

КОНСПЕКТ ЛЕКЦИИ · ЛЕКТИКА

Типы данных и управление потоком в Go

Длительность: 1 ч 29 мин

Подготовлено: 2026-10-17 20:03

Кратко

Занятие по Go посвящено типам данных и управлению потоком выполнения. Деньги в сервисе заказов предлагается хранить в целых копейках в `int64`, потому что дробные типы дают погрешность. Далее разбираются целые и дробные типы, `byte` и `rune`, явные преобразования, переполнение, строки в UTF-8, нулевые значения, объявления через `var`, `:=` и `const` с `iota`. Затем рассматриваются циклы `for`, ветвления `if` и `switch`, а также защита от пустого среза и переполнения. Занятие завершается разбором задачи `ProcessOrder` и домашним заданием, которое проверяется автотестами.

Главные выводы

- ★ Деньги хранят в целых минимальных единицах (копейках) в `int64`, а не во `float`.
- ★ Тип выбирают под данные: диапазон, точность и поведение при переполнении зависят от типа.
- ★ Go не смешивает числовые типы автоматически: преобразования пишутся явно, и потеря точности видна в коде.
- ★ Строка в Go — последовательность байт; для работы с символами используют руны.
- ★ Переменная без значения получает нулевое значение своего типа; `iota` удобен для перечислений.
- ★ Управление потоком строится на `if`, `for` и `switch`; `for` покрывает счётчик, условие и перебор.
- ★ Пустые входы и границы типов проверяют до операции, например деление на `len`.
- ★ Ошибка в Go — обычное возвращаемое значение, которое нужно проверять явно.

Оглавление

1. Организация занятия, домашнее задание и план курса	05:00
2. Какой тип выбрать для денег	07:15
3. Целые и дробные типы, byte, rune и преобразования	11:30
4. Ошибки выбора типа, строки и руны	18:20
5. Нулевые значения, объявления и константы	26:30
6. Циклы for	43:00
7. Map, if с инициализацией и ошибка как значение	48:00
8. Ветвление через switch	59:10
9. Пустой срез и защита от переполнения	01:04:10
10. Ошибки как значения, путь алгоритма и задание	01:08:50
11. Разбор задачи ProcessOrder	01:14:40
12. Остальные задачи, рефлексия и итоги	01:23:00

Задания и сроки

- Выполнить обязательные простые задания домашней работы, проверяя их запуском тестов
- Выполнить более сложные задания домашней работы по желанию
- Создать публичный репозиторий на GitHub и загрузить в него решение домашней работы
- Сообщить преподавателю о трудностях с домашним заданием

Открытые вопросы

- 21:35 Как работает тип char из C++ и похож ли он на rune в Go?

Конспект

ГЛАВА 1

Организация занятия, домашнее задание и план курса

05:00 – 07:15

Занятие открывает блок о типах данных и управлении потоком. Показано, зачем нужна тема и что уже пройдено. Сообщены правила домашнего задания и тема следующего занятия.

Тема занятия

Занятие посвящено типам данных и управлению потоком: переменным, константам, ветвлениям `if` и `switch` и циклу `for`. Первый практический вопрос занятия: в каком типе хранить деньги в заказе.

Зачем это нужно

Выбор типа для хранения денег — практическая задача: она встречается в любом сервисе, который работает с заказами и ценами.

Цели занятия

- **Хранить значения** в переменных и константах.
- **Сравнивать значения** и выбирать ветку выполнения.
- **Повторять действия** циклом.

Главная цель: научиться писать небольшие алгоритмы, которые работают с данными и учитывают их границы.

Что уже пройдено

1. Создание модуля.
2. Запуск программы.
3. Запуск тестов.
4. Форматирование кода.

Тесты понадобятся и в домашней работе: по ним видно, правильно ли написана функция.

Домашнее задание

Пункт	Содержание
Задание	Закрепить типы, циклы и ветвления. Не меняя сигнатуры функций и автотесты, решить задачи из репозитория. Проверить преобразования типов и границы значений
Сдача	Ссылка на ветку в репозитории
Оценивание	Процент пройденных автотестов

Важно

Задания со слайда (FizzBuzz через `switch`, `min`, `max` и сумма среза, среднее без потери точности, обработка пустого входа, автотесты до PASS) простые и обязательны. Задания из репозитория сложнее и выбираются по желанию.

Задачи лежат в репозитории на GitHub, решение кладут в публичный репозиторий. Правильность проверяют автотесты, общие вопросы разбирают на следующем занятии.

Следующее занятие

Функции, указатели, `slice`, `map` и `struct`, а также разбор домашнего задания.

ГЛАВА 2

Какой тип выбрать для денег

07:15 – 11:30

Обсуждается, какой тип использовать для цены заказа: `float64`, `int` или `int64`. Показано, что дробная арифметика даёт погрешность, поэтому деньги хранят в копейках в целом типе `int64`.

Какой тип выбрать для цены заказа

Цену можно хранить в `float64`, `int` или `int64`. Мнения слушателей разделились. Чтобы выбрать, важно понять, как каждый тип влияет на точность и диапазон.

Почему не дробные числа

Тип `float64` хранит около 15–17 значащих цифр, и не каждое дробное число в нём представимо точно. Для одного заказа погрешность незаметна. В сервисе с сотнями тысяч заказов в день она накапливается, и суммы начинают расходиться.

Посмотрим на это в коде:

```

package main

import "fmt"

func main() {
    a := 0.1
    b := 0.2

    sum := a + b

    fmt.Printf("a:   %.17f\n", a)
    fmt.Printf("b:   %.17f\n", b)
    fmt.Printf("sum:  %.17f\n", sum)
    fmt.Printf("0.3:  %.17f\n", 0.3)

    fmt.Println(sum == 0.3)
}

```

Вывод программы:

```

a:   0.100000000000000001
b:   0.200000000000000001
sum: 0.300000000000000004
0.3: 0.29999999999999999
false

```

Сумма 0.1 и 0.2 не равна 0.3. Поэтому дробные суммы нельзя сравнивать через `==`.

Ловушка

Точное представление нельзя получить округлением при выводе.
Погрешность уже есть в самом значении.

Деньги в целых минимальных единицах

Деньги удобно хранить в копейках. Целые числа складываются и умножаются без погрешности, поэтому цена не теряет точности:

```

priceKopecks := int64(19990)
quantity := int64(3)

total := priceKopecks * quantity
fmt.Println(total) // 59970

```

Тип `int64` — знаковое 64-битное целое. Он даёт точные значения и вмещает суммы в копейках с большим запасом.

Большие суммы и криптовалюты

У копеек два знака после запятой, а у эфира восемнадцать. Для таких минимальных единиц диапазона `int64` может не хватить.

Уточнение

Встроенного 128-битного целого типа в Go нет: самый большой встроенный целый тип — 64-битный. Для очень больших целых чисел используют тип `big.Int` из пакета `math/big`.

Важно

Цена заказа хранится как `int64` в копейках.

Сравнение `float64`, `int` и `int64` для хранения цены

ТИП	ТОЧНОСТЬ	ДИАПАЗОН	ВЫВОД ДЛЯ ЦЕНЫ
<code>float64</code>	Погрешность: $0.1 + 0.2$ даёт не 0.3	Дробные, около 15–17 знаков	Нет: ошибка накапливается
<code>int</code>	Точный, только целые числа	Зависит от платформы (обычно 64 бита)	Можно, но лучше явный <code>int64</code>
<code>int64</code>	Точный, только целые числа	От -2^{63} до $2^{63}-1$	Да: цена в копейках

Для денег хранят копейки в `int64`, а не рубли во `float64`.

ГЛАВА 3

Целые и дробные типы, `byte`, `rune` и преобразования

11:30 – 18:20

Разбираются целые типы со знаком и без знака, их диапазоны, `byte` и `rune`, `float32` и `float64`. Компилятор не смешивает числовые типы автоматически, поэтому преобразования пишутся явно.

Базовые типы

- `bool` хранит логическое значение: `true` или `false`.
- `string` хранит строку в двойных кавычках.
- Целые со знаком: `int8`, `int16`, `int32`, `int64` и `int`.
- Беззнаковые целые: `uint8`, `uint16`, `uint32`, `uint64` и `uint`.
- `byte`, `rune`, `float32` и `float64`.

```
var _ int = 10
var _ uint64 = 10
var _ bool = true
var _ string = "Go"
var _ byte = 255
var _ rune = 'g'
```

Имя `_` — пустой идентификатор. Значение отбрасывается, а переменная нужна только для демонстрации типа.

Целые типы со знаком и без знака

Целый тип **со знаком** хранит и отрицательные, и положительные числа. `int8` принимает значения от -128 до 127.

Беззнаковый тип не хранит отрицательных чисел, зато его положительный диапазон вдвое шире. `uint8` хранит значения от 0 до 255.

Разрядность целых растёт от 8 до 64 бит. `int64` — самый большой из простых знаковых типов.

Int и uint без разрядности

Если указать просто `int` или `uint`, разрядность выберет компилятор при сборке, исходя из целевой архитектуры. Для 64-битной целевой архитектуры это 64 бита, то есть 8 байт. При этом `int` и `int64` остаются разными типами, даже когда их размеры совпадают.

Byte и rune

- **Byte** — псевдоним (alias) для `uint8`. Он ничего не меняет в работе, но делает код понятнее: сразу видно, что значение — байт.
- **Rune** — символ Unicode. Символ пишут в одинарных кавычках: `'g'`. Строку пишут в двойных: `"Go"`.

Вещественные типы

Дробные числа хранят `float32` и `float64`. Просто `float` в Go нет: разрядность нужно указывать явно.

Преобразования типов

Компилятор не смешивает разные числовые типы автоматически. Сложить `float64` и `int` нельзя, пока одно из значений не приведено к типу другого:

```
var a float64 = 10.2
var b int = 10

// sum := a + b — ошибка компиляции

sum := a + float64(b)
```

Приводить `float64` к `int` нужно осторожно: дробная часть отбрасывается. Поэтому здесь приводят `int` к `float64`, а не наоборот.

Потеря дробной части

`int(a)` компилируется без ошибки, но 10.2 превращается в 10. Приведение нужно писать осознанно, а не ради того, чтобы убрать ошибку компилятора.

```
var count int = 3
var price float64 = 19.9

total := float64(count) * price
fmt.Println(total)
```

Явные преобразования дают три преимущества:

1. Преобразование видно в коде.
2. Потеря точности не происходит незаметно.
3. Намерение автора можно проверить при чтении.

Диапазоны целых типов Go

ТИП (РАЗРЯДНОСТЬ)	ЗНАКОВОСТЬ	МИНИМУМ	МАКСИМУМ
int8 (8 бит)	Со знаком	-128	127
uint8 (8 бит)	Без знака	0	255
int16 (16 бит)	Со знаком	-32768	32767
uint16 (16 бит)	Без знака	0	65535
int64 (64 бита)	Со знаком	-9223372036854775808	9223372036854775807
uint64 (64 бита)	Без знака	0	18446744073709551615

Тип `int` и `uint` без разрядности получает размер, выбранный компилятором под платформу, обычно 64 бита. Тип `byte` — псевдоним `uint8`.

ГЛАВА 4

Ошибки выбора типа, строки и руны

18:20 — 26:30

Неверный выбор типа даёт ошибки на границах: переполнение, потерю дробной части при приведении и неверную длину строки. Строка в Go — последовательность байт, поэтому кириллица занимает больше байт, чем букв; руны позволяют работать с символами.

Неверный выбор типа не всегда заметен: программа компилируется и работает, но ошибается на граничных значениях. Чаще всего встречаются четыре ошибки.

Переполнение целого типа

Целый тип хранит значения только в своём диапазоне. Например, `int8` вмещает от -128 до 127 , поэтому `10 + 127` даёт -119 : результат «перескакивает» через границу. Ошибки при этом не возникает.

Баланс кошелька

Если баланс продавца хранится в слишком узком целом типе, зачисление денег может вызвать переполнение. Перед зачислением нужно проверить, помещается ли итоговая сумма в диапазон типа.

В Go переполняются и `int`, и `uint`. Поэтому тип выбирают под максимально возможное значение.

Важно

Тип выбирают под данные, а не по привычке. Защиту от переполнения, то есть проверку до операции, разберём в разделе про границы.

Сравнение дробных чисел через ==

Как показано в разделе о деньгах, `0.1 + 0.2` даёт `0.30000000000000004`, и сравнение с `0.3` возвращает `false`. Дробные числа на точное равенство не сравнивают: сравнивают разницу с небольшим допуском. Для денег поэтому и выбирают целый тип.

byte и rune

`byte` — это `uint8`, один байт. `rune` — символ Unicode. Их нельзя путать: символ кириллицы не помещается в `byte`, а `rune` для него подходит. Тип должен соответствовать тому, что хранится.

Потеря данных при преобразовании

Преобразование `float64` в `int` отбрасывает дробную часть без ошибки: если `a := 10.2`, то `int(a)` даёт `10`. Поэтому явное преобразование видно в коде, и его нужно проверять так же внимательно, как выбор типа.

Строка — последовательность байт

Функция `len` для строки возвращает число байт. Каждая буква слова «привет» в UTF-8 занимает 2 байта.

```
package main

import "fmt"

func main() {
    name := "привет"
    fmt.Println(name)
    fmt.Println(len(name))           // 12: байты
    fmt.Println(len([]rune(name))) // 6: символы
}
```

Вывод программы:

```
привет
12
6
```

`[]rune(name)` переводит строку в срез рун, и его длина равна числу символов.

`[]byte(name)` переводит строку в срез байт, и его длина равна числу байт.

Внимание

Индекс строки даёт байт, а не букву. `name[0]` вернёт первый байт буквы «п», а не саму букву. Работать с символами удобнее через срез рун.

Руны

Руна — символ Unicode. Из зала спросили, похожа ли она на `char` из C++.

Лектор предложил сравнить самостоятельно, так как с C++ почти не работал.

Строку можно явно преобразовать в срез рун или байт, а каждый из этих срезов — обратно в строку вызовом `T(x)`. Прямого преобразования между `[]rune` и `[]byte` нет. Компилятор не выполнит такие преобразования автоматически.

ГЛАВА 5

Нулевые значения, объявления и константы

26:30 — 43:00

Переменные без значения получают нулевое значение своего типа. Переменные объявляют через `var` или `:=`, константы — через `const`. Для перечислений используется `iota` — автоматическая нумерация констант.

Переменная, объявленная без значения, не остаётся пустой и не получает `null`. Она сразу содержит **нулевое значение** своего типа, и поведение программы становится предсказуемым ещё до явной инициализации.

Нулевое значение (zero value)

Значение, которое получает переменная, объявленная без присваивания. Оно зависит только от типа.

Нулевые значения базовых типов:

- `int`, `uint`, `float64` — `0`;
- `bool` — `false`;
- `string` — пустая строка `""`;
- `byte` и `rune` — `0` (у `rune` это символ с кодом 0).

```
var a int      // 0
var b uint64   // 0
var f float64  // 0
var c bool     // false
var d string   // ""
var e byte     // 0
```

В Go нет `null`: переменная имеет значение своего типа. У перечисленных базовых типов без явной инициализации это `0`, `false` или `""`, но допустимость такого значения для алгоритма всё равно нужно проверять.

Объявление через `var`

Оператор `var` объявляет переменную с указанным типом. Если значение не задано, используется нулевое.

Тип можно не указывать: компилятор выведет его из значения.

```
var n = 10      // int
var f = 10.0     // float64
```

Значение должно соответствовать типу. Целой переменной можно присвоить `2.0`, потому что это число без дробной части, но не `2.5`:

```
var ok int = 2.0 // компилируется
var bad int = 2.5 // ошибка компиляции
```

Короткое объявление :=

Оператор `:=` объявляет переменную и сразу присваивает ей значение. Тип выводится из правой части. Работает он только внутри функции: на уровне пакета доступен только `var`.

```
package main

import "fmt"

func main() {
    count := 3           // объявляем
    count = 4            // меняем уже объявленную
    count, total := 5, 10 // total новая, count только меняется
    fmt.Println(count, total) // 5 10
}
```

Слева от `:=` должна стоять хотя бы одна новая переменная. Иначе компилятор выдаст ошибку `no new variables on left side of :=`.

Уточнение

`:=` объявляет переменную и присваивает значение, а `=` только меняет уже объявленную. Запись `a = 29` без предварительного объявления даёт ошибку `undefined: a`.

Константы

Константа задаётся оператором `const`. Её значение известно уже при компиляции, и компилятор подставляет его во все места использования. Константы можно объявлять вне функций.

Бывают два вида:

- **нетипизированная** константа: `const distance = 100`. Её тип определяется контекстом;
- **типизированная** константа: `const limit int = 100`. Тип зафиксирован.

```

package main

import "fmt"

const distance = 100
const limit int = 100

func main() {
    var f float64 = 2.5
    fmt.Println(distance * f) // 250
    // fmt.Println(limit * f) // ошибка: несовпадение типов int и float64
}

```

Нетипизированная константа `distance` берёт тип `float64` из контекста, поэтому её можно умножить на дробное число. Типизированная `limit` уже имеет тип `int`, и умножить её на `float64` без явного преобразования нельзя.

Нетипизированные константы гибче

Одна нетипизированная константа подходит и для целых, и для дробных вычислений. Тип фиксируют только тогда, когда он действительно важен.

Группы констант и `iota`

Константы удобно объединять в блок `const (...)`. Внутри блока работает счётчик `iota`.

`iota`

Встроенный счётчик внутри блока `const`. Он начинается с 0 и увеличивается на 1 в каждой строке. Строка без выражения повторяет выражение предыдущей строки.

```

package main

import "fmt"

func main() {
    const (
        a = iota
        b
        c
        d
    )
    fmt.Println(a, b, c, d) // 0 1 2 3
}

```

Если нумерацию нужно начать со 100, достаточно записать `a = iota + 100`: тогда значения будут `100 101 102 103`.

С битовым сдвигом удобно задавать единицы размера. Пустой идентификатор `_` пропускает значение с `iota = 0`:

```
const (  
    _ = iota           // пропускаем 0  
    KB = 1 << (10 * iota) // 1024  
    MB           // 1048576  
    GB           // 1073741824  
)
```

Так же задают статусы заказа, чтобы не назначать номера вручную:

```
const (  
    StatusPaid = iota // 0  
    StatusComplete    // 1  
    StatusWaiting     // 2  
)
```

Новые статусы добавляйте только в конец

Если вставить новый статус между `StatusPaid` и `StatusComplete`, номера всех следующих за ним статусов сдвинутся. Записи в базе, где хранились старые номера, начнут указывать на другие статусы.

Из зала спросили, являются ли эти номера индексами. Это значения констант, назначенные автоматической нумерацией. Слушатель сравнил её с протягиванием ячеек в Excel, и лектор согласился с аналогией.

Где живёт переменная

Переменная внутри функции размещается либо в стеке, либо в куче (heap). Чтобы увидеть решение компилятора, соберите программу с флагом:

```
go build -gcflags=-m
```

В выводе будет видно, какие переменные ушли в кучу. Размер `int` зависит от платформы: на 64-битной системе он занимает 8 байт.

Три формы объявления

Форма	Где писать	Тип	Пример
<code>var x int</code>	езде, включая уровень пакета	указан явно или выведен	<code>var retries int (0)</code>
<code>x := value</code>	только внутри функции	выводится из значения	<code>name := "order-service"</code>
<code>const x = value</code>	езде	нетипизированный или явный	<code>const maxRetries = 3</code>

Нулевые значения типов в Go

ТИП	НУЛЕВОЕ ЗНАЧЕНИЕ
int, uint, float64	0
bool	false
string	пустая строка
byte, rune	0

Нулевое значение получает переменная, объявленная без присваивания, например через `var`.

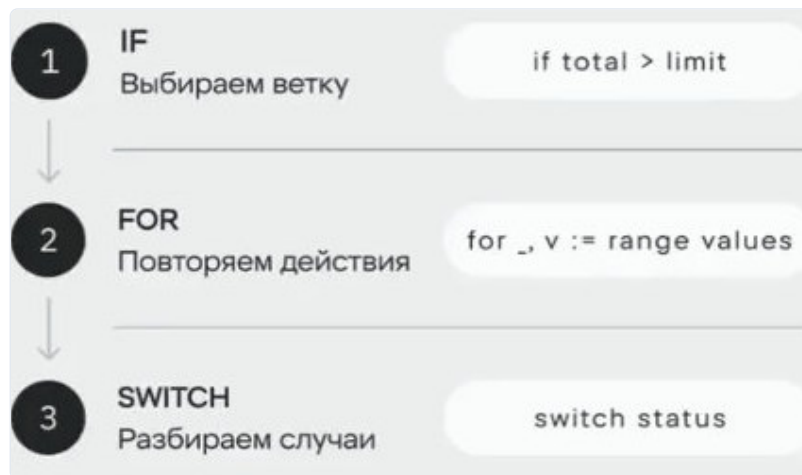
ГЛАВА 6

Циклы for

43:00 – 48:00

Цикл `for` в Go — единственный цикл, и он покрывает три формы: классический счётчик, условие вместо `while` и перебор коллекции через `range`. Форма цикла должна показывать намерение программиста.

На этом занятии рассматриваются три управляющие конструкции Go: `if`, `for` и `switch`. Цикл в языке один, `for`, и он покрывает все случаи повторения.



Три шага управления потоком: IF, FOR, SWITCH

Классическая форма: счётчик, условие и шаг

```
for i := 0; i < 3; i++ {  
    fmt.Println(i)  
}
```

В заголовке три части, разделённые точкой с запятой:

- `i := 0` — инициализация счётчика;
- `i < 3` — условие, которое проверяется перед каждым проходом;
- `i++` — шаг, который выполняется после тела цикла.

Переменная `i` принимает значения 0, 1, 2. Когда условие становится ложным, цикл завершается.

Форма только с условием

Если оставить одно условие, цикл заменяет `while` из других языков:

```
for ready == false {  
    // ... здесь меняется ready  
}
```

Если условие остаётся истинным и нет досрочного выхода, цикл не завершится. Цикл без условия, `for { }`, повторяется, пока выполнение не прервут, например через `break` или `return`.

range: перебор коллекции

`range` перебирает слайс и возвращает два значения: индекс и элемент.

```
arr := []int{1, 2, 3, 4}

for _, value := range arr {
    fmt.Println(value)
}
```

Этот цикл по очереди выводит 1, 2, 3, 4. Индекс здесь не нужен, и вместо него стоит `_`. Go запрещает объявлять локальные переменные, которые не используются, поэтому `_` отбрасывает значение, не создавая лишней переменной.

Если нужен только индекс, достаточно одной переменной: `for i := range arr` даёт 0, 1, 2, 3.

Какую форму выбрать

Задача	Форма	Пример
Выполнить действие заданное число раз	Счётчик	<code>for i := 0; i < 3; i++</code>
Повторять, пока выполнено условие	Только условие	<code>for ready == false</code>
Обойти все элементы коллекции	<code>range</code>	<code>for _, v := range items</code>

Форма цикла должна показывать намерение программиста. По заголовку сразу видно, идёт ли речь о счётчике, ожидании или переборе.

ГЛАВА 7

Map, if с инициализацией и ошибка как значение

48:00 – 59:10

Map — словарь, по ключу которого можно узнать, есть ли значение. Конструкция if с инициализацией позволяет проверить второе возвращаемое значение. Ошибка в Go — обычное значение, которое функция возвращает и которое нужно проверять явно.

Map: словарь по ключу

Map (словарь, ассоциативный массив) хранит пары «ключ — значение». В Python для этого служит `dict`. Тип записывают как `map[ключ]значение`:

```
mapp := map[int]int{0: 1, 1: 2}

fmt.Println(mapp[0]) // 1
```

Значение по ключу 0 равно 1, по ключу 1 равно 2.

Отсутствующий ключ

Если ключа нет, программа не падает. Map возвращает нулевое значение типа значения, для `map[int]int` это `0`.

Из одного значения нельзя понять, есть ключ или нет. Запрос `mapp[3]` вернёт `0`, и тот же `0` вернётся, если ключ `3` существует и хранит `0`.

Ловушка

Проверять, есть ли ключ, по значению `0` нельзя. Для этого используют вторую форму запроса.

Сотта-ок: значение и признак

Запрос с двумя переменными возвращает значение и признак наличия ключа:

```
val, exist := mapp[3]
```

- `val` — значение по ключу, а если ключа нет, нулевое значение типа;
- `exist` — `bool`: `true`, если ключ есть в map, и `false`, если нет.

Имя второй переменной выбирается произвольно. Чаще всего её называют `ok` или `exist`.

Запрос	val	exist
<code>mapp[1]</code>	<code>2</code>	<code>true</code>
<code>mapp[3]</code>	<code>0</code>	<code>false</code>

If с инициализацией

Конструкция `if` в Go может содержать короткое выражение перед условием, отделённое точкой с запятой:

```
if инициализация; условие {  
    // ...  
}
```

Сначала выполняется инициализация, затем проверяется условие. Переменные, объявленные в инициализации, видны в блоках `if` и `else`, но за их пределами не доступны.

```
mapp := map[int]int{0: 1, 1: 2}

if val, exist := mapp[3]; exist {
    fmt.Println(val)
} else {
    fmt.Println(val)
}
```

Ключа `3` в `map` нет, поэтому `exist` равно `false`, и выполняется `else`. Переменная `val` в этом блоке доступна и содержит `0`. Программа выведет `0`.

Совет

Сначала проверяйте признак наличия, и только потом используйте значение. Иначе в программу молча попадёт нулевое значение.

Условия: `&&`, `||`, `!`

Условия объединяют логическими операторами:

- `&&` — «и», оба условия истинны;
- `||` — «или», хотя бы одно истинно;
- `!` — отрицание, аналог `not` в Python.

```
mapp := map[int]int{0: 1, 1: 2, 3: 1}

if val, exist := mapp[3]; exist && val == 1 {
    fmt.Println("ключ 3 есть, значение 1")
}
```

Если значение по ключу не нужно, его заменяют пустым идентификатором `_`. Он отбрасывает значение:

```
if _, exist := mapp[5]; !exist {
    fmt.Println("ключа 5 нет")
}
```

Ошибка как значение

В Go нет `try/catch`. Ошибка — обычное значение типа `error`, которое функция возвращает наряду с результатом, обычно последним. Если всё прошло успешно, ошибка равна `nil`. Поэтому её проверяют явно:

```

package main

import (
    "errors"
    "fmt"
)

func checkAmount(amount int) error {
    if amount < 0 {
        return errors.New("сумма не может быть отрицательной")
    }
    return nil
}

func main() {
    if err := checkAmount(-5); err != nil {
        fmt.Println("ошибка:", err)
    }
}

```

Программа выведет `ошибка: сумма не может быть отрицательной`. Функция `checkAmount` при отрицательной сумме возвращает ошибку, а при корректной — `nil`, и тогда блок `if` пропускается.

Такой подход делает обработку видимой в коде: по строке вызова и проверке `err != nil` сразу понятно, где может возникнуть сбой. Если функция возвращает значение и ошибку, то при ошибке первое значение обычно нулевое, а проверять нужно именно `err`.

Ловушка

Возвращаемую ошибку нельзя просто проигнорировать. Компилятор не заставит её проверить, и сбой пройдёт незамеченным.

Подробно работу с ошибками разберём на следующем занятии.

ГЛАВА 8

Ветвление через switch

59:10 — 01:04:10

Switch сравнивает значение с перечисленными вариантами и выполняет только одну подходящую ветку. Break не нужен, проваливаться в следующий case по умолчанию не нужно.

Как работает switch

`switch` сравнивает значение с вариантами `case` и выполняет только ту ветку, которая подошла. Аналог в Python 3.10 — `match-case`, в большинстве других языков — `switch-case`.

```
status := "paid"

switch status {
case "new":
    fmt.Println("создан")
case "paid", "shipped":
    fmt.Println("в работе")
default:
    fmt.Println("неизвестен")
}
```

При `status := "paid"` выведется «в работе».

Несколько значений и default

- Несколько значений в одном `case` перечисляются через запятую. Ветка выполняется для любого из них.
- `default` выполняется, если ни один `case` не подошёл. Его можно не писать.

Break не нужен

Выполняется одна ветка

После выбранного `case` выполнение сразу выходит из `switch`. В C++ и JavaScript каждую ветку приходится завершать `break`, иначе выполнение провалится в следующую. В Go этого не нужно.

Если всё же нужно выполнить следующий `case`, пишут `fallthrough` явно, последним оператором ветки:

```
switch status {
case "paid":
    fmt.Println("оплачен")
    fallthrough
case "shipped":
    fmt.Println("в работе")
}
```

Для "paid" выведутся обе строки.

Сложные условия

Если после `switch` ничего не указано, каждый `case` содержит логическое условие. Проверка идёт сверху вниз, поэтому выполнится первая истинная ветка:

```
total := 25_000

switch {
case total < 10_000:
    fmt.Println("small")
case total < 50_000:
    fmt.Println("regular")
default:
    fmt.Println("large")
}
```

Здесь выведется `regular`. Вторую часть темы, применение `switch` в конкурентном коде, разберём в конце курса.

ГЛАВА 9

Пустой срез и защита от переполнения

01:04:10 — 01:08:50

Главная проблема алгоритмов — неверное предположение о данных. Пустой срез приводит к делению на ноль, а переполнение целого типа обходят проверкой границ до операции.

Пустой срез и деление на ноль

Главная проблема алгоритмов — неверное предположение о данных. Посмотрим на подсчёт среднего:

```

values := []int{}

sum := 0
for _, v := range values {
    sum += v
}

avg := sum / len(values) // panic

```

Цикл по пустому срезу не выполняется, `sum` остаётся нулём, а `len(values)` равен 0. Целочисленное деление на ноль вызывает **панику** (panic): программа аварийно останавливается. Компилятор эту ошибку не видит, потому что заранее не знает, какие данные придут.

Защита — проверить длину до деления и вернуть ошибку:

```

if len(values) == 0 {
    return 0, errors.New("пустой срез")
}
// ... расчёт среднего

```

Совет

Проверяйте входные данные в начале функции. Так граничный случай обрабатывается один раз и не мешает основному алгоритму.

Переполнение целого типа

Тип `int8` хранит значения от -128 до 127:

```

var a int8 = 10
var b int8 = 127

sum := a + b // 137 не помещается в int8

```

Go не выдаёт ошибку: значение молча «заворачивается» и становится -119. Такой баг проявляется позже, например в балансе продавца.

Границы типа берут из пакета `math`: `math.MaxInt8` (127), `math.MaxInt64` и другие константы. Числа вручную не записывают.

Уточнение

Проверка `a+b >= math.MaxInt8` не спасает: сумма уже переполнилась, и сравнивается испорченное значение. Проверять нужно до сложения, для положительных значений это выглядит так:

```
if b > math.MaxInt8-a {  
    // сложение переполнит int8  
}
```

Перехват паники

Встроенная функция `recover` перехватывает `panic` внутри отложенной функции (`defer`). Это похоже на `try-catch` в других языках, но работает дороже по ресурсам. Поэтому в Go ошибку обычно возвращают как значение, и вызывающий код обрабатывает её явно.

ГЛАВА 10

Ошибки как значения, путь алгоритма и задание

01:08:50 – 01:14:40

Лектор объясняет, почему в Go ошибки возвращают явно, а не через исключения. Затем показан путь от входных данных к результату и формулируется домашнее задание.

Ошибка в Go — обычное значение. Функция, которая может не выполниться, возвращает её последним результатом, а вызывающий код проверяет её явно. Исключений и конструкции `try/catch` в языке нет.

На вопрос из зала, как обходиться без `try/catch`, ответ такой: сразу после вызова функции проверяют ошибку и решают, что делать дальше.

```

package main

import "fmt"

func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, fmt.Errorf("деление на ноль")
    }
    return a / b, nil
}

func main() {
    result, err := divide(10, 0)
    if err != nil {
        fmt.Println("ошибка:", err)
        return
    }
    fmt.Println(result)
}

```

Код читается сверху вниз, а возможность ошибки видна в сигнатуре `(int, error)`.

Ошибку нельзя молча отбрасывать. Запись `result, _ := divide(10, 0)` компилируется, но программа тихо продолжит работу с нулём.

Сравнение с Python

В Python ошибки часто не указаны в документации библиотек. О них узнают при запуске, когда программа падает. В Go возможная ошибка записана в сигнатуре функции, а её обработка — в коде.

В Go есть и аварийное завершение (`panic`), его можно перехватить, но это несёт накладные расходы. Поэтому предсказуемые ситуации, например деление на ноль, проверяют заранее и возвращают как значение.

Путь алгоритма

Любой алгоритм на этом занятии проходит шесть шагов:

1. Выбрать тип данных.
2. Объявить данные.
3. Проверить условие.
4. Пройти циклом.
5. Разобрать `switch`.
6. Обработать границы.

Шаг 6 часто пропускают, хотя именно на границах программы чаще всего ломаются.

Задание для практики

Задачи лежат в репозитории. Сигнатуры функций и автотесты менять нельзя, цель — добиться PASS при запуске `go test ./...`.

1. **FizzBuzz через switch.** Числа от 1 до 30; делимость проверяйте через остаток от деления `%`.
2. **Min, max и сумма среза.** Пройдите срез циклом `range`.
3. **Среднее без потери точности.** Целочисленное деление отбрасывает дробную часть, поэтому сумму приводят к `float64` до деления: `float64(sum) / float64(len(values))`.
4. **Пустой вход.** Проверьте длину среза до обращения к элементам и до деления. Верните ошибку вместо паники.
5. **Автотесты до PASS.** Все проверки в файле тестов должны пройти.

Задача ProcessOrder (повторные попытки обработки заказа) — повышенной сложности. Её разберём на следующем занятии.

ГЛАВА 11

Разбор задачи ProcessOrder

01:14:40 — 01:23:00

Задача ProcessOrder просматривает результаты попыток слева направо и останавливается на первом успехе или при достижении лимита. Разобраны ошибка в первом варианте и её исправление.

Условие задачи

Функция `ProcessOrder(results []bool, maxAttempts int) (success bool, attempts int)` обрабатывает повторные попытки заказа. Каждый элемент `results` — результат одной попытки: `true` означает успех, `false` означает ошибку. Попытки просматриваются слева направо.

Функция останавливается в одном из трёх случаев:

- встретился первый `true`;
- число попыток достигло `maxAttempts`;
- список `results` закончился.

Возвращаемые значения:

- `success` равен `true` только в том случае, если встретился `true`;
- `attempts` — число просмотренных результатов, включая успешный.

Элементы после успешной попытки не учитываются. При `maxAttempts <= 0` или пустом списке функция возвращает `(false, 0)`.

results	maxAttempts	Результат	Почему
<code>[false, false, true, false]</code>	3	<code>(true, 3)</code>	успех на третьей попытке, четвёртая не просматривается
<code>[false, false, true]</code>	2	<code>(false, 2)</code>	лимит достигнут раньше успеха
<code>[false, false, false]</code>	3	<code>(false, 3)</code>	список закончился
<code>[true, false]</code>	0	<code>(false, 0)</code>	лимит нулевой, ни одна попытка не выполняется
<code>[]</code>	5	<code>(false, 0)</code>	список пуст

Тесты

Тест `TestProcessOrder` — таблица из шестнадцати случаев. В каждой строке заданы входные данные, лимит и ожидаемый результат. Функция считается верной, если проходят все строки. Запуск выполняется командой `go test ./...` в папке задачи.

Заготовка из условия содержит только комментарий `TODO` и возвращает `false, 0`.

Решение

Первая версия решения:

```
for _, value := range results {
    if attempts >= maxAttempts {
        break
    }

    attempts = attempts + 1

    if value {
        return true, attempts
    }
}
return false, 0
```

Цикл работает так:

1. `for _, value := range results` перебирает результаты. Индекс здесь не нужен, поэтому стоит `_`.

2. Перед каждой попыткой проверяется `attempts >= maxAttempts`. Если лимит достигнут, выполняется `break`. Проверка стоит до увеличения счётчика, поэтому `attempts` никогда не превышает лимит.
3. Счётчик увеличивается на единицу. Если результат `true`, функция сразу возвращает `true, attempts`. Счётчик уже учитывает успешную попытку, а следующие элементы не просматриваются.
4. Если цикл завершился без успеха, то есть по `break` или по концу списка, выполняется строка после цикла.

Отдельной проверки на пустой список и нулевой лимит не нужно. При пустом списке цикл не выполняется. При `maxAttempts <= 0` условие `attempts >= maxAttempts` выполняется уже на первой итерации.

Имена `success` и `attempts` объявлены в сигнатуре, поэтому внутри функции их не объявляют. `attempts` стартует с нуля, `success` — с `false`, и строка `attempts = attempts + 1` работает без `:=`.

Ошибка первой версии

Последняя строка `return false, 0` неверна. Когда цикл завершается без успеха, функция возвращает `0` попыток, хотя попытки были выполнены. Тесты показывают ошибку:

```
--- FAIL: TestProcessOrder/no_success
    ProcessOrder() = (false,0), want (false,3)
```

Падают четыре случая: `no_success`, `limit_shorter`, `results_exhausted`, `one_allowed_failure`. Во всех них успеха нет. Случаи с `true` проходят, поэтому ошибка незаметна, если проверять только успешные сценарии.

Ловушка

Литерал `0` вместо переменной — частая ошибка в функциях с именованными результатами. Код выглядит правильно, пока не запускается ветка без успеха.

Исправление — вернуть накопленный счётчик:

```
return false, attempts
```

После этой замены все тесты проходят.

Вопрос для пары: `break`, `continue` и `for condition`

- `break` прекращает цикл. Он нужен, когда дальнейшие шаги не имеют смысла, как при достижении лимита попыток.

- `continue` пропускает текущую итерацию и переходит к следующей. В этой задаче он не нужен: все просмотренные попытки учитываются, пропускать нечего.
- `for condition` — цикл только с условием, без счётчика и диапазона. Он подходит, когда число повторов заранее неизвестно, например, пока не достигнут лимит или не получен успех.

ГЛАВА 12

Остальные задачи, рефлексия и итоги

01:23:00 — 01:29:22

Разобраны условия оставшихся задач практики: *FizzBuzz*, статистика по суммам и категории заказа. Подведены итоги занятия и сообщена тема следующего.

Задача 1. FizzBuzz

Для чисел от 1 до 30 нужно вывести `FizzBuzz`, если число делится на 3 и на 5, `Fizz` — если только на 3, `Buzz` — если только на 5, и само число в остальных случаях.

Делимость проверяют через остаток от деления: число `n` делится на `d`, если `n % d == 0`.

Уточнение

Оператор `%` даёт остаток от деления, а не процент: `17 % 5` равно `2`.

Число, которое делится на 15, делится и на 3, и на 5. Поэтому проверку на 15 ставят первой:

Порядок проверок

Если сначала проверить `i%3 == 0`, то 15 выведется как `Fizz`, и ветка `FizzBuzz` никогда не сработает.

Решение через `switch` без выражения: каждый `case` здесь — отдельное логическое условие.

```

for i := 1; i <= 30; i++ {
    switch {
    case i%15 == 0:
        fmt.Println("FizzBuzz")
    case i%3 == 0:
        fmt.Println("Fizz")
    case i%5 == 0:
        fmt.Println("Buzz")
    default:
        fmt.Println(i)
    }
}

```

Задача 2. Статистика по суммам

Срез сумм в копейках непустой. Минимум и максимум находят одним проходом: первый элемент берут как начальное значение, каждый следующий сравнивают с ним. Сумма накапливается в той же петле.

Среднее возвращают как `float64`. Деление целых чисел отбрасывает дробную часть, поэтому сумму и длину приводят к `float64` до деления:

```
avg := float64(sum) / float64(len(values))
```

Пустой срез обрабатывают отдельно: проверка `len(values) == 0` стоит перед циклом, и функция сразу возвращает ошибку или сообщение.

Проверка

Для `[12000, 7500, 31000, 4500]`: минимум `4500`, максимум `31000`, сумма `55000`, среднее `13750`.

Задача 3. Категория заказа

Границы категорий заданы для целых копеек, поэтому промежутков между ними нет: `49_999` и `50_000` — соседние значения. Подчёркивание в `10_000` — разделитель разрядов, на значение оно не влияет.

Отрицательную сумму проверяют первой и не продолжают расчёт. Порядок `case` важен: когда срабатывает `sum < 50_000`, значение уже не меньше 10 000.

```

switch {
case sum < 0:
    fmt.Println("ошибка: отрицательная сумма")
case sum < 10_000:
    fmt.Println("small")
case sum < 50_000:
    fmt.Println("regular")
default:
    fmt.Println("large")
}

```

Задачи 4 и 5

Задачи 4 и 5 из списка простые. Пустой вход обрабатывается проверкой длины до вычислений, как в задаче 2. Для задачи 5 запускают `go test ./...`, смотрят, какие кейсы падают, и правят код. Сигнатуры функций и автотесты менять нельзя.

Задания из репозитория, с более сложными условиями, выполняются по желанию.

Рефлексия

Ответы на вопросы со слайда:

	var	:=
Где писать	в функции и вне её	только внутри функции
Тип	можно указать явно: <code>var n int</code>	выводится из значения
Без значения	<code>var n int</code> даёт 0	нельзя, значение обязательно

- **Тип для денег:** `int64` в копейках. Целая арифметика точна, а `float64` даёт погрешность: `0.1 + 0.2` не равно `0.3`.
- **Формы цикла:** классический счётчик, условие вместо `while` и `range` для перебора.
- **switch:** выбирает одну подходящую ветку, `break` не нужен.

На следующем занятии разберём функции, указатели, slice, map и struct, а также задание.

Шпаргалка

Деньги

- Хранить в копейках в `int64`. При сложении переменных `float64` со значениями `0.1` и `0.2` получается `0.30000000000000004`, не равное значению `float64(0.3)`.
- Пример: `19990 * 3 = 59970` коп.

Числовые типы

- `int8`: от -128 до 127; `uint8`: от 0 до 255.
- `byte = uint8`, `rune` — символ Unicode.
- `int`, `uint` — размер зависит от платформы; на 64-битной системе 64 бита.
- `float32`, `float64` — типы с плавающей точкой; около 15-17 значащих цифр относятся к `float64`.
- Неявного смешивания типов нет — нужно явное приведение: `float64(b); int(f)` отбрасывает дробную часть.
- Переполнение переменной происходит молча: после `var x int8 = 127; x++` значение `x` равно -128.
- Для положительных `a` и `b` типа `int8` проверка переполнения: `if b > math.MaxInt8 - a`, а не `if a+b >= ...`: сумма уже переполнилась.

Строки и руны

- `len("привет") = 12` байт (UTF-8, в этом слове — 2 байта на букву).
- `len([]rune("привет")) = 6` символов.

Объявления и нулевые значения

- `var x int` (ноль), `x := 1` (только внутри функции), `const`.
- Zero values: `0`, `false`, `""`.
- `const ( A = iota; B; C )` даёт `0`, `1`, `2`. Новые значения добавляйте в конец блока.

Циклы

- `for i := 0; i < n; i++ { }`
- `for cond { }` — аналог `while`
- `for i, v := range s { }`; `_` — пропуск значения

Map и if с инициализацией

- `v, ok := m[k]` — если ключа нет, `ok` равен `false`, а `v` получает нулевое значение типа значений `map`.
- `if err := f(); err != nil { }`

Switch

- Без `break`; нескольких значений в одном `case` — через запятую.
- `fallthrough` — явный переход в следующий `case`.

Ошибки и срезы

- Ошибка — последнее возвращаемое значение: `if err != nil`.
- Целочисленное деление на `len(values)` при пустом срезе даёт `panic`; до деления проверяйте `len(values) == 0`.

Задачи

- `ProcessOrder`: проверка `attempts >= maxAttempts` перед попыткой; `attempts++`; `if value { return true, attempts }`; в конце `return false, attempts`.
- Пример: `[false, false, true, false]`, `maxAttempts=3` → `(true, 3)`.
- Статистика `[12000, 7500, 31000, 4500]`: `min 4500, max 31000, sum 55000, avg 13750`.

- Категории: меньше 10 000 — small; 10 000-49 999 — regular; от 50 000 — large; отрицательная сумма — ошибка ввода.
- FizzBuzz: сначала проверять делимость на 15 (через switch).

Ключевые цитаты

«В банках или там, где проходят заказы, например в каком-нибудь маркетплейсе, проходят сотни тысяч заказов в день, и суммы скачут. Поэтому даже самая малая неточность приведёт к ошибке, так как она имеет накопительный эффект.»

08:30 Объясняет, почему для денег нельзя использовать дробные типы даже при маленькой погрешности.

«Точно так же, как мы `int` превратили во `float`, или можно было `float` в `int`, но тогда отрезется часть после запятой.»

25:46 Показывает, что при явном преобразовании к целому типу дробная часть теряется молча, и это надо учитывать.

«Из-за этого нужно тщательно выбирать тип и следить за тем, чтобы он не переполнился.»

20:53 Главный практический вывод о переполнении: Go не предупреждает о нём, поэтому тип и проверки выбирает программист.

«Ошибка — это значение, оно должно всегда быть возвращено.»

56:36 Ключевая идея обработки ошибок в Go, которая определяет структуру всех функций в домашнем задании.

«Программа упадёт. Поэтому здесь нам нужно добавить условие и проверить, равна ли длина нулю. Если она равна нулю, то мы эту операцию не выполняем.»

01:04:59 Пример типичной ошибки на границе данных, которую нужно проверять до вычисления.

Глоссарий

Минорные единицы (копейки)

Способ хранения денег в целых наименьших единицах валюты, например в копейках. Сумма остаётся точной, потому что вычисления идут с целыми числами.

int64

Знаковый 64-битный целочисленный тип. Для денег в копейках используется как основной вариант.

float64

64-битный тип с плавающей точкой. Не все десятичные дроби представимы в нём точно: например, сумма переменных float64 со значениями 0.1 и 0.2 не равна значению float64(0.3), поэтому для денег выбирают целые минимальные единицы.

int и uint

Целые типы, размер которых выбирает компилятор по целевой архитектуре программы. Для 64-битной целевой архитектуры это 64 бита.

int8 и uint8

Целые типы фиксированного размера в 8 бит. int8 хранит значения от -128 до 127, uint8 — от 0 до 255.

byte

Синоним типа uint8, введённый для удобства чтения кода. Компилятор работает с ним как с uint8.

rune

Тип для символа Unicode. Одинарные кавычки задают символьный литерал, например 'g'.

Кодировка UTF-8

Способ хранения символов, при котором один символ занимает от одного до нескольких байтов. Буквы слова «привет» занимают по два байта.

Преобразование типов

Явное приведение значения к другому типу, например float64(count). Go не смешивает разные числовые типы автоматически, поэтому приведение записывается в коде явно.

Переполнение (overflow)

Ситуация, когда результат выходит за границы типа. Go не выдаёт ошибку, значение молча переходит на другой конец диапазона.

Zero value

Значение по умолчанию, которое получает переменная, объявленная без инициализации: 0 для числовых типов, false для bool, пустая строка для string.

var и :=

var объявляет переменную с явным или выведенным типом и может использоваться на уровне пакета. Оператор := объявляет и инициализирует переменную внутри функции, тип выводится из значения.

const и iota

const задаёт константу, которая подставляется на этапе компиляции. iota внутри блока const даёт последовательные значения, начиная с 0.

for range

Форма цикла, которая перебирает элементы массива, среза, строки или map. Для массива и среза даёт индекс и значение, для строки — байтовый индекс и руну, для map — ключ и значение; неиспользуемое можно заменить на _.

Comma-ok (проверка наличия ключа)

Конструкция v, ok := m[k], где ok равен true, если ключ присутствует в map, и false, если его нет. При отсутствии ключа v получает нулевое значение.

if с инициализацией

Конструкция if x := f(); условие { }, в которой переменная объявляется перед проверкой и видна только внутри if и else.

switch без break

Ветвление, в котором выполняется только первый подходящий case и переход к следующему case не происходит. Чтобы всё же перейти к следующему case, используется ключевое слово fallthrough.

error как значение

Ошибка в Go — обычное значение, которое функция возвращает последним результатом. Вызывающий код проверяет его через if err != nil.

panic

Аварийное завершение выполнения, например при делении на ноль. В Go его не используют для обычных ошибок, их возвращают как значения.

len

Встроенная функция, которая возвращает длину строки в байтах или количество элементов среза, массива или map.

Именованные возвращаемые значения

Возвращаемые переменные, объявленные в сигнатуре функции, например (success bool, attempts int). Их не нужно объявлять внутри функции, и они получают нулевые значения.

Вопросы для самопроверки

1. Почему для хранения цены заказа не рекомендуется использовать float64?

ОТВЕТ. Тип float64 не все дробные числа хранит точно. Например, сумма переменных float64 со значениями 0.1 и 0.2 не равна значению float64(0.3), и такие ошибки могут накапливаться при большом количестве операций. Поэтому деньги хранят в целых минорных единицах, например в копейках, в int64.

2. Чему равна стоимость покупки 3 штук по цене 199 руб. 90 коп., если хранить сумму в копейках в int64? Ответ выразить в копейках и в рублях.

ОТВЕТ. Расчёт: $19990 \cdot 3 = 59970$ копеек, то есть 599 руб. 70 коп.

3. Какой диапазон значений у int8 и uint8?

ОТВЕТ. int8 хранит значения от -128 до 127, uint8 — от 0 до 255. Тип byte является синонимом uint8.

4. Почему Go не складывает float64 и int без преобразования?

ОТВЕТ. Go не смешивает числовые типы автоматически. Так видно, где может потеряться точность, и приведение записывается в коде явно, например float64(b). Приведение float64 к int отбрасывает дробную часть.

5. Почему конструкция type byte = uint8 полезна, хотя по сути это один и тот же тип?

ОТВЕТ. Имя byte показывает, что значение используется как байт данных, а не как число. Компилятор видит одно и то же, а читающему код понятнее намерение.

6. Что выведет len("привет") и len([]rune("привет"))?

ОТВЕТ. len("привет") вернёт 12, потому что строка хранится в UTF-8, и каждая буква этого слова занимает два байта. []rune("привет") даёт срез из 6 символов, и len вернёт 6.

7. Какие значения по умолчанию получают переменные, объявленные без присваивания: var a int, var c bool, var d string?

ОТВЕТ. a получит 0, c получит false, d получит пустую строку "". Это zero values.

8. **Какие значения получают A, B, C в блоке `const (A = iota; B; C)`? Что произойдёт, если вставить новую константу в середину блока?**

ОТВЕТ. $A = 0$, $B = 1$, $C = 2$. Вставка в середину сдвигает значения следующих констант, поэтому новые статусы лучше добавлять в конец блока.

9. **Назовите три формы цикла `for` в Go.**

ОТВЕТ. Классическая форма `for i := 0; i < n; i++ { }`, форма только с условием `for cond { }` и форма перебора `for i, v := range s { }`.

10. **Дан `map m := map[int]int{0: 1, 1: 2}`. Что вернут значения `v`, `ok := m[3]` и каким будет результат проверки `if ok`?**

ОТВЕТ. `v` получит нулевое значение 0, `ok` получит `false`. Блок `if ok` не выполнится, так как ключа 3 в `map` нет.

11. **Почему в `switch` не нужен `break`, а в C++ или JavaScript он нужен?**

ОТВЕТ. В Go после выполнения подходящего `case` выполнение сразу выходит из `switch` и не проваливается в следующий `case`. В C++ и JavaScript без `break` выполнение переходит к следующему `case`, поэтому `break` ставят явно.

12. **Функция `ProcessOrder` получает результаты `[false, false, true]` и `maxAttempts = 2`. Какой будет результат?**

ОТВЕТ. Результат `(false, 2)`. Первая попытка неудачная, вторая тоже, затем `attempts` достигает `maxAttempts`, и цикл прерывается до проверки третьего элемента.

13. **Почему перед вычислением среднего значения нужна проверка `len(values) == 0`?**

ОТВЕТ. Для пустого среза `len` равно 0, и деление на него вызывает `panic`, то есть аварийное завершение программы. Проверка позволяет вернуть ошибку как значение и избежать краха.

14. **Для набора сумм `[12000, 7500, 31000, 4500]` найдите минимум, максимум, сумму и среднее. Как классифицировать заказ на 49 999 копеек?**

ОТВЕТ. Минимум 4500, максимум 31000, сумма 55000, среднее $55000/4 = 13750$. Сумма 49 999 копеек попадает в категорию `regular`, так как диапазон `regular` — от 10 000 до 49 999 включительно.

15. **Почему в Go ошибка возвращается как значение, а не через исключения `try/catch`?**

ОТВЕТ. Ошибка видна в сигнатуре функции и обрабатывается явно сразу после вызова через `if err != nil`. Это делает поведение программы предсказуемым и упрощает чтение кода.

Что изучить дополнительно

Пакет `math/big` для чисел больше 64 бит

Лектор упомянул его в связи с криптовалютами, где нужна точность до 18 знаков после запятой, но подробно не раскрыл.

Срезы (`slice`), `map` и `struct`

Срезы и `map` только бегло показаны в цикле и проверке наличия ключа; лектор сказал, что подробно разберёт их на следующем занятии.

Указатели и функции

Тема следующего занятия, которая понадобится для работы с функциями в домашнем задании.

Флаги сборки и анализ размещения переменных в куче и стеке

Лектор показал, что флагом сборки можно узнать, куда попала переменная, но не объяснил механизм.

Интерфейсы и `switch` по типам

Лектор отложил вторую часть `switch`, связанную с интерфейсами, на следующие занятия.

Горутины и конкурентность в `switch`

Лектор сказал, что основное свойство `switch` для конкурентности будет рассмотрено в конце курса.

Упоминания

Python (инструмент) — Используется для сравнения: `match-case`, словари `dict` и обработка строк, которые сравниваются с поведением `Go`.

JavaScript (инструмент) — Сравнивается с `Go` в поведении `switch`: в JavaScript нужен `break`, чтобы не провалиться в следующий `case`.

C++ (инструмент) — Упоминается в сравнении поведения `switch`, типов `byte` и `rune` и переполнения беззнаковых целых.

math (пакет стандартной библиотеки) (инструмент) — Содержит константы максимальных значений типов, например `math.MaxInt8`, для проверки переполнения.

math/big (инструмент) — Упомянут как пакет для очень больших целых чисел, которые не помещаются в 64 бита, например для криптовалют.

go test (инструмент) — Команда запуска автотестов в домашнем задании, по которым проверяется правильность решения.

Excel (инструмент) — Приведён пример: автоматический индекс в константах `iota` напоминает протягивание номеров строк в Excel.

Репозиторий на GitHub с заданиями (ссылка) — Место, где находятся домашние задания с тестами и где студенты публикуют своё решение в отдельной ветке.

Расширение для просмотра Markdown в редакторе (инструмент) — Лектор предложил его для удобного чтения файлов README без разметки.

Карточки для повторения

#	ВОПРОС	ОТВЕТ
1	Почему для хранения денег не рекомендуется использовать float?	Дробные числа хранятся с ограниченной точностью, поэтому вычисления дают погрешность. При сотнях тысяч операций в день она накапливается и приводит к расхождению сумм.
2	В каких единицах предлагается хранить деньги в заказах?	В целых минимальных единицах, например в копейках.
3	Что показал пример со сложением 10 копеек и 20 копеек в float?	Вместо 30 копеек программа выводит число с погрешностью в последних знаках после запятой.
4	Какой диапазон значений у int8?	От -128 до 127.
5	Чем отличается uint от int?	uint не хранит отрицательные числа, поэтому его положительный диапазон вдвое шире, чем у int того же размера.
6	Что такое byte в Go?	Синоним uint8, добавленный для того, чтобы компилятору и читателю было понятнее, что используется байт.
7	Что такое rune в Go?	Тип для одного символа. Символ записывается в одинарных кавычках.
8	Почему в Go нет просто float, а есть float32 и float64?	Разрядность указывается явно, чтобы компилятор не нарушал точность вычислений.
9	Почему Go не складывает int и float64 без преобразования?	Компилятор не смешивает числовые типы автоматически, поэтому одно из значений нужно явно привести к типу другого.
10	Что происходит при приведении float64 к int?	Дробная часть отбрасывается, и значение теряет знаки после запятой.
11	Что такое переполнение целого типа?	Значение выходит за границы диапазона типа и становится неверным, например при сложении баланса с суммой больше максимума типа.
12	Почему длина строки 'Привет' в Go равна 12, а не 6?	Строка хранится в UTF-8, и каждый символ кириллицы занимает 2 байта. Функция len считает байты.

#	ВОПРОС	ОТВЕТ
13	Как проверить, достигнет ли баланс максимума типа, прежде чем добавлять сумму?	Проверкой до сложения: сравнить текущее значение с разностью между максимумом типа и добавляемой суммой.
14	Что такое нулевое значение (zero value) переменной в Go?	Значение по умолчанию, которое получает переменная, объявленная без явного присваивания: 0 для числовых типов, пустая строка для string.
15	Чем отличаются var и := при объявлении переменных?	var можно использовать и вне функции, а := работает только внутри функции.
16	Что делает := при объявлении переменной?	Объявляет переменную и присваивает ей значение, а тип выводится из присвоенного значения.
17	Что такое iota?	Автоматическая нумерация констант внутри блока const, которая начинается с 0 и увеличивается на 1 для каждой следующей константы.
18	Почему новые статусы, созданные через iota, лучше добавлять в конец списка?	Вставка в середину сдвигает номера всех последующих констант, и значения, которые уже использовались, меняются.
19	Сколько циклов есть в Go?	Один: for. Он покрывает все формы цикла.
20	Что означает for без условия?	Бесконечный цикл, который завершается только через break или return.
21	Зачем вместо индекса в range ставят нижнее подчёркивание?	Так отбрасывают ненужное значение и не создают неиспользуемую переменную.
22	Что такое map в Go?	Словарь: коллекция пар ключ-значение, аналог dict в Python.
23	Зачем нужна форма обращения к map с двумя значениями?	Вторым значением возвращается булево ok, которое показывает, есть ли ключ в map.
24	Почему без проверки нельзя понять, есть ли ключ в map?	При отсутствии ключа возвращается zero value, например 0, и его не отличить от реально сохранённого нулевого значения.
25	Что такое ошибка в Go?	Обычное значение, которое функция возвращает и которое нужно проверять явно, например через if err != nil.

#	ВОПРОС	ОТВЕТ
26	Что делает switch?	Сравнивает значение с вариантами и выполняет только подходящую ветку. Если ни одна не подошла, выполняется default.
27	Нужен ли break в case в Go?	Нет. После выполнения ветки управление не переходит в следующий case.
28	Как в Go явно перейти в следующий case?	С помощью ключевого слова fallthrough.
29	Почему пустой срез опасен при вычислении среднего?	Деление на длину 0 приводит к панике, и программа аварийно завершается.
30	Как защититься от деления на ноль при пустом срезе?	Проверить, что длина больше 0, до операции, а при нулевой длине вернуть ошибку как значение.
31	Что такое паника (panic) в Go?	Аварийное завершение программы при критической ошибке во время выполнения, например при делении на ноль.
32	Почему в Go ошибку возвращают явно, а не через исключения?	По сигнатуре функции и по коду видно, какие ошибки возможны, и их нужно обработать на каждом уровне.
33	Почему panic и recover не используют для обычных ошибок?	Перехват паники имеет накладные расходы, поэтому ожидаемые ошибки лучше возвращать как значение.
34	Что такое именованные возвращаемые значения?	Переменные, объявленные в сигнатуре функции, которые можно использовать внутри без отдельного объявления.
35	Что возвращает ProcessOrder, если список результатов пуст?	false и 0 попыток, так как цикл не выполняется ни одного раза.
36	Как ProcessOrder обходит результаты попыток?	Просматривает результаты слева направо и останавливается на первом успешном результате или при достижении лимита попыток.
37	Какая операция определяет, делится ли число на 3 или на 5?	Остаток от деления, который в Go вычисляется оператором %.
38	Что выводит FizzBuzz для числа, которое делится и на 3, и на 5?	Строку FizzBuzz.