# Golang для начинающих

Клестов Дима,
Golang-разработчик ООО «Инносети»

https://t.me/MaJloe_3Jlo
https://github.com/MaJloe3Jlo

# Результаты занятия 5

Все молодцы! Задача 3 была на внимательность:
Условие: Напишите программу с ДВУМЯ горутинами, первая отправляет числа в канал, а вторая считает количество пришедших чисел и их сумму. Вывести: количество и сумму.

- func main сама по себе является горутиной и достаточно было создать только одну.
  Кто заметил - молодец! :)

Хорошие решения:
1)https://play.golang.org/p/SIJdKHhwliA
2)https://play.golang.org/p/HcZGfv25RSm
3)https://play.golang.org/p/W7tq7bDR3K5

https://play.golang.org/

# План занятия

На прошлом занятии:
1) Структуры
2) Интерфейсы
3) Горутины и каналы



План занятия на сегодня:
1) Повторное использование кода.
Пакеты в Go
2) Документирование кода
3) Обработка ошибок
4) Домашняя работа

# Повторное использование кода и пакеты в GO


- DRY — Dont Repeat Yourself - Не повторяйте себя.
- Все программы на Go представляются в виде пакетов.
- Пакеты позволяют разделить код на отдельные части и модули.
- Код пакета располагает в одном или нескольких файлах с расширением go.
- Для определения пакета используется служебное слово package.

Пример:

```
package main
```

# Импорт пакетов

- Чтобы использовать какой-либо пакет в своей программе, то его необходимо импортировать с помощью слова import.

Пример:

```
package main

import “fmt“
import “net/http“
```

Также import можно объединять с помощью () (круглых скобок):

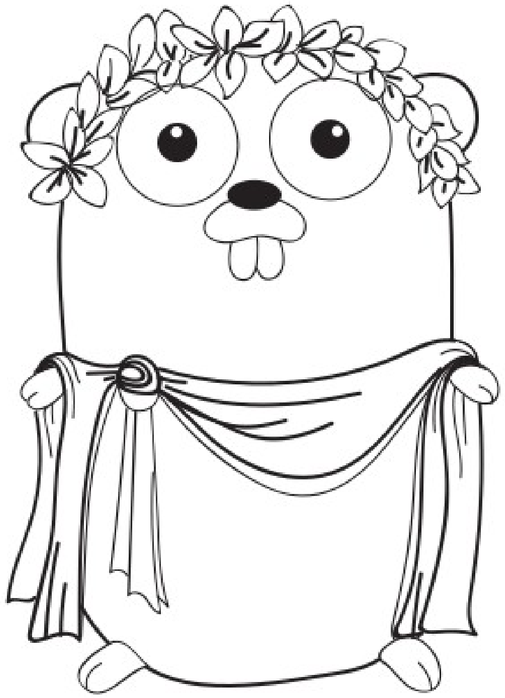
```
import (
    “fmt“
    “net/http“
)
```

# Использование пакетов

- Чтобы использовать, функции пакета в теле программы:
  1)пишем имя пакета
  2)через точку вызываем необходимую функцию пакета.

Пример:

```
fmt.Scan(&number)
```



# Примеры на практике

# Документирование кода

- Go позволяет автоматически создавать документацию к пользовательским пакетам так же, как и документировать стандартные пакеты. Запустите эту команду в терминале:

$ go doc <путь к пакету который, мы написали выше>

Пример из стандартной библотеки:
```
go doc fmt Scan
```

```
package fmt // import "fmt"

func Scan(a ...interface{}) (n int, err error)
    Scan scans text read from standard input, storing successive space-separated
    values into successive arguments. Newlines count as space. It returns the
    number of items successfully scanned. If that is less than the number of
    arguments, err will report why.
```

- Вся информация которая написана в комментарии выше сущностей в пакетах будет выведена с помощью go doc.

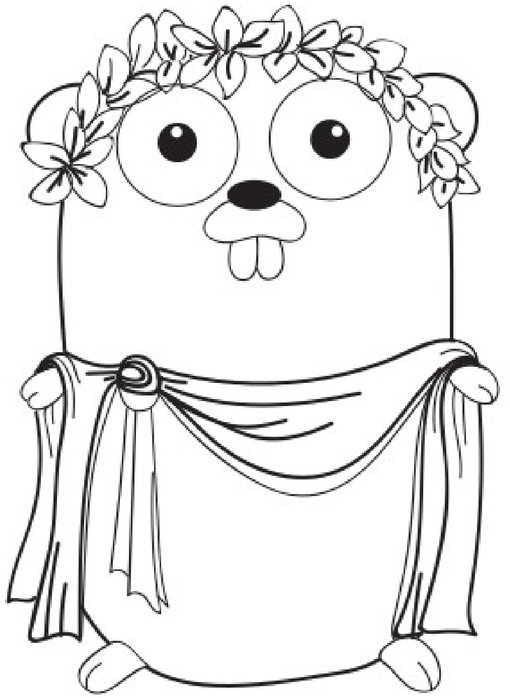
# Обработка ошибок

- Хороший код должен правильно реагировать на непредвиденные обстоятельства.

**Обработка ошибок** — это процесс обнаружения ситуаций, когда ваша программа находится в неожиданном состоянии, а также принятие мер для записи диагностической информации, которая будет полезна при последующей отладке.

- Ошибки в Go — это значения с типом errог, возвращаемые функциями, как и любые другие значения.

- Самая простая обработка - это проверка ошибки на пустоту.



# Обработка ошибок

- Многие методы и функции в GO при вызове возвращают не только нужный результат, но и ошибку.

Пример:

```
package main

import "fmt"

func multiply(a int, b int) int {
	return a * b
}

func main() {
	var number int
	_, err := fmt.Scan(&number) // Функция Scan возвращает два параметра, но нам сейчас важно проверить только ошибку.
	if err != nil {
		fmt.Println("Проверьте типы входных параметров")
	} else {
		fmt.Println(multiply(number, 3)) // Выведем результат, если ошибок нет.
	}
}
```

# Создание ошибок

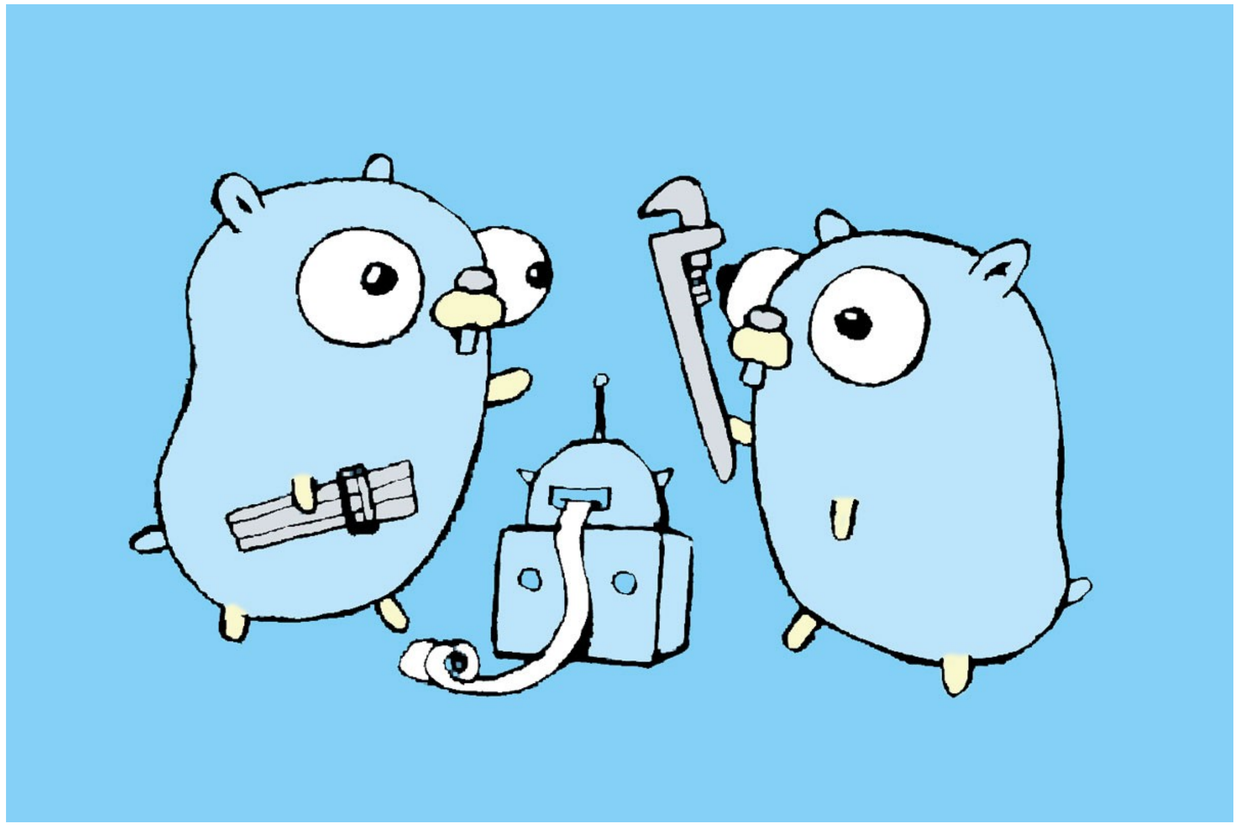
- Стандартная библиотека: errors.New и fmt.Errorf.
- Обе эти функции позволяют нам указывать настраиваемое сообщение об ошибке, которое вы можете отображать вашим пользователям.

Пример

```
package main

import (
    "errors"
    "fmt"
)

func main() {
    err := errors.New("my error")
    fmt.Println("", err)
}
// Вывод: my error
```



# error — это интерфейс

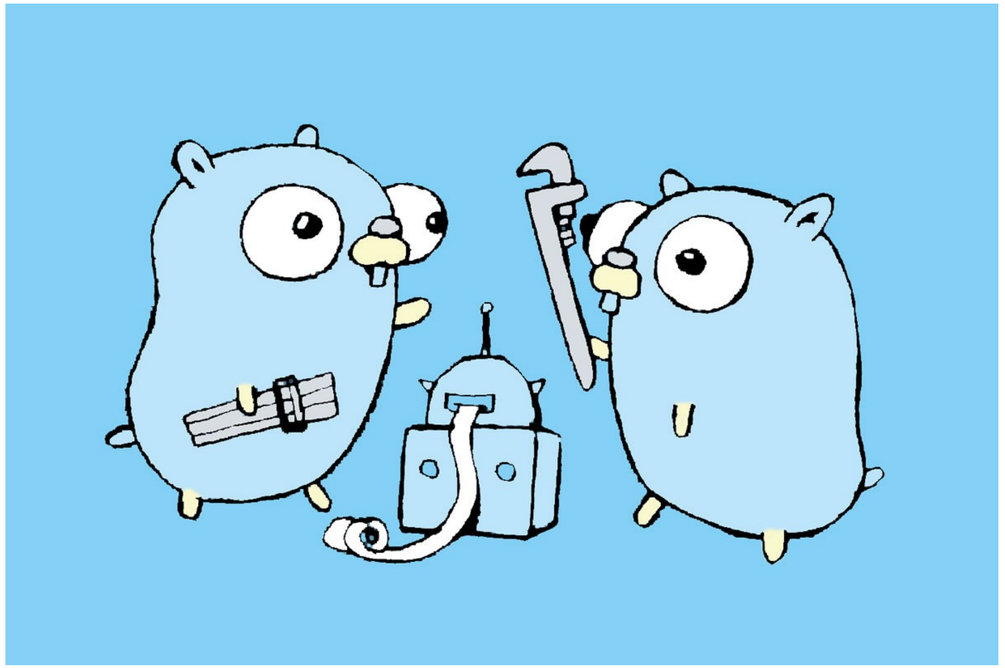
- error в Go является интерфейсом.

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

- Мы можем создавать свои методы Error для структур с ошибками.

Пример.

```
type RequestError struct {
   StatusCode int
   Err error
}
func (r *RequestError) Error() string {
   return fmt.Sprintf("status %d: err %v", r.StatusCode, r.Err)
}
func doRequest() error {
   return &RequestError{
      StatusCode: 503,
      Err:        errors.New("unavailable"),
   }
}
func main() {
    err := doRequest()
    if err != nil {
          fmt.Println(err)
    }
}
```



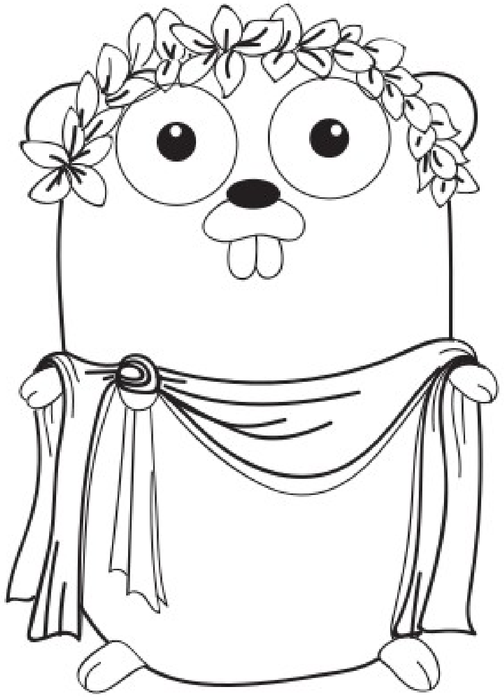
# Оператор panic

- Оператор panic позволяет сгенерировать ошибку и выйти из программы.
- Оператор panic может принимать любое сообщение, которое будет выводиться на консоль.
- В конце вывода будет идти диагностическая информация о том, где возникла ошибка

Пример:

```
package main
import "fmt"

func main() {
    fmt.Println(divide(15, 5))
    fmt.Println(divide(4, 0))
    fmt.Println("Program has been finished")
}
func divide(x, y float64) float64{
    if y == 0{
        panic("division by zero!")
    }
    return x / y
}
```



# Оператор defer

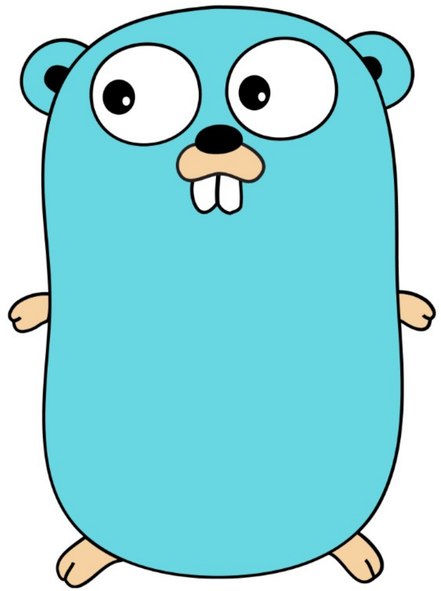
- Оператор defer позволяет выполнить определенную операцию после каких-то действий (даже если сработает panic), при этом не важно, где в реальности вызывается эта функция.

Пример:

```
package main
import "fmt"

func main() {
    defer finish()
    fmt.Println("Program has been started")
    fmt.Println("Program is working")
}

func finish(){
    fmt.Println("Program has been finished")
}
```



Вывод:

```
Program has been started
Program is working
Program has been finished
```

# Несколько операторов defer

- Если несколько функций вызываются с оператором defer, то те функции, которые вызываются раньше, будут выполняться позже всех.

Пример:

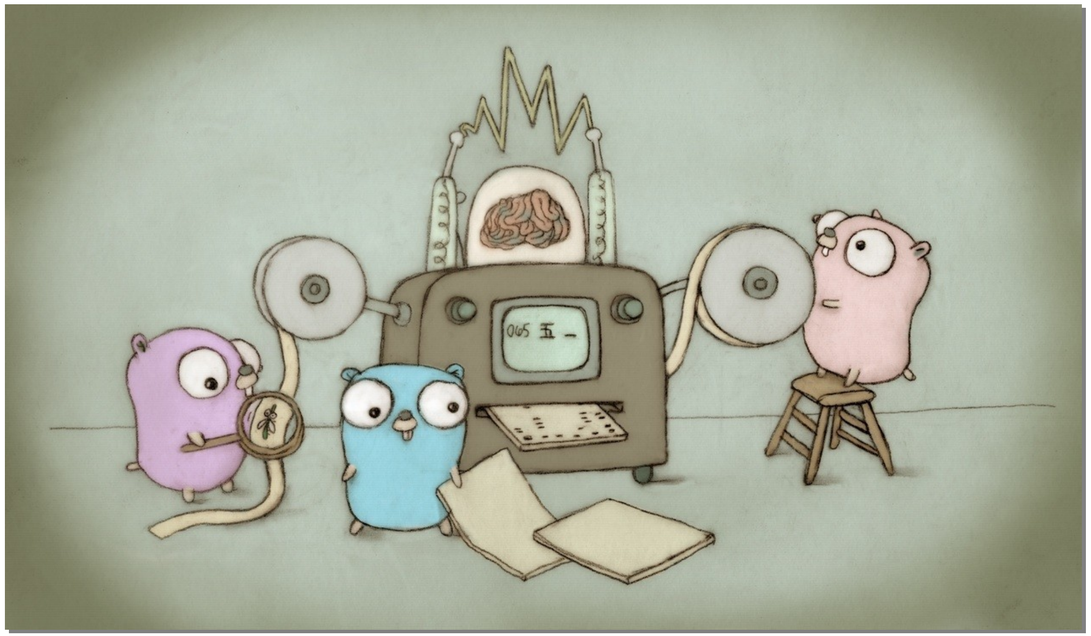
```
package main
import  "fmt"

func main() {
    defer finish()
    defer fmt.Println("Program has been started")
    fmt.Println("Program is working")
}

func finish(){
    fmt.Println("Program has been finished")
}
```

Консольный вывод:

```
Program is working
Program has been started
Program has been finished
```



# Оператор defer

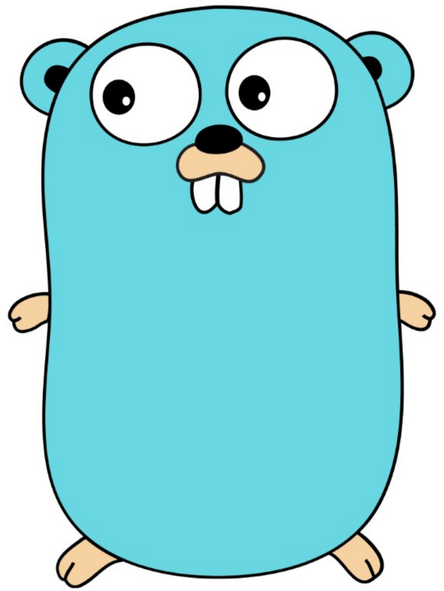
- defer помещает вызов функции в стек. Поэтому они выполняются в очередности -LIFO (Last-In, First-Out).
- defer запоминает значения переменных, переданных в функцию, на момент объявления defer, а не на момент его вызова.

Пример:

```
number := 5

defer math(number)
// Будет передано значение 5, а не 7

number = 7
```



# Оператор recover

**Recover** — это встроенная функция, которая восстанавливает контроль над паникующей го-процедурой. Recover полезна только внутри отложенного вызова функции. Во время нормального выполнения, recover возвращает nil и не имеет других эффектов. Если же текущая го-процедура паникует, то вызов recover возвращает значение, которое было передано panic и восстанавливает нормальное выполнение.

Пример:

```
func recoveryFunction() {
  if recoveryMessage:=recover(); recoveryMessage != nil {
    fmt.Println(recoveryMessage)
  }
  fmt.Println("This is recovery function...")
}

func executePanic() {
  defer recoveryFunction()
  panic("This is Panic Situation")
  fmt.Println("The function executes Completely")
}

func main() {
  executePanic()
  fmt.Println("Main block is executed completely...")
}
```

На следующем занятии:

1) Тестирование
2) Стандартная библиотека
3) Следующие шаги


Задание здесь:
https://bit.ly/gostudy6

# Благодарю за внимание!!!