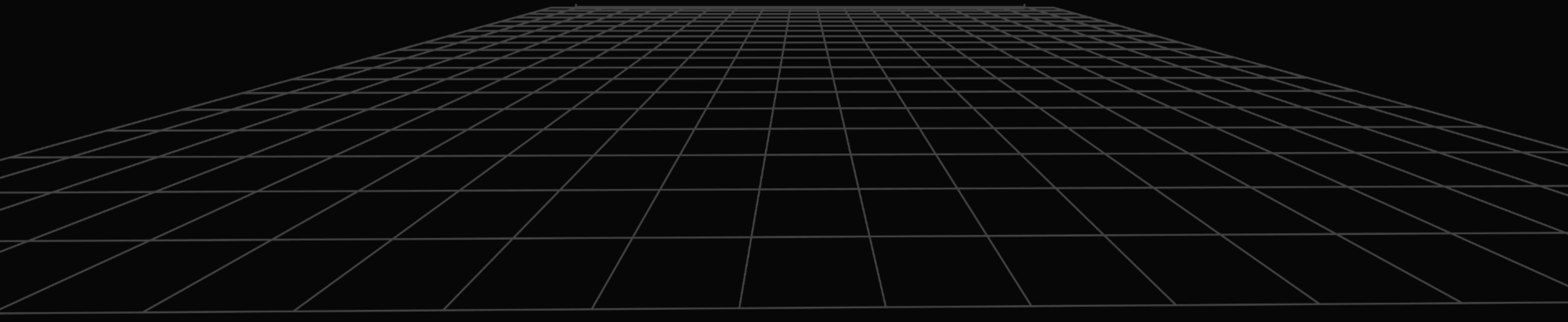
ИНТЕРФЕЙСЫ

Balun.Courses

ПРОВЕРЬ ЗАПИСЬ

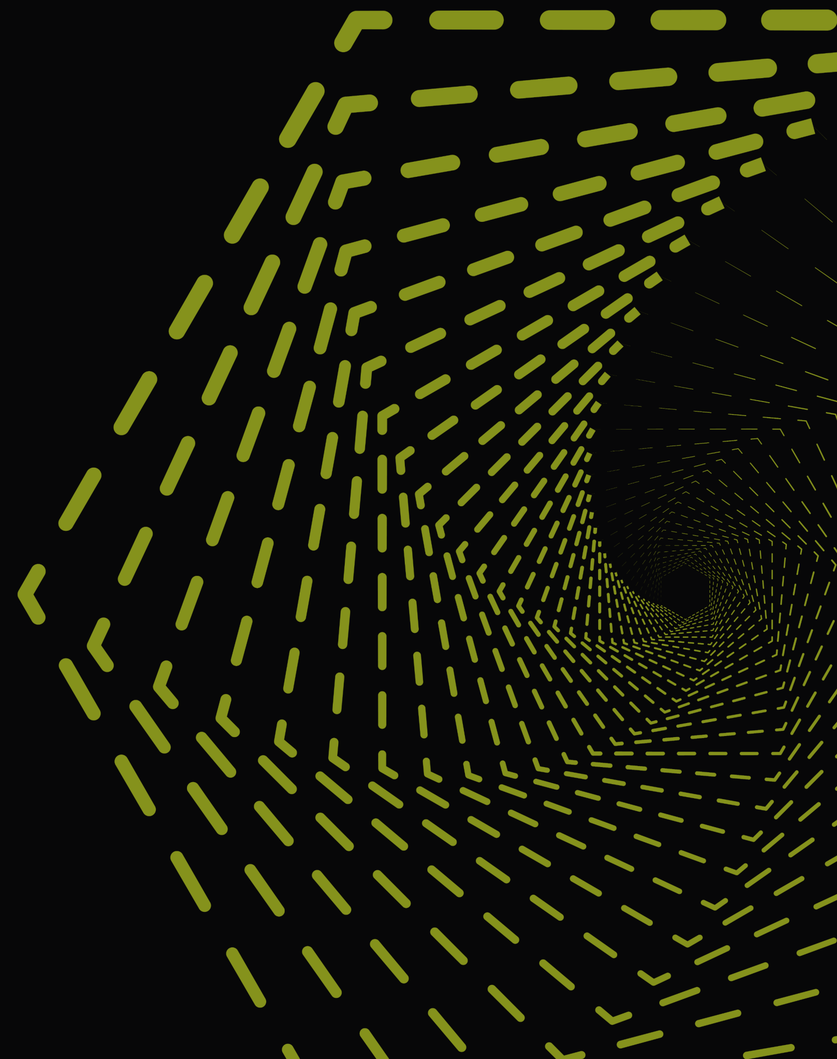


ПРАВИЛА ЗАНЯТИЯ

1. вопросы в чате можно задавать **в любое время**

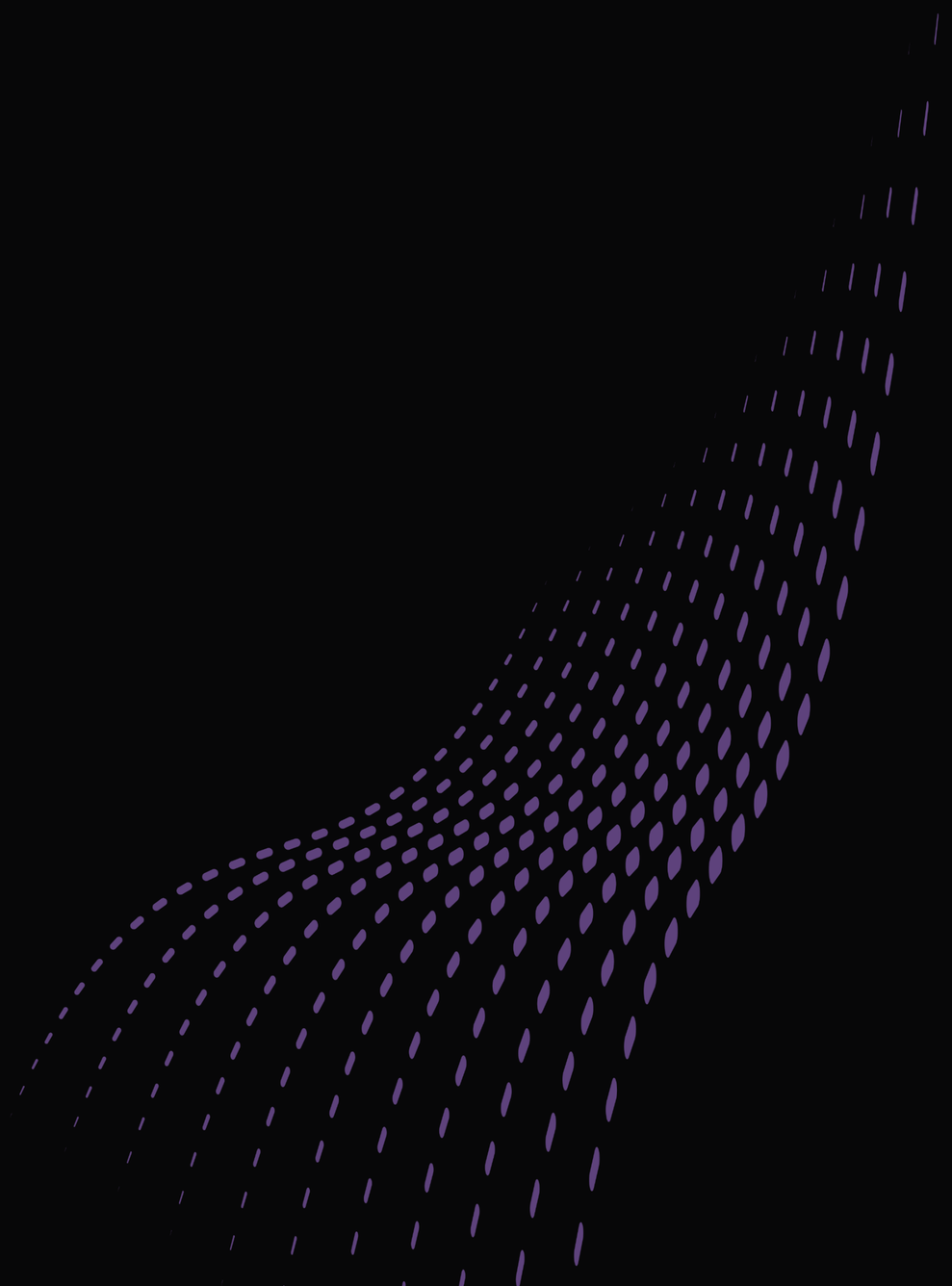
2. вопросы голосом задаем **по поднятой руке в Zoom**

3. ответы на вопросы будут **в запланированных местах**



ПЛАН ЛЕКЦИИ

1. Интерфейсы
2. Внутреннее устройство
3. Тонкости интерфейсов
4. Расположение интерфейсов

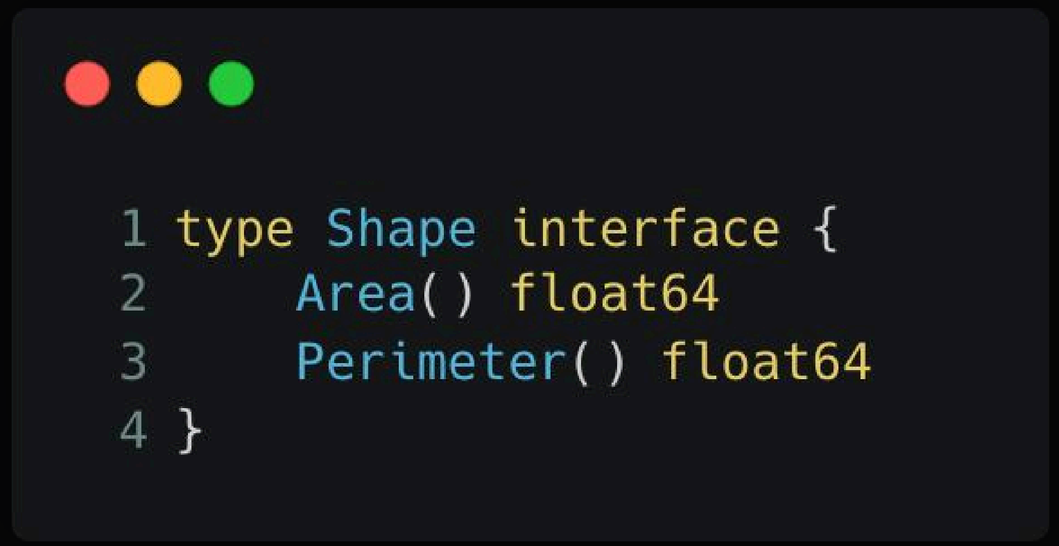


ИНТЕРФЕЙСЫ

ИНТЕРФЕЙС

набор сигнатур методов, которые нужно реализовать, чтобы удовлетворить контракту:

- одному интерфейсу могут соответствовать много типов
- тип может реализовывать несколько интерфейсов



```go
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

NIL - НУЛЕВОЕ ЗНАЧЕНИЕ ДЛЯ ИНТЕРФЕЙСНОГО ТИПА

До Go 1.18 все типы интерфейсов можно было использовать как типы значений, но начиная с Go 1.18 некоторые типы интерфейсов можно использовать **только как ограничения типов (constraints)**

EXAMPLE

Different interfaces

ЕЩЕ ГОВОРЯТ, ЧТО **ИНТЕРФЕЙСЫ ОПРЕДЕЛЯЮТ НЕКОТОРОЕ ПОВЕДЕНИЕ**

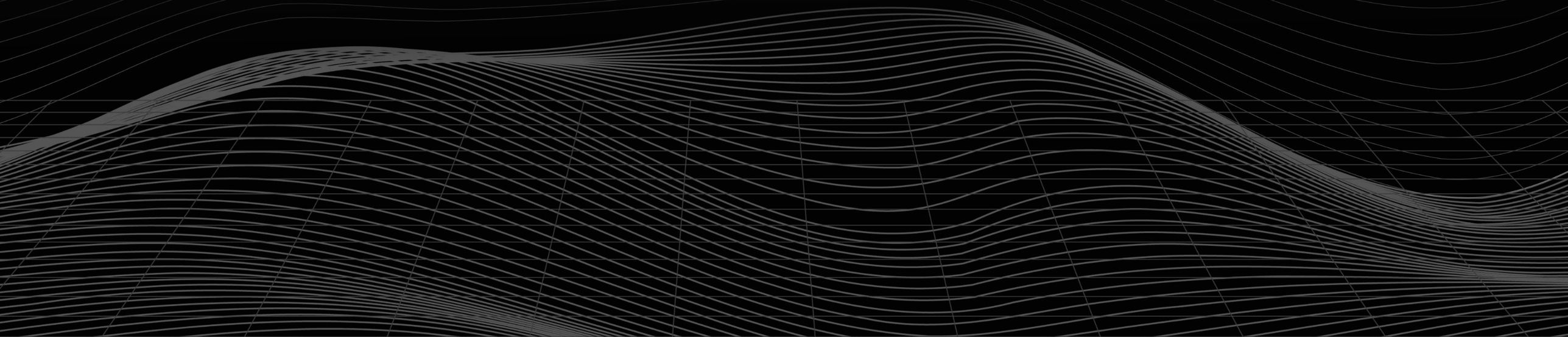
Terminal: deep_go

ПОЛИМОРФИЗМ

С греческого языка слово «полиморфизм» означает **«многообразие», которое позволяет разным сущностям выполнять одни и те же действия.** При этом неважно, как эти сущности устроены внутри и чем они различаются

EXAMPLE

Polymorphism

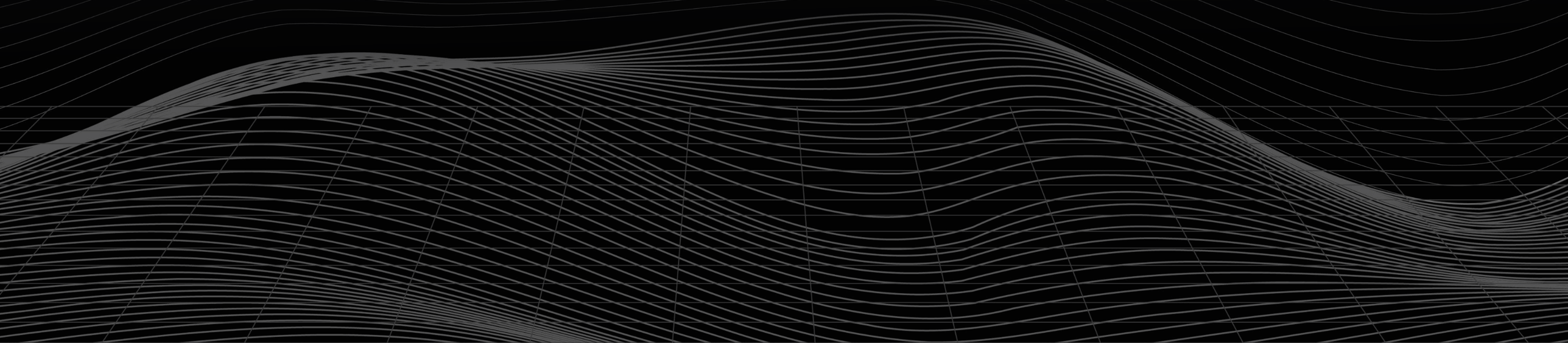


EXAMPLE

Duck typing

EXAMPLE

Duck typing problem



EXAMPLE

Static and dynamic type

ИНТЕРФЕЙСЫ

- **Минималистичность** — не должно быть огромного набора методов
- **Независимость от реализации** — не должен ничего знать о тех типах, которые его реализуют

«НЕ ПРОГРАММИРУЙТЕ ИНТЕРФЕЙСЫ, ОТКРЫВАЙТЕ ИХ»

© Роб Пайк

В Java или C++ вы, скорее всего, начинаете с объявления абстрактных классов и интерфейсов, и далее переходите к конкретным реализациям.

В Go же наоборот – вы пишете сначала конкретный тип, определяете данные и методы, и только в том случае, если действительно появляется необходимость абстрагировать поведение – создаете отдельный интерфейс.

МЫ НЕ ДОЛЖНЫ НАЧИНАТЬ СОЗДАВАТЬ АБСТРАКЦИИ В КОДЕ

если для этого нет веской причины. Нужно не конструировать интерфейсы, а ждать возникновения конкретной потребности в них (создавайте интерфейс только тогда, когда он действительно нужен).

Возвращайте значения из функции, которые являются конкретными типами, а не интерфейсами (потребители функции смогут выбирать способ использования возвращаемого типа, как интерфейса или как конкретного типа)

```
1 func Pipe() (*PipeReader, *PipeWriter)
2 func Copy(dst Writer, src Reader) (written int64, err error)
```

Это не значит, что вы всегда должны делать именно так… Можно возвращать интерфейс из функции, чтобы гарантировать, что **возвращаемое значение действительно удовлетворяет интерфейсу**

```
1 func TeeReader(r Reader, w Writer) Reader
```

Terminal: deep_go

ПУСТОЙ ИНТЕРФЕЙС

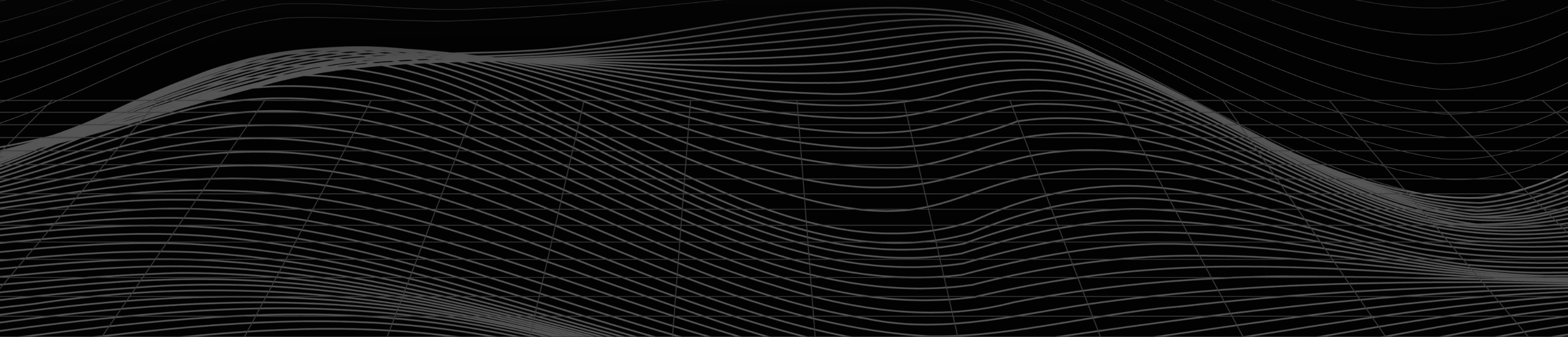
Интерфейс, которому удовлетворяет
абсолютно любой тип

EXAMPLE

Empty interface

EXAMPLE

Empty interface conversion



EXAMPLE

Empty interface use case

Используя пустые интерфейсы, мы теряем преимущества языка Go как со статической типизацией, поэтому **следует стараться избегать пустых интерфейсов** и делать сигнатуры более явными.

ИСКЛЮЧЕНИЯ

```go
func Marshal(v any) ([]byte, error) {
    ...
}
```

```go
func (c *Conn) QueryContext(
    ctx context.Context,
    query string,
    args ...any,
) ([]byte, error) {
    ...
}
```

EXAMPLE

Numbers comparison

Terminal: question + ⚙ –

КАК УЗНАТЬ, ЧТО СКРЫВАЕТСЯ ЗА ПУСТЫМ ИНТЕРФЕЙСОМ?

EXAMPLE

Type assertion 1

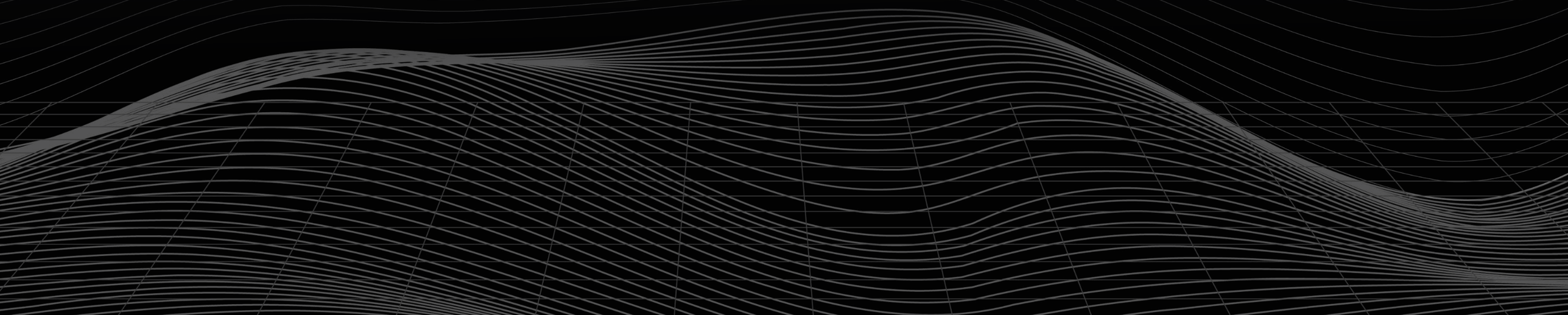
TYPE ASSERTION

x.(T) проверяет, что **x != nil** и:

- если T не интерфейс, то проверяет, что динамический тип x это T
- если T интерфейс, то проверяет, что динамический тип x его реализует

EXAMPLE

Type assertion 2



EXAMPLE

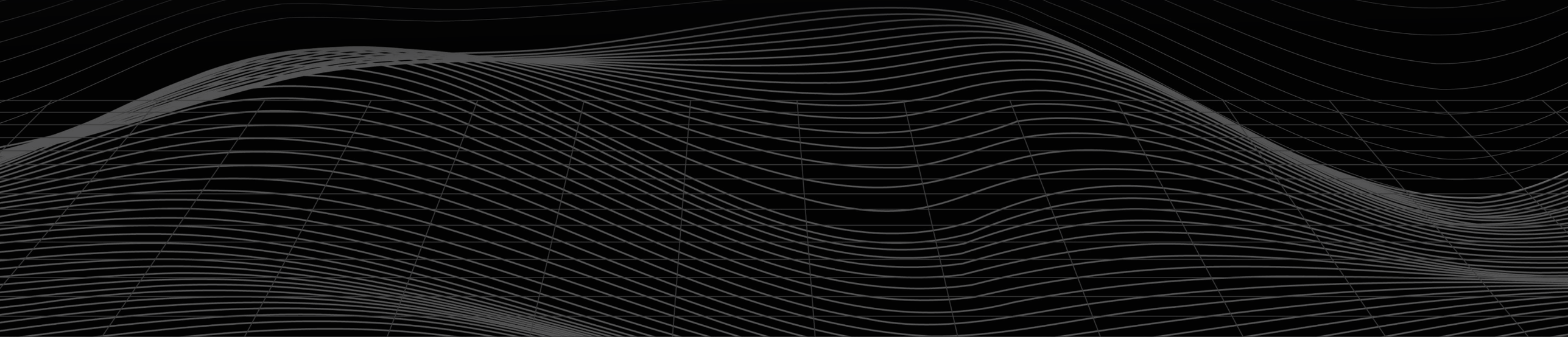
Type assertion 3

EXAMPLE

Type switch 1

EXAMPLE

Type switch 2



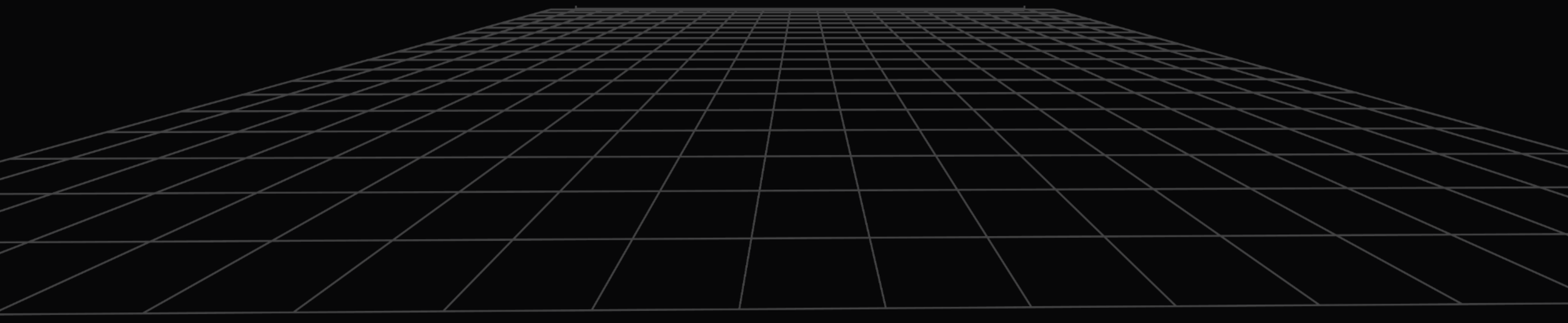
Как раньше до появления дженериков приходилось реализовывать обобщенные контейнеры или структуры данных?

EXAMPLE

Generic tree

FAQ

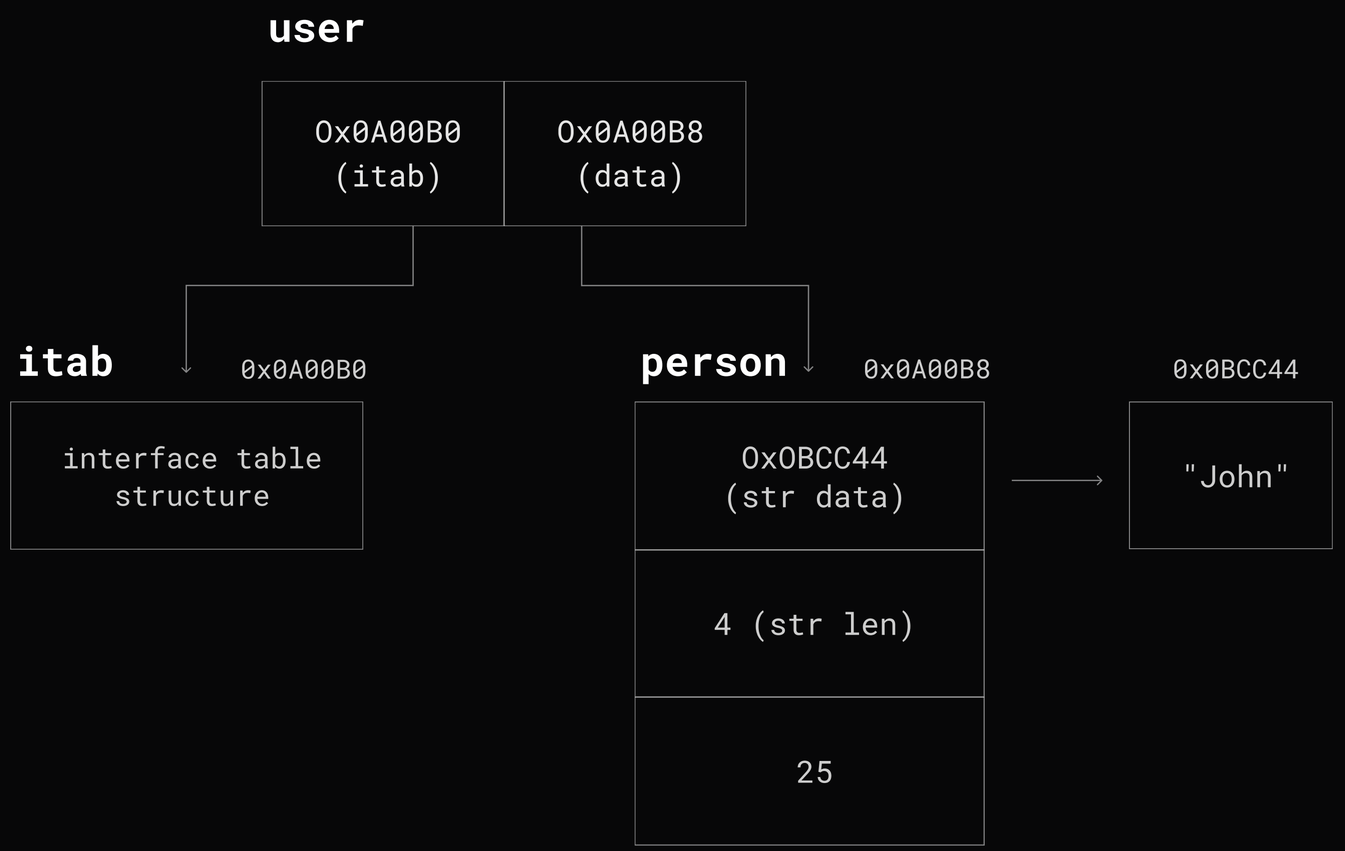
Интерфейсы



ВНУТРЕННЕЕ УСТРОЙСТВО

interface


```go
type iface struct {
    tab  *itab
    data unsafe.Pointer
}

type itab struct {
    inter *interfacetype
    _type *_type
    hash  uint32
    _     [4]byte
    fun   [1]uintptr
}
```




```go
type User interface {
    Name() string
    Age() int
}

func main() {
    person := Person{ "John", 25 }
    user := User(p)
}
```

ITABLE

Эта таблица будет уникальна для каждой пары интерфейс-статический тип,
то есть просчитывать её на этапе компиляции (early binding) будет нерационально и неэффективно

i Вместо этого, компилятор генерирует метаданные для каждого статического типа, в которых хранится список методов типа. Аналогично генерируются метаданные со списком методов для каждого интерфейса. Во время исполнения программы, Go может вычислить itable на лету (late binding) для каждой конкретной пары (itable кешируется, поэтому просчёт происходит только один раз)

table

Area(receiver)
Perimeter(receiver)

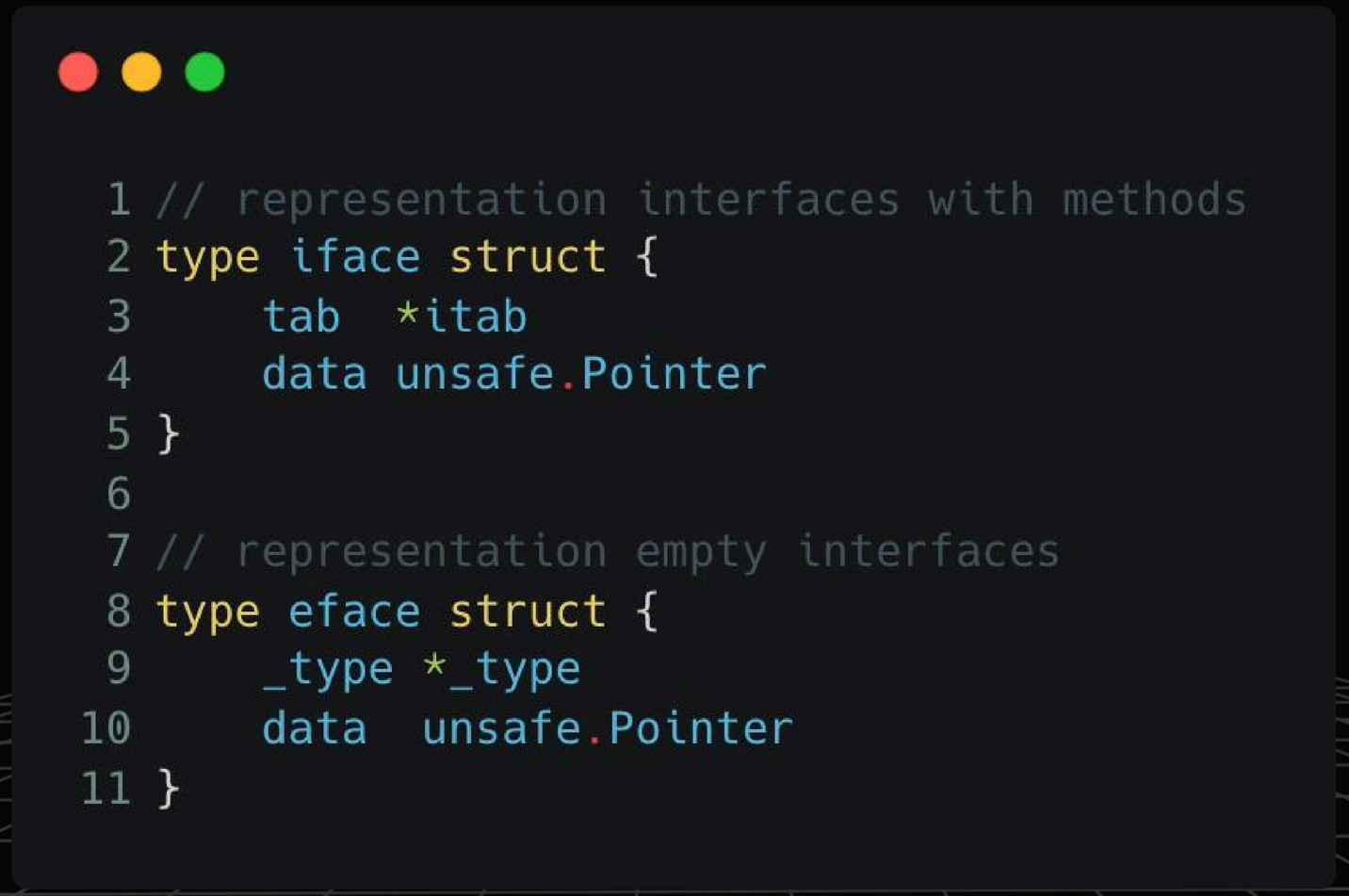
```go
type Figure interface {
    Area() float64
    Perimeter() float64
}

type Square struct{}

func (s *Square) Area() float64 {
    return 0.0
}

func (s *Square) Perimeter() float64 {
    return 0.0
}

func main() {
    square := &Square{}
    var figure Figure = square
    figure.Area()
    figure.Perimeter()
    _ = figure
}
```

EXAMPLE

Interface with receiver

EXAMPLE

Interface with receiver reason

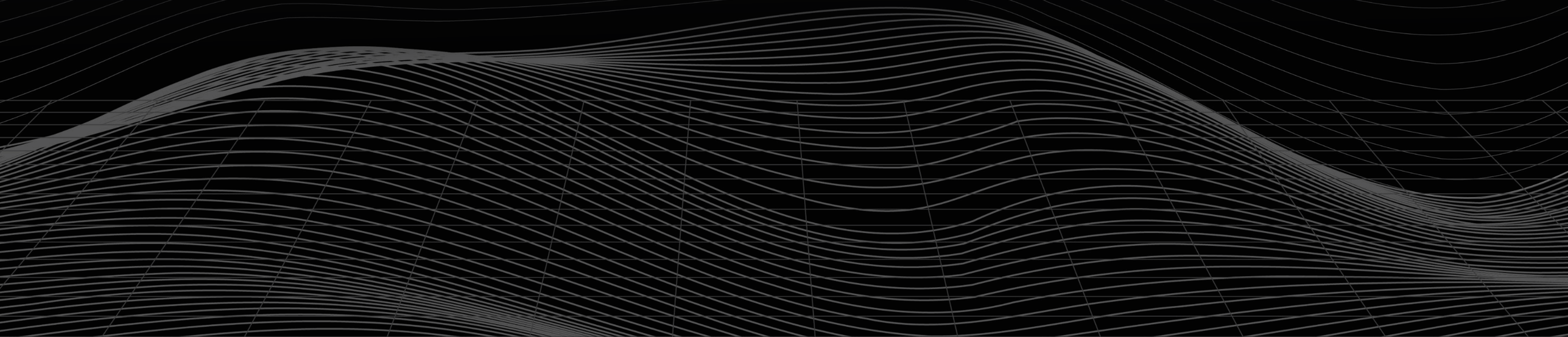


```go
// representation interfaces with methods
type iface struct {
    tab  *itab
    data unsafe.Pointer
}

// representation empty interfaces
type eface struct {
    _type *_type
    data  unsafe.Pointer
}
```

Поскольку у пустого интерфейса нет никаких методов, то и **itab для него просчитывать и хранить не нужно** — достаточно только метаинформации о динамическом типе.

EXAMPLE

Interface internals



Terminal: deep_go

СТАТИЧЕСКАЯ ДИСПЕТЧЕРИЗАЦИЯ

Когда тип экземпляра для вызываемого метода известен, мы имеем ясное представление о том какую функцию вызывать

Terminal: deep_go

ДИНАМИЧЕСКАЯ ДИСПЕТЧЕРИЗАЦИЯ

Когда тип экземпляра неизвестен, мы должны
найти способ вызова на нем правильного метода

EXAMPLE

Interface implementation

EXAMPLE

Interface performance

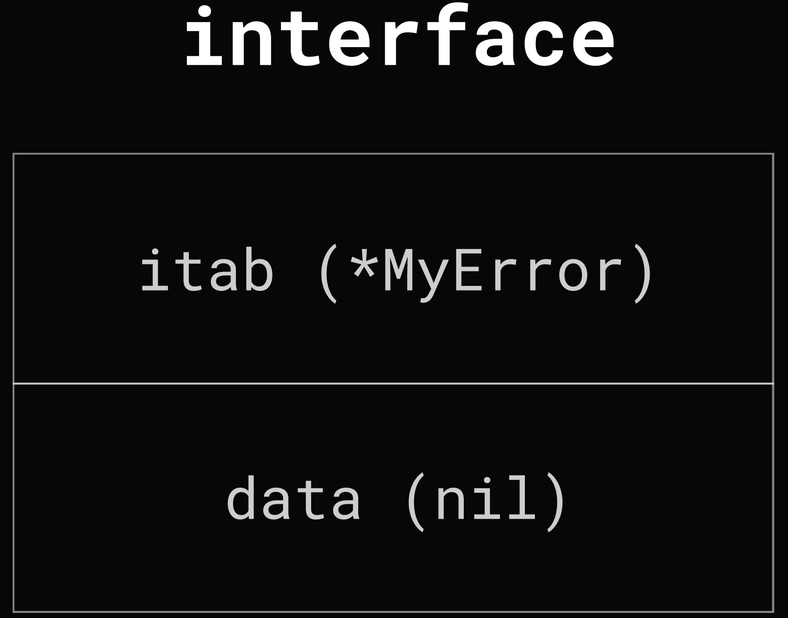
Terminal: question

ВСЕГДА ЛИ ИНТЕРФЕЙС РАВЕН NIL?

EXAMPLE

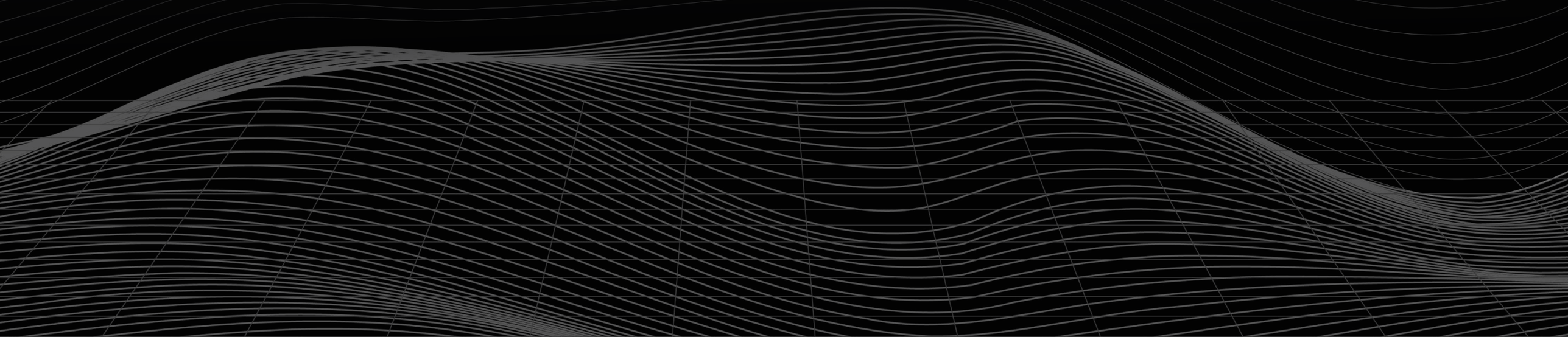
Interface not nil 1

Указатель на данные равен nil, но указатель на itable не равен nil, поэтому и интерфейс не равен nil



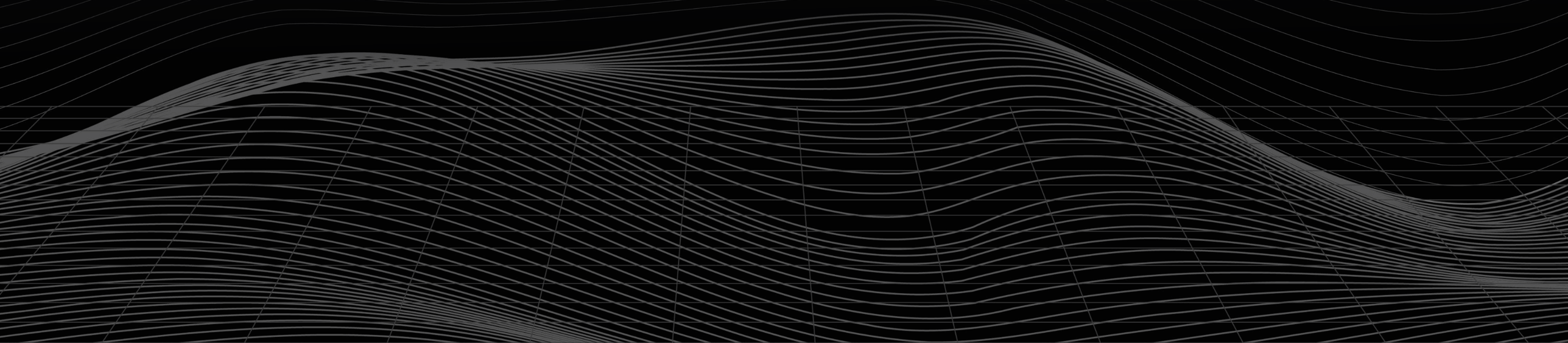
EXAMPLE

Interface not nil 2



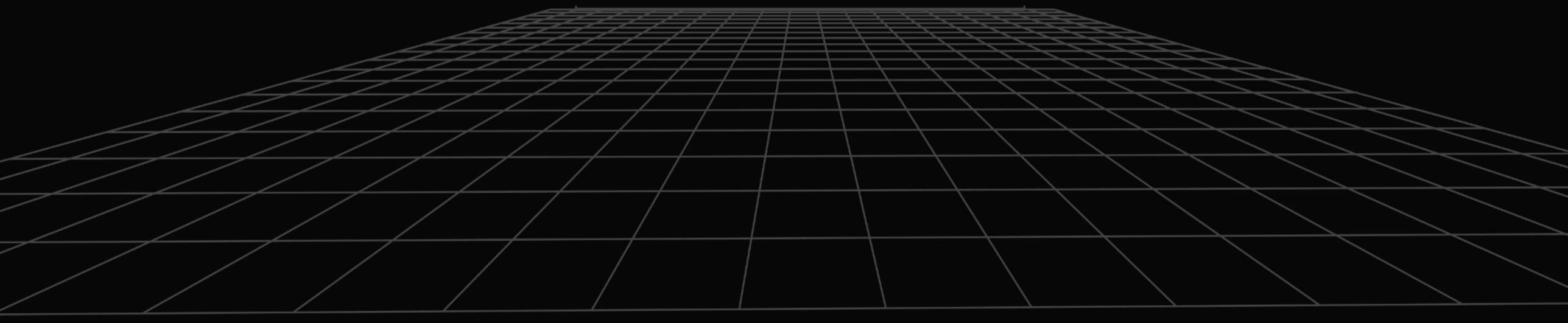
EXAMPLE

Interface not nil 3



FAQ

Внутреннее устройство



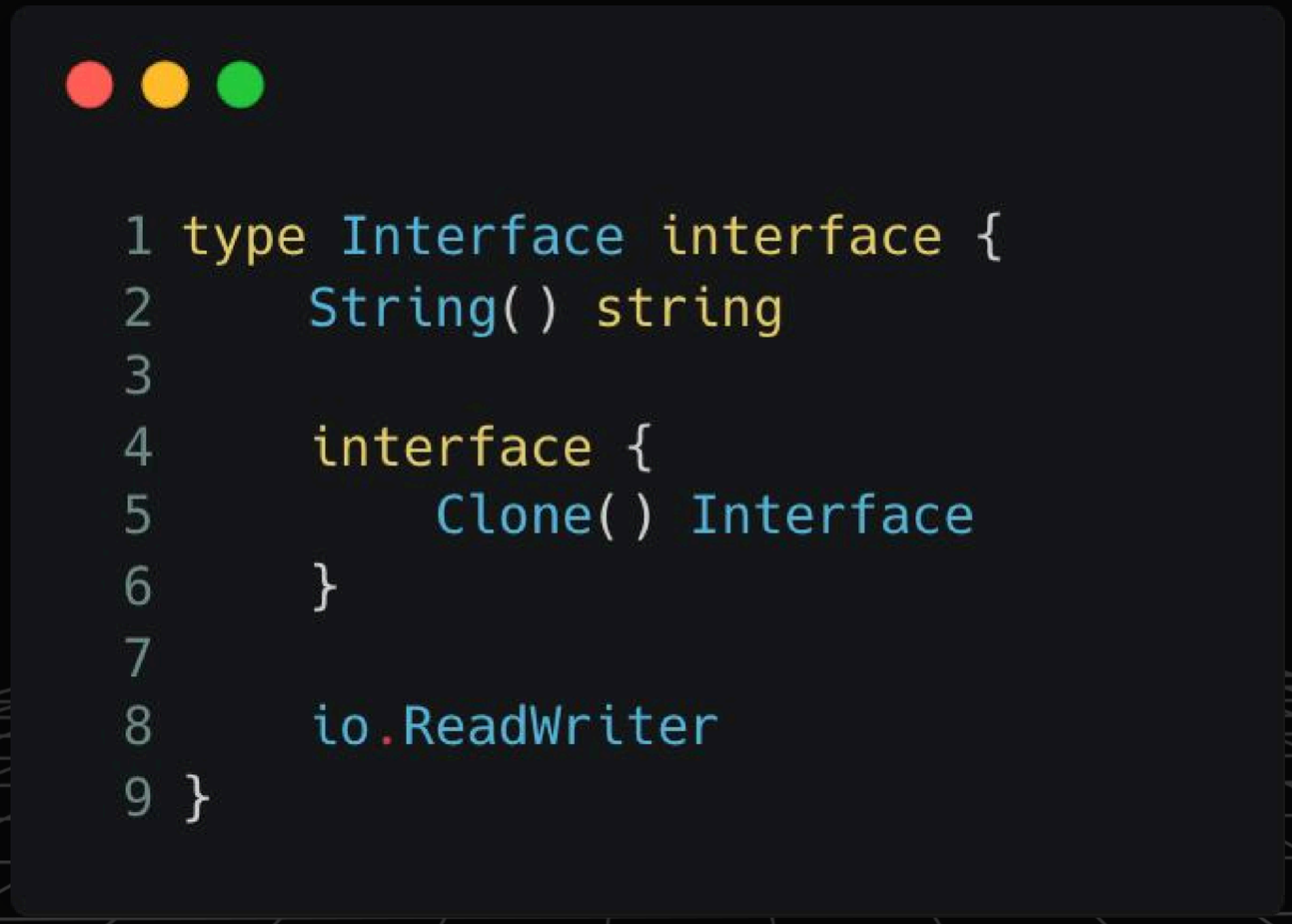
ПЕРЕРЫВ 5 МИНУТ

ТОНКОСТИ ИНТЕРФЕЙСОВ

EXAMPLE

Interface copy

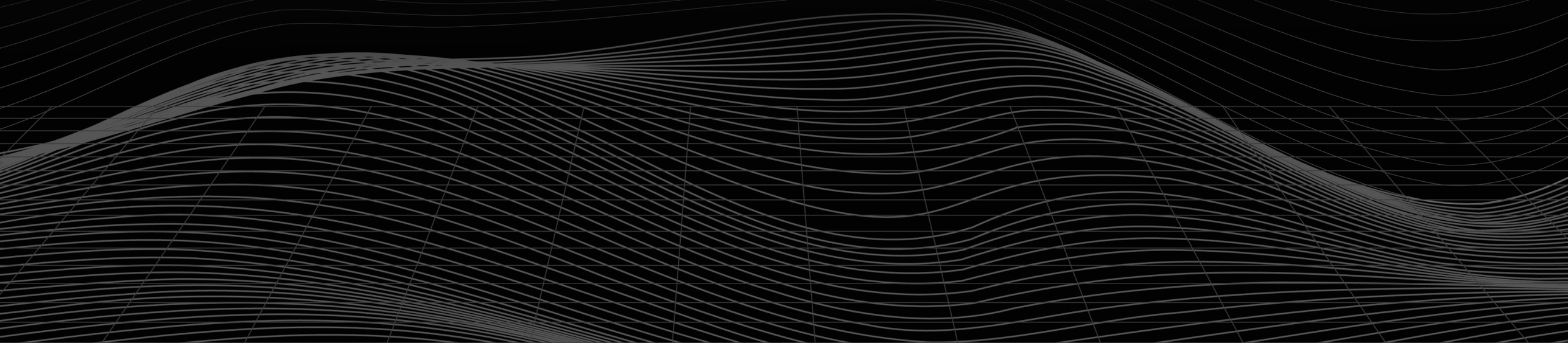
Интерфейсы, как и структуры, можно встраивать



```
type Interface interface {
    String() string

    interface {
        Clone() Interface
    }

    io.ReadWriter
}
```

EXAMPLE

Interface composition

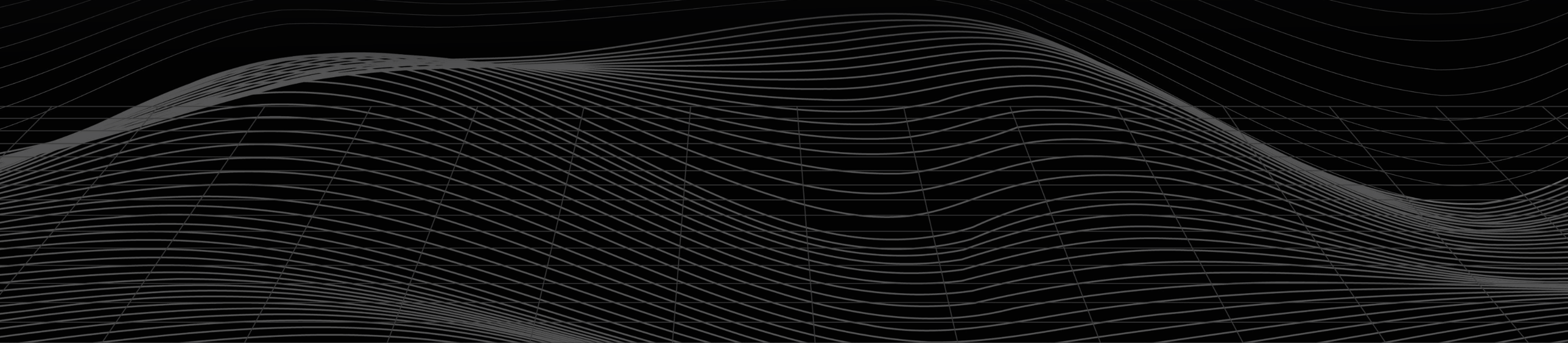


EXAMPLE

Interface composition with struct

EXAMPLE

Interface assignment

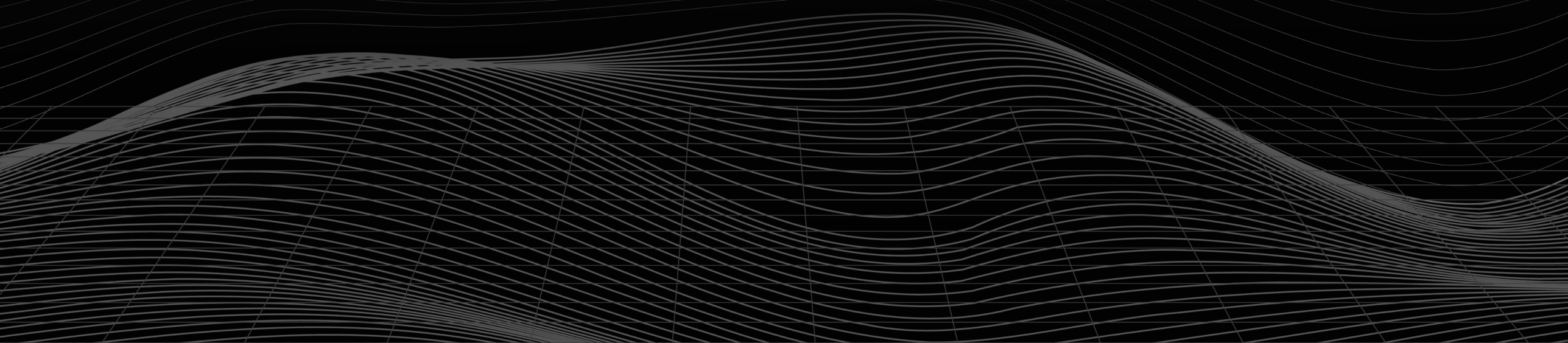


EXAMPLE

Interface cast

EXAMPLE

Slice values to slice interfaces



EXAMPLE

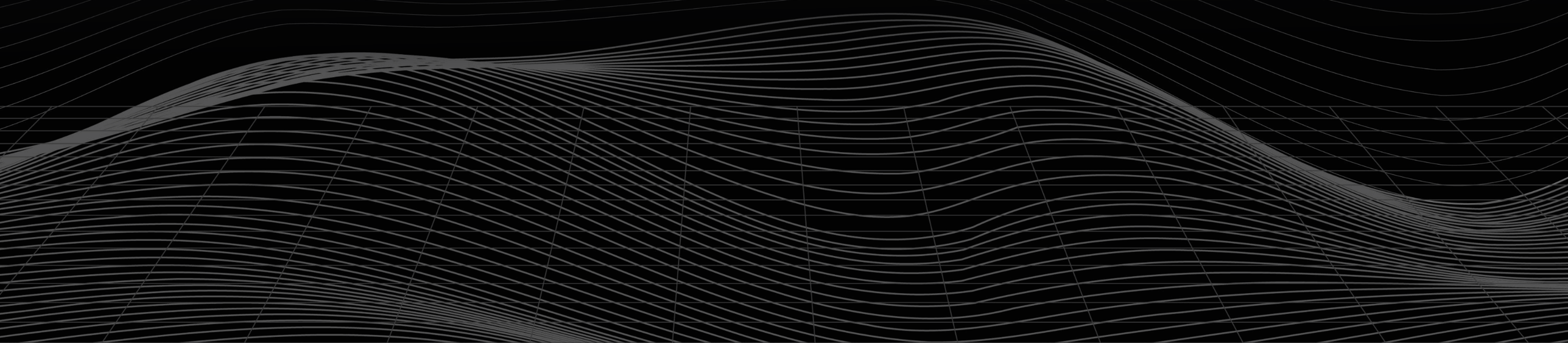
Interface allocation

EXAMPLE

Interface key in map

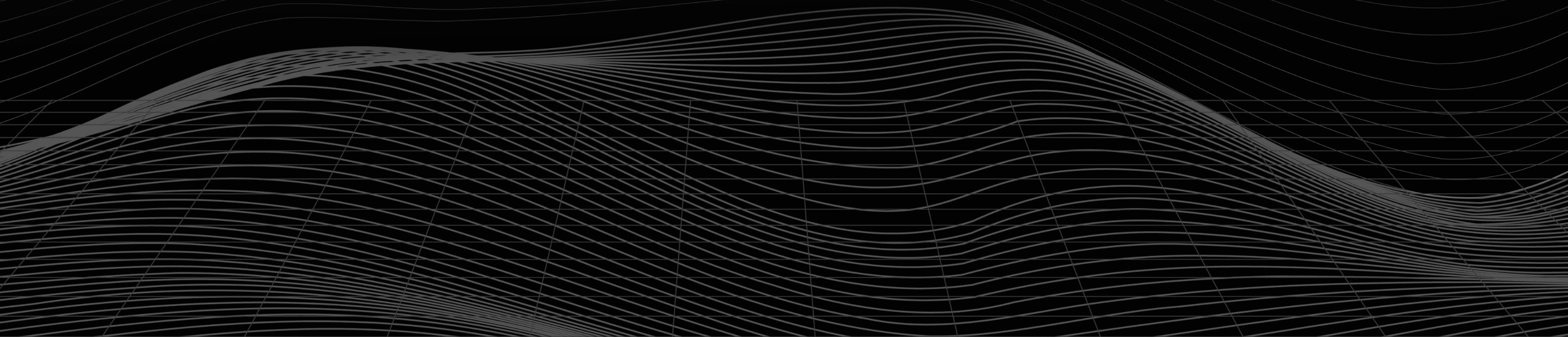
EXAMPLE

Compare interfaces



EXAMPLE

Incorrect methods



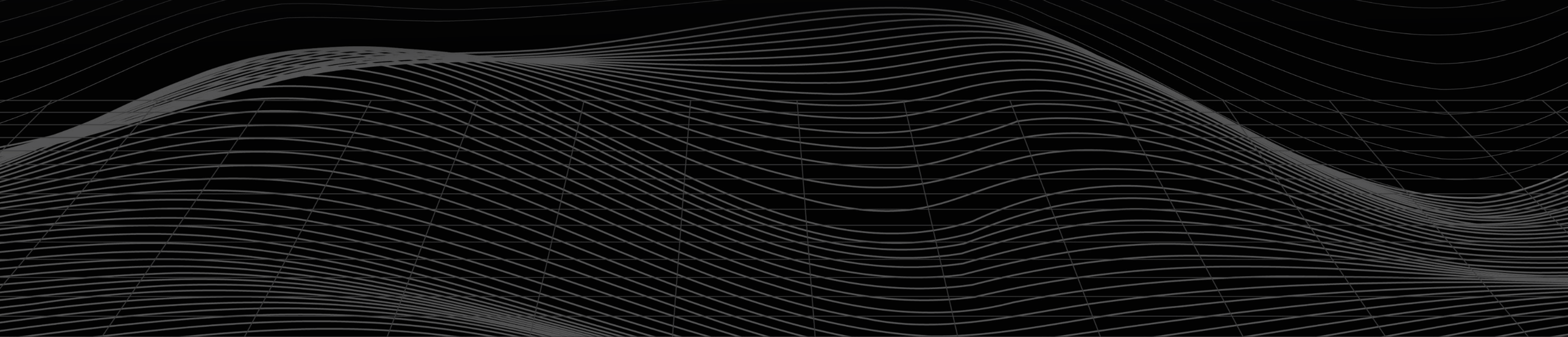
EXAMPLE

Interface guard

“Don’t do this for every type that satisfies an interface, though. By convention, such declarations are only used when there are no static conversions already present in the code, which is a rare event”

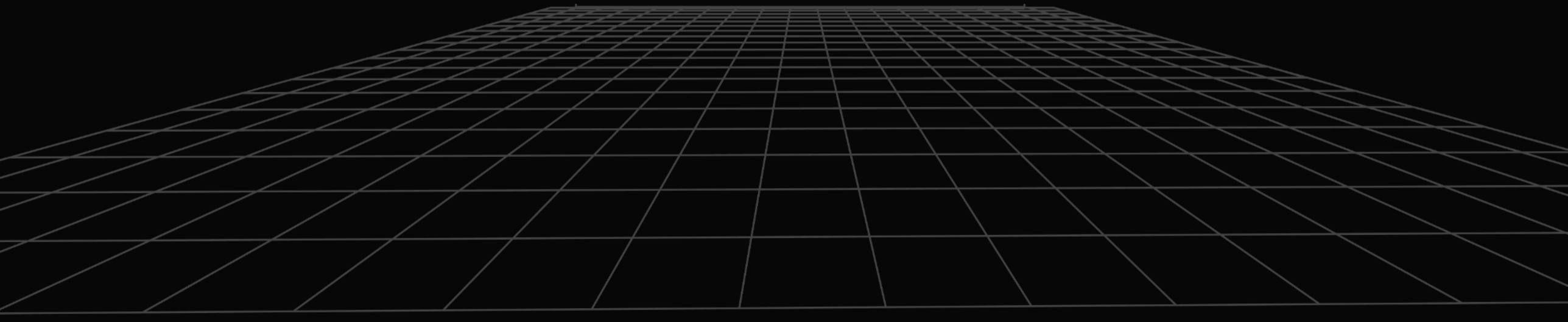
EXAMPLE

Call interface method



FAQ

Тонкости интерфейсов

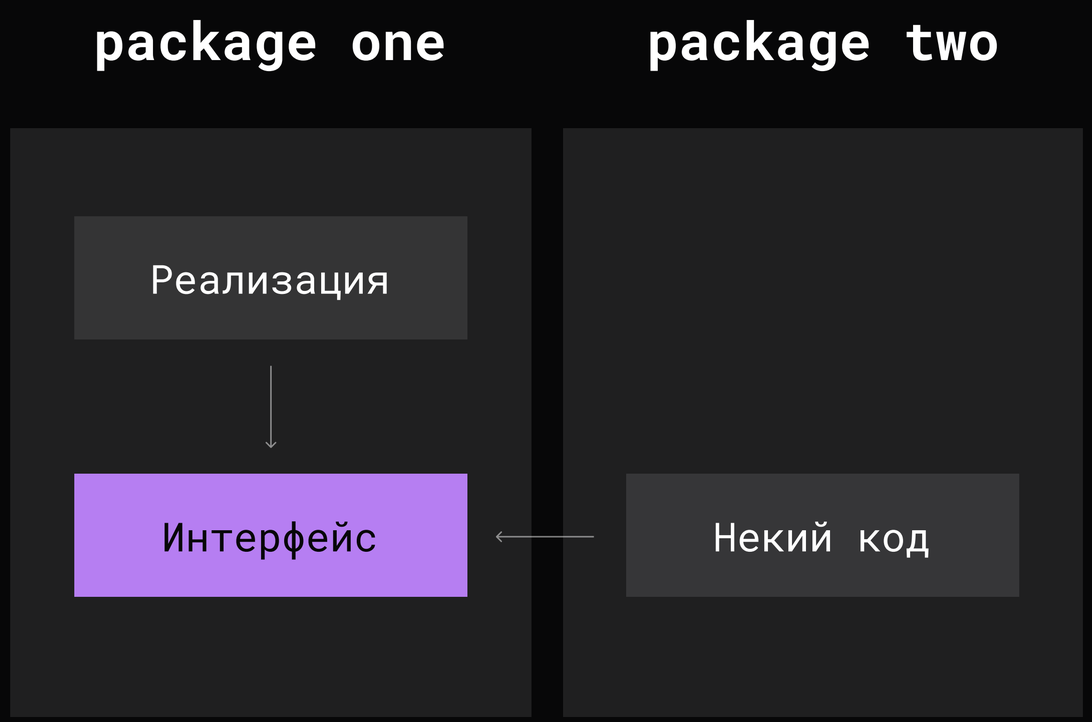


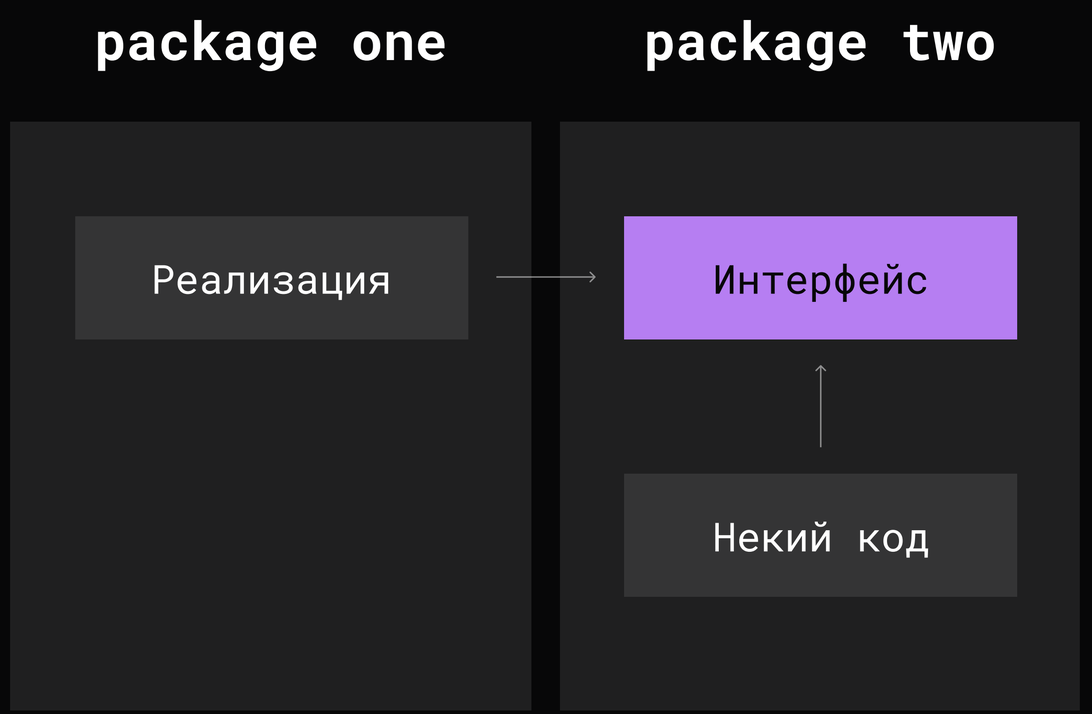
РАСПОЛОЖЕНИЕ ИНТЕРФЕЙСОВ

ИНТЕРФЕЙСЫ

- **На стороне производителя (producer)** – определенный в том же пакете, что и конкретная реализация
- **На стороне потребителя (consumer)** – определенный во внешнем пакете, где он используется

PRODUCER





CONSUMER

EXAMPLE

Producer interface

НА СТОРОНЕ ПРОИЗВОДИТЕЛЯ

- **Связность пакетов** – пакет service зависит от пакета storage
- **Понятность кода** – много ненужных методов в интерфейсе
- **Гибкость** – сложно реализацию интерфейса заменить другой реализацией (придется писать заглушки для интерфейса)
+ **Изменяемость** – просто изменить сигнатуру интерфейса в одном месте

EXAMPLE

Consumer interface

НА СТОРОНЕ ПОТРЕБИТЕЛЯ

+ **Связность пакетов** — пакет service не зависит от пакета storage

+ **Понятность кода** — нет не нужных методов в интерфейсе

+ **Гибкость** — просто реализацию интерфейса заменить другой реализацией (не придется писать никакие заглушки для интерфейса)

− **Изменяемость** — трудно изменить сигнатуру интерфейсов

В GOLANG ПРИНЯТО ОПРЕДЕЛЯТЬ ИНТЕРФЕЙСЫ ПО МЕСТУ ИСПОЛЬЗОВАНИЯ!

а также возвращать из функций не интерфейсы, а конкретные реализации

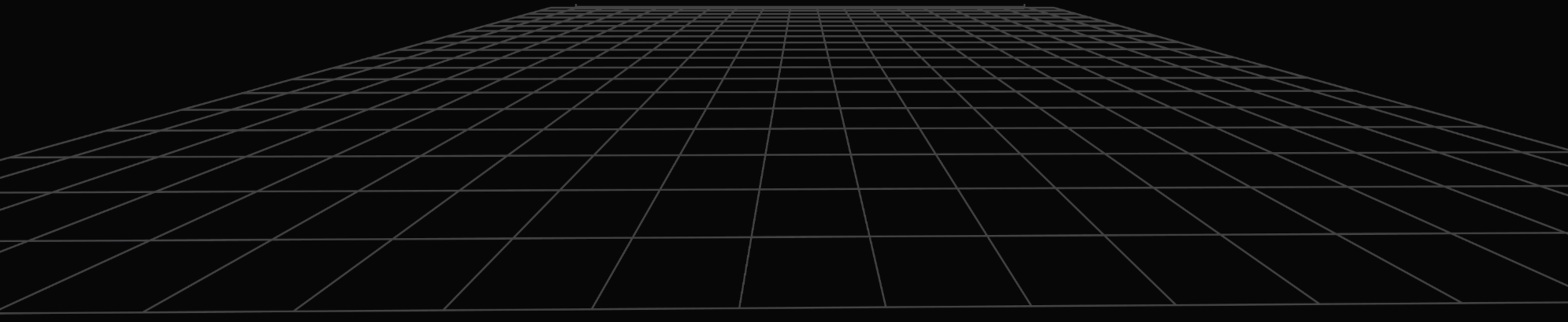
НО В СТАНДАРТНОЙ БИБЛИОТЕКЕ КОЕ-ГДЕ ИНТЕРФЕЙСЫ РАСПОЛОЖЕНЫ НА СТОРОНЕ ПРОИЗВОДИТЕЛЯ

Например, пакет **encoding** определяет интерфейсы, которые реализованы другими субпакетами **encoding/json** и **encoding/binary.**

Поэтому в определенных ситуациях (например, когда точно знаем, что абстракция будет полезна для потребителя) можно сделать интерфейс на стороне потребителя.

FAQ

Расположение интерфейсов



ПОЖАЛУЙСТА, ЗАПОЛНИ ОПРОС О ЗАНЯТИИ

Ссылка в чате и в группе участников