NGEE ANN POLYTECHNIC

Go Track

Week 4 Practical

NGEE ANN POLYTECHNIC

Error Handling

Description	*In this lesson, you will learn about error handling mechansim in Go.*
Learning Objectives	• Understand the error handling mechanism mechanism in Go • How to make use of the multiple return values for error handling • How to define package-scope error types and check for these types in the program • How to defined custom error types if needed.
Duration	40 minutes

Keywords

Unlike several languages that make use of try-catch-except exception handling mechanisms as observed in Java, C# and Python, Go's error handling mechanism makes good use of multiple return values feature of functions in the Go language. Error is often returned as the last value, alongside other values or data. Then the program checks for whether the returned error is nil, if it is not, then it is determined that an error has occurred, and the program will handle accordingly.

And because error-handling is achieved using the multiple return values, it encourages Go programmers to incorporate error-handling whenever they are writing functions in their programs. This encourages error-handling to be at the fore-front, getting the developers used to practice of coding defensively, rather than only handle errors when they happen.

Again, while many programming languages like to define several custom exception classes or types, in usual cases, the generic error type in Go language will suffice. This is because it was realized that most errors do not have special information that are better conveyed by custom error types.

In cases however, when errors with more information needs to be captured, Go does allow us to create custom error types. Defning package-scope generic error variables sometimes will do the job too. The whole rationale is to keep error handling to be done actively, but yet keep it simple to code.

Activity #1: Create a simple main program that checks for radius, before calculating the area of the circle created.

1. Write a program that passes in a radius value into a function calCircleArea(), that checks if the radius is more than 0. If it is more than 0, area of the circle, formed from the radius, is calculated. But if radius is less than 0, function will return an error.
2. Main program to print the area of the circle formed, or error message if no circle is formed.

Activity #2: Create custom error types in program that checks if 3 given sides can form a triangle

Given the length of all three sides of a triangle, a, b, c, the area of the triangle can be computed by using Heron's formula:

area = $\sqrt{(s(s-a)(s-b)(s-c))}$ where s = $(a+b+c)/2$

The values of the three sides can form a triangle only if sum of any two sides is larger than the third side.

1. Define an invalidTriangleError error type that consists of the 3 sides (that gives the invalid triangle) and an error message for the error.
2. Define an invalidSideError error type that consists of the erroneous side value and error message.
3. Write a createTriangle() function that checks if any of the 3 sides is less than 0. If yes, an invalidSideError instance is constructed and returned. It then checks if the 3 sides can form

a triangle, in which case if no, an invalidTriangleError would be returned instead. If none of the errors occur, the function will go ahead and calculate the area of the triangle formed and return the calculated value.

4. Write a main program that will test the above.

(Hint can test with values 20, 5, 12 which will not form a triangle, 20, 5, 20 will form a triangle)

NGEE ANN POLYTECHNIC

Panic

Description	*In this lesson, you will learn panic system in Go.*
Learning Objectives	• Understand panic, defer, recover mechanism in Go
Duration	60 minutes

While Go expects programmers to be able to predict errors that may happen during program execution and put in place error handling mechanisms to handle them, sometimes events can happen to the tasks due to unforeseen circumstances. Examples of common panics that may occur are out of bounds, file not found or nil receivers, just to name a few. A good understanding of panic system in Go allows for programmers to recover gracefully from panics even though they occur.

Panic system involves a 3 prong process: panic, defer, recover. A deferred function calls recover() to capture any data involving a panic. The panic data can then be analysed to determine the follow-up action. The deferred function is guaranteed to execute at the moment its parent function returns.

Activity #1: Adding recovery to panic due to out of bounds

1. Write a program that declares a slice of strings representing the names of various board games (or you can choose movies if you do not play board games)
2. Print out the board games and ask the user to choose which his favourite is.
3. Upon reading of the user's choice, the program will acknowledge by printing out the user's choice.

A sample run with valid user values is as follows:

```
The boardgames are:

1 : Carcasssone
2 : Wildlife Safari
3 : Civilization
What is your favourite game?
2
Oh I see. So your favorite game is: Wildlife Safari
```

However, the user may give values that may cause panic in the program, and result in undesirable program termination:

```
The boardgames are:

1 : Carcasssone
2 : Wildlife Safari
3 : Civilization
What is your favourite game?
4
panic: runtime error: index out of range [3] with length 3

goroutine 1 [running]:
main.main()
        C:/Projects/Go/src/goAdvanced/errorsPanics/panics/outOfBounds/outOfBounds.go:22 +0x3f9
exit status 2
```

4. Now enhance the program to incorporate panic recovery mechanism such that the program will inform the user that he has input the program incorrectly, before ending the program properly.

```
The boardgames are:

1 : Carcasssone
2 : Wildlife Safari
3 : Civilization
What is your favourite game?
4
You have entered an invalid choice. Value should be between 0 and 3 _
```

Activity #2: Recovery from panic due to invalid inputs

Write a program that asks user for first name and last name, and responds with a greeting and the full name.

A possible sample run with valid inputs is as follows:

```
Please enter your first name: Yenny
Please enter your last name: Soo
Hello, Yenny Soo !! Nice to meet you
```

Modify the program that will generate panic when either of the first name or last name is empty:

```
Please enter your first name: Yenny
Please enter your last name:
panic: runtime error: last name cannot be nil

main.main()
        C:/Projects/Go/src/goAdvanced/errorsPanics/panics/nilReceiver/nilReceiver.go:26 +0x20e
exit status 2
```

Now improve on your code to incorporate panic recovery mechanism that will inform the user what has gone wrong, before ending the program properly.

NGEE ANN POLYTECHNIC

Concurrency (Mutexes and Atomic Functions)

Description	*In this lesson, you will learn about Concurrency in Go.*
Learning Objectives	• Understand how to run code with goroutines • Understand about race conditions • Understand how to use synchronization techniques such as atomic functions and mutex.
Duration	1 hr 30 min – 2 hours

Concurrency in Go does not provide a threading model. Instead, it makes use of goroutines, and channels.

It is important to note the differences between processes and threads, parallelism and concurrency. To be able to deal with problems involving concurrency, it is necessary to understand what are the factors that can give rise to race conditions, which may result in incorrect data in shared resource. There are several synchronization techniques in Go that can tackle such problems of race conditions. Some traditional ones that are still offered by the language include use of atomic functions and mutex.

Activity #1: Create a simple program that manages the price list of the games that are sold in video games shop.

1. Write a simple program that allows user to add, display game information sold in a game shop.
2. The add function will add game information to a dictionary (or can be parallel slices) to store game name and price.
3. The display function will retrieve the list of game items in the dictionary (or slices) and display to the user.
4. Now write a function manageGameItems() that will allows user option to choose to add game item or display the information of the game items.
5. In main program, create 2 goroutines that will call manageGameItems()
6. Observe if there is race condition.
7. Now put in place synchronization techniques in your program, either making use of atomic functions or mutex.

Activity #2: Create a simple program for handling of a shared account bank balance, with synchronization put in place using mutex

3. Declare a struct called bankBalance which consists of amount in float and its currency in string.
4. Write functions for bankBalance struct to deposit, withdraw, and display information about the bank balance.
5. Write a function processBankBalance() that simulates the bank account application whereby user can choose to deposit into, withdraw from and display information about the balance.
6. Now in main program, make two function calls to processBankBalance() via go keyword to generate 2 goroutines.
7. Observe if there is any race condition.
8. Now put in place synchronization techniques, making use of mutex, to tackle the race condition.

(Note: It is expected that you might observe that the outputs interleave and as a result it may be confusing. The reason is because at this juncture, all of the goroutines are doing output to the same screen. In Go in Action 1 and other further courses, you will create applications with output generated separately at the respective clients, which will be more in line with what is usually observed in fully functional applications.)

NGEE ANN POLYTECHNIC

Channels

Description	*In this lesson, you will learn about Channels in Go.*
Learning Objectives	• Understand how to use channels for communication between goroutines • Understand how to use channels for locking
Duration	1 hr 30 min – 2 hours

Channels provide a way to send messages from one goroutine to another. Channels can be unidirectional, bidirectional, are typed and send structured data. There are also buffered and unbuffered channels that can be created.

Sends and receives to a channel are blocking by default. When a data is sent to a channel, the control is blocked in the send statement until some other Goroutine reads from that channel. Similarly when data is read from a channel, the read is blocked until some Goroutine writes data to that channel.

This property of channels is what helps Goroutines communicate effectively without the use of explicit locks or conditional variables that are quite common in other programming languages.

Activity #1:

Consider the following code:

```
package main

import (
"fmt"
"time"
)

func hello(num int) {
fmt.Println("Hello world goroutine", num)
}
func main() {
for i := 1; i <= 4; i++ {
   go hello(i)
}
time.Sleep(4 * time.Second)
fmt.Println("main function")
}
```

8. In main, sleep was called so as to allow the goroutines to have some time to complete their execution before main continues rest of its execution.
9. Now, instead of using sleep, rewrite the above program to make use of channels that allow each goroutine to inform main once they are done with execution.
10. Next, make each goroutine sleep for 4 seconds in hello() before informing main. What would you observe?

Activity #2: Create a program that calculates sum of squares and cubes based on individual digits of a number

9. Obtain input of a number from user.

10. Then the program will calculate the sum of squares and cubes based on individual digits of the given number. For example, if input is 123, the program will compute

    squares = (1 * 1) + (2 * 2) + (3 * 3)
    cubes = (1 * 1 * 1) + (2 * 2 * 2) + (3 * 3 * 3)
    output = squares + cubes = 50

11. Allow the sum of squares and sum of cubes to be calculated in separate goroutines.
12. Then using channels, the goroutines are to send the calculated answers to main, which will then calculated the sum of squares and cubes to give the final result.

Activity #3: Create a program that involves goroutines reading from files

1. Create 2 goroutines write1() and write2() that will read game titles from 2 separate text files and store them in slices or arrays
2. Now in your main program, use select case which will check for receipt of the game titles from write1 or write 2 via channels.
3. Whichever the main program receive via the channel first, it will print out the details of the game titles.
4. In case, the goroutines takes a lot time to come back, a timeout, should be imposed, after which the program will end execution.

(hint: select helps you to wait on multiple operations)