Опановування основами Go: Практичний посібник з освоєння мови Go

5. Функції та методи

Функції в Go можна визначити як незалежні блоки коду, які можуть бути викликані з різних місць у вашій програмі. Вони можуть приймати параметри, виконувати операції та повертати результати. У Go, функції також можуть бути "громадянами першого класу" (first-class citizens), що означає, що вони можуть бути присвоєні змінним, передані як аргументи іншим функціям, або ж використані як повертаємі значення.

Методи у Go - це функції, що специфічно прив'язані до типу. Це означає, що метод "належить" конкретній структурі чи типу і використовує контекст цього типу для виконання своїх завдань.

5.1 Визначення та Виклик Функцій

Як визначити функцію? Ну, це просто, потрібні:

- параметри,
- рецепт (тіло функції)
- і, звісно, результат (повернене значення).

Функція може не повертати значення та навіть не мати параметрів, чудовим прикладом є функція `main()` яка э точкою входу будь-якого застосунку на GO.

```
package main

func main() {
    println("Hello, World!")
}
```

Визначення Функції

```
package main
func Sum(a int, b int) int {
    return a + b
}
```

Тут ми маємо функцію `Sum`, яка приймає два числа типу `int` і повертає їх суму, також типу `int`. Просто, правда?

Параметри одного типу можуть мати тільки одне визначення типу:

```go
package main

func Sum(a, b int) int {
    return a + b
}

func main() {
    println("Hello, World!", Sum(1, 2))
}
```

Параметри Функції

Параметри це твої інгредієнти. Ти можеш їх назвати, як хочеш, але типи дуже важливі. Go строго типізована мова, тому ти не можеш просто вкинути щось невизначене у функцію і сподіватися на найкраще.

```go
package sample

import "fmt"

func Greet(name string) {
    fmt.Printf("Привіт, %s!\n", name)
}
```

Виклик Функції

Щоб викликати функцію, просто використовуй її ім'я та передай необхідні параметри:

```go
package main

import "fmt"

func Greet(name string) {
    fmt.Printf("Привіт, %s!\n", name)
}

func Sum(a, b int) int {
    return a + b
}

func main() {
    result := Sum(3, 4)
    Greet("Василь")
    fmt.Println(result)
}
```

Повернення Значення

О, так, це одна з фішок Go, яка часто приємно здивовує новачків у мові. Функції в Go можуть повертати не один, а кілька результатів одночасно. Це дозволяє тобі, наприклад, повертати результат операції разом із статусом про успіх чи помилку, що є дуже поширеним паттерном у Go.

Ось як це працює:

```
package main

import "fmt"

// SumAndEven Функція, яка повертає суму двох чисел та булеве значення, яке
показує, чи є сума парним числом
func SumAndEven(a int, b int) (int, bool) {
    sum := a + b
    isEven := sum%2 == 0
    return sum, isEven
}

// Тут ми використовуємо цю функцію
func main() {
    sum, isEven := SumAndEven(5, 3)
    fmt.Printf("Сума: %d, Чи парне число: %t\n", sum, isEven)
}
```

У цьому прикладі функція `SumAndEven` повертає два результати: суму двох чисел та булеве значення, яке каже, чи є ця сума парним числом. При виклику функції ми можемо присвоїти обидва результати змінним `sum` та `isEven`.

Ця можливість повертати кілька значень дуже корисна при обробці помилок:

```
package main

import "fmt"

func Divide(a int, b int) (int, error) {
    if b == 0 {
        return 0, errors.New("на нуль ділити не можна")
    }
    return a / b, nil
}

func main() {
    fmt.Println(Divide(5, 0))
}
```

Перевантаження Функцій

В Go немає перевантаження функцій (function overloading), як у деяких інших мовах. Тому кожна функція має мати унікальне ім'я (в рамках модуля / пакета).

Анонімні Функції та Замикання

В Go можна створювати анонімні функції (функції без імені), які можна використовувати для створення замикань (closures).

```
adder := func(a, b int) int {
    return a + b
}
```

Така функція може бути використана відразу або збережена в змінну, як у цьому прикладі.

Функції Вищого Порядку

Функції в Go можуть приймати інші функції як параметри і повертати їх, це називається функціями вищого порядку.

```
func ProcessFunction(f func(int, int) int, a int, b int) int {
    return f(a, b)
}
```

Тут `ProcessFunction` приймає іншу функцію `f` як параметр і викликає її.

Приклад використання Пакета sort з Функціями

Пакет `sort` може використовувати функції як параметри для кастомного сортування.

```
people := []Person{
    {"Андрій", 25},
    {"Богдан", 31},
    {"Віктор", 19},
}
sort.Slice(people, func(i, j int) bool {
    return people[i].Age < people[j].Age
})
```

Тут ми сортуємо слайс людей за віком, використовуючи анонімну функцію.

Go робить роботу з функціями легкою та гнучкою, дозволяючи тобі будувати потужні абстракції та ефективно управляти поведінкою програм

5.1.2 Використання '...' для Невизначеної Кількості Параметрів

У Go є дуже гнучкий спосіб роботи з функціями, який дозволяє тобі приймати невизначену кількість параметрів одного типу. Це здійснюється за допомогою еліпсиса (`...`), що ставиться перед типом параметра у визначенні функції. Такі параметри називаються "variadic parameters".

Ось як це виглядає на практиці:

```
package main

import "fmt"

// Add Функція приймає будь-яку кількість аргументів типу int
func Add(numbers ...int) int {
    total := 0
    for _, number := range numbers {
        total += number
    }
    return total
}

// Тепер ми можемо передати в Add будь-яку кількість int
func main() {
    fmt.Println(Add(1, 2))       // Виведе: 3
    fmt.Println(Add(1, 2, 3, 4)) // Виведе: 10
}
```

У цьому прикладі функція `Add` може приймати будь-яку кількість цілих чисел. Всередині функції параметр `numbers` доступний як слайс `[]int`.

Варіативні параметри часто використовуються в стандартних функціях, таких як `fmt.Printf`, яка може приймати різну кількість аргументів в залежності від форматувального рядка.

```
fmt.Printf("Це %s з використанням %d варіативних %s.\n", "приклад", 2,
"параметрів")
```

Такий підхід надає тобі фантастичну гнучкість і можливість створювати дуже адаптивні функції, які можуть обробляти різну кількість вхідних даних без зміни сигнатури функції.

5.1.3 Повернення Замикань (Closures) як Результату Функції

У Go, замикання (closures) це особливий вид анонімних функцій, які можуть захоплювати змінні з контексту, в якому вони були створені. Функції в Go можуть повертати замикання як результат. Це дозволяє не просто повернути певну функціональність, а й зберегти стан, який був у момент створення замикання.

Ось як можна створити та повернути замикання у Go:

```go
package main

import "fmt"

// Функція Incrementor повертає іншу функцію, яка використовує змінну
'start'.
func Incrementor(start int) func() int {
    // 'count' захоплюється і зберігається між викликами повернутої
функції.
    count := start
    return func() int {
        // Кожен раз, коли викликається ця функція, 'count' збільшується на
одиницю.
        count++
        return count
    }
}

func main() {
    // Створюємо замикання з початковим значенням 10.
    inc := Incrementor(10)

    // Кожен виклик inc() збільшує 'count' і повертає нове значення.
    fmt.Println(inc()) // Виведе: 11
    fmt.Println(inc()) // Виведе: 12
    fmt.Println(inc()) // Виведе: 13
}
```

У цьому коді, `Incrementor` є функцією, яка повертає іншу функцію замикання. Кожен раз, коли ви викликаєте `inc`, замикання збільшує змінну `count` на одиницю. Це дуже потужна особливість мови Go, оскільки дозволяє легко створювати генератори, ітератори, та інші конструкції, які "пам'ятають" свій стан між викликами.

Замикання можуть бути корисними, коли потрібно зробити так, щоб функція "запам'ятала" якусь інформацію зі свого оточення, і це особливо важливо для функцій, які будуть використовуватись як колбеки, обробники подій чи у конвеєрах даних.

5.1.4 Передача Посилань на Змінні як Параметрів

У Go можливість передавати змінні посиланням (через покажчики) дозволяє тобі модифікувати оригінальні дані зсередини функції, замість створення та повернення копій цих даних. Це може бути корисним для оптимізації продуктивності, особливо коли працюєш з великими даними, або коли потрібно, щоб декілька функцій мали доступ до одних і тих самих даних і могли їх змінювати.

Ось базовий приклад функції, що приймає покажчик на змінну:

```go
package main

import "fmt"
```

```
// Функція increment збільшує значення, на яке вказує покажчик 'n'.
func increment(n *int) {
    *n++ // Збільшуємо значення 'n' на 1
}

func main() {
    number := 10
    increment(&number)  // Передаємо адресу змінної 'number'
    fmt.Println(number) // Виведе: 11
}
```

У цьому коді, `increment` приймає покажчик на ціле число (`*int`). Замість копіювання значення `number` у функцію, ми передаємо адресу цієї змінної. Функція `increment` має доступ до оригінального значення і може його змінити.

Передача Структур Посиланням

Передача структур посиланням також є поширеною практикою у Go:

```
package main

import "fmt"

type MyData struct {
    A, B int
}

// Функція modifyData змінює поля структури, на яку вказує 'd'.
func modifyData(d *MyData) {
    d.A = 100
    d.B = 200
}

func newData() *MyData {
    return &MyData{1, 2}
}

func main() {
    data := newData()  // Створюємо покажчик на нову структуру
    modifyData(data)   // Передаємо покажчик на структуру 'data'
    fmt.Println(*data) // Виведе: {100 200}
}
```

У цьому прикладі, `modifyData` змінює поля `a` та `b` структури `MyData`. Знову ж таки, ми передаємо структуру посиланням, тому зміни відображаються на оригінальній змінній `data` у функції `main`.

Коли Використовувати Посилання

Передача посиланням може бути більш ефективною, але також вимагає обережності, оскільки зміни в даних відбуваються глобально. Це може призвести до більш складного управління станом, особливо в конкурентних (мультипотокових) умовах. Завжди треба враховувати потребу в оптимізації та читабельності коду, а також безпечному управлінні пам'яттю.

5.2 Методи та Приймачі

Методи у Go це функції, прив'язані до типу. Вони схожі на функції, але визначаються всередині контексту структури або типу. Це означає, що все, що ми обговорювали для функцій, також застосовне до методів: можливість приймати параметри, повертати значення, бути використаними як функції вищого порядку і так далі.

Приймачі

Коли ти визначаєш метод, ти визначаєш "приймач" (receiver) - це змінна, яка представляє екземпляр типу, для якого метод працює. Це може бути або сам тип, або покажчик на тип.

Значення vs Покажчик Приймачі

Ось основна різниця між методами з приймачами значень та приймачами покажчиків:

- **Методи з приймачем значення** отримують копію екземпляру структури. Це означає, що метод не може змінювати оригінальний екземпляр, який викликає метод.

```
type MyStruct struct {
    field int
}

// Метод з приймачем значення
func (ms MyStruct) ValueReceiverMethod() {
    ms.field = 2 // це змінить поле в копії MyStruct, але не в
оригінальному екземплярі
}
```

- **Методи з приймачем покажчика** отримують адресу екземпляру структури, що дозволяє методу змінювати структуру, з якої він був викликаний.

```
// Метод з приймачем покажчика
func (ms *MyStruct) PointerReceiverMethod() {
    ms.field = 2 // це змінить поле в оригінальному екземплярі MyStruct
}
```

> Зауваження: Лінтери сприймають як помилку одночасне використання в коді приймача за значенням та покажчиком для одного типу даних.

Коли Використовувати Який Приймач

- **Приймач значення**:
    - Використовуй, коли метод не потребує змінювати екземпляр структури.
    - Коли структура мала або копіювання є дешевшим за індиректну дереференцію покажчика.
    - Коли ти хочеш бути впевненим, що метод не змінить структуру.
- **Приймач покажчика**:
    - Використовуй, коли ти хочеш змінити екземпляр структури.
    - Коли структура велика, і копіювання її є витратним за ресурсами.
    - Щоб уникнути копіювання на кожен виклик методу, що може бути важливим для продуктивності.

Загалом, приймачі покажчиків є більш поширеними у Go, оскільки вони дозволяють методам змінювати стан екземпляра та є більш ефективними для великих структур. Однак, вибір між приймачем значення та приймачем покажчика може залежати від конкретних потреб твоєї програми.

5.2.1 Методи для Простих Типів Даних

Одна з крутих можливостей Go - це здатність визначати методи не тільки для структур, а й для власних типів даних, які базуються на простих типах, таких як `int`, `float64`, `string` тощо. Це можливо завдяки тому, що в Go можна створювати нові типи на основі існуючих.

Ось приклад того, як можна додати метод до простого типу:

```
package main

import "fmt"

type Counter int

// Increment збільшує лічильник на одиницю
func (c *Counter) Increment() {
    *c++
}

// Value повертає поточне значення лічильника
func (c Counter) Value() int {
    return int(c)
}

func main() {
    var c Counter
    c.Increment()
    fmt.Println(c.Value()) // Виведе: 1
}
```

У цьому прикладі ми створили новий тип `Counter`, який є аліасом для `int`. Потім ми визначили два методи для цього типу: `Increment`, який збільшує лічильник, і `Value`, який повертає поточне значення лічильника. Таким чином, ми можемо використовувати `Counter` так, як якщо б це була структура з методами, що дозволяє нам інкапсулювати поведінку пов'язану з лічильниками в спеціалізованому типі даних.

Така можливість робить Go особливо потужною для створення читабельного і добре структурованого коду, а також дозволяє вам визначати методи для типів, які найкраще відображають інтенції вашого коду.

5.3 Defer, Panic та Recover: Обробка Помилок в Go

Управління помилками в Go часто відрізняється від інших мов програмування. Три ключових концепції, які ви повинні знати, це `defer`, `panic` та `recover`. Вони використовуються для обробки помилок та очищення ресурсів, і вони працюють трохи інакше, ніж може бути звичним.

Defer

Оператор `defer` використовується для забезпечення виконання певної інструкції або функції наприкінці виконання функції. Зазвичай, `defer` використовується для очищення ресурсів, які потрібно звільнити незалежно від того, як завершується функція - чи то успішно, чи з помилкою.

```
package somefileop
import (
  "errors"
  "strings"
)

func readFile(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer file.Close() // Закриємо файл на виході з функції

    var line string
    // ... читаємо файл ...
    if strings.Contains(line,"\t") {
        return errors.New("tab symbols do not supported")
    }
    // ... читаємо файл ...
    return nil
}
```

Panic

`panic` - це вбудована функція, яка зупиняє звичайний потік виконання програми і починає "панічну" послідовність, зупиняючи кожну функцію, поки не буде знайдено відповідний обробник `recover`. Якщо `recover` не знайдено, програма завершиться з помилкою.

```
package main

import "fmt"

func mayPanic() {
    panic("a problem")
}

func main() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered from error:", r)
        }
    }()
    mayPanic()
    fmt.Println("After mayPanic()")
}
```

Recover

`recover` - це вбудована функція, яка дозволяє відновити контроль над програмою після `panic`. `Recover` потрібно викликати в рамках відкладеної (`defer`) функції. Коли викликана відкладена функція виконує `recover`, вона зупиняє " панічну" послідовність і повертає значення, яке було передано в `panic`.

Використання `defer`, `panic` та `recover` є потужним механізмом для управління помилками та очищення ресурсів в Go, але його слід використовувати обережно, оскільки надмірне використання може ускладнити читабельність і підтримку коду.

5.3.1 Стандартна Обробка Помилок

У Go, переважним способом обробки помилок є їх повернення з функції як частина результуючого значення. Це протистоїть використанню `panic` для неконтрольованих помилок, які не можуть бути адекватно оброблені. `Panic` використовується лише як останній засіб, коли програма не може продовжувати виконання або коли виникає ситуація, яка вимагає негайного завершення програми як, наприклад, порушення кордонів масиву.

Ось типовий шаблон обробки помилок у Go:

```
package main
import (
    "log"
    "errors"
)
func doSomething() (resultType, error) {
  // якась логіка
  if somethingWentWrong {
    return resultType{}, errors.New("descriptive error message")
  }
```

```
  return actualResult, nil
}

func main() {
  result, err := doSomething()
  if err != nil {
    // Обробляємо помилку адекватно
    log.Printf("An error occurred: %v", err)
    return
  }
// Логіка з використанням result
}
```

У цьому прикладі, функція `doSomething` повертає два значення: результат та помилку. Коли функція викликається, ви перевіряєте, чи була повернута помилка, і відповідно реагуєте на неї. Якщо помилка є, то обробляєте її і припиняєте подальше виконання. Якщо помилки немає, ви можете використовувати результат.

Цей підхід до обробки помилок забезпечує чітке управління помилками та дозволяє легко відстежити, де і коли виникає помилка, що робить код більш надійним і підтримуваним.

5.3.2 Патерн Раннього Виходу

Патерн раннього виходу (early return) в Go допомагає підтримувати код чистим і зрозумілим, зменшуючи вкладеність і спрощуючи логіку обробки помилок. Замість того, щоб мати глибоку вкладеність if-else блоків, ви перевіряєте умови та повертаєте результат (часто з помилкою) відразу ж, як тільки щось йде не так.

Ось приклад функції з раннім виходом:

```
package example

import (
    "errors"
    "os"
)

var somethingWentWrong bool

func readFile(filename string) error {
    file, err := os.Open(filename)
    if err != nil {
        return err // Ранній вихід у разі помилки
    }
    defer file.Close() // Заплануємо закриття файла на вихід з функції

    // ... робота з файлом ...
    if somethingWentWrong {
        return errors.New("something went wrong")
    }
    // ... робота з файлом ...
```

```
    return nil // Успішне завершення функції
}
```

У цьому прикладі, якщо виникає помилка при спробі відкриття файлу, функція відразу повертає помилку, не виконуючи жодних інших дій. Використання `defer` гарантує, що файл буде закритий, навіть якщо пізніше виникне помилка і функція завершиться раніше.

Defer особливо корисний у поєднанні з патерном раннього виходу, тому що він дозволяє вам легко очистити ресурси незалежно від того, на якому етапі виконання функції ви вирішили вийти. Це допомагає уникнути помилок з витоком ресурсів і забезпечує, що вся необхідна очистка виконується надійно і консистентно.

В цілому, патерн раннього виходу і використання `defer` роблять код більш лінійним і легким для читання, дозволяючи легше розуміти логіку обробки помилок і потік управління програмою.

5.4 Пакети та Імпорт / Експорт

У Go, організація коду в пакети є ключовою концепцією, яка сприяє створенню чистих та модульних програм. Пакети використовуються для групування взаємопов'язаного коду разом, що допомагає в управлінні залежностями, а також надає засоби для уникнення конфліктів імен у більших програмах.

Кожен пакет це майже завжди це окрема папка в структурі вашого проєкту. Можуть бути виключення, коли папка містить не один пакет, але майже завжди це пакет для юніт-тестів.

Імпорт Пакетів

Щоб використовувати код з іншого пакету, вам потрібно "імпортувати" його за допомогою ключового слова `import`. Go включає стандартну бібліотеку багатих пакетів, але ви також можете імпортувати свої власні пакети або пакети сторонніх.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    fmt.Println("Квадратний корінь з 16 це", math.Sqrt(16))
}
```

Експорт

Якщо ви хочете, щоб функції, методи, типи або змінні були доступними поза межами пакету, ви повинні їх "експортувати", розпочавши їх ім'я з великої літери. Це відмінність від багатьох інших мов програмування, де існують спеціальні ключові слова, такі як `public` або `export`.

```
package mypackage

// PublicFunc буде доступна зовні, оскільки починається з великої літери
func PublicFunc() {
    // ...
}

// privateFunc не буде доступна зовні, оскільки починається з маленької
літери
func privateFunc() {
    // ...
}
```

Приклад Структури Пакету

У більш складних програмах, ви можете організувати свій код у підпакети для кращої модульності та управління залежностями.

```
/myapp
    /cmd
        main.go
    /pkg
        /mypackage
            mypackage.go
    /internal
        /myinternal
            myinternal.go
    go.mod
    go.sum
```

- Каталог `cmd` зазвичай містить `main.go` файл, який є точкою входу вашої програми.
- Каталог `pkg` містить пакети, які можуть бути використані іншими програмами.
- Каталог `internal` містить пакети, які призначені для внутрішнього використання програмою та не можуть бути імпортовані іншими програмами.

Пакети - це потужний засіб для структурування коду в Go, і правильне використання імпортів/експортів допомагає забезпечити чистоту, порядок та читабельність ваших програм.5.4 Пакети та Імпорт / Експорт

> Зауважти на назву пакету в go.mod. Для пакетів - бібліотек потрібно використовувати формат: <репозітарій>/<власник>/<назва пакету> Тільки в цьому випадку Ваш пакет буде доступний для менеджеру пакетів. Пакет - бібліотеку краще розташувати в середені GOPATH для сумісності (але не обов'язково для сучасних версій).

> Файли які мають суфікс "*_test.go" не будуть экспортовані або використані прі компіляції коду Каталоги з назвою testdata не будуть экспортовані або використані прі компіляції коду

5.4.1 Експорт Структур, Констант, Глобальних Змінних Пакетів

Експорт в Go це механізм визначення видимості іменованих сутностей, таких як структури, константи, змінні та інтерфейси, за межами визначення їх пакетів. Це ключова частина управління залежностями в Go і сприяє створенню добре організованих, легко інтегрованих модульних пакетів.

Експорт Структур

Щоб експортувати структуру, її ім'я має починатися з великої літери. Тільки експортовані поля (з великою першою літерою) будуть видимі поза пакетом.

```
package data

// User  експортована структура з експортованими полями
type User struct {
    Name  string
    Email string
    level string // Приватне поле (доступне тільки всередині пакету)
}

// admin  неекспортована структура з неекспортованими полями
type admin struct {
    level string
}
```

> Зауважте, що при використанні серіалізації (JSON або іншого типу) приватні поля не будуть серіалізовані.

Експорт Констант та Змінних

Те ж саме правило застосовується до констант і глобальних змінних. Експортовані константи та змінні мають назви, які починаються з великої літери.

```
package some

// Pi  експортована константа
const Pi = 3.14159

// debugMode  неекспортована константа
const debugMode = false
```

Експорт Функцій

Функції також слідують цьому правилу; методи, що мають імена з великої літери, експортуються і доступні для інших пакетів.

```
package somefunc

import "fmt"
```

```
// SayHello  експортована функція
func SayHello(name string) {
    fmt.Printf("Hello, %s!\n", name)
}

// whisper  неекспортована функція
func whisper(secret string) {
    fmt.Printf("quietly saying: %s\n", secret)
}
```

Експорт Через Пакети

Коли ви створюєте програми або бібліотеки, вам може знадобитися організувати код у підпакети. Важливо пам'ятати, що кожен під-пакет має свій власний простір імен, тому правила експорту застосовуються незалежно в кожному під-пакеті.

> Будьте уважними, щоб запобігти "циклічному імпорту". Це може статися при використанні субпакетом один одного навіть "крізь декілька" пакетів.

Використання Internal Пакетів

Іноді вам може знадобитися обмежити доступ до певних частин вашого коду, навіть якщо вони теоретично експортовані. Go дозволяє це зробити за допомогою `internal` пакетів. Якщо пакет знаходиться в піддиректорії з назвою `internal`, його експортовані ідентифікатори будуть доступні тільки тим пакетам, які знаходяться в тому ж каталозі, що й `internal` пакет (або його підкаталоги).

Використання `internal` пакетів є потужним способом контролю над тим, які частини вашого коду можуть бути використані іншими розробниками, навіть у рамках тієї ж програми чи бібліотеки.

У Go піддиректорія `testdata` має особливе значення і використовується для зберігання даних, які можуть бути використані під час тестування. Це конвенція, прийнята в екосистемі Go, і вона має кілька переваг:

1. **Ізоляція тестових даних:** Поміщення тестових файлів у `testdata` допомагає тримати їх організовано та чітко відокремленими від виробничого коду.
2. **Ігнорування при збірці:** Go інструментарій автоматично ігнорує вміст директорії `testdata` під час збірки пакету. Це означає, що файли і директорії, розміщені в `testdata`, не будуть впливати на збірку вашої програми або бібліотеки.
3. **Легкість доступу в тестах:** Файли з `testdata` можна легко читати в тестах, використовуючи відносні шляхи. Це робить тести більш читабельними і легшими для написання, оскільки вам не потрібно використовувати складні шляхи або налаштування конфігурацій для доступу до тестових даних.
4. **Використання в стандартних інструментах:** Багато стандартних інструментів Go, як-от `go test`, мають вбудовану підтримку розуміння і використання директорії `testdata`.

Ось приклад структури проєкту, де тестові дані організовані в `testdata`:

```
/myapp
    /cmd
        main.go
    /pkg
        /mypackage
            mypackage.go
    /internal
        /myinternal
            myinternal.go
            /testdata
                testfile1.txt
                testfile2.txt
                /subdir
                    testfile3.txt
```

У тестах ви можете використовувати ці файли, читаючи їх із шляхом, який починається з `testdata/`, наприклад:

```go
package myinternal

import (
    "os"
    "testing"
)

func TestReadData(t *testing.T) {
    data, err := os.ReadFile("testdata/testfile1.txt")
    if err != nil {
        t.Fatal(err)
        t.Log(data)
    }
    // ... тестування з використанням даних ...
}
```

Цей підхід допомагає зберегти тести консистентними та легко відтворюваними, незалежно від середовища, в якому вони виконуються.