Programming Paradigms CSI2120 – Winter 2018

Jochen Lang
EECS, University of Ottawa
Canada

Université d'Ottawa | University of Ottawa

uOttawa

L'Université canadienne
Canada's university

uOttawa.ca

System Programming: Go

- **Stream I/O**
  - Console I/O
  - File I/O
- **Panic and Recover**
- **Methods**
- **Interfaces**
- **Embedded Types**

Console I/O

- **Console I/O with package fmt. C-like syntax with scanf and printf in addition to print (println).**

```
package main
import "fmt"

func main() {
    var inStr string
    fmt.Printf("Input? ")
    fmt.Scanf("%s", &inStr)
    fmt.Printf("\nOutput: %s\n", inStr)
}
```

File I/O

- **File I/O follows familiar notion of stream I/O**
- **Relevant packages are os and bufio besides fmt**

```
func fCopy(outN, inN string) (bytesOut int64, err error) {
    inF, err := os.Open(inN)
    if err != nil {
        return
    }
    outF, err := os.Create(outN)
    if err != nil {
        inF.Close()
        return
    }
    bytesOut, err = io.Copy(outF, inF)
    inF.Close(); outF.Close()
    return
}
```

Final Evaluation with `defer`

- **With `defer` the execution of a statement can be deferred to the end of a function or block.**
- **Convenient for cleaning up at the end of multiple control path through a function.**
  - E.g., file handles
- **The parameters of a delayed function are evaluated at the issue of the defer statement.**
- **The execution of a deferred function takes place even if the function is exited with an error**

File Copy with Deferred Clean-up

- **Defer simplifies error handling**

```
func fCopy(outN, inN string) (bytesOut int64, err error) {
    inF, err := os.Open(inN)
    if err != nil {
        return
    }
    defer inF.Close()
    outF, err := os.Create(outN)
    if err != nil {
        return
    }
    defer outF.Close()
    bytesOut, err = io.Copy(outF, inF)
    return
}
```

Errors and Panic in Go

- **No exceptions in Go**
- **Instead of exception return error codes of type error**
  - Convention: On success error is `nil`
- **When a serious error occurs, use the `panic` statement**
  - Only to be used for unforeseen errors
  - Corresponds to violations of assertions (C assert statement)
- **In case of panic:**
  - function stops immediately
  - deferred functions are executed
  - stack unwinding occurs
    - return to the calling functions
    - executing their deferred functions until main exits
    - can be stopped with `recover`

Panic Toy Example

```
package main
import "fmt"
func main() {
    defer fmt.Println("Last Output")
    fmt.Println("Before Panic" )
    panic("Out of here")
    fmt.Println("After Panic")
}

Before Panic
Last Output
panic: Out of here

goroutine 1 [running]:
runtime.panic(0x48d4e0, 0xc084005260)
     C:/Users/ADMINI~1/AppData/Local/Temp/2/makerelease250988475/go
/src/pkg/runtime/panic.c:266 +0xc8
main.main()
     c:/teaching/CSI2120/demoCode/go/panictoy.go:8 +0x18e
```

Panic and Recover

```
package main
import "fmt"
func causePanic() {
        panic("Caused panic")
}
func foo() {
        defer func() {
                if err := recover(); err != nil {
                        fmt.Printf("Recovered from >> %s <<\n", err)
                }
        }()
        causePanic()
        fmt.Println("Regular after panic")  // not shown
}
func main() {
        fmt.Println("In main:")
        foo()
        fmt.Println("End main:")
}
```

Example Run Recover and Go

- **Program terminates regularly**
- **Note that function foo is exited through a deferred function call**

- **Output:**

```
In main:
Recovered from >> Caused panic <<
End main:
```

Methods and Receivers

- **A method is a function acts on a certain type (its receiver)**
- **Receiver type can be almost anything**
  - But no interfaces, no pointer type (but a pointer to an allowed type)
  - Often a structure is the receiver
- **The structure and methods are not grouped together**
  - they only have to be in the same package (but can be in different source files)
  - no encapsulation as with classes
- **No method overloading (just as with functions)**

Definition of and Calling a Method

```
package main
import (
        "fmt"
        "math"
)
type Point struct {
        x float64
        y float64
}
func (pt *Point) norm() float64 {
        return math.Sqrt(pt.x*pt.x + pt.y*pt.y)
}
func main() {
        a := Point{2.,4.}
        n := a.norm()
        fmt.Printf("2-Norm = %f\n", n)
}
```

Interfaces

- **Interfaces define a set of methods**
  - no code for the method
  - abstract definitions
- **Naming convention**
  - Interface name should end in “er”, e.g., Writer, Reader, Logger etc.
- **Implementing an interface**
  - A type does not need to state that implements an interface
  - A type can implement multiple interfaces
  - Interface can embed other interfaces
  - Interfaces can be assigned to variables

Interface example

```
type ColorPt struct {
       pt Point
       color string
}
type Box struct {
       weight float64
       color string
}
// interface implemented by ColorPt and Box
type Color interface {
       SetColor(string)
       Color() string
}
```

Implementation of an Interface

- **Interface implementation for type `ColorPt`**

```
func (p *ColorPt) Color() string {
    return p.color
}
func (p *ColorPt) SetColor(col string) {
    p.color = col
}
```

- **Interface implementation for type `Box`**

```
func (p *Box) SetColor(col string) {
    p.color = col
}
func (p *Box) Color() string {
    return p.color
}
```

Polymorphism with Interfaces

- **Use an array of pointers to structures implementing the `Color` interface**

```
func main() {
    table := […]Color{
          &ColorPt{Point{1.1,2.2},"red"},
          &Box{32.4, "yellow"},
          &ColorPt{Point{3.1,1.2},"blue"}}

    for _,element := range table {
          fmt.Printf("color= %s\n",
                      element.Color())
    }
}
```

Embedded Types

- **We can embed a type in a structure**
- **The new type contains the embedded type (this is not inheritance but can be used similarly)**

```
type Person struct {
     lastName  string
     firstName string
}
type Student struct {
     Person
     studentId int
}
func (p *Person) print() {
     fmt.Printf("%s, %s", p.lastName, p.firstName)
}
func (s *Student) print() {
     s.Person.print()
     fmt.Printf(": %d ", s.studentId)
}
```

Calling Methods for the Embedded Type

- **The new structure containing the embedded type can be a receiver for methods of the embedded type**

```
type Person struct {
     lastName  string
     firstName string
     dob time.Time
}
func main() {
     s := Student{Person{"Joe", "Smith",
              time.Date(1995,4,7,0,0,0,0,time.UTC)}, 101 }
     s.print()
     s.age()
}
func (p *Person) age() {
     … // omitted here
}
```

Summary

- **Stream I/O**
  - Console I/O
  - File I/O
- **Panic and Recover**
- **Methods**
- **Interfaces**
- **Embedded types**