

# Golang

## Функции, условия



# Информация

Данная публикация стала возможной благодаря помощи американского народа, оказанной через Агентство США по международному развитию (USAID). Алиф Академия несёт ответственность за содержание публикации, которое не обязательно отражает позицию USAID или Правительства США.



# ПРЕДИСЛОВИЕ



# Предисловие

На прошлой лекции мы посмотрели, как использовать "чужие" функции - т.е. функции из стандартной библиотеки. Пришло время научиться писать свои функции.



# ФУНКЦИИ



# Функции

Функция - это кусочек кода, которому дали имя. Используя это имя, можно запускать эту функцию (таким образом будет исполняться этот самый кусочек кода).

Но зачем это нужно? Ведь до этого мы вполне обходились без всяких функций (ну, кроме `main`). Нам достаточно было "собрать"\* программу и запускать её с разными аргументами командной строки, чтобы получать нужный нам результат.

Примечание\*: под термином "собрать" мы будем понимать компиляцию и сборку исполняемого файла.



# Функции

А теперь представьте следующую задачу: у нас с вами есть 200 000 клиентов. И мы хотим им рассылать push-уведомления об операциях по их картам (счетам). Например, клиент платит с карты в магазине, мы получаем об этом информацию и **отправляем ему уведомление о покупке.**



# Функции

Обратите внимание: мы говорим "**отправляем уведомление о покупке**", хотя на самом деле это операция включает в себя следующие шаги:

1. Получить имя клиента, сумму операции, название магазина
2. Взять шаблон вида "{username} вы совершили платёж на сумму {amount} с. в магазине {store}. Если это были не вы, позвоните по телефону {phone}!".
3. Заменить в этом шаблоне плейсхолдеры (placeholder - заполнитель) реальными данными
4. Отправить, используя сервисы Google или Apple\*

Примечание\*: чаще всего используют сервис Firebase, который работает с обеими системами.





# Функции

Почему мы говорим именно так? Потому что нам так проще, удобнее - мы можем один раз вам объяснить, что мы будем называть "отправить уведомление о покупке" именно такую последовательность шагов и далее везде использовать это "название", чем каждый раз проговаривать всю последовательность.



# Функции

Возвращаемся к нашим 200 000 клиентам. Мы, конечно, можем каждый раз запускать программу с разными аргументами, но было бы лучше, если бы она постоянно была запущена и при появлении нового платежа сама отправляла уведомление.

Т.е. всё дело только в том, что **нам так удобнее**. Точно так же как разработчики стандартной библиотеки сделали функцию `ReplaceAll`, чтобы вы просто могли её вызывать, а не проходить по каждой букве и заменять её.



# Функции

Давайте попробуем написать нашу первую функцию (помимо `main`). Возьмём одну из предыдущих задач - "Мегафон Спасибо". Вот так выглядит её решение:



The screenshot shows an IDE interface. On the left, the 'OPEN EDITORS' panel displays a project structure: 'MEGAFON' (expanded) contains 'cmd \ bonus', which contains 'main.go' (selected) and 'go.mod'. The main editor area shows the code for 'main.go' with line numbers 1 through 14. The code is as follows:

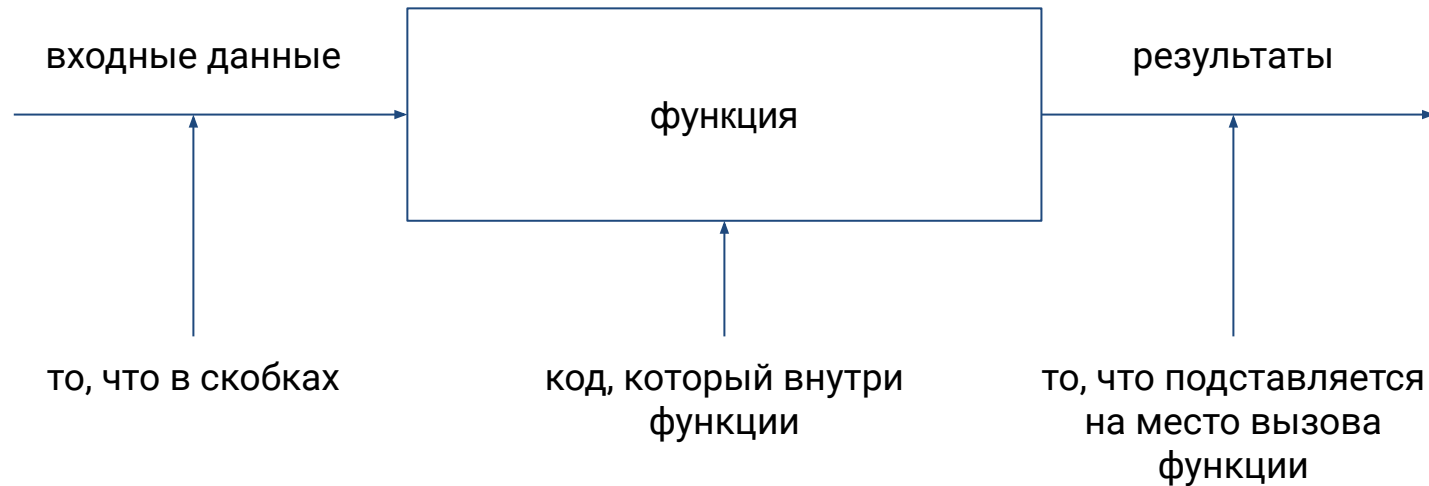
```
cmd > bonus > -go main.go > ...
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      amount := 666
9      welcomeBonus := 5
10     bonusRate := 5
11     bonus := welcomeBonus + amount / bonusRate
12
13     fmt.Println(bonus)
14 }
```

Конечно же, мы использовали знания, полученные в предыдущей лекции.



# Функции

Теперь давайте вспомним вот эту картинку:



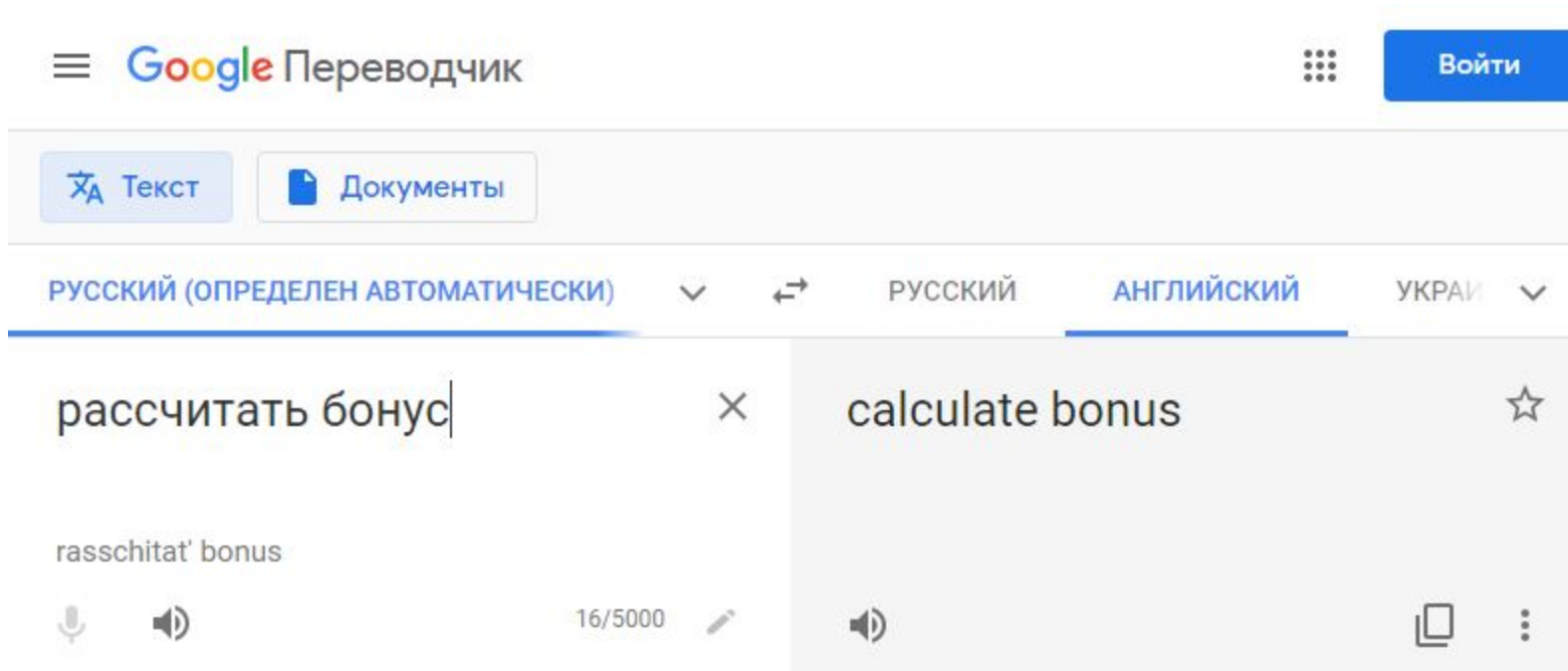
Т.е. нам нужно:

1. Определить входные данные
2. Определить код внутри функции
3. Определить результаты
4. А ещё не забыть придумать имя функции



# Функции

Начнём с имени функции. Если функция рассчитывает бонус, то разумно её так же и назвать - "рассчитать бонус":



# Функции

Функция определяется следующим образом:

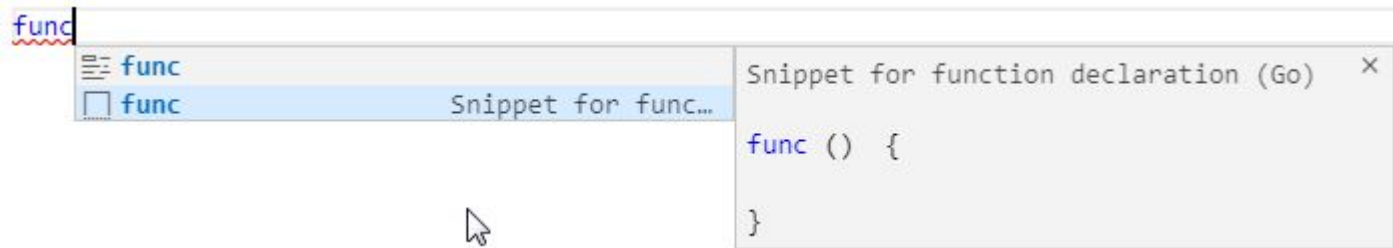
```
func <имя>(<параметр1> <тип параметра>,... , <параметрN> <тип параметра>) <тип результата> {  
    <тело функции>  
}
```

Выглядит сложно? Давайте разбираться по порядку.



# Функции

Начнём, как с `main`: пишем `func` (с помощью стрелки вниз выбираем `Snippet func` и нажимаем `Tab`):



Получаем:

```
func () {  
}
```







# Функции

Теперь у нас есть две функции:

- `main` - "служебная", её запускает сам Go
- `calculateBonus` - наша функция, по расчёту бонуса

Если мы сейчас запустим программу, то ничего не увидим. Почему? Просто потому, что `main` запускает сам Go, а вот `calculateBonus` никто не запускает. Это как функция "позвонить" на вашем смартфоне: пока вы не нажали на кнопку вызова, никакой звонок с вашего смартфона не начинается.



# Функции

Чтобы вызвать функцию, надо написать имя функции, а после него круглые скобки (обязательно используйте автодополнение):

```
func main() {  
    calculateBonus()  
}  
  
func calculateBonus() {  
    amount := 666  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    fmt.Println(bonus)  
}
```



# Функции

Теперь всё работает, но давайте разберёмся "как".



# Debugger

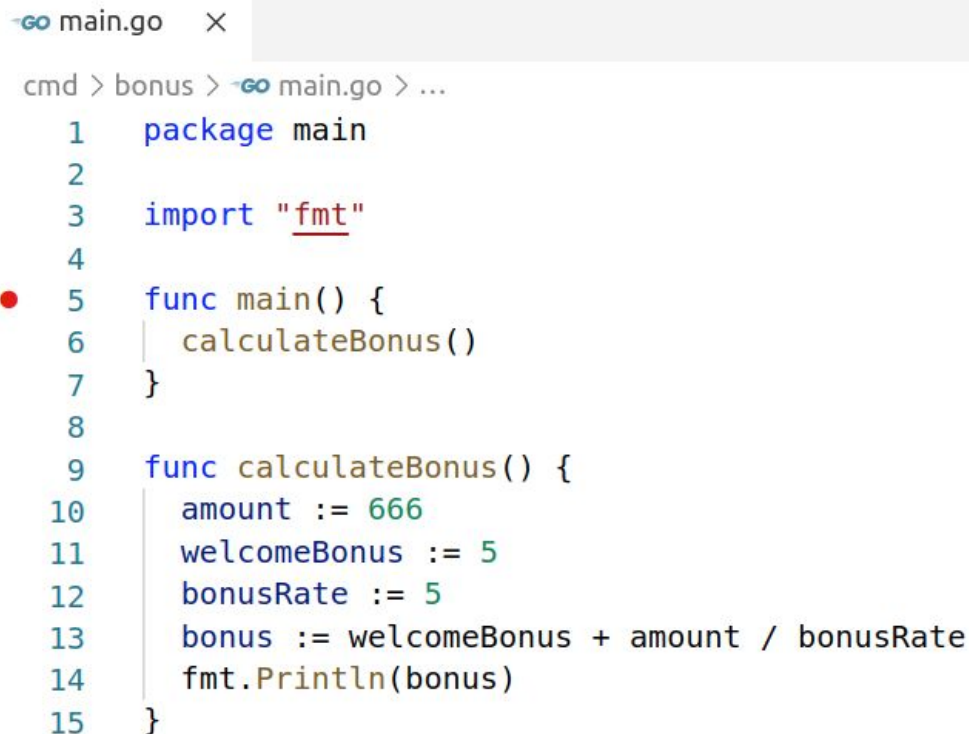
Для того, чтобы увидеть, как Go исполняет наш файл (и исполняет ли вообще), есть специальный инструмент, который называется Debugger (отладчик).

Отладчик – это специальный инструмент, который позволяет перевести Go в режим пошагового выполнения. При этом мы можем смотреть, что и как выполняется.



# Debugger

Открываем файл `main.go` и на в области нумерации строк кликаем левой кнопкой мыши (либо клавиша `F9`) - поставится точка остановки (breakpoint):



```
cmd > bonus > main.go > ...  
1  package main  
2  
3  import "fmt"  
4  
5  func main() {  
6      calculateBonus()  
7  }  
8  
9  func calculateBonus() {  
10     amount := 666  
11     welcomeBonus := 5  
12     bonusRate := 5  
13     bonus := welcomeBonus + amount / bonusRate  
14     fmt.Println(bonus)  
15 }
```

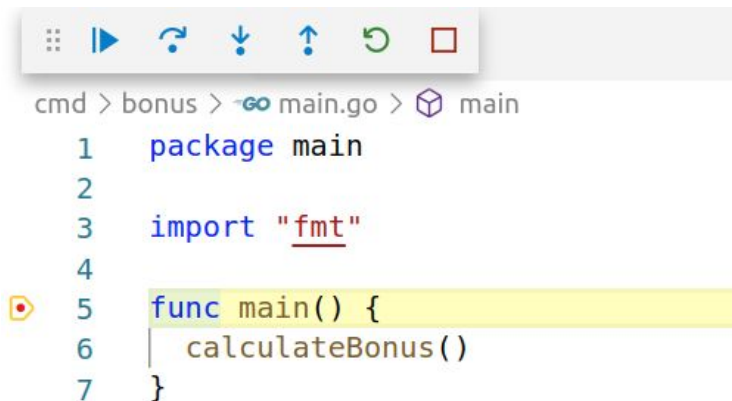
The screenshot shows a code editor window titled 'main.go'. The code is a Go program with a `main` function and a `calculateBonus` function. A red circle with the number '1' and an arrow points to a black dot (breakpoint) on line 5, at the start of the `func main()` line.

Точка остановки – это строка, на которой остановится выполнение. Для того, чтобы её активировать, нужно запустить приложение в режим отладки (`F5`).




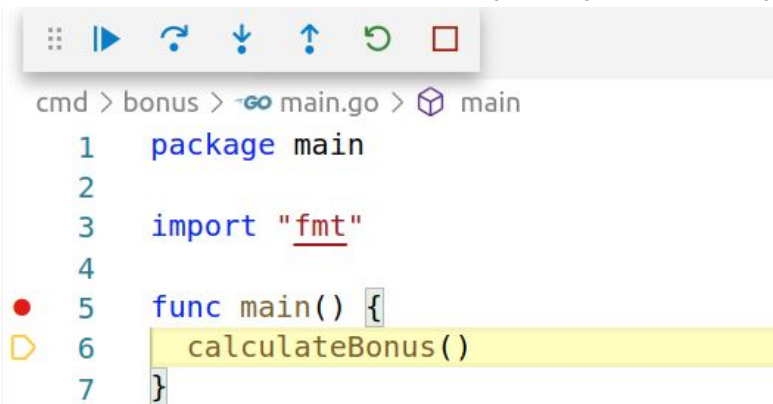
# Debugger

Мы увидим строку, подсвеченную жёлтым - это значит, что эту строку Go ещё не исполнил:



```
cmd > bonus > -go main.go > main
1  package main
2
3  import "fmt"
4
5  func main() {
6      calculateBonus()
7  }
```

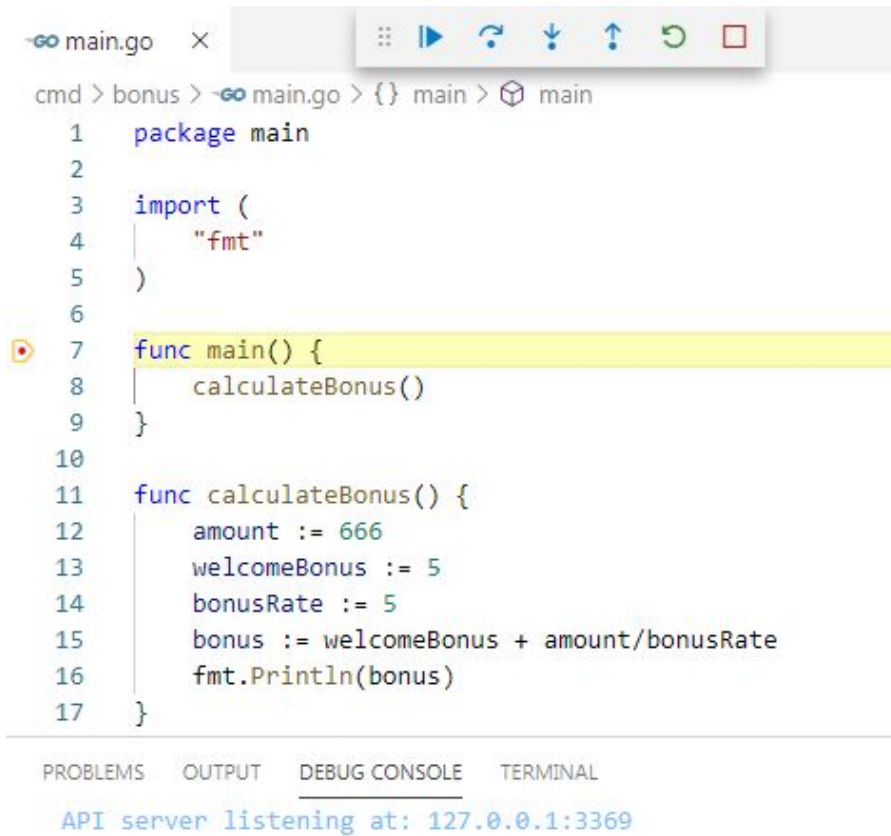
Чтобы перейти на следующую строку, нужно нажать на кнопку  или F10:



```
cmd > bonus > -go main.go > main
1  package main
2
3  import "fmt"
4
5  func main() {
6      calculateBonus()
7  }
```



# Debugger




```
main.go x [Debugger toolbar: Run, Step Into, Step Over, Step Out, Continue, Breakpoint, Close]
```


```
cmd > bonus > -go main.go > {} main > main
```

```
1 package main
2
3 import (
4     "fmt"
5 )
6
7 func main() {
8     calculateBonus()
9 }
10
11 func calculateBonus() {
12     amount := 666
13     welcomeBonus := 5
14     bonusRate := 5
15     bonus := welcomeBonus + amount/bonusRate
16     fmt.Println(bonus)
17 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

API server listening at: 127.0.0.1:3369

Если мы просто будем нажимать на кнопку  то ничего интересного не произойдёт, мы просто "прошагаем" вызов функции, хотя в консоли число 138 появится.

Всё дело в том, что  не заходит внутрь функций.



# Debugger


Чтобы мы увидели, как на самом деле всё происходит, есть два варианта:

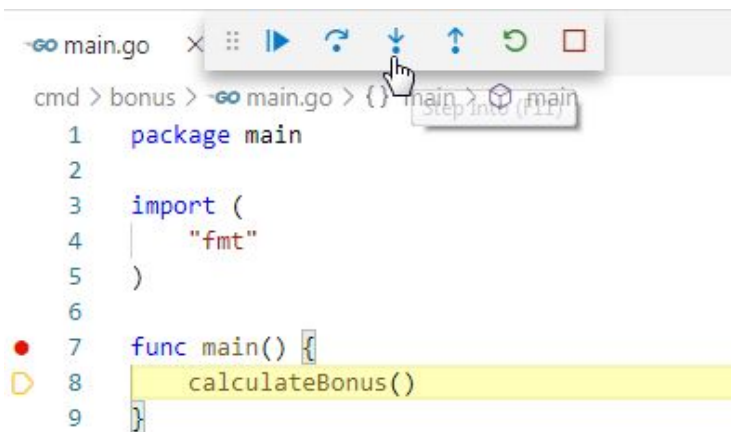
1. Поставить точку остановки внутри `calculateBonus`:

```

7 func main() {
8     calculateBonus()
9 }
10
11 func calculateBonus() {
12     amount := 666
13     welcomeBonus := 5
14     bonusRate := 5
15     bonus := welcomeBonus + amount/bonusRate
16     fmt.Println(bonus)
17 }

```

2. При нахождении на 8-ой строке нажать на кнопку  :





# Debugger

И в том, и в другом случае мы увидим, что "прошагав" всю функцию `calculateBonus` мы вернулись в `main`:

```
func main() {  
    calculateBonus()  
}  
  
func calculateBonus() {  
    amount := 666  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    fmt.Println(bonus)  
}
```

A diagram illustrating the execution flow of the code. A red line starts at the opening curly brace of the `main()` function, goes down to the `calculateBonus()` call, then turns right and goes down to the opening curly brace of the `calculateBonus()` function. From there, it goes down to the closing curly brace of `calculateBonus()`, then turns left and goes up to the closing curly brace of `main()`, indicating a return to the caller.

# Debugger

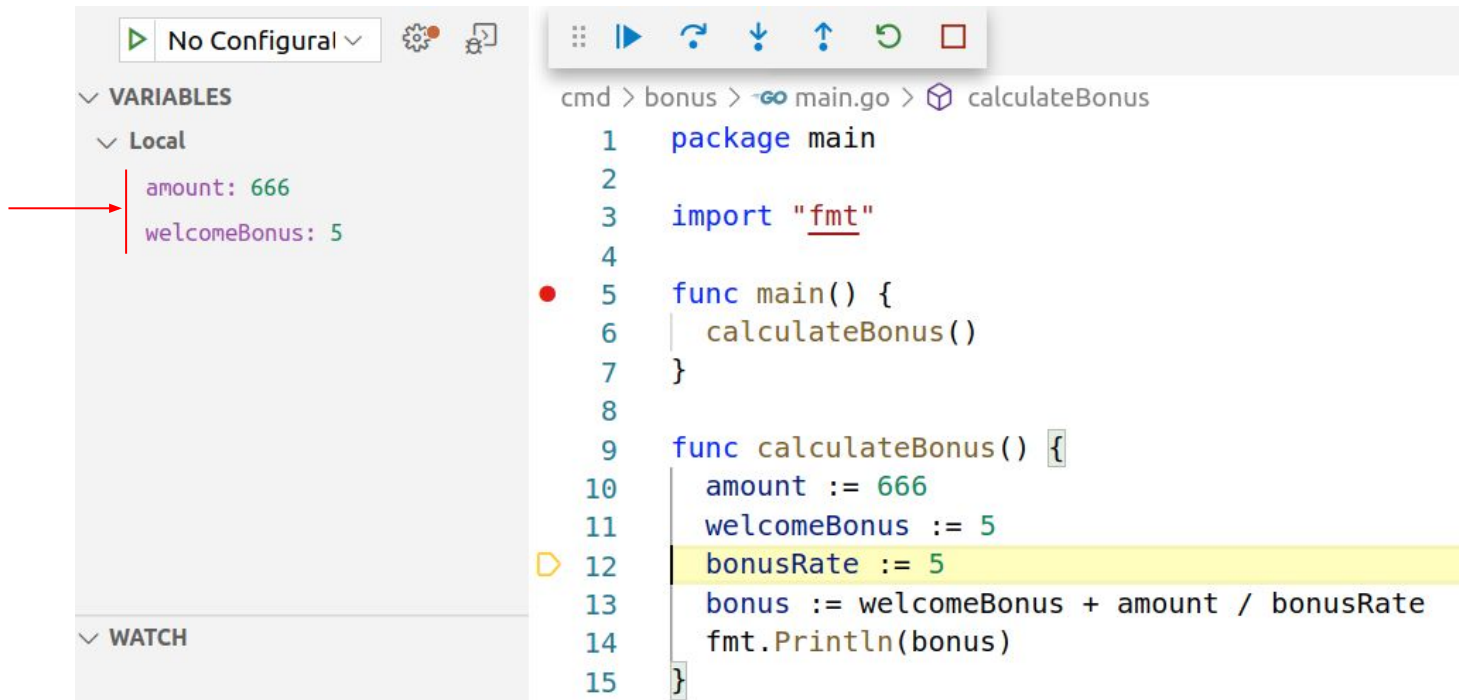
Нужно отметить, что не обязательно по шагам начинать с `main`, вы можете поставить точку остановки внутри `calculateBonus` и при начале отладки попадёте сразу туда:

```
7 func main() {  
8     calculateBonus()  
9 }  
10  
11 func calculateBonus() {  
12     amount := 666  
13     welcomeBonus := 5  
14     bonusRate := 5  
15     bonus := welcomeBonus + amount/bonusRate  
16     fmt.Println(bonus)  
17 }
```




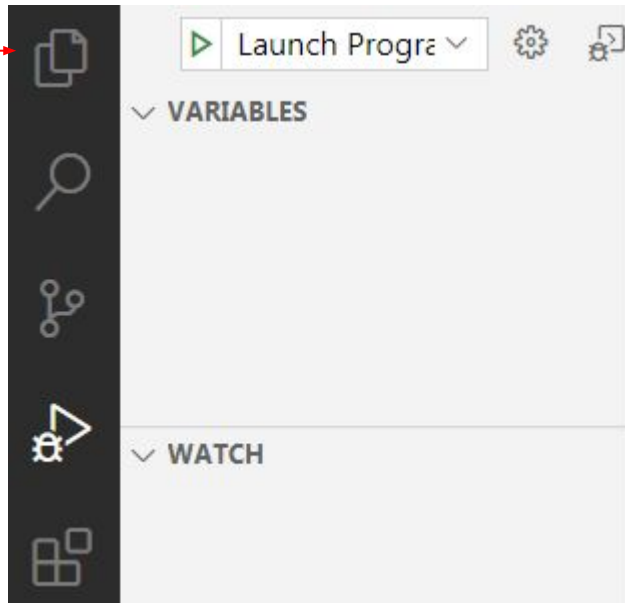
# Debugger

Когда в процессе исполнения создаются переменные, вы будете их видеть в боковой панели:



# Debugger

После того, как вы дошли до последней строки, нужно нажать на кнопку  или **F5**, чтобы не проваливаться в код Go.



Чтобы обратно переключиться в режим отображения файлов, просто выберите в боковой панели файловый менеджер

Таким образом, Debugger позволяет нам прошагать всю программу, и понять, как она выполняется на самом деле.

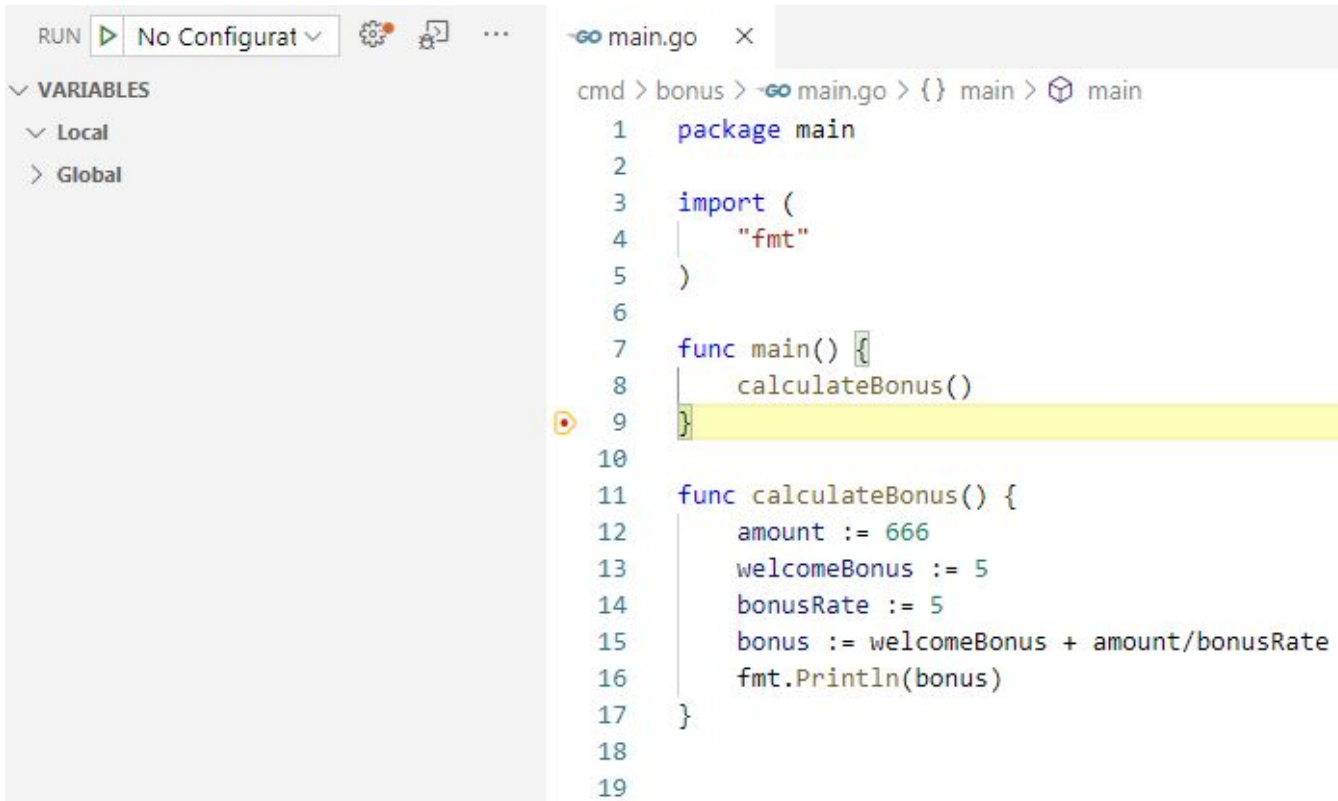


# ПЕРЕМЕННЫЕ



# Переменные

Теперь давайте внимательно посмотрим на панельку с переменными:



The screenshot shows an IDE window with a Go file named `main.go`. The code is as follows:

```
cmd > bonus > -go main.go > {} main > main
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      calculateBonus()
9  }
10
11 func calculateBonus() {
12     amount := 666
13     welcomeBonus := 5
14     bonusRate := 5
15     bonus := welcomeBonus + amount/bonusRate
16     fmt.Println(bonus)
17 }
18
19
```

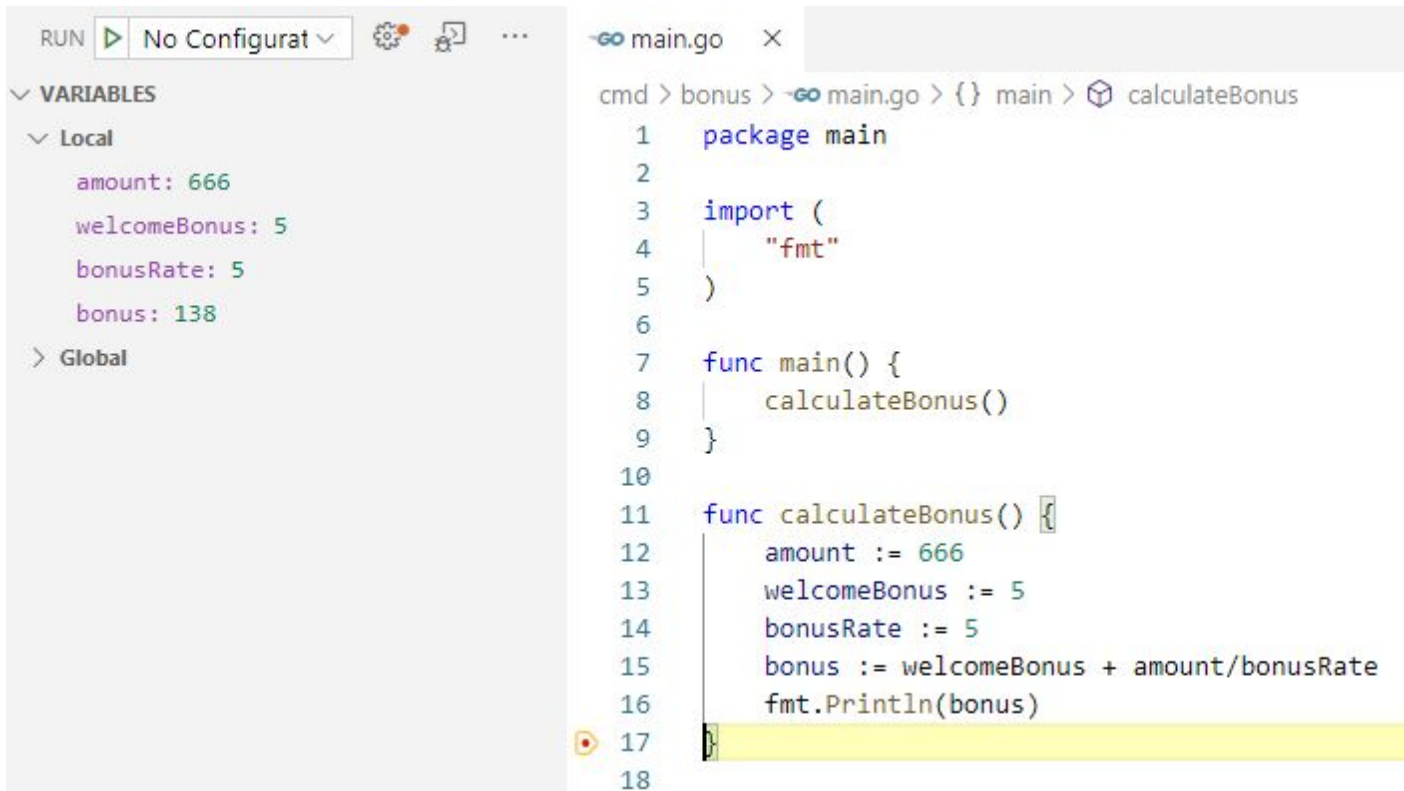
On the left side, there is a 'VARIABLES' panel. It has a 'RUN' button and a dropdown menu showing 'No Configurat'. Under the 'VARIABLES' heading, there are two sections: 'Local' and 'Global', both of which are currently empty.

Мы сейчас находимся в конце `main`, но никаких переменных нет (панелька Variables Local пустая)



# Переменные

Теперь давайте внимательно посмотрим на панельку с переменными:



The screenshot shows a Go IDE interface. On the left, the 'VARIABLES' panel is expanded, showing local variables: `amount: 666`, `welcomeBonus: 5`, `bonusRate: 5`, and `bonus: 138`. The 'Global' section is collapsed. On the right, the source code for `main.go` is displayed. The code defines a `main` package, imports `fmt`, and contains a `main` function that calls `calculateBonus`. The `calculateBonus` function is defined with local variables `amount`, `welcomeBonus`, `bonusRate`, and `bonus`, which are assigned the same values as in the variables panel. The `calculateBonus` function calculates `bonus` as `welcomeBonus + amount/bonusRate` and prints it. The line `calculateBonus()` in the `main` function is highlighted in yellow.

```
cmd > bonus > -go main.go > {} main > calculateBonus
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      calculateBonus()
9  }
10
11 func calculateBonus() {
12     amount := 666
13     welcomeBonus := 5
14     bonusRate := 5
15     bonus := welcomeBonus + amount/bonusRate
16     fmt.Println(bonus)
17 }
18
```

А внутри `calculateBonus` всё есть. О чём это говорит?



# Переменные

Это говорит о том, что имена, объявленные внутри функции, доступны только внутри этой функции.

Т.е. внутри `main` недоступны ни `amount`, ни `welcomeBonus`, ни другие переменные.





# **ПАРАМЕТРЫ И РЕЗУЛЬТАТ**



# Вызов функции

Теперь, когда у нас есть функция, мы можем вызывать её столько раз, сколько ХОТИМ:

```
7 func main() {  
8     calculateBonus()  
9     calculateBonus()  
10    calculateBonus()  
11 }  
12  
13 func calculateBonus() {  
14     amount := 666  
15     welcomeBonus := 5  
16     bonusRate := 5  
17     bonus := welcomeBonus + amount/bonusRate  
18     fmt.Println(bonus)  
19 }
```

Но сколько бы раз мы её не вызывали, результат всегда будет 138.

Потому что каждый раз, как мы запускаем `calculateBonus`, в ней заново создаются переменные `amount` и т.д. и им присваются одни и те же значения.

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

138  
138  
138



# Параметры

Чтобы это изменить, мы должны объявить входные параметры. Но для этого нужно сначала понять - что может быть входными параметрами? Например, в `Println` это была строка, которую мы хотели напечатать, а в `ReplaceAll` - строка, на базе которой производятся замены, что меняем и на что. Т.е. в параметры чаще всего выносят то, что изменяется.

Что должно изменяться в нашей функции? Конечно же, для каждого пользователя разным будет количество потраченных средств (т.к. начальный бонус и "стоимость" одного бонуса фиксированы).



# Параметры

Что мы делаем? Мы просто `amount` пишем внутри круглых скобок и пишем для него тип (`int`):

```
func calculateBonus() {  
    amount := 666  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    fmt.Println(bonus)  
}
```



```
func calculateBonus(amount int) {  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    fmt.Println(bonus)  
}
```

Теперь внутри `main` ошибки:

```
import (  
    "fmt"  
)
```

```
func main() {  
    calculateBonus()  
    calculateBonus()  
    calculateBonus()  
}
```

not enough arguments in call to calculateBonus  
have ()  
want (int) go

Peek Problem (Alt+F8) No quick fixes available



# Параметры

Компилятор Go достаточно строгий: если вы сказали, что у функции должен быть параметр, то вы должны его передавать при вызове.

Поэтому давайте передадим:

```
7 func main() {  
8     calculateBonus(0)  
9     calculateBonus(1)  
10    calculateBonus(55)  
11 }  
12  
13 func calculateBonus(amount int) {  
14     welcomeBonus := 5  
15     bonusRate := 5  
16     bonus := welcomeBonus + amount/bonusRate  
17     fmt.Println(bonus)  
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

5  
5  
16

Здесь есть особый момент: когда вы вызываете функцию, то, что в круглых скобках (0, 1 и 55) называют аргументом, а когда объявляете (`amount int`) называют параметром.

Но достаточно часто эти термины используют как взаимозаменяемые (мы будем делать так же).



# Параметры

Теперь давайте попробуем понять, что происходит. По факту, когда мы пишем `calculateBonus(0)`, происходит следующее\*:

```
func calculateBonus() {  
    amount := 0  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    fmt.Println(bonus)  
}
```

Грубо говоря, то, что мы пишем в скобках при вызове, просто "копируется" в "локальную" переменную внутри функции.

Когда `calculateBonus(1)`:

```
func calculateBonus() {  
    amount := 1  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    fmt.Println(bonus)  
}
```

Про "копируется" надо запомнить.

Примечание\*: конечно же внутри всё гораздо сложнее, но если на первых порах вы будете представлять это так, вам будет значительно проще.



# Результат

Теперь, когда мы разобрались, как работают параметры, давайте подумаем о том, что не так с нашей функцией? Она печатает всё в консоль. Это очень ограничивает её применение (мы же собираемся писать сервера, а не консольные утилиты\*).

Представьте, что Алиф.Моби или приложение Мегафона печатают баланс счёта или сумму бонусов в консоль. Где у телефона консоль? Может, она и существуют, но в обычной жизни вы её точно не видите.

Примечание\*: утилитами называют небольшие вспомогательные программки.



# Результат

Задача нашей функции - вычислять и нам число бонусов. Т.е. она должна возвращать результат. Для этого у нас есть ключевое слово `return`, которое переводится как вернуть:

```
func calculateBonus() {  
    amount := 1  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    return bonus  
}
```

Но и тут ошибка! Давайте смотреть, что нам пишут:

```
}  
func  
    bonus int  
    too many arguments to return  
        have (int)  
        want () go  
    Peek Problem (Alt+F8) No quick fixes available Rate  
    return bonus  
}
```





# Результат

О чём идёт речь? О том, что если мы собираемся что-то возвращать, то должны об этом явно объявить, чтобы все пользователи нашей функции знали об этом:

```
func calculateBonus(amount int) int {  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    return bonus  
}
```



# Результат

Теперь, благодаря тому, что мы возвращаем что-то из функции, на то место, где стоял вызов функции подставляется то, что написано в **return**:

```
func main() {  
    bonus := calculateBonus(0)  
    fmt.Println(bonus)  
}  
  
func calculateBonus(amount int) int {  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    return bonus  
}
```

Т.е. после выполнения функции на место её вызова подставится конкретное значение:

```
func main() {  
    bonus := 5  
    fmt.Println(bonus)  
}
```



# Переменные

Обратите внимание, что `bonus` в `main` и `bonus` в `calculateBonus` - это разные переменные (даже несмотря на то, что они называются одинаково - они связаны только тем, что в `bonus` в `main` записывается число, по факту копируется, которое было в `bonus` в `calculateBonus`).



# Вызов

Можно не объявлять дополнительную переменную `bonus` в `main` и сократить всё до:

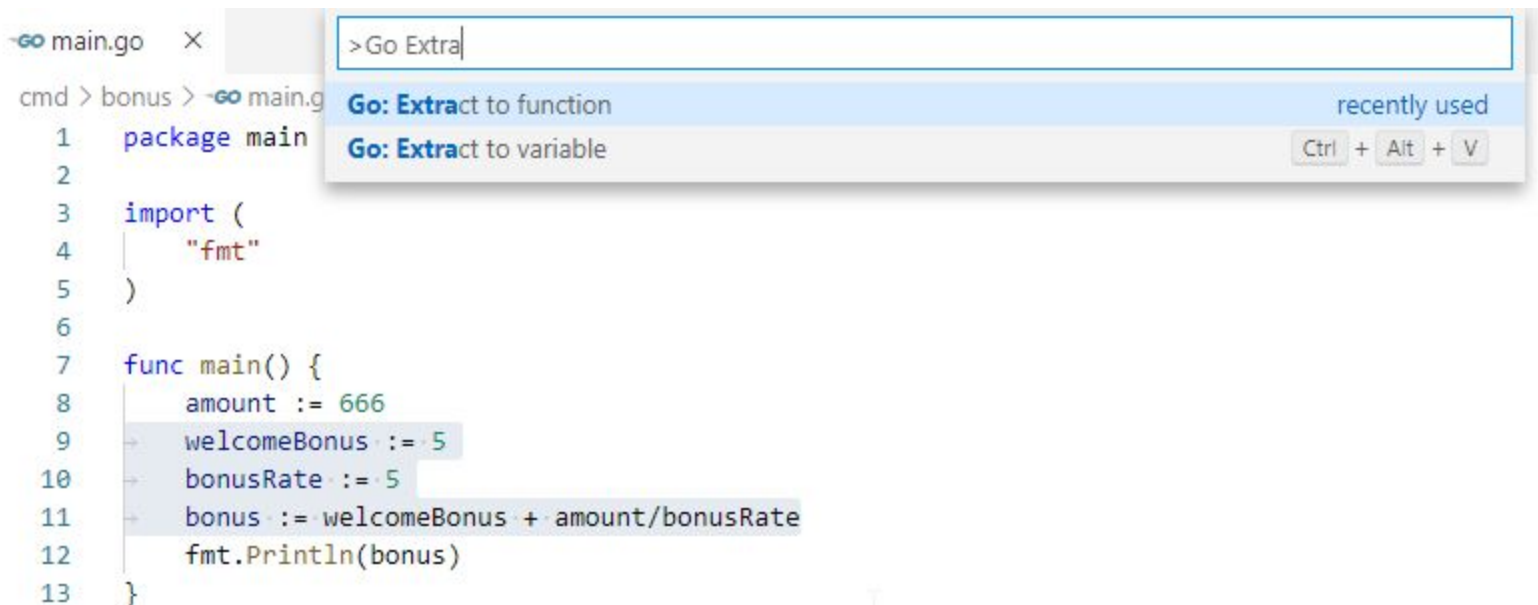
```
func main() {  
    fmt.Println(calculateBonus(0))  
}  
  
func calculateBonus(amount int) int {  
    welcomeBonus := 5  
    bonusRate := 5  
    bonus := welcomeBonus + amount/bonusRate  
    return bonus  
}
```

Как это будет работать? Сначала вычислится `calculateBonus(0)` (т.е. самые вложенные скобки), на место вызова этой функции подставится результат `5` и получится `fmt.Println(5)`.



# VS Code

Умение работать со средой разработки может сэкономить вам кучу времени. В VS Code по сочетанию **Ctrl + Shift + P** открывается панель команд, которая позволяет вам быстро выполнять разные действия. Команды, относящиеся к Go, начинаются с **Go:**. Например, мы можем вернуться, к первому варианту, выделить код и выбрать **Extract to function**:



The screenshot shows the VS Code editor with a Go file named `main.go`. The code contains a `package main`, an `import` for `fmt`, and a `func main()` block. Inside the `main` function, lines 9 through 11 are selected: `welcomeBonus := 5`, `bonusRate := 5`, and `bonus := welcomeBonus + amount/bonusRate`. The Command Palette is open, showing the search input `>Go Extra` and two results: `Go: Extract to function` (highlighted) and `Go: Extract to variable`. The `recently used` label and the `Ctrl + Alt + V` shortcut are also visible.

```
cmd > bonus > -go main.g
1  package main
2
3  import (
4      "fmt"
5  )
6
7  func main() {
8      amount := 666
9      welcomeBonus := 5
10     bonusRate := 5
11     bonus := welcomeBonus + amount/bonusRate
12     fmt.Println(bonus)
13 }
```

>Go Extra

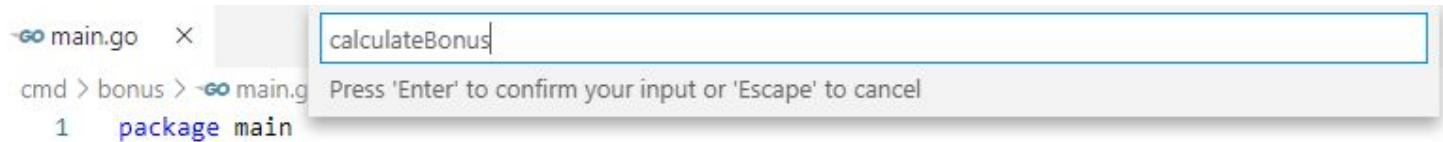
Go: Extract to function recently used

Go: Extract to variable Ctrl + Alt + V



# VS Code

Затем ввести название функции и нажать **Enter**:



```
-go main.go X calculateBonus|
cmd > bonus > -go main.g Press 'Enter' to confirm your input or 'Escape' to cancel
1 package main
```

Получим:

```
7 func main() {
8     amount := 666
9     bonus := calculateBonus()
10    fmt.Println(bonus)
11 }
12
13 func calculateBonus() int {
14     amount := 666
15     welcomeBonus := 5
16     bonusRate := 5
17     bonus := welcomeBonus + amount/bonusRate
18     return bonus
19 }
```

Не идеально, но поправить **amount** проще,  
чем всё писать руками.



# VS Code

Там же можно найти добавление импорта (по имени пакета), переход в пакет (легко перейти в `builtin`, `strings` и т.д.) и другие интересные команды.



# ПАКЕТЫ





# Пакеты

Если мы вернёмся к Standard Go Project Layout там будет написано следующее: если вы считаете, что какой-то код может быть переиспользован, то он не должен размещаться в `cmd`, он должен размещаться в `pkg` (сокращённо от `package`).

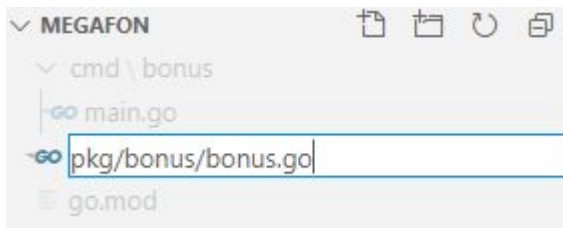
Созданная нами функция `calculateBonus` относится к категории таких функций. А если быть конкретнее она относится к бизнес-логике. Бизнес-логика - это код, который отвечает за предметную область. Что такое предметная область? Когда мы говорим про Мегафон, то предметная область - это тарифы, счета, минуты, мегабайты, SMS и т.д. При этом `main` к предметной области не относится - он просто запускает нашу функцию, передавая туда параметры, и выводит в консоль результат.

Обратите внимание: в предметной области и понятия такого нет - консоль.

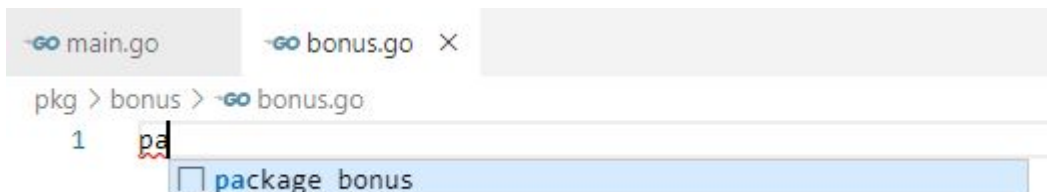


# Пакеты

Поэтому мы смело (**Ctrl + x**, **Ctrl + v**) переносим нашу функцию в файл `pkg/bonus/bonus.go`: (для этого нужно кликнуть на свободной области в панельке файлов, чтобы файл не создавался внутри `cmd` или `bonus`):



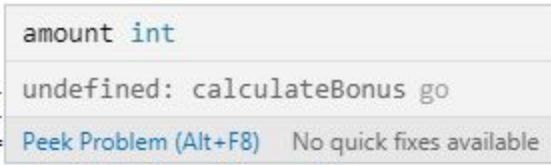
Обратите внимание: VS Code вам предложит вам имя пакета:



# Пакеты

Но теперь есть проблема: в main функция `calculateBonus` недоступна:

```
3 import (  
4     "fmt"  
5 )  
6  
7 func main() {  
8     amount :=   
9     bonus := calculateBonus(amount)  
10    fmt.Println(bonus)  
11 }
```



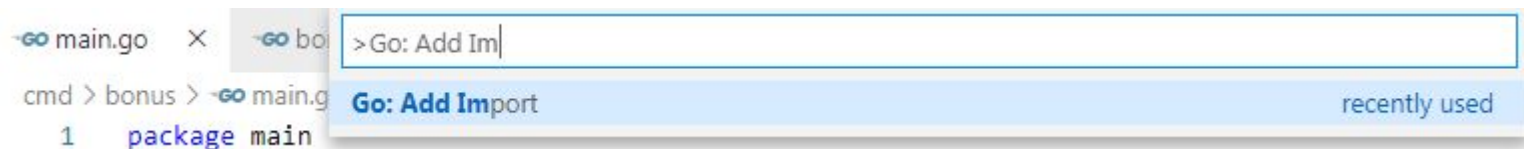
The image shows a Go code snippet with an IDE error tooltip. The code defines a `main` function that imports the `fmt` package and declares a variable `amount` of type `int`. It then attempts to call `calculateBonus(amount)`. The IDE highlights the `calculateBonus` function call with a red squiggly line, indicating it is undefined. The tooltip shows the variable `amount` is of type `int`, the error message is `undefined: calculateBonus go`, and a suggestion to `Peek Problem (Alt+F8)` with the note `No quick fixes available`.

Попытки импортировать её успехом тоже не увенчаются. Всё потому, что она написана с маленькой буквы: а в Go, всё, что написано с маленькой буквы в одном пакете, не доступно в другом.

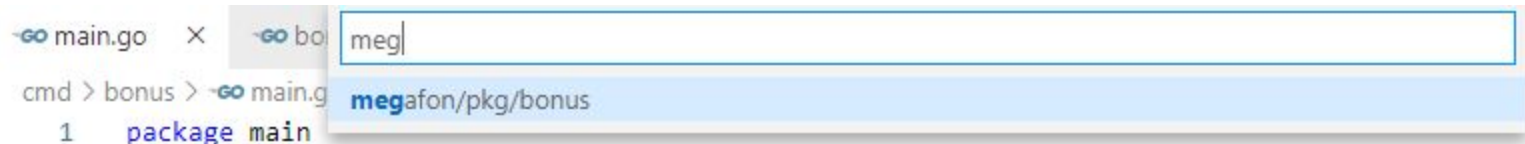


# Пакеты

Переназовём функцию и добавим ручную import с помощью **Ctrl + Shift + P**:



Обратите внимание: имя будет `megafon/pkg/bonus`:



# Пакеты

После этого мы спокойно можем использовать это имя:

```
cmd > bonus > go main.go > ...
```

```
1  package main
2
3  import (
4      "fmt"
5      "megafon/pkg/bonus"
6  )
7
8  func main() {
9      amount := 666
10     result := bonus.CalculateBonus(amount)
11     fmt.Println(result)
12 }
```

Обратите внимание: мы переименовали переменную **bonus** в **result**, чтобы имя модуля и имя переменной различались.



# go build ./...

Если мы попробуем запустить `go build ./...` в надежде получить `bonus.exe` - то этого файла не появится, но Go честно пройдёт по всем пакетам, скомпилирует их (удостоверится, что они успешно компилируются) и удалит. Почему? Потому что `go build ./...` формирует исполняемый файл только если у вас один каталог с исходниками и там есть `main`.

А значит теперь мы снова возвращаемся к команде, которая позволяет получить исполняемый файл из конкретного каталога: `go build -o bonus cmd/bonus/main.go`.

Зачем же нам `go build ./...`? Он нужен для того, чтобы удостовериться, что весь ваш код успешно компилируется (а не только файл `main.go`, который вы указали, и все зависимости в нём).



# bonus.CalculateBonus

Последний штрих, связанный с пакетами заключается в том, что название `bonus.CalculateBonus` выглядит немного странно. В нём два раза фигурирует слово `bonus`.

В сообществе Go принято следующее правило: если в названии пакета уже фигурирует нужное по смыслу вам слово, то из названия функций его нужно убирать.

Т.е. наша функция будет называться `bonus.Calculate`. Вы можете легко переименовать имя с помощью клавиши **F2**.



# Важно

Соглашения очень важны в мире Go: если вы им не следуете, то ваш код просто не будут воспринимать и, скорее всего, просто не возьмут вас на работу, поскольку переучивать дороже, чем научить заново.

Поэтому внимательно относитесь к тем рекомендациям по именованию, которые мы даём.





**УСЛОВИЯ**



# Условия

Давайте вернёмся к программе с Мегафоном. Мы специально показали вызов функции с `amount = 0` и `amount = 1`. Т.е. получается что человек может ничего не потратить и уже получил 5 бонусных баллов.

Давайте немного изменим условия: сделаем так, чтобы бонусы начислялись только при условии, если человек подключился и потратил более 10 сомони.



# Условия

Обратите внимание, как мы говорим: если человек потратил больше, то делаем одно, иначе же - другое.

В языках программирования для таких случаев (когда в зависимости от условия нужно выполнить одну или другую логику) существуют специальные конструкции, которые называются условиями. В Go это выглядит следующим образом:

```
if <условие> {  
    <код, который выполняется, если условие верно>  
} else {  
    <код, который выполняется, если условие не верно>  
}
```



# Условия

Есть также сокращённая форма, которую мы будем использовать чаще всего:

```
if <условие> {  
    <код, который выполняется, если условие верно>  
}
```



# Условия

Что может быть в качестве условий? Это может быть любое выражение (включая вызов функции), которое возвращает тип `bool`.

Начнём мы с операторов сравнения:

- `>` (больше), `<` (меньше)
- `>=` (больше или равно), `<=` (меньше или равно)
- `==` (равно), `!=` (не равно)



# Условия

pkg > bonus > ~~go~~ bonus.go > ...

```
1  package bonus
2
3  func CalculateBonus(amount int) int {
4      minAmount := 10
5      bonus := 0
6      welcomeBonus := 5
7      bonusRate := 5
8
9      if amount > minAmount {
10         bonus = welcomeBonus + amount/bonusRate
11     } else {
12         bonus = 0
13     }
14
15     return bonus
16 }
```

Что здесь плохо:

1. Строка 12 - лишняя, т.к. `bonus` и так равен нулю
2. Чем больше у вас `{}`, тем сложнее читается код (и будет больше ошибок)



# Условия

```
pkg > bonus > go bonus.go > ...  
1  package bonus  
2  
3  func CalculateBonus(amount int) int {  
4      minAmount := 10  
5      bonus := 0  
6      welcomeBonus := 5  
7      bonusRate := 5  
8  
9      if amount > minAmount {  
10         bonus = welcomeBonus + amount/bonusRate  
11     }  
12  
13     return bonus  
14 }
```

Стало лучше, но не намного. По факту, мы делаем слишком много лишних действий: объявляем переменные, в которых не будет смысла, если `amount` будет меньше 10.



# Early Exit

Чтобы улучшить структуру нашего кода, мы можем использовать подход Early Exit (быстрый выход). Т.е. мы можем быстро проверить нужное нам условие и выйти, если нас что-то не устраивает:

```
pkg > bonus > go bonus.go > ...  
1  package bonus  
2  
3  func CalculateBonus(amount int) int {  
4      minAmount := 10  
5  
6      if amount < minAmount {  
7          return 0  
8      }  
9  
10     bonus := 0  
11     welcomeBonus := 5  
12     bonusRate := 5  
13     bonus = welcomeBonus + amount/bonusRate  
14  
15     return bonus  
16 }
```

Обратите внимание: мы поменяли знак условия и написали `return 0`. Т.е. если `amount < minAmount`, мы возвращаем 0.

Что значит возвращаем? Это значит, что никакой другой код внутри функции выполнен не будет, и сразу с 7-ой строки мы вернёмся туда, откуда вызывали функцию `CalculateBonus`.





# Early Exit

Early Exit будет повсеместно использоваться в Go, поэтому привыкайте к нему с первых дней.



# ИТОГИ



# Итоги

В этой лекции мы обсудили:

- создание функций
- передачу параметров и возврат значений
- разделение на пакеты
- компиляцию при наличии нескольких пакетов
- условия и early exit

Полученных вами знаний хватит для решения ДЗ. Конечно же, и пакеты, и функции, и условия обладают ещё многими возможностями, которые мы ещё не изучили, но мы следуем принципу "изучил - сразу примени". Поэтому раскрывать их возможности будем по мере прохождения курса.



# ДОМАШНЕЕ ЗАДАНИЕ



# Орг.моменты

Курс состоит из 33 обязательных занятий. Каждый **вторник 19:00 по Душанбе дедлайн** сдачи домашнего задания.

Если не успеете сдать в срок домашнее задания, тогда этот курс будет для вас закончен и вы сможете зарегистрироваться на запуск следующего через несколько месяцев.



# ДЗ №1: Комиссия за перевод

В рамках [alif.mobi](#) есть [услуга перевода электронных денежных средств на карты НПС "Корти милли"](#) (такие комиссии есть у большинства банков):

## Операции

Перевод электронных денежных средств  
на карты НПС «Корти милли»  
(до 5 000 сомони в календарный месяц)

0%

Перевод электронных денежных средств на  
карты НПС «Корти милли»  
(свыше 5 000 сомони в календарный месяц)

0,5% (мин. 0,25 с.)



# ДЗ №1: Комиссия за перевод

Вам необходимо написать приложение, в котором есть функция `Calculate`, которая по переданной сумме (сумма переводов за весь месяц), высчитывает, сколько заплатил пользователь комиссии. Функция `Calculate` должна возвращать результат в дирамах.

Пример вызова функции:

```
import (  
    "mobi/pkg/commission"  
    "fmt"  
)  
  
func main() {  
    result := commission.Calculate(9_999_99) // расходы так же передаются в дирамах  
    fmt.Println(result)  
}
```



# ДЗ №1: Комиссия за перевод

`main.go` не должен принимать никаких аргументов командной строки (его содержимое должно соответствовать предыдущему слайду (не забудьте указать пакет)).

`main.go` должен располагаться в каталоге `/cmd/commission`.

Каталог с проектом и модуль должны называться `mobi`.





# ДЗ №2: Оплата за Интернет

Мегафон Таджикистан предлагает [линейку тарифов M new](#):

## Базовые интернет пакеты

Выбирай свой пакет в новой линейке «M new» и пользуйся Интернетом на неограниченной скорости в том объеме, который тебе удобен!

Пакет серии «M new»	Включенный в Пакет объем трафика (Мб)	Абонентская плата	USSD-команда для подключения
M new 1000	1 000 Мб	35,00 сомони	*411*1#
M new 2000	2 000 Мб	55,00 сомони	*411*2#
M new 3000	3 000 Мб	70,00 сомони	*411*3#
M new 5000	5 000 Мб	95,00 сомони	*411*4#
M new 10 000	10 000 Мб	170,00 сомони	*411*5#



# ДЗ №2: Оплата за Интернет

Мы хотим, что вы написали программу, которая высчитывает, сколько пользователь должен заплатить за интернет исходя из количества потраченных им мегабайт (абонентская плата + дополнительные расходы).

Напишите 5 функций:

- `Calculate1000`
- `Calculate2000`
- `Calculate3000`
- `Calculate5000`
- `Calculate10000`

Каждая из которых считает сумму для соответствующего тарифа (на следующей лекции мы научимся писать одну и расскажем, когда это имеет смысл, а когда - нет).



# ДЗ №2: Оплата за Интернет

Обратите внимание:

## Особенности

- Срок действия интернет пакетов серии «М new» равен 30 дням с момента подключения пакета, независимо от даты подключения.
- По истечении первого месяца, интернет пакет серии «М new» не требует дополнительного подключения и продлевается автоматически, при наличии на лицевом счете суммы, достаточной для снятия абонентской платы в соответствии стоимостью интернет пакета.
- В случае если на момент снятия абонентской платы, баланс лицевого счёта является недостаточным для снятия абонентской платы в полном размере в соответствии со стоимостью интернет пакета, предоставление услуги «Передача данных» временно приостанавливается до момента пополнения лицевого счета. В этом случае дата начала отчетного периода для снятия абонентской платы изменяется на новую дату, т.е. в момент пополнения баланса лицевого счёта и возобновления предоставления услуги «Передача данных». Снятие начислений по абонентской плате происходит в момент включения услуги «Передача данных» (после пополнения баланса лицевого счёта абонента).
- Стоимость 1 мб после окончания трафика на пакетах серии «М new» и дополнительного пакета «М + new» равна 0,06 сомони за 1 Мб.



# ДЗ №2: Оплата за Интернет

Пример вызова функции:

```
import (  
    "megafon/pkg/billing"  
    "fmt"  
)  
  
func main() {  
    result := billing.Calculate1000(9999) // трафик передаётся в мегабайтах, результат должен быть в дирамах  
    fmt.Println(result)  
}
```



# ДЗ №2: Оплата за Интернет

[main.go](#) не должен принимать никаких аргументов командной строки (его содержимое должно соответствовать предыдущему слайду (не забудьте указать пакет)).

[main.go](#) должен располагаться в каталоге [/cmd/billing](#).

Каталог с проектом и модуль должны называться [megafon](#).



Спасибо за внимание

alif academy

2021

