Unit:3
Error Handling

Introduction

- Errors are a language-agnostic part that helps to write code in such a way that no unexpected thing happens
- When something occurs which is not supported by any means then an error occurs
- Errors help to write clean code that increases the maintainability of the program

What is an error?

- An error is a well developed abstract concept which occurs when an exception happens
- That is whenever something unexpected happens an error is thrown
- Errors are common in every language which basically means it is a concept in the realm of programming

Why do we need Error?

- Errors are a part of any program
- An error tells if something unexpected happens
- Errors also help maintain code stability and maintainability
- Without errors,
  - the programs we use today will be extremely buggy due to a lack of testing

Errors in GoLang

- GoLang has support for errors in a really simple way
- Go functions returns errors as a second return value
- That is the standard way of implementing and using errors in Go
- If you have written any Go code you have probably encountered the built-in error type
- Go code uses error values to indicate an abnormal state
- For example, the os.Open function returns a non-nil error value when it fails to open a file

Errors in GoLang

- func Open(name string) (file *File, err error)
- The following code uses os.Open to open a file
- If an error occurs it calls log.Fatal to print the error message and stop

Execute | Share main.go STDIN

```go
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
// do something with the open *File f
```

The error type

- The error type is an interface type
- An error variable represents any value that can describe itself as a string
- Here is the interface's declaration:

```
type error interface {
    Error() string
}
```

- The error type, as with all built in types, is predeclared in the universe block

The error type

- The most commonly-used error implementation is the errors package's unexported errorString type

```
// errorString is a trivial implementation of error
type errorString struct {
    s string
}
func (e *errorString) Error() string {
    return e.s
}
```

Simple Error Methods

- There are multiple methods for creating errors
- **Using the New function**
- GoLang errors **package** has a function called **New()** which can be used to create errors easily
- Below it is in action

Simple Error Methods

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "errors"
)

func e(v int) (int, error) {
    if v == 0 {
        return 0, errors.New("Zero cannot be used")
    } else {
        return 2*v, nil
    }
}

func main() {
    v, err := e(0)

    if err != nil {
        fmt.Println(err, v)      // Zero cannot be used 0
    }
}
```

Result

```
$go run main.go
Zero cannot be used 0
```

Simple Error Methods

- **Using the Errorf function**
- The fmt package has an Errorf() method that allows formatted errors as shown below
  - `fmt.Errorf("Error: Zero not allowed! %v", v)` `// Error: Zero not allowed! 0`

Checking for an Error

- To check for an error we simply get the second value of the function and then check the value with the nil
- Since the **zero value of an error is nil**
- So, we check if an error is a nil
- If it is then no error has occurred and all other cases the error has occurred

Checking for an Error

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "errors"
)

func e(v int) (int, error) {
    return 42, errors.New("42 is unexpected!")
}

func main() {
    _, err := e(0)

    if err != nil {   // check error here
        fmt.Println(err)      // 42 is unexpected!
    }
}
```

Result

```
$go run main.go
42 is unexpected!
```

Panic and recover

- Panic occurs when an unexpected wrong thing happens
- It stops the function execution
- Recover is the opposite of it
- It allows us to recover the execution from stopping
- Below shown code illustrates the concept

Panic and recover

Execute | Share | main.go | STDIN

```go
package main

import (
    "fmt"
)

func f(s string) {
    panic(s)      // throws panic
}

func main() {
        // defer makes the function run at the end
    defer func() {      // recovers panic
        if e := recover(); e != nil {
                fmt.Println("Recovered from panic")
            }
    }()

    f("Panic occurs!!!") // throws panic

    // output:
    // Recovered from panic
}
```

Result

```
$go run main.go
Recovered from panic
```

Creating custom errors

- As we have seen earlier the function errors.New() and fmt.Errorf() both can be used to create new errors
- But there is another way we can do that
  - And that is implementing the error interface

Execute | Share main.go STDIN

```go
type CustomError struct {
    data string
}

func (e *CustomError) Error() string {
    return fmt.Sprintf("Error occured due to... %s", e.data)
}
```

Returning error alongside values

- Returning errors are pretty easy in Go
- Go supports multiple return values
- So we can return any value and error both at the same time and then check the error
- Here is a way to do that

Returning error alongside values

Execute | Share main.go STDIN

```go
import (
    "fmt"
    "errors"
)

func returnError() (int, error) {  // declare return type here
    return 42, errors.New("Error occured!")  // return it here
}

func main() {
    v, e := returnError()
    if e != nil {
        fmt.Println(e, v)  // Error occured! 42
    }
}
```

Ignoring errors in GoLang

- Go has the skip (-) operator which allows skipping returned errors at all
- Simply using the skip operator helps here

Ignoring errors in GoLang

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "errors"
)

func returnError() (int, error) {  // declare return type here
    return 42, errors.New("Error occured!")  // return it here
}

func main() {
    v, _ := returnError()   // skip error with skip operator

    fmt.Println(v)    // 42
}
```

Simplifying repetitive error handling

- In Go, error handling is important
- The language's design and conventions encourage you to explicitly check for errors where they occur (as distinct from the convention in other languages of throwing exceptions and sometimes catching them)
- In some cases this makes Go code verbose, but fortunately there are some techniques you can use to minimize repetitive error handling

Simplifying repetitive error handling

- Consider an App Engine application with an HTTP handler that retrieves a record from the datastore and formats it with a template

Execute | Share main.go STDIN

```go
1  func init() {
2      http.HandleFunc("/view", viewRecord)
3  }
4
5  func viewRecord(w http.ResponseWriter, r *http.Request) {
6      c := appengine.NewContext(r)
7      key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
8      record := new(Record)
9      if err := datastore.Get(c, key, record); err != nil {
10         http.Error(w, err.Error(), 500)
11         return
12     }
13     if err := viewTemplate.Execute(w, record); err != nil {
14         http.Error(w, err.Error(), 500)
15     }
16 }
17
```

Simplifying repetitive error handling

- To reduce the repetition we can define our own HTTP appHandler type that includes an error return value:
  - `type appHandler func(http.ResponseWriter, *http.Request) error`
- Then we can change our viewRecord function to return errors:

Execute | Share | main.go | STDIN

```go
func viewRecord(w http.ResponseWriter, r *http.Request) error {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return err
    }
    return viewTemplate.Execute(w, record)
}
```

Simplifying repetitive error handling

- This is simpler than the original version, but the http package doesn't understand functions that return error
- To fix this we can implement the http.Handler interface's ServeHTTP method on appHandler:

Execute | Share main.go STDIN

```go
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if err := fn(w, r); err != nil {
        http.Error(w, err.Error(), 500)
    }
}
```

Simplifying repetitive error handling

- The ServeHTTP method calls the appHandler function and displays the returned error (if any) to the user
- Notice that the method's receiver, fn, is a function. (Go can do that!) The method invokes the function by calling the receiver in the expression fn(w, r)
- Now when registering viewRecord with the http package we use the Handle function (instead of HandleFunc) as appHandler is an http.Handler (not an http.HandlerFunc)

Simplifying repetitive error handling

```
func init() {
  http.Handle("/view", appHandler(viewRecord))
}
```

- With this basic error handling infrastructure in place, we can make it more user friendly
- Rather than just displaying the error string, it would be better to give the user a simple error message with an appropriate HTTP status code, while logging the full error to the App Engine developer console for debugging purposes

Simplifying repetitive error handling

- To do this we create an appError struct containing an error and some other fields:

```
type appError struct {
    Error   error
    Message string
    Code    int
}
```

Simplifying repetitive error handling

- Next we modify the appHandler type to return *appError values:
  - `type appHandler func(http.ResponseWriter, *http.Request) *appError`
- And make appHandler’s ServeHTTP method display the appError’s Message to the user with the correct HTTP status Code and log the full Error to the developer console:

Simplifying repetitive error handling

Execute | Share main.go STDIN

```go
func (fn appHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    if e := fn(w, r); e != nil { // e is *appError, not os.Error.
        c := appengine.NewContext(r)
        c.Errorf("%v", e.Error)
        http.Error(w, e.Message, e.Code)
    }
}
```

Simplifying repetitive error handling

- Finally, we update viewRecord to the new function signature and have it return more context when it encounters an error:

Execute | Share main.go STDIN

```go
func viewRecord(w http.ResponseWriter, r *http.Request) *appError {
    c := appengine.NewContext(r)
    key := datastore.NewKey(c, "Record", r.FormValue("id"), 0, nil)
    record := new(Record)
    if err := datastore.Get(c, key, record); err != nil {
        return &appError{err, "Record not found", 404}
    }
    if err := viewTemplate.Execute(w, record); err != nil {
        return &appError{err, "Can't display record", 500}
    }
    return nil
}
```

Control Flow - Looping

- Basic looping
- Iterating with collections
- Exiting loops early

Control Flow Looping - If

- The **if** statement looks as it does in C or Java, except that the **( )** are gone and the **{ }** are required
- Like **for**, the **if** statement can start with a short statement to execute before the condition
- Variables declared by the statement are only in scope until the end of the **if**
- Variables declared inside an if short statement are also available inside any of the else blocks

Control Flow Looping - If

- **If** statement example

```
if answer != 42 {
    return "Wrong answer"
}
```

- If with a short statement

```
if err := foo(); err != nil {
    panic(err)
}
```

For Loop

- Go has only one looping construct, the for loop
- The basic for loop looks as it does in C or Java, except that the ( ) are gone (they are not even optional) and the { } are required
- As in C or Java, you can leave the pre and post statements empty

For Loop

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

- For loop without pre/post statements

```
sum := 1
for ; sum < 1000; {
    sum += sum
}
```

- For loop as a `while` loop

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

Switch case statement

- Most programming languages have some sort of switch case statement to allow developers to avoid doing complex and ugly series of **if else** statements

Switch case statement

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    now := time.Now().Unix()
    mins := now % 2
    switch mins {
    case 0:
        fmt.Println("even")
    case 1:
        fmt.Println("odd")
    }
}
```

Result

```
$go run main.go
even
```

Switch case statement

- There are a few interesting things to know about this statement in Go:
  - You can only compare values of the same type
  - You can set an optional default statement to be executed if all the others fail
  - You can use an expression in the case statement, for instance you can calculate a value to use in the case:

Switch case statement

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    num := 3
    v := num % 2
    switch v {
    case 0:
        fmt.Println("even")
    case 3 - 2:
        fmt.Println("odd")
    }
}
```

Result

```
$go run main.go
odd
```

Switch case statement

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

func main() {
    score := 7
    switch score {
    case 0, 1, 3:
        fmt.Println("Terrible")
    case 4, 5:
        fmt.Println("Mediocre")
    case 6, 7:
        fmt.Println("Not bad")
    case 8, 9:
        fmt.Println("Almost perfect")
    case 10:
        fmt.Println("hmm did you cheat?")
    default:
        fmt.Println(score, " off the chart")
    }
}
```

Result

```
$go run main.go
Not bad
```

Switch case statement

- You can execute all the following statements after a match using the **fallthrough** statement:

Switch case statement

Execute | Share | main.go | STDIN

```go
package main
import "fmt"
func main() {
    n := 4
    switch n {
    case 0:
        fmt.Println("is zero")
        fallthrough
    case 1:
        fmt.Println("is <= 1")
        fallthrough
    case 2:
        fmt.Println("is <= 2")
        fallthrough
    case 3:
        fmt.Println("is <= 3")
        fallthrough
    case 4:
        fmt.Println("is <= 4")
        fallthrough
    case 5:
        fmt.Println("is <= 5")
        fallthrough
    case 6:
        fmt.Println("is <= 6")
        fallthrough
    case 7:
        fmt.Println("is <= 7")
        fallthrough
    case 8:
        fmt.Println("is <= 8")
```

Result

```
$go run main.go
is <= 4
is <= 5
is <= 6
is <= 7
is <= 8
Try again!
```

Switch case statement

- You can use a **break** statement inside your matched statement to exit the switch processing:

Switch case statement

Execute | Share | main.go | STDIN

```go
package main
import (
    "fmt"
    "time"
)
func main() {
    n := 1
    switch n {
    case 0:
        fmt.Println("is zero")
        fallthrough
    case 1:
        fmt.Println("<= 1")
        fallthrough
    case 2:
        fmt.Println("<= 2")
        fallthrough
    case 3:
        fmt.Println("<= 3")
        if time.Now().Unix()%2 == 0 {
            fmt.Println("un pasito pa lante maria")
            break
        }
        fallthrough
    case 4:
        fmt.Println("<= 4")
        fallthrough
    case 5:
        fmt.Println("<= 5")
    }
```

Result

```
$go run main.go
<= 1
<= 2
<= 3
<= 4
<= 5
```

Golang basic: looping

- **Write simple code loop**
- Golang has only have a loop it is for loop lets we code

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

func main() {
    for i := 0; i < 10; i++ {
        fmt.Printf("Loop in number %d\n", i)
    }
}
```

Result

```
$go run main.go
Loop in number 0
Loop in number 1
Loop in number 2
Loop in number 3
Loop in number 4
Loop in number 5
Loop in number 6
Loop in number 7
Loop in number 8
Loop in number 9
```

Golang basic: looping

- We can write loop from 0 (i := 0), and make a condition i<10, if the condition is not true looping will be stop
- Increase the variable i i++ so the the loop will stop

Loop iteration data type

- Now the condition is what happen if we want to get value of slice? we can create like below

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    fruits := []string{"banana","watermelon","apple","coconut"}

    for i := 0; i < len(fruits); i++ {
        fmt.Printf("Fruit index %d is %s\n", i, fruits[i])
    }
}
```

Result

```
$go run main.go
Fruit index 0 is banana
Fruit index 1 is watermelon
Fruit index 2 is apple
Fruit index 3 is coconut
```

Loop iteration data type

- We create loop i until i have value less than 4
- And we get fruit at index of i, but it is not a best practice, we know that we can use range instead

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    fruits := []string{"banana","watermelon","apple","coconut"}

    for i, item := range fruits {
        fmt.Printf("Fruit index %d is %s\n", i, item)
    }
}
```

Result

```
$go run main.go
Fruit index 0 is banana
Fruit index 1 is watermelon
Fruit index 2 is apple
Fruit index 3 is coconut
```

Nested loop

- Imagine we want to create coordinate that have latitude and longitude
- We can create is nested as below

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    for i := 0; i < 10; i++ {
        for j := 0; j < 20; j++ {
            fmt.Printf("%d%d ",i,j)
        }
        fmt.Println()
    }
}
```

Result

```
$go run main.go
00 01 02 03 04 05 06 07 08 09 010 011 012 013 014 015 016 017 018 019
10 11 12 13 14 15 16 17 18 19 110 111 112 113 114 115 116 117 118 119
20 21 22 23 24 25 26 27 28 29 210 211 212 213 214 215 216 217 218 219
30 31 32 33 34 35 36 37 38 39 310 311 312 313 314 315 316 317 318 319
40 41 42 43 44 45 46 47 48 49 410 411 412 413 414 415 416 417 418 419
50 51 52 53 54 55 56 57 58 59 510 511 512 513 514 515 516 517 518 519
60 61 62 63 64 65 66 67 68 69 610 611 612 613 614 615 616 617 618 619
70 71 72 73 74 75 76 77 78 79 710 711 712 713 714 715 716 717 718 719
80 81 82 83 84 85 86 87 88 89 810 811 812 813 814 815 816 817 818 819
90 91 92 93 94 95 96 97 98 99 910 911 912 913 914 915 916 917 918 919
```

Infinity loop

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    for true {
        fmt.Prinft("Infinite Loop")
    }
}
```

Result