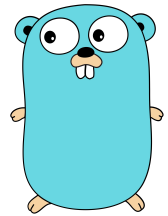
Go from scratch: Beginners-Friendly Guide

by Denis Shchuka

Adding Behaviour



Function

a small piece of code that takes some input parameters, does some processing on the inputs and produces some output results. Can take 0 or more arguments.

```
func fibo(x int) int {
        if x <= 1 {
         return x
    }
    return fibo(n-1) +
fibo(n-2)
}
```

Method

```
func (rocket Rocket) launch()
{
        //...
}
var r Rocket
r.Launch()
```

receiver

Interface

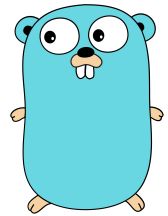
a set of method signatures. Does not contain any implementation.

```
type launcher interface {
    launch()
}
```

Go from scratch: Beginners-Friendly Guide

by Denis Shchuka

Adding Behaviour. Functions



Functions

Syntax

```
func name_of_function(list_of_parameters) (list_of_results) {
      //function body
}
```

Example 1

```
func add(x, y int) int {
      return x + y
}
result := add(2, 3)
```

Example 2

```
func rectanglePerimeter(x, y int) int {
      result := add(x, y)
      return result
}
```

Functions allow to pack a set of operators to be invoked from other place in your program. It is the **one more way to structure** your code!

- Accept 0 or more parameters (arguments)
- Return a result
- Can invoke other functions inside

Functions

Example 3. No results

```
func generateReport(filepath string){
      //report generation code
}
```

Example 4. Multiple results

```
 func generateReport(filepath string) (size int, err error) {
       //report generation code
 }
```

Example 5. Recursion

```
 func fibo(x int) int {
       if x <= 1 {
         return x
      }
      return fibo(n-1) + fibo(n-2)
 }
```

- Can return nothing
- Can return multiple results
- Can invoke other functions inside
- Can invoke themselves

Functions

Functions in Go are “first-class citizens”. You can assign them to variables

```
func add(x, y int) int {
	return x + y
}
proc := add
fmt.Println(proc(2,6))
```

User-defined functions types. You can define a new function type on your own

```
type processor func(filepath string) (fsize int,
err error)
```

Higher-order functions. Functions can use other functions as arguments and return values.

```
type procSelector func(ftype string) processor
```

Functions

Practice

```
func twoSum(nums []int, target int) []int
{
	//TODO:
}
```

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have *exactly* one solution, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Output: Because nums[0] + nums[1] == 9, we return [0, 1].
```

https://leetcode.com/

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Anonymous Functions

- Do not have names
- Can be invoked immediately
- Can be assigned to a variable
- Can be defined inside other functions
- Can be passed as a parameter to other function

Example 1

```
func main() {
	proc := func(){
		fmt.Println(“Inside anonymous function”)
	}
	proc()
}
```

Example 2

```
go func(x int){
	fmt.Println(x * x)
}(10)
```

Example 3

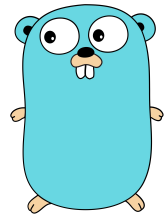
```
fruits := []string{“Banana”,“Apple”,“Pineapple”}
comparator := func(i, j int) bool {
	return len(fruits[i] < len(fruits[j])
}

sort.Slice(fruits, comparator)
```

Go from scratch: Beginners-Friendly Guide

by Denis Shchuka

Adding Behaviour. Methods



Methods

Method is function that has receiver.

Methods is the Go-way to add behaviour to objects.

Methods can have value or pointer receiver.

```
type Employee struct {
      ID          int
      Name        string
      Salary      float64
}
```

Method definition

```
func (e *Employee) ToString() string{
      return fmt.Sprintf("%v | %s - %v", e.ID, e.Name, e.Salary)
}

func (e *Employee) UpdateName(name string) bool {
      if len(name) > 0 {
            e.Name = name
            return true
      }
      return false
}
```

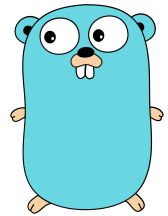
Usage

```
func main() {
      empl := &Employee{ID: 0, Name: "Mike", Salary: 10000}
      fmt.Println(empl)
}
```

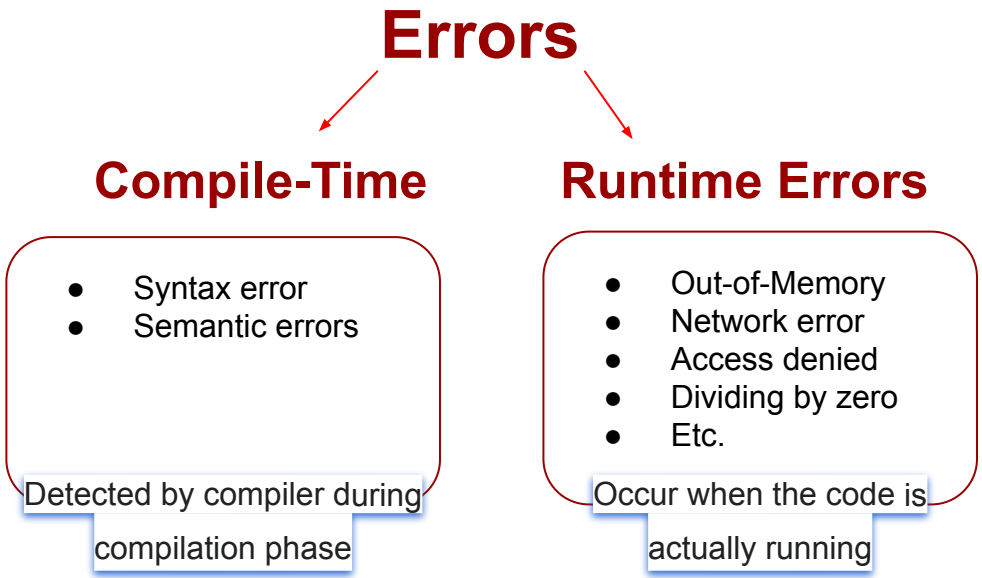
Go from scratch: Beginners-Friendly Guide

by Denis Shchuka

Adding Behaviour. Error Handling



Error Handling



Error Handling

Basic principles of error handling

Success is not guaranteed
Errors should be expected behaviour in your program
You can’t rely on infrastructure (memory, network, storage, etc.)
You have to choose strategy to handle errors

Error Handling

Strategies of error handling

Propagation to a caller code

```
func getDescription(fpath string) string, error {
      //report generation code
      …
      //if something went wrong
      return nil, fmt.Errorf("Task successfully
failed!")
}
```

Operation retry

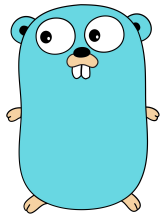
```
func getDescription(url string) string, error {
      deadline := time.Now().Add(1*time.Minute)
      for i := 0; time.Now().Before(deadline); i++ {
            res, err := http.Head(url)
            if err == nil {
                  return res, nil //Success
            }
            log.Printf("Retry…", err)
            time.Sleep(10*time.Second)
      }
      return nil, fmt.Errorf("Server is not responding")
}

s, err := getDescription("http://example.com/1")
if err != nil {
      return
}
```

Go from scratch: Beginners-Friendly Guide

by Denis Shchuka

Adding Behaviour.
Interfaces.



Interfaces

Interface is a set of method signatures. It **does not contain any implementation**.

Syntax

```
type Launcher interface {
    launch()
    calculateRoute(lat, lon float64) bool
}
```

- Interface is an abstraction of other types behaviour
- Interfaces in **Go** are implemented **implicitly** - you don't need to specify what interfaces a type implements
- Interfaces can contain other interface types

Interfaces

Example

```
type Launcher interface {
    launch() bool
    calculateRoute(lat, lon float64) bool
}
```

```
type SpaceLauncher struct{
       title string
       //other fields definition...
}

func (e *SpaceLauncher) launch(){
       //do something
       return true
}

func (e *SpaceLauncher) calculateRoute(lat,lon float64) bool {
       //calculating route
       return true
}

func Attack(l Launcher) {
       l.launch()
}
```

```
//somewhere in main function...
sl := SpaceLauncher{title="Space Bird"}
ml := MarineLauncher{fields…}
if Attack(sl) {
      Attack(ml)
}
```