Unit : 4

Control Flow - Branch & Functions

- If
- Switch
- Panic
- Functions
  - Declaration
  - Parameters
  - Variadic functions
  - Returning data
  - Anonymous function

If else statement

- if is a statement that has a boolean condition and it executes a block of code if that condition evaluates to true
- It executes an alternate else block if the condition evaluates to false
- **If statement syntax**
- The syntax of the if statement is provided below

```
if condition {

}
```

If else statement

- If the `condition` is true, the lines of code between the braces `{` and `}` is executed
- Unlike in other languages like C,
  - the braces `{ }` are mandatory even if there is only one line of code between the braces`{ }`

Example

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    num := 10
    if num%2 == 0 {
        fmt.Println("The number", num, "is even")
        return
    }
    fmt.Println("The number", num, "is odd")
}
```

Result

```
$go run main.go
The number 10 is even
```

If else statement

- The if statement has an optional else construct which will be executed if the condition in the if statement evaluates to false

```
if condition {
} else {
}
```

- In the above snippet, if condition evaluates to false, then the lines of code between else { and } will be executed

If else statement

- Rewrite the program to find whether the number is odd or even using if else statement

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    num := 11
    if num%2 == 0 {
        fmt.Println("The number", num, "is even")
    } else {
        fmt.Println("The number", num, "is odd")
    }
}
```

Result

```
$go run main.go
The number 11 is odd
```

If else statement

- In the above code, instead of returning if the condition is true , we create an else statement that will be executed if the condition is false
- In this case, since 11 is odd, the if condition is false and the lines of code within the else statement is executed
- The above program will print the number 11 is odd

If ... else if ... else statement

- The if statement also has optional else if and else components
- The syntax for the same is provided below

```
if condition1 {
...
} else if condition2 {
...
} else {
...
}
```

If ... else if ... else statement

- The condition is evaluated for the truth from the top to bottom
- In the above statement if condition1 is true, then the lines of code within if condition1 { and the closing brace } are executed
- If condition1 is false and condition2 is true, then the lines of code within else if condition2 { and the next closing brace } is executed
- If both condition1 and condition2 are false, then the lines of code in the else statement between else { and } are executed
- There can be any number of else if statements

If ... else if ... else statement

- In general, whichever if or else if's condition evaluates to true, it's corresponding code block is executed
- If none of the conditions are true then else block is executed
- Let's write a program that uses else if

If ... else if ... else statement

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

func main() {
    num := 99
    if num <= 50 {
        fmt.Println(num, "is less than or equal to 50")
    } else if num >= 51 && num  <= 100 {
        fmt.Println(num, "is beteween 51 and 100")
    } else {
        fmt.Println(num, "is greater than 100")
    }
}
```

Result

```
$go run main.go
99 is beteween 51 and 100
```

If with assignment

- There is one more variant of if which includes an optional shorthand assignment statement that is executed before the condition is evaluated
- Its syntax is :

  if assignment-statement; condition {

  }
- In the above snippet, assignment-statement is first executed before the condition is evaluated

If with assignment

- Let's rewrite the program which finds whether the number is even or odd using the above syntax

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    if num := 10; num % 2 == 0 {
        fmt.Println(num, "is even")
    } else {
        fmt.Println(num, "is odd")
    }
}
```

Result

```
$go run main.go
10 is even
```

Switch

- The switch statement lets you check multiple cases
- You can see this as an alternative to a list of if-statements that looks like spaghetti
- A switch statement provides a clean and readable way to evaluate cases
- While the switch statement is not unique to Go,

Syntax

- The basic syntax of a switch statement is:

Go

```go
switch var1 {
    case val1:
        ...
    case val2:
        ...
    default:
        ...
}
```

Syntax

- You can also check conditions:

Go

```go
switch {
    case condition1:
        ...
    case condition2:
        ...
    default:
        ...
}
```

Syntax

- The second form of a switch statement is not to provide any determined value (which is actually defaulted to be true)
- Instead it test different conditions in each case branch
- So what is a condition?
- A condition can be x > 10 or x == 8.
- *The code of the branch is executed when the test result of either branch is true*
- The third form of a switch statement is to include an initialization statement:

Syntax

Go

```
switch initialization {
    case val1:
        ...
    case val2:
        ... default:
        ...
} switch result: = calculcate (); {
    case result < 0: ...
    case result > 0: ... default:
        // 0}
```

Syntax

- Variable var1 can be any type, and val1 and val2 can be any value of the same type
- The type is not limited to a constant or integer,
  - but must be the same type;
- The front braces {must be on the same line as the switch keyword
- You can test multiple potentially eligible values at the same time, using commas to split them, for example: case val1, val2, val3:.

Example

Execute | Share main.go STDIN

```go
package main
import "fmt"
func main() {
    switch a := 1; {
    case a == 1:
        fmt.Println("The integer was == 1")
        fallthrough
    case a == 2:
        fmt.Println("The integer was == 2")
    case a == 3:
        fmt.Println("The integer was == 3")
        fallthrough
    case a == 4:
        fmt.Println("The integer was == 4")
    case a == 5:
        fmt.Println("The integer was == 5")
        fallthrough
    default:
        fmt.Println("default case")
    }
}
```

Result

```
$go run main.go
The integer was == 1
The integer was == 2
```

What is Panic?

- The idiomatic way of handling abnormal conditions in a Go program is using errors
- Errors are sufficient for most of the abnormal conditions arising in the program
- **But there are some situations where the program cannot continue execution after an abnormal condition**
- **In this case, we use `panic` to prematurely terminate the program**

What is Panic?

- When a function encounters a panic, its execution is stopped, any deferred functions are executed and then the control returns to its caller
- This process continues until all the functions of the current goroutine have returned at which point the program prints the panic message, followed by the stack trace and then terminates
- It is possible to regain control of a panicking program using `recover`

What is Panic?

- panic and recover can be considered similar to try-catch-finally idiom in other languages
  - Such as Java except that they are rarely used in Go

When Should Panic Be Used?

- One important factor is that you should avoid panic and recover and use errors where ever possible
- Only in cases where the program just cannot continue execution should panic and recover mechanism be used
- There are two valid use cases for panic

When Should Panic Be Used?

- **An unrecoverable error where the program cannot simply continue its execution**
- One example is a web server that fails to bind to the required port
  - In this case, it's reasonable to panic as there is nothing else to do if the port binding itself fails

When Should Panic Be Used?

- **A programmer error**
- Let's say we have a method that accepts a pointer as a parameter and someone calls this method using a nil argument
- In this case, we can panic as it's a programmer error to call a method with nil argument which was expecting a valid pointer

Panic Example

- The signature of the built-in `panic` function is provided below,
  - `func panic(interface{})`
- The argument passed to the panic function will be printed when the program terminates
- The use of this will be clear when we write a sample program
- Start with a contrived example which shows how panic works

Panic Example

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

func fullName(firstName *string, lastName *string) {
    if firstName == nil {
        panic("runtime error: first name cannot be nil")
    }
    if lastName == nil {
        panic("runtime error: last name cannot be nil")
    }
    fmt.Println("%s %s\n", *firstName, *lastName)
    fmt.Println("returned normally from fullName")
}

func main() {
    firstName := "Elon"
    fullName(&firstName, nil)
    fmt.Println("returned normally from main")
}
```

Result

```
$go run main.go
panic: runtime error: last name cannot be nil

goroutine 1 [running]:
main.fullName(0xc420083f48, 0x0)
        /home/cg/root/7812113/main.go:10 +0x219
main.main()
        /home/cg/root/7812113/main.go:18 +0x4d
exit status 2
```

Panic Example

- The above is a simple program to print the full name of a person
- The fullName function in line no.7 prints the full name of a person
- This function checks whether the firstName and lastName pointers are nil in line nos. 8 and 11 respectively
- If it is nil the function calls panic with a corresponding message
- This message will be printed when the program terminates
- Running this program will print the following output,

Panic Example

```
panic: runtime error: last name cannot be nil


goroutine 1 [running]:
main.fullName(0xc00006af58, 0x0)
    /tmp/sandbox210590465/prog.go:12 +0x193
main.main()
    /tmp/sandbox210590465/prog.go:20 +0x4d
```

Panic Example

- Analyze this output to understand how panic works and how the stack trace is printed when the program panics
- In line no. 19 we assign Elon to firstName
- We call fullName function with lastName as nil in line no. 20
- Hence the condition in line no. 11 will be satisfied and the program will panic
- When panic is encountered, the program execution terminates, the argument passed to the panic function is printed followed by the stack trace

Panic Example

- Since the program terminates following the panic function call in line no. 12, the code in line nos. 13, 14, and 15 will not be executed
- This program first prints the message passed to the panic function,
  - panic: runtime error: last name cannot be nil
- and then prints the stack trace
- The program panicked in line no. 12 of fullName function and hence,

Panic Example

```
goroutine 1 [running]:
main.fullName(0xc00006af58, 0x0)
    /tmp/sandbox210590465/prog.go:12 +0x193
```

- will be printed first
- Then the next item in the stack will be printed

Panic Example

- In our case, line no. 20 where the fullName is called is the next item in the stack trace
- Hence it is printed next

  main.main()

  /tmp/sandbox210590465/prog.go:20 +0x4d
- Now we have reached the top level function which caused the panic and there are no more levels above, hence there is nothing more to print

Function

- A function is a group of statements that exist within a program for the purpose of performing a specific task
  - At a high level, a function takes an input and returns an output
- Function allows you to extract commonly used block of code into a single component
- The single most popular Go function is main(), which is used in every independent Go program

Creating a Function

- A declaration begins with the `func` keyword, followed by the name you want the function to have, a pair of parentheses (), and then a block containing the function's code
- The following example has a function with the name SimpleFunction. It takes no parameter and returns no values

Creating a Functions

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

// SimpleFunction prints a message
func SimpleFunction() {
    fmt.Println("Hello World")
}

func main() {
    SimpleFunction()
}
```

Result

```
$go run main.go
Hello World
```

Simple function with parameters in Golang

- Information can be passed to functions through arguments
- An argument is just like a variable
- Arguments are specified after the function name, inside the parentheses
  - You can add as many arguments as you want, just separate them with a comma
- The following example has a function with two arguments of int type
- When the add() function is called, we pass two integer values (e.g. 20,30)

Example

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

// Function accepting arguments
func add(x int, y int) {
    total := 0
    total = x + y
    fmt.Println(total)
}

func main() {
    // Passing arguments
    add(20, 30)
}
```

Result

```
$go run main.go
50
```

The return values of a function can be named in Golang

- Golang allows you to name the return values of a function
- We can also name the return value by defining variables,
  - Here a variable total of integer type is defined in the function declaration for the value that the function returns

Example

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

func rectangle(l int, b int) (area int) {
    var parameter int
    parameter = 2 * (l + b)
    fmt.Println("Parameter: ", parameter)

    area = l * b
    return // Return statement without specify variable name
}

func main() {
    fmt.Println("Area: ", rectangle(20, 30))
}
```

Result

```
$go run main.go
Parameter:  100
Area:  600
```

Naming Conventions for Golang Functions

- A name must begin with a letter, and can have any number of additional letters and numbers
- A function name cannot start with a number
- A function name cannot contain spaces
- If the functions with names that start with an uppercase letter will be exported to other packages
  - If the function name starts with a lowercase letter, it won't be exported to other packages, but you can call this function within the same package

Naming Conventions for Golang Functions

- If a name consists of multiple words, each word after the first should be capitalized like this:
  - empName, EmpAddress, etc
- function names are case-sensitive (car, Car and CAR are three different variables)

Anonymous Functions in Golang

- An anonymous function is a function that was declared without any named identifier to refer to it
- Anonymous functions can accept inputs and return outputs, just as standard functions do
- Assigning function to the variable

Anonymous Functions in Golang

Execute | Share main.go STDIN

```go
package main

import "fmt"

var (
    area = func(l int, b int) int {
        return l * b
    }
)

func main() {
    fmt.Println(area(20, 30))
}
```

Result

```
$go run main.go
600
```

Example - Passing arguments to anonymous functions

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    func(l int, b int) {
        fmt.Println(l * b)
    }(20, 30)
}
```

Result

```
$go run main.go
600
```

Example- Function defined to accept a parameter and return value

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    fmt.Printf(
        "100 (°F) = %.2f (°C)\n",
        func(f float64) float64 {
            return (f - 32.0) * (5.0 / 9.0)
        }(100),
    )
}
```

Higher Order Functions

- A Higher-Order function is a function that receives a function as an argument or returns the function as output
- Higher order functions are functions that operate on other functions, either by taking them as arguments or by returning them

Passing Functions as Arguments to other Functions

Execute | Share main.go STDIN

```go
package main

import "fmt"

func sum(x, y int) int {
    return x + y
}
func partialSum(x int) func(int) int {
    return func(y int) int {
        return sum(x, y)
    }
}
func main() {
    partial := partialSum(3)
    fmt.Println(partial(7))
}
```

Result

```
$go run main.go
10
```

Returning Functions from other Functions

Execute | Share main.go STDIN

```go
package main

import "fmt"

func squareSum(x int) func(int) func(int) int {
    return func(y int) func(int) int {
        return func(z int) int {
            return x*x + y*y + z*z
        }
    }
}
func main() {
    // 5*5 + 6*6 + 7*7
    fmt.Println(squareSum(5)(6)(7))
}
```

Result

```
$go run main.go
110
```

User Defined Function Types in Golang

- Golang also support to define our own function types
- The modified version of above program with function types as below:

Execute | Share main.go STDIN

```go
package main

import "fmt"

type First func(int) int
type Second func(int) First

func squareSum(x int) Second {
    return func(y int) First {
        return func(z int) int {
            return x*x + y*y + z*z
        }
    }
}

func main() {
    // 5*5 + 6*6 + 7*7
    fmt.Println(squareSum(5)(6)(7))
}
```

Variadic Functions

- A variadic function is a function that accepts a variable number of arguments
- In Golang, it is possible to pass a varying number of arguments of the same type as referenced in the function signature
- To declare a variadic function, the type of the final parameter is preceded by an ellipsis, "...",
  - which shows that the function may be called with any number of arguments of this type

Variadic Functions

- This type of function is useful when you don't know the number of arguments you are passing to the function,
  - the best example is built-in Println function of the fmt package which is a variadic function

Select single argument from all arguments of variadic function

- In below example we will are going to print s[0] the first and s[3] the forth, argument value passed to variadicExample() function

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {
    variadicExample("red", "blue", "green", "yellow")
}

func variadicExample(s ...string) {
    fmt.Println(s[0])
    fmt.Println(s[3])
}
```

Result

```
$go run main.go
red
yellow
```

Select single argument from all arguments of variadic function

- Needs to be precise when running an empty function call,
  - if the code inside of the function expecting an argument and absence of argument will generate an error
  - "panic: run-time error: index out of range"
- In above example you have to pass at least 4 arguments

Passing multiple string arguments to a variadic function

- The parameter s accepts an infinite number of arguments
- The tree-dotted ellipsis tells the compiler that this string will accept, from zero to multiple values

Execute | Share main.go STDIN

```go
package main

import "fmt"

func main() {

    variadicExample()
    variadicExample("red", "blue")
    variadicExample("red", "blue", "green")
    variadicExample("red", "blue", "green", "yellow")
}

func variadicExample(s ...string) {
    fmt.Println(s)
}
```

Result

```
$go run main.go
[]
[red blue]
[red blue green]
[red blue green yellow]
```

Normal function parameter with variadic function parameter

Execute | Share | main.go | STDIN

```go
package main

import "fmt"

func main() {
    fmt.Println(calculation("Rectangle", 20, 30))
    fmt.Println(calculation("Square", 20))
}

func calculation(str string, y ...int) int {

    area := 1

    for _, val := range y {
        if str == "Rectangle" {
            area *= val
        } else if str == "Square" {
            area = val * val
        }
    }
    return area
}
```

Result

```
$go run main.go
600
400
```

Pass different types of arguments in variadic function

Execute | Share main.go STDIN

```go
package main

import (
    "fmt"
    "reflect"
)

func main() {
    variadicExample(1, "red", true, 10.5, []string{"foo", "bar", "baz"},
        map[string]int{"apple": 23, "tomato": 13})
}

func variadicExample(i ...interface{}) {
    for _, v := range i {
        fmt.Println(v, "--", reflect.ValueOf(v).Kind())
    }
}
```

Result

```
$go run main.go
1 -- int
red -- string
true -- bool
10.5 -- float64
[foo bar baz] -- slice
map[apple:23 tomato:13] -- map
```