to the function in a special way.

In `main`, notice how the usual dot-notation is used for calling methods, e.g. `r.area()` calls the area method on `r`.

Often, the objects are passed to methods by reference. For example:

```
func (r *Rectangle) inflate(scale float64) {
    r.width *= scale
    r.height *= scale
}
```

The `inflate` method does *not* make a copy of the rectangle sent to it. Instead, it passes `r` by reference so that `r` is actually changed. Note that there is no change in the syntax for how the fields of `r` are accessed: the regular dot-notation is used. Similarly, `inflate` is called the same way, e.g. `r.inflate(2.2)`. There is no special `->` operator as in C/C++.

Interfaces

When you first see Go methods, you might ask yourself why one parameter is singled-out as special. Why not just pass it along with the other parameters? For example, what is the difference between these two functions:

```
// called as r.area()
func (r Rectangle) area() float64 {
    return r.width * r.height
}

// called as area(r)
func area(r Rectangle) float64 {
    return r.width * r.height
}
```

The only difference here is the syntax, e.g. calling `r.area()` versus `area(r)` for the regular function. However, syntax is not the main reason Go uses methods.

Go has methods to allow for interfaces. For example:

```
type Shape interface {
    area() float64
    perimeter() float64
}
```

The name of this interface is `Shape`, and it lists the signatures of the two methods that are necessary to satisfy it. For example, the `area` and `perimeter` methods we wrote above for `Rectangle` satisfy this interface. We say that a `Rectangle` *implements* the `Shape` interface.

Notice that only the *signatures* of the methods are listed in the interface. The bodies of the required functions are not mentioned at all. The implementation is *not* part of the interface.

Methods with those signatures are considered to be an instance of `Shape`. For example, suppose we add this code:

```
type Circle struct {
    radius float64
}

func (c Circle) area() float64 {  // a method
    return 3.14 * c.radius * c.radius
}

func (c Circle) perimeter() float64 {  // a method
    return 2 * 3.14 * c.radius
}
```

`Circle` objects implement the `Shape` interface because of they have `area` and `perimeter` methods associated with them.

Now we can write code that works on any value implementing an interface. For example:

```
func printShape(s Shape) {
    fmt.Printf("      area: %.2f\n", s.area())
    fmt.Printf(" perimeter: %.2f\n", s.perimeter())
}
```

The input to `printShape` is of type `Shape`, i.e. `s` is *any* object that implements the `Shape` interface. We can call it like this:

```
func main() {
   r := Rectangle{width:5, height:3}
   fmt.Println("Rectangle ...")
   printShape(r)

   c := Circle{radius:5}
   fmt.Println("\nCircle ...")
   printShape(c)
}
```

An interesting detail about Go interfaces is that you don't need to tell Go that a `struct` implements a particular interface: the compiler figures it out for itself. This contrasts with, for example, Java, where you must explicitly indicate when a class implements an interface.

Example: Using the Sort Interface

Lets see how we can use the standard Go sorting package.

Suppose we want to sort records of people. In practice, such records might contain lots of information (such as a person's name, address, email address, relatives, etc.), but for simplicity we will use the following basic structure called `Person`:

```
type Person struct {
    name string
    age int
}
```

For efficiency, lets sort a slice of pointers to `Person` objects; this will avoid moving and copying strings. To help with this, we define the type `People` as follows:

```
type People []*Person  // slice of ptrs to People objs
```

It turns out that it's essential that we create the type `People`. The code we write below won't compile if we directly use `[]*Person` instead. That's because the `[]*Person` type does *not* satisfy the `sort.Interface` interface we'll be using. We will add methods on `People` that make it implement `sort.Interface`.

For convenience, here's a `String` method that prints a `People` object:

```
func (people People) String() (result string) {
   for _, p := range people {
      result += fmt.Sprintf("%s, %d\n", p.name, p.age)
   }
   return result
}
```

The name of this is `String()`, and it returns a `string` object. A method with this signature is special: print functions in the `fmt` package will use it for printing.

Now we can write code like this:

```
users := People{
            {"Mary", 34},
            {"Lou", 3},
            {"Greedo", 77},
            {"Zippy", 50},
            {"Morla", 62},
}

fmt.Printf("%v\n", users)  // calls the People String
method
```

To sort the items in the `users` slice, we must create the methods listed in `sort.Interface`:

```
type Interface interface {
        // number of elements in the collection
        Len() int

        // returns true iff the element with
        // index i should come
        // before the element with index j
        Less(i, j int) bool

        // swaps the elements with indexes i and j
        Swap(i, j int)
}
```

This interface is pre-defined in the `sort` package. Notice that this is a very general interface. It does not even assume that you will be sorting slices or arrays!

Three methods are needed:

```
func (p People) Len() int {
    return len(p)
}

func (p People) Less(i, j int) bool {
    return p[i].age > p[j].age
}
```

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

`Less` is the function that controls the order in which the objects will be sorted. By examining `Less` you can see that we will be sorting people by age, from oldest to youngest.

With these functions written, we can now sort `users` like this:

```
users := People{
            {"Mary", 34},
            {"Lou", 3},
            {"Greedo", 77},
            {"Zippy", 50},
            {"Morla", 62},
}

fmt.Printf("%v\n", users)

sort.Sort(users)

fmt.Printf("%v\n", users)
```

To change the sort order, modify `Less`. For instance, this will sort `users` alphabetically by name:

```
func (p People) Less(i, j int) bool {
    return p[i].name < p[j].name
}
```

Another way to sort by different orders is shown in the examples section of the Go sort package documentation. The trick there is to create a new type for every different order you want to sort.

Anonymous Fields in structs

Go lets you create new structs built from previously defined structs. For example:

```
type Point struct {
  x, y int
}

type Color struct {
  red, green, blue uint8  // each ranges from 0 to 255
}

type ColoredPoint struct {
  Point   // these two fields don't have names;
  Color   // they are anonymous
}
```

`ColoredPoint has two different fields, but neither has a name: they are anonymous. We can use a ``ColoredPoint` like this:

```
cp := ColoredPoint{Point{10, 5}, Color{120, 0, 0}}

fmt.Println(cp.x)
fmt.Println(cp.red)
```

This is an example of **composition**: a `ColoredPoint` is composed (i.e. made up of) two other objects.

We won't go any further into any of the details of this idea. However, it is worth mentioning that it can, among other things, essentially simulate inheritance as done in languages like Java and C++.

A Generic Printing Function with the Empty Interface

One of the most important interfaces in Go is the **empty interface**, which has type `interface{}`. `interface{}` means that a type has 0 or more methods associated with, thus *all* types implement it. This means you can use `interface{}` to pass values of *any* type.

For example, suppose we have this interface:

```
type Displayer interface {
    toString() string
}
```

Then we can write a function called `display` that is similar in spirit to `fmt.Print`:

```
// x can be a value of any type --- all types
// implement the empty interface
func display(x interface{}) {
    switch val := x.(type) {
    case string:
        fmt.Println("\"" + val + "\"")
    case int, int32, int64, float32, float64:
        fmt.Println(val)
    case Displayer:
        fmt.Println(val.toString())
    default:
        fmt.Println("can't display: unknown type!")
    }
}
```

The `switch` structure in this function is called a **type switch** because it does different things depending upon the type of `x`. It lets us write code like this:

```
type Point3d struct {
    x, y, z float32
}

func (p Point3d) toString() string {
    return fmt.Sprintf("(%v, %v, %v)", p.x, p.y, p.z)
}

func main() {
    display(3.55)
    display("apple")
    display(Point3d{2.3, -4.2, 3})
}
```

Any type that implements the `Displayer` interface can be printed in a reasonable way by `display`.

Here is the entire program:

```
package main

import (
    "fmt"
)

type Displayer interface {
    toString() string
}
```

```
// x can be a value of any type ---
// all types implement the empty interface
func display(x interface{}) {
    switch val := x.(type) {
    case string:
        fmt.Println("\"" + val + "\"")
    case int, int32, int64, float32, float64:
        fmt.Println(val)
    case Displayer:
        fmt.Println(val.toString())
    default:
        fmt.Println("can't display: unknown type!")
    }
}

type Point3d struct {
    x, y, z float32
}

func (p Point3d) toString() string {
    return fmt.Sprintf("(%v, %v, %v)", p.x, p.y, p.z)
}

func main() {
    display(3.55)
    display("apple")
    display(Point3d{2.3, -4.2, 3})
}
```

Maps

Maps are a very useful data structure that store (key, value) pairs in a way that lets you efficiently retrieve any pair if you know its key.

Creating a Map

Here is a map that stores the names of candidates for an election and the number of votes they've received so far:

```
votes := map[string] int {"Yan":4, "Jones":2,
                          "White":2}
```

On the right side of `:=` is a map literal. Its type is `map[string]int`, and the map itself is specified using *key*:*value* pairs (similar to the notation used in languages like Python, JavaScript, and JSON).

You can also create a map using the `make` function. Here is an alternative way to create the `votes` map:

```
votes := make(map[string]int)
                          // creates an empty map
votes["Yan"] = 4
votes["Jones"] = 2
votes["White"] = 2
```

Accessing Elements of a Map

You access a particular value using a key, e.g. `votes["yan"]` evaluates to 4. If you want to add 1 vote for Jones, you can do it like this:

```
votes["Jones"]++  // add 1 to the value associated
with "Jones"
```

To add a new item to the map, assign it like this:

```
votes["Harper"] = 3
```

If the key `"Harper"` already happened to be in `votes`, then this statement would just set its value to 3.

Missing Keys

If you try to access the value for a key that doesn't exist, then the zero- value associated with the value's type is returned. For example:

```
fmt.Println(votes["Kennedy"]) // prints 0, because
                              // "Kennedy" is not a key
```

This presents a problem: how can you distinguish between a key that doesn't exist, and a key that is paired with 0? The solution Go provides is to return an optional flag indicating whether or not the key was found:

```
k, ok := votes["Kennedy"]
if ok {
        fmt.Printf("%v\n", k)
} else {
        fmt.Println("no candidate by that name")
}
```

It’s entirely up to the programmer to check this flag!

A common use of this is to test if a given key is in a map. For instance:

```
_, present := votes["Kennedy"]
       // _ is the blank identifier; we use it
       // here because we don't care about the
       // associated value
```

If `present` is `true`, they `"Kennedy"` is a key in the list. If it’s `false`, then `Kennedy` is not in the list.

Deleting Items in a Map

To delete a key and its associated value from a map, use the built-in `delete` function:

```
delete(votes, "Yan")
```

If `"Yan"` happened to not be a key in `votes`, then this statement doesn’t modify the map.

Limitations on Keys

Not all data types are allowed to be keys in a map. Any data type that supports equality, such as integers, floats, strings, pointers, structs, and arrays **can** be used as a key. But, slices and other maps **cannot** be keys because they do not have equality defined for them.

Processing a Map with a For-loop

It's easy to process every element of a map using a ranged for-loop:

```
for key, value := range votes {
        fmt.Printf("votes[\"%s\"] = %d\n", key, value)
}
```

Or if you just want the keys:

```
for key := range votes {
        fmt.Printf("votes[\"%s\"] = %d\n", key,
votes[key])
}
```

Questions

1. What is the type of a map whose keys are integers and whose values are booleans.
2. What are two different data types that *cannot* be used as keys in a map?
3. Can `nil` be a key in a map? If no, why not? If yes, then what is the type for a map that can have `nil` as a key?
4. Write a function that takes a map of type `map[string]int` as input, and returns the key *and* value of the pair with the greatest key.
5. Write a function that takes a map of type `map[string]int` as input along with a target string `val`, and returns a slice containing all the keys whose value equals `val`.

Sample Program

The following program is based on an idea from the XKCD comic strip. It asks the user to type in some words, and then it checks to see which of those words are in `xkcdWordlist.txt`, a file of the 3000 or so most common English words.

A map is a good data structure for this problem because testing if a word is in it can be done in O(1) time, on average. If, instead, you used a slice, then the retrieval would take O(n) time on average.

```
package main

import (
    "bufio"
    "fmt"
    "io/ioutil"
    "os"
    "strings"
)

func main() {
    //
    // load all the words into a map
    //
    var dictionary map[string]bool =
              make(map[string]bool)

    // read entire file into one slice of bytes
    allBytes, err :=
              ioutil.ReadFile("xkcdWordlist.txt")
    if err != nil {
         panic("no word file to read!")
    }

    // convert the byte slice to a string
    bigString := string(allBytes)

    // split the string into words
    words := strings.Split(bigString, " ")

    // add the words to the dictionary
    for _, w := range words {
        dictionary[w] = true
    }
```

```
    fmt.Printf("%v words in dictionary\n",
              len(dictionary))

    //
    // check words typed by the user
    //
    console := bufio.NewReader(os.Stdin)
    for {
        // print a prompt
        fmt.Print("--> ")

        // read, as a slice of bytes, the entire
        // line of text entered by the user
        lineBytes, _, _ := console.ReadLine()
        // fmt.Printf("(input=\"%v\")\n", lineBytes)

        // convert the line to a string
        line := string(lineBytes)

        // split the string into words
        userWords := strings.Split(line, " ")
        // fmt.Printf("(%v)\n", userWords)

        // check each word to see if it is in
        // the dictionary
        for _, w := range userWords {
            if _, exists := dictionary[w]; !exists {
               fmt.Printf("\"%v\" is too complex!\n",
                             w)
            }
        }
    } // for
}
```