

Golang для начинающих



Клестов Дима,
Golang-разработчик ООО «Инносети»

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

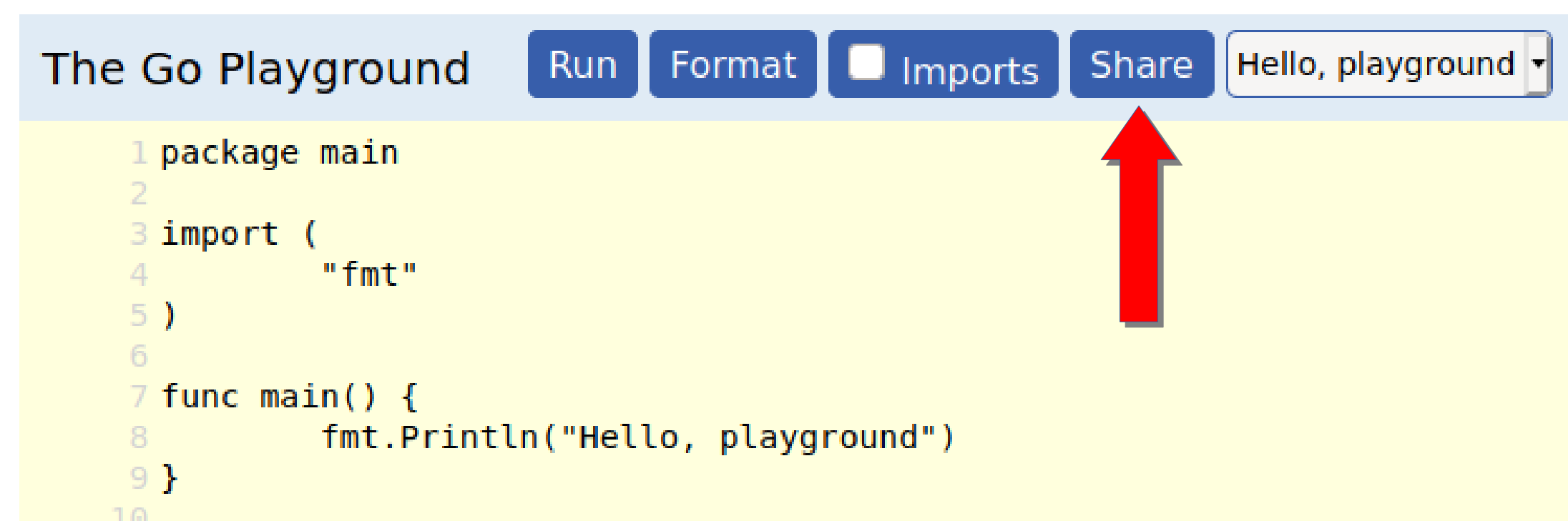
Результаты занятия 4

Все молодцы!

Хорошие решения:

- 1) Правильный ответ: служебное слово func (2 ответили: имя функции. А как же анонимные функции?)
- 2) <https://play.golang.org/p/vDTyceKTm8I>
- 3) https://play.golang.org/p/_Ryqcq1hOoj
- 4) <https://play.golang.org/p/yx-2FldIS52>
- 5) <https://play.golang.org/p/yB4f7QetUBQ>

<https://play.golang.org/>



The screenshot shows the 'The Go Playground' interface. At the top, there is a header bar with the title 'The Go Playground' and several buttons: 'Run', 'Format', 'Imports' (with a small square icon), and 'Share'. To the right of these buttons is a text input field containing 'Hello, playground'. Below the header bar is a large yellow area for code. The code is as follows:

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     fmt.Println("Hello, playground")
9 }
10
```

A red arrow points to the 'Share' button in the header bar.

На прошлом занятии:

- 1) Функции
- 2) Указатели
- 3) Домашняя работа



План занятия на сегодня:

- 1) Структуры и интерфейсы
- 2) Конкурирование
- 3) Домашняя работа

```
package main

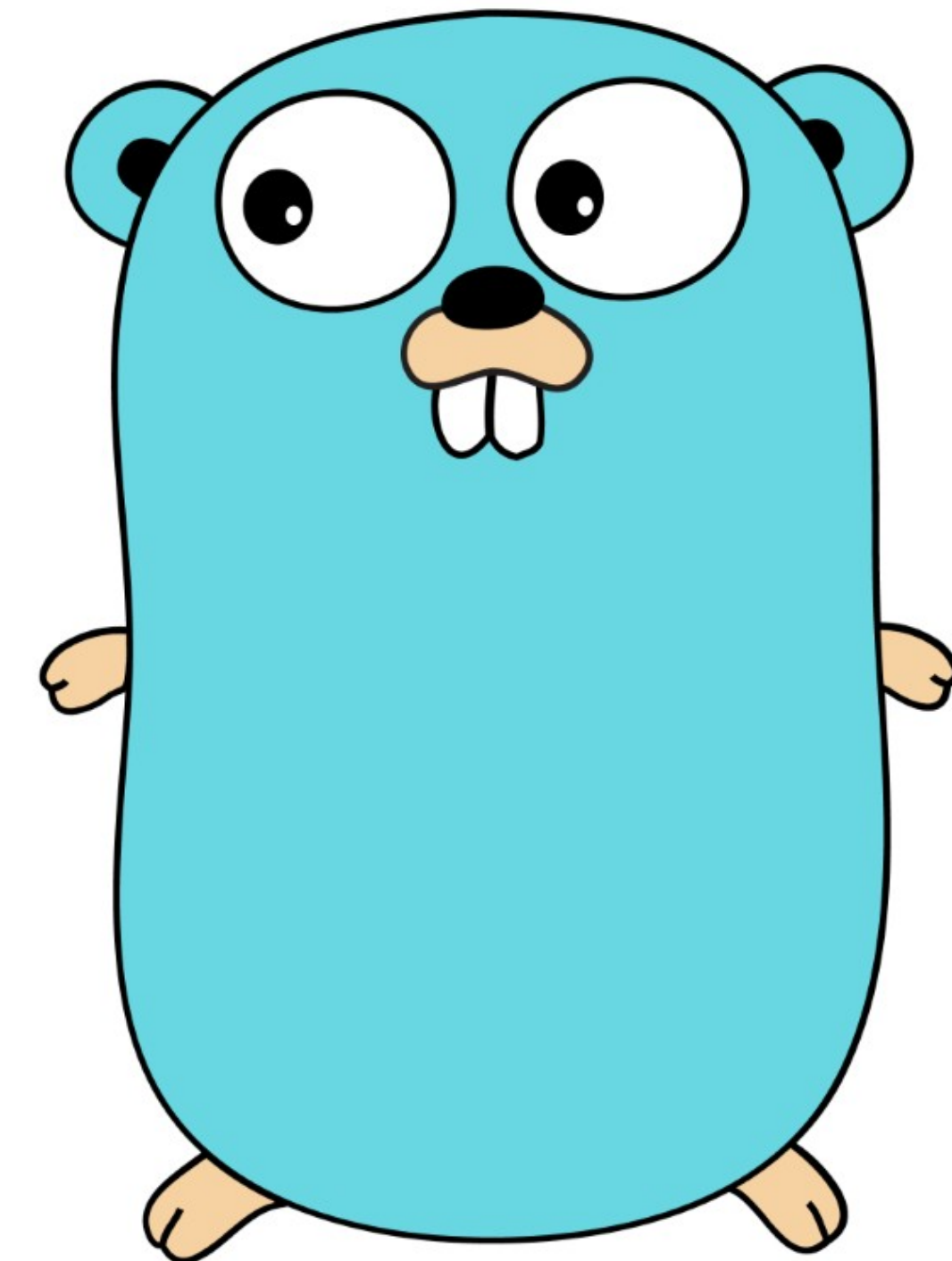
import ("fmt"; "math")

func rectangleArea(x1, y1, x2, y2 float64) float64 {
    l := distance(x1, y1, x1, y2)
    w := distance(x1, y1, x2, y1)
    return l * w
}

func circleArea(x, y, r float64) float64 {
    return math.Pi * r * r
}

func main() {
    var rx1, ry1 float64 = 0, 0
    var rx2, ry2 float64 = 10, 10
    var cx, cy, cr float64 = 0, 0, 5

    fmt.Println(rectangleArea(rx1, ry1, rx2, ry2))
    fmt.Println(circleArea(cx, cy, cr))
}
```



Объявление структуры в общем виде:

```
type <название> struct {  
    // Поля структуры  
}
```

Примеры:

```
type Circle struct {  
    x float64  
    y float64  
    r float64  
}
```

или проще

```
type Circle struct {  
    x, y, r float64  
}
```



Создание и инициализация структуры

Создание:

```
var circ Circle
```

или

```
circ := Circle{}
```

или с помощью new:

```
circ := new(Circle)
```

Инициализировать начальные значения для полей структуры:

```
circ := Circle{x: 1, y: 3, r: 5}
```

или (если мы знаем порядок полей, то их имена можно опустить)

```
circ := Circle{1,3,5}
```



Доступ к полям

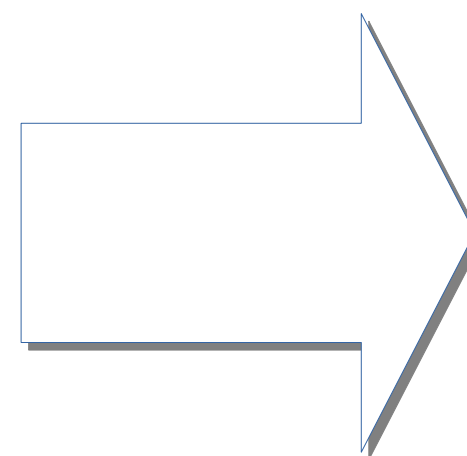
✓ Чтобы получить доступ к полям структуры используется . (точка)

Пример:

```
circ := Circle{  
    circ.x = 5  
    fmt.Println(circ.x) // Вывод: 5
```

Можно изменить нашу функцию

```
func circleArea(x, y, r float64) float64 {  
    return math.Pi * r * r  
}
```



```
func circleArea(circ Circle) float64 {  
    return math.Pi * c.r * c.r  
}
```

```
func main() {  
    circ := Circle{1, 2, 5}  
    fmt.Println(circleArea(circ))  
}
```

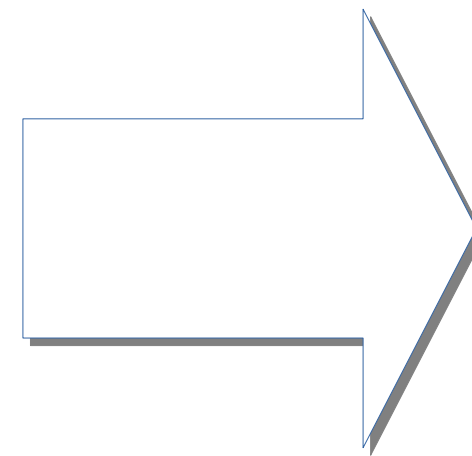

Методы структур

Методы в общем виде:

```
func (<имя получателя> <тип получателя>) <имя метода> (<параметры>) (<типы  
возвращаемых данных>) {  
    // Выполняемые операции  
}
```

Нашу функцию по вычислению площади круга мы можем сделать еще лучше с помощью метода:

```
func circleArea(circ Circle) float64 {  
    return math.Pi * c.r * c.r  
}
```



```
func (c *Circle) area() float64 {  
    return math.Pi * c.r * c.r  
}
```

```
func main() {  
    circ := Circle{1,2,3}  
    circ.area()  
}
```

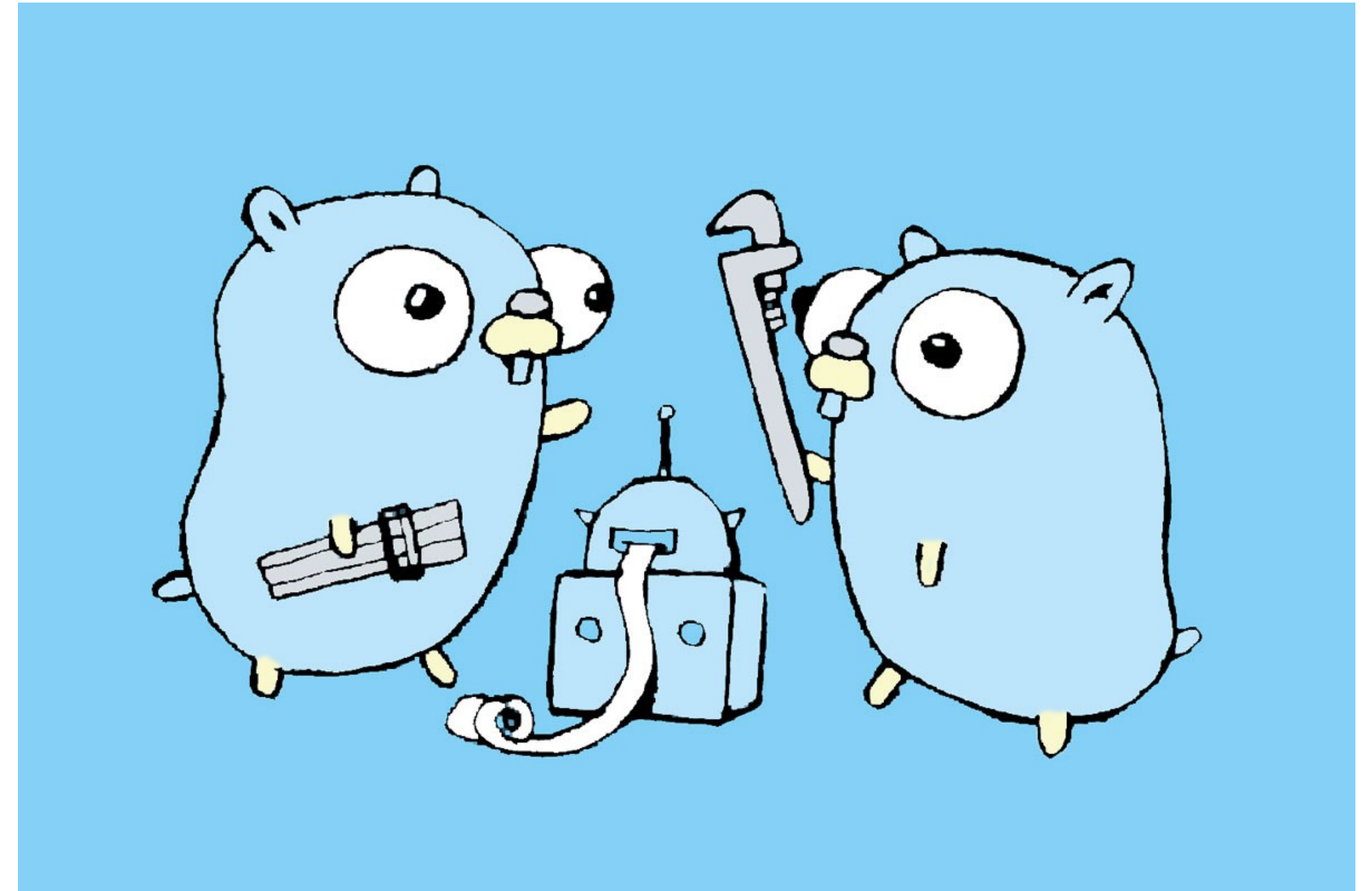

Пример на практике

«Видимость» и экспортируемость

- Если начинается с большой буквы, то это является экспортируемой (публичной) сущностью (переменная, поле структуры, метод и т. д.). То есть она будет видна извне пакета в котором она объявлена.
- Если начинается с маленькой буквы, то наоборот будет не экспортируемой (то есть приватной) и не будет видна извне пакета, где она объявлена.

Например структуры:

```
type Person struct {  
    name string // не экспортируемая (приватная)  
    Address string // экспортируемая (публичная)  
}
```



Встраиваемые структуры

Поля любой структуры могут являться другими структурами.

```
type Animal struct {  
    Class string  
}
```

```
func (a *Animal) Talk(voice string) {  
    fmt.Println(voice)  
}
```

```
type Cat struct {  
    Animal Animal  
    Type string  
}
```

```
type Dog struct {  
    Animal // анонимное встраивание  
    Type string  
}
```

Вызов из main:

```
cat := Cat{}  
cat.Animal.Talk("meow") // Вывод: meow
```

```
cat.Talk("another meow")  
// ОШИБКА cat.Talk undefined (type Cat has no  
field or method Talk)
```

```
dog := Dog{}  
dog.Talk("woof") // Вывод: woof
```

```
dog.Animal.Talk("another woof")  
// Вывод: another woof
```

- ✓ Интерфейсы представляют объект поведения других типов.
- ✓ Интерфейсы позволяют определять функции, которые не привязаны к конкретной реализации.
- ✓ Интерфейсы определяют некоторую функциональность, но не реализуют ее.

Пример объявления из стандартной библиотеки `io`.

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}
```



Пример на практике

Пустой интерфейс

- ✓ Интерфейс, который не содержит ни одного метода называется пустым интерфейсом: `interface{}`.
- ✓ Пустой интерфейс может содержать значение любого типа.
- ✓ Пустые интерфейсы используются в коде, где необходимо работать со значениями неизвестного типа.

Пример из стандартной библиотеки `fmt`:

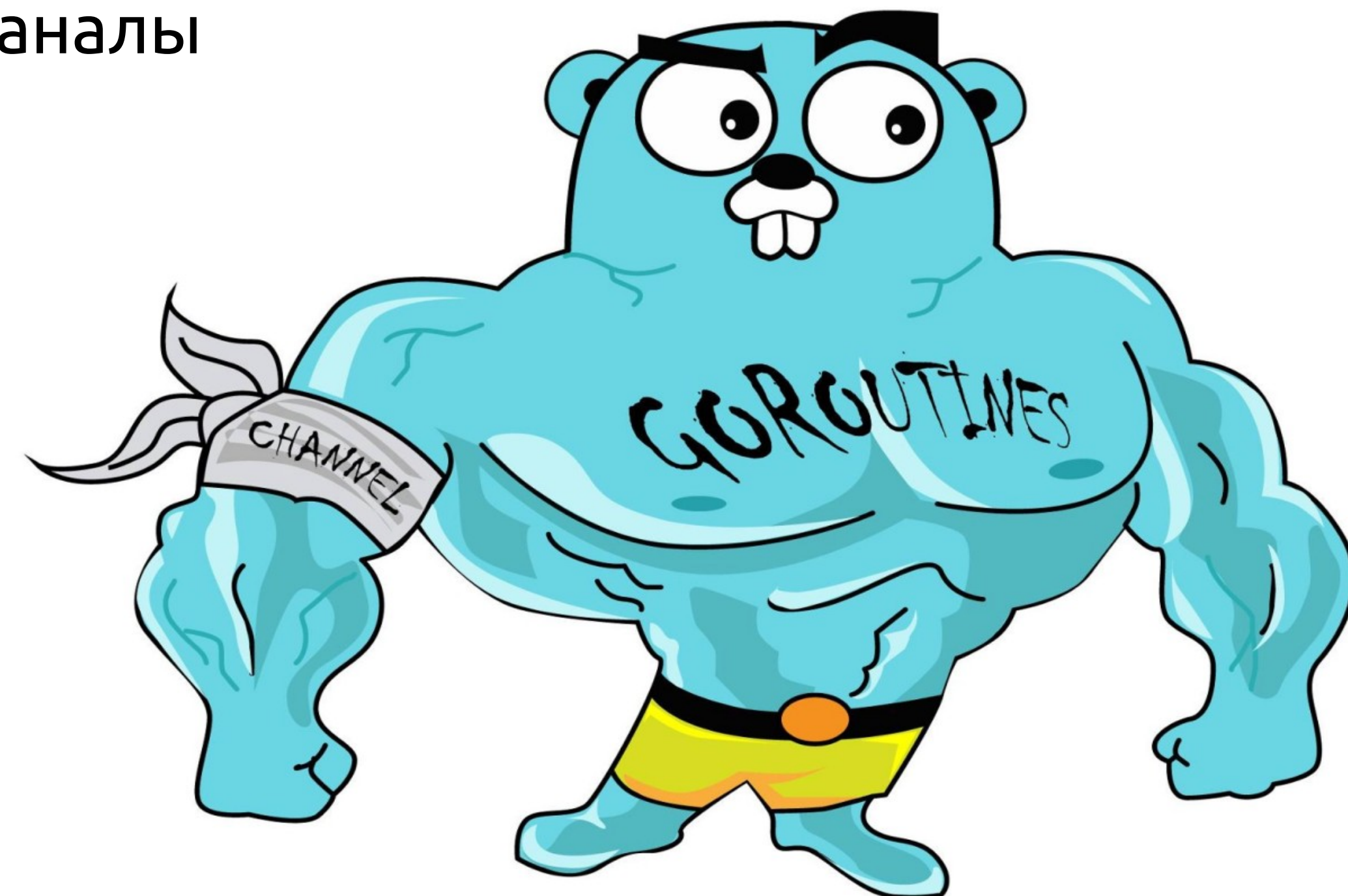
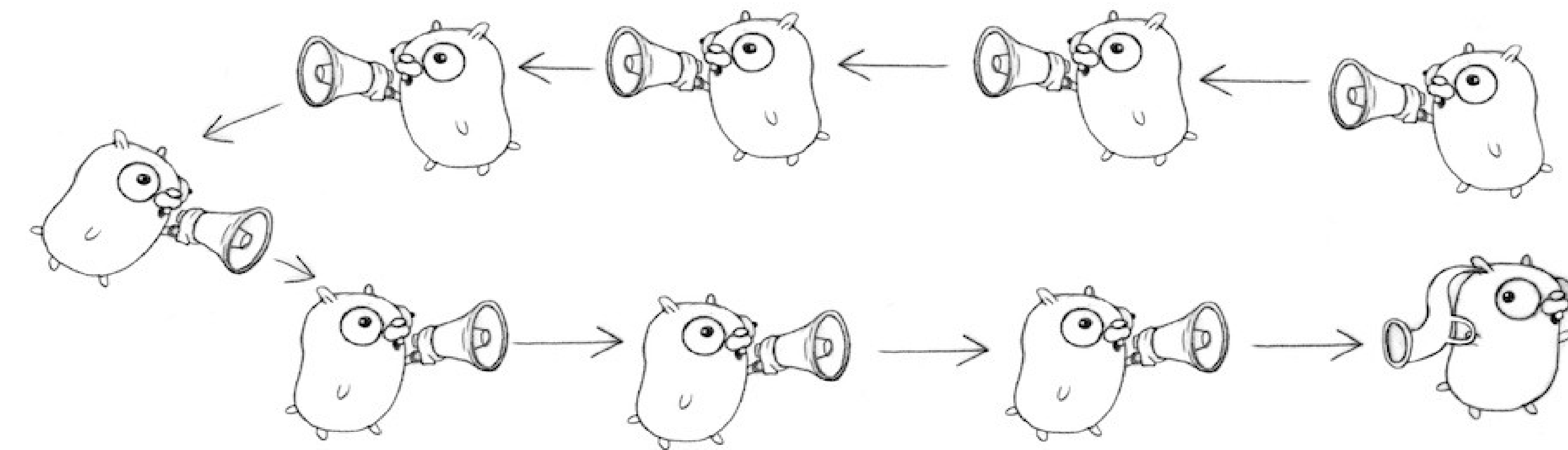
```
func Print(a ...interface{}) (n int, err error) {  
    return Fprint(os.Stdout, a...)  
}
```


Конкурентность

Многопоточность — обозначает параллельную обработку данных на многих вычислительных узлах. Но многопоточность в Go реализована в виде поддержки конкурентности (concurrency).

Конкурентность позволяет иметь дело с несколькими вещами одновременно, а параллелизм — делать несколько вещей одновременно.

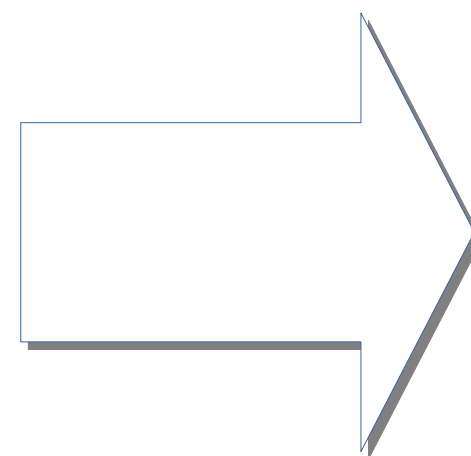
Горутины и каналы



Горутина — это функция, которая может работать параллельно с другими функциями. Для создания горутины используется ключевое слово `go`, за которым следует вызов функции.

```
func f(n int) {  
    for i := 0; i < 10; i++ {  
        fmt.Println(n, ":", i)  
    }  
}
```

```
func main() {  
    go f(0)  
    time.Sleep(10 * time.Second)  
}
```



// Вывод:

```
0:0  
0:1  
0:2  
0:3  
0:4  
0:5  
0:6  
0:7  
0:8  
0:9
```



Горутина

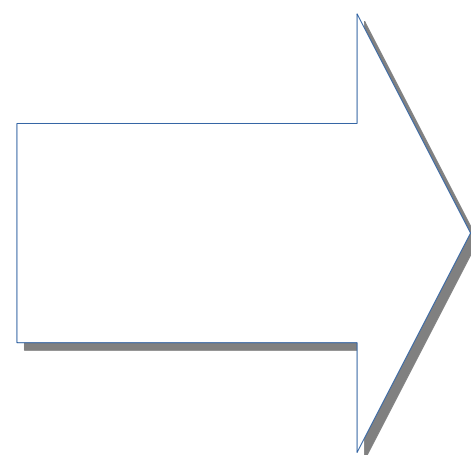
Функция `main`, сама по себе, является горутинной.

Вторая горутина создаётся, когда мы вызываем `go f(0)`.

✓ С горутинной программа немедленно перейдет к следующей строке, не дожидаясь, пока вызываемая функция завершится. Горутины очень легкие, мы можем создавать их тысячами.

Пример запуска с 10 горутинами:

```
func main() {  
    for i := 0; i < 10; i++ {  
        go f(i)  
    }  
    time.Sleep(10 * time.Second)  
}
```



// Вывод:

9:0

7:0

7:1

7:2

7:3

7:4

7:5

...



Примеры на практике

- ✓ Каналы обеспечивают возможность общения нескольких горутин друг с другом.
- ✓ Создается канал с помощью служебного слова `chan`. Пример:

`c := make(chan string)`
- ✓ Каналы нужны чтобы синхронизировать выполнение горутин.

Пример программы с использованием каналов:

```
func pinger(c chan string) {  
    for i := 0; ; i++ {  
        c <- "ping"  
    }  
}
```

```
func printer(c chan string) {  
    for {  
        msg := <-c  
        fmt.Println(msg)  
        time.Sleep(time.Second * 1)  
    }  
}
```

```
func main() {  
    var c chan string = make(chan string)  
    go pinger(c)  
    go printer(c)  
    time.Sleep(10 * time.Second)  
}
```

Примеры на практике

Направление каналов

Каналы могут быть:

- ✓ Отправляющими
- ✓ Принимающими
- ✓ Двухнаправленными

Примеры:

- ✓ `func pinger(c chan<- string)` - канал с будет только отправлять сообщение.
Попытка получить сообщение из канала с вызовет ошибку компилирования.
- ✓ `func printer(c <-chan string)` — канал с будет только принимать сообщение.
- ✓ `func printer(c chan string)` — канал с будет двухнаправленный.

Особенности:

- ✓ Двухнаправленные каналы, которые могут быть переданы в функцию, принимающую только принимающие или отправляющие каналы.
- ✓ Но отправляющие или принимающие каналы не могут быть переданы в функцию, требующую двухнаправленного канала!

Работает как switch, но для каналов:

```
go func() {  
    for {  
        c1 <- "from 1"  
        time.Sleep(time.Second * 2)  
    }  
}()  
go func() {  
    for {  
        c2 <- "from 2"  
        time.Sleep(time.Second * 3)  
    }  
}()  
go func() {  
    for {  
        select {  
        case msg1 := <-c1:  
            fmt.Println(msg1)  
        case msg2 := <-c2:  
            fmt.Println(msg2)  
        }  
    }  
}()
```

*Оператор select

Обычно select используется для таймеров:

```
go func() {  
    for {  
        select {  
        case msg1 := <-c1:  
            fmt.Println("Message 1", msg1)  
        case msg2 := <-c2:  
            fmt.Println("Message 2", msg2)  
        case <-time.After(time.Second):  
            fmt.Println("timeout")  
            return  
        }  
    }  
}()
```

✓ Так же в select есть default значение.

Пример в коде

*Буферизированный канал

При инициализации канала можно использовать второй параметр:

`c := make(chan int, 1)` - буферизированный канал с ёмкостью 1.

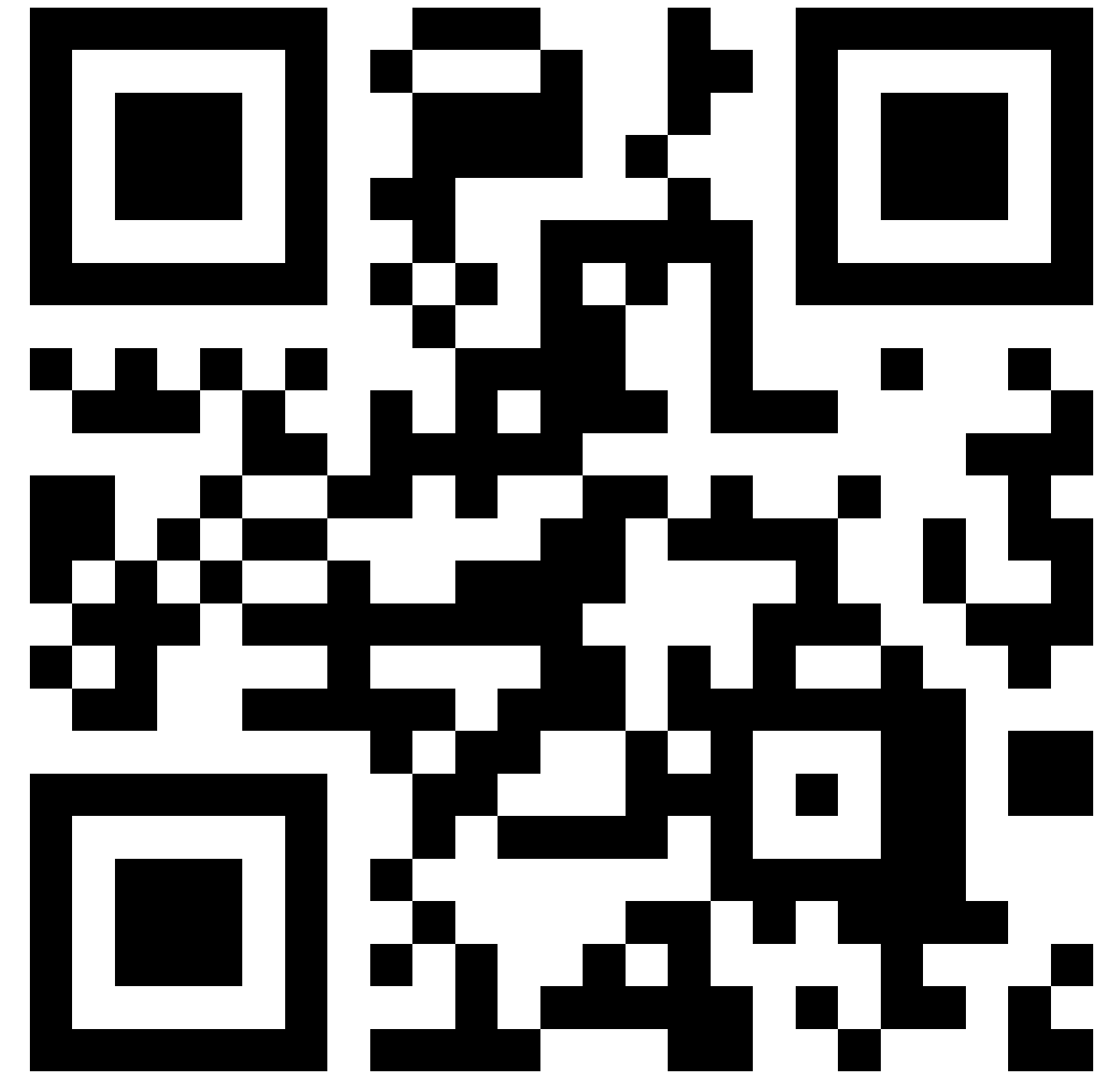
- ✓ Обычно каналы работают синхронно - каждая из сторон ждёт, когда другая сможет получить или передать сообщение.
- ✓ Буферизованный канал работает асинхронно — получение или отправка сообщения не заставляют стороны останавливаться.
- ✓ Канал теряет пропускную способность, когда он занят, в данном случае, если мы отправим в канал 1 сообщение, то мы не сможем отправить туда ещё одно до тех пор, пока первое не будет получено.

На следующем занятии:

- 1) Пакеты и повторное использование кода
- 2) Документирование кода
- 3) Обработка ошибок



Задание здесь:
https://bit.ly/gostudy__5



Благодарю за внимание!!!

