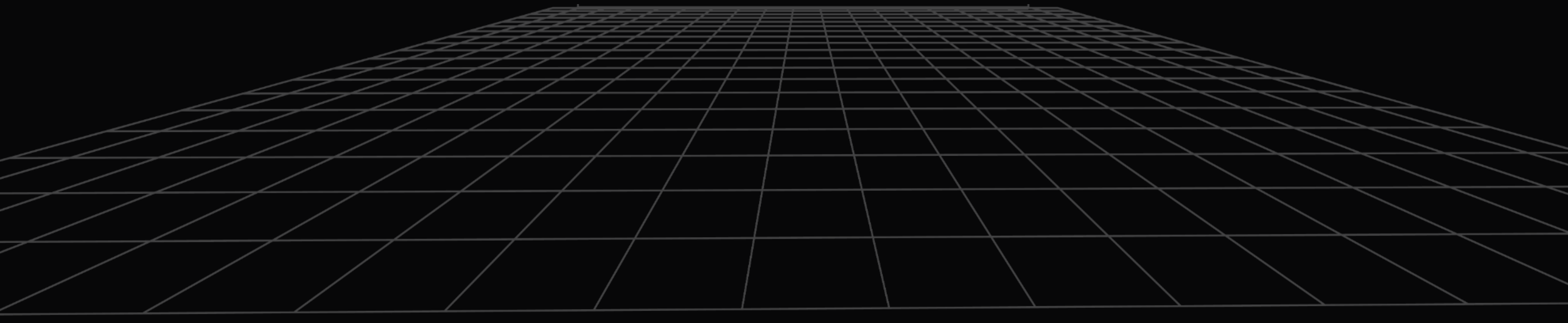
ДЖЕНЕРИКИ И РЕФЛЕКСИЯ

Balun.Courses

ПРОВЕРЬ ЗАПИСЬ

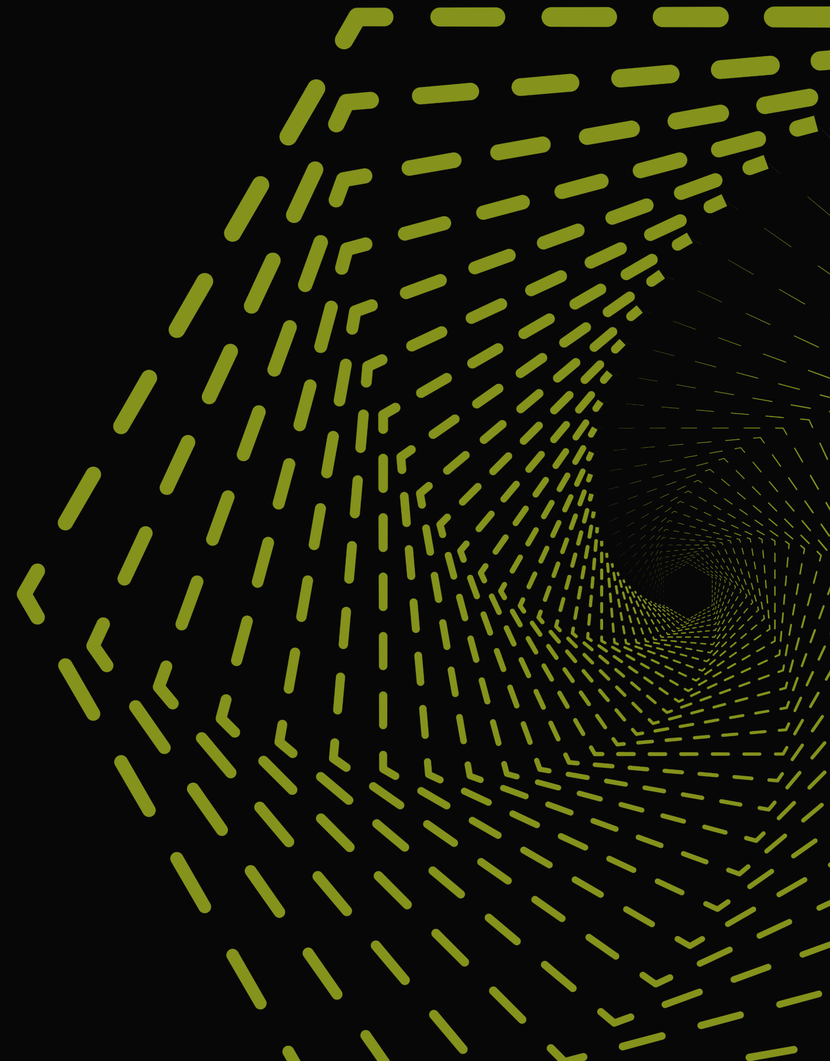


ПРАВИЛА ЗАНЯТИЯ

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

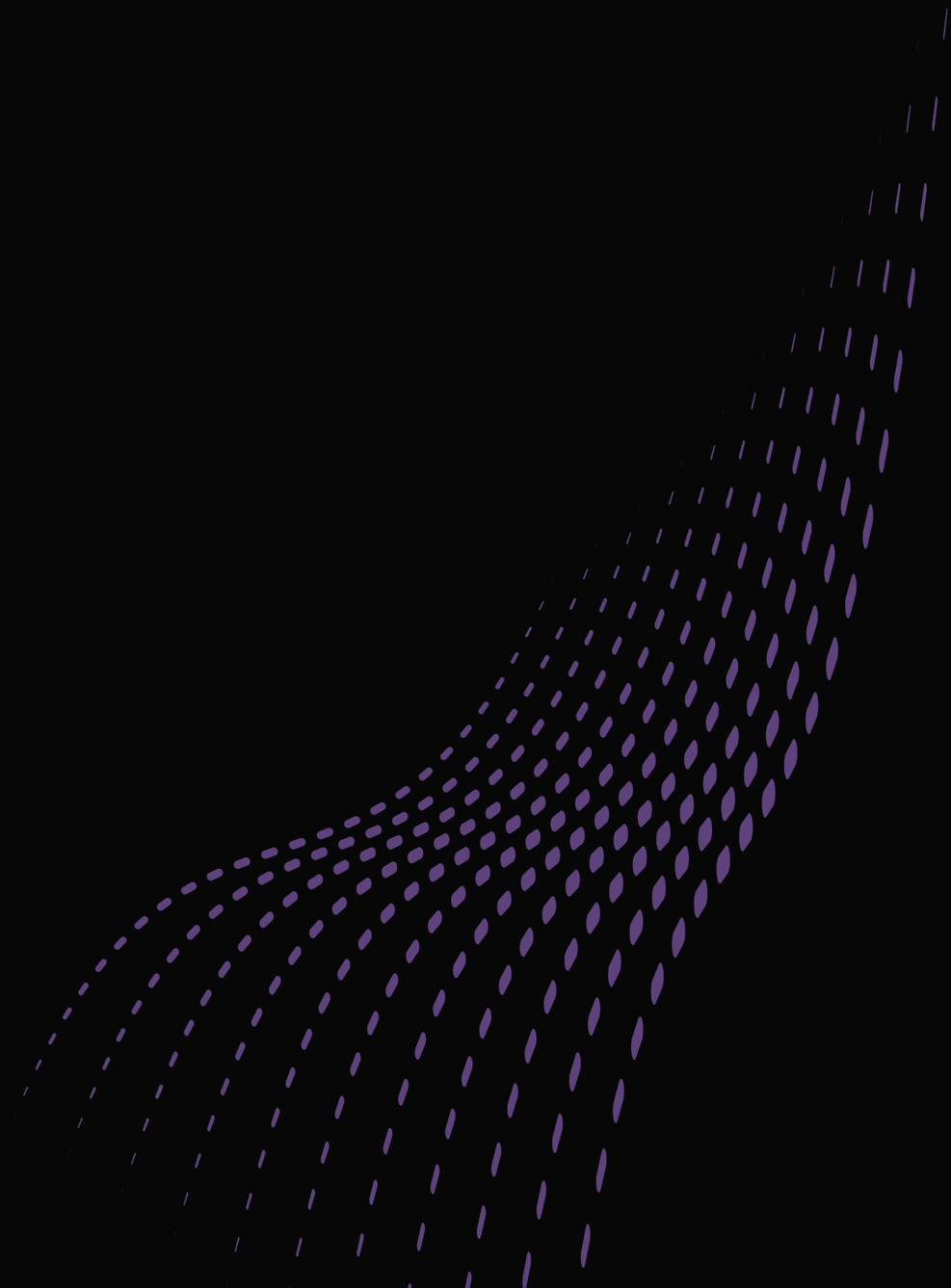
2. вопросы голосом задаем **по поднятой руке в Zoom**

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



ПЛАН ЛЕКЦИИ

1. Дженерики
2. Рефлексия



ДЖЕНЕРИКИ

Terminal: question

ПОЧЕМУ БЕЗ ДЖЕНЕРИКОВ ЖИВЕТСЯ ПЛОХО?

EXAMPLE

Max

Согласно результатам опроса **88% респондентов назвали отсутствие дженериков критической проблемой**.
18% опрошенных сказали, что не используют Go именно из-за отсутствия этой функциональности

Обобщенное программирование (метапрограммирование) представляет собой парадигму разработки программного обеспечения, которая позволяет писать гибкий и универсальный код, способный работать с различными типами данных

Начиная с версии 1.0, в Go поддерживаются встроенные универсальные типы, которые включают в себя некоторые встроенные типы универсальных типов (map, chan, …) и универсальные функции (new, make, len, close, …)

До версии 1.18 в Go было несколько способов реализации обобщенного программирования:

1. С помощью интерфейсов
2. С помощью рефлексии
3. С помощью пакета unsafe
4. С помощью кодогенерации

EXAMPLE

Max generic

Передача аргументов типа называется **инстанцированием** и она выполняется на этапе компиляции (это позволяет сохранить безопасность типов, избежать оверхеда во время выполнения, но увеличить скорость компиляции)

«**Generic functions, rather than generic types, can probably be compiled using an interface-based approach.** That will optimize compile time, in that the function is only compiled once, but there will be some run time cost.

Generic types may most naturally be compiled multiple times for each set of type arguments. This will clearly carry a compile time cost, but there shouldn't be any run time cost. Compilers can also choose to implement generic types similarly to interface types, using special purpose methods to access each element that depends on a type parameter»

EXAMPLE

Map keys

Ограничение аргументов типа для соответствия определенным требованиям называется **constraint** (ограничения) – это тип интерфейса, который может содержать набор методов или произвольные типы (связь между ограничением и параметром типа аналогична связи между типом и значением)

EXAMPLE

Constraints 1

EXAMPLE

Constraints 2

Terminal: deep_golang

constraints package

Version: v0.0.0-...-2d47ceb Latest | Published: Nov 8, 2024 | License: BSD-3-Clause

Details Valid go.mod file | Redistributable license | Tagged version | Stable version | Learn more about best practices

Repository cs.opensource.google/go/x/exp

Links Report a Vulnerability | Open Source Insights

Jump to ... f

Documentation

Overview

Package constraints defines a set of useful constraints to be used with type parameters.

Index

Использование **int** ограничивает тип только этим типом, а **~int** ограничивает все типы, базовым типом которых является int

EXAMPLE

Constraints union

EXAMPLE

Constraints intersection

EXAMPLE

Constraint with method

EXAMPLE

Constraint with field incorrect

Terminal: question

КАК БЫТЬ С CONSTRAINTS ДЛЯ ПОЛЕЙ СТРУКТУР?

EXAMPLE

Constraint with field correct

EXAMPLE

Generic constraint

В GO МОГУТ БЫТЬ ОБОБЩЕННЫМИ НЕ ТОЛЬКО ФУНКЦИИ, НО ЕЩЕ И СТРУКТУРЫ

EXAMPLE

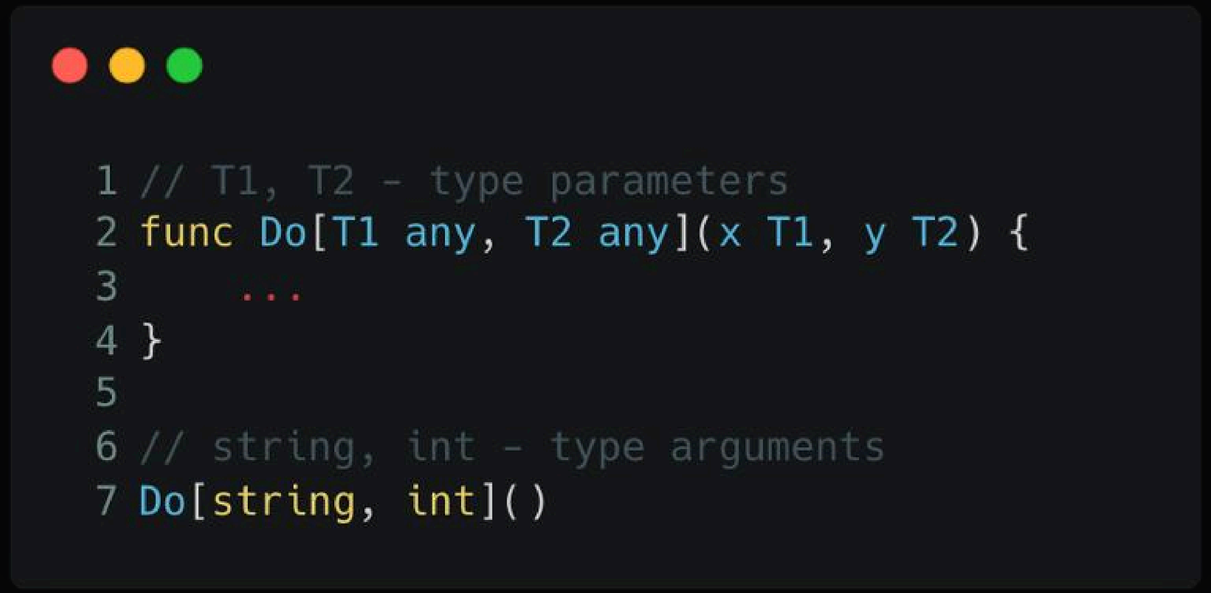
Generic set

EXAMPLE

Generic variadic parameters

Типы параметры – типы
в объявлении функции

Типы аргументы – типы,
переданные при вызове функции



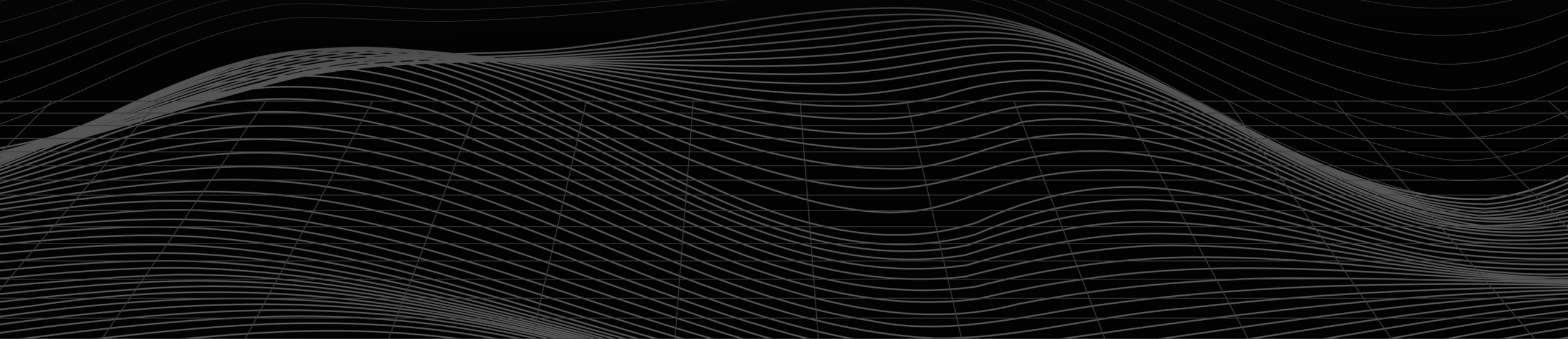
```
1 // T1, T2 - type parameters
2 func Do[T1 any, T2 any](x T1, y T2) {
3     ...
4 }
5
6 // string, int - type arguments
7 Do[string, int]()
```

TYPE INFERENCE

Type inference пытается угадать, какой тип вы имели в виду, исследуя типы аргументов

EXAMPLE

Type inference



EXAMPLE

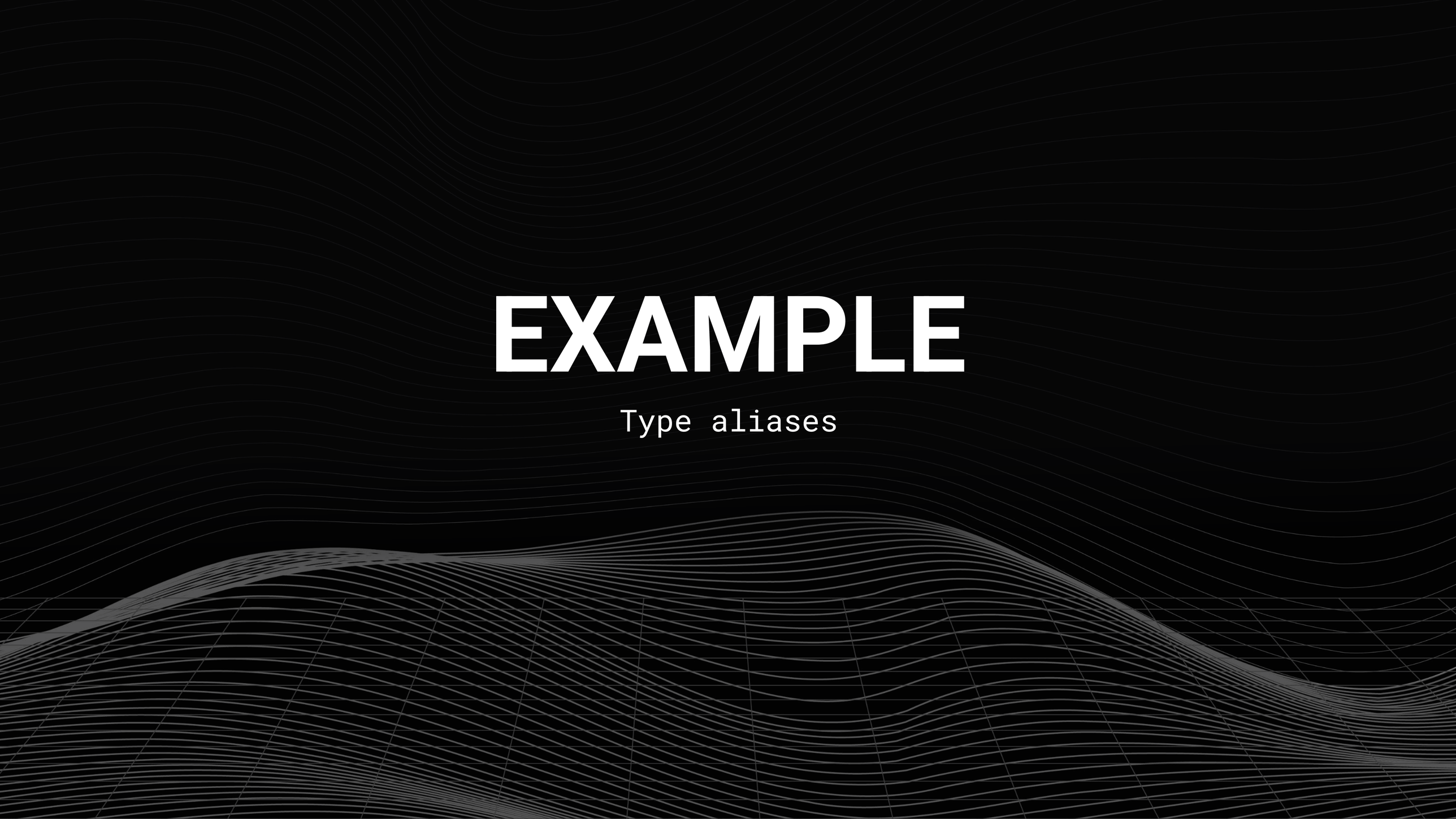
Type parameters skipping

EXAMPLE

Unnamed types

EXAMPLE

Type aliases

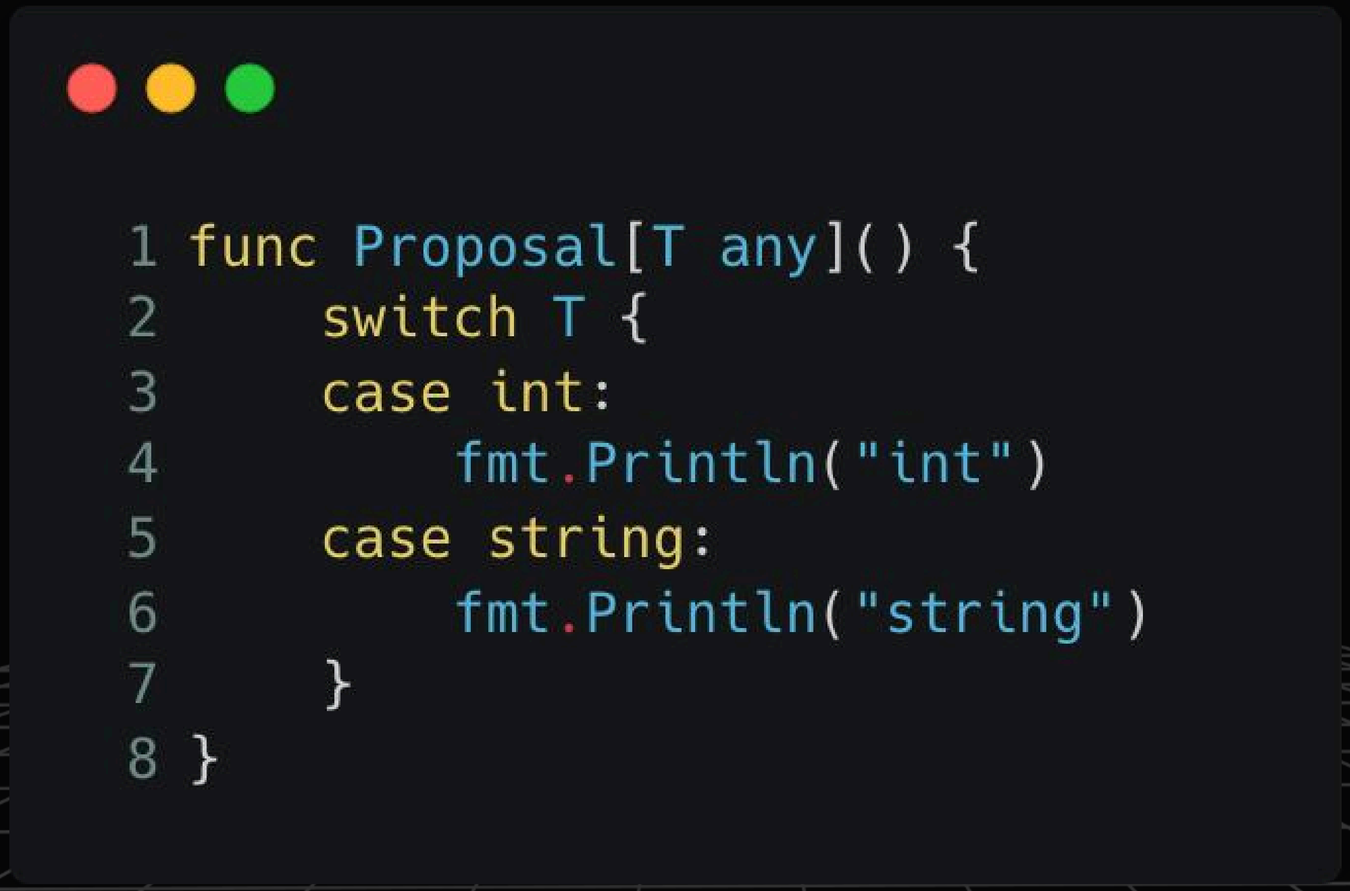


EXAMPLE

Partial type alias

EXAMPLE

Type assertions



```go
func Proposal[T any]() {
    switch T {
    case int:
        fmt.Println("int")
    case string:
        fmt.Println("string")
    }
}
```

EXAMPLE

Type parameter embedding

Terminal: question

КОГДА ИСПОЛЬЗОВАТЬ ДЖЕНЕРИКИ?

Если вы обнаружите, что пишете один и тот же код несколько раз, и единственная разница между копиями заключается в том, что код использует разные типы, рассмотрите возможность использования дженериков

ДЖЕНЕРИКИ

Структуры данных – когда реализуем различные обобщенные структуры данных (например куча, двоичное дерево, …)

Функции – когда функция работает со срезами, картами, каналами любых типов (например функция слияния каналов)

ОСНОВНАЯ ЦЕЛЬ ИСПОЛЬЗОВАНИЯ ДЖЕНЕРИКОВ —

ИЗБЕЖАТЬ ПОВТОРЕНИЯ КОДА

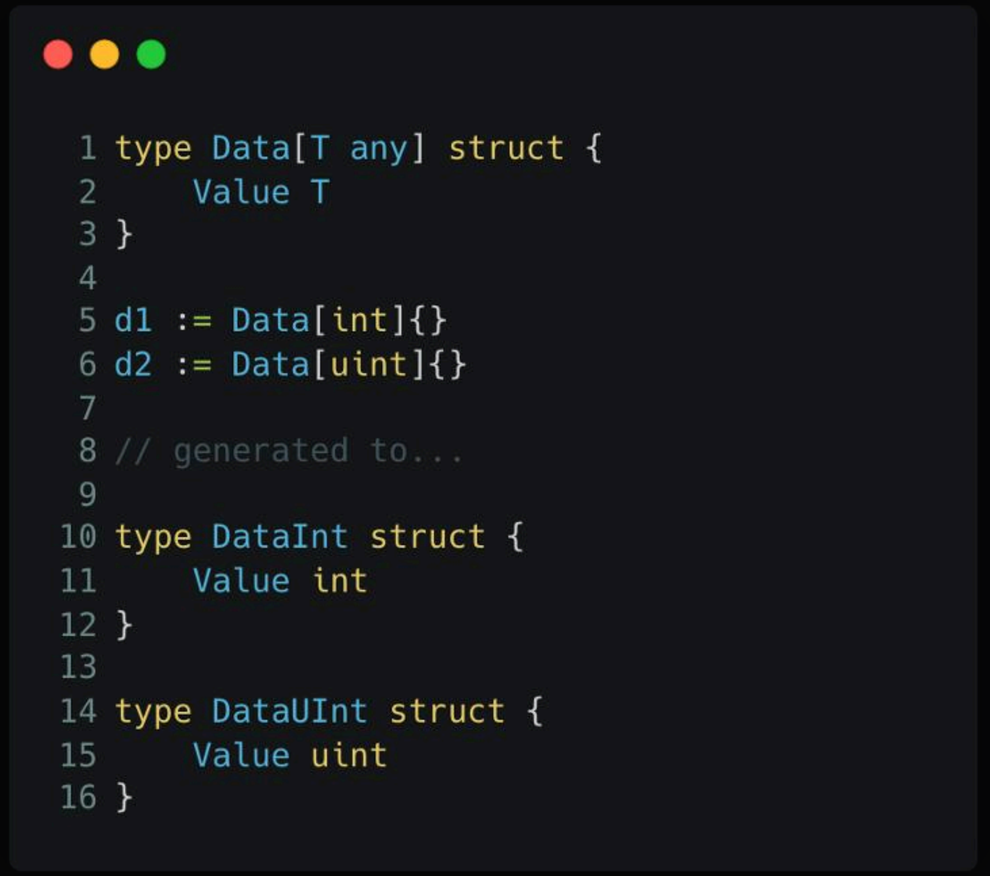
или, другими словами, повысить возможность повторного использования кода

В некоторых ситуациях дженерики также могут привести к более чистому коду и упрощению API, а также могут повысить производительность выполнения

ВАЖНО, ЧТОБЫ ДЖЕНЕРИКИ НЕ ДЕЛАЛИ КОД СЛОЖНЕЕ!

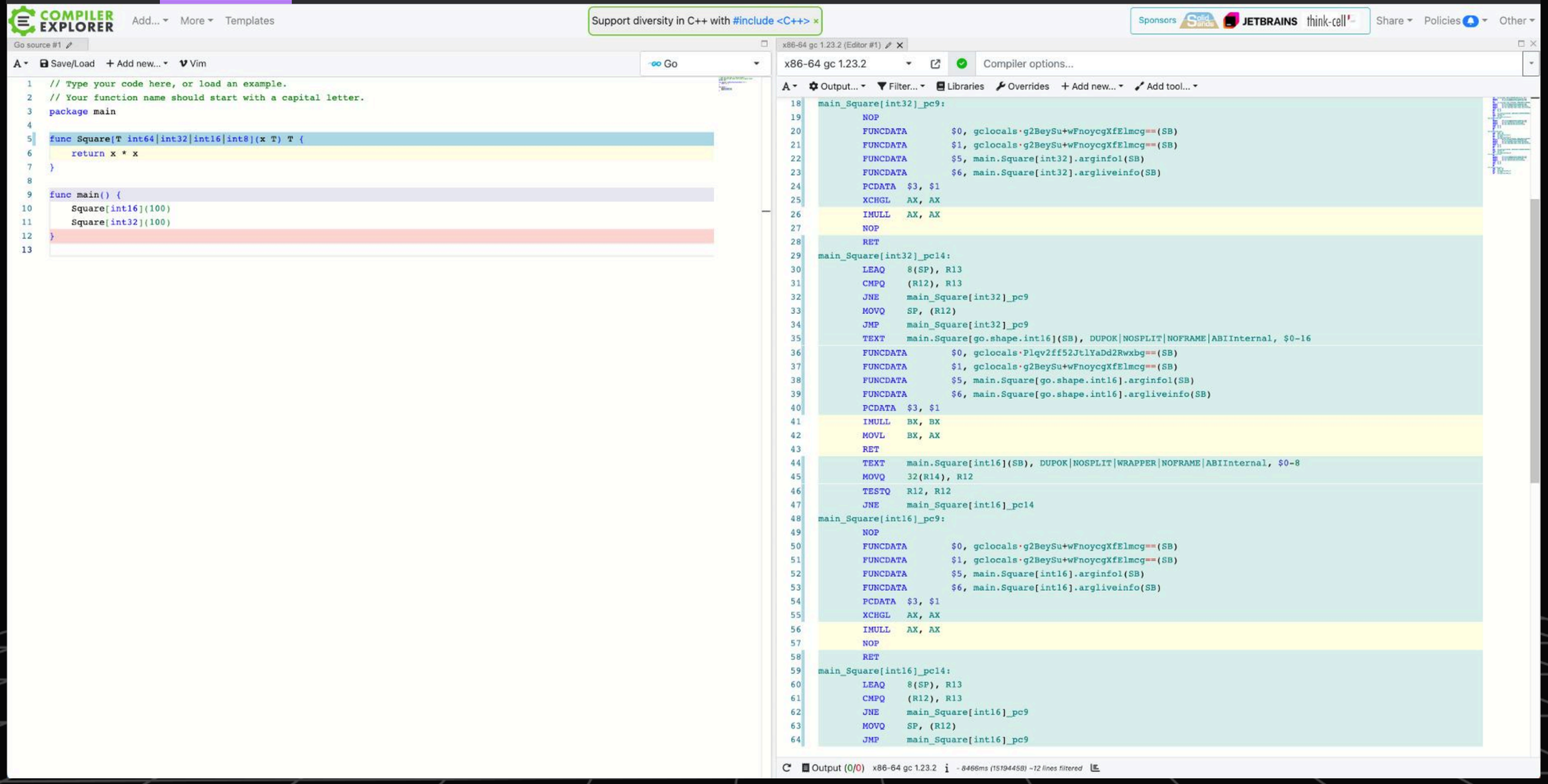
Нужно помнить, что дженерики — это не просто синтаксический сахар, они **могут влиять на размер вашего скомпилированного кода.**

Потому что каждая специализация дженерик-функции или типа создает новую версию этой функции или типа для каждого используемого набора типов



```
type Data[T any] struct {
    Value T
}

d1 := Data[int]{}
d2 := Data[uint]{}

// generated to...

type DataInt struct {
    Value int
}

type DataUInt struct {
    Value uint
}
```

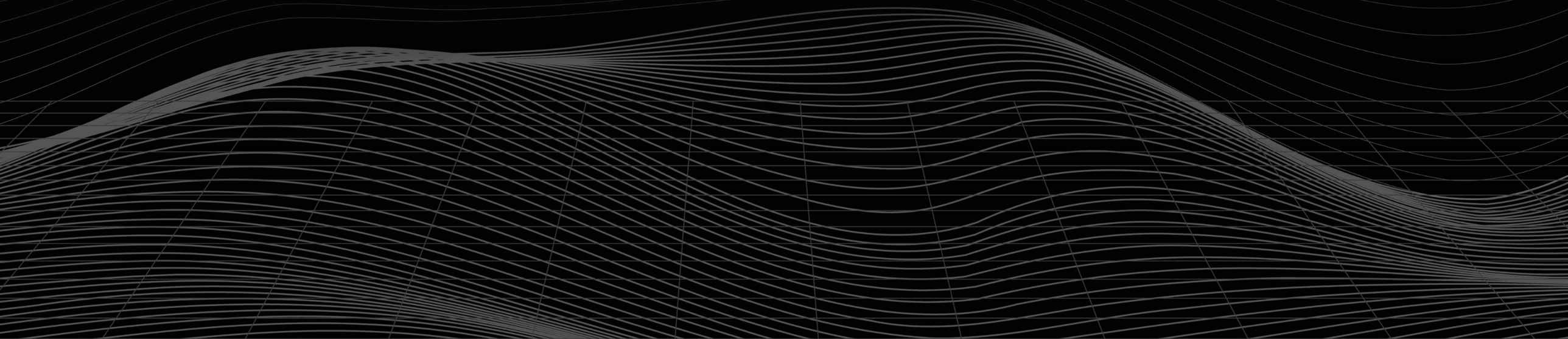
Terminal: deep_golang



ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ДЖЕНЕРИКОВ

EXAMPLE

Generics with unsafe



EXAMPLE

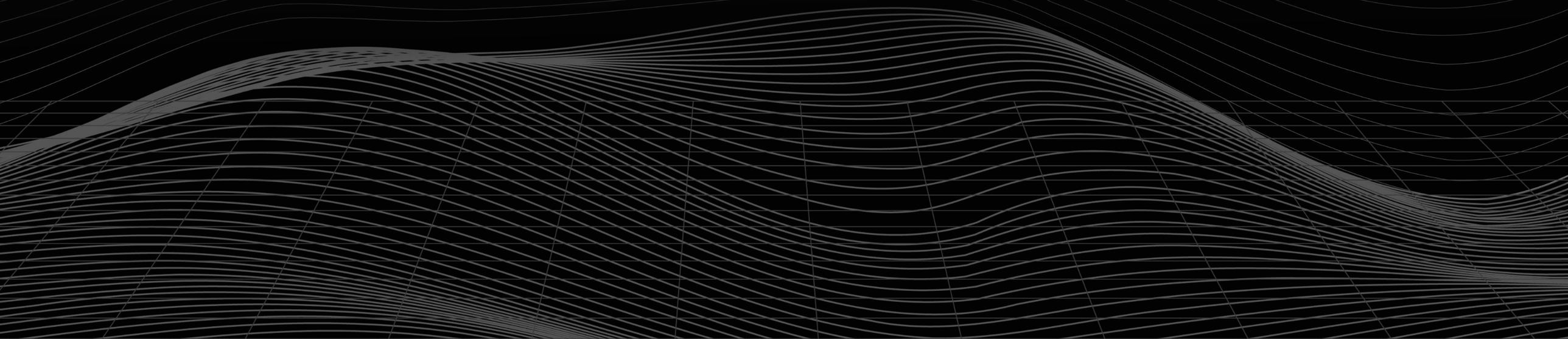
Universal fabric

EXAMPLE

Generic decorator

EXAMPLE

Cloneable mixin

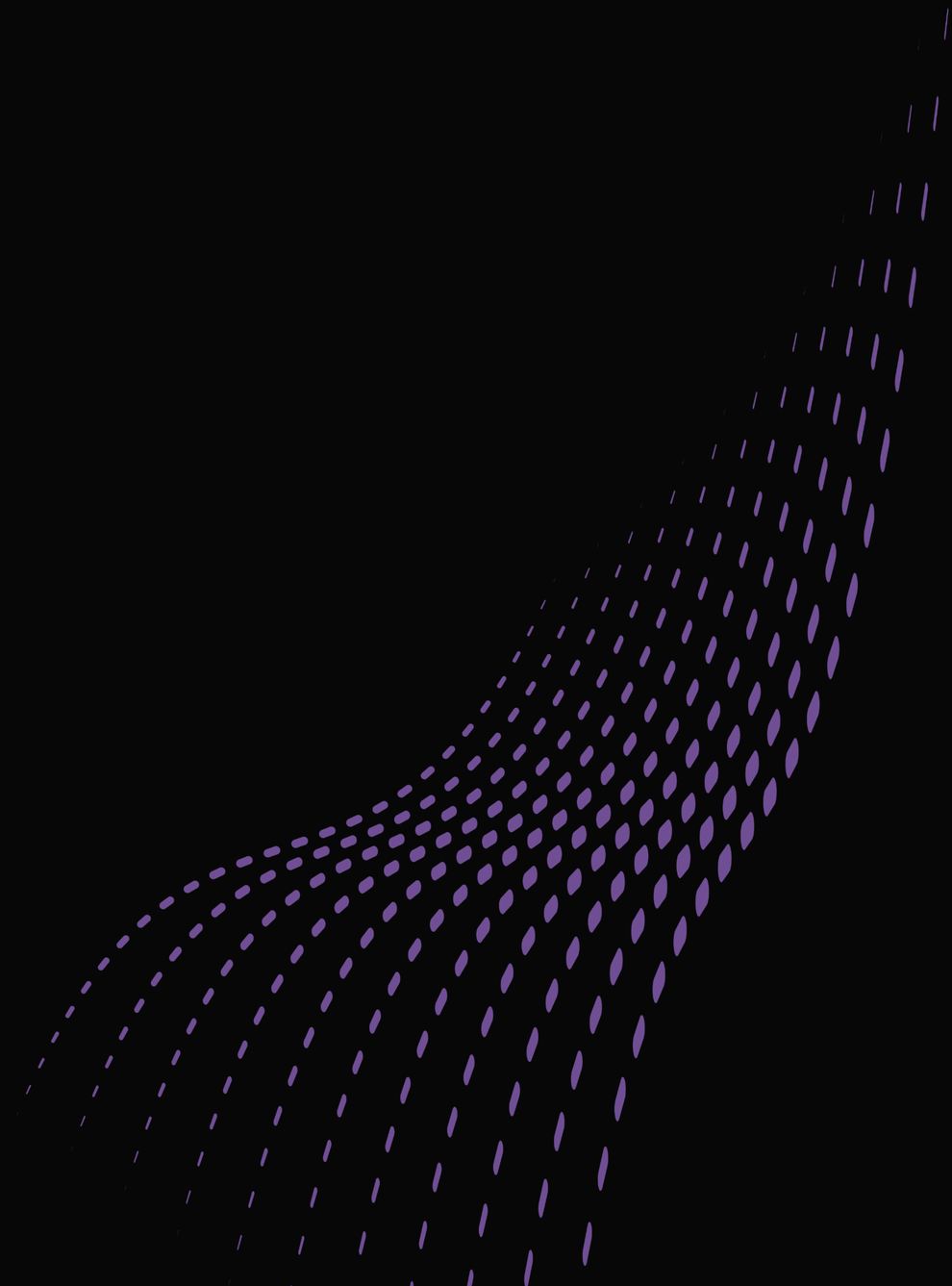


ПРЕИМУЩЕСТВА

Повторное использование кода - дженерики позволяют создавать гибкие функции и типы данных, которые можно использовать с различными типами данных без дублирования кода

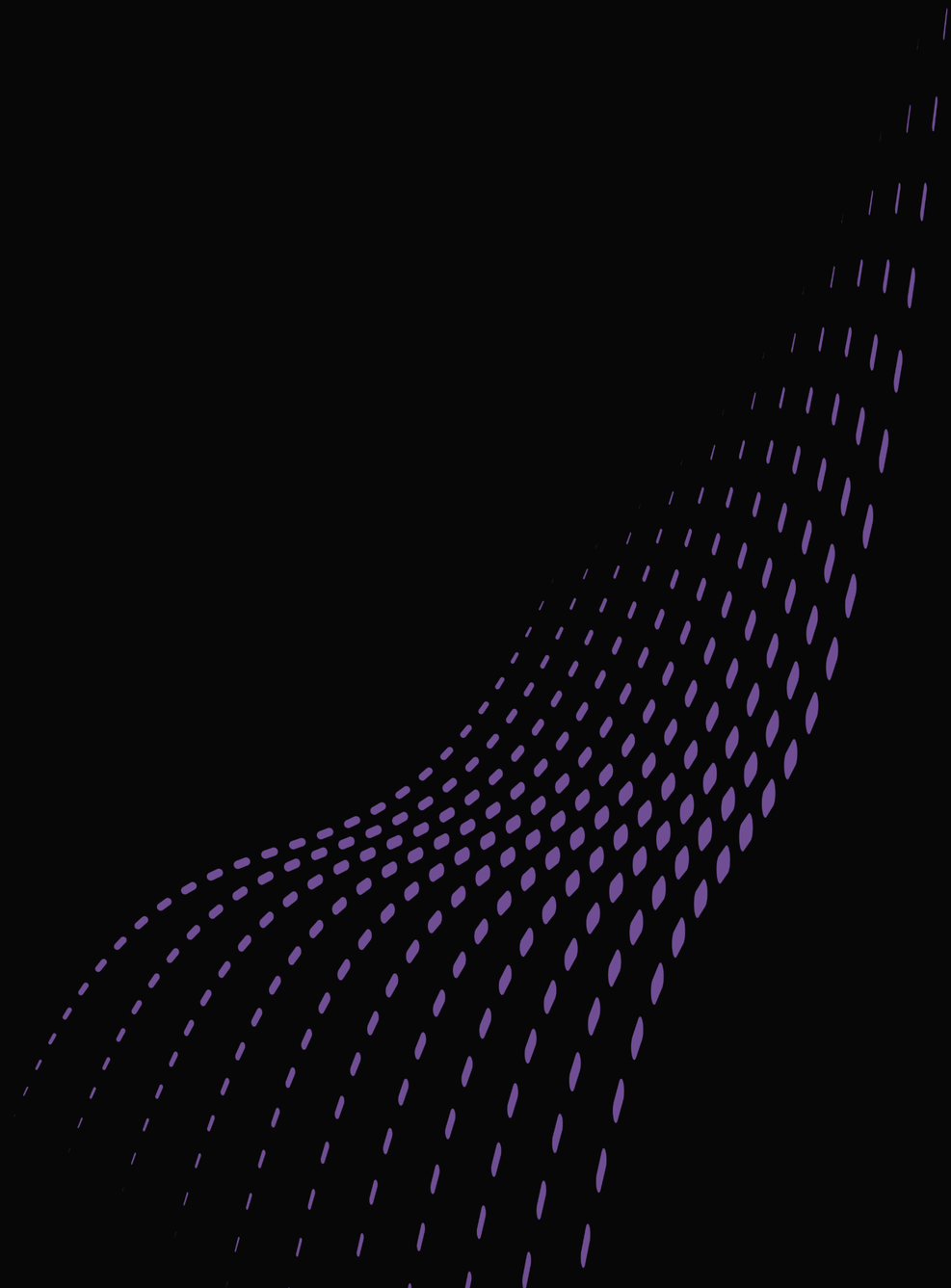
Типобезопасность - в отличие от использования пустых интерфейсов, дженерики обеспечивают проверку типов на этапе компиляции, что уменьшает риск ошибок времени выполнения

Производительность - использование дженериков может улучшить производительность, так как избавляет от необходимости преобразования типов



НЕДОСТАТКИ

Сложность - синтаксис дженериков может быть сложным для понимания, особенно для начинающих разработчиков, а также обобщенное программирование может существенно усложнить структуру кода и его читаемость



Terminal: question

ЧТО НЕЛЬЗЯ ДЕЛАТЬ С ДЖЕНЕРИКАМИ В GO?

EXAMPLE

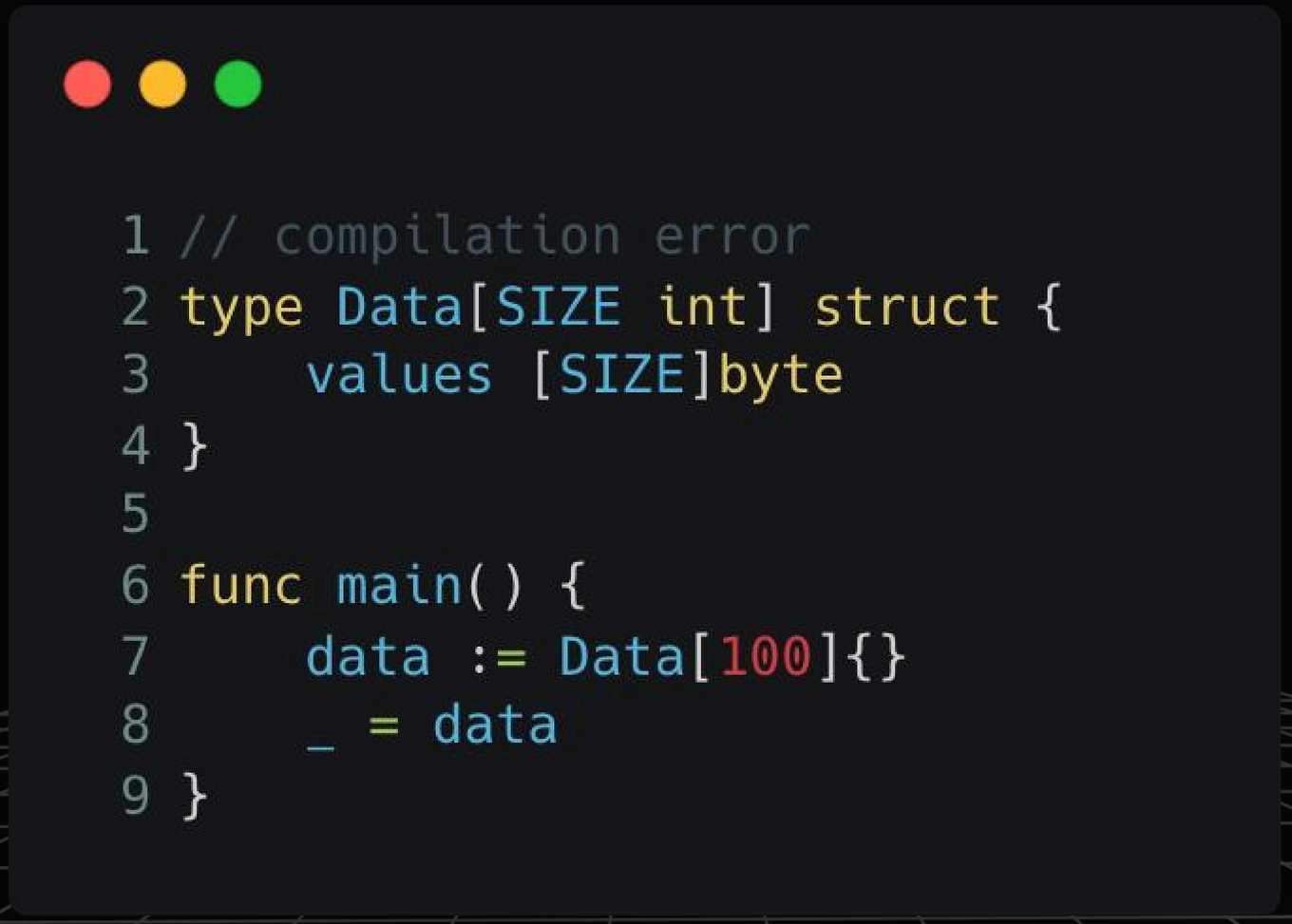
Generic method incorrect

EXAMPLE

Generic method correct

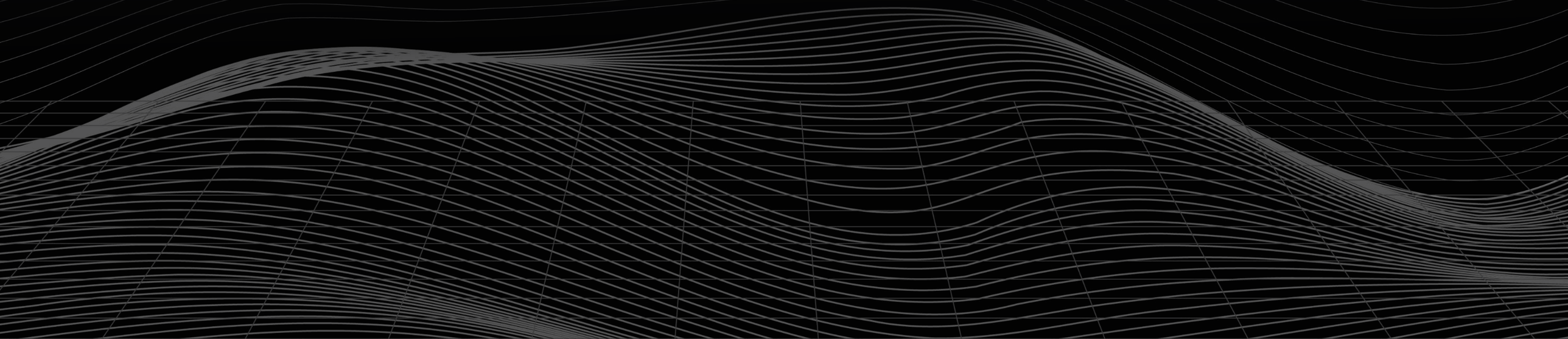
EXAMPLE

Generics with constants


// compilation error
type Data[SIZE int] struct {
values [SIZE]byte
}
func main() {
data := Data[100]{}
_ = data
}

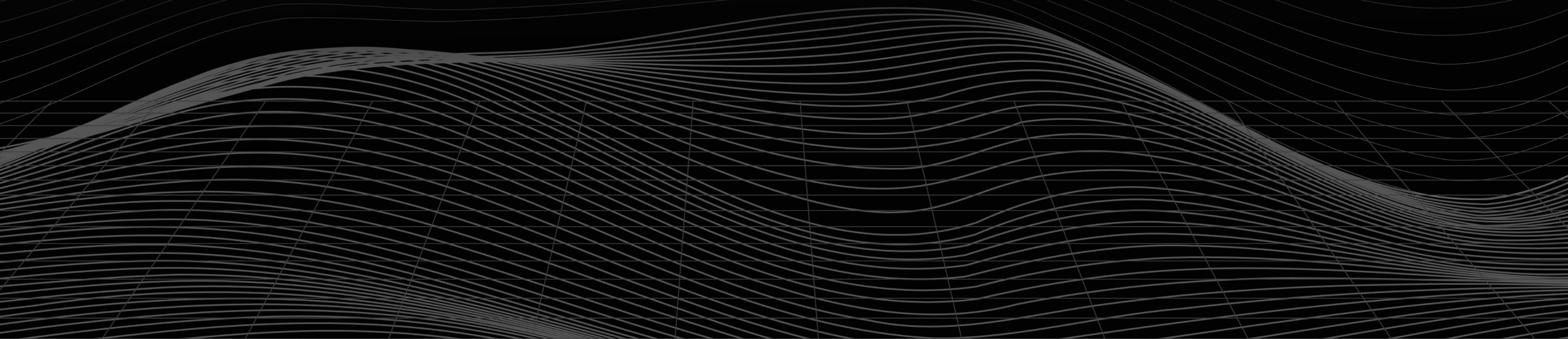
EXAMPLE

Generics with pointers



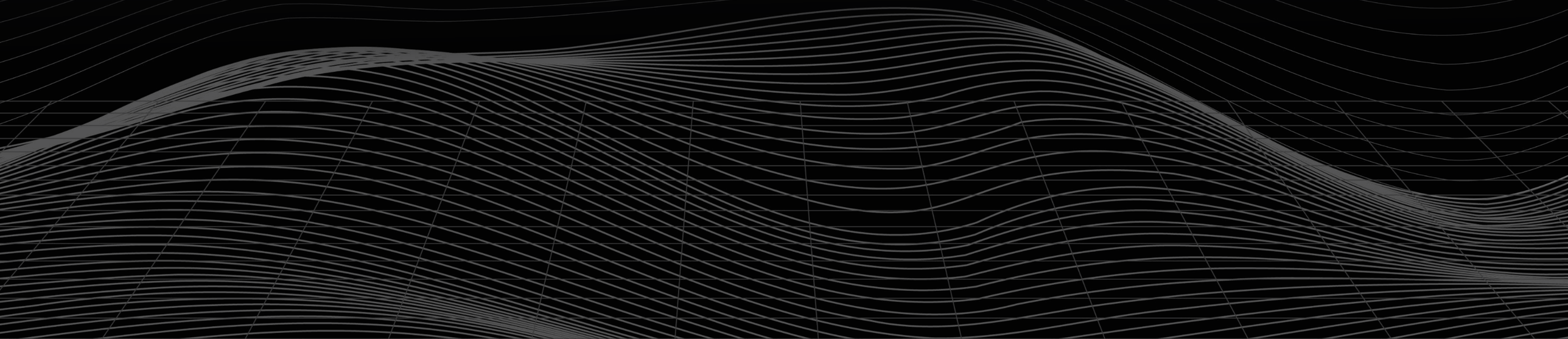
EXAMPLE

Generics maps and slices



EXAMPLE

Generics slices and strings

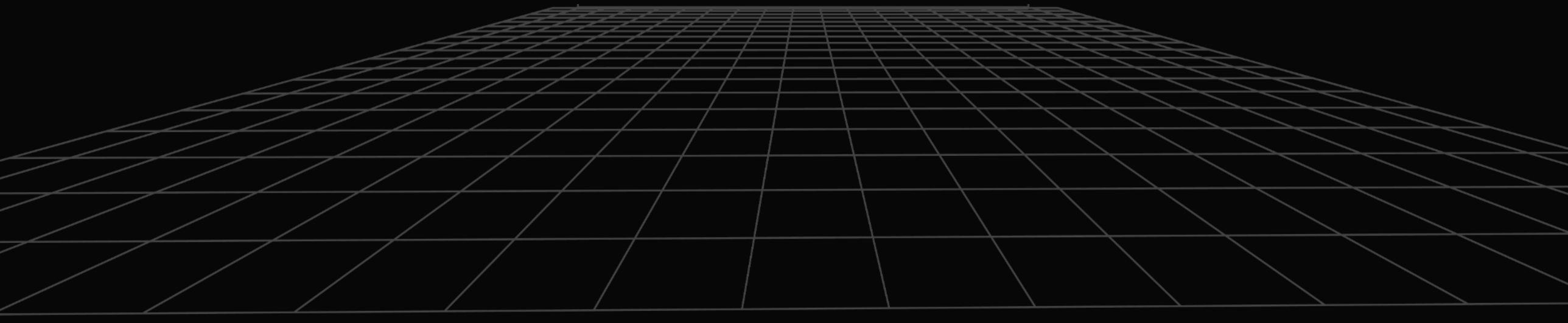


EXAMPLE

Generics method set

FAQ

Дженерики



ПЕРЕРЫВ 5 МИНУТ

РЕФЛЕКСИЯ

Terminal: deep_go

ИНТРОСПЕКЦИЯ

Способность программы исследовать тип или свойства объектов во время работы программы

Terminal: deep_go

РЕФЛЕКСИЯ

Способность компьютерной программы изучать
и модифицировать свою структуру и поведение (значения,
мета-данные, свойства и функции) во время выполнения
(отдельная форма метапрограммирования)

У РЕФЛЕКСИИ В GO ЕСТЬ НЕКОТОРЫЕ ОСОБЕННОСТИ

#1 РЕФЛЕКСИЯ РАСПРОСТРАНЯЕТСЯ

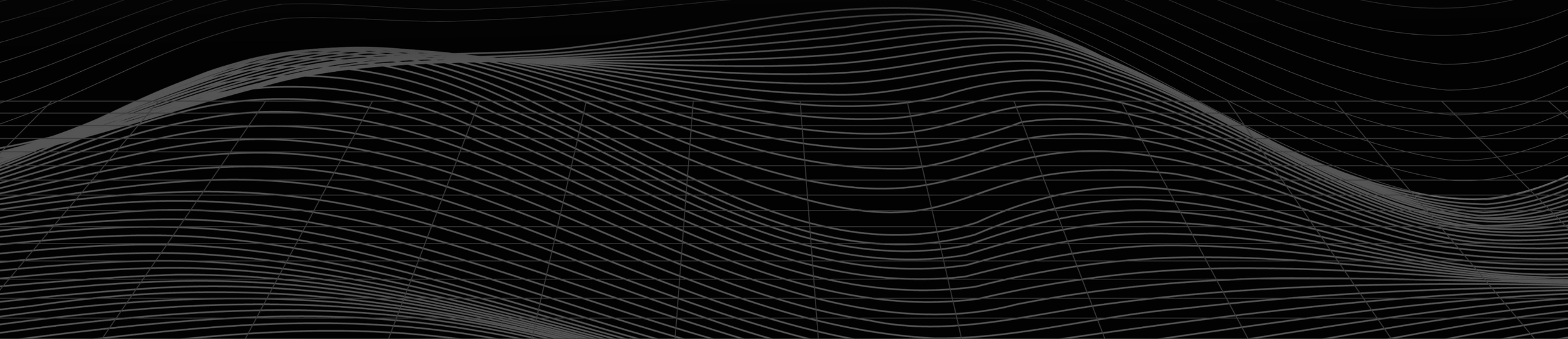
ОТ ИНТЕРФЕЙСА ДО REFLECTION ОБЪЕКТА

На базовом уровне reflect является всего лишь механизмом для изучения пары тип и значение, хранящейся внутри переменной интерфейса.

Чтобы начать работу, есть два типа, о которых нам нужно знать: **reflect.Type и reflect.Value.** Эти два типа предоставляют доступ к содержимому интерфейсной переменной и возвращаются простыми функциями, **reflect.TypeOf()** и **reflect.ValueOf()** соответственно (они выделяют части из значения интерфейса)

EXAMPLE

Reflect typeof and valueof

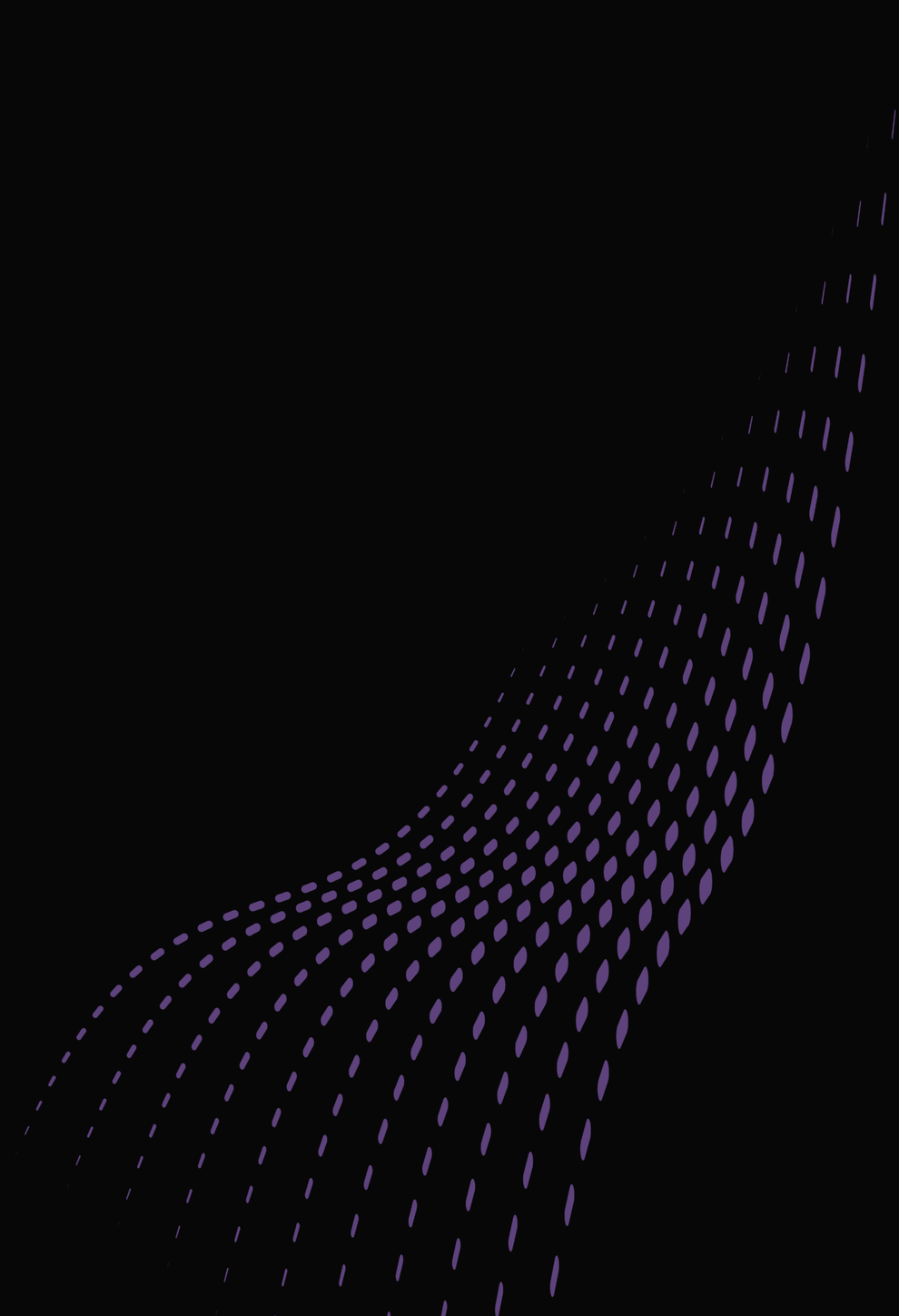


БИБЛИОТЕКА REFLECT ИМЕЕТ ПАРУ СВОЙСТВ, КОТОРЫЕ НУЖНО ВЫДЕЛИТЬ

ВО-ПЕРВЫХ

ЧТОБЫ API БЫЛ ПРОСТ

«getter» и «setter» методы Value действуют на самый большой тип, который может содержать значение: int64 для всех целых чисел со знаком



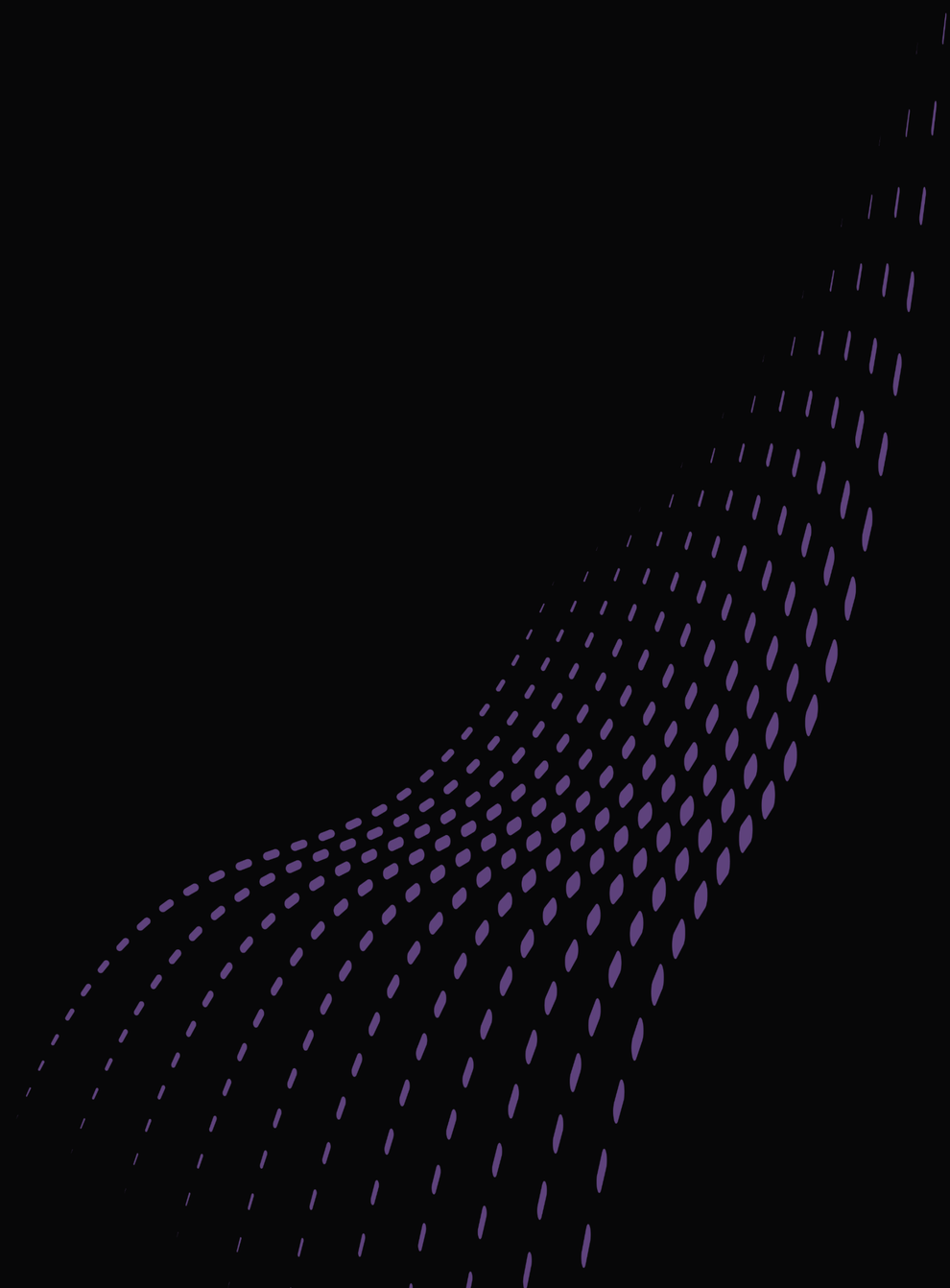
EXAMPLE

Reflect with big type

ВТОРОЕ СВОЙСТВО

KIND() REFLECT ОБЪЕКТА

описывает базовый тип, например
в случае определения нового типа



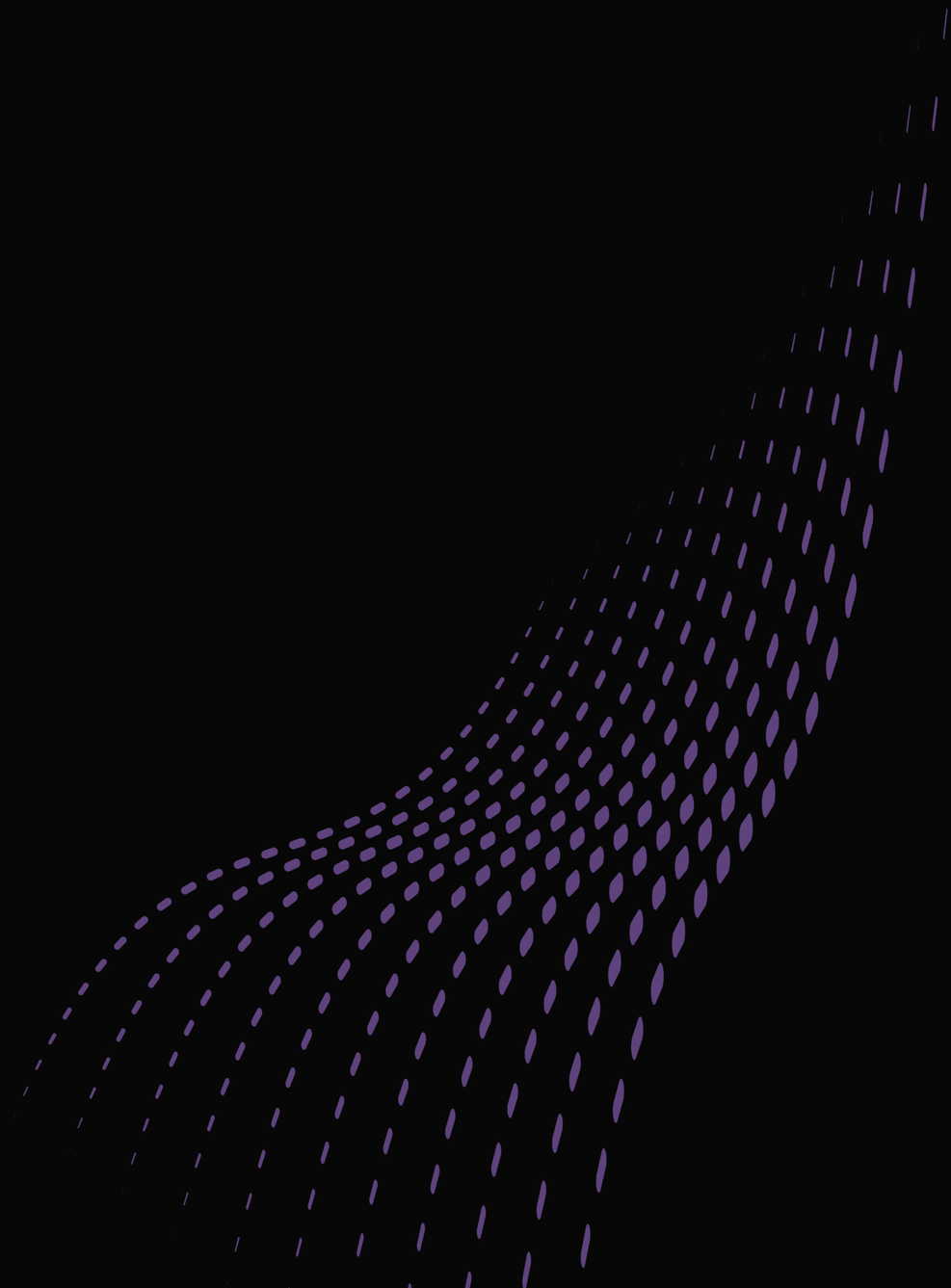
EXAMPLE

Reflect with type definition

#2 РЕФЛЕКСИЯ РАСПРОСТРАНЯЕТСЯ

ОТ REFLECTION ОБЪЕКТА ДО ИНТЕРФЕЙСА

Имея reflect.Value, мы можем восстановить значение интерфейса с помощью метода Interface(). Метод упаковывает информацию о типе и значении обратно в интерфейс и возвращает результат



EXAMPLE

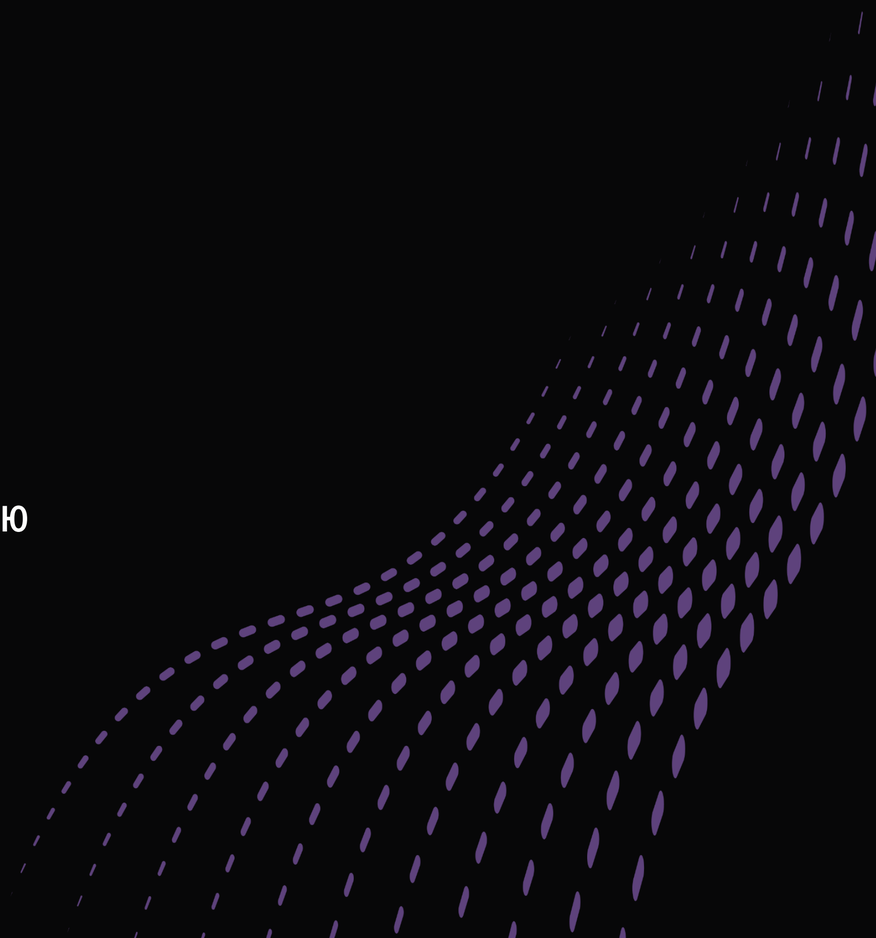
Reflect interface method

#3 ЧТОБЫ ИЗМЕНИТЬ ОБЪЕКТ
С ИСПОЛЬЗОВАНИЕМ РЕФЛЕКСИИ

ЗНАЧЕНИЕ ДОЛЖНО БЫТЬ УСТАНАВЛИВАЕМЫМ

Устанавливаемость немного напоминает адресуемость, но строже. Это свойство при котором reflection объект может изменить хранимое значение, которое было использовано при создании reflection объекта.

Устанавливаемость определяется тем, содержит ли reflection объект исходный элемент или только его копию



EXAMPLE

Reflect can set

Это не должно казаться странным. Например, при передачи значения в функцию, мы не ожидаем, что функция сможет изменить значение, потому что мы передали копию значения. Если мы хотим, чтобы функция меняла значение, **мы должны передать функции указатель на значение**

Рефлексия работает также - если мы хотим изменить значение с помощью reflection, мы должны предоставить библиотеке reflection указатель на значение, которое мы хотим изменить

EXAMPLE

Reflect elem

EXAMPLE

Reflect with struct fields

ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ РЕФЛЕКСИИ

“One of the Go reflection design goals is any non-reflection operation should be also possible to be applied through the reflection ways.

For all kinds of reasons, this goal is not 100 percent achieved. However, most non-reflection operations can be applied through the reflection ways now”

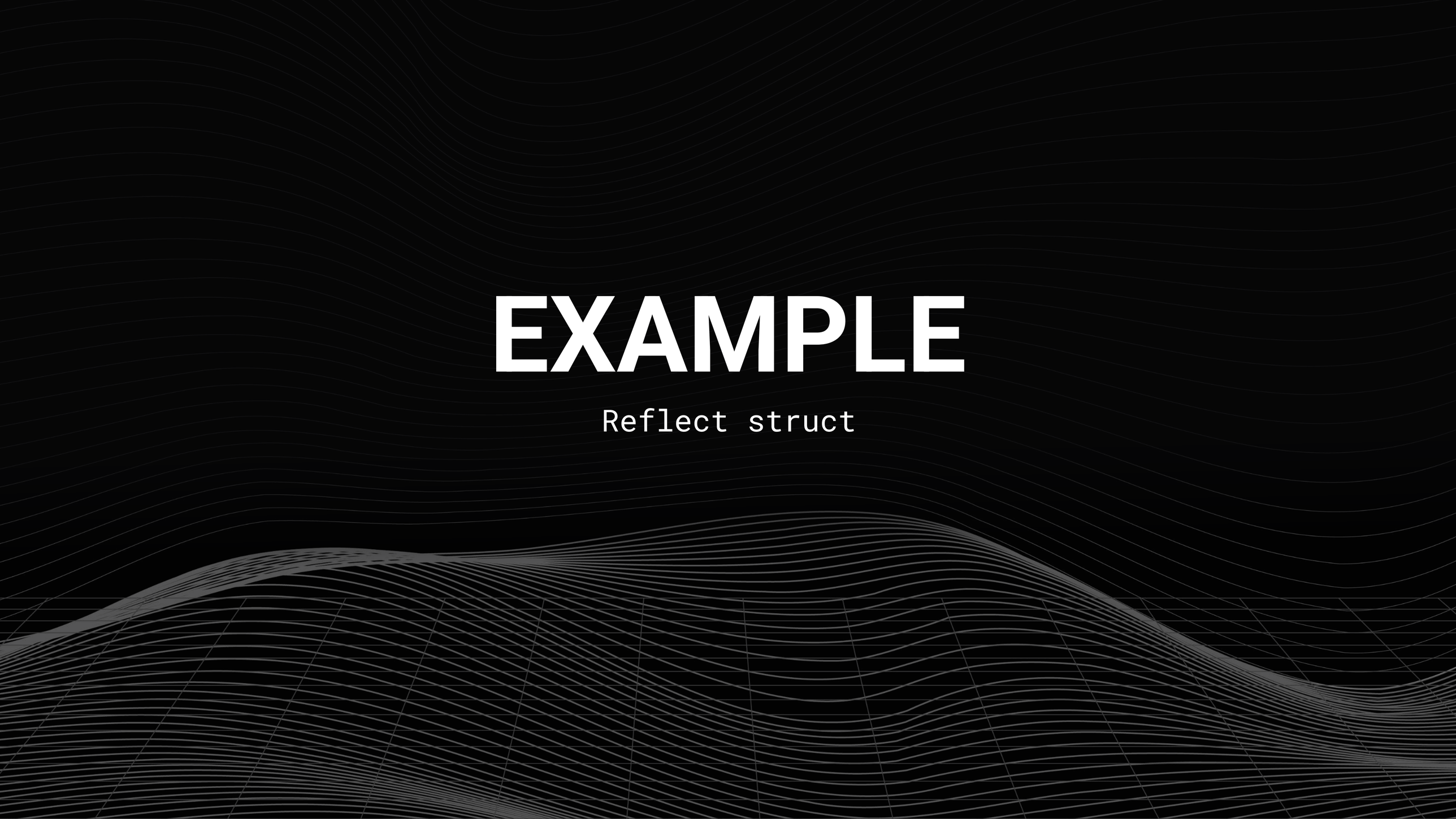
Note, up to now (Go 1.22) there are no ways to create interface types through reflection.

Another limitation is, although we can create a struct type embedding other types as anonymous fields through reflection, the struct type may or may not obtain the methods of the embedded types, and creating a struct type with anonymous fields even might panic at run time. In other words, the behavior of creating struct types with anonymous fields is partially compiler dependent.

The third limitation is we can't declare new types through reflection.

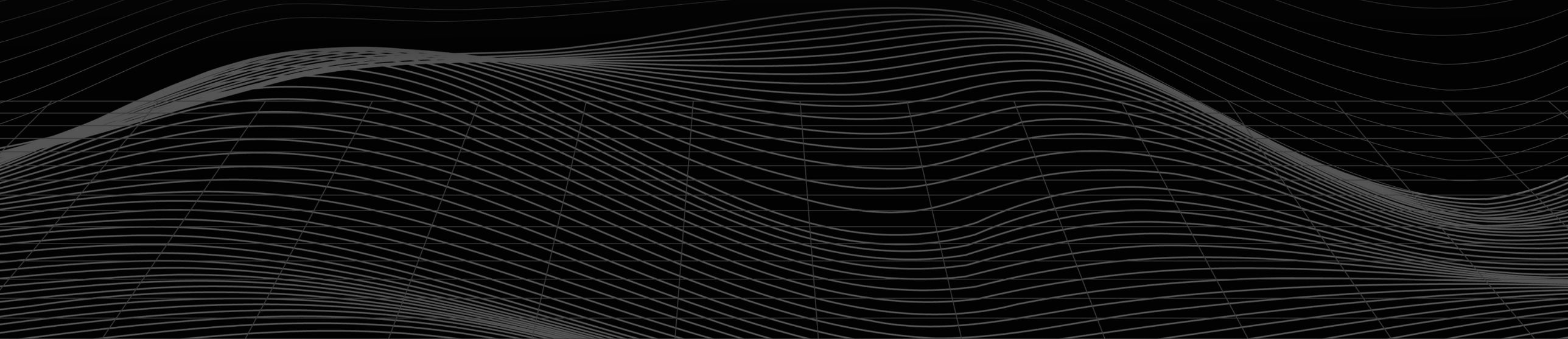
EXAMPLE

Reflect struct



EXAMPLE

Reflect struct field tags



EXAMPLE

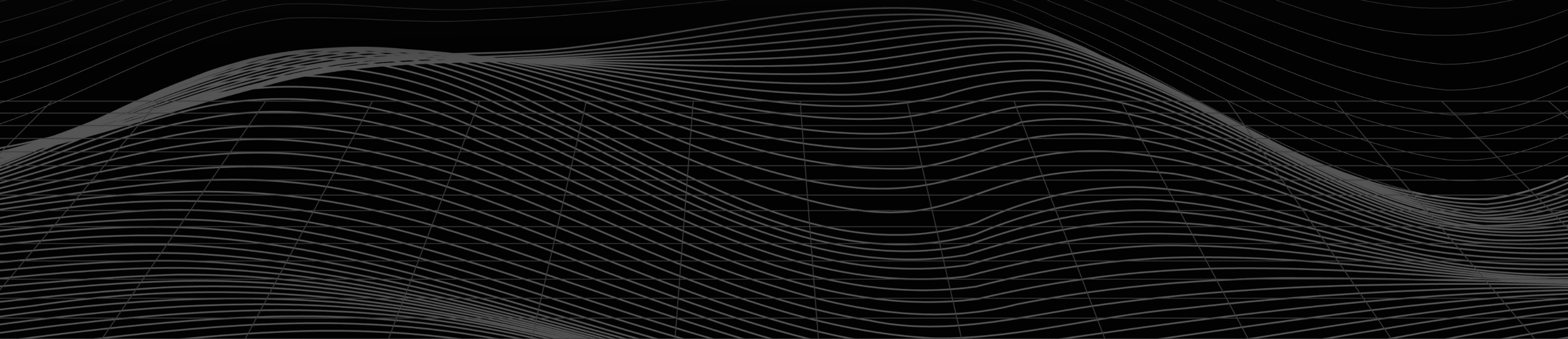
Reflect creation

EXAMPLE

Reflect function call

EXAMPLE

Reflect dynamic type



EXAMPLE

Reflect implements

EXAMPLE

Reflect map

EXAMPLE

Reflect convertible

EXAMPLE

Reflect comparable

EXAMPLE

Reflect channel

EXAMPLE

Reflect select

EXAMPLE

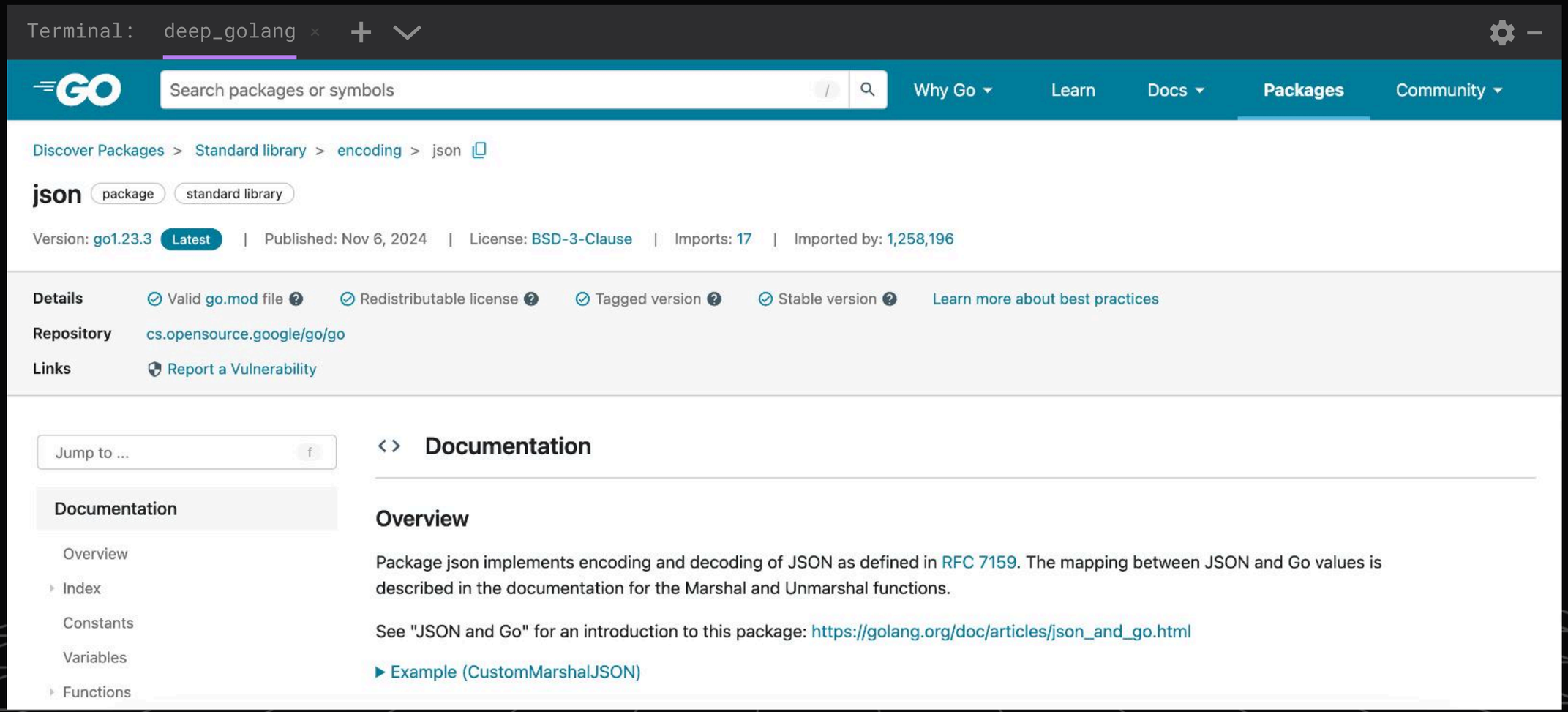
Reflect nothing

Terminal: question

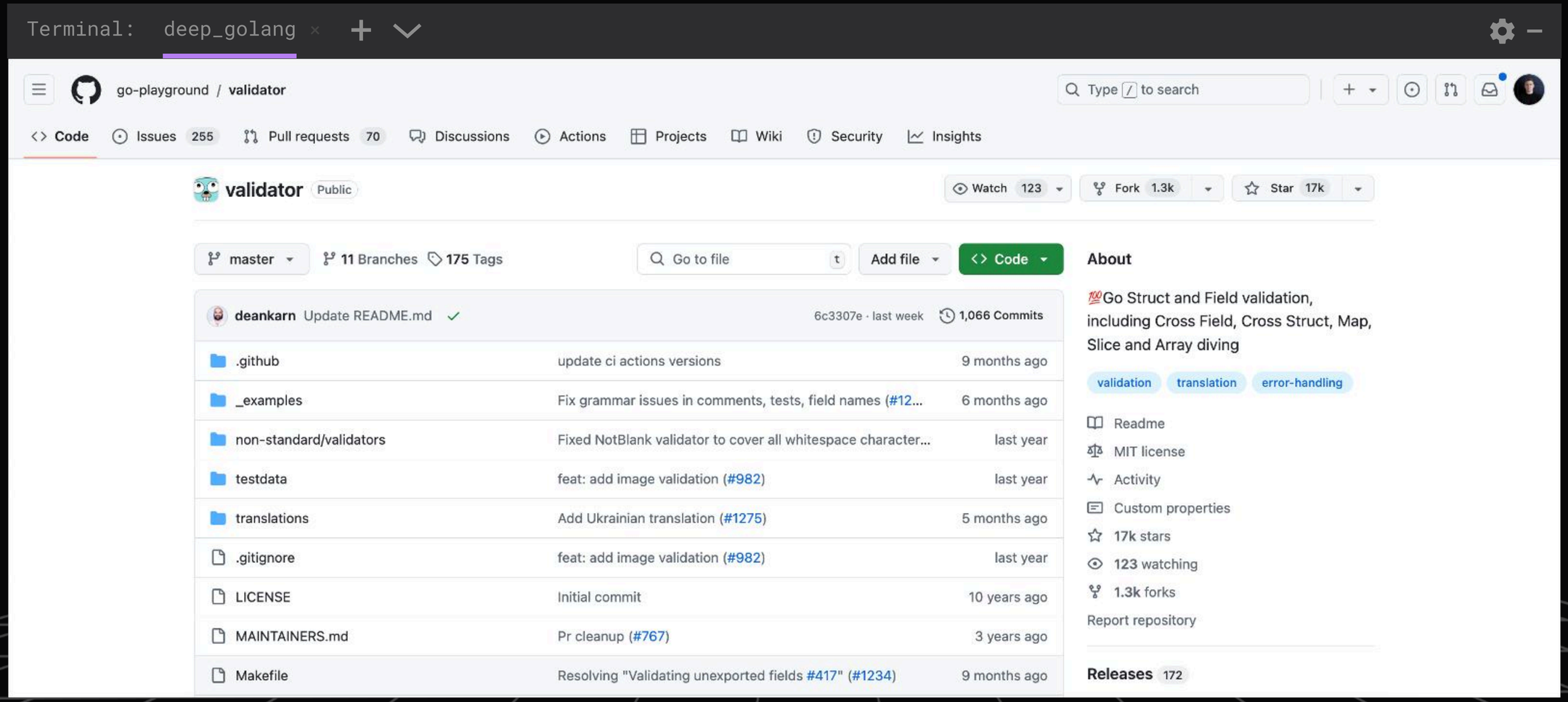
ГДЕ МОЖНО ИСПОЛЬЗОВАТЬ РЕФЛЕКСИЮ?

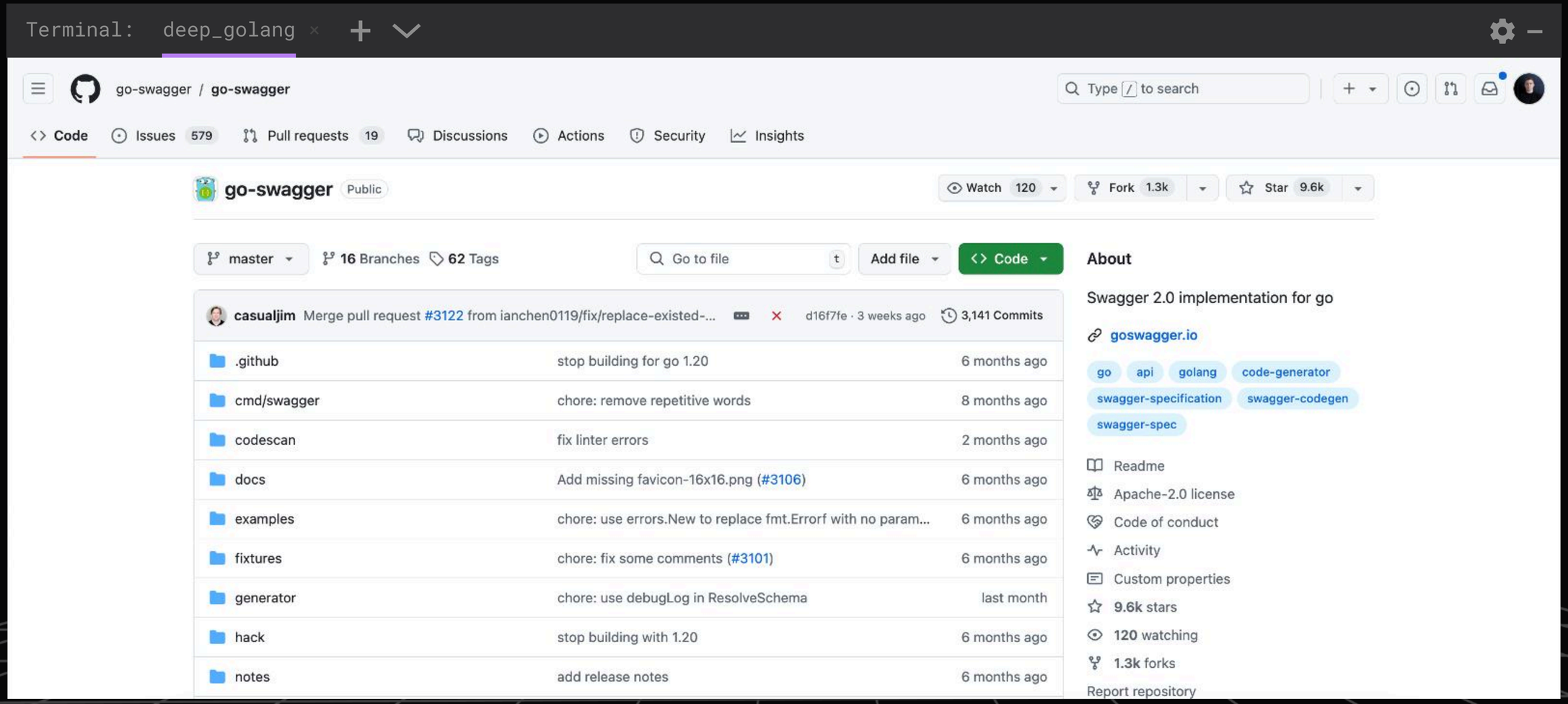
Примером, проясняющим использование рефлексии, может служить, например сериализация объекта в JSON или XML

Без рефлексии необходимо было бы явным образом указывать все имена полей класса и ссылаться на их значения для сериализации, но рефлексия позволяет программе самой определить все имеющиеся поля и получить их текстовые имена (таким образом, сериализация становится доступна для любого объекта без написания лишнего кода)


Terminal: deep_golang
Search packages or symbols
Why Go
Learn
Docs
Packages
Community
Discover Packages > Standard library > encoding > json
json
package
standard library
Version: go1.23.3 Latest | Published: Nov 6, 2024 | License: BSD-3-Clause | Imports: 17 | Imported by: 1,258,196
Details
Valid go.mod file
Redistributable license
Tagged version
Stable version
Learn more about best practices
Repository
cs.opensource.google/go/go
Links
Report a Vulnerability
Jump to ...
Documentation
Overview
Index
Constants
Variables
Functions
Documentation
Overview
Package json implements encoding and decoding of JSON as defined in RFC 7159. The mapping between JSON and Go values is described in the documentation for the Marshal and Unmarshal functions.
See "JSON and Go" for an introduction to this package: https://golang.org/doc/articles/json_and_go.html
Example (CustomMarshalJSON)

- Рефлексия может использоваться для **автоматической генерации кода**, например, для создания API-клиентов или ORM-моделей
- Рефлексия позволяет создавать **валидаторы**, которые могут проверять данные на основе аннотаций или других метаданных
- Рефлексия позволяет **создавать функции во время выполнения** – это может быть полезно для создания динамических плагинов или для реализации различных стратегий поведения


Terminal: deep_golang
go-playground / validator
Type / to search
Code
Issues 255
Pull requests 70
Discussions
Actions
Projects
Wiki
Security
Insights
validator
Public
Watch 123
Fork 1.3k
Star 17k
master
11 Branches
175 Tags
Go to file t
Add file
Code
deankarn Update README.md
6c3307e · last week
1,066 Commits
.github update ci actions versions 9 months ago
_examples Fix grammar issues in comments, tests, field names (#12... 6 months ago
non-standard/validators Fixed NotBlank validator to cover all whitespace character... last year
testdata feat: add image validation (#982) last year
translations Add Ukrainian translation (#1275) 5 months ago
.gitignore feat: add image validation (#982) last year
LICENSE Initial commit 10 years ago
MAINTAINERS.md Pr cleanup (#767) 3 years ago
Makefile Resolving "Validating unexported fields #417" (#1234) 9 months ago
About
💯Go Struct and Field validation, including Cross Field, Cross Struct, Map, Slice and Array diving
validation
translation
error-handling
Readme
MIT license
Activity
Custom properties
17k stars
123 watching
1.3k forks
Report repository
Releases 172


Terminal: deep_golang
go-swagger / go-swagger
Type / to search
Code
Issues 579
Pull requests 19
Discussions
Actions
Security
Insights
go-swagger
Public
Watch 120
Fork 1.3k
Star 9.6k
master
16 Branches
62 Tags
Go to file
t
Add file
Code
About
Swagger 2.0 implementation for go
goswagger.io
go
api
golang
code-generator
swagger-specification
swagger-codegen
swagger-spec
Readme
Apache-2.0 license
Code of conduct
Activity
Custom properties
9.6k stars
120 watching
1.3k forks
Report repository
casualjim Merge pull request #3122 from ianchen0119/fix/replace-existed-...
d16f7fe · 3 weeks ago
3,141 Commits
.github stop building for go 1.20 6 months ago
cmd/swagger chore: remove repetitive words 8 months ago
codescan fix linter errors 2 months ago
docs Add missing favicon-16x16.png (#3106) 6 months ago
examples chore: use errors.New to replace fmt.Errorf with no param... 6 months ago
fixtures chore: fix some comments (#3101) 6 months ago
generator chore: use debugLog in ResolveSchema last month
hack stop building with 1.20 6 months ago
notes add release notes 6 months ago

НЕДОСТАТКИ

- **Снижение производительности** - использование рефлексии может существенно снизить производительность программы
- **Сложность кода** - код с использованием рефлексии может быть сложнее для понимания и отладки
- **Ошибки во время выполнения** - рефлексия может привести к ошибкам во время выполнения, если программа пытается получить доступ к недоступным данным или выполнить недопустимые операции

EXAMPLE

Reflect leak

EXAMPLE

Reflect generic

EXAMPLE

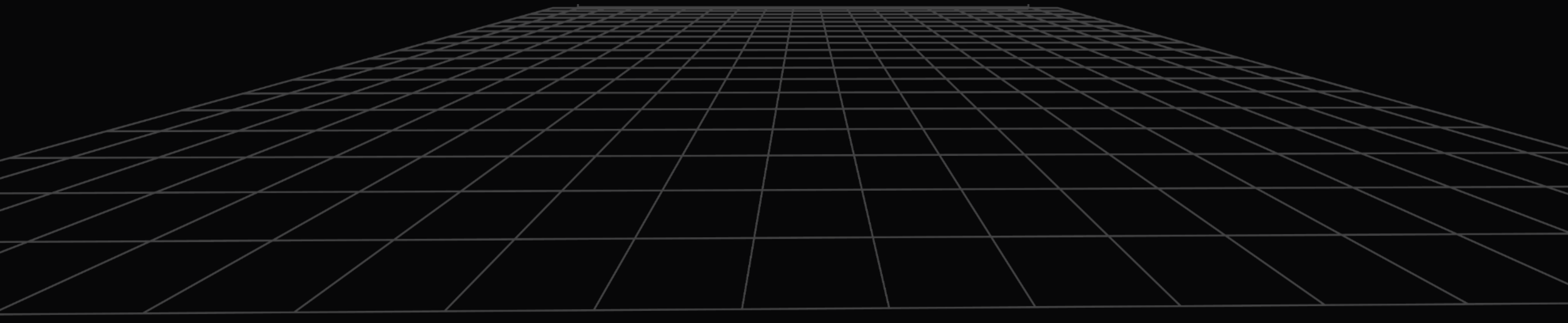
Private fields 1

EXAMPLE

Private fields 2

FAQ

Рефлексия



ПОЖАЛУЙСТА, ЗАПОЛНИ ОПРОС О ЗАНЯТИИ

Ссылка в чате и в группе участников