Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Програмування вебзастосунків

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітньою програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2025

УДК 004.43
П78

Укладачі: *Недашківський Олексій Леонідович*, д-р техн. наук, доц.
Гусєва Ірина Ігорівна, канд. екон. наук, доц.
Голець Владислав Олександрович
Гнатишин Михайло Степанович
Пироговська Тетяна Володимирівна
Передера Віктор Романович

Рецензенти *Жураковський Б.Ю.,* д-р техн. наук, проф., професор кафедри інформаційних систем та технологій Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського»

Відповідальний редактор *Стативка Ю. І.,* канд. техн. наук, доцент

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 9 від 26.06.2025 р.)
за поданням вченої ради Навчально-наукового інституту атомної та теплової енергетики
(протокол № 12 від 03.06.2025 р.)

Програмування вебзастосунків. [Електронний ресурс] : навч. посіб. для здобувачів
П78 ступеня бакалавра за освіт. програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці» спец. 121 Інженерія програмного забезпечення / КПІ ім. Ігоря Сікорського ; уклад.: О. Л. Недашківський та ін. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2025. – 335 с.

Навчальний посібник призначений для здобувачів ступеня бакалавр за спеціальністю 121 Інженерія програмного забезпечення освітньої програми «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці» Національного технічного університету України «Київського політехнічного інституту імені Ігоря Сікорського».
У посібнику приділено увагу вивченню мови Go для програмування вебзастосунків. Навчальний посібник буде також корисний тим, хто цікавиться розробкою програмного забезпечення мовою програмування Go та навчається за іншими спеціальностями галузі знань 12 Інформаційні технології.

УДК 004.43

Реєстр. № НП 24/25-594. Обсяг 9,5 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

ЗМІСТ

ВСТУП

Навчальний посібник «Програмування вебзастосунків» присвячено мові Go для програмування вебзастосунків.

Мова Go є компільованою статично типізованою мовою програмування від компанії Google. Мова Go призначена для створення різноманітних застосунків, але насамперед це вебсервіси та клієнт-серверні програми. Ця мова також має можливості для роботи з графікою, низькорівневими можливостями тощо.

Обсяг матеріалу становить необхідний мінімум в підготовці слухачів за освітньою програмою «Інженерія програмного забезпечення кіберфізичних систем в енергетиці» спеціальності 121 Інженерія програмного забезпечення при вивченні дисципліни: «Програмування вебзастосунків».

В навчальному посібнику подано загальний та практичний матеріал, опис програмних засобів, наведені рекомендації для поглибленого вивчення як самої мови програмування Go так і інструментальних засобів розробки та тестування.

Основним завдання навчального посібника, згідно з вимогами програми зазначеної навчальної дисципліни, є формування у студентів після засвоєння кредитного модуля таких знань та умінь.

Знання:

- з розробки архітектури, модулів та компонентів програмних систем;
- зі створення інтерактивних, компактних вебзастосунків та вебсистем;
- методичних основ та технологій створення інформаційних систем та мережевого програмного забезпечення.

Уміння:

- мотивовано обирати мови програмування та технології розробки для розв’язання завдань створення і супроводження програмного забезпечення;
- створювати інтерактивні, компактні вебзастосунки та вебсистеми, володіти методичними основами та технологіями створення інформаційних систем та мережевого програмного забезпечення з врахуванням специфіки предметної області енергетичної галузі;
- вміти створювати програмне забезпечення для інтелектуальних кібер-фізичних систем, в тому числі з врахуванням специфіки предметної області енергетичної галузі.

1. НАЛАШТУВАННЯ ЗАСОБІВ РОЗРОБКИ МОВОЮ GO ТА ПОЧАТОК РОБОТИ

1.1. Коротка довідка про Go

Go є компільованою статично типізованою мовою програмування від компанії Google. Мова Go призначена для створення різноманітних застосунків, але насамперед це вебсервіси та клієнт-серверні програми. Ця мова також має можливості для роботи з графікою, низькорівневими можливостями тощо.

Робота над мовою Go розпочалася у 2007 у надрах компанії Google. Одним із авторів є Кен Томпсон, який, до речі, є й одним із авторів мови Сі поряд із Денисом Рітчі. Мова Go була анонсована 10 листопада 2009 року, а в березні 2012 року вийшла версія 1.0. При цьому мова продовжує розвиватись. Поточною версією на момент підготовки цього навального посібника є версія 1.22, яка вийшла у лютому 2024 року.

Мова Go розвивається як проєкт з відкритим вихідним кодом і всі його програми та компілятор можна знайти та використовувати безкоштовно. Офіційний сайт проєкту розташовано за посиланням **https://go.dev/** [1], де можна знайти багато корисної інформації про мову.

Go є кросплатформною мовою, вона дозволяє створювати програми під різні операційні системи – Windows, Mac OS, Linux, FreeBSD. Код має переносимість: програми, написані для однієї з цих операційних систем, можуть легко після перекомпіляції бути перенесені на іншу ОС.

Основні особливості мови Go є:

- компільована – компілятор транслює програму на Go у машинний код, зрозумілий для певної платформи;
- статично типізована;
- присутній збирач сміття, який автоматично очищує пам'ять;
- підтримка роботи з мережевими протоколами;
- підтримка багатопоточності та паралельного програмування.

Сьогодні Go знаходить широке застосування у різних сферах. Зокрема, серед відомих проєктів, які застосовують мову Go, можна знайти такі як Google, Dropbox, Netflix, Kubernetes, Docker, Twitch, Uber, CloudFlare та інші.

Для початку роботи з Go необхідний текстовий редактор для набору коду і компілятор для перетворення коду у виконуваний файл. Також можна використовувати спеціальні інтегровані середовища розробки (IDE), які підтримують Go, наприклад GoLand від компанії JetBrains. Існують плагіни для Go для інших IDE, зокрема IntelliJ IDEA і Netbeans.

1.1.1. Установка Go

Пакет для встановлення компілятора можна завантажити на офіційному сайті за посиланням **https://go.dev/dl/** [2], зовнішній вигляд якого наведено на рисунку 1.1.

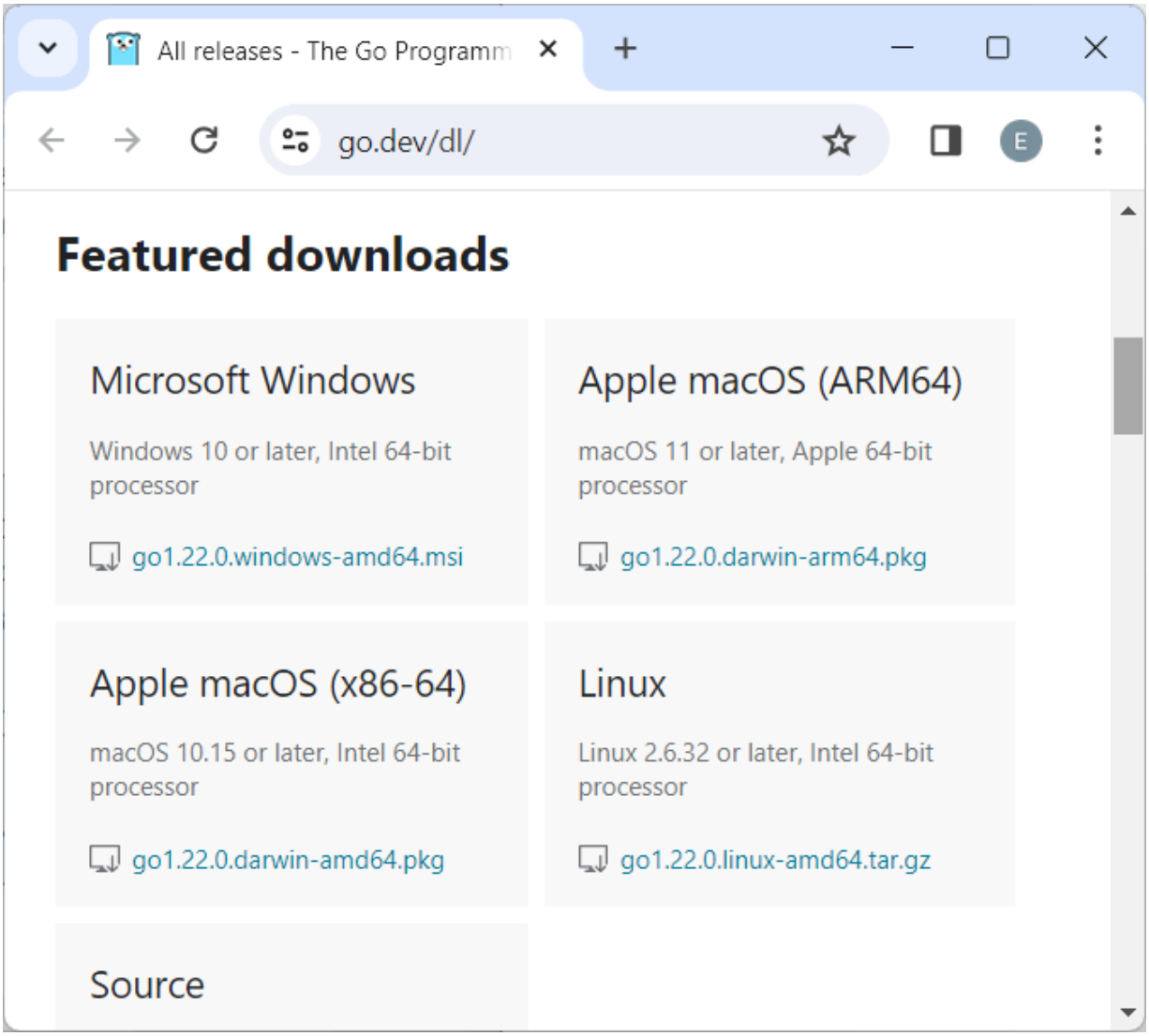

Рисунок 1.1. Зовнішній вигляд офіційної сторінки завантаження пакета компілятора для мови Go

За цією адресою розташовуються пакети інсталяторів для різних операційних систем. Зверніть увагу на версії систем, що підтримуються. Так, на момент написання поточної статті підтримувалися лише версії Windows 10 і вище, MacOS 10.15 і вище та Linux 2.6.32 та вище, але всі версії мають бути 64-розрядними. Завантажимо відповідний для нашої ОС пакет установника та запустимо його. Процес встановлення відносно простий, треба лише пройтись послідовно по всіх вікнах майстра інсталяції.

1.1.1.1. Встановлення на Windows

Приклад головного вікна інсталятора для Windows наведено на рисунку 1.2.

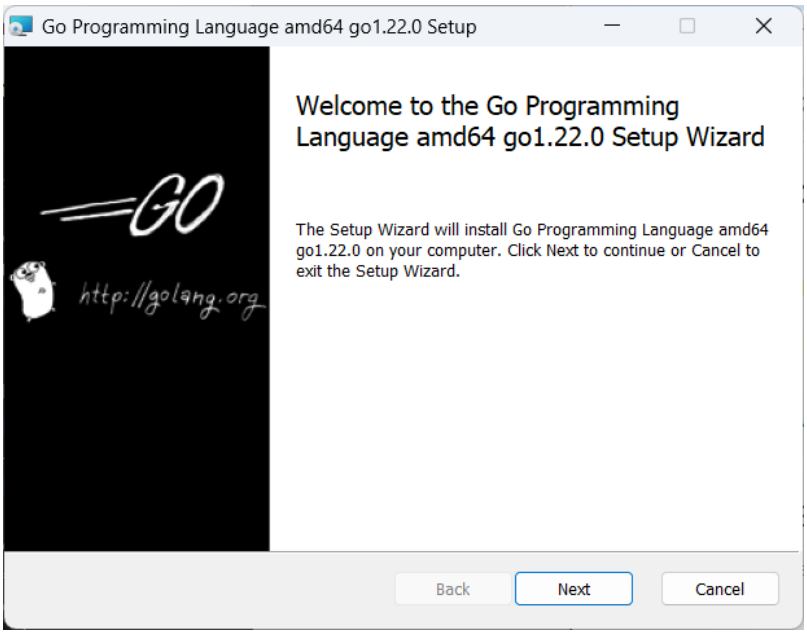

Рисунок 1.2. Головне вікно інсталятора для Windows

Після прийняття ліцензійної угоди з'явиться вікно для вибору місця встановлення, як показано на рисунку 1.3.

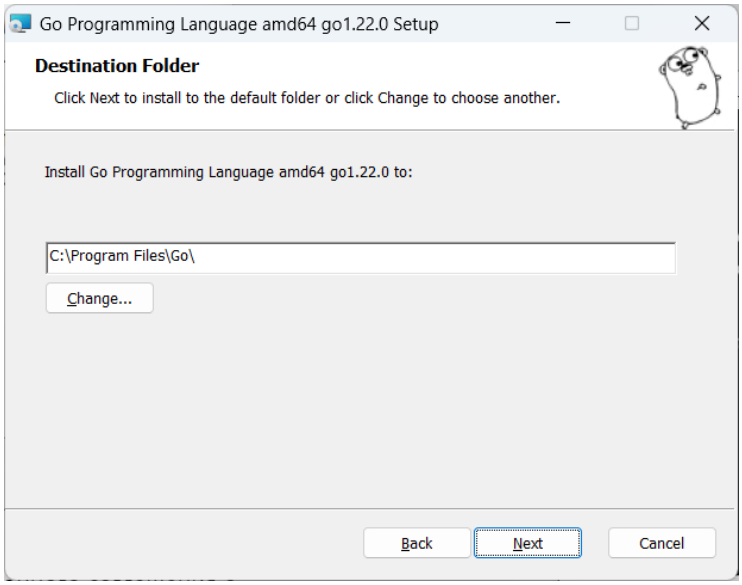

Рисунок 1.3. Вікно вибору місця для встановлення

На Windows, наприклад, за замовчуванням використовується шлях "**C:\Program Files\Go**". Залишимо шлях за замовчуванням і перейдемо до наступного вікна, на якому натиснемо кнопку **Install**, як показано на рисунку 1.4.

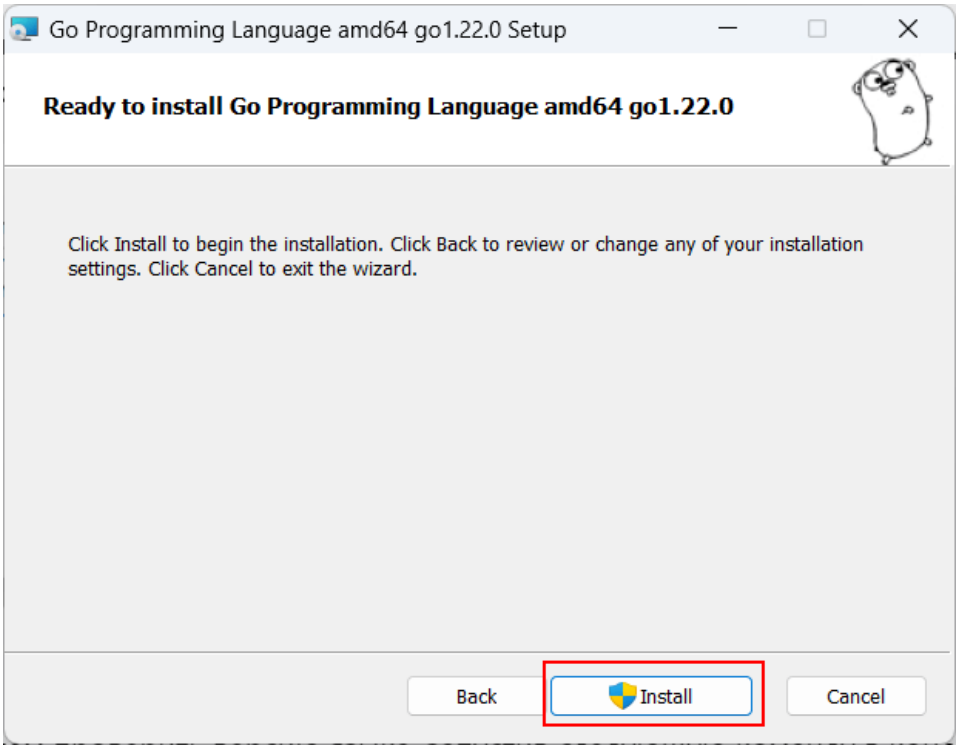

Рисунок 1.4. Вікно запуску інсталяції

Після успішної інсталяції в теці установки будуть встановлені всі файли, необхідні для роботи з Go. Зокрема, у підкаталозі **bin** можна знайти файл **go** (**go.exe** на Windows), який виконує роль компілятора, як показано на рисунку 1.5.

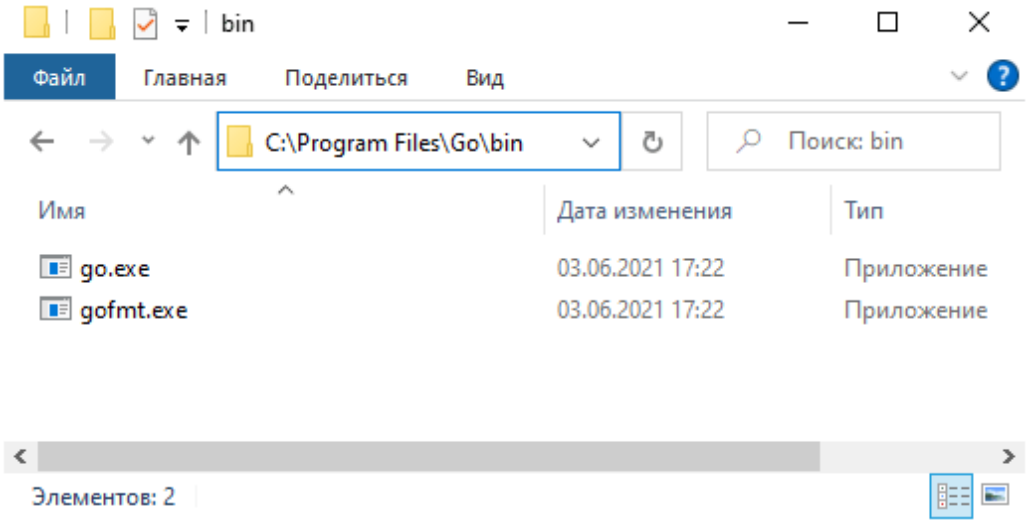

Рисунок 1.5. Вигляд теки зі встановленими компонентами компілятора

1.1.1.2. Встановлення на MacOS

Установка на MacOS виконується за допомогою майстра установки, як показано на рисунку 1.6.

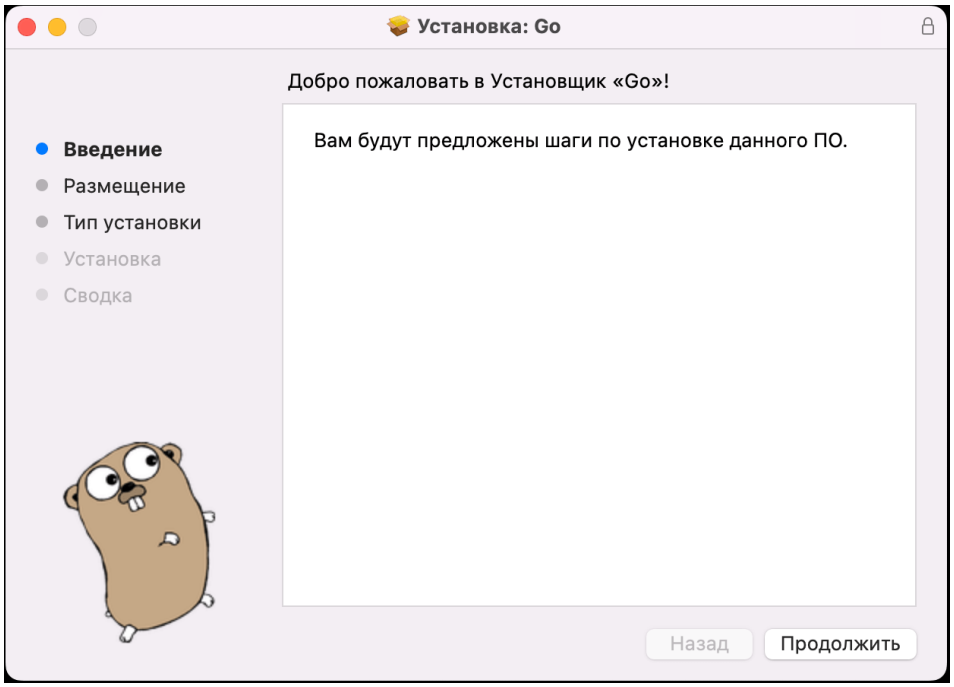

Рисунок 1.6. Вигляд вікна майстра установки для MacOS

Зовнішній вигляд вікна стандартної установка на MacOS показано на рисунку 1.7.

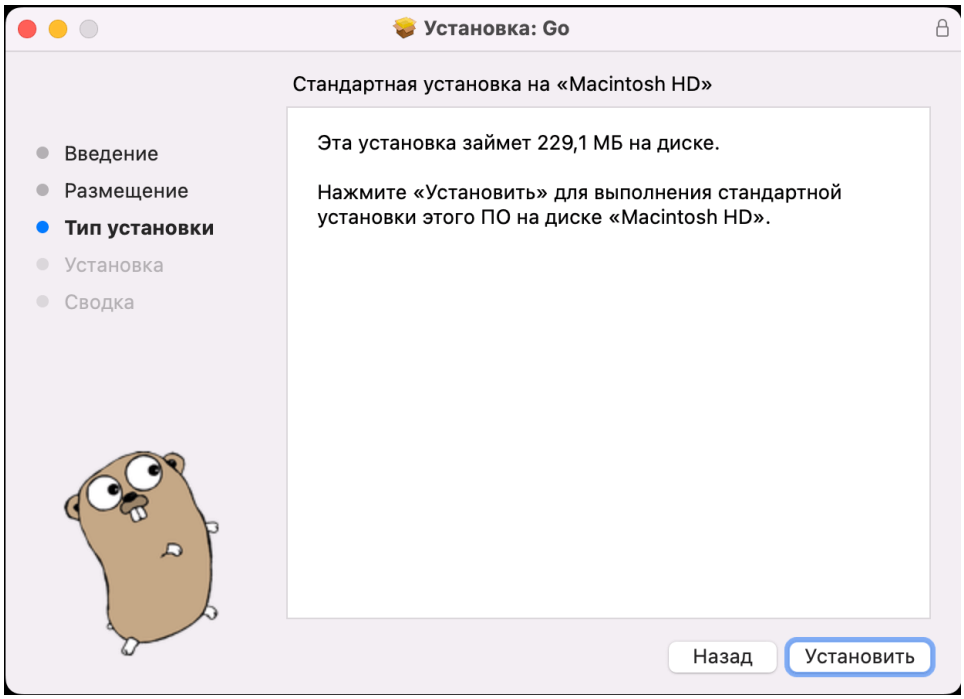

Рисунок 1.7. Зовнішній вигляд вікна стандартної установка на MacOS

1.1.1.3. Встановлення на Linux

Для Linux офіційний сайт надає архів. Для прикладу використаємо файл **go1.22.0.linux-amd64.tar.gz**. Для встановлення цього архіву виконаємо команду, як показано на рисунку 1.8.

```
sudo rm -rf /usr/local/go && tar -C /usr/local -xzf go1.22.0.linux-amd64.tar.gz
```

Рисунок 1.8. Зовнішній вигляд вікна введення команди

Ця команда видаляє раніше встановлену версію, якщо така є, і встановлює нову в теку **/usr/local/go**, відповідно компілятор розташовуватиметься в теці **/usr/local/go/bin**

Після цього шлях до теки **/usr/local/go/bin** необхідно записати в змінні середовища. Для цього додамо до кінця файлу **$HOME/.profile або /etc/profile** рядок, як показано на рисунку 1.9.

```
1 export PATH=$PATH:/usr/local/go/bin
```

Рисунок 1.9. Зовнішній вигляд вікна введення команди

І для негайного її застосування виконаємо команду, як показано на рисунку 1.10.

```
source $HOME/.profile
```

Рисунок 1.10. Зовнішній вигляд вікна введення команди

1.1.1.4. Перевірка установки Go

Після інсталяції можна перевірити встановлену версію мови Go, запустивши в консолі команду **go version**, як показано на рисунку 1.11.

```
C:\Users\udjen>go version
go version go1.22.0 windows/amd64

C:\Users\udjen>
```

Рисунок 1.11. Зовнішній вигляд вікна введення команди

1.2. Перша програма

Створимо першу програму мовою Go. Для написання коду нам знадобиться текстовий редактор. Можна взяти будь-який редактор, наприклад, вбудований блокнот чи популярний Notepad++ чи будь-який інший. Для трансляції вихідного коду у застосунок потрібен компілятор.

1.2.1. Створення програми

Визначимо на жорсткому диску теку для зберігання файлів із вихідним кодом. Нехай це буде тека **C:\golang**. У цій теці створимо новий текстовий файл, який назвемо **hello.go**, як показано на рисунку 1.12.

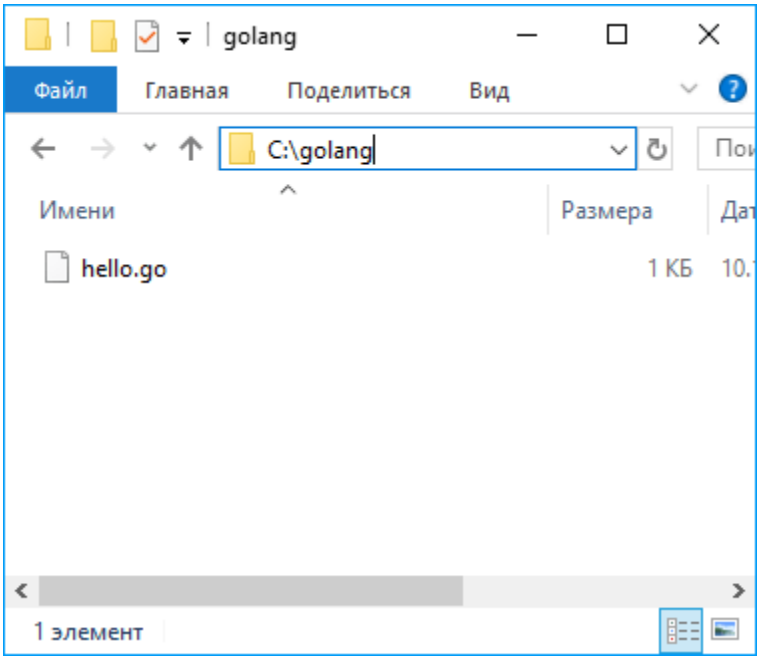

Рисунок 1.12. Вікно створення нового текстового файлу

Відкриємо цей файл у будь-якому текстовому редакторі та визначимо в ньому наступний код:

```
1   package main
2   import "fmt"
3
4   func main() {
5       fmt.Println("Hello Go!")
6   }
```

Програма мовою Go визначається як набір пакетів. Програмний код повинен бути визначений у певному пакеті. Тому на початку файлу за допомогою оператора package вказується, до якого пакета належатиме файл. В наведеному прикладі це пакет **main**:

```
1   package main
```

Причому пакет повинен називатися саме **main**, оскільки саме цей пакет визначає файл, що виконується.

При складанні програмного коду може знадобитися функціонал інших пакетів. У мові Go є багато вбудованих пакетів, які містять код, що виконує певні дії. Наприклад, у наведеному прикладі виводиться повідомлення на

консоль. І для цього було використано функцію **Println**, яка визначена в пакеті **fmt**. Тому у другому рядку за допомогою директиви **import** було підключено сам цей пакет наступним чином:

```
2   import "fmt"
```

Далі йде записано функцію **main**. Це головна функція будь-якої програми на мові Go. Насправді все, що виконується у програмі, виконується саме функцією **main**.

Визначення функції починається зі слова **func**, після якого вказується назва функції, тобто **main**. Після назви функції у дужках вказується перелік параметрів. Оскільки функція **main** не приймає жодних параметрів, то в даному випадку вказуються порожні дужки.

Потім у фігурних дужках визначається тіло функції **main**, тобто ті дії, які і виконує функція:

```
4   func main() {
```

В наведеному прикладі функція **main** виводить на консоль рядок "**Hello Go!**". Для цього застосовується функція **Println()**, яка визначена у пакеті **fmt**. Тому при виклику функції спочатку вказується ім'я пакета та через крапку ім'я функції. В дужках функції передається повідомлення, яке вона повинна вивести на консоль, наприклад:

```
5   fmt.Println("Hello Go!")
```

1.2.2. Компіляція та виконання програми

Тепер скомпілюємо та виконаємо цю програму. Для цього необхідно передати файл із вихідним кодом компілятору **go.exe** та вказати потрібну команду. Для цього необхідно відкрити командний рядок, або термінал, і перейти в ньому за допомогою команди **cd** до теки, де зберігається файл з вихідним кодом **hello.go**, наприклад це тека **C:\golang**, як показано на рисунку 1.13.

```
cd C:\golang
```

Рисунок 1.13. Зовнішній вигляд вікна введення команди

Потім запустимо програму на виконання за допомогою команди, як показано на рисунку 1.14.

```
go run hello.go
```

Рисунок 1.14. Зовнішній вигляд вікна введення команди

В наведеній конструкції **go** позначає компілятор. Оскільки при установці шлях до компілятора автоматично прописується в змінній **PATH** у змінних оточення, то не потрібно вказувати повний шлях **C:\Go\bin\go.exe**, а достатньо вказати просто ім'я програми **go**. Далі записується параметр **run**, який вказує на необхідність виконання програми. І насамкінець вказується власне файл програми **hello.go**.

У результаті після виконання програми на консоль буде виведено повідомлення " **Hello Go!** ", як показано на рисунку1.15.

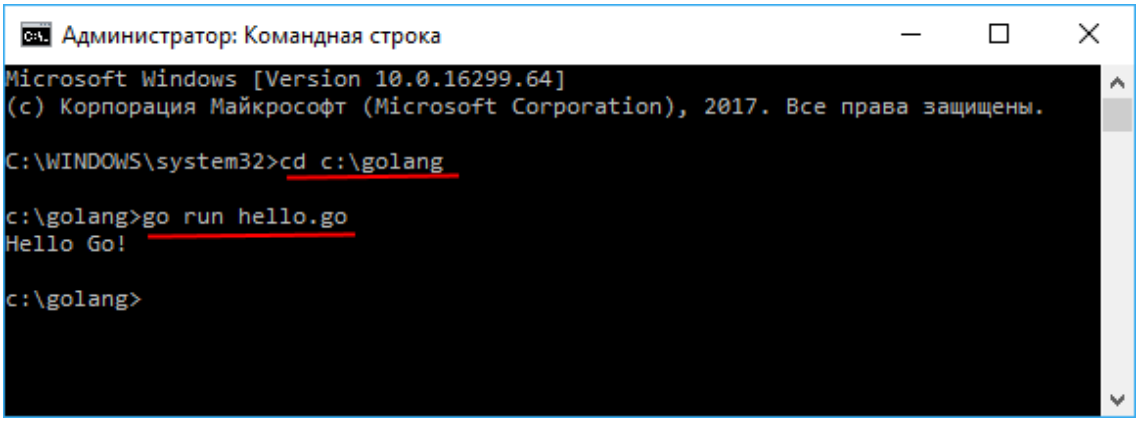

Рисунок 1.15. Результат роботи програми

Наведена команда запускає на виконання програму, але не компілює програму в окремий файл. Для компіляції необхідно виконати іншу команду, як показано на рисунку 1.16.

```
go build hello.go
```

Рисунок 1.16. Зовнішній вигляд вікна введення команди

Після виконання цієї команди у теці з вихідним файлом з'явиться ще один файл, який називатиметься **hello.exe** і який вже можна запускати, як звичайний виконуваний файл. Після цього можна виконати програму, запустивши в консолі цей файл, як показано на рисунку 1.17.

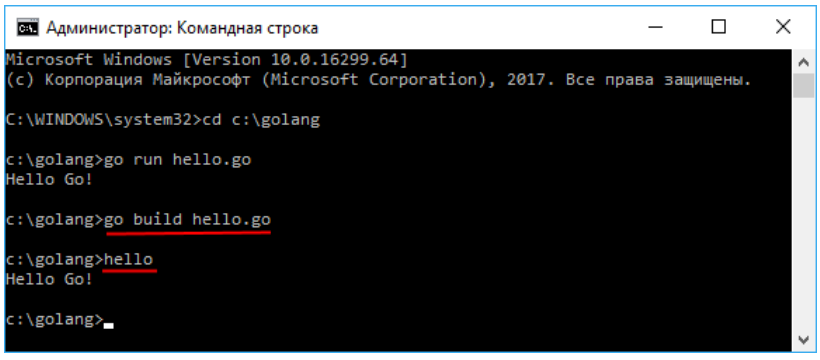

Рисунок 1.17. Результат роботи програми

1.3. Go в LiteIDE

Використання інтегрованих середовищ розробки (IDE) часом спрощує управління проєктом і створення програми. Для мови Go одним з найпопулярніших середовищ розробки є LiteIDE. Це безкоштовне кросплатформне середовище, яке можна вільно завантажити на робочий комп'ютер. Офіційний сайт IDE LiteIDE можна знайти за посиланням **https://code.google.com/archive/p/liteide/** [3].

Безпосередньо завантажити всі файли цієї IDE можна за посиланням **https://github.com/visualfc/liteide/** [4], як показано на рисунку 1.18.

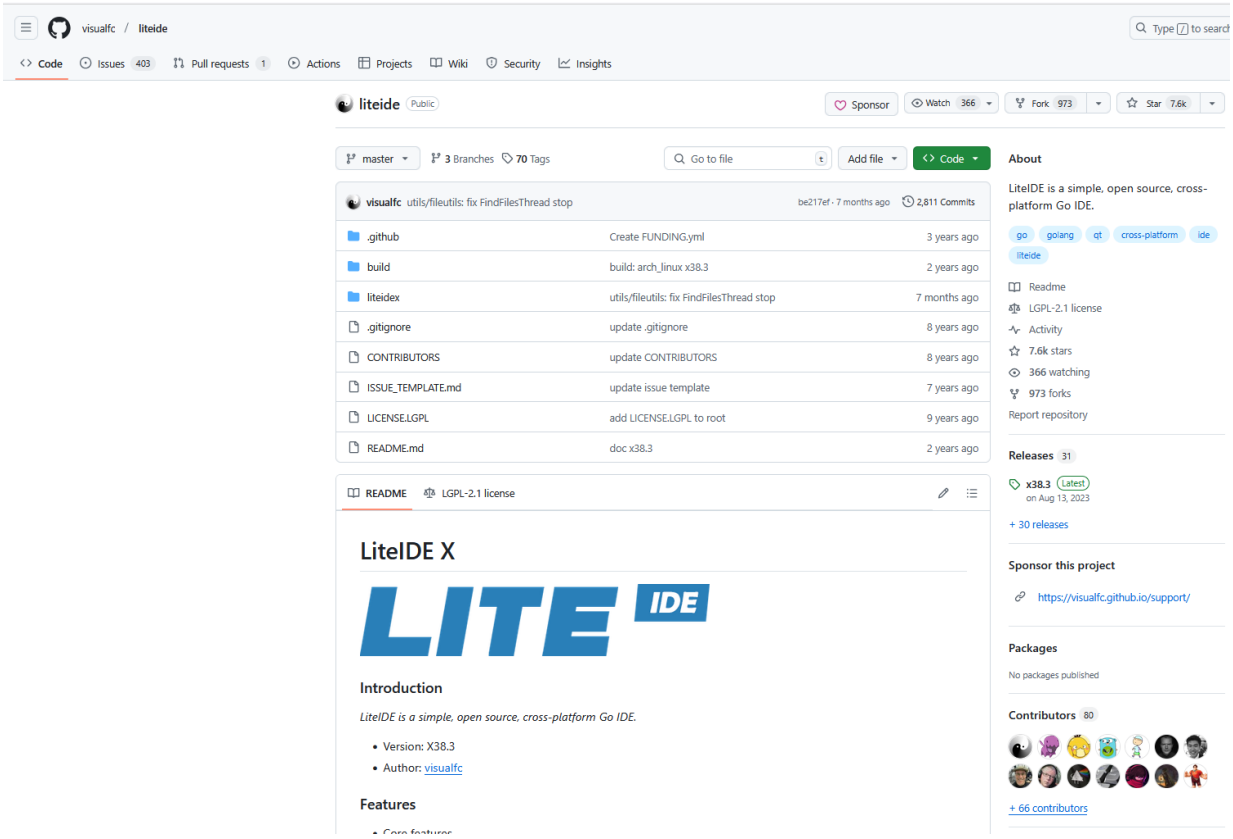

Рисунок 1.18. Зовнішній вигляд офіційної сторінки завантаження пакета LiteIDE

На цій сторінці можна знайти посилання для завантаження. Причому нічого не потрібно встановлювати. За посиланням завантажується zip-архів, який після завантаження достатньо розпакувати. Після розпакування архіву в кореневій теці в каталозі **bin** можна знайти файл виконуваної програми **liteide.exe** (на Windows), через який можна запустити середовище розробки, як показано на рисунку 1.19.

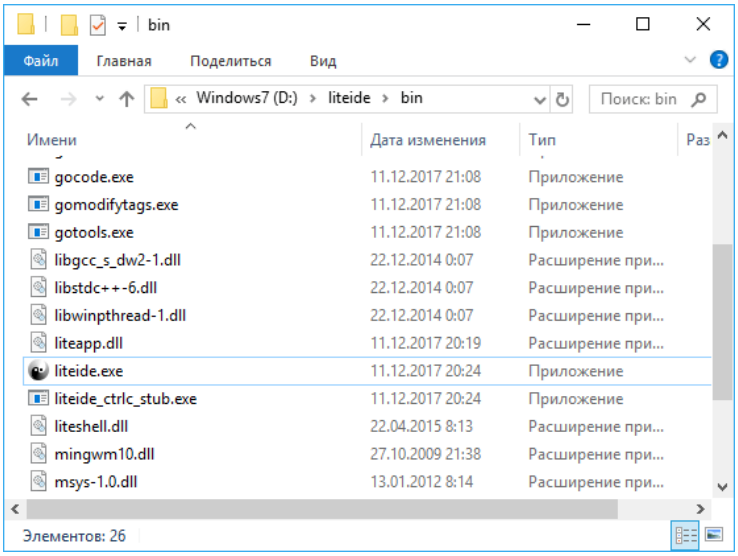

Рисунок 1.19. Склад завантаженого пакета LiteIDE

Запустимо цей файл, і за замовчуванням відриється стартова сторінка. LiteIDE має локалізовані версії, і через параметри можна переключати мову середовища на будь-яку із запропонованих.

Для створення нового проєкту перейдемо до меню **File(Файл) -> New (Створити)**, як показано на рисунку 1.20.

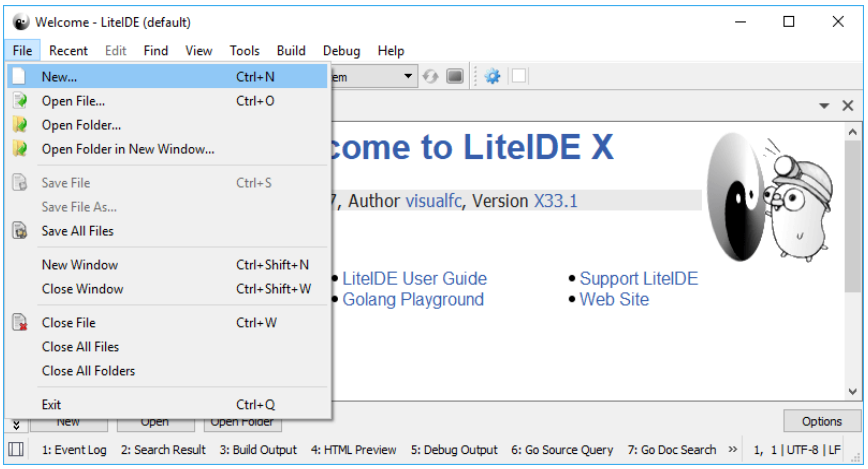

Рисунок 1.20. Вікно створення нового проєкту

Після цього з'явиться вікно вибору шаблону проєкту, як показано на рисунку 1.21.

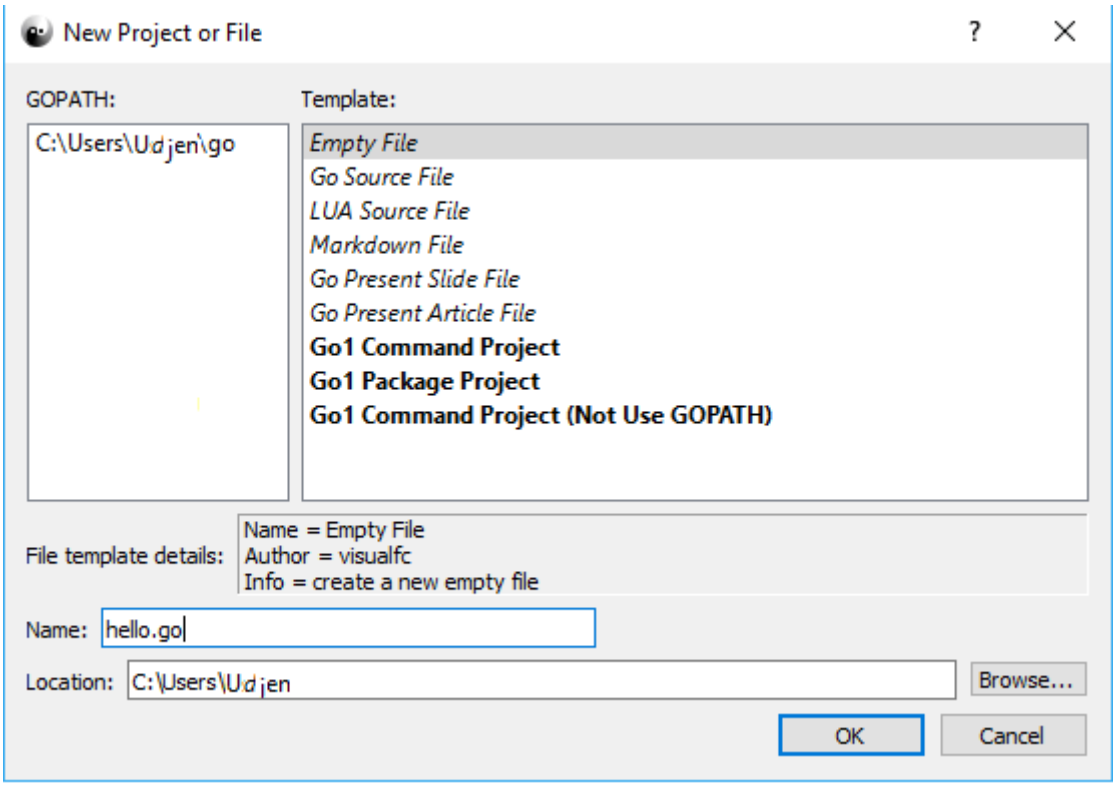

Рисунок 1.21. Вікно вибору шаблону проєкту

Як шаблон виберемо **Empty File**. А знизу в полі **Name** вкажемо для файлу ім'я **hello.go**. В іншому полі можна побачити шлях до теки, де розташовуватиметься файл. Цей шлях також можна змінити. Після цього буде створено проєкт і відкриється вікно, де необхідно буде підтвердити відкриття проєкту, як показано на рисунку 1.22.

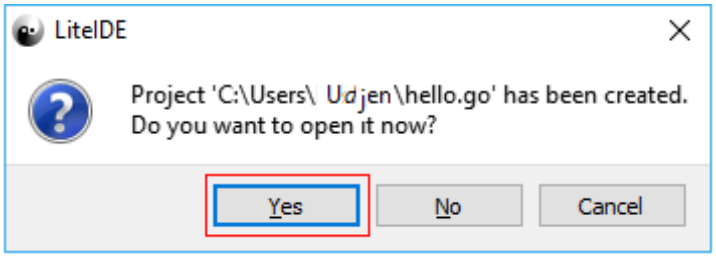

Рисунок 1.22. Вікно підтвердження відкриття проєкту

Потім у центральній частині вікна програми відкриється текстовий редактор, як показано на рисунку 1.23. Введемо до нього наступний код:

```
1    package main
2
3    import "fmt"
4
5    func main() {
```

```
6         fmt.Println("Hello Go")
7     }
```

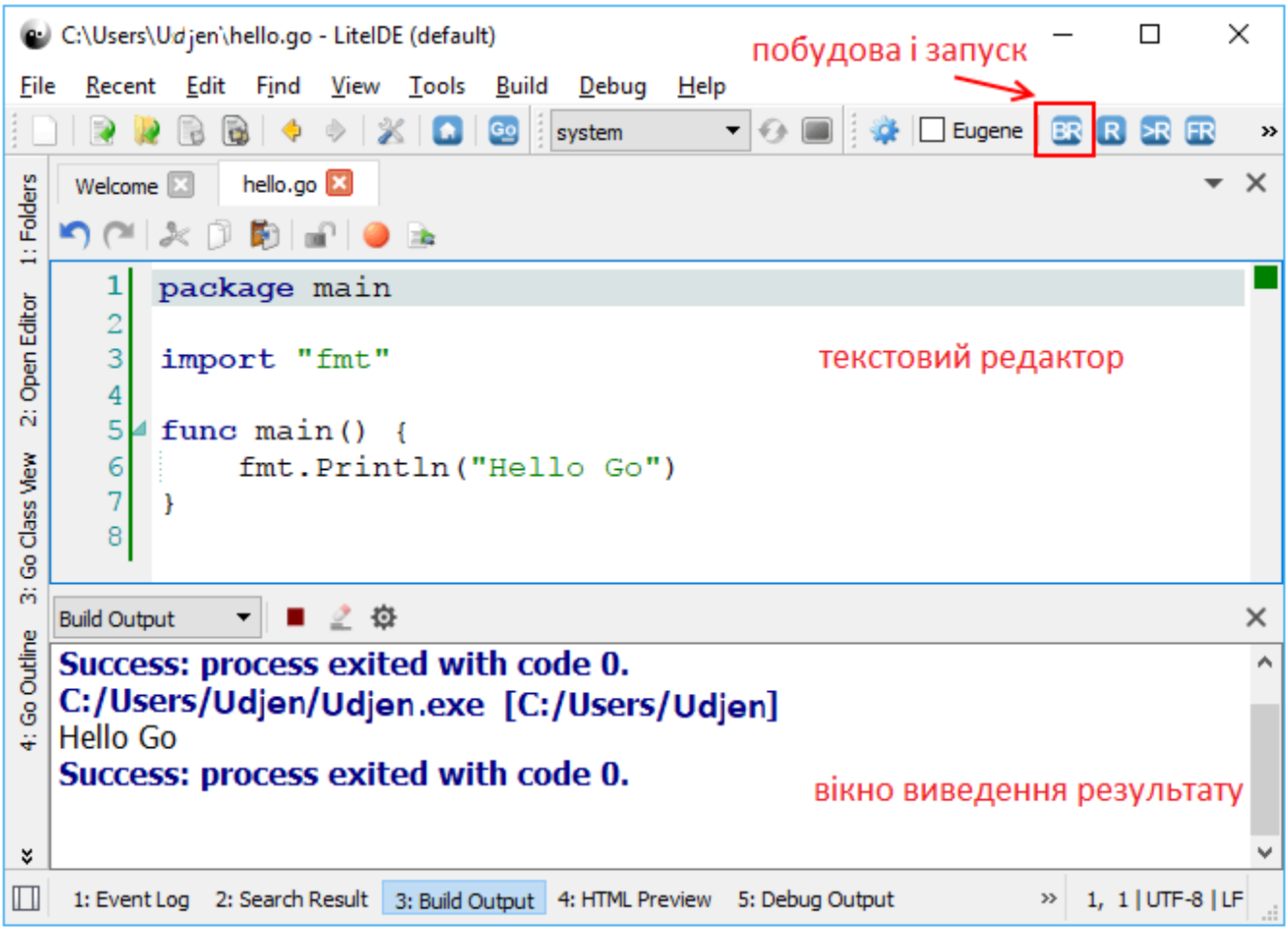

Рисунок 1.23. Зовнішній вигляд LiteIDE

Для збірки та запуску проєкту необхідно натиснути на панелі інструментів кнопку **BR**. І внизу вікна IDE з'явиться поле виведення, де можна побачити результат виконання програми.

Також у LiteIDE можна створювати та використовувати інші типи проєктів. Наприклад, створимо проєкт **Go1 Command Project**, який нехай називається **hello**, як показано на рисунку 1.24.

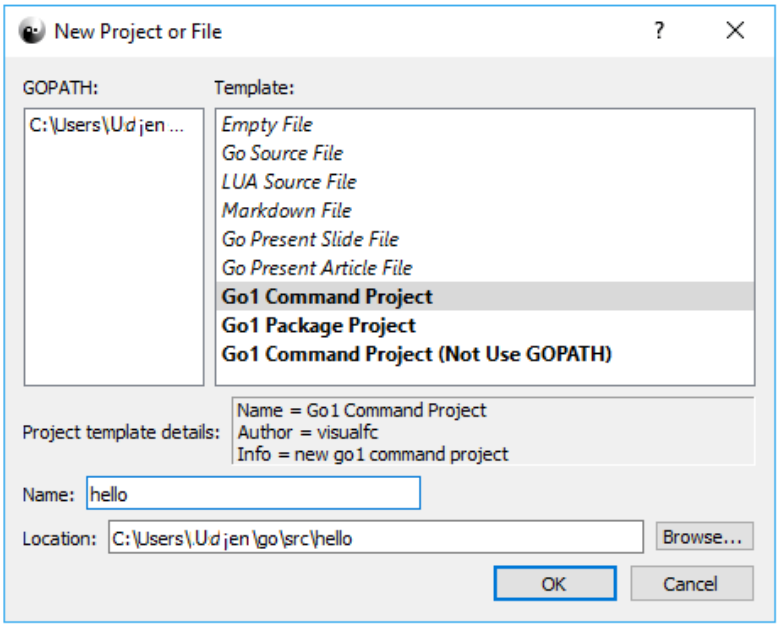

Рисунок 1.24. Створення проєкту Go1 Command Project

В результаті буде створено проєкт, який розташовуватиметься в окремій теці та який за замовчуванням складатиметься з двох файлів, як показано на рисунку 1.25.

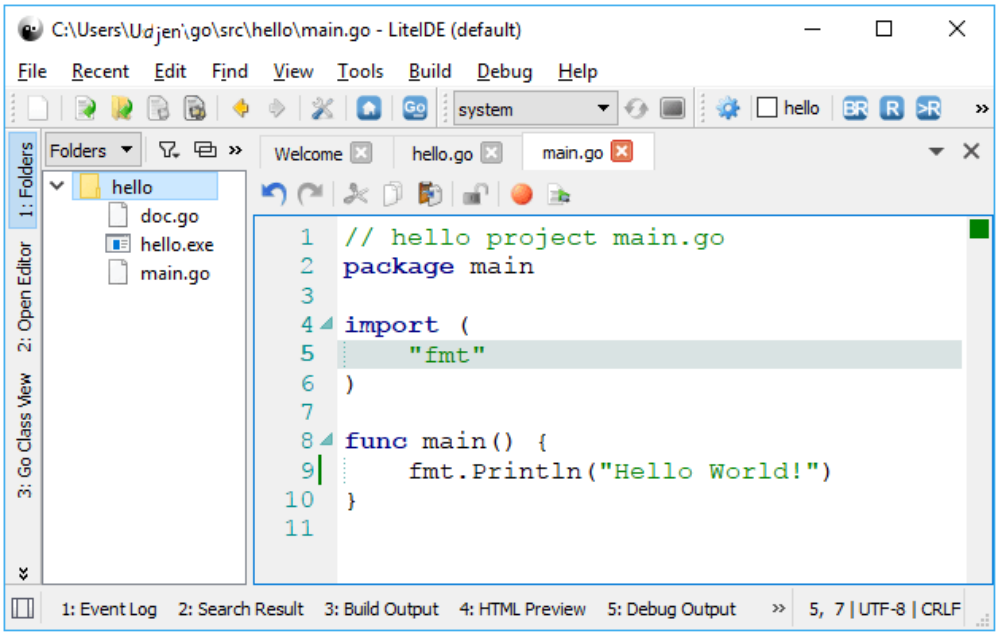

Рисунок 1.25. Зовнішній вигляд LiteIDE для проєкту Go1 Command Project

1.4. Go у Visual Studio Code

Інтегроване середовище розробки Visual Studio Code представляє кросплатформний потужний та легкий текстовий редактор від компанії Microsoft, який підтримує підсвічування синтаксису, інтелектуальну підказку

для різних мов програмування та багато іншого. Розглянемо, як можна використовувати цей текстовий редактор для розробки мовою Go.

Насамперед необхідно встановити цей текстовий редактор. Інсталятор для потрібної операційної системи (є підтримка для Windows, Mac OS, Linux) можна знайти за посиланням **https://code.visualstudio.com/** [5].

Після інсталяції Visual Studio Code за замовчуванням немає жодної підтримки мови Go. Тому необхідно встановити відповідне розширення. Для цього необхідно перейти до **Visual Studio Code** в меню **View -> Extensions (Розширення)**, як показано на рисунку 1.26.

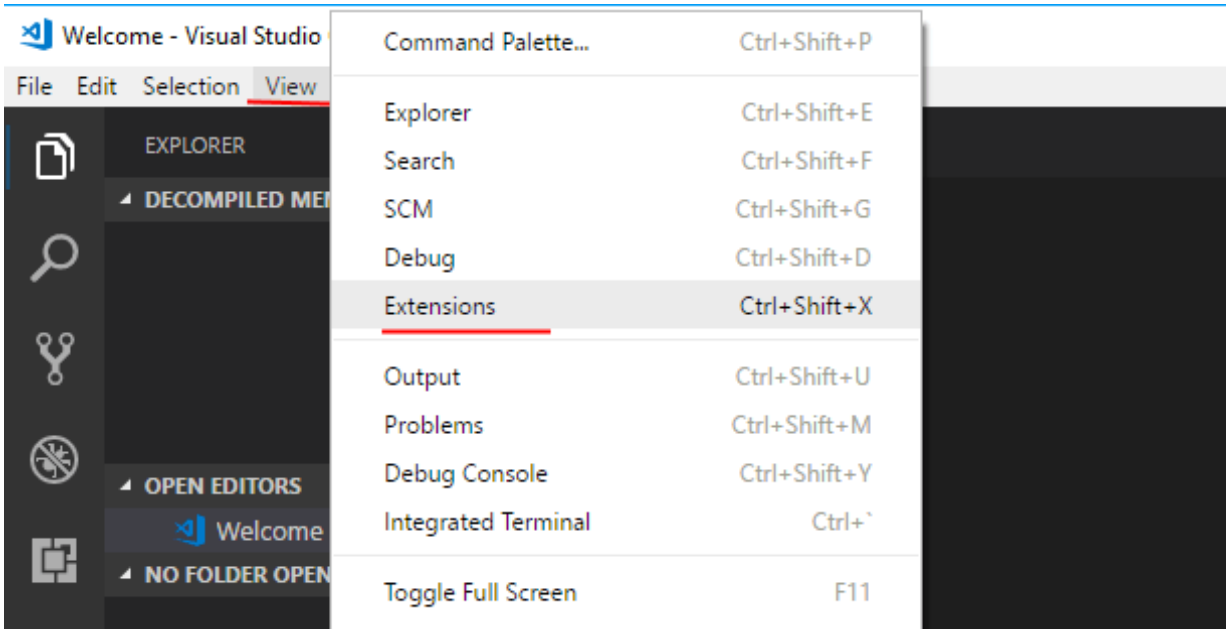

Рисунок 1.26. Вікно встановлення розширень для Visual Studio Code

У рядок пошуку розширень необхідно ввести "**go**", після чого з'явиться список знайдених розширень. Потрібно інсталювати перше розширення у цьому списку, яке має найбільшу кількість скачувань, як показано на рисунку 1.27.

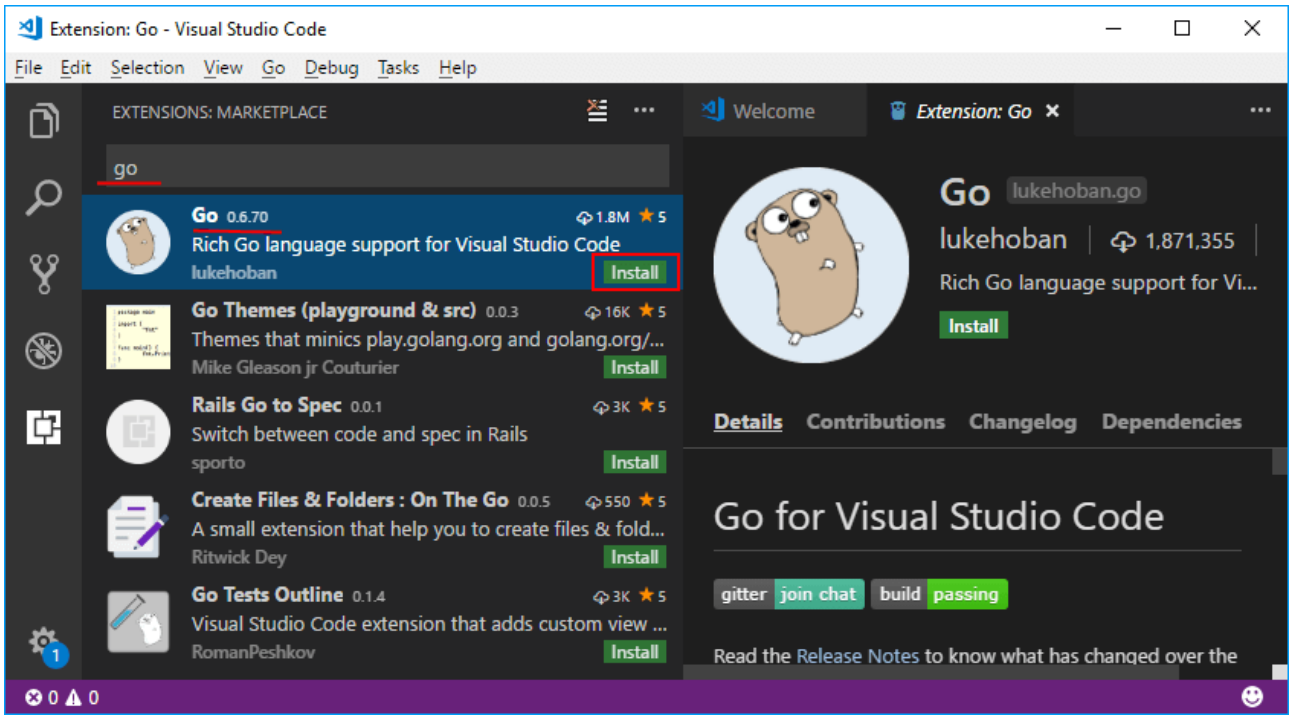

Рисунок 1.27. Вікно пошуку та вибору розширення Go для Visual Studio Code

Після інсталяції розширення необхідно визначити на жорсткому диску теку для зберігання файлів проєкту та відкрити цю папку у Visual Studio Code. Відкрити теку можна через пункт меню **File->Open Folder**. Потім створимо у **Visual Studio Code** новий файл, який назвемо **main.go**, як показано на рисунку 1.28.

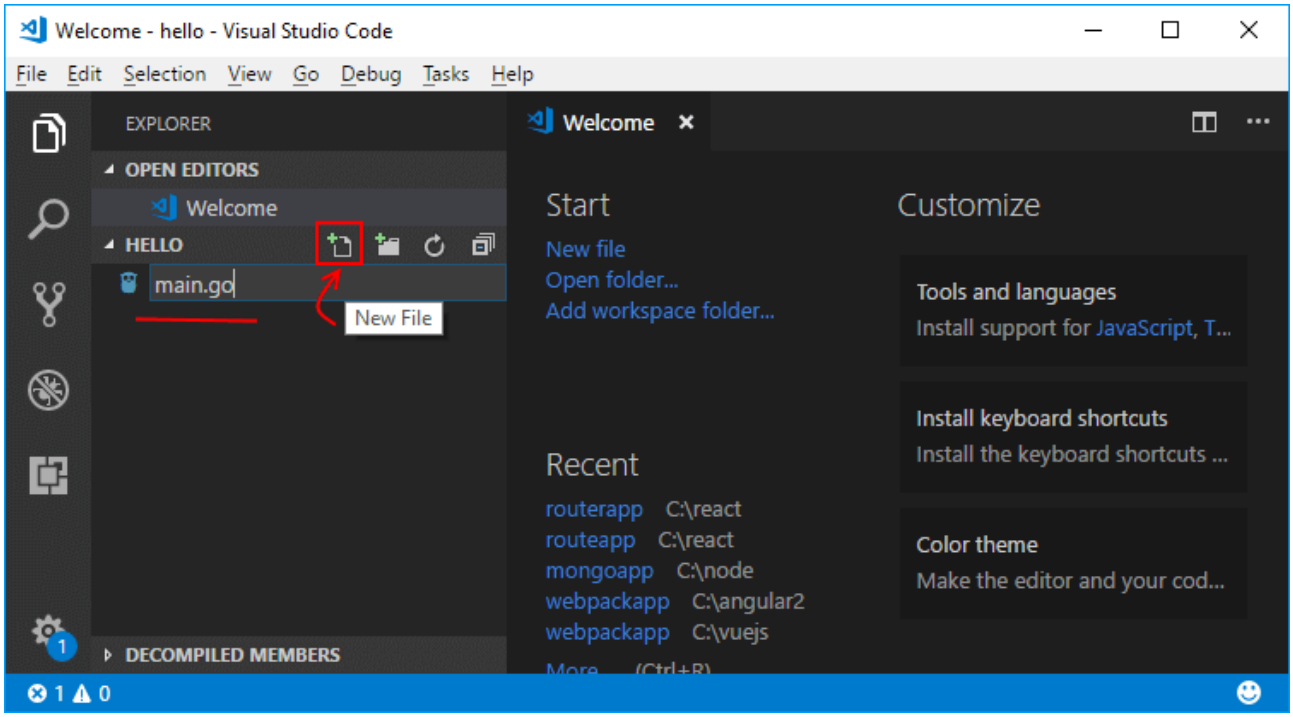

Рисунок 1.28. Вікно створення нового файлу

Відкриємо файл **main.go** і введемо до нього наступний код:

```
1   package main
2
3   import "fmt"
4
5   func main() {
6
7       fmt.Println("Hello Go")
8   }
```

Збережемо введений код, натиснувши комбінацію **Ctrl+S**, як показано на рисунку 1.29.

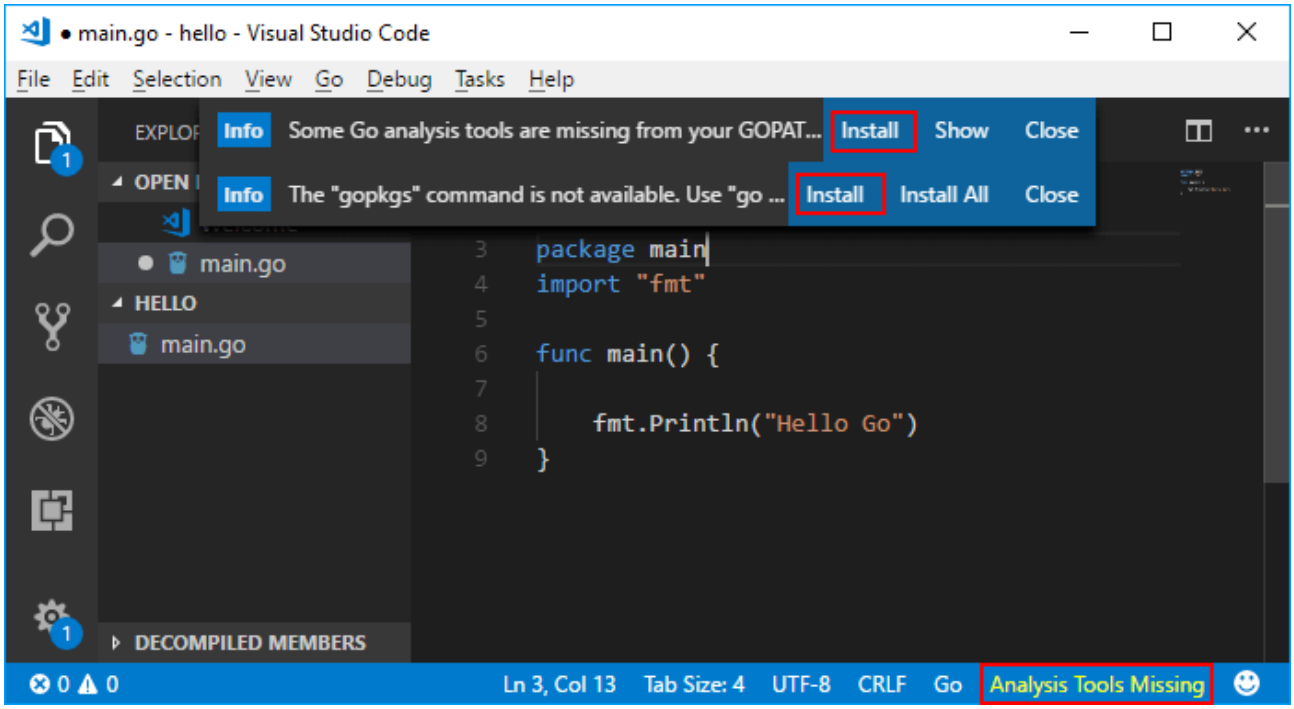

Рисунок 1.29. Вікно збереження програмного коду у новому файлі

Під час роботи з файлами **go** у **Visual Studio Code** можуть з'являтися різні повідомлення про необхідність встановлення додаткових плагінів. Наприклад, внизу вікна у статусному рядку може відображатись повідомлення "**Analysis Tools Missing**". Можна натиснути на це повідомлення, і нагорі **Visual Studio Code** відобразиться список пакетів, які бажано інсталювати додатково. Для їх встановлення необхідно натиснути кнопку **Install**.

Перевагою **Visual Studio Code** є те, що редактор має вбудований термінал. Відкривається термінал через пункт меню **View -> Integrated Terminal**. Після цього у вікні **Visual Studio Code** знизу відкриється вбудований термінал. За замовчуванням відкривається поточна тека проєкту.

Введемо в термінал команду, як показано на рисунку 1.30, та натиснемо **Enter**.

```
go run main.go
```

Рисунок 1.30. Зовнішній вигляд вікна введення команди

Після цього у вбудованому терміналі відобразиться результат роботи програми, як показано на рисунку 1.31.

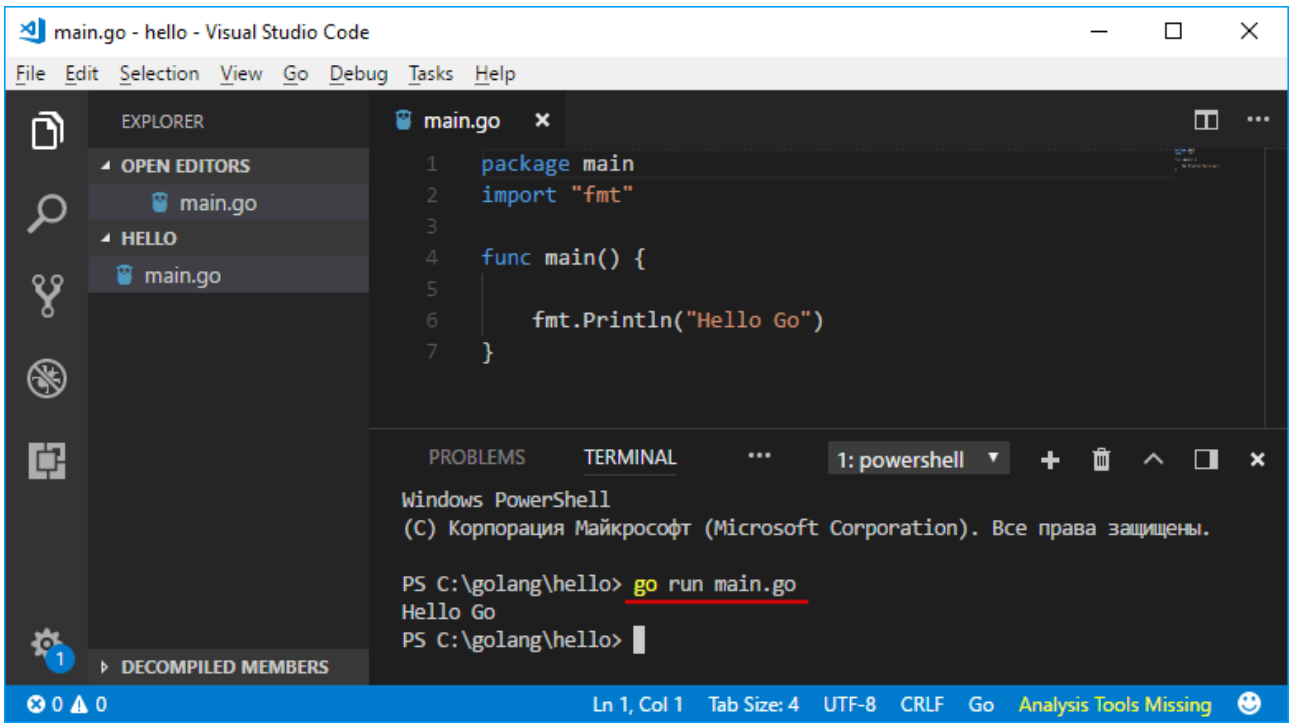

Рисунок 1.31. Результат роботи програми

Резюме

Go – це компільована, статично типізована мова програмування, розроблена Google, що спеціалізується на створенні вебсервісів, клієнт-серверних застосунків та інших програм. Її основні характеристики включають простоту, ефективність, підтримку багатопоточності та кросплатформність. Go активно використовується в проєктах таких компаній, як Google, Netflix, Docker, Kubernetes і Twitch. Розробка почалася у 2007 році, а перша стабільна версія вийшла у 2012 році. Мова має відкритий код, офіційні ресурси доступні на go.dev. Інструменти для роботи з Go включають текстові редактори, IDE (наприклад, GoLand) та командний рядок.

Викладений матеріал демонструє базові аспекти розробки мовою програмування Go. Описано процес створення та виконання першої програми,

включаючи налаштування середовища розробки. Пояснюється структура програми, імпорт бібліотек, визначення функції main та її роль у виконанні. Наведені приклади використання команд для компіляції та запуску програм у консолі. Також розглянуто використання інтегрованих середовищ розробки, таких як LiteIDE та Visual Studio Code, із детальними інструкціями щодо встановлення, налаштування та роботи з ними. Матеріал забезпечує зрозумілий вступ до розробки на Go.

Наступні розділи будуть послідовно висвітлювати все більш складні і важливі аспекти використання мови Go для програмування як вебзастосунків так і застосунків загального призначення.

Контрольні запитання і завдання

1. Що таке Go і яка основна мета її розробки?
2. Хто розробив мову Go і коли вона була анонсована?
3. Які основні сфери застосування мови Go?
4. Назвіть ключові характеристики мови Go.
5. Чому Go вважається кросплатформною мовою програмування?
6. Які компанії активно використовують Go у своїх проєктах?
7. Для яких операційних систем доступні інсталяційні пакети Go?
8. Яка версія Go є актуальною станом на 2024 рік?
9. Поясніть, у чому переваги використання Go для розробки вебсервісів.
10. Яка основна роль функції **main** у програмі мовою Go?
11. Що означає директива **package main**?
12. Як підключити бібліотеку **fmt** у програмі Go?
13. Яка команда використовується для виконання програми Go без створення окремого виконуваного файлу?
14. Як створити виконуваний файл програми Go?
15. Яке значення має команда **go build hello.go**?
16. Чому ім’я пакета у програмі, що виконується, має бути саме **main**?
17. Що відбудеться після виконання команди **go run hello.go**?
18. Як виглядає базовий шаблон програми на Go?
19. Встановіть компілятор Go на свою операційну систему, виконавши інструкції з матеріалу, і перевірте встановлення за допомогою команди **go version**.
20. Напишіть просту програму на Go і спробуйте її скомпілювати для різних операційних систем.

21. Проаналізуйте, чому Go обрали для таких систем, як Kubernetes і Docker. Які переваги мови допомагають у цих проєктах?
22. Виконайте інсталяцію Go на Linux або MacOS, скориставшись наданими в матеріалі інструкціями.
23. Додайте у програму імпорт бібліотеки **time** та виведіть поточний час за допомогою функції з цієї бібліотеки.
24. Створіть файл **hello.go**, напишіть у ньому програму, яка виводить «Hello, Go!». Скомпілюйте файл у виконуваний файл **hello.exe** і запустіть його.
25. Створіть програму з двома функціями: одна виводить повідомлення «Привіт!», а інша — «До побачення!». Викличте ці функції у потрібному порядку з функції **main**.
26. Встановіть LiteIDE, створіть новий проєкт, напишіть програму, яка обчислює квадрат числа, та запустіть її через LiteIDE.
27. Встановіть розширення Go у Visual Studio Code, створіть проєкт, напишіть програму, яка перевіряє, чи є число парним, і запустіть її через вбудований термінал.
28. Напишіть програму, яка виводить числа від 1 до 10, використовуючи цикл **for**.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Хто є одним із авторів мови Go?

a) Брендан Айх
b) Гвідо ван Россум
c) Кен Томпсон
d) Джеймс Гослінг

Правильна відповідь: c).

Що означає директива package main у програмі Go?

a) Визначає імпорт бібліотеки
b) Вказує основний пакет програми
c) Оголошує змінну
d) Вказує тип даних

Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які з цих компаній використовують Go?

a) Google

b) Netflix

d) Dropbox

Правильна відповідь: a), b), c).

Які середовища підтримують розробку на Go?

a) Visual Studio Code

b) Notepad++

c) LiteIDE

d) Eclipse

Правильна відповідь: a), b), c).

Завдання на відповідність

Встановіть відповідність між командами та їх функціями:

a) go version → 1. Перевірка версії Go.

b) go build → 2. Компіляція програми.

c) go run → 3. Виконання програми без попередньої компіляції.

d) go fmt → 4. Форматування коду.

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

d) → 4.

Встановіть відповідність між командою та її функцією:

a) `go build` → 1. Компіляція програми

b) `go run` → 2. Виконання програми

c) `fmt.Println` →3. Виведення тексту

d) `package main` →4. Визначення основного пакета

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

d) → 4.

Завдання з вибором істинності

Go підтримує лише 32-розрядні операційні системи.

a) Так

b) Ні

Правильна відповідь: b)

Команда go build створює виконуваний файл програми.

a) Так

b) Ні

Правильна відповідь: a).

Завдання з кодом

Напишіть команду для перевірки версії Go.

Правильна відповідь: go version

Яка помилка в програмі?

```
package main
import "fmt"
func main {
    fmt.Println("Hello, Go!")
}
```

a) Відсутній імпорт пакета

b) Немає дужок у визначенні функції

c) Неправильний синтаксис функції Println

d) Усе правильно

Правильна відповідь: b).

Завершіть код, щоб вивести:

```
The answer is 42.
```

Правильна відповідь: `fmt.Println("The answer is", 42)`

Вставити пропущені слова

Go — це компільована _____ типізована мова програмування.

Правильна відповідь: …статично…

Доповніть програму для виведення "Hello, Go!":

```
_________ main
import "fmt"
func main() {
```

```
    fmt._________ ("Hello, Go!")
}
```

Правильна відповідь: **…package…, …Println…**

Відкриті запитання

Що таке збирач сміття?

Правильна відповідь: Збирач сміття автоматично очищає пам'ять, видаляючи невикористовувані об'єкти, що зменшує ризик помилок і підвищує ефективність роботи програми.

2. ОСНОВИ МОВИ GO

Програма на мові Go зберігається в одному або кількох файлах. Кожен файл із програмним кодом повинен належати якомусь пакету. І на початку кожного файлу має здійснюватись оголошення пакета, до якого цей файл належить. Пакет оголошується за допомогою ключового слова package.

2.1. Структура програми

Всередині файлу програми можна використовувати функціонал з інших пакетів. У цьому випадку пакети, що використовуються, потрібно імпортувати за допомогою ключового слова **import**. Пакети, що імпортуються, повинні вказуватись після оголошення пакета для поточного файлу, наприклад:

```
1   package main
2   import "fmt"
```

У цьому прикладі поточний файл перебуватиме у пакеті **main**. І потім він підключає пакет **fmt**.

Головний пакет програми обов’язкова повинен мати назву "**main**". Оскільки саме цей пакет визначає, що створюватиметься виконуваний файл програми, який після компіляції можна буде запускати на виконання.

Після підключення інших пакетів розміщуються оголошення типів, змінних, функцій, констант.

Вхідною точкою застосунку є функція з ім'ям **main**. Вона обов'язково має бути визначена у програмі. Все, що виконується у програмі, виконується саме у функції **main**, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       fmt.Println("Hello Go!")
6   }
```

Базовим елементом програми є інструкції. Наприклад, виклик функції **fmt.Println("Hello Go!")** є окремою інструкцію. Кожна інструкція виконує певну дію та розміщується з нового рядку, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
```

```
5       fmt.Println("Hello Go!")
6       fmt.Println("Hello Golang!")
7       fmt.Println("Hello Go!")
8   }
```

В наведеному прикладі функція **main** містить три інструкції, які виводить на консоль рядок, і кожна інструкція розміщується з нового рядка.

Існує можливість розміщувати кілька інструкцій і на одному рядку, але тоді їх треба відокремлювати символом крапки з комою (**;**), наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       fmt.Println("Hello Go!");fmt.Println("Hello
    Golang!");fmt.Println("Hello Go!")
6   }
```

У той же час розміщення інструкцій з нового рядка забезпечує більш зручний формат для прийняття, тому краще для використання.

2.1.1. Коментарі

Програма може мати коментарі. Коментарі служать для опису дій, які виконує програма або її частини. При компіляції коментарі не враховуються і не впливають на роботу програми. Коментарі бувають однорядковими та багаторядковими.

Однорядковий коментар розташовується в один рядок після символу подвійного слешу (//). Все, що знаходиться після цих символів, сприймається компілятором як коментар. Багаторядковий коментар розташовується між символами **/*** та ***/** і може займати декілька рядків, наприклад:

```
1   /*
2       Перша програма
3       мовою Go
4   */
5   package main    // Визначення пакета для поточного
    файлу
6   import "fmt"    // Підключення пакета fmt
7
8   // Визначення функції main
9   func main() {
```

```
10      fmt.Println("Hello Go!")     // Виведення рядка на
    консоль
11  }
```

2.2. Змінні

Для зберігання даних у програмі використовуються змінні.

2.2.1. Визначення змінних

Змінною є іменована ділянка в пам'яті, яка може зберігати деяке значення. Для визначення змінної використовується ключове слово **var**, після якого вказується ім'я змінної, а потім вказується її тип, наприклад:

```
1   var ім'я_змінної тип_даних
```

Ім'я змінної представляє довільний ідентифікатор, що складається з алфавітних та цифрових символів та символу підкреслення. При цьому першим символом може бути або алфавітний символ, або символ підкреслення. Імена не повинні представляти одне з ключових слів: **break**, **case**, **chan**, **const**, **continue**, **default**, **defer**, **else**, **fallthrough**, **for**, **func**, **go**, **goto**, if, **import**, **interface**, **map**, **package**, **range**, **return**, **select**, **struct**, **switch**, **type**, **var**.

Наведемо приклад найпростішого визначення змінної, а саме:

```
1   var hello string
```

В наведеному прикладі змінна називається **hello** і вона представляє тип **string**, тобто деякий рядок.

Можна одночасно оголосити відразу кілька змінних через кому, наприклад:

```
1   var a, b, c string
```

В наведеному прикладі визначено змінні **a, b, c**, які мають тип **string**.

Після визначення змінної їй можна надавати будь-яке значення, яке відповідає її типу, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       var hello string
6       hello = "Hello world"
7       fmt.Println(hello)
8   }
```

Оскільки змінна **hello** має тип **string**, то їй можна присвоїти рядок. У наведеному вище прикладі змінна **hello** зберігає рядок "**Hello world**". За допомогою функції **Println** значення цієї змінної виводиться на консоль.

Також важливо враховувати, що мова Go реєстрозалежна мова, тобто змінні з іменами **hello** та **Hello** представлятимуть різні змінні, наприклад:

```
var hello string
hello = "Hello world"
fmt.Println(Hello)  // ! Помилка змінної Hello немає, є змінна hello
```

У мові Go можна відразу при оголошенні змінної надати їй початкове значення. Такий прийом називається ініціалізацією, наприклад:

```
package main
import "fmt"

func main() {
    var hello string = "Hello world"
    fmt.Println(hello)
}
```

Якщо необхідно одразу визначити кілька змінних і присвоїти їм початкові значення, то можна обернути їх у дужки, наприклад:

```
package main
import "fmt"

func main() {
    var (
        name string = "Tom"
        age int = 27
    )

    fmt.Println(name)    // Tom
    fmt.Println(age)     // 27
}
```

Відмінною особливістю змінних є те, що їх значення можна багаторазово змінювати, наприклад:

```
package main
import "fmt"

func main() {
```

```
5       var hello string = "Hello world"
6       fmt.Println(hello)  // Hello world
7
8       hello = "Hello Go"
9       fmt.Println(hello)  // Hello Go
10
11      hello = "Go Go Go Ole Ole Ole"
12      fmt.Println(hello)  // Go Go Go Ole Ole Ole
13  }
```

2.2.2. Скорочене визначення змінної

Мова Go підтримує скорочене визначення змінної у наступному форматі:

```
1   ім'я_змінної := значення
```

Тобто після імені змінної ставиться символ двокрапки і знак рівності (**:=**) і потім вказується її значення, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       name := "Tom"
6       fmt.Println(name)
7
8   }
```

У наведеному прикладі тип даних явно не зазначено, він виводиться автоматично зі значення, що присвоюється.

2.3. Типи даних

Всі дані, які зберігаються в пам'яті, насправді представляють набір бітів. І саме тип даних визначає, як інтерпретуватимуться ці дані та які операції з ними можна виконувати. Мова Go є статично типізованою мовою, тобто всі дані, що використовуються в програмі, мають певний тип.

Мова Go має низку вбудованих типів даних, і навіть дозволяє визначати свої власні типи. Розглянемо базові типи даних, які можна використовувати в мові Go.

2.3.1. Цілочислові типи

Мова Go має низку вбудованих типів, які представляють цілі числа, а саме:

- **int8** : представляє ціле число від -128 до 127 і займає у пам'яті 1 байт (8 біт);
- **int16** : представляє ціле число від -32768 до 32767 і займає у пам'яті 2 байти (16 біт);
- **int32** : представляє ціле число від -2147483648 до 2147483647 і займає 4 байти (32 біти);
- **int64** : представляє ціле число від -9223372036854775808 до 9223372036854775807 і займає 8 байт (64 біта);
- **uint8** : представляє ціле число від 0 до 255 і займає 1 байт;
- **uint16** : представляє ціле число від 0 до 65535 і займає 2 байти;
- **uint32** : представляє ціле число від 0 до 4294967295 і займає 4 байти;
- **uint64** : представляє ціле число від 0 до 18446744073709551615 і займає 8 байт;
- **byte** : синонім типу **uint8**, представляє ціле число від 0 до 255 і займає 1 байт;
- **rune** : синонім типу **int32**, представляє ціле число від -2147483648 до 2147483647 і займає 4 байти;
- **int** : представляє ціле число зі знаком, яке залежно від платформи може займати або 4 байти, або 8 байт. Тобто відповідати або **int32**, або **int64**;
- **uint** : представляє ціле беззнакове число (тобто без знака), яке, аналогічно типу **int**, залежно від платформи може займати або 4 байти, або 8 байт. Тобто відповідати або **uint32**, або **uint64**.

Не складно запам'ятати, що є типи зі знаком, тобто які можуть бути негативними, і є безнакові позитивні типи, що починаються з префікса **u**, наприклад **uint32**. Тип **byte** є синонімом для **uint8**, а **rune** є синонімом для **int32**.

Варто додатково розглянути типи **int** та **uint**. Вони мають найбільш ефективний розмір для певної платформи (32 або 64 біти). Це найбільш використовуваний тип представлення цілих чисел у програмі. Причому різні компілятори можуть надавати різний розмір для цих типів навіть для однієї платформи.

Наведемо приклади визначення змінних цілих типів:

```
1   var a int8 = -1
2   var b uint8 = 2
3   var c byte = 3  // byte - синонім типу uint8
4   var d int16 = -4
5   var f uint16 = 5
6   var g int32 = -6
7   var h rune = -7     // rune - синонім типу int32
8   var j uint32 = 8
9   var k int64 = -9
10  var l uint64 = 10
11  var m int = 102
12  var n uint = 105
```

2.3.2. Числа з рухомою крапкою

Для представлення дробових чисел мова Go має два типи, а саме:

- **float32** : представляє число з рухомою крапкою від $1.4*10^{-45}$ до $3.4*10^{38}$(для позитивних). Займає в пам'яті 4 байти (32 біти);
- **float64** : представляє число з рухомою крапкою від $4.9*10^{-324}$ до $1.8*10^{308}$ (для позитивних) і займає 8 байт.

Тип **float32** забезпечує шість десяткових цифр точності, тоді як точність, що забезпечується типом **float64**, становить близько 15 цифр.

Наведемо приклади використання типів **float32** та **float64**, а саме:

```
1   var f float32 = 18
2   var g float32 = -4.5
3   var d float64 = -0.23
4   var pi float64 = 3.14
5   var e float64 = 2.7
```

У якості роздільника між цілою і дробовою частиною у мові Go застосовується символ крапки (**.**).

2.3.3. Комплексні числа

У мові Go існують окремі типи для представлення комплексних чисел, а саме:

- **complex64** : комплексне число, де дійсна та уявна частини представляють числа **float32**;

- **complex128** : комплексне число, де дійсна та уявна частини представляють числа **float64**.

Наведемо приклади використання комплексних чисел, а саме:

```
1   var f complex64 = 1+2i
2   var g complex128 = 4+3i
```

2.3.4. Логічний тип

Логічний тип або тип **bool** може мати одне із двох значень: **true** (істина) або **false** (хиба), наприклад:

```
1   var isAlive bool = true
2   var isEnabled bool = false
```

2.3.5. Рядки

Рядки у мові Go представлені типом **string** і позначається послідовністю символів у подвійних лапках, наприклад:

```
1   var name string = "Олекса Довбуш"
```

Окрім звичайних символів рядок може містити спеціальні послідовності - керуючі послідовності, які починаються зі зворотного слешу. Наведемо найбільш поширені спеціальні послідовності, а саме:

- **\n** : перехід на новий рядок;
- **\r** : повернення каретки;
- **\t** : табуляція;
- \" : подвійні лапки всередині рядка;
- \\ : зворотний слеш.

2.3.6. Значення за замовчуванням

Якщо змінній не присвоєно значення, то вона матиме значення за замовчуванням, що визначено для її типу. Для числових типів це число **0**, для логічного типу це **false**, для рядків це порожній рядок ("").

2.3.7. Неявна типізація

При визначенні змінної можна не вказувати тип у разі, якщо явно ініціалізувати змінну яким-небудь значенням, наприклад:

```
1   var name = "Tom"
```

У цьому випадку компілятор на підставі присвоєного значення явно виводить тип змінної. Якщо присвоюється рядок, то змінна буде представляти тип **string**, якщо присвоюється ціле число, то змінна представляє тип **int** і т.д.

Те саме відбувається при скороченому визначенні змінної, коли також явно не вказується тип даних, а саме:

```
name := "Tom"
```

Варто пам’ятати, що якщо не вказується тип змінної, то змінній обов'язково потрібно надати деяке початкове значення. Оголошення змінної одночасно без вказівки типу даних та початкового значення буде помилкою, наприклад:

```
var name     // ! Помилка
```

Тому необхідно або вказати тип даних і змінна матиме значення за замовчуванням, наприклад:

```
var name string
```

Або вказати початкове значення, виходячи з якого буде автоматично виведено тип даних, наприклад:

```
var name = "Tom"
```

Або і те, і інше одночасно, наприклад:

```
var name string = "Tom"
```

Неявна типізація кількох змінних виконується наступним чином:

```
var (
        name = "Tom"
        age = 27
)
```

Або наступним чином:

```
var name, age = "Tom", 27
```

2.4. Константи

Константи, як і змінні, зберігають деякі дані, але на відміну від змінних значення констант не можна змінити, вони встановлюються один раз. Обчислення констант проводиться під час компіляції. Завдяки цьому зменшується кількість роботи, яку необхідно зробити під час виконання програми, спрощується пошук помилок, пов'язаних із константами, оскільки деякі з помилок можна виявити на момент компіляції.

Для визначення констант в мові Go застосовується ключове слово **const**, наприклад:

```
const pi float64 = 3.1415
```

На відміну від змінної, значення константи не можна змінити. А якщо спробувати це зробити, то при компіляції згенерується помилка, наприклад:

```
const pi float64 = 3.1415
```

```
2   pi = 2.7182                 // ! Помилка
```

В одному визначенні можна оголосити кілька констант, наприклад:

```
1   const (
2       pi float64 = 3.1415
3       e float64 = 2.7182
4   )
```

або наступним чином:

```
1   const pi, e = 3.1415, 2.7182
```

Якщо у константи не вказано тип, то він виводиться неявно виходячи з того значення, яким ініціалізується константа, наприклад:

```
1   const n = 5       //  тип int
```

У той самий час необхідно обов'язково ініціалізувати константу початковим значенням під час її оголошення. Наприклад, наступні визначення констант є неприпустимими, оскільки вони не ініціалізовані:

```
1   const d
2   const n int
```

Якщо визначається послідовність констант, то ініціалізацію значеннями можна опустити для всіх констант, окрім першої. У цьому випадку константа без значення отримає значення попередньої константи, наприклад:

```
1   const (
2       a = 1
3       b
4       c
5       d = 3
6       f
7   )
8   fmt.Println(a, b, c, d, f)       // 1, 1, 1, 3, 3
```

Константи можна ініціалізувати лише константними значеннями, наприклад, літералами типу чисел чи рядків, або значеннями інших констант. Але ініціалізувати константу значенням змінних не можна, наприклад:

```
1   var m int = 7
2   // const k = m       // ! Помилка: m - змінна
3   const s = 5      // Ок: 5 - Числова константа
4   const n = s      // Ок: s - Константа
```

2.5. Арифметичні операції

Мова Go підтримує всі основні операції. Нижче наведено бінарні оператори Go (арифметичні, логічні та оператори порівняння) у порядку зменшення пріоритету:

```
*    /    %    «    »    &    &^
+    -    |    ^
==   !=   <    <=   >    >=
&&
||
```

Є лише п'ять рівнів пріоритету для бінарних операторів. Оператори на однаковому рівні лівоасоціативні, тому для ясності або для того, щоб дії виконувались у потрібному порядку, можуть бути необхідні дужки, як, наприклад, у виразі на зразок **mask & (1 << 28)**.

Кожен оператор у перших двох рядках, наприклад +, має відповідний оператор, що присвоює, такий як +=, який може використовуватися для скорочення запису інструкції присвоювання.

Арифметичні оператори +, -, * та / можуть застосовуватися до цілих чисел, чисел з рухомою точкою та комплексними числами, але оператор отримання залишку від ділення **%** застосовується тільки до цілих чисел. Поведінка оператора **%** для негативних чисел відрізняється у різних мовах програмування. У мові Go знак залишку завжди такий самий, як і знак діленого, отже і **-5%3**, і **-5%-3** дорівнюватимуть **-2** . Поведінка оператора / залежить від того, чи є його операнди цілими числами, тому **5.0/4.0** дорівнюватиме **1.25**, але **5/4** дорівнюватиме **1**, так як при цілочисловому діленні відбувається усічення результату.

Якщо результат арифметичної операції, як знакової, так і беззнакової, має більше бітів, ніж може бути представлено типом результату, то говорять про переповнення (**overflow**). При цьому старші біти, які не містяться в результаті, мовчки відкидаються. Якщо вихідне число має знаковий тип, результат може бути негативним, якщо лівий біт дорівнює **1**, як показано в наступному прикладі з **int 8**:

```
1   var u unit = 255
2   fmt.Println(u, u+1, u*u)    // 255 0 1
3   var i int8 = 127
5   fmt.Println(i, i+1, i*i)    // 127 -128 1
```

Два цілі числа одного й того ж типу можна порівняти з використанням бінарних операторів відношення, що наведені нижче:

== дорівнює;

!= не дорівнює;

< менше ніж;

<= менше або дорівнює;

> більше ніж;

>= більше або дорівнює.

Результат виразу відношення є логічним значенням.

Фактично всі значення фундаментальних типів (логічні, числа та рядки) є порівнюваними, а це означає, що два значення однакового типу можна порівнювати за допомогою операторів == та **!=**. Крім того, цілі числа, числа з рухомою крапкою і рядки є впорядкованими за допомогою операторів відношення. Значення багатьох інших типів не є порівнюваними, а багато інших типів не є впорядкованими.

У мові Go також є унарні оператори:

+ унарний знак плюс (не виконує жодної дії);

- унарний знак мінус.

Для цілих чисел запис **+x** є скороченою формою запису для **0+x**, а запис **-x**- скороченою формою запису для **0-x**. Для чисел з рухомою крапкою та комплексних чисел запис **+x** буде еквівалентний просто **x**, а запис **-x** означатиме значення **x** із протилежним знаком.

Розглянемо використання арифметичних операторів, що підтримуються мовою Go на конкретних прикладах.

Оператор додавання (+) повертає суму двох чисел, наприклад:

```
1  package main
2  import "fmt"
3
4  func main() {
5      var a = 4
6      var b = 6
7      var c = a + b
8      fmt.Println(c)       // 10
9  }
```

Оператор віднімання (-) повертає різницю двох чисел, наприклад:

```
1  package main
2  import "fmt"
```

```
3
4   func main() {
5       var a = 4
6       var b = 6
7       var c = a - b
8       fmt.Println(c)      // -2
9   }
```

Оператор множення (*****) повертає добуток двох чисел, наприклад:

```
1   var a = 4
2   var b = 6
3   var c = a * b    // 24
```

Оператор ділення (**/**) двох чисел, наприклад:

```
1   var a int = 10
2   var b int = 4
3   var c int = a / b
4   fmt.Println(c)      // 2
5
6   var k float32 = 10
7   var l float32 = 4
8   var m float32 = k / l
9   fmt.Println(m)      // 2.5
```

При ділення варто бути уважним, тому що, якщо в операції беруть участь два цілих числа, то результат ділення буде округлятися до цілого числа, навіть якщо результат присвоюється змінній типу **float32** або **float64**, наприклад:

```
1   var m float32 = 10 / 4      // 2
```

Щоб результат представляв дійсне число, один з операндів також повинен представляти дійсне число, наприклад:

```
1   var m float32 = 10 / 4.0        // 2.5
```

Оператор (**%**) повертає залишок від ділення. У цій операції можуть брати участь лише цілочисельні операнди, наприклад:

```
1   var c int = 35 % 3      // 2 (35 - 33 = 2)
```

Оператор постфіксного інкремента (**x++**). Збільшує значення змінної на одиницю, наприклад:

```
1   var a int = 8
2   a++
3   fmt.Println(a)      // 9
```

Оператор постфіксного декремента (**x--**). Зменшує значення змінної на одиницю, наприклад:

```
var a int = 8
a--
fmt.Println(a)      // 7
```

2.6. Умовні вирази

Умовні вирази представляють логічні операції та операції відношення. Вони представляють деяку умову і повертають значення типу **bool**: **true**, якщо умова є істинною або **false**, якщо умова хибна.

2.6.1. Операції відношення

Операції відношення дозволяють порівняти два значення. Мова Go підтримує наступні бінарні оператори відношення.

Оператор «дорівнює» (==). Повертає **true**, якщо обидва операнди рівні, і **false**, якщо вони не дорівнюють один одному, наприклад:

```
package main
import "fmt"

func main() {
    var a int = 8
    var b int = 3
    var c bool = a == b
    fmt.Println(c)      // false
}
```

Оператор «більше ніж» (>). Повертає **true**, якщо перший операнд більший за другий, і **false**, якщо перший операнд менший за другий, наприклад:

```
var a int = 8
var b int = 3
var c bool = a > b   // true
```

Оператор «менше ніж» (<). Повертає **true**, якщо перший операнд менший за другий, і **false**, якщо перший операнд більший за другий, наприклад:

```
var a int = 8
var b int = 3
var c bool = a < b   // false
```

Оператор «менше або дорівнює» (<=). Повертає **true**, якщо перший операнд менше або дорівнює другому, і **false**, якщо перший операнд більший за другий, наприклад:

```
var a int = 8
```

```
2   var b int = 3
3   var c bool = a <= b  // false
```

Оператор «більше або дорівнює» (>=). Повертає **true**, якщо перший операнд більший або дорівнює другому, і **false**, якщо перший операнд менший за другий, наприклад:

```
1   var a int = 8
2   var b int = 3
3   var c bool = a >= b  // true
```

Оператор «не дорівнює» (**!=**). Повертає **true**, якщо перший операнд не дорівнює другому і **false**, якщо обидва операнда рівні, наприклад:

```
1   var a int = 8
2   var b int = 3
3   var c bool = a != b // true
4   var d bool = a != 8 // false
```

Як правило, операції відношення застосовуються в умовних конструкціях типу **if...else**, які будуть розглянуті пізніше.

2.6.2. Логічні операції

Логічні операції застосовуються до логічних виразів, якими можуть бути відношення, логічні змінні чи константи тощо. Розглянемо призначення і приклади застосування логічних операторів.

Унарний оператор заперечення (**!**), який інвертує значення. Якщо операнд дорівнює **true**, то повертається **false**, інакше повертається **true**, наприклад:

```
1   var a bool = true
2   var b bool = !a      // false
3   var c bool = !b      // true
```

Оператор кон'юнкції або логічного множення (**&&**), який повертає **true**, якщо обидва операнди не дорівнюють **false**. Якщо хоча б один операнд дорівнює **false**, то повертається **false**, , наприклад:

```
1   var b bool = 4 > 5 && 6 > 8        // false
2   var c bool = 3 <= 5 && 10 > 8 // true
```

Оператор диз'юнкції або логічного додавання (||) повертає **true**, якщо хоча б один операнд не дорівнює **false**. Якщо обидва операнди дорівнюють **false**, то повертається **false**, наприклад:

```
1   var b bool = 4 > 5 || 6 > 8        // false
2   var c bool = 3 == 5 || 10 > 8    // true
```

2.7. Порозрядні операції

Порозрядні операції виконуються над окремими розрядами чисел у двійковому поданні. Наприклад, число п'ять у двійковій системі має три розряди 101, а число вісім має чотири розряди 1000.

2.7.1. Операції зсуву

Операції зсуву дозволяють зміщувати двійкове подання числа на кілька розрядів праворуч чи ліворуч. Операції зсуву застосовуються тільки до цілих операндів. У мові Go є два оператора зсуву.

Оператор (<<), який зсуває двійкове подання числа, представленого першим операндом, ліворуч на певну кількість розрядів, що задається другим операндом.

Оператор (>>), який зсуває двійкове подання числа праворуч на певну кількість розрядів.

Наведемо приклади застосування операторів зсуву:

```
1   var b int = 2 << 2;            // 2DEC=10BIN  зсув на
    два розряди ліворуч => 1000BIN=8DEC
2   var c int = 16 >> 3;           // 16DEC=10000BIN зсув
    на три розряди праворуч => 100BIN=2DEC
```

В наведеному прикладі число 2 у двійковому поданні відповідатиме 10. Якщо зсунути число 10 на два розряди ліворуч, то вийде 1000, що в десятковій системі відповідатиме числу 8.

Число 16 у двійковому поданні відповідатиме 10000. Якщо зсунути число 10000 на три розряди праворуч, тобто три останні розряди відкидаються, то вийде число 10, що в десятковій системі відповідатиме числу 2.

2.7.2. Порозрядні операції

Порозрядні операції виконуються тільки над розрядами цілих операндів. У мові Go є кілька операторів для порозрядних операцій.

Оператор порозрядної кон'юнкції (**&**) або операція **І**, яка є операцією порозрядного множення. Вона повертає 1, якщо обидва з відповідних розрядів обох чисел дорівнюють 1. Та повертає 0, якщо розряд хоча б одного з чисел дорівнює 0.

Оператор порозрядної диз'юнкції (|) або операція **АБО**, яка є операцією порозрядного складання. Вона повертає 1, якщо хоча б один із відповідних розрядів обох чисел дорівнює 1.

Оператор порозрядного виключення (^) або операція **виключного АБО**. Вона повертає 1, якщо тільки один із відповідних розрядів обох чисел дорівнює 1.

Оператор скидання біта (**&^**) або операція **І-НЕ**. У виразі **z = x &^ y** кожен біт **z** дорівнює 0, якщо відповідний біт **y** дорівнює 1. Якщо біт **y** дорівнює 0, то береться значення відповідного біта з **x**.

Наведемо приклади застосування порозрядних операцій:

```
1   package main
2   import "fmt"
3
4   func main() {
5       var a int = 5 | 2; // 101 | 010 = 111  =>7DEC
6       var b int = 6 & 2; // 110 & 010 = 10  => 2DEC
7       var c int = 5 ^ 2; // 101 ^ 010 = 111 => 7DEC
8       var d int = 5 &^ 6;// 101 &^ 110 = 001 => 1DEC
9   }
```

У прикладі вираз 5 | 2 дорівнює 7. Число 5 у двійковому записі дорівнює 101, а число 2 - 10 або 010. Складемо відповідні розряди обох чисел. Як було зазначено вище, при додаванні якщо хоча б один розряд дорівнює 1, то сума обох розрядів дорівнює 1. Тому отримаємо наступне:

1 0 1
0 1 0
1 1 1

Отже результат дорівнюватиме числу 111, що у десятковому записі становить число 7.

Візьмемо інший вираз 6 & 2. Число 6 у двійковому поданні дорівнює 110, а число 2 - 10 або 010. Помножимо відповідні розряди обох чисел. Пам’ятаймо, що добуток обох розрядів дорівнює 1, якщо обидва ці розряди дорівнюють 1. Інакше добуток дорівнює 0. Тому отримаємо наступне:

1 1 0
0 1 0
0 1 0

Отже результат дорівнюватиме числу 010, що у десятковій системі дорівнює 2.

2.8. Масиви

Масиви є послідовністю елементів певного типу.

2.8.1. Визначення масивів

У мові Go масив визначається наступним способом:

```
var numbers [число_елементів]тип_елементів
```

Наприклад, масив із п'яти елементів типу **int** оголошуватиметься наступним чином:

```
var numbers [5]int
```

За такого визначення всі елементи масиву ініціалізуються значеннями за замовчуванням. Але також можна ініціалізувати елементи масиву іншими значеннями, наприклад:

```
var numbers [5]int = [5]int{1,2,3,4,5}
```

Значення, які передаються масиву, подаються у фігурних дужках через кому. При цьому значень не може бути більшим за довжину масиву. В наведеному прикладі довжина масиву дорівнює 5, тому не можна у фігурних дужках визначити більше, ніж п'ять елементів. Але можна визначити менше елементів, наприклад:

```
var numbers [5]int = [5]int{1,2}
fmt.Println(numbers)     // [1 2 0 0 0]
```

У цьому разі елементи, для яких не вказано значення, матимуть значення за замовчуванням.

Також можна застосовувати скорочене визначення змінної масиву, а саме:

```
numbers := [5]int{1,2,3,4,5}
```

Якщо в квадратних дужках замість довжини вказано три крапки, то довжина масиву визначається, виходячи з кількості переданих йому елементів:

```
    var numbers = [...]int{1,2,3,4,5}    // Довжина масиву
1   5
2   numbers2 := [...]int{1,2,3}          // Довжина масиву
3   3
4   fmt.Println(numbers)                 // [1 2 3 4 5]
    fmt.Println(numbers2)              // [1 2 3]
```

У наступному прикладі довжина масиву є частиною його типу. І такі два масиви представляють різні типи даних, хоча їх елементи зберігають дані одного типу:

```
var numbers [3]int = [3]int{1, 2, 3}
var numbers2 [4]int = [4]int{1, 2, 3, 4}
numbers = numbers2  // ! Помилка
```

В цьому випадку при присвоюванні згенерується помилка, оскільки дані одного типу не можна передати змінній іншого типу.

2.8.2. Індекси

Для звернення до елементів масиву застосовуються індекси. Індекс це номер елемента. У мові Go нумерація починається з нуля, тобто перший елемент матиме індекс 0. Індекс вказується у квадратних дужках. За індексом можна отримати значення елемента або змінити його, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       var numbers [5]int = [5]int{1,2,3,4,5}
6       fmt.Println(numbers[0])      // 1
7       fmt.Println(numbers[4])      // 5
8       numbers[0] = 87
9       fmt.Println(numbers[0])      // 87
10  }
```

Індекси в масиві фактично виступають у ролі ключа, за яким можна звернутися до відповідного значення. Можна явно вказати, якому ключу яке значення буде відповідати. Числові ключі необов'язково розташовувати в порядку зростання, наприклад:

```
1   colors := [3]string{2: "blue", 0: "red", 1: "green"}
2   fmt.Println(colors[2])       // blue
```

2.9. Умовні конструкції

Умовні конструкції перевіряють істинність деякої умови і в залежності від результатів перевірки дозволяють направити хід програми одним із шляхів.

Розглянемо їх більш детально.

2.9.1. Конструкція if...else

Конструкція **if** приймає умову-вираз, що повертає значення типу **bool**. І якщо ця умова є істинною, то виконується наступний блок інструкцій, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       a := 6
```

```
7       b := 7
8       if a < b {
9           fmt.Println("a менше b")
10      }
11  }
```

Умова розташовується після оператора **if**. У наведеному прикладі перевіряється умова, чи менше значення змінної **a** за значення змінної **b**. Оскільки значення змінної **a** дійсно менше значення змінної **b**, тобто умова повертає **true**, то виконуватиметься наступний блок коду, який виводить на консоль повідомлення.

Якщо необхідно задати альтернативну логіку, яка виконується, якщо умова неправильна, то додається вираз **else**, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       a := 6
7       b := 7
8       if a < b {
9           fmt.Println("a менше b")
10      }else{
11          fmt.Println("a більше b")
12      }
13  }
```

Таким чином, якщо умовний вираз після оператора **if** вірний, то виконується блок після **if**, якщо хибний, то виконується блок після **else**.

Якщо необхідно перевірити кілька альтернативних варіантів, можна додати вирази **else if**, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       a := 8
7       b := 8
8       if a < b {
9           fmt.Println("a менше b")
```

```
10      }else if a > b{
11          fmt.Println("a більше b")
12      } else{
13          fmt.Println("a дорівнює b")
14      }
15  }
```

Таким чином, якщо вираз після **if** вірний, спрацьовує блок **if**. Інакше перевіряється вираз після **else if**. Якщо він є вірним, то виконується блок **else if**. Якщо він хибний, то виконується блок **else**.

Кількість виразів **else if** у мові Go не обмежується. наприклад:

```
1   if a ==9 {
2       fmt.Println("a = 9")
3   }else if a == 8{
4       fmt.Println("a = 8")
5   } else if a == 7{
6       fmt.Println("a == 7")
7   }
```

2.9.2. Конструкція switch

Конструкція **switch** перевіряє значення певного виразу. За допомогою операторів **case** визначаються значення порівняння. Якщо значення після оператора **case** збігається зі значенням виразу зі **switch**, то виконується код даного блоку **case**, наприклад.

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       a := 8
7       switch(a) {
8           case 9:
9               fmt.Println("a = 9")
10          case 8:
11              fmt.Println("a = 8")
12          case 7:
13              fmt.Println("a = 7")
14      }
```

```
15  }
```

У цьому прикладі у якості виразу конструкція **switch** використовує змінну **a**. Її значення послідовно порівнюється зі значеннями після операторів **case**. Оскільки змінна **a** дорівнює 8, то буде виконуватися блок **case 8: fmt.Println("a = 8")**. Інші блоки **case** не виконуються.

При цьому після оператора **switch** можна вказати будь-який вираз, який повертає значення, наприклад, операцію додавання:

```
1   a := 7
2   switch(a + 2) {
3       case 9:
4           fmt.Println("9")
5       case 8:
6           fmt.Println("8")
7       case 7:
8           fmt.Println("7")
9   }
```

Також конструкція **switch** може містити необов'язковий блок **default**, який виконується, якщо жоден з операторів **case** не містить потрібного значення, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       a := 87
7       switch(a) {
8           case 9:
9               fmt.Println("a = 9")
10          case 8:
11              fmt.Println("a = 8")
12          case 7:
13              fmt.Println("a = 7")
14          default:
15              fmt.Println("Значення змінної a не
    визначено")
16      }
17  }
```

Також можна вказувати після оператора **case** відразу кілька значень, наприклад:

```
1   a := 5
2   switch(a) {
3       case 9: fmt.Println("a = 9")
4       case 8: fmt.Println("a = 8")
5       case 7: fmt.Println("a = 7")
6       case 6, 5, 4:
7           fmt.Println("a = 6 або 5 або 4, але це не
    точно ")
8       default:
9           fmt.Println("Значення змінної a не визначено")
10  }
```

2.10. Цикли

Цикли дозволяють залежно від певної умови виконувати деякі дії багато разів.

2.10.1. Визначення циклів

Фактично в мові Go є лише один цикл, а саме цикл **for**, який може набувати різних форм. Цей цикл має наступне формальне визначення:

```
1   for [Ініціалізація лічильника]; [Умова]; [Зміна
    лічильника] {
2       // Дії
3   }
```

Наприклад, виведемо за допомогою циклу квадрати чисел:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       for i := 1; i < 10; i++{
7           fmt.Println(i * i)
8       }
9   }
```

Оголошення циклу **for** розбивається на три частини. Спочатку виконується ініціалізація лічильника **i := 1**. Фактично вона представляє

оголошення змінної, яка використовуватиметься всередині циклу. В наведеному прикладі лічильнику **i** встановлюється початкове значення, яке дорівнює 1.

Друга частина представляє умову **i < 10**. Поки ця умова вірна, тобто повертається **true**, то цикл продовжується.

Третя частина представляє зміну лічильника на одиницю.

У тілі циклу на консоль виводиться квадрат числа **i.**

Таким чином, цикл спрацює 9 разів, поки значення і не стане рівним 10. І щоразу це значення збільшуватиметься на 1. Кожен окремий прохід циклу називається ітерацією. Тобто у наведеному прикладі буде 9 ітерацій. Результат роботи програми наведено на рисунку 2.1.

Рисунок 2.1. Результат роботи програми

Вказувати всі умови під час оголошення циклу необов'язково. Наприклад, можна винести оголошення змінної назовні, а саме:

```
1   var i = 1
2   for ; i < 10; i++{
3       fmt.Println(i * i)
4   }
```

В мові Go можна перенести змінну лічильника в саме тіло циклу і залишити лише умову, наприклад:

```
1   var i = 1
2   for ; i < 10;{
3       fmt.Println(i * i)
4       i++
5   }
```

Якщо цикл використовує лише умову, його можна скоротити наступним чином:

```
1   var i = 1
2   for i < 10{
3       fmt.Println(i * i)
4       i++
5   }
```

2.10.2. Вкладені цикли

Цикли можуть бути вкладеними, тобто розташовуватись усередині інших циклів. Наприклад, виведемо на консоль таблицю множення:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       for i := 1; i < 10; i++{
7           for j := 1; j < 10; j++{
8               fmt.Print(i * j, "\t")
9           }
10          fmt.Println()
11      }
12  }
```

Результат роботи програми наведено на рисунку 2.2.

```
1   2   3   4   5   6   7   8   9
2   4   6   8   10  12  14  16  18
3   6   9   12  15  18  21  24  27
4   8   12  16  20  24  28  32  36
5   10  15  20  25  30  35  40  45
6   12  18  24  30  36  42  48  54
7   14  21  28  35  42  49  56  63
8   16  24  32  40  48  56  64  72
9   18  27  36  45  54  63  72  81
```

Рисунок 2.2. Результат роботи програми

2.10.3. Перебір масивів

Для перебору елементів масивів можна використовувати наступну форму циклу:

```
for Індекс, Значення := range Масив{
    // Дії
}
```

При переборі можна окремо отримати індекс елемента у масиві та значення цього елемента. Наприклад, переберемо масив рядків:

```
var users = [3]string{"Tom", "Alice", "Kate"}
for index, value := range users{
    fmt.Println(index, value)
}
```

Результат роботи програми наведено на рисунку 2.3.

```
0 Tom
1 Alice
2 Kate
```

Рисунок 2.3. Результат роботи програми

Якщо не планується використовувати значення чи індекси елементів, то можна замість них вказати символ нижнього підкреслення (_). Наведемо приклад, де не потрібні індекси:

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

Звичайно, для перебору елементів масиву можна використовувати стандартну версію циклу **for**, наприклад:

```
var users = [3]string{"Tom", "Alice", "Kate"}
for i:= 0; i < len(users); i++{
    fmt.Println(users[i])
}
```

У наведеному прикладі лічильником **i** є індекс масиву. Цикл виконується, доки лічильник **i** не стане рівним довжині масиву, яку можна отримати за допомогою функції **len()**.

2.10.4. Оператори break і continue

Може виникнути ситуація, коли за певних умов необхідно завершити поточну ітерацію циклу і не виконувати всі інструкції циклу, а одразу перейти до наступної ітерації. У цьому випадку можна використовувати оператор **continue**. Наприклад, у масиві можуть бути як позитивні, так і негативні числа. Припустимо, необхідно знайти суму лише позитивних чисел, тому, якщо

зустрінеться негативне число, можна просто перейти до наступної ітерації за допомогою **continue**, наприклад:

```
1   var numbers = [10]int{1, -2, 3, -4, 5, -6, -7, 8, -9,
    10}
2   var sum = 0
3
4   for _, value := range numbers{
5       if value < 0{
6           continue        // Перехід до наступної
    ітерації
7       }
8       sum += value
9   }
10  fmt.Println("Sum:", sum)     // Sum: 27
```

Оператор **break** повністю здійснює вихід із циклу, наприклад:

```
1   var numbers = [10]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
2   var sum = 0
3
4   for _, value := range numbers{
5       if value > 4{
6           break       // Якщо число більше 4, то виходимо
    з циклу
7       }
8       sum += value
9   }
10  fmt.Println("Sum:", sum)     // Sum: 10
```

2.11. Функції та їх параметри

Функція в мові Go дозволяє «обгорнути» послідовність операторів у єдине ціле для виконання якогось певного завдання. За допомогою функцій можна повторно викликати її блок операторів як єдине ціле в інших частинах програми.

Функція оголошується наступним чином:

```
1   func Ім'я_функції (Список_параметрів)
    (Типи_значень_що_повертаються){
2       Виконувані_оператори
3   }
```

Функція визначається за допомогою ключового слова **func**, після якого вказується ім'я функції. Потім у дужках розташовується список параметрів. Після списку параметрів визначаються типи значень, які повертаються з функції, якщо функція повертає значення. І далі у фігурних дужках вказуються ті оператори, з яких складається функція.

Назва функції разом із типами її параметрів і типами значень, що повертаються, ще називають сигнатурою.

За замовчуванням кожна програма на мові Go повинна містити щонайменше одну функцію. Це функція **main**, яка є вхідною точкою додатку, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5       fmt.Println("Hello Go")
6   }
```

Функція **main** починається з ключового слова **func**, потім вказується назва **main**. Функція не приймає жодних параметрів, тому після назви записані порожні дужки. Функція **main** не повертає жодного результату, тому після порожніх дужок не вказується тип значення, що повертається. Тіло функції у фігурних дужках фактично складається із виклику іншої функції, а саме **fmt.Println()**, яка виводить рядок на консоль.

Визначимо ще одну функцію наступним чином:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6   }
7
8   func hello(){
9       fmt.Println("Hello World")
10  }
```

У цьому прикладі визначено функцію **hello**, яка не приймає жодних параметрів, нічого не повертає та просто виводить на консоль рядок. Визначити функцію можна у тому самому файлі, де розташована функція **main**. Але якщо запустити цю програму, то на консолі нічого не виведеться. Тому що програма виконує ті дії, які визначені всередині функції **main**. Саме функція **main** є

вхідною точкою додатку. І якщо необхідно виконати у програмі функцію **hello**, то функцію **hello** треба викликати у функції **main**, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5       hello()
6       hello()
7       hello()
8   }
9
10  func hello(){
11      fmt.Println("Hello World")
12  }
```

Для виклику функції вказується її ім'я, після якого в дужках вказуються значення параметрів функції. Але оскільки в наведеному прикладі функція **hello** не має параметрів, після її назви записуються порожні дужки. І таким чином, при виконанні програми на консоль тричі буде виведено рядок "Hello World". Результат роботи програми наведено на рисунку 2.4.

```
Hello World
Hello World
Hello World
```

Рисунок 2.4. Результат роботи програми

Таким чином, оформлення групи операторів у вигляді функції дозволяє не писати щоразу всю групу операторів, а звертатися до них за ім'ям функції.

2.11.1. Параметри функції

Через параметри функція отримує вхідні дані. Параметри вказуються у дужках після імені функції. Для кожного параметра вказується ім'я та тип, як для змінної. Один від одного параметри розділяються комами. Під час виклику функції необхідно передати значення для всіх її параметрів. Наприклад, створимо функцію, яка додає два будь-які числа:

```
1   package main
2   import "fmt"
3
```

```
4   func main() {
5       add(4, 5)    // x + y = 9
6       add(20, 6)   // x + y = 26
7   }
8
9   func add(x int, y int){
10      var z = x + y
11      fmt.Println("x + y = ", z)
12  }
```

В наведеному прикладі функція **add** має два параметри: **x** та **y**. Обидва параметри мають тип **int**, тобто цілі числа. У самій функції визначається змінна, що зберігає суму цих чисел. І потім сума чисел виводиться на консоль.

Далі у функції **main** викликається функція **add**. Так як функція приймає два параметри, то при виклику їй необхідно передати значення цих параметрів або два аргументи. Причому ці значення повинні відповідати параметрам типу. Тобто якщо параметр представляє тип **int**, йому необхідно передати число з типом **int**.

Значення передаються за позицією. Тобто перше значення отримає перший параметр, друге значення отримає другий вхідний параметр і т.д. Результат роботи програми наведено на рисунку 2.5.

```
x + y = 9
x + y = 26
```

Рисунок 2.5. Результат роботи програми

Якщо кілька параметрів поспіль мають один і той же тип, то можна вказати тип тільки для останнього параметра, тоді попередні параметри також представлятимуть цей же тип, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       add(1, 2, 3.4, 5.6, 1.2)
6   }
7   func add(x, y int, a, b, c float32){
8       var z = x + y
9       var d = a + b + c
10      fmt.Println("x + y = ", z)
```

```
11      fmt.Println("a + b + c = ", d)
12  }
```

Як аргументи під час виклику функції можна передавати значення змінних, результати операцій чи інших функцій, але при цьому слід враховувати, що аргументи в функцію завжди передаються за значенням, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5       var a = 8
6       fmt.Println("a before: ", a)
7       increment(a)
8       fmt.Println("a after: ", a)
9   }
10  func increment(x int){
11
12      fmt.Println("x before: ", x)
13      x = x + 20
14      fmt.Println("x after: ", x)
15  }
```

Результат роботи програми наведено на рисунку 2.6.

```
a before: 8
x before: 8
x after: 28
a after: 8
```

Рисунок 2.6. Результат роботи програми

В наведеному прикладі як аргумент у функцію **increment** передається значення змінної **a**. Параметр **x** отримує це значення, і воно збільшується на 20. Однак незважаючи на те, що значення параметра **x** збільшилося, значення змінної **a** ніяк не змінилося. Тому що під час виклику функції передається копія значення змінної.

2.11.2. Невизначена кількість параметрів

У мові Go функція може приймати невизначену кількість параметрів одного типу. Наприклад, необхідно отримати суму чисел, але точної кількості

чисел, які будуть передані до функції наперед не відомо. Тоді код програми виглядатиме наступним чином:

```
1   package main
2   import "fmt"
3
4   func main() {
5       add(1, 2, 3)          // sum = 6
6       add(1, 2, 3, 4)       // sum = 10
7       add(5, 6, 7, 2, 3)    // sum = 23
8   }
9
10  func add(numbers ...int){
11      var sum = 0
12      for _, number := range numbers{
13          sum += number
14      }
15      fmt.Println("sum = ", sum)
16  }
```

В наведеному прикладі для визначення параметра, який представляє невизначену кількість значень, перед типом цих значень ставиться символ трикрапки (**…**), а саме **numbers ...int**. Тобто через такий параметр можна отримати декілька значень типу **int**.

Під час виклику функції **add** стає можливим передати в неї різну кількість чисел, наприклад:

```
1   add(1, 2, 3)          // sum = 6
2   add(1, 2, 3, 4)       // sum = 10
3   add(5, 6, 7, 2, 3)    // sum = 23
```

Варто наголосити, шо в такому разі слід відрізняти передачу зрізу[1] як параметр, а саме:

```
1   add([]int{1, 2, 3})
2   add([]int{1, 2, 3, 4})
3   add([]int{5, 6, 7, 2, 3})
```

В наведеному прикладі згенерується помилка, оскільки передача зрізу не еквівалентна передачі невизначеної кількості параметрів того самого типу.

[1] Детальніше про зрізи буде пояснено в підрозділі 2.17. Зрізи.

Якщо необхідно передати зріз, то необхідно після аргументу-масиву вказати символ трикрапки (**...**), наприклад:

```
add([]int{1, 2, 3}...)

add([]int{1, 2, 3, 4}...)

var nums = []int{5, 6, 7, 2, 3}
add(nums...)
```

2.12. Повернення результату з функції

Функції можуть повертати результат. Для цього необхідно після списку параметрів функції вказати тип результату, що повертається. А в самому тілі функції використати оператор **return**, після якого вказати значення, що повертається, а саме:

```
func ім'я_функції (Список_параметрів)
тип_значення_що_повертається {
    Виконувані_оператори
    return Значення,що повертається
}
```

Розглянемо приклад повернення з функції суми двох чисел, а саме:

```
package main
import "fmt"

func main() {
    var a = add(4, 5)    // 9
    var b = add(20, 6)   // 26
    fmt.Println(a)
    fmt.Println(b)
}

func add(x, y int) int {
    return x + y
}
```

В наведеному прикладі функція **add** повертає значення типу **int**, тому цей тип вказано після переліку параметрів. У самій функції після оператора **return** вказується значення, що повертається. При цьому це значення може бути значенням змінної, літералом, або ж, як в наведеному прикладі, результатом

операції або виклику функції. Тобто вираз **x + y** визначає значення, що буде повернене.

Оскільки функція повертає значення, то при виклику функції можна отримати це значення та передати його змінній, наприклад:

```
1   var a = add(4, 5)    // 9
2   var b = add(20, 6)   // 26
```

2.12.1. Іменовані результати, що повертаються

Результат, що повертається, може бути іменованим, наприклад:

```
1   func add(x, y int) (z int) {
2       z = x + y
3       return
4   }
```

В наведеному прикладі у дужках після списку параметрів фактично визначається змінна, значення якої повертатиметься. Наприкінці функції ставиться оператор **return**, але тепер необов'язково після цього оператора ставити значення, що повертається. Фактично також можна було б написати наступний код:

```
1   func add(x, y int) int {
2       var z int = x + y
3       return z
4   }
```

2.12.2. Повернення кількох значень

У мові Go функція може повертати відразу кілька значень. У цьому випадку після списку параметрів вказується в дужках список типів значень, що повертаються. А після оператора **return** через кому розташовуються всі значення, що повертаються, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       var age, name = add(4, 5, "Tom", "Simpson")
6       fmt.Println(age)     // 9
7       fmt.Println(name)    // Tom Simpson
8   }
9
```

```
10  func add(x, y int, firstName, lastName string) (int ,
    string) {
11      var z int = x + y
12      var fullName = firstName + " " + lastName
13      return z, fullName
14  }
```

В наведеному прикладі функція **add** приймає чотири параметри, а саме: два числа та два рядки. Повертає функція **add** число (значення типу **int**) та рядок. Значення, що повертаються вказані після оператора **return**.

Оскільки функція тепер повертає два значення, то при виклику цієї функції можна надати її результат двом різним змінним, наприклад:

```
1   var age, name = add(4, 5, "Tom", "Simpson")
```

В цьому прикладі перше значення, що повертається, передається першій змінній **age**, а друге значення передається другій змінній name.

Існує і альтернативний спосіб передачі змінним результатів функції, а саме:

```
1   age, name := add(4, 5, "Tom", "Simpson")
```

Також можна використовувати іменовані результати, а саме:

```
1   func add(x, y int, firstName, lastName string) (z
    int, fullName string) {
2       z = x + y
3       fullName = firstName + " " + lastName
4       return
5   }
```

2.13. Тип функції

Кожна функція має певний тип, який складається зі списку типів параметрів та списку типів результатів, що повертається. Наприклад, візьмемо наступну функцію:

```
1   func add(x int, y int) int{
2       return x + y
3   }
```

Ця функція має тип **func(int, int) int**. Цьому ж типу відповідатиме і наступна функція:

```
1   func multiply(x int, y int) int{
2       return x * y
3   }
```

Хоча ця функція називається інакше, виконує інші дії, але за типом параметрів і за типом результату, що повертається вона відповідає вище зазначеному типу функції.

Розглянемо ще одну функцію, а саме:

```
1   func display(message string){
2       fmt.Println(message)
3   }
```

Ця функція має тип **func(string)**. Тобто знову ж таки спочатку вказано слово **func**, а потім у дужках типи параметрів. Так як функція не повертає ніякого результату, то тип, що повертається, не вказується.

Це означає, що можна визначати змінні або параметри функцій, які представлятимуть певний тип функції. Тобто фактично змінна може бути функцією, наприклад:

```
1   package main
2
3   import "fmt"
4
5   func add(x int, y int) int{
6       return x + y
7   }
8
9   func main() {
10
11      var f func(int, int) int = add
12      fmt.Println(f(3, 4))
13
14      var x = f(4, 5)      // 9
15      fmt.Println(x)
16  }
```

В наведеному прикладі змінна **f** має тип **func(int, int) int**, тобто представляє будь-яку функцію, яка має два параметри типу **int** і повертає значення типу **int**. Тому можна присвоїти цій змінній функцію **add**, яка відповідає даному типу, а саме:

```
1   var f func(int, int) int = add
```

Після цього можна викликати функцію через змінну, передаючи потрібні значення для її параметрів, наприклад:

```
1   var x = f(4, 5) // 9
```

При цьому змінна може змінювати функцію, яку вона вказує, але функція обов'язково повинна відповідати її типу, а саме:

```
1   package main
2
3   import "fmt"
4
5   func add(x int, y int) int{ return x + y}
6   func multiply(x int, y int) int{ return x * y}
7
8   func display(message string){
9       fmt.Println(message)
10  }
11
12  func main() {
13
14      f := add         //Або так var f func(int, int) int
    = add
15      fmt.Println(f(3, 4))         // 7
16
17      f = multiply    // Тепер змінна f вказує на
    функцію multiply
18      fmt.Println(f(3, 4))         // 12
19
20      // f = display       // Помилка, оскільки функція
    display має тип func(string)
21
22      var f1 func(string) = display   // Ок
23      f1("hello")
24  }
```

2.13.1. Функції, як інші параметри

У мові Go функція може передаватися як параметр до іншої функції, наприклад:

```
1   package main
2
3   import "fmt"
4
```

```
5   func add(x int, y int) int {
6
7       return x + y
8   }
9   func multiply(x int, y int) int {
10
11      return x * y
12  }
13  func action(n1 int, n2 int, operation func(int, int)
    int){
14
15      result := operation(n1, n2)
16      fmt.Println(result)
17  }
18  func main() {
19
20      action(10, 25, add)      // 35
21      action(5, 6, multiply)   // 30
22  }
```

В цьому прикладі функція **action** приймає три параметри. Перші два параметри це числа, а третій параметр це функція, яка відповідає типу **func(int , int) int**. Тобто третій параметр представляє деяку дію і може бути представлений будь-якою функцією, яка отримує два значення типу **int** і повертає також значення типу **int**. Для прикладу тут якраз визначено дві подібні функції, які відповідають даному типу, а саме: **add** та **multiply**. Через ім'я параметра **operation** можна викликати цю функцію.

Розглянемо ще один приклад, а саме:

```
1   package main
2
3   import "fmt"
4
5   func isEven(n int) bool{
6       return n % 2 == 0
7   }
8   func isPositive(n int) bool{
9       return n > 0
10  }
11
```

```
12 func sum(numbers []int, criteria func(int) bool) int{
13
14     result := 0
15     for _, val := range numbers{
16         if(criteria(val)){
17             result += val
18         }    }
19     return result
20 }
21 func main() {
22
23     slice := []int{-2, 4, 3, -1, 7, -4, 23}
24
25     sumOfEvens := sum(slice, isEven)     // Сума парних
26 чисел
       fmt.Println(sumOfEvens)              // -2
27
28     sumOfPositives := sum(slice, isPositive)     //
29 Сума додатних чисел
       fmt.Println(sumOfPositives)                  // 37
30 }
31
```

В наведеному прикладі функція **sum** обчислює суму елементів зрізу. Але не всіх елементів, а лише тих, що відповідають умові. Умова передається у вигляді функції як другий параметр. Умова повинна відповідати функціям типу **func(int) bool**. Тобто функція повинна приймати як параметр значення типу **int** і повертати значення типу **bool**, яке вказує, чи відповідає передане число умові.

У прикладі також визначено дві допоміжні функції, а саме **isEven**, яка повертає **true**, якщо число парне та **isPositive**, яке повертає **true**, якщо число додатне. Ці функції відповідають типу **func(int) bool**, тому їх можна використовувати як умову.

2.13.2. Функція як результат іншої функції

Функція також може повертатися з іншої функції як результат, наприклад:

```
1  package main
2
3  import "fmt"
```

```
4
5   func add(x int, y int) int{ return x + y}
6   func subtract(x int, y int) int{ return x - y}
7   func multiply(x int, y int) int{ return x * y}
8
9   func selectFn(n int) (func(int, int) int){
10      if n==1 {
11          return add
12      }else if n==2{
13          return subtract
14      }else{
15          return multiply
16      }
17  }
18
19  func main() {
20
21      f := selectFn(1)
22      fmt.Println(f(3, 4))        // 7
23
24      f = selectFn(3)
25      fmt.Println(f(3, 4))        // 12
26  }
```

В наведеному прикладі, залежно від значення параметра, функція **selectFn** повертає одну з трьох функцій, а саме: **add**, **subtract** або **multiply**.

2.14. Анонімні функції

Анонімні функції – це функції, яким не призначено ім'я. Вони відрізняються від звичайних функцій тим, що вони можуть визначатися всередині інших функцій і також можуть мати доступ до контексту виконання. Анонімна функція також відома як функціональний літерал. Формальне визначення анонімної функції виглядає наступним чином:

```
1   func(список_параметрів)(тип_повернення) {
2    // Код...
3    // Використовуйте оператор return, якщо задано
    тип_повернення
4    // Якщо тип_повернення не вказано, тоді не
```

```
    вказується
5   return
6   }()
```

Анонімні функції дозволяють визначити певну дію безпосередньо там, де вона застосовується. Наприклад, необхідно виконати додавання двох чисел, але більше ніде ця дія в програмі не потрібна:

```
1   package main
2
3   import "fmt"
4
5   func main() {
6
7       f := func(x, y int) int{ return x + y}
8       fmt.Println(f(3, 4))          // 7
9       fmt.Println(f(6, 7))          // 13
10  }
```

Фактично змінна **f** визначена як:

```
1   var f func(int, int) int = func(x, y int) int{ return
    x + y}
```

Тобто змінній **f** можна присвоїти будь-яку функцію, яка відповідає типу **func(int, int) int**.

2.14.1. Анонімна функція як аргумент функції

В мові Go дуже зручно використовувати анонімні функції як аргументи інших функцій, наприклад:

```
1   package main
2
3   import "fmt"
4
5   func action(n1 int, n2 int, operation func(int, int)
    int){
6
7       result := operation(n1, n2)
8       fmt.Println(result)
9   }
10  func main() {
11
```

```
12      action(10, 25, func (x int, y int) int { return x
    + y })    // 35
13      action(5, 6, func (x int, y int) int { return x *
    y })      // 30
14  }
```

2.14.2. Анонімна функція як результат функції

У мові Go анонімна функція може бути результатом іншої функції, наприклад:

```
1   package main
2
3   import "fmt"
4
5   func selectFn(n int) (func(int, int) int){
6       if n==1 {
7           return func(x int, y int) int{ return x + y}
8       }else if n==2{
9           return func(x int, y int) int{ return x - y}
10      }else{
11          return func(x int, y int) int{ return x * y}
12      }
13  }
14
15  func main() {
16
17      f := selectFn(1)
18      fmt.Println(f(2, 3))          // 5
19      fmt.Println(f(4, 5))          // 9
20      fmt.Println(f(7, 6))          // 13
21  }
```

2.14.3. Анонімна функція без попереднього зв’язування

У мові Go анонімна функція може бути визначена і застосована без попереднього зв’язування зі змінною, тобто на місці, наприклад:

```
1   package main
2
3   import "fmt"
```

```
4
5       func(a, b, c string) {
6           fmt.Println(a+b+c)
7       }("Hello", " ", "World!")
8   }
```

2.14.4. Доступ до оточення

Перевагою анонімних функцій є те, що вони мають доступ до оточення, у якому визначаються, наприклад:

```
1   package main
2
3   import "fmt"
4
5   func square() func() int {
6       var x int = 2
7       return func() int {
8           x++
9           return x * x
10      }
11  }
12
13  func main() {
14
15      f := square()
16      fmt.Println(f())        // 9
17      fmt.Println(f())        // 16
18      fmt.Println(f())        // 25
19  }
```

В цьому прикладі функція **square** визначає локальну змінну **x** та повертає анонімну функцію. Анонімна функція збільшує значення змінної **x** та повертає її квадрат.

Таким чином, можна зафіксувати у зовнішній функції **square** стан у вигляді змінної **x**, який змінюватиметься в анонімній функції.

2.15. Рекурсивні функції

Рекурсивна функція представляє таку функцію, яка викликає себе. Рекурсивні функції представляють потужний інструмент обробки рекурсивних структур даних, наприклад, різних дерев.

Наприклад, визначимо функцію обчислення факторіала числа, що отримує результат рекурсивним способом:

```
1   package main
2
3   import "fmt"
4
5   func factorial(n uint) uint{
6
7       if n == 0{
8           return 1
9       }
10      return n * factorial(n - 1)
11  }
12  func main() {
13
14      fmt.Println(factorial(4))    // 24
15      fmt.Println(factorial(5))    // 120
16      fmt.Println(factorial(6))    // 720
17  }
```

В цьому прикладі функція **factorial** отримує деяке позитивне число, для якого необхідно обчислити факторіал. Отриманий результат повертається із функції. Спочатку вказується умова, що коли число дорівнює 0, то функція повертає 1. Інакше функція повертає добуток числа **n** на результат цієї функції для числа **n-1**.

При створенні рекурсивної функції у ній обов'язково має бути деякий базовий варіант, який використовує оператор **return** і розміщується на початку функції. У наведеному прикладі для факторіалу це **if x == 0 {return 1}**.

Крім цього, всі рекурсивні виклики повинні звертатися до підфункцій, які зрештою сходяться до базового варіанта. Так, при передачі в функцію позитивного числа при подальших рекурсивних викликах підфункцій у них передаватиметься щоразу число, менше на одиницю. І зрештою алгоритм дійде до ситуації, коли число дорівнюватиме 0, та буде використаний базовий варіант.

Наприклад, виклик **factorial(4)** фактично можна записати так, як показано на рисунку 2.7.

```
factorial(4)
4 * factorial(3)
4 * 3 * factorial(2)
4 * 3 * 2 * factorial(1)
4 * 3 * 2 * 1 * factorial(0)
4*3*2*1*1
```

Рисунок 2.7. Результат роботи програми

Іншим поширеним показовим прикладом рекурсивної функції є функція, що обчислює числа Фібоначчі. Член послідовності Фібоначчі з індексом **n** визначається за формулою **f(n)=f(n-1) + f(n-2)**, причому **f(0)=0, а f(1)=1**. Тоді програма виглядатиме наступним чином:

```
1   package main
2
3   import "fmt"
4
5   func fibbonachi(n uint) uint{
6
7       if n == 0 {
8           return 0
9       }
10      if n == 1 {
11          return 1
12      }
13      return fibbonachi(n - 1) + fibbonachi(n - 2)
14  }
15  func main() {
16
17      fmt.Println(fibbonachi(4))   // 3
18      fmt.Println(fibbonachi(5))   // 5
19      fmt.Println(fibbonachi(6))   // 8
20  }
```

2.16. Оператори defer та panic

2.16.1. Оператор defer

Оператор **defer** дозволяє виконати певну функцію в кінці програми, при цьому не важливо, де насправді викликається ця функція, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       defer finish()
6       fmt.Println("Program has been started")
7       fmt.Println("Program is working")
8   }
9
10  func finish(){
11      fmt.Println("Program has been finished")
12  }
```

В цьому прикладі функція **finish** викликається з оператором **defer**, тому ця функція насправді буде викликатися наприкінці виконання програми, незважаючи на те, що її виклик визначено на початку функції **main**. Результат роботи програми наведено на рис. 2.8.

```
Program has been started
Program is working
Program has been finished
```

Рисунок 2.8. Результат роботи програми

Якщо кілька функцій викликаються з оператором **defer**, то ті функції, які викликаються раніше, виконуватимуться пізніше за всіх, наприклад:

```
1   package main
2   import  "fmt"
3
4   func main() {
5
6       defer finish()
7       defer fmt.Println("Program has been started")
8       fmt.Println("Program is working")
```

```
9   }
10
11  func finish(){
12      fmt.Println("Program has been finished")
13  }
```

Результат роботи програми наведено на рисунку 2.9.

```
Program is working
Program has been started
Program has been finished
```

Рисунок 2.9. Результат роботи програми

2.16.2. Оператор panic

Оператор **panic** дозволяє згенерувати помилку та вийти з програми, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5       fmt.Println(divide(15, 5))
6       fmt.Println(divide(4, 0))
7       fmt.Println("Program has been finished")
8   }
9   func divide(x, y float64) float64{
10      if y == 0{
11          panic("Division by zero!")
12      }
13      return x / y
14  }
```

Оператору **panic** можна передати будь-яке повідомлення, яке виводитиметься на консоль. В наведеному прикладі якщо другий параметр функції **divide** дорівнюватиме 0, то здійснюватиметься виклик **panic("Division by zero!")**.

У функції **main** у виклику **fmt.Println(divide(4, 0))** буде виконуватися оператор **panic**, оскільки другий параметр функції **divide** дорівнює 0. І в цьому випадку всі наступні операції, які вказані після цього виклику, наприклад, у цьому випадку це виклик **fmt.Println("Program has been finished")**, не будуть виконуватися. Результат роботи програми наведено на рисунку 2.10.

```
3
panic: Division by zero!
```

Рисунок 2.10. Результат роботи програми

Наприкінці виводиметься діагностична інформація про те, де саме виникла помилка.

2.17. Зрізи

У мові Go зрізи (**slice**) це послідовність елементів однакового типу змінної довжини. На відміну від масивів, довжина в зрізах не фіксована і динамічно може змінюватися, тобто можна додавати нові елементи або видаляти вже існуючі.

Зріз визначається так само, як і масив, за винятком, що в ньому не вказується довжина, а саме:

```
var users []string
```

Також зріз можна ініціалізувати значеннями, наприклад:

```
var users = []string{"Tom", "Alice", "Kate"}
// Або так
users2 := []string{"Tom", "Alice", "Kate"}
```

Звернення до елементів зрізу відбувається так само, як і до елементів масиву, а саме за індексом. Елементи зрізу можна перебирати за допомогою циклу **for**, наприклад:

```
var users []string = []string{"Tom", "Alice", "Kate"}
fmt.Println(users[2])   // Kate
users[2] = "Katherine"

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

За допомогою функції **make()** можна створити зріз із кількох елементів, які будуть мати значення за замовчуванням, наприклад:

```
var users []string = make([]string, 3)
users[0] = "Tom"
users[1] = "Alice"
users[2] = "Bob"
```

2.17.1. Додавання до зрізу

Для додавання до зрізу застосовується вбудована функція **append (slice, value)**. Перший параметр функції це зріз, у який треба додати, а другий параметр це значення, яке потрібно додати. Результатом функції є збільшений зріз, наприклад.

```
1   users := []string{"Tom", "Alice", "Kate"}
2   users = append(users, "Bob")
3
4   for _, value := range users{
5       fmt.Println(value)
6   }
```

2.17.2. Оператор зрізу

Оператор зрізу **s[i:j]** створює з послідовності **s** новий зріз, який містить елементи послідовності **s** з **i** по **j-1**. При цьому повинна дотримуватися умова **0 <= i <= j <= cap(s)**, де **cap()** є вбудованою функцією, за допомогою якої можна отримати кількість елементів у послідовності. Як вихідна послідовність, з якої беруться елементи, може використовуватися масив, покажчик на масив або інший зріз. У результаті в отриманому зрізі буде **j-i** елементів.

Якщо значення **i** не вказано, то застосовується за замовчуванням значення 0. Якщо значення **j** не вказано, замість нього використовується довжина вихідної послідовності **s**, наприклад:

```
1   // Вихідний масив
2   initialUsers := [8]string{"Bob", "Alice", "Kate",
    "Sam", "Tom", "Paul", "Mike", "Robert"}
3   users1 := initialUsers[2:6]     // з 3-го по 6-й
4   users2 := initialUsers[:4]      // з 1-го по 4-й
5   users3 := initialUsers[3:]      // з 4-го до кінця
6
7   fmt.Println(users1)     // ["Kate", "Sam", "Tom",
    "Paul"]
8   fmt.Println(users2)     // ["Bob", "Alice", "Kate",
    "Sam"]
9   fmt.Println(users3)     // ["Sam", "Tom", "Paul",
    "Mike", "Robert"]
```

2.17.3. Видалення елемента

Для видалення певного елемента, можна комбінувати функцію **append** та оператор зрізу, наприклад:

```
users := []string{"Bob", "Alice", "Kate", "Sam",
"Tom", "Paul", "Mike", "Robert"}
//Видаляємо 4-й елемент
var n = 3
users = append(users[:n], users[n+1:]...)
fmt.Println(users)      //["Bob", "Alice", "Kate",
"Tom", "Paul", "Mike", "Robert"]
```

2.18. Відображення map

Відображення або **map** представляє посилання на хеш-таблицю (структуру даних), де кожен елемент представляє пару «ключ-значення». У **map** кожен елемент має унікальний ключ, за яким можна отримати значення елемента. Відображення визначається як об'єкт типу **map[K]V**, де **K** представляє тип ключа, а **V** - тип значення. Причому тип ключа **K** повинен підтримувати операцію порівняння (==), щоб відображення могло зіставити значення з одним із ключів з хеш-таблиці.

Розглянемо приклад, визначення відображення, яке має ключі типу **string**, а значення типу **int**:

```
var people map[string]int   // Ключі представляють тип
string, значення - тип int
```

Встановлення значень відображення:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
fmt.Println(people)      // map[Tom:1 Bob:2 Sam:4
Alice:8]
```

Як і в масиві або в зрізі елементи розташовують у фігурних дужках. Кожен елемент представляє пару ключ-значення. Спочатку вказується ключ і через двокрапку значення. Визначення елемента завершується комою.

2.18.1. Основні операції з map

Для звернення до елементів слід застосовувати ключі, наприклад:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
fmt.Println(people["Alice"])        // 8
fmt.Println(people["Bob"])          // 2
people["Bob"] = 32
fmt.Println(people["Bob"])          // 32
```

Для перевірки наявності елемента за певним ключем можна застосовувати вираз **if**, наприклад:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
if val, ok := people["Tom"]; ok{
    fmt.Println(val)
}
```

Якщо значення за заданим ключем є у відображенні, то змінна **ok** дорівнюватиме **true**, а змінна **val** буде зберігати отримане значення. Якщо змінна **ok** дорівнює **false**, то значення ключа у відображенні немає.

Для перебору елементів застосовується цикл, наприклад:

```
var people = map[string]int{
    "Tom": 1,
    "Bob": 2,
    "Sam": 4,
    "Alice": 8,
}
for key, value := range people{
    fmt.Println(key, value)
}
```

Результат роботи програми наведено на рисунку 2.11

```
Tom 1
Bob 2
Sam 4
Alice 8
```

Рисунок 2.11. Результат роботи програми

Функція **make** є альтернативним варіантом створення відображення. Вона створює порожню хеш-таблицю і формально визначається наступним чином:

```
people := make(map[string]int)
```

2.18.2. Додавання та видалення елементів

Для додавання елементів досить просто виконати ініціалізацію нового значення за новим ключем і елемент із цим ключем буде доданий до колекції, наприклад:

```
var people = map[string]int{ "Tom": 1, "Bob": 2}
people["Kate"] = 128
fmt.Println(people)      // map[Tom:1  Bob:2
Kate:128]
```

Для видалення елементів використовується вбудована функція **delete(map, key)**, першим параметром якої є відображення, а другим є ключ, за яким треба видалити елемент, наприклад:

```
var people = map[string]int{ "Tom": 1, "Bob": 2,
"Sam": 8}
delete(people, "Bob")
fmt.Println(people)      // map[Tom:1  Sam:8]
```

Резюме

Матеріал розділу присвячено основам програмування мовою Go. Зокрема, описано структуру програм, що складається з оголошення пакета, імпорту бібліотек, визначення функцій і змінних. Головним пакетом є **main**, у якому обов'язково повинна бути функція **main**, яка слугує точкою входу програми. Розкрито різні типи даних, включаючи цілочислові, дробові, логічні, рядкові і зазначено їхні значення за замовчуванням. Наведено способи оголошення змінних, ініціалізації, скороченого оголошення та роботи з константами. Розглянуто основні арифметичні операції, умовні конструкції (**if...else**, **switch**) та цикли (**for**), включаючи вкладені і з використанням операторів **break** і

continue. Також пояснено особливості роботи з масивами, їх ініціалізацією, перебором елементів і доступом за індексами. Пояснюється, як оголошуються та викликаються функції, зокрема стандартна функція main, яка є точкою входу для програми. Описано, як передавати параметри функціям, працювати з іменованими та анонімними функціями, а також з функціями, які приймають або повертають інші функції. Матеріал також охоплює концепцію рекурсії, приклади роботи з операторами **defer** та **panic**, які забезпечують управління завершенням програми та обробку помилок. Крім цього, розглядається використання зрізів (**slice**) та відображень (**map**) для роботи з динамічними структурами даних. Усі концепції супроводжуються прикладами, що демонструють їх практичне застосування.

Наступні розділи будуть продовжувати послідовно висвітлювати все більш складні і важливі аспекти використання мови Go для програмування як вебзастосунків так і застосунків загального призначення.

Контрольні запитання і завдання

1. Що таке пакет у мові Go, і як його оголосити у програмі?
2. Як оголошується змінна за допомогою ключового слова **var**? Наведіть приклад.
3. Які типи використовуються для представлення чисел із рухомою крапкою?
4. Яка різниця між операторами **&&** та **||** ?
5. Що повертає оператор **%** і для яких типів даних він використовується?
6. Як працює конструкція **if...else**? Наведіть приклад із використанням декількох умов.
7. Для чого використовується блок **default** у конструкції **switch**?
8. Як оголошується цикл **for** у Go? Які його складові?
9. Чим відрізняються оператори **break** і **continue**?
10. Як оголосити масив у Go і надати йому початкові значення?
11. Як працює оператор зсуву << ? Наведіть приклад.
12. Як перебрати всі елементи масиву за допомогою циклу **range**?
13. У чому різниця між передачею масиву як параметра та використанням трикрапки (**...**) для передачі параметрів у функцію?
14. Який тип має функція **func add(x int, y int) int**?
15. Як оператор **defer** впливає на порядок виконання функцій?
16. Яка різниця між операторами **defer** і **panic**?

17. Як створити зріз у Go і додати до нього елементи за допомогою функції **append**?
18. Що робить оператор зрізу **s[i:j]** і які обмеження застосовуються до його індексів?
19. Що таке **map** у Go, і як його ініціалізувати?
20. Як перевірити, чи є в **map** елемент за певним ключем?
21. Які типи даних можуть використовуватися як ключі у **map**?
22. Додайте в програму імпорт пакета **math** і використайте функцію для обчислення квадратного кореня числа.
23. Напишіть програму, яка запитує температуру в градусах Цельсія. Якщо температура менше 0, виведіть «Холодно»; якщо від 0 до 20 – «Прохолодно»; якщо більше 20 – «Тепло».
24. За допомогою вкладених циклів створіть програму, яка виведе таблицю множення (від 1 до 10).
25. Створіть функцію **add(x int, y int) int**, яка приймає два цілі числа і повертає їхню суму. Використайте її в функції **main** з кількома прикладами.
26. Напишіть функцію **splitSumAndProduct(x int, y int) (int, int)**, яка повертає одночасно суму та добуток двох чисел. Використайте її в **main** для виведення результатів.
27. Напишіть функцію **applyOperation(a, b int, operation func(int, int) int) int**, яка приймає два числа та функцію-операцію (додавання, віднімання чи множення). Використайте її з кількома операціями.
28. Створіть функцію **filter(numbers []int, criteria func(int) bool) []int**, яка повертає лише ті числа, що відповідають умові (наприклад, парні або додатні). Використайте її з різними умовами.
29. Створіть функцію для підрахунку частоти слів у рядка:
    ```
    func wordCount(sentence string) map[string]int
    ```
    Використайте її для підрахунку слів у тексті "**go is fun, go is powerful**".
30. Реалізуйте функцію **fibonacci(n uint) uint**, яка повертає **n**-не число Фібоначчі використовуючи рекурсію. Викличте її з числами 6 і 10 та виведіть результати.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яке ключове слово використовується для оголошення пакета в Go?

a) package

b) import

c) func

d) module

Правильна відповідь: a).

Яке ключове слово використовується для оголошення функції в Go?

a) func

b) function

c) define

d) declare

Правильна відповідь: a).

Закриті запитання з кількома правильними відповідями

Які з наведених операторів використовуються для роботи з умовами в Go?

a) ==

b) !=

c) <

d) <<

Правильна відповідь: a), b), c).

Які функції правильно оголошені в Go?

a) func hello() {}

b) function greet() {}

c) func add(a int, b int) int { return a + b }

d) func subtract(a int, b int): int { return a - b }

Правильна відповідь: a), c).

Завдання на доповнення коду

Заповніть пропуск у коді, щоб програма вивела "Hello, Go!":

```
package main
import "fmt"
```

```
func main() {
    _________ ("Hello, Go!")
}
```

a) print

b) fmt.Println

c) echo

d) fmt.Output

Правильна відповідь: b).

Заповніть пропуски в коді для створення мапи з початковими значеннями:

```
people := map[string]int{
    "Alice": _____________,
    "Bob": 35,
    "Tom": _____________,
}
```

Правильна відповідь: …30…, …40… (або інші числа, які відповідають типу int)

Завдання з кодом

Який результат виведе наступний код?

```
package main
import "fmt"
func main() {
    var x = 10 / 4
    fmt.Println(x)
}
```

a) 2.5

b) 2

c) 3

d) Помилка компіляції

Правильна відповідь: b).

Який буде результат виконання програми?

```
func main() {
    defer fmt.Println("Done")
    fmt.Println("Working")
}
```

Правильна відповідь:

```
Working
Done
```

Напишіть функцію, яка повертає анонімну функцію для підрахунку квадрату числа.

Правильна відповідь:

```
func square() func(int) int {
    return func(x int) int {
        return x * x
    }
}
```

Завдання з вибором істинності

Чи може змінна бути оголошена без явного вказання типу?

a) Так

b) Ні

Правильна відповідь: a).

Завдання на визначення помилок у коді

Що не так у цьому коді?

```
package main
import "fmt"
func main() {
    const pi
    pi = 3.14
    fmt.Println(pi)
}
```

a) Константа **pi** має бути ініціалізована під час оголошення

b) Константу можна змінювати після оголошення

c) Неправильне оголошення функції

Правильна відповідь: a).

Завдання на відповідність

Співставте типи даних із їх описами:

a) int8 → 1. Ціле число зі знаком (-128 до 127)

b) uint8 → 2. Ціле число без знака (0–255)

c) float32 → 3. Число з плаваючою точкою

d) string → 4. Рядок тексту

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

d) → 1.

Завдання на проведення розрахунків та визначення результатів

Які значення будуть виведені програмою?

```
package main
import "fmt"
func main() {
    for i := 1; i <= 3; i++ {
        fmt.Println(i)
    }
}
```

a) 1, 2, 3

b) 0, 1, 2

c) 1, 2, 3, 4

d) 1, 2

Правильна відповідь: a)

Відкриті запитання

Напишіть функцію, яка приймає два параметри типу int і повертає їх добуток.

Правильна відповідь:

```
func multiply(x int, y int) int {
    return x * y
}
```

3. ПОКАЖЧИКИ

Покажчики є об'єктами, значенням яких є адреси пам’яті інших об'єктів, наприклад, змінних.

3.1. Поняття покажчика

Покажчик визначається як звичайна змінна з додаванням перед типом даних символу зірочки (*****). Наприклад, визначення покажчика на об'єкт типу **int** виглядатиме наступним чином:

```
1   var p *int
```

Цьому покажчику можна присвоїти адресу змінної типу **int**. Для отримання адреси використовується операція **&**, після якої вказується ім'я змінної (**&x**), наприклад:

```
1   package main
2
3   import "fmt"
4
5   func main() {
6
7       var x int = 4          // Визначаємо змінну
8       var p *int             // Визначаємо покажчик
9       p = &x                 // Покажчик отримує адресу
    змінної
10      fmt.Println(p)         // Значення самого покажчика є
    адресою змінної x
11  }
```

В наведеному прикладі покажчик **p** зберігає адресу змінної **x**. Важливо пам’ятати, що змінна **x** має тип **int** і покажчик **p** вказує саме на об'єкт типу **int**. Тобто є відповідність за типом. Але якщо спробувати вивести адресу змінної на консоль, то побачимо, що вона становить шістнадцяткове значення, як показано на рисунку 3.1.

```
0xc0420120a0
```

Рисунок 3.1. Результат виведення адреси на консоль

У кожному окремому випадку адреса може відрізнятися, але в наведеному прикладі машинна адреса змінної **x - 0xc0420120a0**. Тобто в пам'яті комп'ютера є адреса **0xc0420120a0**, за якою розміщується змінна **x**.

За адресою, яку зберігає покажчик, можна отримати значення змінної **x**. Для цього застосовується операція ***** або операція розіменування. Результатом цієї операції є значення змінної, на яку вказує покажчик. Застосуємо цю операцію та отримаємо значення змінної **x**, наприклад:

```
1   package main
2
3   import "fmt"
4
5   func main() {
6
7       var x int = 4
8       var p *int  = &x                    // Покажчик
    отримує адресу змінної
9       fmt.Println("Address:", p)         // Значення
    покажчика це адреса змінної x
10      fmt.Println("Value:", *p)          // Значення
    змінної x
11  }
```

Результат роботи програми наведено на рисунку 3.2.

```
Address: 0xc0420c058
Value: 4
```

Рисунок 3.2. Результат роботи програми

Використовуючи покажчик, можна змінювати значення за адресою, яка зберігається в покажчику, наприклад:

```
1   var x int = 4
2   var p *int = &x
3   *p = 25
4   fmt.Println(x)       // 25
```

Для визначення покажчиків можна також використовувати скорочену форму, а саме:

```
1   f := 2.3
2   pf := &f
3
4   fmt.Println("Address:", pf)
5   fmt.Println("Value:", *pf)
```

3.1.1. Порожній вказівник

Якщо покажчику не присвоєно адресу будь-якого об'єкта, такий покажчик за замовчуванням має значення **nil**, по суті відсутність значення. Якщо спробувати отримати значення за таким порожнім покажчиком, то згенерується помилка, наприклад

```
1   var pf *float64
2   fmt.Println("Value:", *pf)  // ! Помилка, покажчик не
    вказує на жоден об'єкт
```

Тому при роботі з покажчиками іноді буває доцільно перевіряти значення на **nil** наприклад:

```
1   var pf *float64
2   if pf != nil{
3       fmt.Println("Value:", *pf)
4   }
```

3.1.2. Функція new

У мові Go змінна представляє іменований об'єкт у пам'яті. Мова Go також дозволяє створювати безіменні об'єкти і вони також розміщуються у пам'яті, але не мають імені як змінні. Для цього використовується функція **new(type)**. У цю функцію передається тип, об'єкт якого потрібно створити. Функція повертає покажчик на створений об'єкт, наприклад:

```
1   package main
2
3   import "fmt"
4
5   func main() {
6
7       p := new(int)
8       fmt.Println("Value:", *p)        // Value: 0 -
    Значення за замовчуванням
9       *p = 8                           // Змінюємо
    значення
10      fmt.Println("Value:", *p)        // Value: 8
11  }
```

В наведеному прикладі покажчик **p** матиме тип ***int**, оскільки він вказує на об'єкт типу **int**. Об'єкт, що створюється, має значення за замовчуванням (для типу **int** це число 0).

Об'єкт, створений за допомогою функції **new**, нічим не відрізняється від звичайної змінної. Єдине, щоб звернутися до цього об'єкта, отримати або змінити його адресу, необхідно використовувати покажчик.

3.2. Покажчики та функції

Розглянемо взаємозв’язок між покажчиками та функціями.

3.2.1. Покажчики як параметри функції

За замовчуванням усі параметри передаються у функцію за значенням наприклад:

```
1   package main
2   import "fmt"
3
4   func changeValue(x int){
5       x = x * x
6   }
7   func main() {
8
9       d := 5
10      fmt.Println("d before:", d)      // 5
11      changeValue(d)                   // Змінюємо
    значення
12      fmt.Println("d after:", d)       // 5 - Значення не
    змінилося
13  }
```

В наведеному прикладі функція **changeValue** змінює значення параметра, зводячи його у квадрат. Але після виклику цієї функції значення змінної **d**, яка передається в **changeValue**, не змінилося. Адже функція отримує копію даної змінної та працює з нею незалежно від оригінальної змінної **d**. Тому **d** не змінюється.

Однак що, якщо все-таки необхідно змінювати значення змінної, що передається? І в цьому випадку можна використовувати покажчики, наприклад:

```
1   package main
2
3   import "fmt"
4
5   func changeValue(x *int){
```

```
6       *x = (*x) * (*x)
7   }
8   func main() {
9
10      d := 5
11      fmt.Println("d before:", d)      // 5
12      changeValue(&d)                  // Змінюємо
    значення
13      fmt.Println("d after:", d)       // 25 - Значення
    змінилося!
14  }
```

Тепер функція **changeValue** приймає як параметр покажчик на об'єкт типу **int**. При виклику функції **changeValue** до неї передається адреса змінної **d** (**changeValue(&d)**). І після її виконання бачимо, що значення змінної **d** змінилося.

3.2.2. Покажчик, як результат функції

У мові Go функція може повертати покажчик, наприклад:

```
1   package main
2   import "fmt"
3
4   func createPointer(x int) *int{
5       p := new(int)
6       *p = x
7       return p
8   }
9
10  func main() {
11
12      p1 := createPointer(7)
13      fmt.Println("p1:", *p1)      // p1: 7
14      p2 := createPointer(10)
15      fmt.Println("p2:", *p2)      // p2: 10
16      p3 := createPointer(28)
17      fmt.Println("p3:", *p3)      // p3: 28
18  }
```

В наведеному прикладі функція **createPointer** повертає покажчик на об'єкт **int**.

Резюме

Матеріал в цьому розділі розглядає поняття вказівників у мові програмування Go, їх оголошення та використання. Вказівники - це змінні, які зберігають адреси пам'яті інших об'єктів, таких як змінні. Пояснено, як використовувати оператори ***** для розіменування та **&** для отримання адреси змінної. Наведені приклади демонструють зміну значень змінних за допомогою вказівників і їх використання як параметрів і результатів функцій. Також розглядається поняття порожнього вказівника (**nil**) і важливість перевірки на його наявність перед використанням. Описано функцію **new** для створення об'єктів у пам'яті без імені та її практичне застосування.

Наступні розділи будуть продовжувати послідовно висвітлювати все більш складні і важливі аспекти використання мови Go для програмування як вебзастосунків так і застосунків загального призначення.

Контрольні запитання і завдання

1. Що таке вказівник у мові Go?
2. Як оголосити вказівник на змінну типу **int**?
3. Яка операція використовується для отримання адреси змінної?
4. Що означає оператор ***** у контексті вказівників?
5. Що таке порожній вказівник і яке його значення за замовчуванням?
6. Що відбудеться, якщо спробувати отримати значення з порожнього вказівника?
7. Як перевірити, чи вказівник є порожнім?
8. Для чого використовується функція **new** у Go?
9. Яке значення має об'єкт, створений за допомогою **new**, за замовчуванням?
10. Як змінити значення змінної через вказівник?
11. Як передати вказівник у функцію?
12. Чому значення змінної не змінюється, якщо передати її у функцію без використання вказівника?
13. Як функція може повертати вказівник? Наведіть приклад.
14. Який тип даних має вказівник, створений функцією **new(int)**?
15. Як отримати значення змінної за її адресою, використовуючи вказівник?
16. Чи може вказівник зберігати адресу об'єкта іншого типу, ніж його оголошено?
17. Як виглядає адреса змінної у шістнадцятковому форматі?

18. Які переваги має використання вказівників у Go?
19. Як змінити значення за адресою змінної, використовуючи вказівник, у функції?
20. У яких випадках вказівники можуть викликати помилки, і як цього уникнути?
21. Оголосіть змінну **x** типу **int** зі значенням 10. Створіть вказівник на цю змінну, виведіть її адресу та значення через вказівник.
22. Оголосіть змінну y типу **float64**. Присвойте їй значення 5.5. Створіть вказівник, який вказує на **y**, і змініть значення y на 10.0 через цей вказівник.
23. Оголосіть порожній вказівник типу ***string**. Додайте перевірку на **nil**, а якщо вказівник не є порожнім, спробуйте отримати значення.
24. Напишіть функцію **updateValue**, яка приймає вказівник на **int** і змінює значення змінної на квадрат її поточного значення. Перевірте функцію в дії.
25. Створіть змінну типу int за допомогою функції **new**. Змініть значення цієї змінної через вказівник і виведіть результат.
26. Напишіть програму, яка оголошує масив цілих чисел і функцію **updateFirst**, що змінює перший елемент масива через вказівник.
27. Реалізуйте функцію **createPointer**, яка приймає значення типу **float64** і повертає вказівник на новостворений об'єкт з цим значенням.
28. Створіть дві змінні типу **int**. Визначте два вказівники, кожен із яких вказує на одну зі змінних. Поміняйте місцями значення змінних через вказівники.
29. Використовуючи скорочену форму оголошення змінних, створіть вказівник на **float64**, змініть значення змінної через вказівник та виведіть адресу та нове значення.
30. Напишіть програму, яка створює кільцеву структуру, де кожен елемент містить вказівник на наступний використовуючи рекурсію. Перевірте, як працює доступ до елементів у такій структурі.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яке значення має порожній вказівник за замовчуванням?

a) 0

b) nil

c) undefined
d) null
Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які типи даних можна використовувати із вказівниками в Go?
a) int
b) float64
c) string
d) func
Правильна відповідь: a), b), c), d).

Завдання на відповідність

Встановіть відповідність між операцією та її функцією:
a) & → 1. Отримати значення за адресою
b) * → 2. Отримати адресу змінної
Правильна відповідь:
a) → 1.
b) → 2.

Вставити пропущені слова

Для отримання адреси змінної потрібно використовувати оператор _____.
Правильна відповідь: …&…

Завдання з кодом

Що виведе наступна програма?
```
var x int = 5
var p *int = &x
fmt.Println(*p)
```
a) 5
b) Адреса змінної x
c) nil
d) Помилка виконання
Правильна відповідь: a).

Завдання з вибором істинності

Вказівники у Go можуть бути використані для роботи з об'єктами будь-якого типу.

a) Так

b) Ні

Правильна відповідь: а).

Відкриті запитання

Вкажіть, чи коректний цей код. Якщо ні, виправте його:

```
var p *int
*p = 10
```

Правильна відповідь: Некоректний. Порожній вказівник потрібно ініціалізувати наступним чином:

```
var p *int = new(int)
*p = 10
```

4. ПОХІДНІ ТИПИ

4.1. Оголошення типів

4.1.1. Іменовані типи

У мові Go оператор **type** дозволяє визначати іменований тип на основі іншого, наприклад:

```
1   type mile uint
```

В наведеному прикладі визначається іменований тип **mile**, який ґрунтується на типі **uint**. По суті, **mile** представляє тип **uint** і робота з ним буде проводитися так само, як і з типом **uint**. Однак у той самий час фактично це новий тип.

Можна визначати змінні цього типу, працювати з ними як з об'єктами базового типу **uint**, наприклад:

```
1   package main
2   import "fmt"
3
4   type mile uint
5
6   func main() {
7
8       var distance mile = 5
9       fmt.Println(distance)
10      distance += 5
11      fmt.Println(distance)
12  }
```

Але може виникнути питання, навіщо це потрібно, навіщо визначати саме іменований тип, якщо він все одно веде себе як тип **uint**? Розглянемо наступний приклад:

```
1   package main
2   import "fmt"
3
4   type mile uint
5   type kilometer uint
6
7   func distanceToEnemy (distance mile){
8
```

```
9       fmt.Println("Відстань до супротивника:")
10      fmt.Println(distance, " миль")
11  }
12
13  func main() {
14
15      var distance mile = 5
16      distanceToEnemy(distance)
17
18
19      // var distance1 uint = 5
20      // distanceToEnemy(distance1)    // !Помилка
21
22      // var distance2 kilometer = 5
23      // distanceToEnemy(distance2)    // !Помилка
24  }
```

В наведеному прикладі визначено два іменовані типи **mile** і **kilometer**, які по суті представляють тип **uint** і призначені для вираження відстані в милях і кілометрах відповідно. Також визначено функцію **distanceToEnemy()**, яка відображає відстань в милях до умовного супротивника. Як параметр приймається значення **mile**: саме значення типу **mile**, а не типу **uint**. Це дозволяє зменшити можливість передачі некоректних даних. Тобто дані, що передаються, повинні бути явним чином визначені в програмі як значення типу **mile**, а не типу **uint** або типу **kilometer**. Два іменовані типи вважаються різними, навіть якщо вони засновані на деякому загальному типі, наприклад як **uint** в наведеному коді програми.

Також іменовані типи дозволяють надати типу деякий додатковий зміст. Так, використання в коді типу "**kilometer**" або "**mile**" дозволяє вказати на призначення змінної або параметра і буде більш описовим, ніж тип **uint**.

Ще одна ситуація, де можна застосовувати іменовані типи це скорочення назви типів у тому випадку, якщо вони занадто довгі або громіздкі. Наприклад, розглянемо наступний код програми:

```
1   package main
2
3   import "fmt"
4
5   func action(n1 int, n2 int, op func(int, int) int){
6
```

```
7       result := op(n1, n2)
8       fmt.Println(result)
9   }
10
11  func add(x int, y int) int {
12
13      return x + y
14  }
15
16  func main() {
17
18      var myOperation func(int, int) int = add
19      action(10, 25, myOperation)      // 35
20  }
```

В наведеному прикладі визначено функцію **action**, яка приймає два числа та деяку іншу функцію з типом **func(int, int) int**, тобто функцію, яка приймає два числа і також повертає число. У функції **main** визначається змінна **myOperation**, яка представляє функцію типу **func(int, int) int**, яка отримує посилання на функцію **add** і передається у виклик **action(10, 25, myOperation)**.

Тепер визначимо саме іменований тип для типу **func(int, int) int**, а саме:

```
1   package main
2
3   import "fmt"
4
5   type BinaryOp func(int, int) int
6
7   func action(n1 int, n2 int, op BinaryOp){
8
9       result := op(n1, n2)
10      fmt.Println(result)
11  }
12
13  func add(x int, y int) int {
14
15      return x + y
16  }
17
18  func main() {
```

```
19
20      var myOperation BinaryOp = add
21      action(10, 35, myOperation)      // 45
22  }
```

Тепер можна спроектувати тип функції **func(int, int) int** на іменований тип **BinaryOp**, який представляє бінарну операцію над двома операндами, а саме:

```
1   type BinaryOp func(int, int) int
```

Така назва коротша за оригінальне визначення типу і в той же час є більш описовою. Відповідно, далі її можна використовувати для вказування типу параметра, а саме:

```
1   func action(n1 int, n2 int, op BinaryOp){ ... }
```

або змінної:

```
1   var myOperation BinaryOp = add
```

4.1.2. Псевдоніми

На іменовані типи схожі псевдоніми. Вони також визначаються за допомогою оператора **type**, тільки при присвоєнні типу застосовується операція присвоєння, а саме:

```
1   type псевдонім = наявний_тип
```

Важливо розуміти, що псевдонім не визначає нового типу і всі псевдоніми одного й того самого типу вважаються ідентичними, наприклад:

```
1   package main
2   import "fmt"
3
4   type mile = uint
5   type kilometer = uint
6
7   func distanceToEnemy (distance mile){
8
9       fmt.Println("Відстань до супротивника:")
10      fmt.Println(distance, "миль")
11  }
12
13  func main() {
14
15      var distance mile = 5
```

```
16      distanceToEnemy(distance)
17
18      var distance1 uint = 5
19      distanceToEnemy(distance1)    // Ок
20
21      var distance2 kilometer = 5
22      distanceToEnemy(distance2)    // Ок
23  }
```

В наведеному прикладі для типу **uint** визначаються два псевдоніми **mile** і **kilometer**. І незважаючи на те, що параметр функції **distanceToEnemy** визначений як параметр типу **mile**, йому можна передати і значення власне типу **uint**, і значення його псевдоніма **kilometer**.

Зазвичай псевдоніми застосовуються для скорочення назв інших типів чи визначення більш описового імені.

4.2. Структури

Структури представляють тип даних, що визначається розробником і спеціалізованих уявлень про будь-які об'єкти. Структури містять набір полів, які представляють різні атрибути об'єкта. Для визначення структури застосовуються ключові слова **type** та **struct**, а саме:

```
1   type ім'я_структури struct{
2       поля_структури
3   }
```

Кожне поле структури має назву та тип даних, як і змінні. Наприклад, визначимо структуру, яка описує людину:

```
1   type person struct{
2       name string
3       age int
4   }
```

В наведеному прикладі структура називається **person**. Вона має два поля: **name** – ім'я людини, яке має тип **string**, та **age** – вік людини, який має тип **int**.

4.2.1. Створення та ініціалізація структур

Структура представляє новий тип даних. У мові Go можна визначити змінну цього типу наступним чином:

```
1   var tom person
```

За допомогою ініціалізації можна передати структурі початкові значення, наприклад:

```
var tom person = person{"Tom", 23}
```

Ініціалізатор представляє набір значень у фігурних дужках. Причому ці значення передаються полям структури у порядку, у якому поля визначено у структурі. Наприклад, рядок " **Tom** " передається першому полю **name**, а друге значення 23 передається другому полю **age**.

Також можна явно вказати, які значення передаватимуться властивостям, наприклад:

```
var alice person = person{age: 23, name: "Alice"}
```

Можна використовувати скорочені способи ініціалізації змінної структури, а саме:

```
var tom = person {name: "Tom", age: 24}
bob := person {name: "Bob", age: 31}
```

Можна навіть не вказувати жодних значень, тоді у цьому випадку властивості структури отримають значення за замовчуванням, наприклад:

```
undefined := person {}  // name - Порожній рядок, age - 0
```

4.2.2. Звернення до полів структури

Для звернення до полів структури після імені змінної ставиться символ крапки (**.**) та вказується поле структури, наприклад:

```
package main
import "fmt"

type person struct{
    name string
    age int
}

func main() {

    var tom = person {name: "Tom", age: 24}
    fmt.Println(tom.name)       // Tom
    fmt.Println(tom.age)        // 24

    tom.age = 38    // Змінюємо значення
```

```
16      fmt.Println(tom.name, tom.age)      // Tom 38
17  }
```

4.2.3. Покажчики на структури

Як і у випадку зі звичайними змінними, можна створювати покажчики на структури, наприклад:

```
1   package main
2   import "fmt"
3
4   type person struct{
5       name string
6       age int
7   }
8
9   func main() {
10
11      tom := person {name: "Tom", age: 22}
12      var tomPointer *person = &tom
13      tomPointer.age = 29
14      fmt.Println(tom.age)        // 29
15      (*tomPointer).age = 32
16      fmt.Println(tom.age)        // 32
17  }
```

Для ініціалізації покажчика на структуру необов'язково надавати йому адресу змінної. Можна надати адресу безіменного об'єкта наступним чином:

```
1   var tom *person = &person{name:"Tom", age:23}
2   var bob *person = new(person)
```

Для звернення до полів структури через покажчик можна використовувати операцію розіменування **(*tomPointer).age**, або безпосередньо звертатися за покажчиком **(tomPointer.age)**.

Також можна визначати покажчики на окремі поля структури, наприклад:

```
1   tom := person {name: "Tom", age: 22}
2   var agePointer *int = &tom.age  // Покажчик на поле
    tom.age
3   *agePointer = 35            // Змінюємо значення поля
4   fmt.Println(tom.age)        //  35
```

4.3. Вкладені структури

Поля одних структур можуть бути іншими структурами. Розглянемо детальніше призначення і правила роботи з такими структурами.

4.3.1. Базові операції з вкладеними структурами

Для ознайомлення з базовими операціями з вкладеними структурами розглянемо наступний приклад:

```
1   package main
2   import "fmt"
3
4   type contact struct{
5       email string
6       phone string
7   }
8
9   type person struct{
10      name string
11      age int
12      contactInfo contact
13  }
14
15  func main() {
16
17      var tom = person {
18          name: "Tom",
19          age: 24,
20          contactInfo: contact{
21              email: "tom@gmail.com",
22              phone: "+1234567899",
23          },
24      }
25      tom.contactInfo.email = "supertom@gmail.com"
26
27      fmt.Println(tom.contactInfo.email)      //
    supertom@gmail.com
28      fmt.Println(tom.contactInfo.phone)      //
    +1234567899
```

```
}
```

В наведеному прикладі структура **person** має поле **contactInfo**, яке представляє іншу структуру **contact**.

Можна скоротити визначення поля наступним чином:

```
package main
import "fmt"

type contact struct{
    email string
    phone string
}

type person struct{
    name string
    age int
    contact
}

func main() {

    var tom = person {
        name: "Tom",
        age: 24,
        contact: contact{
            email: "tom@gmail.com",
            phone: "+1234567899",
        },
    }
    tom.email = "supertom@gmail.com"

    fmt.Println(tom.email)      // supertom@gmail.com
    fmt.Println(tom.phone)      // +1234567899
}
```

Поле **contact** у структурі **person** фактично еквівалентно властивості **contact** структури **contact**, тобто властивість називається **contact** і представляє тип **contact**. Це дозволяє скоротити шлях до полів вкладеної структури. Наприклад, можна записати **tom.email**, а не **tom.contact.email**. Хоча можна використати і другий варіант.

4.3.2. Зберігання посилання на структуру того ж типу

Треба враховувати, що структура не може мати поле, яке представляє тип цієї структури, наприклад:

```
type node struct{
    value int
    next node
}
```

Подібне визначення буде неправильним. Натомість поле має представляти покажчик на структуру, а саме:

```
package main
import "fmt"

type node struct{
    value int
    next *node
}

// Рекурсивне виведення списку
func printNodeValue(n *node){

    fmt.Println(n.value)
    if n.next != nil{
        printNodeValue(n.next)
    }
}
func main() {

    first := node{value: 4}
    second := node{value: 5}
    third := node{value: 6}

    first.next = &second
    second.next = &third

    var current *node = &first
    for current != nil{
        fmt.Println(current.value)
```

```
29            current = current.next
30        }
31    }
```

В наведеному прикладі визначено структуру **node**, яка представляє типовий вузол однозв'язного списку. Вона зберігає значення в полі **value** та посилання на наступний вузол через покажчик **next**.

У функції **main** створюються три пов'язані структури, і з допомогою циклу **for** і допоміжного покажчика **current** виводяться їх значення. Результат роботи програми наведено на рисунку 4.1.

```
4
5
6
```

Рисунок 4.1. Результат роботи програми

4.4. Методи

Методом є функція з певним типом. Методи визначаються так само як і звичайні функції за винятком того, що у визначенні метода також необхідно вказати одержувача або **receiver**. Одержувач – це параметр типу, до якого прикріплюється метод, а саме:

```
1   func (ім'я_параметра тип_одержувача) ім'я_метода
          (параметри) (типи_результатів_які_повертаються) {
2       тіло_метода
3   }
```

Допустимо, визначений іменований тип, що представляє зріз з рядків, а саме:

```
1   type library []string
```

Для виведення всіх елементів із зрізу можна визначити наступний метод:

```
1   func (l library) print(){
2
3       for _, val := range l{
4           fmt.Println(val)
5       }
6   }
```

Та частина, яка розташована між ключовим словом **func** та ім'ям метода для якого буде визначено цей метод і представляє визначення одержувача, а саме: **(l library)**. Використовуючи параметр метода – ідентифікатор об’єкта (**l**),

можна звертатися до одержувача. Наприклад, якщо одержувач представляє зріз (набір об'єктів), то за допомогою циклу **for** можна пройтися цим зрізом і вивести всі його елементи на консоль.

Оскільки **print** представляє саме метод, визначений для типу **library**, а не звичайну функцію, то можна викликати цей метод в будь-якого об'єкта типу **library**, наприклад:

```
var lib library = library{ "Book1", "Book2", "Book3"
}
lib.print()
```

В наведеному фрагменті коду **lib** є об'єктом типу **library**, тому можна викликати метод **print**. В даному випадку об'єкт **lib** це і буде значення, яке буде передаватися у функцію **print** через параметр **(l library)**.

4.4.1. Методи структур

Подібним чином можемо визначати методи і для структур. наприклад:

```
package main

import "fmt"

type person struct{
    name string
    age int
}
func (p person) print(){
    fmt.Println("Ім'я:", p.name)
    fmt.Println("Вік:", p.age)
}

func (p person) eat(meal string){
    fmt.Println(p.name, "їсть", meal)
}

func main() {

    var tom = person { name: "Tom", age: 24 }
    tom.print()
    tom.eat("борщ з капустою, але не зелений")
```

```
23  }
```

Результат роботи програми наведено на рисунку 4.2.

```
Ім'я: Tom
Вік: 24
Tom їсть борщ з капустою, але не зелений
```

Рисунок 4.2. Результат роботи програми

У наведеному прикладі для структури **person** визначено дві функції: **print** та **eat**. Функція **print** виводить інформацію про поточний об'єкт **person**. А функція **eat** імітує вживання їжі. Кожна з цих функцій визначає об'єкт та тип структури, до якої функція відноситься, а саме:

```
1   func (p person) ім'я_функції
```

За допомогою об'єкта **p** можна звертатися до властивостей структури **person**. В іншому це звичайні функції, які можуть приймати параметри та повертати результат.

Для звернення до функцій структури вказується змінна структури і через

```
1   tom.print()
2   tom.eat("борщ")
```

У наведеному прикладі **tom** – це і є об'єкт **p person** у визначенні функції.

4.4.2. Методи вказівників

При виклику метода, об'єкт структури, котрим визначено метод, передається йому за значенням. Для розуміння того, що це означає, розглянемо наступний приклад:

```
1   package main
2   import "fmt"
3
4   type person struct{
5       name string
6       age int
7   }
8   func (p person) updateAge(newAge int){
9       p.age = newAge
10  }
11
12  func main() {
```

```
13
14      var tom = person { name: "Tom", age: 24 }
15      fmt.Println("before", tom.age)
16      tom.updateAge(33)
17      fmt.Println("after", tom.age)
18  }
```

Для структури **person** визначено метод **updateAge**, який приймає параметр **newAge** і змінює значення поля **age** у структурі. Тобто під час виклику цього метода очікується, що вік людини зміниться. Однак результат роботи програми на консолі (рис. 4.3) показує, що значення поля **age** не змінюється.

```
before 24
after 24
```

Рисунок 4.3. Результат роботи програми

Це відбулося завдяки тому, що під час виклику **tom.updateAge(33)** метод **updateAge** отримує копію структури **tom**. Тобто структура **tom** копіюється в іншу ділянку пам'яті, і далі метод **updateAge** працює з копією, ніяк не торкаючись оригінальної структури **tom**.

Однак така поведінка може бути небажаною. Що якщо все ж таки необхідно саме таким чином змінювати стан структури. В цьому випадку необхідно використати покажчики на структури, наприклад:

```
1   package main
2   import "fmt"
3
4   type person struct{
5       name string
6       age int
7   }
8   func (p *person) updateAge(newAge int){
9       (*p).age = newAge
10  }
11
12  func main() {
13
14      var tom = person { name: "Tom", age: 24 }
15      var tomPointer *person = &tom
```

```
16      fmt.Println("before", tom.age)
17      tomPointer.updateAge(33)
18      fmt.Println("after", tom.age)
19  }
```

Тепер метод **updateAge** приймає покажчик на структуру **person: p *person**, тобто фактично адресу структури пам'яті. За допомогою операції розіменування отримується значення за цією адресою в пам'яті та змінюється поле **age**, а саме:

```
1   (*p).age = newAge
```

У функції **main** визначено покажчик на структуру **person** та передану йому адресу структури **tom**, а саме:

```
1   var tomPointer *person = &tom
```

Потім викликається метод **updateAge**, а саме:

```
1   tomPointer.updateAge(33)
```

Таким чином, метод **updateAge** отримує адресу, яка зберігається в **tomPointer** і за цією адресою звертається до структури **tom**, змінивши значення її властивості **age**, як показано на рисунку 4.4.

```
before 24
after 33
```

Рисунок 4.4. Результат роботи програми

Варто зазначити, що незважаючи на те, що метод **updateAge** визначений для покажчика на структуру **person**, як і раніше, можна викликати цей метод і для об'єкта **person**, наприклад:

```
1   var tom = person { name: "Tom", age: 24 }
2   fmt.Println("before", tom.age)       // before 24
3   tom.updateAge(33)
4   fmt.Println("after", tom.age)        // after 33
```

Резюме

У наведеному матеріалі розглянуто використання похідних типів у мові Go, зокрема іменованих типів та псевдонімів. Іменовані типи створюються за допомогою оператора **type** і базуються на існуючих типах, але вважаються окремими. Вони допомагають зробити код більш зрозумілим, зменшити ризик помилок і забезпечують додаткову семантику, наприклад, для позначення одиниць виміру. На відміну від них, псевдоніми не створюють нового типу, а

просто дають нову назву існуючому. Псевдоніми корисні для скорочення довгих імен типів або для більш описових назв. Наведено приклади використання обох підходів, щоб показати їх переваги в реальному коді.

Також розглянуто такі структури в мові Go як користувацькі типи даних для опису об'єктів із різними атрибутами. Описано створення структур за допомогою ключових слів **type** та **struct**, а також способи ініціалізації змінних цих типів із заданням початкових значень. Пояснюється, як звертатися до полів структури, змінювати їх значення та працювати з вказівниками на структури. Наведено приклади вкладених структур і способів їх скороченого використання. Додатково описано визначення методів для структур, які дозволяють виконувати операції над їхніми даними, а також розглянуто особливості передачі структур за значенням і за вказівником.

Наступні розділи будуть продовжувати послідовно висвітлювати все більш складні і важливі аспекти використання мови Go для програмування як вебзастосунків так і застосунків загального призначення.

Контрольні запитання і завдання

1. Що таке іменований тип у мові Go? Як він створюється?
2. Чим іменований тип відрізняється від базового типу, на якому він базується?
3. Які переваги використання іменованих типів у програмному коді?
4. Чому функція, яка приймає значення іменованого типу, не може приймати значення базового типу без явного приведення?
5. Як іменовані типи можуть сприяти зменшенню кількості помилок у програмі?
6. У чому відмінність між іменованим типом і псевдонімом у мові Go?
7. Як створюється псевдонім типу? Наведіть приклад.
8. Чи можуть псевдоніми різних назв одного базового типу використовуватися взаємозамінно? Поясніть чому.
9. Яка функція використовується в прикладі для демонстрації переваг іменованих типів?
10. Чому доцільно використовувати іменовані типи для функціональних типів (наприклад, **func(int, int) int**)?
11. Що таке структура в мові Go, і для чого її використовують?
12. Як оголосити структуру в мові Go? Наведіть приклад.
13. Як ініціалізувати структуру з початковими значеннями? Наведіть приклад.

14. Як звернутися до полів структури?
15. Чи можна ініціалізувати структуру без вказання значень? Що тоді відбувається з її полями?
16. Як створити вказівник на структуру? Наведіть приклад.
17. Як змінювати значення полів структури через вказівник?
18. Що таке вкладена структура? Як до неї звертатися?
19. Як створити однозв’язний список за допомогою структур?
20. Чим метод відрізняється від звичайної функції?
21. Чому метод, визначений для вказівника на структуру, можна викликати для самої структури?
22. Чим відрізняється ініціалізація структури за допомогою **new()** від явного присвоєння значень?
23. Створіть іменований тип для позначення ваги в кілограмах (**kilogram**), базуючись на типі **float64**. Напишіть функцію, яка приймає значення цього типу та виводить вагу у форматі: "**Вага: X кг**".
24. Визначте іменований функціональний тип **MathOperation** для операцій над двома числами типу **int**. Створіть функції **add** (додавання) та **multiply** (множення), які відповідатимуть цьому типу, і викличте їх через загальну функцію **executeOperation**.
25. Визначте два іменовані типи: **mile** та **kilometer**, базуючись на типі **uint**. Напишіть програму, яка створює змінні цих типів та намагається скласти їх. Що відбувається? Як можна виправити цю ситуацію?
26. Створіть псевдоніми для типів **float64** та **int**, які називаються **money** та **quantity** відповідно. Напишіть функцію, яка обчислює загальну вартість товару на основі ціни та кількості, використовуючи ці псевдоніми.
27. Визначте іменований тип **Temperature** на основі типу **float64**. Напишіть функцію, яка приймає температуру у градусах Цельсія та конвертує її у градуси за Фаренгейтом.
28. Визначте іменований тип **UserData** для типу **map[string]string**. Напишіть програму, яка створює словник із даними користувача (name, email) і виводить їх у форматі JSON.
29. Створіть псевдоніми **meters** та **centimeters** для типу **float64**. Напишіть функцію, яка приймає відстань у метрах і перетворює її в сантиметри. Передайте значення обох типів у функцію та перевірте сумісність.
30. Створіть структуру **Book** з полями **title**, **author** та **pages**. Ініціалізуйте змінну типу **Book** та виведіть її поля на екран.

31. Оголосіть структуру **User** із полями name та age. Ініціалізуйте змінну цієї структури, створіть вказівник на неї, змініть значення полів через вказівник та виведіть змінені значення.
32. Створіть структуру **Address** з полями **city**, **street** та **houseNumber**, а також структуру **Person** з полями **name**, **age** та **address** (типу **Address**). Ініціалізуйте змінну **Person** та виведіть на екран дані про цю людину.
33. Додайте до структури **Rectangle** з полями **width** та **height** метод **Area**, який обчислює площу прямокутника. Перевірте роботу метода, створивши змінну цієї структури.
34. Створіть структуру **Account** із полями **owner** та **balance**. Напишіть метод **Deposit**, який приймає вказівник на **Account** та збільшує баланс. Перевірте метод, викликавши його для змінної структури.
35. Створіть структуру **Node** для однозв'язного списку з полем **value** та вказівником на наступний вузол. Напишіть функцію для виводу значень вузлів списку за допомогою рекурсії.
36. Оголосіть структуру **Employee** з полями **name** та **salary**. Напишіть метод для збільшення зарплати на заданий відсоток. Використайте метод для змінної типу **Employee**.
37. Створіть структури **Library** (із полем **name**) та **Book** (із полями **title** і **author**). Визначте метод для структури **Library**, який додає книгу до бібліотеки (зберігаючи список книг у зрізі). Напишіть програму, яка додає кілька книг у бібліотеку та виводить список усіх книг.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Що створює оператор type у Go?

a) Новий базовий тип

b) Іменований тип

c) Псевдонім для існуючого типу

d) Усі варіанти правильні

Правильна відповідь: b).

Яке ключове слово використовується для визначення структури в Go?

a) struct

b) class

c) type

d) object

Правильна відповідь: c).

Закриті запитання з кількома правильними відповідями

Що з переліченого є перевагами використання іменованих типів у Go?

a) Полегшують читабельність коду

b) Зменшують ризик помилок

c) Додають нові властивості базовому типу

d) Покращують продуктивність програми

Правильна відповідь: a), b).

Які з наступних способів ініціалізації структури правильні?

a) var p = person{name: "Tom", age: 24}

b) p := person{"Tom", 24}

Правильна відповідь: a), b).

Завдання на відповідність

Встановіть відповідність між термінами та їх описами:

a) Іменований тип → 1. Новий тип, який базується на існуючому

b) Псевдонім типу → 2. Альтернативна назва для існуючого типу

c) Функціональний тип → 3. Приймає та/або повертає функцію

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

У якому порядку слід виконувати дії для створення структури та роботи з нею?

a) Оголосити структуру → 1.

b) Ініціалізувати змінну структури → 2.

c) Звернутися до полів структури → 3.

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Завдання з кодом

Що виведе наступний код?

```
type kilometer float64
var distance kilometer = 10.5
fmt.Println(distance)
```

a) 10.5

b) 10.5 км

c) Помилка компіляції

d) 10

Правильна відповідь: a)

Вставити пропущені слова

Іменований тип створюється за допомогою оператора _____.
Правильна відповідь:…type…

Заповніть пропуски для створення структури Car з полями brand і year.

```
type _____________struct {
    brand string
    year  int
}
```

Правильна відповідь:…Car…

Завдання з кодом

Що виведе наступний код?

```
type person struct {
    name string
    age  int
}
func main() {
    tom := person{name: "Tom", age: 25}
    fmt.Println(tom.name, tom.age)
}
```

a) Tom 25

b) Tom

c) Tom, 25

d) Помилка компіляції

Правильна відповідь: а).

Відкриті запитання

Чому функція, яка приймає значення типу mile, не може приймати значення типу uint?

Правильна відповідь: Тому що mile і uint є різними типами, навіть якщо базуються на одному значенні.

У чому помилка?

```
type person struct {
    name string
    age  int
}
func main() {
    var p person
    fmt.Println(p.fullName)
}
```

Правильна відповідь: Поля fullName немає у структурі person.

5. ПАКЕТИ ТА МОДУЛІ

5.1. Пакети та їх імпорт

Весь код у мові Go організовано у пакети. Пакети представляють зручну організацію поділу коду на окремі частини чи модулі. Модульність дозволяє визначати один раз пакет із потрібною функціональністю і потім використовувати його багаторазово у різних програмах.

Код пакета розташовується в одному або кількох файлах із розширенням **.go**. Для визначення пакета використовується ключове слово package, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       fmt.Println("Hello Go")
7   }
```

У наведеному прикладі пакет називається **main**. Визначення пакета має оголошуватись на початку файлу.

У мові Go два типи пакетів: виконувані (**executable**) та бібліотеки (**reusable**). Для створення файлів, що виконуються, пакет повинен мати ім'я **main**. Всі інші пакети не виконуються. При цьому пакет **main** повинен містити функцію **main**, яка є вхідною точкою програми.

5.1.1. Імпорт пакетів

Якщо вже є готові пакети з потрібною функціональністю, яку необхідно використовувати, то для їх використання можна їх імпортувати в програму за допомогою оператора **import**. Наприклад у попередньому коді програми реалізовано функціональність виведення повідомлення на консоль за допомогою функції **Println**, визначеної в пакеті **fmt**. Відповідно, щоб використовувати цю функцію, необхідно імпортувати пакет **fmt**, а саме:

```
1   import "fmt"
```

Нерідко програми підключають відразу кілька зовнішніх пакетів. У цьому випадку можна послідовно імпортувати кожен пакет, наприклад:

```
1   package main
2   import "fmt"
3   import "math"
```

```
4
5   func main() {
6
7       fmt.Println(math.Sqrt(16))   // 4
8   }
```

У наведеному прикладі підключається вбудований пакет **math**, який містить функцію **Sqrt()**, що повертає квадратний корінь числа.

Або щоб скоротити визначення імпорту пакетів можна перелічити всі пакети в дужках, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "math"
5   )
6
7   func main() {
8
9       fmt.Println(math.Sqrt(16))
10  }
```

Подібним чином можна імпортувати як вбудовані пакети, так і власні. Повний список вбудованих пакетів Go можна знайти за адресою **https://golang.org/pkg/** [6].

5.1.2. Пакет із кількох файлів

Один пакет може складатися з кількох файлів. Наприклад, визначимо в теці два файли, як показано на рисунку 5.1.

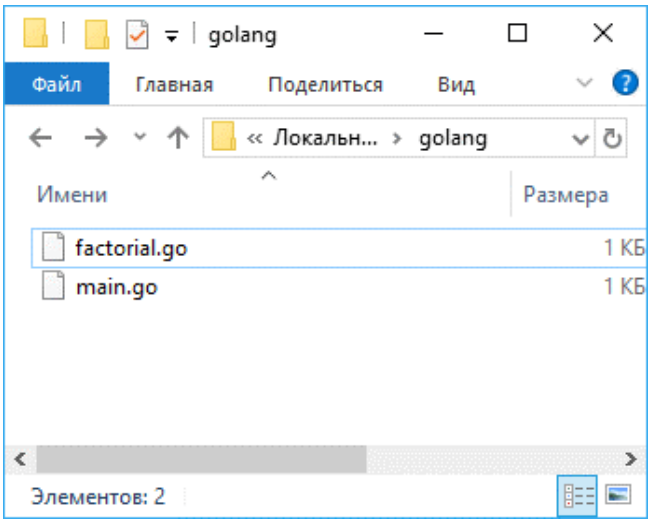

Рисунок 5.1. Створення пакета із двох файлів

У файлі **factorial.go** визначимо функцію для підрахунку факторіала, а саме:

```
1   package main
2
3   func factorial(n int) int {
4
5       var result = 1
6       for i:=1; i <= n; i++{
7           result *= i
8       }
9       return result
10  }
```

Цей файл належить пакету **main**.

У файлі **main.go** використовуємо функцію для обчислення факторіала, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       fmt.Println(factorial(4))
7       fmt.Println(factorial(5))
8   }
```

Цей файл також належить пакету **main**. Файлів може бути і більше. Тепер скомпілюємо з цих файлів програму. Для цього перейдемо в консолі до теки проєкту та виконаємо наступну команду, як показано на рисунку 5.2.

```
go build -o main.exe factorial.go main.go
```

Рисунок 5.2. Зовнішній вигляд вікна введення команди

Прапорець **-o** вказує, як називатиметься вихідний файл: у наведеному прикладі це **main.exe**. Потім вказуються усі файли, що компілюються. Після виконання цієї команди буде створено файл **main.exe**, який можна запустити у консолі. Результа роботи програми наведено на рисунку 5.3.

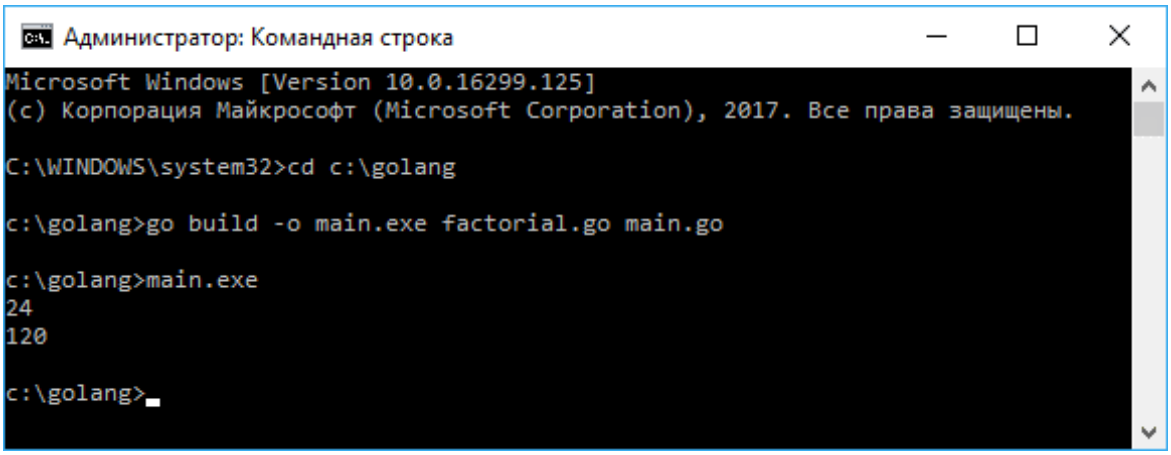

Рисунок 5.3. Результат роботи програми

5.1.3. Файли у різних пакетах

Тепер розглянемо іншу ситуацію, коли файли програми розділені на різні пакети. Визначимо в теці проєкту каталог **computation**. Потім до каталогу **computation** додамо наступний файл **factorial.go**:

```
1   package computation
2
3   func Factorial(n int) int {
4
5       var result = 1
6       for i:=1; i <= n; i++{
7           result *= i
8       }
9       return result
10  }
```

Код файлу **factorial.go** належить пакету **computation**. Важливо відзначити, що назва функції починається з великої літери. Завдяки цьому ця функція буде відображатись в інших пакетах, як показано на рисунку 5.4.

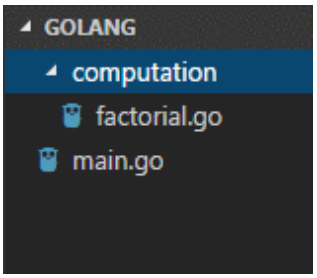

Рисунок 5.4. Вікно менеджера пакетів і файлів

Щоб використовувати функцію **factorial**, треба імпортувати цей пакет у файлі **main.go**, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "./computation"
5   )
6
7   func main() {
8
9       fmt.Println(computation.Factorial(4))
10      fmt.Println(computation.Factorial(5))
11  }
```

Шлях **"./computation"** вказує, що пакет знаходиться у теці **computation**.

Компіляція та виконання програми здійснюється також як і раніше без необхідності вказівки всіх файлів з інших пакетів, як показано на рисунку 5.5.

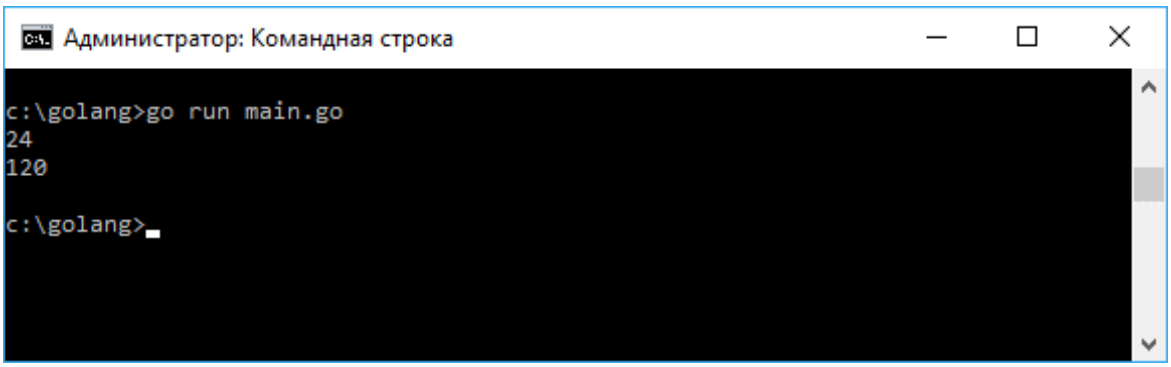

Рисунок 5.5. Вікно запуску компіляції та виконання програми

5.2. Введення у модулі

Модулі є набором пакетів Go, які мають вбудований механізм відстеження версій і які можна опублікувати для використання в інших проєктах.

Мова Go має багату функціональність, однієї цієї вбудованої функціональності може бути недостатньо для конструювання програми. Однак, мова Go має велику екосистему розробників, які розробляють та публікують різні модулі. Ці модулі можуть надавати функціональність, яка відсутня у вбудованій бібліотеці пакетів мови Go, але яка може бути потрібна іншим програмістам. Тож Go також дозволяє використовувати ці модулі. Розглянемо

як створювати власні модулі та підключати і використовувати зовнішні модулі у власних додатках. Починаючи з версії Go 1.16 для використання зовнішніх модулів необхідно визначити власні модулі.

5.2.1. Створення модуля

Для створення модуля використовується команда `go mod ini`, якому передається ім'я модуля.

Спочатку визначимо теку, де розташовуватиметься модуль. Нехай, наприклад, це буде тека **C:\golang**. Далі перейдемо у терміналі (командному рядку) до цієї теки. Нехай, модуль буде називатися **helloapp**. Для його створення виконаємо наступну команду, як показано на рисунку 5.6

```
go mod init helloapp
```

Рисунок 5.6. Вікно створення модуля за допомогою команди **go mod ini**

Результат роботи команди **go mod ini** наведено на рисунку 5.7.

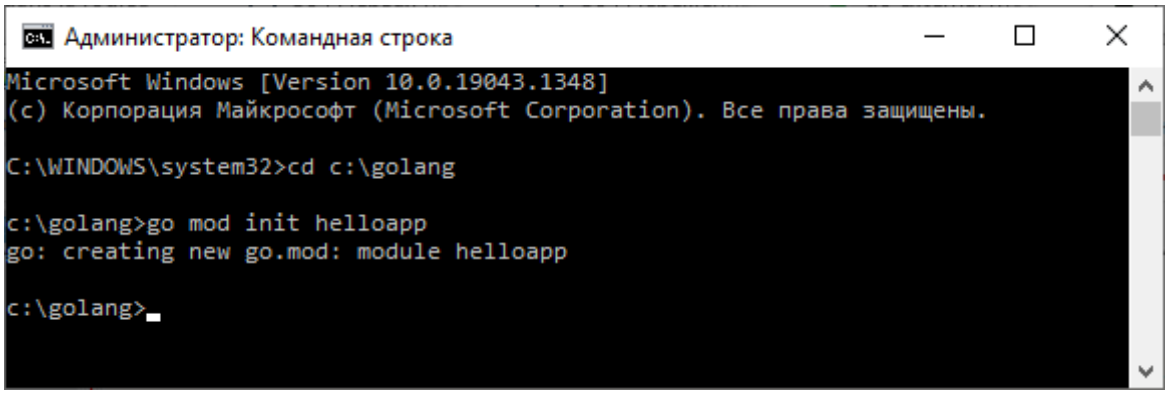

Рисунок 5.7. Результат роботи команди **go mod ini**

Після виконання цієї команди у теці з'явиться файл **go.mod**. За умовчанням він матиме наступний вміст:

```
1   module helloapp
2
3   go 1.17
```

Перший рядок з директивою **module** визначає шлях модуля, а саме **"helloapp"**. Другий рядок визначає версію **go**, що використовується для модуля, а саме: 1.17.

Крім шляху модуля та версії **go** цей файл дозволяє керувати залежностями, так званими зовнішніми модулями, які підключаються до програми.

5.2.2. Завантаження зовнішнього модуля

Тепер підключимо до модулю **helloapp** зовнішній модуль. Наприклад візьмемо модуль **"rsc.io/quote"**. Це модуль, визначений спеціально для цілей тестування. Він містить набір функцій, які повертають текст.

Для завантаження модуля **"rsc.io/quote"** необхідно виконати наступну команду, як показано на рисунку 5.8.

```
go get rsc.io/quote
```

Рисунок 5.8. Зовнішній вигляд вікна введення команди

Під час виконання цієї команди Go завантажить необхідні залежності, як показано на рисунку 5.9.

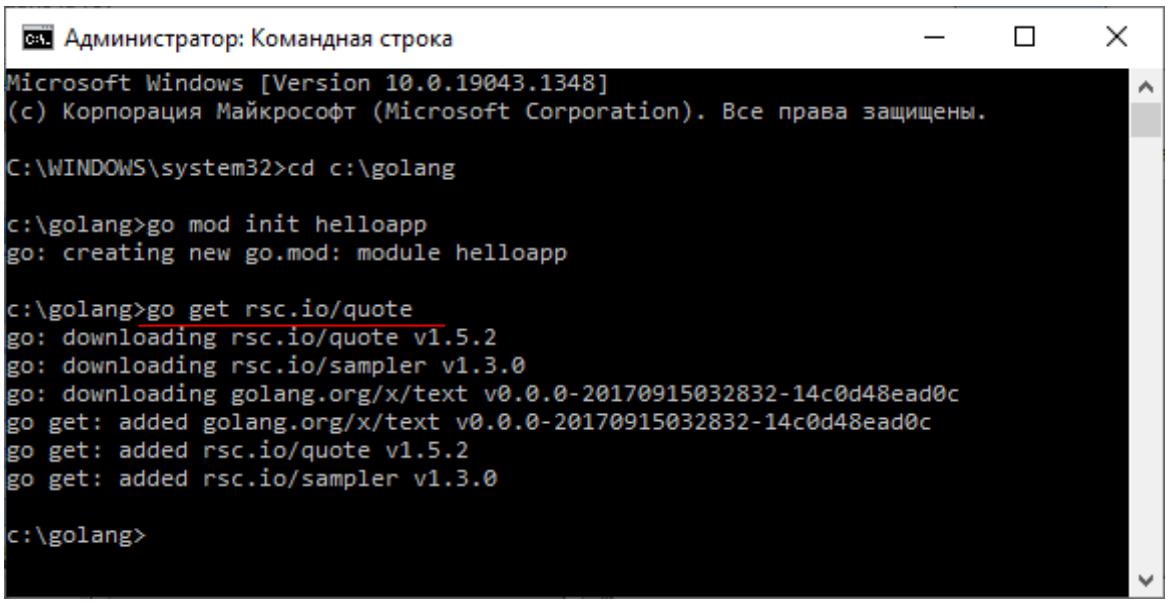

Рисунок 5.9. Зовнішній вигляд вікна завантаження зовнішнього модуля

Всі пакети, що завантажуються, зберігаються по шляху **$GOPATH\pkg\mod**. Зазвичай змінна середовища **GOPATH** вказує на теку **[Тека поточного користувача]\go**. Наприклад, у наведеному прикладі пакети завантажуватимуться по шляху **C:\golang\pkg\mod**.

І якщо тепер знову відкрити файл **go.mod**, то побачимо, що його вміст змінився, а саме:

```
1   module helloapp
2
3   go 1.17
4
5   require (
```

```
6       golang.org/x/text v0.0.0-20170915032832-
    14c0d48ead0c // indirect
7       rsc.io/quote v1.5.2 // indirect
8       rsc.io/sampler v1.3.0 // indirect
9   )
```

У файл додалася директива **require()**, яка містить визначення залежностей, що підключаються. В наведеному прикладі це всі ті залежності, які потрібні для роботи з пакетом **"rsc.io/quote"**.

Крім того, також можна помітити, що у теці проєкту **c:\golang** з'явився ще один файл з іменем **go.sum**, вмістом якого виглядає наступним чином:

```
1   golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
    h1:qgOY6WgZOaTkIIMiVjBQcw93ERBE4m30iBm00nkL0i8=
2   golang.org/x/text v0.0.0-20170915032832-
    14c0d48ead0c/go.mod
    h1:NqM8EUOU14njkJ3fqMW+pc6Ldnwhi/IjpwHt7yyuwOQ=
3   rsc.io/quote v1.5.2
    h1:w5fcysjrx7yqtD/aO+QwRjYZOKnaM9Uh2b40tElTs3Y=
4   rsc.io/quote v1.5.2/go.mod
    h1:LzX7hefJvL54yjefDEDHNONDjII0t9xZLPXsUe+TKr0=
5   rsc.io/sampler v1.3.0
    h1:7uVkIFmeBqHfdjD+gZwtXXI+RODJ2Wc4O7MPEh/QiW4=
6   rsc.io/sampler v1.3.0/go.mod
    h1:T1hPZKmBbMNahiBKFy5HrXp6adAjACjK9JXDnKaTXpA=
```

Цей файл містить контрольну суму для пакетів, що підключаються.

5.2.3. Підключення зовнішнього модуля

Пакети із зовнішнього модуля підключаються так само, як і інші пакети за допомогою директиви **import**. Наприклад, визначимо у теці **c:\golang** файл для коду **main.go**, як показано на рисунку 5.10.

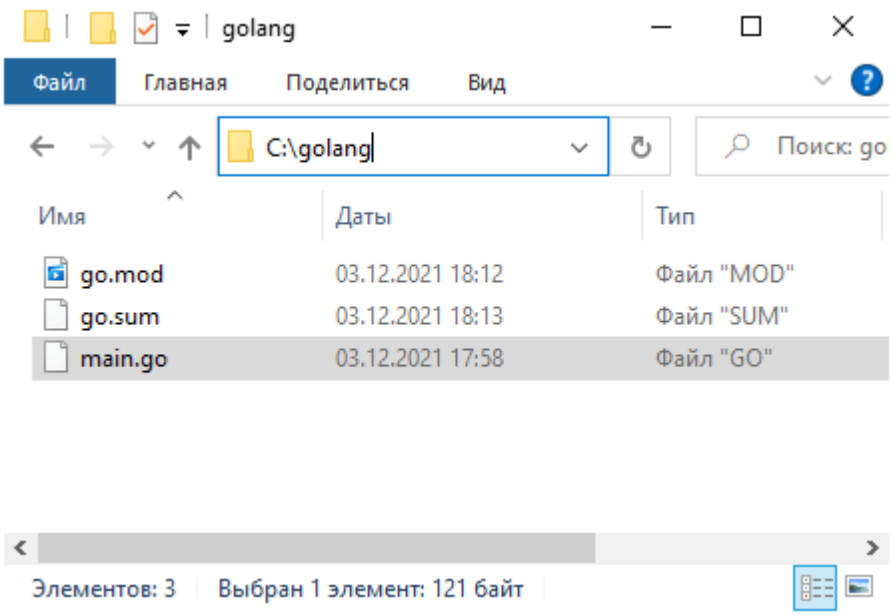

Рисунок 5.10. Визначення у теці **c:\golang** файлу для коду **main.go**

Визначимо в **main.go** наступний вміст:

```
1   package main
2
3   import (
4       "fmt"
5       "rsc.io/quote"
6   )
7
8   func main() {
9       message := quote.Hello()
10      fmt.Printf(message)
11  }
```

В наведеному прикладі викликається функція **quote.Hello()** з пакета **"rsc.io/quote"**, яка повертає деяке повідомлення, а точніше рядок **Hello World**. Результат роботи програми наведено на рисунку 5.11.

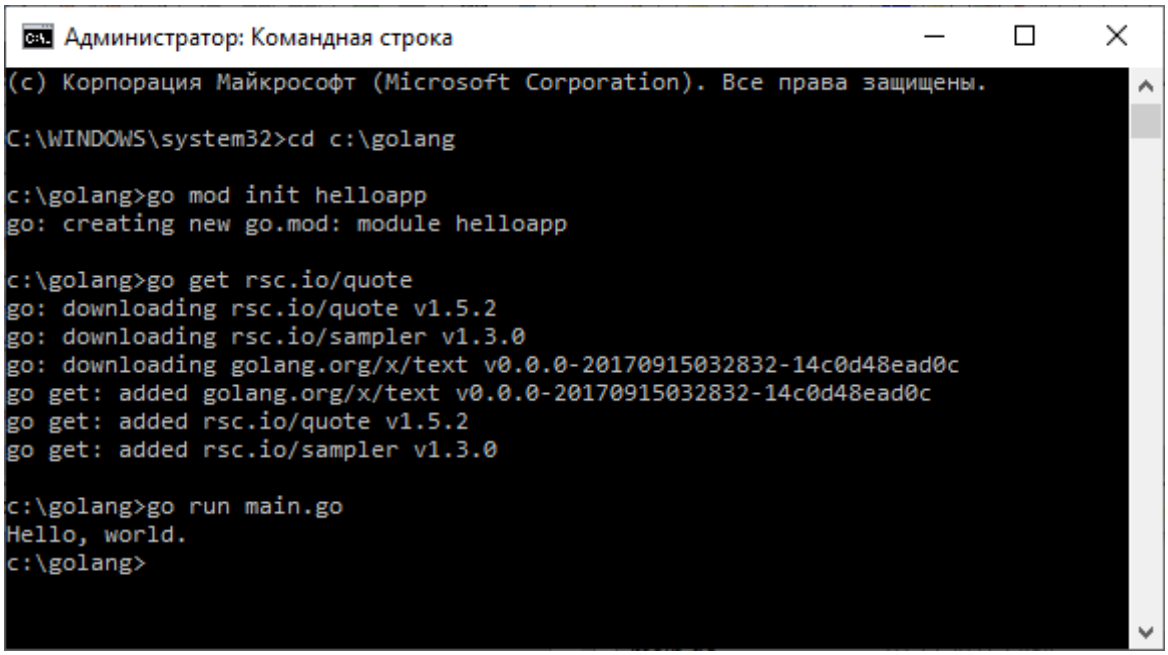

Рисунок 5.11. Результат роботи програми

Резюме

В розділі розглянуто організацію коду в мові Go за допомогою пакетів і модулів. Пакети використовуються для структурування коду, поділу на логічні частини та повторного використання функціональності. Основними типами є виконувані пакети (з пакетом **main**, що містить функцію **main**) і бібліотеки.

Імпорт пакетів здійснюється за допомогою директиви **import**, яка може включати вбудовані або користувацькі пакети. Показано приклади роботи з файлами, що належать до одного пакета, а також використання зовнішніх модулів. Для створення модулів застосовується команда **go mod init**, яка генерує файл **go.mod** для управління залежностями. Зовнішні модулі додаються командою **go get**, і їхні пакети підключаються як стандартні.

Результатом є можливість створювати масштабовані й організовані програми, інтегруючи в них сторонню функціональність та забезпечуючи простоту підтримки.

Наступні розділи будуть продовжувати послідовно висвітлювати все більш складні і важливі аспекти використання мови Go для програмування як вебзастосунків так і застосунків загального призначення.

Контрольні запитання і завдання

1. Що таке пакет у мові програмування Go і для чого він використовується?
2. Які два основні типи пакетів існують у Go?

3. Які особливості має пакет із назвою **main**?
4. Як виконується імпорт пакетів у Go?
5. Як організувати код, якщо пакет складається з кількох файлів?
6. Які переваги дає поділ програми на пакети?
7. Що означає директива **import** і як вона використовується?
8. Як імпортувати кілька пакетів у Go?
9. Чим відрізняється імпорт вбудованих пакетів від користувацьких?
10. Що означає шлях **./computation** у директиві імпорту?
11. Що таке модуль у Go і як він відрізняється від пакета?
12. Для чого використовується команда go mod init?
13. Що таке файл **go.mod** і яку інформацію він містить?
14. Як додати зовнішній модуль до проекту?
15. Що таке файл **go.sum** і яку роль він виконує?
16. Як Go забезпечує керування версіями модулів?
17. Як створити пакет, що містить функцію обчислення факторіала?
18. Як використовувати функцію з іншого пакета у головному файлі програми?
19. Як підключити зовнішній модуль **"rsc.io/quote"** і використати його функціональність?
20. Що відображає функція **quote.Hello()** з модуля **"rsc.io/quote"**?
21. Пояснити різницю між виконуваними пакетами (**executable**) та бібліотеками (**reusable**) у Go. Наведіть приклади.
22. Описати структуру пакета, що складається з кількох файлів. Які ключові моменти потрібно враховувати?
23. Напишіть короткий опис файлу go.mod і поясніть, яку інформацію він містить.
24. Роз'яснити, як і для чого використовується команда **go get**.
25. Пояснити механізм керування залежностями у Go. Яка роль файлів **go.mod** та **go.sum**?
26. Створіть програму на Go, яка використовує два власні пакети: один для обчислення факторіала числа, інший – для перевірки, чи є число простим.
27. Імпортуйте вбудовані пакети **math** та **fmt**, щоб обчислити квадратний корінь числа і вивести результат у консоль.
28. Напишіть програму, яка використовує зовнішній модуль **"rsc.io/quote"**. Використайте функцію **quote.Hello()** і відобразіть результат у консоль.

29. Організуйте проект із двох модулів, один з яких залежить від іншого. Використовуйте функції з залежного модуля у головному.
30. Напишіть програму, яка приймає число від користувача, обчислює його факторіал і виводить результат у консоль, використовуючи функцію з окремого пакета.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Яке ключове слово використовується для визначення пакета в Go?
a) module
b) import
c) package
d) func
Правильна відповідь: c).

Закриті запитання з кількома правильними відповідями

Які з наведених дій виконуються під час команди go get?
a) Завантаження модуля
b) Створення пакета
c) Оновлення залежностей
d) Компіляція програми
Правильна відповідь: a), c).

Вставити пропущені слова

Для імпорту пакета fmt у програму використовується команда:
_____ `"fmt"`
Правильна відповідь: ….import…

Завдання на відповідність

Встановіть відповідність між командами та їх функціями:
a) go mod init → 1. Ініціалізація нового модуля
b) go get →2. Завантаження зовнішнього модуля
c) go build → 3. Компіляція програми
Правильна відповідь:
a) → 1.

b) *→ 2.*
c) *→ 3.*

Завдання з вибором істинності

Пакет main обов'язково повинен містити функцію main.
a) Так
b) Ні
Правильна відповідь: a)

Завдання з кодом

Який код імпортує пакет fmt та виконує виведення на консоль?
a)

```
import (
    "fmt"
)
fmt.Println("Hello World")
```

b)

```
    package main
    import "fmt"
    func main() {
        fmt.Println("Hello World")
    }
```

Правильна відповідь: b)

Відкриті запитання

Що таке пакет у Go, і які його основні особливості?

Відповідь: Пакет у Go – це базова одиниця організації коду, яка дозволяє структурувати програму та повторно використовувати функціональність.

6. ІНТЕРФЕЙСИ

6.1. Поняття і призначення інтерфейсів

Інтерфейси є абстракцією поведінки інших типів. Інтерфейси дозволяють визначати функції, які прив'язані до конкретної реалізації. Тобто інтерфейси визначають певний функціонал, але не реалізують його.

Для визначення інтерфейсу застосовується ключове слово **interface**, наприклад:

```
type ім’я_інтерфейсу interface{
    визначення_сигнатури
}
```

Інший найпростіший спосіб визначення інтерфейсу виглядатиме наступним чином:

```
type vehicle interface{
    move()
}
```

В наведеному прикладі інтерфейс називається **vehicle**. Припустимо, цей інтерфейс є деяким транспортним засобом. Він визначає функцію **move()**, яка не приймає жодних параметрів та нічого не повертає.

При цьому важливо розуміти, що інтерфейс - це абстракція, а не конкретний тип, як от **int**, **sring** або структури. Наприклад, можна безпосередньо створити об'єкт інтерфейсу наступним чином:

```
var v vehicle = vehicle{}
```

Інтерфейс представляє свого роду контракт, якому має відповідати тип даних. Щоб тип даних відповідав деякому інтерфейсу, цей тип має реалізувати як методи всі функції цього інтерфейсу. Наприклад, визначимо дві структури наступним чином:

```
package main

import "fmt"

type Vehicle interface{
    move()
}

// Структура "Автомобіль"
type Car struct{ }
```

```
11
12 // Структура "Літак"
13 type Aircraft struct{}
14
15
16 func (c Car) move(){
17     fmt.Println("Автомобіль їде")
18 }
19 func (a Aircraft) move(){
20     fmt.Println("Літак летить")
21 }
22
23 func main() {
24
25     var tesla Vehicle = Car{}
26     var boing Vehicle = Aircraft{}
27     tesla.move()
28     boing.move()
29 }
```

В наведеному прикладі визначено дві структури: **Car** та **Aircraft**, які, описують, автомобіль та літак відповідно. Для кожної структури визначено метод **move()**, який імітує переміщення транспортного засобу. Цей метод **move** відповідає функції **move** інтерфейсу **Vehicle** за типом параметрів і типом значень, що повертаються. Оскільки між методом структур та функцій в інтерфейсі є відповідність, подібні структури неявно реалізують даний інтерфейс.

У мові Go інтерфейс реалізується неявно. Тобто, немає необхідності спеціально вказувати, що структури застосовують певний інтерфейс, як в деяких інших мовах програмування. Для реалізації типу даних достатньо реалізувати методи, що визначають інтерфейс.

Оскільки структури **Car** і **Aircraft** реалізують інтерфейс **Vehicle**, то можна визначити змінні даного інтерфейсу, передавши їм об'єкти структур, наприклад:

```
1  var tesla Vehicle = Car{}
2  var boing Vehicle = Aircraft{}
```

Де можуть використовуватись інтерфейси? Інтерфейси дозволяють визначити якусь узагальнену реалізацію без прив'язки до конкретного типу. Наприклад, розглянемо таку ситуацію:

```
1  package main
2
3  import "fmt"
4
5  type Car struct{ }
6  type Aircraft struct{}
7
8
9  func (c Car) move(){
10     fmt.Println("Автомобіль їде")
11 }
12 func (a Aircraft) move(){
13     fmt.Println("Літак летить")
14 }
15
16 func driveCar(c Car){
17     c.move()
18 }
19 func driveAircraft(a Aircraft){
20     a.move()
21 }
22
23 func main() {
24
25     var tesla Car = Car{}
26     var boing Aircraft = Aircraft{}
27     driveCar(tesla)
28     driveAircraft(boing)
29 }
30 }
```

В наведеному прикладі визначено дві структури **Car** та **Aircraft**, які представляють автомобіль та літак. Для кожної структури визначено метод переміщення **move()**, який умовно переміщає транспортний засіб. Також визначено дві функції **driveCar()** і **driveAircraft()**, які приймають відповідно структури **Car** та **Aircraft** і призначені для водіння цих транспортних засобів.

Видно, що обидві функції **driveCar** і **driveAircraft** фактично ідентичні, вони виконують ті самі дії, тільки для різних типів. І було б непогано, якби можна було б визначити одну узагальнену функцію для різних типів. Особливо

враховуючи, що у реальних програмах може бути і більше транспортних засобів: велосипед, пароплав, дрон тощо. І для керування кожним транспортним засобом доведеться визначати окремий метод, що не дуже зручно. Саме в цьому випадку можна скористатися інтерфейсами, наприклад:

```
1   package main
2   import "fmt"
3
4   type Vehicle interface{
5       move()
6   }
7
8   func drive(vehicle Vehicle){
9       vehicle.move()
10  }
11
12  type Car struct{ }
13  type Aircraft struct{}
14
15
16  func (c Car) move(){
17      fmt.Println("Автомобіль їде")
18  }
19  func (a Aircraft) move(){
20      fmt.Println("Літак летить")
21  }
22
23  func main() {
24
25      tesla := Car{}
26      boing := Aircraft{}
27      drive(tesla)
28      drive(boing)
29  }
```

Тепер замість двох функцій визначено одну загальну функцію **drive()**, яка в якості параметра приймає значення типу **Vehicle**. Оскільки цьому інтерфейсу відповідають обидві структури і **Car**, і **Aircraft**, то можна передавати ці структури у функцію **drive** як аргументи.

6.2. Відповідність інтерфейсу

Щоб тип даних відповідав інтерфейсу, він має реалізувати всі методи цього інтерфейсу, наприклад:

```
1  package main
2
3  import "fmt"
4
5  type Stream interface{
6      read() string
7      write(string)
8      close()
9  }
10
11 func writeToStream(stream Stream, text string){
12     stream.write(text)
13 }
14 func closeStream(stream Stream){
15     stream.close()
16 }
17
18 // Структура файл
19 type File struct{
20     text string
21 }
22 // Структура тека
23 type Folder struct{}
24
25 // Реалізація методів типу *File
26 func (f *File) read() string{
27     return f.text
28 }
29 func (f *File) write(message string){
30     f.text = message
31     fmt.Println("Запис рядка у файл ", message)
32 }
33 func (f *File) close(){
34     fmt.Println("Файл закрито")
```

```
35 }
36 // Реалізація методів типу *Folder
37 func (f *Folder) close(){
38     fmt.Println("Теку закрито")
39 }
40
41 func main() {
42
43     myFile := &File{}
44     myFolder := &Folder{}
45
46     writeToStream(myFile, "hello world")
47     closeStream(myFile)
48     //closeStream(myFolder)     // Помилка: тип
   *Folder не реалізує інтерфейс Stream
49     myFolder.close()            // А так можна
50 }
```

В наведеному прикладі визначено інтерфейс **Stream**, який умовно представляє певний потік даних та визначає три методи: **close**, **read** та **write**. І також є дві структури **File** та **Folder**, які представляють відповідно файл та теку. Для типу ***File** реалізовано всі три методи інтерфейсу **Stream**. А для типу ***Folder** реалізовано лише один метод інтерфейсу **Stream**. Тому тип ***File** реалізує інтерфейс **Stream** і відповідає цьому інтерфейсу, а тип ***Folder** не відповідає інтерфейсу **Stream**. Тому скрізь, де потрібний об'єкт **Stream**, можна буде використовувати об'єкт типу ***File**, але ніяк не об’єкт типу ***Folder**.

Наприклад, у функцію **closeStream()**, яка умовно закриває потік, як параметр передається об'єкт **Stream**. Під час виклику в цю функцію можна передати об'єкт типу ***File**, який відповідає інтерфейсу **Stream** наприклад:

```
1  closeStream(myFile)
```

А ось об'єкт ***Folder** передати не можна, наприклад:

```
1  closeStream(myFolder)        // Помилка: тип *Folder не
   реалізує інтерфейс Stream
```

Але, як і раніше, можна викликати у об'єкта ***Folder** метод **close**, але він розглядатиметься як власний метод, який не має жодного відношення до інтерфейсу **Stream**.

Ще важливо відзначити, що в наведеному прикладі методи реалізовані для типу ***File**, тобто покажчика на об'єкт **File**, а не для типу **File**. Це два різні

типи. Тому тип **File**, як і раніше, **НЕ** відповідає інтерфейсу **Stream**. І, наприклад, не можна написати наступне:

```
1   myFile2 := File{}
2   closeStream(myFile2)        //! Помилка: тип File не
    відповідає інтерфейсу Stream
```

6.2.1. Реалізація кількох інтерфейсів

У мові Go новий тип даних необов'язково повинен реалізувати лише методи інтерфейсу. Для нового типу даних можна визначити його власні методи або реалізувати методи інших інтерфейсів, наприклад:

```
1   package main
2
3   import "fmt"
4
5   type Reader interface{
6       read()
7   }
8
9   type Writer interface{
10      write(string)
11  }
12
13  func writeToStream(writer Writer, text string){
14      writer.write(text)
15  }
16  func readFromStream(reader Reader){
17      reader.read()
18  }
19
20  type File struct{
21      text string
22  }
23
24  func (f *File) read(){
25      fmt.Println(f.text)
26  }
27  func (f *File) write(message string){
```

```
28      f.text = message
29      fmt.Println("Запис рядка у файл ", message)
30  }
31
32  func main() {
33
34      myFile := &File{}
35      writeToStream(myFile, "hello world")
36      readFromStream(myFile)
37  }
```

В наведеному прикладі для типу ***File** реалізовані методи обох інтерфейсів і **Reader**, і **Writer**. Відповідно можна використовувати об'єкти типу ***File** як об'єкти **Reader** та **Writer**.

6.2.2. Вкладені інтерфейси

Одні інтерфейси можуть містити інші інтерфейси, наприклад:

```
1   type Reader interface{
2       read()
3   }
4
5   type Writer interface{
6       write(string)
7   }
8
9   type ReaderWriter interface{
10      Reader
11      Writer
12  }
```

Для забезпечення відповідності подібному інтерфейсу, типи даних повинні реалізувати всі його інтерфейси, наприклад:

```
1   package main
2
3   import "fmt"
4
5   type Reader interface{
6       read()
7   }
```

```
8
9  type Writer interface{
10     write(string)
11 }
12
13 type ReaderWriter interface{
14     Reader
15     Writer
16 }
17
18 func writeToStream(writer ReaderWriter, text string){
19     writer.write(text)
20 }
21 func readFromStream(reader ReaderWriter){
22     reader.read()
23 }
24
25 type File struct{
26     text string
27 }
28
29 func (f *File) read(){
30     fmt.Println(f.text)
31 }
32 func (f *File) write(message string){
33     f.text = message
34     fmt.Println("Запис рядка в файл ", message)
35 }
36
37 func main() {
38
39     myFile := &File{}
40     writeToStream(myFile, "hello world")
41     readFromStream(myFile)
42     writeToStream(myFile, "hello Go")
43     readFromStream(myFile)
44 }
```

В наведеному прикладі визначено три інтерфейси. Причому інтерфейс **ReaderWriter** містить інтерфейси **Reader** та **Writer**. Щоб структура **File** відповідала інтерфейсу **ReaderWriter**, вона має реалізувати методи **read** і **write**, тобто методи обох вкладених інтерфейсів, що тут і зроблено.

6.3. Поліморфізм

Поліморфізм описує здатність набувати різноманітних форм. Зокрема у попередніх розділах було розглянуто використання інтерфейсів, яким можуть відповідати різні структури, наприклад:

```
1   package main
2   import "fmt"
3
4   type Vehicle interface{
5       move()
6   }
7
8   type Car struct{ model string}
9   type Aircraft struct{ model string}
10
11
12  func (c Car) move(){
13      fmt.Println(c.model, "їде")
14  }
15  func (a Aircraft) move(){
16      fmt.Println(a.model, "летить")
17  }
18
19  func main() {
20
21      tesla := Car{"Tesla"}
22      volvo := Car{"Volvo"}
23      boeing := Aircraft{"Boeing"}
24
25      vehicles := [...]Vehicle{tesla, volvo, boeing}
26      for _, vehicle := range vehicles{
27          vehicle.move()
28      }
```

```
29 }
```

В наведеному прикладі визначено масив **vehicles**, який містить набір структур, які відповідають інтерфейсу **Vehicle**, тобто **Car** і **Aircraft**. Тобто об'єкт **Vehicle** може набувати різних форм: або структури **Car**, або структури **Aircraft**. При переборі масиву кожного об'єкта викликається метод **move**. І в залежності від реального типу структури динамічно визначається, яка саме реалізація метода **move** для якої структури повинна викликатися. Результат роботи програми наведено на рисунку 6.1.

```
Tesla їде
Volvo їде
Boeing летить
```

Рисунок 6.1. Результат роботи програми

Резюме

У матеріалі розділу було розглянуто концепцію інтерфейсів у мові програмування Go. Інтерфейси описують набір методів, які має реалізувати конкретний тип, забезпечуючи абстракцію поведінки. Реалізація інтерфейсів у Go неявна, тобто достатньо, щоб тип реалізував усі методи інтерфейсу. Наведені приклади демонструють, як використовувати інтерфейси для створення узагальнених функцій, наприклад, обробки різних типів транспортних засобів. Також розглянуто вкладені інтерфейси, поліморфізм, і те, як об'єкти можуть змінювати форми, реалізуючи різні інтерфейси. Інтерфейси дозволяють писати гнучкий і масштабований код, придатний для розширення.

Наступні розділи будуть продовжувати послідовно висвітлювати все більш складні і важливі аспекти використання мови Go для програмування як вебзастосунків так і застосунків загального призначення.

Контрольні запитання і завдання

1. Що таке інтерфейс у мові програмування Go?
2. Як визначається інтерфейс у Go? Наведіть приклад.
3. Чим інтерфейс відрізняється від конкретного типу, наприклад, **int** чи **string**?
4. Як неявно реалізується інтерфейс у Go?
5. Які умови повинні виконуватися, щоб тип відповідав певному інтерфейсу?

6. Чи потрібно явно вказувати, що структура реалізує інтерфейс?
7. Що станеться, якщо тип не реалізує всі методи інтерфейсу?
8. Як можна використовувати інтерфейси для створення узагальнених функцій?
9. У чому перевага використання інтерфейсів замість функцій, прив’язаних до конкретних структур?
10. Як передати об'єкт структури в змінну інтерфейсу? Наведіть приклад.
11. Що таке вкладені інтерфейси, і як вони працюють?
12. Як створити інтерфейс, що містить інші інтерфейси? Наведіть приклад.
13. Що таке поліморфізм, і як він пов'язаний з інтерфейсами?
14. Як викликаються методи для об'єкта, збереженого в змінній інтерфейсу?
15. Чи можна реалізувати кілька інтерфейсів одночасно? Як це зробити?
16. Чим відрізняються методи, реалізовані для типу і для вказівника на тип?
17. Що станеться, якщо передати у функцію інтерфейсу об'єкт, який частково реалізує його методи?
18. Як можна перевірити, чи відповідає об'єкт певному інтерфейсу?
19. Яку користь дає використання інтерфейсів у великих проектах?
20. Що станеться, якщо змінити визначення інтерфейсу, не оновивши реалізацію у структурах?
21. Створіть інтерфейс **Shape** з методом **Area() float64**. Реалізуйте його для двох структур: **Circle** (з радіусом) і **Rectangle** (з довжиною та шириною). Напишіть функцію, яка обчислює площу будь-якої фігури, що реалізує цей інтерфейс.
22. Реалізуйте інтерфейс **Printer** з методом **Print()**. Використайте структуру **Document**, яка має поле **Content**. Метод має виводити вміст документа на екран.
23. Створіть два інтерфейси: **Reader** (з методом **Read() string**) і **Writer** (з методом **Write(string**)). Реалізуйте їх у структурі **File**. Потім створіть інтерфейс **ReadWriter**, який поєднує ці два інтерфейси, і напишіть функцію для роботи з ним.
24. Створіть інтерфейс **Vehicle** з методом **Move()**. Реалізуйте його для структур **Car**, **Bike** і **Boat**. Збережіть об'єкти цих структур у масиві і викличте метод **Move()** для кожного з них у циклі.

25. Створіть інтерфейс **Closable** з методом **Close()**. Реалізуйте його для структур **File** і **Connection**. Створіть список об'єктів, де деякі реалізують **Closable**, а інші ні. Напишіть функцію, яка закриває лише ті об'єкти, що реалізують цей інтерфейс.
26. Створіть два інтерфейси: **Flyer** з методом **Fly()** і **Swimmer** з методом **Swim()**. Реалізуйте їх для структури **Duck**. Напишіть функції, які приймають об'єкти **Flyer** і **Swimmer**.
27. Створіть структуру **Laptop** з методом **Start()**. Створіть інтерфейс **Device** з методом **Start()**. Передайте об'єкт **Laptop** у змінну типу **Device** і викличте метод.
28. Реалізуйте інтерфейс **Calculator** з методом **Calculate(a, b int) int**. Реалізуйте цей інтерфейс для операцій додавання та множення, а також напишіть функцію, яка приймає об'єкти типу **Calculator** і обчислює результат.
29. Реалізуйте структуру **Person** і інтерфейс **Introducer** з методом **Introduce()**. Використайте конструкцію **type assertion**, щоб перевірити, чи реалізує об'єкт цей інтерфейс.
30. Створіть інтерфейс **Worker** з методом **Work()**. Реалізуйте його для структур **Employee** і **Robot**. Напишіть функцію, яка приймає змінну типу **Worker** і викликає її метод. Передайте об'єкти різних типів і перевірте результат.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Що описує інтерфейс у Go?
a) Структуру даних
b) Поведінку, яку має реалізувати тип
c) Конкретний тип даних
Правильна відповідь: b).

Як називається інтерфейс без методів у Go?
a) Порожній інтерфейс
b) Абстрактний інтерфейс
c) Базовий інтерфейс
Правильна відповідь: a).

Закриті запитання з кількома правильними відповідями

Які з тверджень є правильними щодо інтерфейсів у Go?

a) Інтерфейс визначає набір методів.

b) Інтерфейс реалізується неявно.

c) Типи даних мають явно вказувати, що вони реалізують інтерфейс.

d) Інтерфейси можна вкладати один в одного.

Правильна відповідь: a), b), d).

Завдання з вибором істинності

Чи можуть інтерфейси в Go містити поля?

a) Так

b) Ні

Правильна відповідь: b).

Завдання на відповідність

Співвіднесіть типи з їхньою відповідністю інтерфейсу Stream:

a) *File → 1. Реалізує всі методи інтерфейсу

b) *Folder → 2. Реалізує лише деякі методи

Правильна відповідь:

a) → *1.*

b) → *2.*

Відкриті запитання

Напишіть приклад інтерфейсу, який визначає метод Calculate(a int, b int) int.

Правильна відповідь:

```
type Calculator interface {
    Calculate(a int, b int) int
}
```

7. ПАРАЛЕЛЬНЕ ПРОГРАМУВАННЯ. ГОРУТИНИ

7.1. Горутини

Горутини описують паралельні операції, які можуть виконуватися незалежно від функції, в якій вони запущені. Головна особливість горутин полягає в тому, що вони можуть виконуватися паралельно. Тобто на багатоядерних архітектурах є можливість виконувати окремі горутини на окремих ядрах процесора, тим самим горутини виконуватимуться паралельно, і програма завершиться швидше.

Кожна горутина, як правило, представляє виклик функції і послідовно виконує всі свої інструкції. Коли запускається програма на Go, то починається виконуватись як мінімум одна горутина, яка представлена функцією **main**. Ця функція послідовно виконує всі інструкції, визначені в ній.

Для визначення горутин застосовується оператор **go**, який ставиться перед викликом функції, а саме:

```
go виклик функції
```

Наприклад, визначимо кілька горутин, що обчислюють факторіал числа:

```
package main
import "fmt"

func main() {

    for i := 1; i < 7; i++{
        go factorial(i)
    }
    fmt.Println("The End")
}

func factorial(n int){
    if(n < 1){
        fmt.Println("Invalid input number")
        return
    }
    result := 1
    for i := 1; i <= n; i++{
        result *= i
    }
```

```
21      fmt.Println(n, "-", result)
22  }
```

В наведеному прикладі у циклі послідовно запускаються шість горутин за допомогою **go factorial(i)**. Тобто, фактично, це звичайний виклик функції з оператором **go**.

Однак замість шести факторіалів на консолі під час виклику програми «можна побачити» лише рядок **"The End"**, як наведено на рисунку 7.1.

Рисунок 7.1. Результат роботи програми

«Можна побачити» означає, що поведінка програми не детермінована. Наприклад результат роботи програми може бути іншим, а саме таким, який наведено на рисунку 7.2.

Рисунок 7.2. Результат роботи програми

З’ясуємо, чому так відбувається. Після виклику **go factorial(i)** функція **main** запускає горутину, яка починає виконуватися у своєму контексті, незалежно від функції **main**. Тобто фактично **main** та **factorial** виконуються паралельно. Однак головною горутиною є виклик функції **main**. І якщо завершується виконання цієї функції, то завершується виконання всієї програми. Оскільки виклики функції **factorial** представляють горутини, то функція **main** не чекає на їх завершення і після їх запуску продовжує своє виконання. Якісь горутини можуть завершитися раніше функції **main**, і можна буде побачити на консолі їх результат. Але може скластися ситуація, що функція **main** виконається раніше викликів функції **factorial**. В цьому разі на консоль факторіали чисел не будуть виведені.

Щоб все ж таки побачити результат виконання горутин, запишемо в кінці функції **main** виклик функції **fmt.Scanln()**, яка очікує введення користувачем числа з консолі, наприклад:

```
1   package main
2   import "fmt"
```

```
3
4   func main() {
5
6       for i := 1; i < 7; i++{
7           go factorial(i)
8       }
9       fmt.Scanln()        // Чекаємо введення даних від
    користувача
10      fmt.Println("The End")
11  }
12
13  func factorial(n int){
14      if(n < 1){
15          fmt.Println("Unvalid input number")
16          return
17      }
18      result := 1
19      for i := 1; i <= n; i++{
20          result *= i
21      }
22      fmt.Println(n, "-", result)
23  }
```

Тепер можна буде побачити результати всіх викликів функції **factorial**, як показано на рисунку 7.3.

```
1 - 1
3 - 6
5 - 120
4 - 24
2 - 2
6 - 720

The End
```

Рисунок 7.3. Результат роботи програми

Варто зазначити, що так як кожна горутина запускається у своєму власному контексті і виконується незалежно і паралельно в порівнянні з іншими горутинами, то в даному випадку не можна чітко передбачити, яка з

горутин завершиться раніше. Наприклад, горутина **go factorial(2)** запускається до **go factorial(5)**, проте може завершитися і після.

Горутини також можуть представляти виклики анонімних функцій, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       for i := 1; i < 7; i++{
7
8           go func(n int){
9               result := 1
10              for j := 1; j <= n; j++{
11                  result *= j
12              }
13              fmt.Println(n, "-", result)
14          }(i)
15      }
16      fmt.Scanln()
17      fmt.Println("The End")
18  }
```

7.2. Канали

Канали (**channels**) представляють інструменти комунікації між горутинами. Для визначення каналу застосовується ключове слово **chan**, а саме:

```
1   chan тип_елемента
```

Після слова **chan** вказується тип даних, які передаються за допомогою каналу, наприклад:

```
1   var intCh chan int
```

В наведеному прикладі змінна **intCh** позначає собою канал, який передає дані типу **int**.

Для передачі в канал чи, навпаки, з каналу застосовується операція <- (спрямована вліво стрілка).

Наприклад, передача даних у канал виглядатиме наступним чином:

```
1   intCh <- 5
```

В наведеному прикладі в канал надсилається число 5.

А отримання даних з каналу в змінну виглядатиме наступним чином:

```
1   val := <- intCh
```

Якщо раніше у канал було відправлено число 5, то, при виконанні операції **<- intCh** можна одержати це число у змінній **val**.

Варто враховувати, що можна відправити в канал і отримати з каналу дані тільки такого типу, який представляє канал. Так, у прикладі з каналом **intCh** це дані типу **int**.

Як правило, відправником даних є одна горутина, а одержувачем буде інша горутина.

При простому визначенні змінної каналу вона має значення **nil**, тобто, по суті, канал не ініціалізований. Для ініціалізації використовується функція **make()**. Залежно від визначення ємності каналу, він може бути буферизованим або небуферизованим.

7.2.1. Небуферизовані канали

Для створення небуферизованого каналу викликається функція **make()** без вказівки ємності каналу, а саме:

```
1   var intCh chan int = make(chan int) // Канал для даних
    типу int
2   strCh := make(chan string)  // Канал для даних типу
    string
```

Якщо канал порожній, горутина-одержувач блокується, поки в каналі не з’являться дані. Коли горутина-відправник надсилає дані, горутина-одержувач отримує ці дані та відновлює роботу.

Горутина-відправник може надсилати дані лише у порожній канал. Горутина-відправник блокується доти, доки дані з каналу не будуть отримані, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int)
7
8       go func(){
9               fmt.Println("Go routine starts")
10              intCh <- 5 // Блокування, поки дані не
```

```
    будуть отримані функцією main
11      }()
12      fmt.Println(<-intCh) // Отримання даних із каналу
13      fmt.Println("The End")
14  }
```

Через небуферизований канал **intCh** горутина, представлена анонімною функцією, передає число 5, а саме:

```
1   intCh <- 5
```

А функція **main** отримує це число:

```
1   fmt.Println(<-intCh)
```

Загальний хід виконання програми виглядає наступним чином:

- запускається функція **main**. Вона створює канал **intCh** і запускає горутин у вигляді анонімної функції;
- функція **main** продовжує виконуватись і блокується на рядку **fmt.Println(<-intCh)**, доки не будуть отримані дані;
- паралельно виконується запущена горутина у вигляді анонімної функції. Наприкінці свого виконання вона надсилає дані через канал **intCh <- 5**. Горутина блокується, поки функція **main** не отримає дані;
- функція **main** отримує надіслані дані, деблокується і продовжує свою роботу.

У наведеному прикладі горутина визначена у вигляді анонімної функції і тому вона має доступ до оточення, у тому числі змінної **intCh**. Якщо робота ведеться зі звичайними функціями, то об'єкти каналів треба передавати через параметри, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int)
7
8       go factorial(5, intCh)   // Виклик горутини
9       fmt.Println(<-intCh) // Отримання даних із каналу
10      fmt.Println("The End")
11  }
12
13  func factorial(n int, ch chan int){
14
```

```
15      result := 1
16      for i := 1; i <= n; i++{
17          result *= i
18      }
19      fmt.Println(n, "-", result)
20
21      ch <- result      // Відправлення даних у канал
22  }
```

У наведеному прикладі варто звернути увагу, як визначається параметр, який є каналом даних типу **int: ch chan int**. Результат роботи програми наведено на рисунку 7.4.

```
5 - 120
120
The End
```

Рисунок 7.4. Результат роботи програми

Таким чином, при використанні каналу викликаючий потік, функція **main** в наведеному прикладі, очікує завершення виконання горутини.

Варто зазначити, що одна горутина повинна відправляти дані, а інша їх отримувати. Наприклад, якщо визначити відправлення та отримання даних через канал у самій функції **main**, то відразу зіткнемося із взаємоблокуванням, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int)
7       intCh <- 10       // Функція main блокується
8       fmt.Println(<-intCh)
9   }
```

Від цієї ситуації слід відрізняти ситуацію, коли дві горутини по черзі обмінюються даними, при цьому одна горутина виступає відправником, а інша одержувачем, наприклад:

```
1   package main
2   import "fmt"
3
```

```
4
5   func main() {
6
7       intCh := make(chan int)
8
9       go square(intCh)          // square очікує отримання
    через канал
10      intCh <- 4                // Відправляємо в канал
    число
11      fmt.Println("result := ", <-intCh)  // Отримуємо з
    каналу результат
        fmt.Println("The End")
12  }
13  // Функція зведення у квадрат
14  func square(ch chan int){
15
16      num := <-ch                    // Отримуємо з каналу
17  число
18      fmt.Println("num := ", num)
19      ch <- num * num                // Відправляємо
    квадрат числа назад
20  }
```

У наведеному прикладі визначено функцію **square**, яка отримує через канал число, зводить його у квадрат і повертає назад у канал. Функція **main** запускає функцію **square** у вигляді горутини, відправляє в канал деяке число і очікує отримати у відповідь з каналу квадрат цього числа.

У результаті спочатку запускається горутина **square** а саме:

```
9   go square(intCh)
```

Горутина **square** блокується на наступному рядку:

```
16  num := <-ch
```

У цей момент одержувачем є горутина **square**, а відправником функція **main**. І функція **main** відправляє дані до потоку за допомогою наступного рядка:

```
10  intCh <- 4
```

Після цього горутина **square** отримує з каналу число, і ролі змінюються: тепер відправником стає горутина **square**, а одержувачем стає функція **main**, яка в очікуванні даних блокується на наступному рядку:

```
11  fmt.Println("result := ", <-intCh)
```

Функція **square** обробляє отримане число і відправляє квадрат числа потік, за допомогою рядка:

```
19  ch <- num * num
```

Функція **main** отримує з каналу квадрат числа та завершує свою роботу. Результат роботи програми наведено на рисунку 7.5.

```
num := 4
result := 16
The End
```

Рисунок 7.5. Результат роботи програми

7.2.2. Буферизовані канали

Буферизовані канали також створюються за допомогою функції **make()**, тільки як другий аргумент у функцію передається ємність каналу. Якщо канал порожній, то одержувач чекає, поки у каналі з'явиться хоча б один елемент.

При надсиланні даних горутина-відправник очікує, доки у каналі не звільниться місце ще для одного елемента і відправляє елемент лише тоді, коли у каналі звільняється для нього місце, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int, 3)
7       intCh <- 10
8       intCh <- 3
9       intCh <- 24
10      fmt.Println(<-intCh)      // 10
11      fmt.Println(<-intCh)      // 3
12      fmt.Println(<-intCh)      //24
13  }
```

У наведеному прикладі відправником і одержувачем даних є функція **main**. У ній створюється канал із трьох елементів, і послідовно відправляються три значення типу **int**.

У той же час має бути відповідність між кількістю даних, що надсилаються і одержуються. Якщо функції **main** буде одночасно відправлено значень більше, ніж вміщає канал, то функція заблокується, наприклад:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int, 3)
7       intCh <- 10
8       intCh <- 3
9       intCh <- 24
10      intCh <- 15  // Блокування - функція main чекає,
    коли звільниться місце в каналі
11
12      fmt.Println(<-intCh)
13      fmt.Println("The End")
14  }
```

За допомогою вбудованих функцій **cap()** і **len()** можна отримати відповідно ємність та кількість елементів у каналі:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int, 3)
7       intCh <- 10
8
9       fmt.Println(cap(intCh))      // 3
10      fmt.Println(len(intCh))      // 1
11
12      fmt.Println(<-intCh)
13  }
```

7.2.3. Односпрямовані канали

У мові Go можна визначити канал, як доступний лише для надсилання даних або лише для отримання даних.

Визначення каналу лише для надсилання даних виглядає наступним чином:

```
1   var inCh chan<- int
```

Визначення каналу лише для отримання даних виглядає наступним чином:

```
var outCh <-chan int
```

Розглянемо наступний приклад:

```
package main
import "fmt"

func main() {

    intCh := make(chan int, 2)
    go factorial(5, intCh)
    fmt.Println(<-intCh)
    fmt.Println("The End")
}

func factorial(n int, ch chan<- int){

    result := 1
    for i := 1; i <= n; i++{
        result *= i
    }
    ch <- result
}
```

В наведеному прикладі другий параметр функції **factorial** визначений як канал, доступний лише для надсилання даних: **ch chan<- int**. Відповідно всередині функції **factorial** можна лише надсилати дані в канал, але не отримувати їх.

7.2.4. Повернення каналу

Канал може бути значенням функції, що повертається. Однак слід уважно підходити до операцій запису і читання у каналі, що повертається, наприклад:

```
package main
import "fmt"

func main() {
    fmt.Println("Start")
     // Створення каналу та отримання з нього даних
```

```
7       fmt.Println(<-createChan(5)) // Блокування
8       fmt.Println("End")
9   }
10  func createChan(n int) chan int{
11      ch := make(chan int)     // Створюємо канал
12      ch <- n       // Надсилаємо дані в канал
13      return ch    // Повертаємо канал
14  }
```

Функція **createChan** повертає канал. Однак при виконанні операції **ch<-n** зіткнемося з блокуванням, оскільки відбувається очікування на отримання даних з каналу. Тому наступний вираз **return ch** не виконуватиметься.

І якщо все ж таки необхідно визначити функцію, яка повертає канал, то всі операції читання-запису в канал слід винести в окрему горутину, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5       fmt.Println("Start")
6        // Створення каналу та отримання з нього даних
7       fmt.Println(<-createChan(5)) // 5
8       fmt.Println("End")
9   }
10  func createChan(n int) chan int{
11      ch := make(chan int)     // Створюємо канал
12      go func(){
13          ch <- n       // Надсилаємо дані в канал
14      }()              // Запускаємо горутину
15      return ch    // Повертаємо канал
16  }
```

7.3. Закриття каналу

Після ініціалізації канал готовий передавати дані. Він знаходиться у відкритому стані, і можна з ним взаємодіяти, доки його не буде закрито за допомогою вбудованої функції **close()**, наприклад:

```
1   package main
2   import "fmt"
3
```

```
4   func main() {
5
6       intCh := make(chan int, 3)
7       intCh <- 10
8       intCh <- 3
9       close(intCh)
10      // intCh <- 24        // Помилка, адже канал вже
    закритий
11      fmt.Println(<-intCh)      // 10
12      fmt.Println(<-intCh)      // 3
13      fmt.Println(<-intCh)      // 0
14  }
```

Після закриття каналу стає вже неможливим надсилання до нього нових даних. Якщо спробувати це зробити, то згенерується помилка. Однак, можна отримати раніше додані дані. Але при спробі отримати дані з каналу, яких вже немає, буде отримано значення за замовчуванням. Наприклад, у прикладі вище в канал додаються два значення. При спробі отримати третє значення, якого немає в каналі, буде отримано значення за замовчуванням, а саме 0.

Щоб не зіткнутися з проблемою, коли канал вже закрито, можна перевірити стан каналу. Зокрема з каналу можна отримати наступні два значення:

```
1   val, opened:= <-intCh
```

Перше значення **val** – це власне дані з каналу, а **opened** представляє логічне значення, яке дорівнює **true**, якщо канал відкритий і можна успішно отримувати з нього дані. Наприклад, стан каналу можна перевіряти за допомогою умовної конструкції:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int, 3)
7       intCh <- 10
8       intCh <- 3
9       close(intCh)     // Канал закрито
10
11      for i := 0; i < cap(intCh); i++ {
12          if val, opened := <-intCh; opened {
```

```
13              fmt.Println(val)
14          } else {
15              fmt.Println("Channel closed!")
16          }
17      }
18  }
```

Результат роботи програми наведено на рисунку 7.6.

```
10
3
Channel closed!
```

Рисунок 7.6. Результат роботи програми

7.4. Синхронізація

Використання каналів відкриває можливості синхронізації між різними горутинами. Наприклад, одна горутина виконує певну дію, результат якої використовується в іншій горутині. У цьому плані можна використовувати канали синхронізації. Наприклад, одна горутина обчислює факторіал числа, а результат виводиться в іншій горутині, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6       intCh := make(chan int)
7
8       go factorial(5, intCh)
9
10      fmt.Println(<-intCh)
11  }
12
13  func factorial(n int, ch chan int){
14       result := 1
15       for i:=1; i <= n; i++{
16           result *= i
17       }
```

```
18      ch <- result
19  }
```

Канал не обов'язково повинен містити дані, які становлять певний результат, від якого залежить подальше виконання горутини. Іноді це може бути холостий об'єкт, наприклад порожня структура, яка необхідна тільки для синхронізації горутин, а саме:

```
1   package main
2   import "fmt"
3
4   func main() {
5
6      results := make(map[int]int)
7      structCh := make(chan struct{})
8
9      go factorial(5, structCh, results)
10
11     <-structCh          // Очікуємо закриття каналу
    structCh
12
13     for i, v := range results{
14      fmt.Println(i, " - ", v)
15     }
16  }
17
18  func factorial(n int, ch chan struct{}, results
    map[int]int){
19      defer close(ch)      // Закриваємо канал після
    завершення горутини
20      result := 1
21      for i:=1; i <= n; i++{
22          result *= i
23          results[i] = result
24      }
25  }
```

В наведеному прикладі функція **factorial**, як і раніше, обчислює факторіал, але поміщає всі факторіали чисел від **1** до **n** у відображення результатів, де ключі представляють числа, а значеннями є факторіали чисел.

Канал, через який горутини взаємодіють, є типом **chan struct{}**. Причому функція **factorial** не відправляє конкретні дані у канал, а просто закриває його після виконання всіх своїх інструкцій за допомогою виклику **defer close (ch)**. Після закриття каналу функції **main** отримує відповідний сигнал у рядку **<-structCh**, і після цього функція **main** продовжує свою роботу.

7.5. Передача потоків даних

Нерідко одна горутина транслює іншій горутині через канал не одиночні значення, а певний потік даних. У цьому випадку загальний алгоритм полягає в тому, що горутина-відправник протягом деякого періоду надсилає дані. Коли дані для надсилання закінчилися, робота виконана, відправник закриває канал.

Горутина-одержувач у нескінченному циклі отримує дані з каналу. Якщо буде отримано маркер закриття каналу, здійснюється вихід з нескінченного циклу, наприклад.

```
1   package main
2   import "fmt"
3
4   func main(){
5      intCh := make(chan int)
6
7      go factorial(7, intCh)
8
9      for {
10           num, opened := <- intCh     // Отримуємо
    дані з потоку
11           if !opened {
12                break    // Якщо потік закритий, то
    вихід із циклу
13           }
14           fmt.Println(num)
15     }
16  }
17
18  func factorial(n int, ch chan int){
19      defer close(ch)
20      result := 1
21      for i := 1; i <= n; i++{
```

```
22          result *= i
23          ch <- result      // Надсилаємо числа
24      }
25  }
```

В наведеному прикладі функція **main** і горутин **factorial** взаємодіють через канал **intCh**. Функція **factorial** послідовно обчислює факторіали чисел від **1** до **n**. І всі обчислені значення передаються до каналу. Після завершення функції **factorial** канал закривається викликом **defer close (ch)**.

У функції **main** в нескінченному циклі надіслані дані отримуються з каналу. При цьому також перевіряється, чи канал відкритий. Коли раптом канал закритий і відповідно немає сенсу отримувати з нього дані, то відбувається вихід з нескінченного циклу, а саме:

```
1   for {
2       num, opened := <- intCh
3       if !opened {
4           break
5       }
6       fmt.Println(num)
7   }
```

Результат роботи програми наведено на рисунку 7.7.

```
1
2
6
24
120
720
5040
```

Рисунок 7.7. Результат роботи програми

При отриманні значень з каналу можна використовувати таку форму циклу **for**, яка застосовується для перебору колекцій, а саме:

```
1   for змінна := канал{
2       //...........Дії
3   }
```

Перепишемо попередній приклад наступним чином:

```
1   package main
2   import "fmt"
```

```
3
4   func main(){
5      intCh := make(chan int)
6
7      go factorial(7, intCh)
8
9       for num := range intCh{
10          fmt.Println(num)
11      }
12  }
13
14  func factorial(n int, ch chan int){
15      defer close(ch)
16      result := 1
17      for i := 1; i <= n; i++{
18          result *= i
19          ch <- result
20      }
21  }
```

Тоді, коли канал буде закритий, автоматично відбудеться вихід із циклу **for**.

7.6. М'ютекси

Для спрощення синхронізації між горутинами у Go є пакет **sync**, який надає ряд можливостей, зокрема м’ютекси. М'ютекси дозволяють розмежувати доступ до деяких загальних ресурсів, гарантуючи, що тільки одна горутина має до них доступ у певний момент часу. І доки одна горутина не звільнить загальний ресурс, інша горутина не може з ним працювати.

На рівні коду м'ютекс є типом **sync.Mutex**. Для блокування доступу до загального ресурсу, що розділяється, у м'ютексі викликається метод **Lock()**, а для розблокування доступу викликається метод **Unlock()**.

З’ясуємо у якій ситуації можуть допомогти м'ютекси. Розглянемо наступний приклад:

```
1   package main
2   import "fmt"
3
4   var counter int = 0              //  Спільний ресурс
```

```
5   func main() {
6
7       ch := make(chan bool)          // Канал
8       for i := 1; i < 5; i++{
9           go work(i, ch)
10      }
11      // Очікуємо завершення всіх горутин
12      for i := 1; i < 5; i++{
13          <-ch
14      }
15      fmt.Println("The End")
16  }
17  func work (number int, ch chan bool){
18      counter = 0
19      for k := 1; k <= 5; k++{
20          counter++
21          fmt.Println("Goroutine", number, "-",
22  counter)
23      }
24      ch <- true
    }
```

В наведеному прикладі функція **work** скидає значення змінної **counter** до нуля і в циклі послідовно збільшує її значення до 5. У функції **main** запускається чотири горутини **work**. Але який у цьому випадку буде мати результат роботи програми на консолі? Він може бути, наприклад, таким, як показано на рисунку 7.8.

Незважаючи на те, що в кожній горутині значення **counter** скидається до 0, а потім збільшується до 5, бачимо, що кілька горутин після скидання змінної працюють зовсім з іншим значенням. Тобто при запуску горутин кожна з них отримує значення змінної **counter** і починає працювати з нею. Поки одна горутина ще не закінчила роботу з **counter** у циклі, з цією ж змінною починає працювати й інша горутина. Тобто з одним і тим же спільним ресурсом, що розділяється, а саме зі змінною **counter** одночасно працюють відразу кілька горутин. Це може призвести до некоректних результатів, як у наведеному вище прикладі.

```
Goroutine 3 - 1
Goroutine 3 - 2
Goroutine 3 - 3
Goroutine 3 - 4
Goroutine 3 - 5
Goroutine 2 - 1
Goroutine 2 - 6
Goroutine 2 - 7
Goroutine 2 - 8
Goroutine 2 - 9
Goroutine 1 - 1
Goroutine 1 - 10
Goroutine 1 - 11
Goroutine 1 - 12
Goroutine 1 - 13
Goroutine 4 - 1
Goroutine 4 - 14
Goroutine 4 - 15
Goroutine 4 - 16
Goroutine 4 - 17
The End
```

Рисунок 7.8. Результат роботи програми

За допомогою м'ютексів можна обмежити доступ до змінної таким чином, щоб тільки одна горутина мала до неї монопольний доступ в один момент часу, наприклад:

```
1   package main
2   import (
3   "fmt"
4   "sync"
5   )
6
7   var counter int = 0                  //  Спільний ресурс
8   func main() {
9
10      ch := make(chan bool)            // Канал
11      var mutex sync.Mutex             // Визначаємо м'ютекс
12      for i := 1; i < 5; i++{
```

```
13          go work(i, ch, &mutex)
14      }
15
16      for i := 1; i < 5; i++{
17          <-ch
18      }
19
20      fmt.Println("The End")
21  }
    func work (number int, ch chan bool, mutex
22  *sync.Mutex){
        mutex.Lock()     // Блокуємо доступ до змінної
23  counter
24      counter = 0
25      for k := 1; k <= 5; k++{
26          counter++
27          fmt.Println("Goroutine", number, "-", counter)
28      }
29      mutex.Unlock()  // Деблокуємо доступ
30      ch <- true
31  }
```

Тепер функція **work** приймає покажчик на м'ютекс. За допомогою виклику **mutex.Lock()** м'ютекс блокується даною горутиною. Це означає, що до наступного коду має доступ тільки та горутина, яка перша заблокувала м'ютекс. Інші горутини чекають поки, м'ютекс звільниться.

Далі горутина скидає значення змінної **counter** до нуля і потім у циклі послідовно збільшує його. В кінці, коли всі дії із загальним ресурсом вже виконані, горутина звільняє м'ютекс за допомогою виклику **mutex.Unlock()**. Очікуючі горутини отримують сигнал, що м'ютекс звільнився, і одна з горутин блокує м'ютекс і починає виконувати дії зі змінною **counter**. І так далі горутини послідовно захоплюють та звільняють м'ютекс. У підсумку до наступної секції коду:

```
1   mutex.Lock()     // Блокуємо доступ до змінної counter
2   counter = 0
3   for k := 1; k <= 5; k++{
4       counter++
5       fmt.Println("Goroutine", number, "-", counter)
6   }
```

```
7   mutex.Unlock()   // Деблокуємо доступ
```

матиме доступ тільки та горутина, яка перша заблокувала м'ютекс. Як наслідок отримаємо результат, як показано на рисунку 7.9.

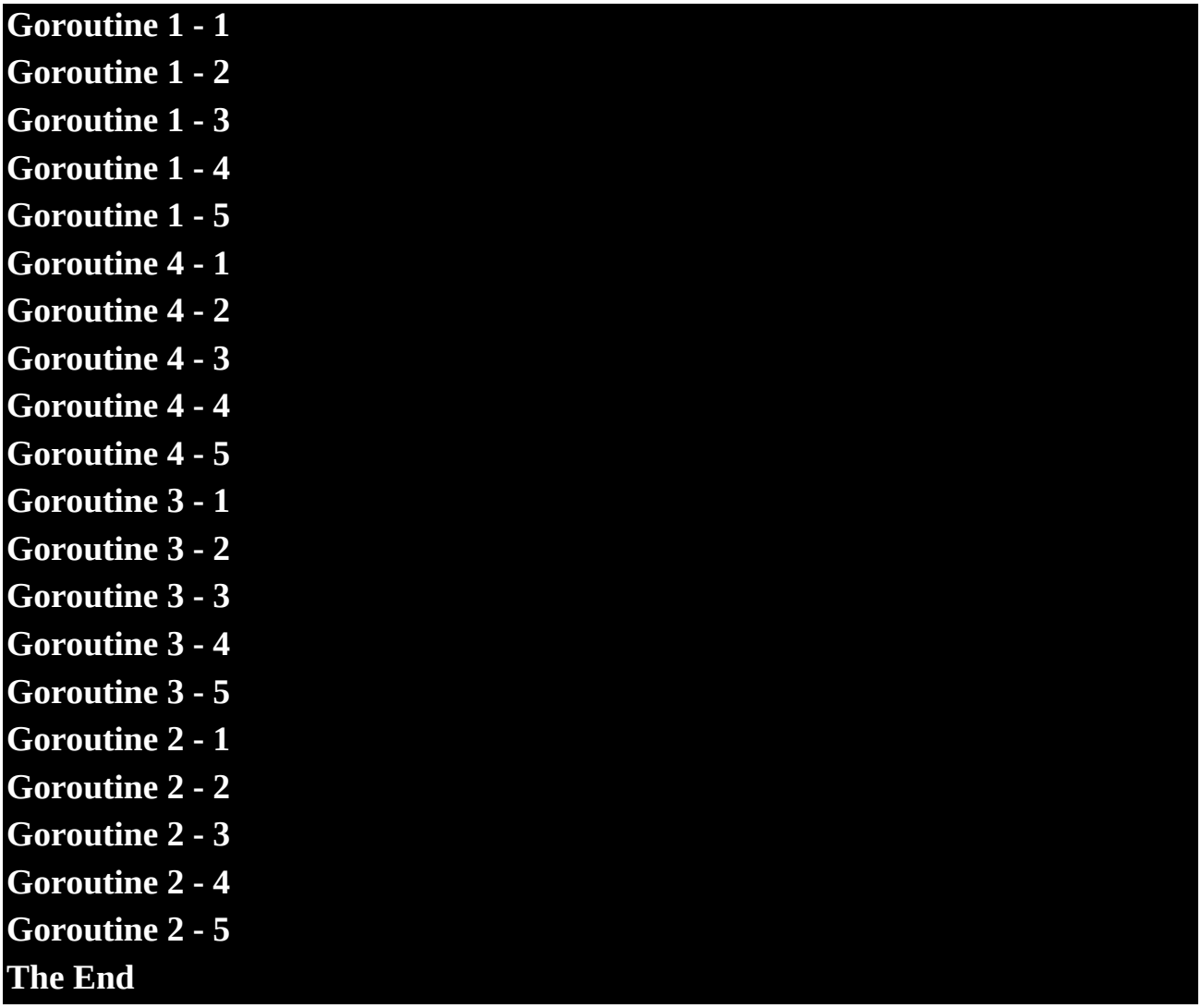

Рисунок 7.9. Результат роботи програми

7.7. Синхронізація горутин за допомогою типу WaitGroup

Ще одну можливість синхронізації горутин представляє використання типу **sync.WaitGroup**. Цей тип дозволяє визначити групу горутин, які повинні виконуватися разом як одна група. Можна також встановити блокування, яке призупинить виконання функції, доки не завершить виконання вся група горутин, наприклад:

```
1   package main
2   import (
3   "fmt"
```

```
4   "sync"
5   "time"
6   )
7
8
9   func main() {
10      var wg sync.WaitGroup
11      wg.Add(2)         // У групі дві горутини
12      work := func(id int) {
13          defer wg.Done()
14          fmt.Printf("Горутина %d почала виконання \n",
    id)
15          time.Sleep(2 * time.Second)
16          fmt.Printf("Горутина %d завершила виконання
    \n", id)
17      }
18
19      // Викликаємо горутину
20      go work(1)
21      go work(2)
22
23      wg.Wait()         // Очікуємо завершення обох
    горутин
24      fmt.Println("Горутини завершили виконання")
25  }
```

В наведеному прикладі спочатку визначено групу як змінну **wg sync.WaitGroup**. За допомогою метода **Add** визначено, що група складатиметься із двох елементів, а саме:

```
1   wg.Add(2)
```

Число, що передається в метод **Add**, визначає значення внутрішнього лічильника активних елементів.

Всі елементи групи **wg** будуть представляти анонімну функцію у вигляді змінної **work**, яка як параметр приймає умовний числовий ідентифікатор горутини. Ця функція буде викликатись у вигляді горутин. Щоб сигналізувати, що елемент групи завершив своє виконання, у горутині необхідно викликати метод **Done()**, а саме:

```
1   defer wg.Done()
```

Виклик метода **wg.Done()** зменшує внутрішній лічильник активних елементів на одиницю.

У самій функції **work()** за допомогою затримки часу на дві секунди **(time.Sleep(2 * time.Second))** імітується тривала робота горутини.

Далі викликаються дві горутини, а саме:

```
1   go work(1)
2   go work(2)
```

Причому кількість горутин, які викликають метод **wg.Done()** має відповідати кількості елементів групи **wg**, тобто 2 елемента для наведеного прикладу.

Потім викликається метод **Wait()**, який очікує завершення всіх горутин із групи **wg**, а саме:

```
1   wg.Wait()
```

Метод деблокує функцію **main**, коли внутрішній лічильник активних елементів у групі **wg** стає рівним 0. Тому коли всі горутини з групи **wg** завершать виконання, функція **main** продовжить свою роботу.

Результат роботи програми наведено на рисунку 7.9.

```
Горутина 1 почала виконання
Горутина 2 почала виконання
Горутина 1 завершила виконання
Горутина 2 завершила виконання
Горутини завершили виконання
```

Рисунок 7.10. Результат роботи програми

Резюме

Матеріал розділу пояснює концепцію паралельного програмування в мові Go, зокрема горутини та канали. Горутини – це легковагові потоки, які дозволяють виконувати код паралельно, незалежно від функції, в якій вони були запущені. Основна горутина завжди представлена функцією **main**. Канали забезпечують обмін даними між горутинами і можуть бути буферизованими або небуферизованими, залежно від потреб. У матеріалі описано роботу з каналами, включно з їх створенням, передачею даних і закриттям, а також наведено приклади синхронізації між горутинами. Особливу увагу приділено використанню анонімних функцій та передачі каналів як параметрів для досягнення ефективної комунікації.

Також розглянуто принципи передачі потоків даних та синхронізації в горутинах мови Go. Описано використання каналів для обміну даними між горутинами, де закриття каналу сигналізує завершення передачі даних. Наведено приклад обчислення факторіалів із використанням каналів. Також висвітлено проблему одночасного доступу до спільних ресурсів різними горутинами, що може призводити до некоректних результатів. Для вирішення цієї проблеми запропоновано використання м'ютексів із пакета **sync**, які забезпечують монопольний доступ до ресурсу в будь-який момент часу. Окремо описано механізм синхронізації з використанням **sync.WaitGroup**, який дозволяє координувати виконання групи горутин і чекати їх завершення перед продовженням роботи основної програми.

Наступні розділи будуть продовжувати послідовно висвітлювати все більш складні і важливі аспекти використання мови Go для програмування як вебзастосунків так і застосунків загального призначення.

Контрольні запитання і завдання

1. Що таке горутина в мові Go?
2. Як запустити функцію як горутину?
3. Яка функція в Go завжди є головною горутиною?
4. Чому результат роботи горутин може бути недетермінованим?
5. Як змусити програму чекати завершення виконання всіх горутин?
6. Що таке канал у Go, і для чого він використовується?
7. У чому відмінність буферизованих і небуферизованих каналів?
8. Що відбувається, якщо відправляти дані в закритий канал?
9. Як перевірити, чи канал відкритий для передачі даних?
10. Як сигналізувати про завершення передачі даних через канал?
11. У чому відмінність між звичайним циклом **for** і циклом **for range** при отриманні даних з каналу?
12. Для чого використовуються м'ютекси в Go?
13. Які методи м'ютекса дозволяють блокувати і розблокувати доступ до ресурсу?
14. Для чого використовується тип **sync.WaitGroup**?
15. Який метод **WaitGroup** використовується для сигналізації про завершення роботи горутиною?
16. Що відбудеться, якщо викликати **wg.Wait()** до завершення всіх горутин?

17. У чому полягає основна відмінність використання каналів і м’ютексів для синхронізації горутин?
18. Як використати **defer** для автоматичного звільнення м’ютекса або сигналізації про завершення роботи?
19. Чому важливо уникати неконтрольованого доступу до спільних ресурсів у багатопоточних програмах?
20. корисний метод **Add()** у **sync.WaitGroup**, і як він впливає на внутрішній лічильник?
21. Як **time.Sleep** імітує затримку роботи в прикладах із горутинами?
22. Напишіть програму, яка запускає дві горутини: одна генерує випадкові числа, а інша обчислює їх квадрат. Використовуйте канал для передачі чисел між горутинами.
23. Напишіть функцію, яка приймає канал для надсилання даних (**chan<- int**) і відправляє у нього числа від 1 до 5. У головній функції отримайте ці числа через канал.
24. Реалізуйте програму, яка відправляє в канал числа від 1 до 10 і закриває його. Перевірте стан каналу і виведіть **"Channel closed!"** після його закриття.
25. Реалізуйте програму, в якій одна горутина обчислює факторіал числа, а інша чекає сигналу про завершення роботи через канал типу **chan struct{}**.
26. Напишіть програму, в якій дві горутини намагаються одночасно оновити спільний ресурс, але доступ до нього дозволяється тільки по черзі за допомогою м’ютекса.
27. Реалізуйте програму, в якій три горутини виконують свою роботу одночасно. Використовуйте **sync.WaitGroup**, щоб дочекатися завершення їх виконання.
28. Напишіть програму, яка створює N горутин (де N задається користувачем), що виконують прості обчислення. Забезпечте синхронізацію завершення їх роботи за допомогою **WaitGroup**.
29. Реалізуйте паралельне обчислення суми елементів рядків матриці за допомогою горутин. Використовуйте канал для передачі результатів і **WaitGroup** для синхронізації.
30. Напишіть програму, яка за допомогою кількох горутин фільтрує масив чисел за заданою умовою (наприклад, парні числа). Використовуйте м’ютекс для збереження результату в спільний масив.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Що таке горутина в Go?
a) Вид змінної
b) Легковаговий потік для паралельного виконання
c) Тип даних
d) Інструмент для синхронізації
Правильна відповідь: b).

Що відбувається, якщо читати дані з закритого каналу в Go?
a) Виникає помилка компіляції
b) Програма завершується з панікою
c) Повертається нульове значення типу і маркер закриття
d) Канал автоматично відкривається
Правильна відповідь: c).

Закриті запитання з кількома правильними відповідями

Що з наведеного є особливістю горутин? (виберіть дві відповіді)
a) Виконуються послідовно
b) Паралельно виконуються на багатоядерних системах
c) Виконуються незалежно одна від одної
d) Завжди завершуються одночасно
Правильна відповідь: b), c).

Які переваги використання м'ютексів у Go?
a) Забезпечення монопольного доступу до ресурсу
b) Паралельне виконання коду без блокування
c) Запобігання конкурентному доступу до змінних
d) Автоматичне завершення програми при помилці
Правильна відповідь: a), c).

Завдання на відповідність

Співставте тип каналу з його описом:
a) chan int → 1. Канал для двонаправленого обміну даними
b) chan<- int → 2. Канал тільки для надсилання даних
c) <-chan int → 3. Канал тільки для отримання даних

a) → 1.
b) → 2.
c) → 3.

Співвіднесіть методи sync.WaitGroup з їх функціоналом:
a) Add → 1. Додає горутини до групи
b) Done → 2. Сигналізує про завершення роботи
c) Wait → 3. Очікує завершення всіх горутин
Правильна відповідь:
a) → 1.
b) → 2.
c) → 3.

Завдання з кодом

Що виведе цей код?
```
package main
import "fmt"

func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    close(ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```
a) 1, 2, 0
b) 1, 2, runtime error
c) 1, 2, nil
d) Помилка компіляції
Правильна відповідь: a).

Який результат цього коду?
```
ch := make(chan int)
go func() { ch <- 10 }()
fmt.Println(<-ch)
```
Правильна відповідь: 10

Завдання з вибором істинності

Усі горутини виконуються послідовно.

a) Так

b) Ні

Правильна відповідь: b).

Завдання на доповнення коду

Для створення каналу типу string заповніть пропуск у виразі:

```
_____________:= make(chan string)
```

Правильна відповідь: …ch…

Заповніть пропуски в коді для синхронізації горутин:

```
var wg sync. _____________ ()
wg. _____________ (2)
go func() { defer wg. _____________ () }()
go func() { defer wg. _____________ () }()
wg. _____________ ()
```

Правильна відповідь: …sync.WaitGroup…, …Add…, …Done…, …Done…, …Wait…

Відкриті запитання

Як можна оптимізувати наступний код, щоб він працював без блокування:

```
package main
func main() {
    ch := make(chan int)
    ch <- 42
    fmt.Println(<-ch)
}
```

Правильна відповідь: Необхідно додати горутину для запису в канал:

```
    go func() { ch <- 42 }()
```

Напишіть приклад коду, в якому горутина передає дані через канал.

Правильна відповідь:

```
package main
import "fmt"
func main() {
```

```
    ch := make(chan int)
    go func() {
        ch <- 42
        close(ch)
    }()
    fmt.Println(<-ch)
}
```

8. ПОТОКИ ТА ФАЙЛИ

8.1. Операції введення-виведення. Reader та Writer

Мова Go має свою модель роботи з потоками вводу-виводу, яка дозволяє отримувати дані з різних джерел: файлів, мережевих інтерфейсів, об'єктів у пам'яті тощо.

Потік даних у мові Go представлений байтовим зрізом (**[]byte**), з якого можна зчитувати байти або в який можна записувати дані. Ключовими типами роботи з потоками є інтерфейси **Reader** і **Writer** з пакета **io**.

8.1.1. Інтерфейс io.Reader

Інтерфейс **io.Reader** призначений для зчитування даних. Він має наступне визначення:

```
1   type Reader interface {
2       Read(p []byte) (n int, err error)
3   }
```

Метод **Read** повертає загальну кількість переданих байт із зрізу байт та інформацію про помилку, якщо вона виникне. Якщо більше немає даних, то метод повинен повертати помилку типу **io.EOF**.

Розглянемо найпростіший приклад. Наприклад, необхідно зчитувати номери телефонів, які можуть мати різні формати:

```
1   package main
2   import (
3   "fmt"
4   "io"
5   )
6
7   type phoneReader string
8
9   func (ph phoneReader) Read(p []byte) (int, error){
10      count := 0
11      for i := 0; i < len(ph); i++{
12          if(ph[i] >= '0' && ph[i] <= '9'){
13              p[count] = ph[i]
14              count++
15          }
16      }
```

```
17      return count, io.EOF
18  }
19
20  func main() {
21      phone1 := phoneReader("+1(234)567 9010")
22      phone2 := phoneReader("+2-345-678-12-35")
23
24      buffer := make([]byte, len(phone1))
25      phone1.Read(buffer)
26      fmt.Println(string(buffer))      // 12345679010
27
28      buffer = make([]byte, len(phone2))
29      phone2.Read(buffer)
30      fmt.Println(string(buffer))      // 23456781235
31  }
```

Для зчитування номерів телефонів визначено тип **phoneReader**, який насправді представляє тип **string**. Проте **phoneReader** реалізує інтерфейс **Reader**, тобто визначає його метод **Read**. У методі **Read** зчитуємо дані з рядка, який представляє об'єкт **phoneReader** і якщо символи рядка представляють числові дані, передаємо їх у зріз байтів. На виході повертаємо кількість перелічених даних та маркер закінчення читання **io.EOF**. В результаті, при зчитуванні з рядка метод **Read** поверне номер телефону, який складається тільки з цифр.

При виклику метода **Read** створюється зріз байтів достатньої довжини, який передається у метод **Read**, а саме:

```
1   buffer := make([]byte, len(phone1))
2   phone1.Read(buffer)
```

Потім за допомогою ініціалізації **string** можна перетворити зріз байтів у рядок, наприклад:

```
1   fmt.Println(string(buffer)) // 12345679010
```

8.1.2. Інтерфейс io.Writer

Інтерфейс io.**Writer** призначений для запису в потік. Він визначає метод **Write()** наступним чином:

```
1   type Writer interface {
2       Write(p []byte) (n int, err error)
3   }
```

Метод **Write** призначений для копіювання даних зрізу байт **p** в певний ресурс: файл, мережевий інтерфейс і т.д. Метод повертає кількість записаних байтів та об'єкт помилки.

Розглянемо наступний приклад:

```
1   package main
2   import "fmt"
3
4   type phoneWriter struct{ }
5
6   func (p phoneWriter) Write(bs []byte) (int, error){
7       if len(bs) == 0 {
8            return 0, nil
9       }
10      for i := 0; i < len(bs); i++{
11           if(bs[i] >= '0' && bs[i] <= '9'){
12               fmt.Print(string(bs[i]))
13           }
14      }
15      fmt.Println()
16      return len(bs), nil
17  }
18
19  func main() {
20      bytes1 := []byte("+1(234)567 9010")
21      bytes2 := []byte("+2-345-678-12-35")
22
23      writer := phoneWriter{}
24      writer.Write(bytes1)
25      writer.Write(bytes2)
26  }
```

В наведеному прикладі структура **phoneWriter** реалізує інтерфейс **Writer**. У методі **Write** вона приймає зріз байтів. Передбачається, що зріз байтів зберігає номер телефону. Ця інформація належним чином обробляється, тобто із неї виділяються цифри, які виводяться на консоль. Тобто тип **phoneWriter** здійснює запис потоку байт на консоль.

Як результат метод повертає довжину зрізу та значення **nil**.

Для імітації потоку байт визначаються два зрізи байт на основі рядків, які передаються метод **Write**.

На основі вище розглянутих інтерфейсів **Writer** і **Reader** заснована вся система вводу-виводу в мові **Go**, і згодом буде детальніше розглянуто їх застосування під час роботи з файлами та потоками у мережі.

8.2. Створення та відкриття файлів

Для роботи з файлами можна використовувати функціональність **OS**. Всі файли Go представлені типом **os.File**. Цей тип реалізує низку інтерфейсів, наприклад, **io.Reader** та **io.Writer**, які дозволяють читати вміст файлу та зберігати дані у файл.

За допомогою функції **os.Create()** можна створити файл за певним шляхом. Шлях до файлу передається як параметр. Якщо подібний файл вже існує, він перезаписується, наприклад:

```
1   file, err := os.Create("hello.txt")
```

Ця функція повертає об'єкт **os.File** для роботи з файлом та інформацію про помилку, яка може виникнути під час створення файлу.

Раніше створений файл можна відкрити за допомогою функції **os.Open()**, наприклад:

```
1   file, err := os.Open("hello.txt")
```

Ця функція також повертає об'єкт **os.File** для роботи з файлом та інформацію про помилку, яка може виникнути під час відкриття файлу.

Також є функція **os.OpenFile()**, яка відкриває файл, а якщо файлу немає, то створює його. Вона приймає три параметри, а саме:

- шлях до файлу;
- режим відкриття файлу (для читання, для запису тощо);
- дозволи на доступ до файлу.

Наприклад:

```
1   // Відкриття файлу для читання
2   f1, err := os.OpenFile("sometext.txt", os.O_RDONLY,
    0666)
3   // Відкриття файлу для запису
4   f2, err := os.OpenFile("common.txt", os.O_WRONLY,
    0666)
```

Після роботи з файлом його слід закрити за допомогою метода **Close().**

```
1   package main
2   import (
3   "fmt"
4   "os"
```

```
)

func main() {
    file, err := os.Create("hello.txt")     // Створюємо файл
    if err != nil{           // Якщо виникла помилка
        fmt.Println("Unable to create file:", err)
        os.Exit(1)            // Виходимо із програми
    }
    defer file.Close()           // Закриваємо файл
    fmt.Println(file.Name())     // hello.txt
}
```

За допомогою функції **os.Exit()** можна вийти із програми. А метод **Name()**, визначений для типу **os.File**, дозволяє отримати ім'я файлу.

8.3. Читання та запис файлів

8.3.1. Запис у файл

Для запису текстової інформації у файл можна застосовувати метод **WriteString()** об'єкта **os.File**, який заносить у файл рядок, наприклад:

```
package main
import (
    "fmt"
    "os"
)

func main() {
    text := "Hello Gold!"
    file, err := os.Create("hello.txt")

    if err != nil{
        fmt.Println("Unable to create file:", err)
        os.Exit(1)
    }
    defer file.Close()
    file.WriteString(text)

    fmt.Println("Done.")
```

```
19  }
```

В наведеному прикладі створюється файл **hello.txt**, в який записується рядок «**Hello Gold!**».

Для запису нетекстової бінарної інформації як набору байт застосовується метод **Write()** з реалізації інтерфейсу **io.Writer**, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5   )
6
7   func main() {
8       data := []byte("Hello Bold!")
9       file, err := os.Create("hello.bin")
10      if err != nil{
11          fmt.Println("Unable to create file:", err)
12          os.Exit(1)
13      }
14      defer file.Close()
15      file.Write(data)
16
17      fmt.Println("Done.")
18  }
```

8.3.2. Читання з файлу

Оскільки тип **io.File** реалізує інтерфейс **io.Reader**, для читання з файлу можна використовувати метод **Read()**. Цей метод дозволяє отримати вміст файлу у вигляді набору байт, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5       "io"
6   )
7
8   func main() {
9       file, err := os.Open("hello.txt")
```

```
10      if err != nil{
11          fmt.Println(err)
12          os.Exit(1)
13      }
14      defer file.Close()
15
16      data := make([]byte, 64)
17
18      for{
19          n, err := file.Read(data)
20          if err == io.EOF{   // Якщо кінець файлу
21              defer           // виходимо із циклу
22          }
23          fmt.Print(string(data[:n]))
24      }
25  }
```

Для зчитування даних визначається зріз із 64 байтів. У нескінченному циклі вміст файлу зчитується в зріз, а коли буде досягнуто кінець файлу, тобто згенерується помилка **io.EOF**, то відбудеться вихід із циклу. Ну і оскільки дані представляють зріз байтів, хоча файл **hello.txt** зберігає текстову інформацію, то для виведення тексту на консоль зріз байтів перетворюється у рядок наступною конструкцією **string(data[:n])**.

8.4. Стандартні потоки введення-виведення та io.Copy

Пакет **os** визначає три змінні: **os.Stdin**, **os.Stdout** та **os.Stderr**, які представляють стандартні потоки введення, виведення та виведення помилок відповідно. Так, стандартний потік виведення **os.Stdout** забезпечує виведення інформації на консоль.

Функцію **io.Copy()** зручно використовувати для копіювання даних з одного потоку до іншого, наприклад:

```
1   n, err = io.Copy(io.Writer, io.Reader)
```

Ця функція спрощує копіювання даних з об'єкта **io.Reader** в об'єкт **io.Writer**. Як результат, функція повертає кількість скопійованих файлів та інформацію про помилку.

Під час виведення з файлу текстової інформації на консоль набагато простіше передати дані з файлового потоку в **os.Stdout**, ніж виводити дані окремими порціями, наприклад:

```
1  package main
2  import (
3      "fmt"
4      "os"
5      "io"
6  )
7
8  func main() {
9      file, err := os.Open("hello.txt")
10     if err != nil{
11         fmt.Println(err)
12         os.Exit(1)
13     }
14     defer file.Close()
15
16     io.Copy(os.Stdout, file)
17 }
```

Замість **io.Reader** можна використовувати власні створені об'єкти, які реалізують цей інтерфейс, наприклад:

```
1  package main
2  import (
3      "fmt"
4      "io"
5      "os"
6  )
7
8  type phoneReader string
9
10 func (p phoneReader) Read(bs []byte) (int, error){
11     count := 0
12     for i := 0; i < len(p); i++{
13         if(p[i] >= '0' && p[i] <= '9'){
14             bs[count] = p[i]
15             count++
16         }
17     }
18     return count, io.EOF
19 }
```

```
20
21  func main() {
22      phone1 := phoneReader("+1(234)567 90-10")
23      io.Copy(os.Stdout, phone1)
24      fmt.Println()
25  }
```

В наведеному прикладі замість інтерфейсу **io.Reader** передається об'єкт **phoneReader**, який зчитує цифрові символи з номера телефону.

8.5. Форматоване виведення

Пакет **fmt**, документація на який доступна за посиланням **https://golang.org/pkg/fmt/** [7], надає низку можливостей читання та запису файлів. Цей пакет містить функції для запису даних у довільний об'єкт, який реалізує інтерфейс **io.Writer: fmt.Fprint()**, **fmt.Fprintln() та fmt.Fprintf()**.

8.5.1. Функції Fprint та Fprintln

Функції **Fprint** та **Fprintln** мають приблизно однакове визначення, а саме:

```
1   func Fprint(w io.Writer, a ...interface{}) (n int,
    err error)
2   func Fprintln(w io.Writer, a ...interface{}) (n int,
    err error)
```

Першим параметром передається об'єкт, який є реалізацією інтерфейсу **io.Writer**. А другим параметром є набір об'єктів, які записуються в потік. Обидві функції повертають кількість записаних байтів та інформацію про помилку. Відмінністю функції **Fprintln** є те, що вона додає при виведенні на консоль команду переходу на новий рядок, тобто фактично записує рядок, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5   )
6
7   func main() {
8       file, err := os.Create("confeve.txt")
9       if err != nil{
10          fmt.Println(err)
```

```
11          os.Exit(1)
12      }
13      defer file.Close()
14      fmt.Fprint(file, "Сьогодні")
15      fmt.Fprintln(file, " гарна погода")
16  }
```

В наведеному прикладі обидві функції записують деякий текст у файл **confeve.txt**, який буде створений у тій же теці, де розташований тест програми.

8.5.2. Форматування та Fprintf

Функція **Fprintf** спрощує запис складних структурою даних і має наступний формальний опис:

```
1   func Fprintf(w io.Writer, format string, a
    ...interface{}) (n int, err error)
```

Першим параметром також є об'єкт **io.Writer**. Другим параметром є рядок форматування, який вказує, як дані форматуватимуться під час запису. І третій параметр – це набір значень, які передаються у рядок форматування та записуються в потік виведення.

Рядок форматування містить набір специфікаторів. Кожен специфікатор представляє набір символів, які інтерпретуються певним чином і передаються знаком відсотка (**%**). Кожен специфікатор представляє певний тип даних, а саме:

- **%t** : для виведення значень типу **boolean** (**true** або **false**);
- **%b** : для виведення цілих чисел у двійковій системі;
- **%c** : для виведення символів, представлених числовим кодом;
- **%d** : для виведення цілих чисел у десятковій системі;
- **%o** : для виведення цілих чисел у вісімковій системі;
- **%q** : для виведення символів в одинарних лапках;
- **%x** : для виведення цілих чисел у шістнадцятковій системі, літерні символи числа мають нижній регістр **a-f**;
- **%X** : для виведення цілих чисел у шістнадцятковій системі, літерні символи числа мають верхній регістр **A-F**;
- **%U** : для виведення символів у форматі кодів **Unicode**, наприклад, **U+1234**;
- **%e** : для виведення чисел з плаваючою точкою в експоненційному поданні, наприклад: -1.234456e+78;

- **%E** : для виведення чисел з плаваючою точкою в експоненційному поданні, наприклад: -1.234456E+78;
- **%f** : для виведення чисел з плаваючою точкою, наприклад: 123.456;
- **%F** : те саме, що й **%f**;
- **%g** : для довгих чисел із плаваючою точкою використовується **%e**, для інших - **%f**;
- **%G** : для довгих чисел з плаваючою точкою використовується **%E**, для інших - **%F**;
- **%s** : для виведення рядка;
- **%p** : для виведення значення покажчика, тобто адреси у шістнадцятковому поданні.

Також можна застосовувати універсальний специфікатор **%v**, який для типу **boolean** аналогічний **%t**, для цілих типів аналогічний **%d**, для чисел з плаваючою точкою аналогійний **%g**, а для рядків аналогічний **%s**.

До специфікаторів можна додавати різні прапорці, які впливають форматування значень. Наприклад, число перед специфікатором вказує, яку мінімальну довжину в символах займатиме значення, що виводиться. Наприклад, **%9f** сформує число з плаваючою точкою, яке займатиме щонайменше 9 позицій. Якщо ширина більше, ніж потрібно значення, заповнюється пробілами.

Для чисел з плаваючою точкою можна вказати точність чи кількість символів у дробовій частині. Для цього кількість символів вказується після крапки. Наприклад **%.2f** сформує число з плаваючою точкою і двома цифрами після коми.

Наведемо декілька прикладів форматування чисел з плаваючою точкою:

- **%f** : точність та ширина за замовчуванням;
- **%9f** : ширина 9 символів, а точність за замовчуванням;
- **%.2f** : ширина за замовчуванням, а точність 2 символи;
- **%9.2f** : ширина 9 символів, а та точність 2 символи;
- **%9.f** : ширина 9 символів, а точність 0 символів.

Також серед прапорців слід зазначити символ дефісу (-), який доповнює значення пробілами не праворуч, як за замовчуванням, а ліворуч.

Застосуємо функцію **Fprintf** для виведення інформації з файлу, а саме:

```
1   package main
2   import (
3       "fmt"
4       "os"
```

```
5   )
6   type person struct {
7       name string
8       age int32
9       weight float64
10  }
11  func main() {
12      tom := person {
13          name:"Tom",
14          age: 24,
15          weight: 68.5,
16      }
17      file, err := os.Create("person.dat")
18      if err != nil{
19          fmt.Println(err)
20          os.Exit(1)
21      }
22      defer file.Close()
23      fmt.Fprintf( file,
24                  "%-10s %-10d %-10.3f\n",
25                  tom.name, tom.age, tom.weight)
26  }
```

Функція **Fprintf()** у першому параметрі також приймає файл, а у другому параметрі приймає рядок форматування, який визначає, як дані будуть форматуватися. Після рядка форматування перераховуються значення, які вставлятимуться замість специфікаторів. При чому значення передаються до специфікаторів за позицією. Наприклад, перше значення передається замість першого специфікатора, друге значення замість другого специфікатора і так далі. При цьому значення повинні відповідати специфікаторам за типом. Тобто, на місце специфікатора **%s** повинен передаватися рядок, f на місце **%d** повинно передаватися ціле число і т.д.

Таким чином, у прикладі вище було створено в одній з файлом програми теці новий файл **person.dat**, до якого будуть записані дані об'єкта **person**.

8.6. Виведення на консоль

Стандартним потоком виведення у мові Go є об'єкт **os.Stdout**, який фактично є консоллю. Наприклад, виведемо в цей потік дані наступним чином:

```
package main
import (
    "fmt"
    "os"
)

func main() {
    fmt.Fprintln(os.Stdout, "hello cold")
}
```

В цьому прикладі використовується розглянута у минулих розділах функція **Fprintln()**, яка виводить у потік виведення набір значень. Тобто фактично в цьому випадку відбувається запис або виведення на консоль.

Та оскільки запис у стандартний потік **os.Stdout** є досить поширеним завданням, то замість функцій **Fprint/Fprintln/Fprintf** застосовуються їх двійники, а саме: **Println()**, **Print()** та **Printf()**, відповідно, які за замовчуванням виводять дані в **os.Stdout**, наприклад:

```
package main
import "fmt"

type person struct {
   name string
   age int32
   weight float64
}
func main() {
    tom := person {
        name:"Tom",
        age: 24,
        weight: 68.5,
    }
    fmt.Printf("%-10s %-10d %-10.3f\n",
                tom.name, tom.age, tom.weight)
    fmt.Print("Hello ")
    fmt.Println("cold!")
}
```

8.7. Форматоване введення

Для організації читання об'єктів мова Go моє пакет **fmt**, який реалізує інтерфейс **io.Reader**. Основними функціями для забезпечення зчитування даних є функції **Fscan()**, **Fscanln()** та **Fscanf()**.

Розглянемо їх детальніше.

8.7.1 Функції Fscan та Fscanln

Через параметри функцій **Fscan()** і **Fscanln()** можна отримати значення, що вводяться, наприклад:

```
1   func Fscan(r io.Reader, a ...interface{}) (n int, err
    error)
2   func Fscanln(r io.Reader, a ...interface{}) (n int,
    err error)
```

У якості першого параметру передається об'єкт **io.Reader**, з якого треба зчитувати дані, а другий параметр представляє об'єкти, в які дані зчитуються. Як результат обидві функції повертають кількість зчитаних байт та інформацію про помилку, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5   )
6
7   type person struct {
8      name string
9      age int32
10     weight float64
11  }
12  func main() {
13      filename := "hello2.txt"
14      writeData(filename)
15      readData(filename)
16  }
17
18  func writeData(filename string){
19      // Початкові данні
20      var name string = "Tom"
```

```
21      var age int = 24
22
23      file, err := os.Create(filename)
24      if err != nil {
25          fmt.Println(err)
26          os.Exit(1)
27      }
28      defer file.Close()
29
30      fmt.Fprintln(file, name)
31      fmt.Fprintln(file, age)
32  }
33  func readData(filename string){
34
35      var name string
36      var age int
37
38      file, err := os.Open(filename)
39      if err != nil{
40          fmt.Println(err)
41          os.Exit(1)
42      }
43      defer file.Close()
44
45      fmt.Fscanln(file, &name)
46      fmt.Fscanln(file, &age)
47      fmt.Println(name, age)
48  }
```

У наведеному прикладі спочатку записуються дві змінні у файл за допомогою **fmt.Fprintln**, а потім зчитуються записані значення за допомогою **fmt.Fscanln**.

8.7.2. Функція Fscanf

Функція **fmt.Fscanf()** зчитує дані із застосуванням форматування і формально описується наступним чином:

```
1   func Fscanf(r io.Reader, format string, a
    ...interface{}) (n int, err error)
```

Першим параметром функції є об'єкт **io.Reader**. Другим параметром є рядок форматування, що містить специфікатори та визначає послідовність зчитування даних. Третім параметром є набір об'єктів, у які потрібно зчитати дані, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5   )
6
7   type person struct {
8      name string
9      age int32
10     weight float64
11  }
12  func main() {
13      filename := "person.dat"
14      writeData(filename)
15      readData(filename)
16  }
17
18  func writeData(filename string){
19      // Початкові дані
20      tom := person { name:"Tom", age: 24, weight: 68.5
    }
21
22      file, err := os.Create(filename)
23      if err != nil {
24          fmt.Println(err)
25          os.Exit(1)
26      }
27      defer file.Close()
28
29      // Зберігаємо дані у файл
30      fmt.Fprintf(file, "%s %d %.2f\n", tom.name,
    tom.age, tom.weight)
31  }
32  func readData(filename string){
```

```
33
34      // Змінні для зчитування даних
35      var name string
36      var age int
37      var weight float64
38
39      file, err := os.Open(filename)
40      if err != nil{
41          fmt.Println(err)
42          os.Exit(1)
43      }
44      defer file.Close()
45
46      // Зчитування даних із файлу
47      _, err = fmt.Fscanf(file, "%s %d %f\n", &name,
    &age, &weight)
48      if err != nil{
49          fmt.Println(err)
50          os.Exit(1)
51      }
52      fmt.Printf("%-8s %-8d %-8.2f\n", name, age,
    weight)
53  }
```

В наведеному прикладі спочатку дані структури **person** записуються у файл, а потім зчитуються з нього у три змінні. При записі даних їх структура відома. Тому можна взяти рядок форматування з тією ж послідовністю специфікаторів та виконати зворотну дію, а саме зчитати дані. При зчитуванні об'єктів у функцію передаються їх адреси, а саме:

```
1   fmt.Fscanf(file, "%s %d %f\n", &name, &age, &weight)
```

При визначенні рядка форматування та передачі об'єктів для зчитування діють ті самі правила, що й під час запису за допомогою **fmt.Fprintf**. Так, перший специфікатор пов'язаний з першим об'єктом, другий специфікатор пов’язаний з другим об'єктом і т.і. Також специфікатори повинні відповідати об'єктам за типами.

Результат виконання цієї програми наведено на рисунку 8.1.

```
Tom 24 68.50
```

Рисунок 8.1. Результат роботи програми

При цьому об'єкти, в які здійснюється зчитування, необов'язково мають представляти змінні примітивних типів. Наприклад, це може бути і структура:

```
1   func readData(filename string){
2
3       // Змінна для зчитування даних
4       tom := person{}
5
6       file, err := os.Open(filename)
7       if err != nil{
8           fmt.Println(err)
9           os.Exit(1)
10      }
11      defer file.Close()
12
13      // Зчитування даних із файлу
14      _, err = fmt.Fscanf(file, "%s %d %f\n", &tom.name,
    &tom.age, &tom.weight)
15
16      if err != nil{
17          fmt.Println(err)
18          os.Exit(1)
19      }
20      fmt.Printf("%-8s %-8d %-8.2f\n", tom.name,
    tom.age, tom.weight)
21  }
```

Розглянемо складніший приклад, коли файл містить набір даних структур, а саме:

```
1   package main
2   import (
3       "fmt"
4       "os"
5       "io"
6   )
7
8   type person struct {
9      name string
10     age int32
11     weight float64
```

```
12 }
13 func main() {
14     filename := "people.dat"
15     writeData(filename)
16     readData(filename)
17 }
18
19 func writeData(filename string){
20     // Початкові дані
21     var people = []person{
22         { "Tom", 24, 68.5 },
23         { "Bob", 25, 64.2 },
24         { "Sam", 27, 73.6 },
25     }
26
27     file, err := os.Create(filename)
28     if err != nil {
29         fmt.Println(err)
30         os.Exit(1)
31     }
32     defer file.Close()
33
34     for _, p := range people{
35         fmt.Fprintf(file, "%s %d %.2f\n", p.name, p.age,
   p.weight)
36     }
37 }
38 func readData(filename string){
39
40     var name string
41     var age int
42     var weight float64
43
44     file, err := os.Open(filename)
45     if err != nil{
46         fmt.Println(err)
47         os.Exit(1)
48     }
```

```
49      defer file.Close()
50
51      for{
52          _, err = fmt.Fscanf(file, "%s %d %f\n", &name,
    &age, &weight)
53          if err != nil{
54              if err == io.EOF{
55                  break
56              } else{
57                  fmt.Println(err)
58                  os.Exit(1)
59              }
60          }
61          fmt.Printf("%-8s %-8d %-8.2f\n", name, age,
    weight)
62      }
63  }
```

В цьому прикладі спочатку функція **writeData** записує до файлу набір об'єктів **person**. А потім функція **readData** зчитує дані з файлу в нескінченному циклі. Як тільки файл закінчиться, то функція **Fscanf** поверне помилку **io.EOF**.

8.8. Читання з консолі

У мові Go є об'єкт **os.Stdin**, який реалізує інтерфейс **io.Reader** і дозволяє зчитувати дані з консолі. Наприклад, можна використовувати функцію **fmt.Fscan()** для зчитування з консолі за допомогою **os.Stdin**, а саме:

```
1   package main
2   import (
3       "fmt"
4       "os"
5   )
6
7   func main() {
8       var name string
9       var age int
10      fmt.Print("Введіть ім'я: ")
11      fmt.Fscan(os.Stdin, &name)
12
```

```
13      fmt.Print("Введіть вік: ")
14      fmt.Fscan(os.Stdin, &age)
15
16      fmt.Println(name, age)
17  }
```

При запуску програми стає можливим введення даних з консолі, і вони передаватимуться до змінних **name** та **age**, як показано на рисунку 8.2.

```
Введіть ім'я: Tom
Введіть вік: 24
Tom 24
```

Рисунок 8.2. Результат роботи програми

Для отримання введених з консолі даних можна використовувати інші вбудовані функції **fmt.Scan()**, **fmt.Scanln()** та **fmt.Scanf()**, які аналогічні відповідно функціям **fmt.Fscan()**, **fmt.Scanln()** та **fmt.Scanf()**, наприклад:

```
1   func Scan(a ...interface{}) (n int, err error)
2   func Scanf(format string, a ...interface{}) (n int,
    err error)
3   func Scanln(a ...interface{}) (n int, err error)
```

За замовчуванням всі ці функції вже зчитують дані з потоку **os.Stdin**, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5   )
6
7   func main() {
8       var name string
9       var age int
10      fmt.Print("Введіть ім'я: ")
11      fmt.Scan(&name)
12      fmt.Print("Введіть вік: ")
13      fmt.Scan(&age)
14
15      fmt.Println(name, age)
16  }
```

або так

```
1   package main
2   import (
3       "fmt"
4       "os"
5   )
6
7   func main() {
8       var name string
9       var age int
10      fmt.Print("Введіть ім'я та вік: ")
11      fmt.Scan(&name, &age)
12      fmt.Println(name, age)
13
14      // Альтернативний варіант
15      //fmt.Println("Введіть ім'я та вік:")
16      //fmt.Scanf("%s %d", &name, &age)
17      //fmt.Println(name, age)
18  }
```

У разі одночасного введення відразу кількох значень, то роздільником між ними є пропуск ( ). Хоча теоретично рядок може включати внутрішні пропуски, проте зазначені функції зчитують значення рядка та інших типів даних до пропуску, як видно з рисунку 8.3.

```
Введіть ім'я та вік: Tom 34
Tom 34
```

Рисунок 8.3. Результат роботи програми

8.9. Буферизоване введення-виведення

Більшість вбудованих операцій введення-виведення не використовують буфер. Це може мати негативний ефект для продуктивності програм. Для буферизації потоків читання та запису у мові Go визначено низку можливостей, які зосереджені в пакеті **bufio**.

8.9.1. Запис через буфер

Для запису даних через буфер у пакеті **bufio** визначено тип **Writer**. Щоб записати дані, можна скористатися одним із його методів, формальне визначення яких наступне:

```
1   func (b *Writer) Write(p []byte) (nn int, err error)
2   func (b *Writer) WriteByte(c byte) error
3   func (b *Writer) WriteRune(r rune) (size int, err
4   error)
    func (b *Writer) WriteString(s string) (int, error)
```

При цьому призначення цих методів наступне:

- **Write()** : записує зріз байтів;
- **WriteByte()** : записує один байт;
- **WriteRune()** : записує один об'єкт типу **rune**
- **WriteString()** : записує рядок.

При виконанні цих методів дані спочатку накопичуються в буфері. А для передачі їх до одержувача, необхідно викликати метод **Flush()**.

Для створення потоку виведення через буфер застосовується функція **bufio.NewWriter()**, формальне визначення якої наступне:

```
1   func NewWriter(w io.Writer) *Writer
```

Вона приймає об'єкт **io.Writer**, яким може бути будь-який об'єкт, у який йде запис, наприклад: **os.Stdout**, файл і т.д. У якості результату відбудеться повернення об'єкту **bufio.Writer**, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5       "bufio"
6   )
7   func main() {
8      rows := []string{
9            "Hello Go!",
10           "Welcome to Golang",
11     }
12
13     file, err := os.Create("some.dat")
14     writer := bufio.NewWriter(file)
15     if err != nil {
```

```
16          fmt.Println(err)
17          os.Exit(1)
18      }
19      defer file.Close()
20
21      for _, row := range rows {
22          writer.WriteString(row)     // Запис рядка
23          writer.WriteString("\n")    // Перехід на
    новий рядок
24      }
25      writer.Flush()          // Скидаємо дані з буфера у
    файл
26  }
```

У наведеному прикладі файл через буферизований потік виведення записуються у два рядки.

8.9.2. Читання через буфер

Для читання даних із джерела даних через буфер у пакеті **bufio** визначено тип **Reader**. Для читання даних можна скористатися одним із його методів, формальне визначення яких наступне:

```
1   func (b *Reader) Read(p []byte) (n int, err error)
2   func (b *Reader) ReadByte() (byte, error)
3   func (b *Reader) ReadBytes(delim byte) ([]byte, error)
4   func (b *Reader) ReadLine() (line []byte, isPrefix
    bool, err error)
5   func (b *Reader) ReadRune() (r rune, size int, err
    error)
6   func (b *Reader) ReadSlice(delim byte) (line []byte,
    err error)
7   func (b *Reader) ReadString(delim byte) (string,
    error)
```

При цьому призначення цих методів наступне:

- **Read(p []byte)** : зчитує зріз байтів та повертає кількість прочитаних байтів;
- **ReadByte ()**: зчитує один байт;
- **ReadBytes(delim byte)** : зчитує зріз байтів із потоку, доки не зустрінеться байт delim;
- **ReadLine()** : зчитує рядок у вигляді зрізу байт;

- **ReadRune()** : зчитує один об'єкт типу rune;
- **ReadSlice(delim byte)** : зчитує зріз байтів із потоку, доки не зустрінеться байт delim;
- **ReadString(delim byte)** : зчитує рядок, доки не зустрінеться байт delim.

Для створення потоку введення через буфер використовується функція **bufio.NewReader()**, формальний опис якої наступний:

```
1   func NewReader(rd io.Reader) *Reader
```

Ця функція приймає об'єкт **io.Reader**, яким може бути будь-який об'єкт, з якого виконується читання, а саме: **os.Stdin**, файл і т.д. У якості результату відбудеться повернення об'єкту **bufio.Reader**, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "os"
5       "bufio"
6       "io"
7   )
8   func main(){
9       file, err := os.Open("some.data")
10      if err != nil {
11          fmt.Println("Unable to open file:", err)
12          return
13      }
14      defer file.Close()
15
16      reader := bufio.NewReader(file)
17      for {
18          line, err := reader.ReadString('\n')
19          if err != nil {
20              if err == io.EOF {
21                  break
22              } else {
23                  fmt.Println(err)
24                  return
25              }
26          }
27          fmt.Print(line)
```

```
28      }
29  }
```

В наведеному прикладі відбувається зчитування з раніше записаного файлу. Для цього об'єкт файлу **os.File** передається у функцію **bufio.NewReader**, на основі якого створюється об'єкт **bufio.Reader**. Оскільки зчитування відбувається по рядкам, то кожен рядок зчитується з потоку доти, доки не буде виявлено символ перенесення рядка **\n**.

Резюме

У наведеному розділі розглядаються потоки введення-виведення в мові програмування Go, зокрема робота з інтерфейсами **io.Reader** та **io.Writer**. Пояснюється, як за допомогою метода **Read()** можна зчитувати дані з різних джерел, а методом **Write()** записувати їх у файли або інші ресурси. Наведено приклади реалізації власних типів, що імплементують ці інтерфейси для обробки телефонних номерів. Також описано роботу з файлами через пакет **os**, зокрема створення, відкриття, читання і запис даних. Розглядається використання функції **io.Copy()** для зручного копіювання даних між потоками, включаючи стандартні потоки введення-виведення. Матеріал містить практичні приклади коду для роботи з текстовими та бінарними файлами.

Розглянуто основи форматованого введення та виведення даних у мові програмування Go за допомогою пакета **fmt**. Описані функції **Fprint**, **Fprintln**, **Fprintf**, які дозволяють записувати дані у потік виведення з різними форматами, включаючи роботу з файлами. Розглянуто специфікатори форматування, такі як **%d**, **%f**, **%s** та їх використання для точного контролю виводу даних. Також матеріал охоплює введення даних із консолі за допомогою функцій **Scan**, **Scanln**, **Scanf**, які зчитують дані з потоку **os.Stdin**. Окремо пояснюється використання буферизованого введення-виведення через пакет **bufio**, що підвищує продуктивність роботи з потоками даних. Наведено численні приклади коду для запису та зчитування даних із файлів, а також роботи з консоллю.

Наступні розділи об’єднають всі отримані знання та навички з мережевого програмування, інтеграції вебзастосунків з базами даних та створення динамічних вебзастосунків.

Контрольні запитання і завдання

1. Який основний тип використовується в Go для роботи з потоками даних?

2. Який метод повинен реалізовувати інтерфейс **io.Reader**?
3. Що повертає метод **Read()** у разі досягнення кінця потоку даних?
4. Як у Go реалізувати власний тип, що відповідає інтерфейсу **io.Reader**?
5. Який метод визначений в інтерфейсі **io.Writer**?
6. Що повертає метод **Write()** після успішного запису даних?
7. Яка функція використовується для відкриття існуючого файлу?
8. Чим відрізняється функція **os.OpenFile()** від **os.Open()**?
9. Як закрити відкритий файл після роботи з ним?
10. Як у Go можна записати нетекстові (бінарні) дані у файл?
11. Яке основне призначення пакета **fmt** у мові Go?
12. Чим відрізняються функції **Fprint** і **Fprintln**?
13. Які параметри приймає функція **Fprintf**?
14. Який специфікатор використовується для виведення рядка в функції форматованого виведення?
15. Як записати число з плаваючою точкою з двома знаками після коми за допомогою **Fprintf**?
16. Які специфікатори використовуються для виведення цілих чисел у десятковій, двійковій та шістнадцятковій системах?
17. Як відрізняються функції **Fscan** та **Fscanln**?
18. Що повертають функції **Fscan** і **Fscanln** після зчитування даних?
19. Реалізуйте власний тип, що імплементує інтерфейс **io.Reader**, який зчитує тільки великі літери з рядка.
20. Напишіть програму, яка створює файл "**output.txt**", записує в нього рядок "**Hello, Go**!" і закриває файл.
21. Розробіть програму для зчитування текстового файлу частинами по 32 байти та виведення вмісту в консоль.
22. Створіть структуру, яка реалізує інтерфейс **io.Writer**, і записує тільки цифри зі зрізу байтів у консоль.
23. Використайте функцію **io.Copy()**, щоб скопіювати вміст одного файлу в інший.
24. Створіть структуру **product** з полями **name** (**string**), **price** (**float64**) та **quantity** (**int**). Запишіть інформацію про товар у файл у вигляді таблиці, використовуючи **fmt.Fprintf()**.
25. Напишіть програму, яка зчитує ім'я та вік з файлу **info.txt**, використовуючи функції **fmt.Fscanln()**, та виводить ці дані на екран.

26. Реалізуйте програму, яка зчитує з консолі ім'я та вік користувача, використовуючи **fmt.Scan()**, і виводить отримані дані у форматі "**Ім'я: ..., Вік: ...**".
27. Напишіть програму, яка записує у файл значення змінної типу **float64** з двома знаками після коми та вирівнюванням по правому краю шириною в 10 символів.
28. Реалізуйте програму, яка записує у файл кілька рядків тексту через буферизований потік виведення, використовуючи пакет **bufio**.
29. Напишіть програму, яка читає файл рядок за рядком за допомогою **bufio.Reader** та виводить його вміст на екран.
30. Реалізуйте програму, яка обробляє можливу помилку **io.EOF** при зчитуванні файлу та виводить повідомлення про закінчення файлу.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Який інтерфейс у Go використовується для зчитування даних?

a) io.Writer

b) io.Reader

c) io.Scanner

d) io.File

Правильна відповідь: b).

Що повертає метод Read() інтерфейсу io.Reader при досягненні кінця потоку?

a) nil

b) 0, io.EOF

c) -1

d) false

Правильна відповідь: b).

Яке значення повертає метод Write() у разі успішного запису даних?

a) Кількість записаних байтів та nil

b) true або false

c) -1

d) nil

Правильна відповідь: a).

Яка функція у Go використовується для форматованого запису у файл?

a) fmt.Println()

b) fmt.Fprintf()

c) fmt.Write()

d) fmt.Printf()

Правильна відповідь: b).

Яка функція додає перехід на новий рядок під час запису у файл?

a) fmt.Fprint()

b) fmt.Println()

c) fmt.Fprintln()

d) fmt.WriteString()

Правильна відповідь: c).

Який специфікатор використовується для виведення цілих чисел у десятковій системі?

a) %b

b) %c

c) %d

d) %x

Правильна відповідь: c).

Закриті запитання з кількома правильними відповідями

Які стандартні потоки визначені у пакеті os?

a) os.Stdin

b) os.Stdout

c) os.Stderr

d) os.ReadWrite

Правильна відповідь: a), b), c).

Які функції можуть використовуватись для запису даних у файл?

a) fmt.Scan()

b) fmt.Fprintf()

c) fmt.Fprint()

d) fmt.Fprintln()

Правильна відповідь: b), c), d).

Що потрібно зробити після використання буферизованого запису?

a) Викликати Flush()

b) Закрити файл

c) Викликати CloseBuffer()

d) Перезавантажити програму

Правильна відповідь: a),b).

Завдання на відповідність

Співвіднесіть специфікатори з їхнім призначенням:

a). %s → 1 Рядок

b) %d → 2. Ціле число

c) %f → 3. Число з плаваючою точкою

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Завдання з кодом

Який метод використовується для читання файлу частинами?

a) file.Read()

b) file.Scan()

c) file.Fetch()

d) file.Load()

Правильна відповідь: a).

Як у Go оголосити структуру з полями name (string) та age (int)?

Правильна відповідь:

```
type person struct {
    name string
    age int
}
```

Знайдіть помилку в коді:

```
file, err := os.Open("data.txt")
fmt.Fscan(file, name, age)
file.Close()
```

Правильна відповідь: потрібно передати адреси змінних: &name, &age

9. МЕРЕЖЕВЕ ПРОГРАМУВАННЯ

Однією з ключових можливостей мови Go є можливість роботи з мережевими сервісами, а саме: надсилання запитів до ресурсів у мережі та, навпаки, обробляти вхідні запити. Основний функціонал роботи з мережею представлений пакетом **net**, документацію на який можна знайти за посиланням **https://golang.org/pkg/net/** [8]. Цей пакет надає різні низькорівневі мережеві примітиви, через які йде взаємодія через мережу.

9.1. Надсилання запитів

Для надсилання запитів до ресурсів у мережі застосовується функція **net.Dial()**, формальний опис якої наступний:

```
1   func Dial(network, address string) (Conn, error)
```

Ця функція приймає два параметри, а саме: **network** - тип протоколу та **address** - адресу ресурсу.

Найпоширенішими протоколами є такі типи протоколів:

- **tcp**, **tcp4**, **tcp6** : протокол **TCP** (**tcp** за замовчуванням представляє **tcp4**, цифра в кінці вказує, який тип адрес буде використовуватися, а саме: **IPv4** або **IPv6**);
- **udp**, **udp4**, **udp6** : протокол **UDP** (**udp** за замовчуванням представляє **udp4** ;
- **IP**, **IP4**, **IP6**: протокол IP (**ip** за замовчуванням представляє **ip4**);
- **unix**, **unixgram**, **unixpacket** : сокети **Unix**.

Другий параметр визначає мережеву адресу ресурсу (для адрес у мережі Інтернет це доменне ім’я). Ним може бути числова мережна адреса, наприклад, "**127.0.0.1**". Він може додатково включати номер порту, наприклад "**127.0.0.1:80**". Ним також може бути адреса у форматі **IPv6**, наприклад, "**::1**" або **"[2516:b7f0:3421:b16::71]:80"**.

Функція **net.Dial()** повертає об'єкт, який реалізує інтерфейс **net.Conn**. Цей інтерфейс, у свою чергу, застосовує інтерфейси **io.Reader** та **io.Writer**, тобто може використовуватись як потік для читання та запису. Пакет **net** надає базові реалізації цього інтерфейсу у вигляді типів **IPConn**, **UDPConn**, **TCPConn**. Залежно від протоколу, що використовується, повертається і відповідний тип.

Таким чином, використовуючи цю функцію можна надсилати запити протоколів **TCP** і **UDP**, наприклад:

```
1   package main
2   import (
```

```
3       "fmt"
4       "os"
5       "net"
6       "io"
7   )
8   func main() {
9       httpRequest:="GET / HTTP/1.1\n" +
10          "Host: golang.org\n\n"
11      conn, err := net.Dial("tcp", "golang.org:80")
12      if err != nil {
13          fmt.Println(err)
14          return
15      }
16      defer conn.Close()
17
18      if _, err = conn.Write([]byte(httpRequest)); err !=
    nil {
19          fmt.Println(err)
20          return
21      }
22
23      io.Copy(os.Stdout, conn)
24      fmt.Println("Done")
25  }
```

У наведеному прикладі, результат роботи програми якого наведено на рисунку 9.1, фактично відбувається звернення до вебресурсу **golang.org**. Оскільки **net.Conn** реалізує інтерфейси і**o.Reader** та **io.Writer**, то у цей об'єкт можна записувати дані, тобто фактично посилати по мережі дані, а також можна зчитувати з нього дані, тобто отримувати дані з мережі. Наприклад, **conn.Write([]byte(httpRequest))** посилає дані, які тут представлені змінною **httpRequest**. Так як метод **Write** відправляє зріз байтів, то будь-які дані потрібно перетворити на зріз байтів.

Як і будь-який об'єкт **io.Reader**, **net.Conn** можна передати у функцію **io.Copy** і зчитати отримані по мережі дані, наприклад, на консоль **io.Copy(os.Stdout, conn)**.

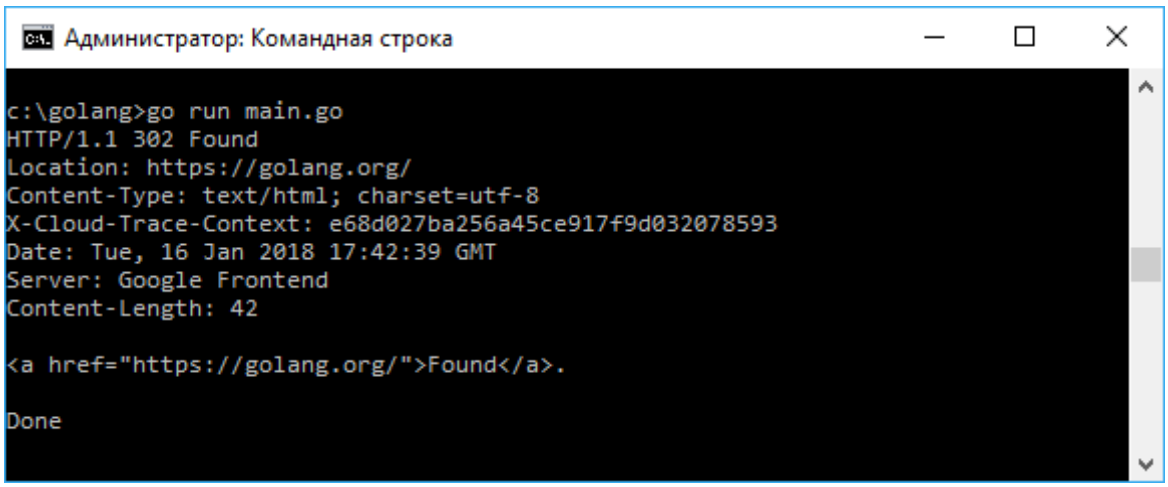

Рисунок 9.1. Результат роботи програми

Варто зазначити, що прикладі вище виконується запит до мережного ресурсу мережі Інтернет за протоколом **TCP**. Однак для цієї ж мети значно зручніше використовувати можливості пакета **net/http**, який призначений спеціально для протоколу **HTTP**, який працює поверх **TCP**.

9.2. Сервер. Обробка підключень

Для прослуховування та прийому вхідних запитів у пакеті **net** визначено функцію **net.Listen**, формальний опис якої наступний:

```
1   func Listen(network, laddr string) (net.Listener,
    error)
```

Функція приймає два параметри, а саме: **network** - протокол, за яким застосунок отримуватиме запити, і **laddr**, який позначає локальний адресу, за якою буде запускатися сервер. Протокол повинен мати одне із значень, с саме: "**tcp**", "**tcp4**", "**tcp6**", "**unix**", "**unixpacket**". Локальна адреса може містити і лише номер порту, наприклад ":**8080**". У цьому разі для мереж **TCP**, якщо хост у параметрі адреси порожній або буквально не визначено IP-адресу, то **Listen** буде прослуховувати всі доступні одноадресні та будь-які IP-адреси локальної системи.

У разі успішного виконання функція повертає об'єкт інтерфейсу **net.Listener**, який надає функціонал прийому вхідних підключень. Залежно від типу використовуваного протоколу об'єкт **Listener**, що повертається, може представляти тип **net.TCPListener** або **net.UnixListener**. Обидва цих типи реалізують інтерфейс **net.Listener**.

Основні методи, які надає **net.Listener** наступні: **Accept()**, який приймає вхідне підключення; **Close()**, який закриває підключення, наприклад:

```
1   package main
```

```
2   import (
3       "fmt"
4       "net"
5   )
6   func main() {
7       message := "Hello, I am a server"   //
    Повідомлення, що надсилається
8       listener, err := net.Listen("tcp", ":4545")
9
10      if err != nil {
11          fmt.Println(err)
12          return
13      }
14      defer listener.Close()
15      fmt.Println("Server is listening...")
16      for {
17          conn, err := listener.Accept()
18          if err != nil {
19              fmt.Println(err)
20              return
21          }
22          conn.Write([]byte(message))
23          conn.Close()
24      }
25  }
```

В наведеному прикладі спочатку у функції **net.Listen("tcp", ":4545")** встановлюється порт **4545** для прослуховування підключень за протоколом **TCP**. Після виклику цієї функції сервер запускається і стає готовим до приймання підключень. Потім у нескінченному циклі **for** отримуються вхідні підключення за допомогою виклику **listener.Accept()**. Цей метод повертає об'єкт **net.Conn**, який представляє підключеного клієнта. Потім стає можливою обробка цього підключення. Наприклад, за допомогою метода **Write** можна надіслати йому повідомлення. Оскільки цей метод приймає зріз байтів, то будь-які повідомлення необхідно транслювати в зріз байтів, я саме: **conn.Write([]byte(message))**.

Для тестування сервера визначимо ще одну програму-клієнт, а саме:

```
1   package main
2   import (
```

```
3       "fmt"
4       "os"
5       "net"
6       "io"
7   )
8   func main() {
9
10      conn, err := net.Dial("tcp", "127.0.0.1:4545")
11      if err != nil {
12          fmt.Println(err)
13          return
14      }
15      defer conn.Close()
16
17      io.Copy(os.Stdout, conn)
18      fmt.Println("\nDone")
19  }
```

Оскільки сервер буде запущений на локальному комп'ютері на порті **4545**, то і клієнт буде підключатися наступним чином:

```
1   net.Dial("tcp", "127.0.0.1:4545")
```

Після цього до сервера надсилатиметься запит, і за допомогою виклику **io.Copy(os.Stdout, conn)** виводиметься отримана відповідь на консоль.

Отже, спочатку запустимо сервер. На Windows може з'явитися вікно з пропозицією дозволити доступ, як наведено на рисунку 9.2.

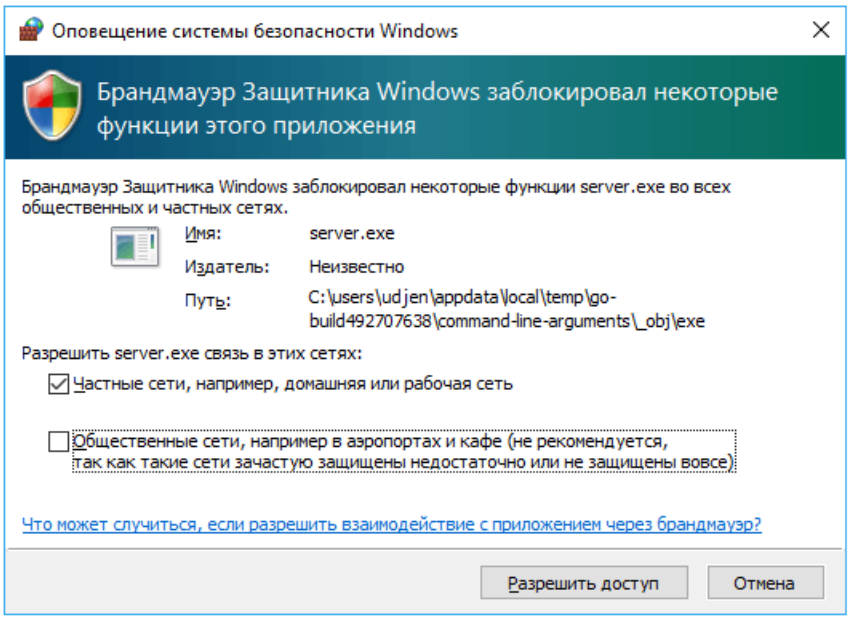

Рисунок 9.2. Вигляд вікна з пропозицією дозволити доступ

Після цього буде запущено сервер, як наведено на рисунку 9.3.

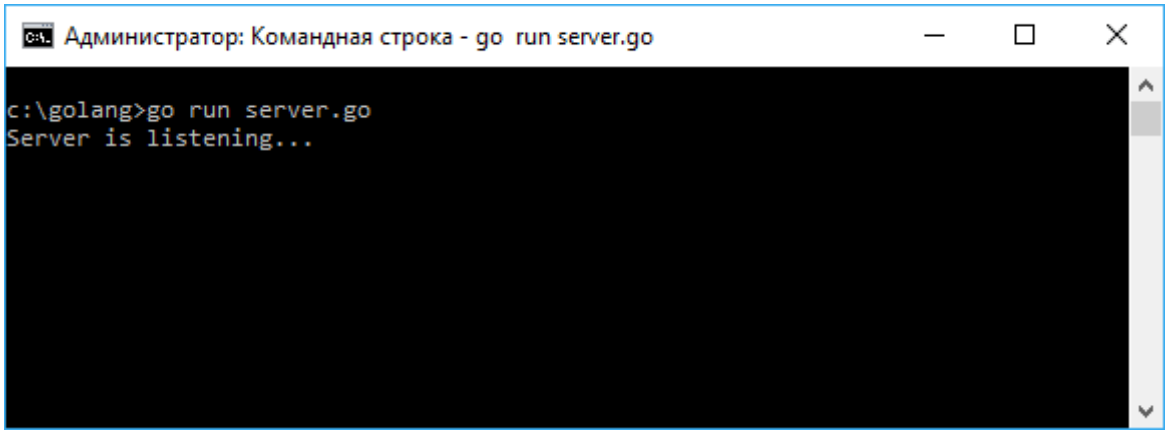

Рисунок 9.3. Вигляд вікна запуску сервера

Потім необхідно запустити клієнт, як наведено на рисунку 9.4.

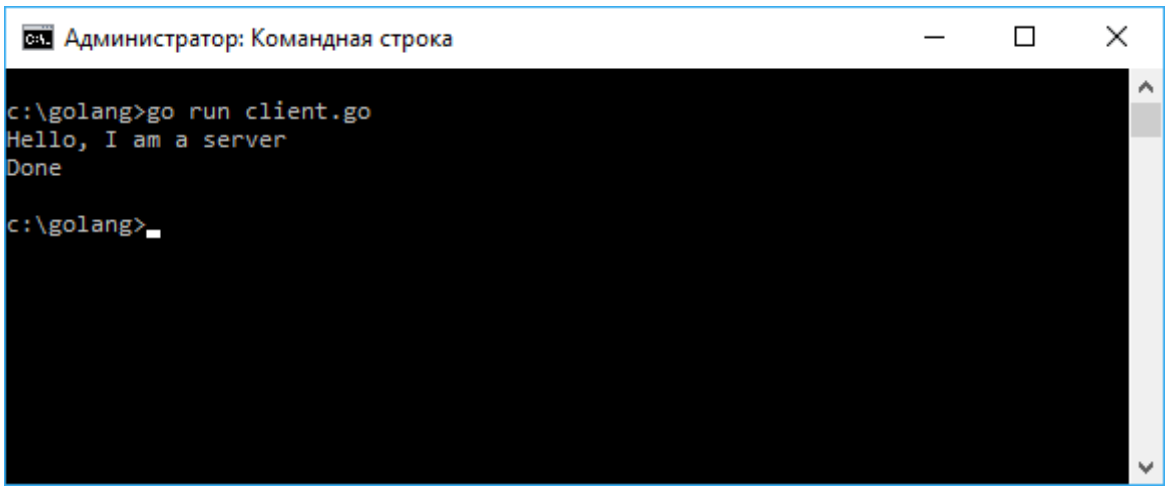

Рисунок 9.4. Вигляд вікна запуску клієнта

Після запуску клієнт підключиться до сервера та отримає від нього повідомлення.

9.3. Взаємодія клієнта та сервера

У попередньому розділі було розглянуто створення найпростішого **TCP** сервера, до якого підключається клієнт, і цьому клієнту відправляється деяке повідомлення. Тепер розглянемо, як клієнт може надсилати повідомлення та отримувати відповідь.

Для початку визначимо наступний сервер, наприклад:

```
1   package main
2   import (
3       "fmt"
```

```
4       "net"
5   )
6   var dict = map[string]string{
7       "red": "Червоний",
8       "green": "Зелений",
9       "blue": "Синій",
10      "yellow": "Жовтий",
11  }
12
13  func main() {
14      listener, err := net.Listen("tcp", ":4545")
15
16      if err != nil {
17          fmt.Println(err)
18          return
19      }
20      defer listener.Close()
21      fmt.Println("Server is listening...")
22      for {
23          conn, err := listener.Accept()
24          if err != nil {
25              fmt.Println(err)
26              conn.Close()
27              continue
28          }
29          go handleConnection(conn)  // Запускаємо
    горутину для обробки запиту
30      }
31  }
32  // Обробка підключення
33  func handleConnection(conn net.Conn) {
34      defer conn.Close()
35      for {
36          // Зчитуємо отримані у запиті дані
37          input := make([]byte, (1024 * 4))
38          n, err := conn.Read(input)
39          if n == 0 || err != nil {
40              fmt.Println("Read error:", err)
```

```
41              break
42          }
43          source := string(input[0:n])
44          // На підставі отриманих даних отримуємо зі
    словника переклад
45          target, ok := dict[source]
46          if ok == false{             // Якщо дані не
    знайдені у словнику
47              target = "undefined"
48          }
49          // Виводимо на консоль сервера діагностичну
    інформацію
50          fmt.Println(source, "-", target)
51          // Надсилаємо дані клієнту
52          conn.Write([]byte(target))
53      }
54  }
```

Сервер імітує поведінку програми для перекладу слів. Для цього визначено словник **dit**, який містить англомовні слова та їхній переклад.

У безкінечному циклі сервер приймає підключення. Однак замість прямої обробки підключення сервер запускає горутину у вигляді функції **handleConnection**, в якій обробляється підключення. Це дозволить клієнтам не чекати, поки перший з них буде оброблений. Таким чином, всі вхідні клієнти до певної міри будуть оброблятися одночасно.

У функції **handleConnection** отримується запит від клієнта. Для цього виділяється буфер достатньої довжини, наприклад 4096 байт, а саме.

```
1   input := make([]byte, (1024 * 4))
2   n, err := conn.Read(input)
```

В наведеному прикладі очікується, що запит від клієнта не перевищить 4096 байт, однак точний розмір запиту та його максимальний розмір не завжди бувають відомі. У цьому випадку можна застосовувати різні техніки, зокрема, в нескінченному циклі зчитувати дані запиту від клієнта і потім їх обробляти. Але спочатку розглянемо простішу ситуацію.

Отримавши запит і перетворивши його рядок, отримуємо значення зі словника та відправляємо його назад клієнту, а саме:

```
1   conn.Write([]byte(target))
```

Для взаємодії з цим сервером визначемо наступний клієнт:

```
1   package main
```

```
2   import (
3       "fmt"
4       "net"
5   )
6   func main() {
7
8       conn, err := net.Dial("tcp", "127.0.0.1:4545")
9       if err != nil {
10          fmt.Println(err)
11          return
12      }
13      defer conn.Close()
14      for{
15          var source string
16          fmt.Print("Введіть слово: ")
17          _, err := fmt.Scanln(&source)
18          if err != nil {
19              fmt.Println("Некоректне введення ", err)
                continue
20          }
21          // Надсилаємо повідомлення серверу
22          if n, err := conn.Write([]byte(source));
23          n == 0 || err != nil {
24              fmt.Println(err)
25              return
26          }
27          // Отримуємо відповідь
28          fmt.Print("Переклад:")
29          buff := make([]byte, 1024)
30          n, err := conn.Read(buff)
31          if err !=nil{ break}
32          fmt.Print(string(buff[0:n]))
33          fmt.Println()
34      }
35  }
36
```

На клієнті в нескінченному циклі вводиться слово для перекладу та надсилається серверу повідомлення, наприклад:

```
1   if n, err := conn.Write([]byte(source));
```

Після цього отримуємо від сервера відповідь та виводимо її на консоль. Так як відповідь від сервера може бути змінної довжини, то для отримання відповіді в нескінченному циклі зчитуємо дані за допомогою метода **Read**, а саме:

```
1   buff := make([]byte, 1024)
2   n, err := conn.Read(buff)
3   if err !=nil{ break}
4   fmt.Print(string(buff[0:n]))
```

Запустимо сервер, як наведено на рисунку 9.5.

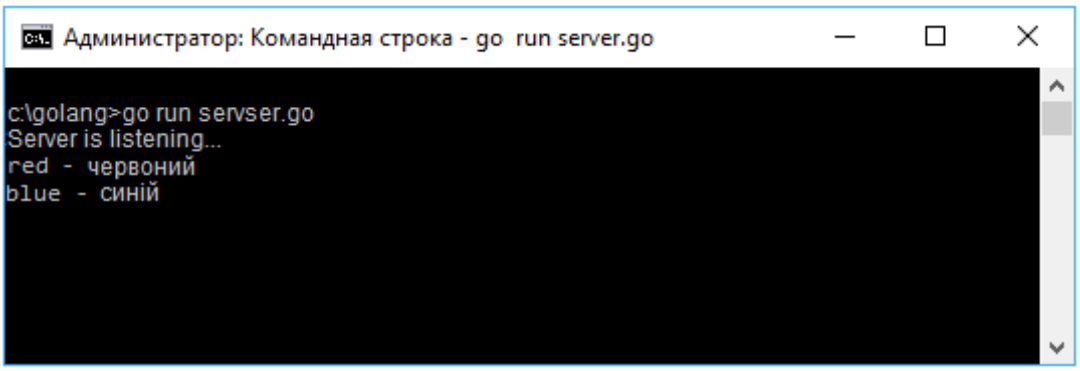

Рисунок 9.5. Вигляд вікна запуску сервера

Потім запустимо програму клієнта і введемо якесь значення, як наведено на рисунку 9.6. І сервер поверне переклад слів.

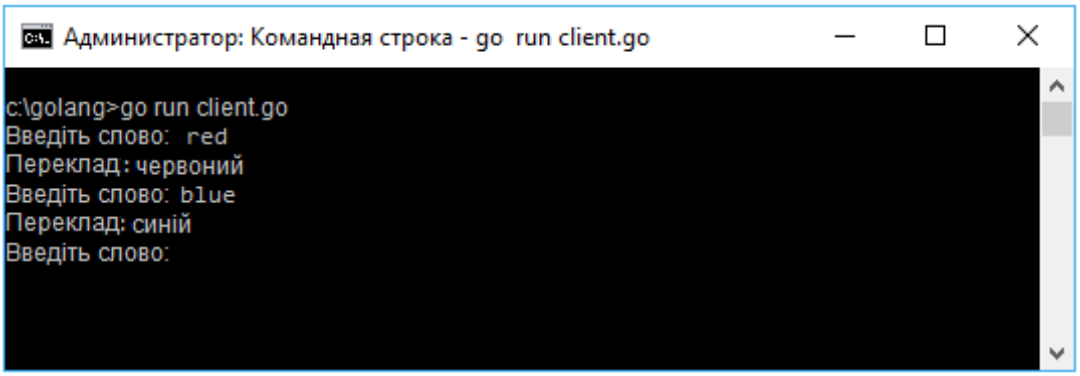

Рисунок 9.6. Результат роботи програми

Для виходу із програм сервера та клієнта необхідно натиснути комбінацію клавіш **Ctrl+C**.

9.4. Встановлення тайм-ауту

При взаємодії клієнта та сервера можна встановлювати тайм-аут, після якого з'єднання між сервером і клієнтом за відсутності взаємодії буде розірвано. Для цього тип **net.Conn** містить наступні методи:

- **SetDeadline(t time.Time) error** : встановлює тайм-аут на всі операції введення-виведення. Для встановлення часу застосовується структура **time.Time**;
- **SetReadDeadline(t time.Time) error** : встановлює тайм-аут на операції введення в потік;
- **SetWriteDeadline(t time.Time) error** : встановлює тайм-аут на операції виведення з потоку.

Розглянемо, коли вони можуть стати в нагоді? Раніше було розглянуто взаємодію сервера та клієнта. Для читання даних від клієнта сервер використовував буфер фіксованого розміру, а саме:

```
input := make([]byte, (1024 * 4))
n, err := conn.Read(input)
```

Однак у низці ситуацій це не найкращий спосіб, особливо коли розмір переданих даних перевищує розмір буфера. Наперед може бути точно не відомо скільки даних поверне нам сервер. Тому визначимо наступний код клієнта:

```
package main
import (
    "fmt"
    "net"
    "time"
)
func main() {

    conn, err := net.Dial("tcp", "127.0.0.1:4545")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer conn.Close()
    for{
        var source string
        fmt.Print("Введіть слово: ")
        _, err := fmt.Scanln(&source)
```

```
19          if err != nil {
20              fmt.Println("Некоректне введення", err)
21              continue
22          }
23          // Надсилаємо повідомлення серверу
24          if n, err := conn.Write([]byte(source));
25          n == 0 || err != nil {
26              fmt.Println(err)
27              return
28          }
29          // Отримуємо відповідь
30          fmt.Print("Переклад:")
31          conn.SetReadDeadline(time.Now().Add(time.Sec
    ond * 5))
32          for{
33              buff := make([]byte, 1024)
34              n, err := conn.Read(buff)
35              if err !=nil{ break}
36              fmt.Print(string(buff[0:n]))
37              conn.SetReadDeadline(time.Now().Add(time
    .Millisecond * 700))
38          }
39          fmt.Println()
40      }
41  }
```

В цьому прикладі для отримання даних створено окремий цикл **for**, а саме:

```
1   for{
2       buff := make([]byte, 1024)
3       n, err := conn.Read(buff)
4       if err !=nil{ break}
5       fmt.Print(string(buff[0:n]))
6       conn.SetReadDeadline(time.Now().Add(time.Milliseco
    nd * 700))
7   }
```

Тому, якщо сервер передасть більше 1024 байт, то всі вони однаково будуть оброблені. Але крім того, тут також встановлюється тайм-аут на читання даних. Перед самим циклом встановлюється тайм в 5 секунд, а саме:

```
conn.SetReadDeadline(time.Now().Add(time.Second * 5))
```

Це означає, що клієнт може очікувати на читання від сервера протягом 5 секунд. Після цього часу операція читання генерує помилку і відбувається вихід із циклу, де була спроба прочитати дані від сервера. П’ять секунд – це досить великий період, але на початку перед першою взаємодією краще встановлювати більший період. І потім після прочитання перших 1024 байт тайм-аут скидається до 700 мілісекунд. Тобто якщо протягом наступних 700 мілісекунд сервер не надішле жодних даних, то відбувається вихід із циклу і відповідно читання даних закінчується.

Важливо розуміти роль таких затримок, оскільки вони дозволяють згенерувати помилку під час читання даних. Отже можна отримати цю помилку і належним чином її обробити, наприклад, вийти з нескінченного циклу. Якби установка тайм-ауту не використовувалась, то могла б скластися ситуація, коли сервер очікував дані від клієнта в операції читання, а клієнт очікував дані від сервера також в операції читання. І було б свого роду блокування.

Код сервера залишається тим самим, що й у минулому розділі, а саме:

```
package main
import (
    "fmt"
    "net"
)
var dict = map[string]string{
    "red": "Червоний",
    "green": "Зелений",
    "blue": "Синій",
    "yellow": "Жовтий",
}

func main() {
    listener, err := net.Listen("tcp", ":4545")

    if err != nil {
        fmt.Println(err)
        return
    }
    defer listener.Close()
    fmt.Println("Server is listening...")
    for {
```

```
23          conn, err := listener.Accept()
24          if err != nil {
25              fmt.Println(err)
26              conn.Close()
27              continue
28          }
29          go handleConnection(conn)  // Запускаємо
    горутину для обробки запиту
30      }
31  }
32  // Обробка підключення
33  func handleConnection(conn net.Conn) {
34      defer conn.Close()
35      for {
36          // Зчитуємо отримані у запиті дані
37          input := make([]byte, (1024 * 4))
38          n, err := conn.Read(input)
39          if n == 0 || err != nil {
40              fmt.Println("Read error:", err)
41              break
42          }
43          source := string(input[0:n])
44          // На підставі отриманих даних отримуємо зі
    словника переклад
45          target, ok := dict[source]
46          if ok == false{              // Якщо дані не
    знайдені у словнику
47              target = "undefined"
48          }
49          // Виводимо на консоль сервера діагностичну
    інформацію
50          fmt.Println(source, "-", target)
51          // Надсилаємо дані клієнту
52          conn.Write([]byte(target))
53      }
54  }
```

9.5. Надсилання запитів по HTTP

Особливу сферу застосування у мові Go мають запити за протоколом **HTTP**. Протокол **HTTP** працює поверх **TCP**, і практично можна написати програму, яка приймає або надсилає запити за протоколом **TCP** і тим самим надсилати та отримувати запити і за протоколом **HTTP**. Однак у зв'язку з тим, що цей протокол і в цілому мережа Інтернет грає велику роль, то всі відповідні функції по роботі з **http** були виділені в окремий пакет **net/http**, документацію на який можна отримати за посиланням **https://golang.org/pkg/net/http/** [9].

Для надсилання запитів у пакеті **net/http** визначено низку функцій, а саме:

```
1  func Get(url string) (resp *Response, err error)
2  func Head(url string) (resp *Response, err error)
3  func Post(url string, contentType string, body
   io.Reader) (resp *Response, err error)
4  func PostForm(url string, data url.Values) (resp
   *Response, err error)
```

Призначення наведених функцій наступне:

- **Get()** : надсилає запит **GET**;
- **Head()** : надсилає запит **HEAD**;
- **Post()** : надсилає запит **POST**;
- **PostForm()** : надсилає форму у запиті **POST**.

Розглянемо виконання найпростішого запиту, а саме запиту **GET**, для якого застосовується однойменний метод, наприклад:

```
1  package main
2  import (
3      "fmt"
4      "net/http"
5  )
6  func main() {
7      resp, err := http.Get("https://google.com")
8      if err != nil {
9          fmt.Println(err)
10         return
11     }
12     defer resp.Body.Close()
13     for true {
14
15         bs := make([]byte, 1014)
```

```
16          n, err := resp.Body.Read(bs)
17          fmt.Println(string(bs[:n]))
18
19          if n == 0 || err != nil{
20              break
21          }
22      }
23  }
```

В наведеному прикладі метод **Get()** як параметр приймає адресу ресурсу, до якого потрібно виконати запит, і повертає об'єкт ***http.Response**, який інкапсулює відповідь. Поле **Body** структури **http.Response** представляє відповідь від вебресурсу і також представляє інтерфейс **io.ReadCloser**. А це означає, що це поле по суті є потоком для читання, і можна зчитати дані через метод **Read**. І крім цього, для того, щоб закрити потік, необхідно викликати метод **Close**. Тому після запиту викликається метод **defer resp.Body Close()**, і в циклі зчитується через метод **Read** дані, які виводяться на консоль, як показано на рисунку 9.7.

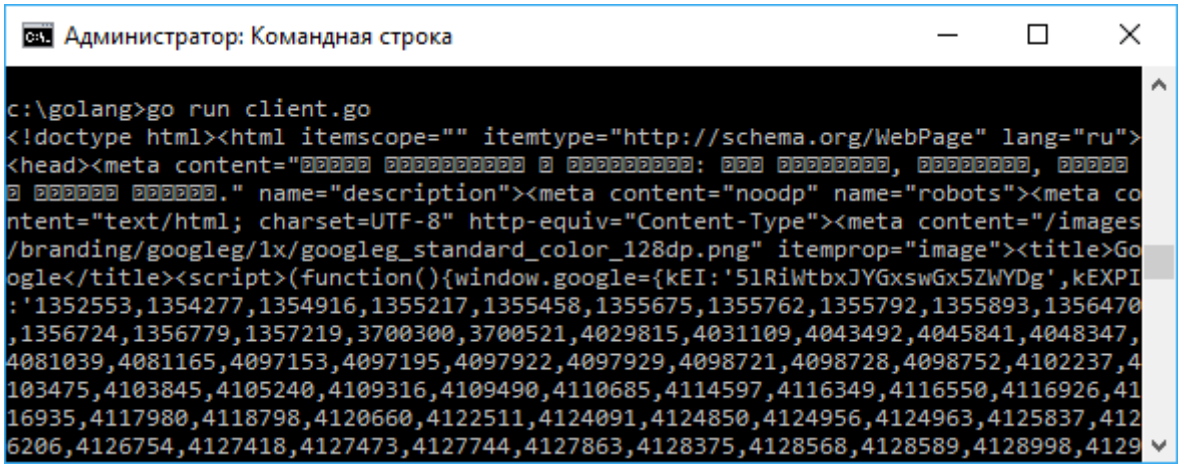

Рисунок 9.7. Результат роботи програми

Оскільки в наведеному прикладі відповідь від вебресурсу все одно

```
1  package main
2  import (
3      "fmt"
4      "net/http"
5      "io"
6      "os"
7  )
8  func main() {
```

```
9       resp, err := http.Get("https://google.com")
10      if err != nil {
11              fmt.Println(err)
12              return
13      }
14      defer resp.Body.Close()
15      io.Copy(os.Stdout, resp.Body)
16  }
```

9.6. Створення HTTP-запитів за допомогою структури http.Client

Для здійснення **HTTP-запитів** може застосовуватися структура **http.Client**. Щоб надіслати запит до вебресурсу, можна використати один із її методів, а саме:

```
1   func (c *Client) Do(req *Request) (*Response, error)
2   func (c *Client) Get(url string) (resp *Response, err
    error)
3   func (c *Client) Head(url string) (resp *Response,
    err error)
4   func (c *Client) Post(url string, contentType string,
    body io.Reader) (resp *Response, err error)
5   func (c *Client) PostForm(url string, data
    url.Values) (resp *Response, err error)
```

Багато в чому вони аналогічні тим однойменним функціям, за винятком метода **Do**, визначених у пакеті **net/http** і які були розглянуті в попередньому розділі. Наприклад, виконання найпростішого запиту **GET** виглядатиме наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "io"
6       "os"
7   )
8   func main() {
9       client := http.Client{}
10      resp, err := client.Get("https://google.com")
11      if err != nil {
```

```
12            fmt.Println(err)
13            return
14        }
15        defer resp.Body.Close()
16        io.Copy(os.Stdout, resp.Body)
17    }
```

9.6.1. Налаштування клієнта

Структура **http.Client** має низку полів, які дозволяють налаштувати її поведінку, а саме:

- **Timeout** : встановлює тайм-аут для запиту;
- **Jar** : встановлює кукі, що надсилаються у запиті;
- **Transport** : визначає механізм виконання запиту.

Розглянемо приклад встановлення тайм-ауту:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "io"
6       "os"
7       "time"
8   )
9   func main() {
10      client := http.Client{
11          Timeout: 6 * time.Second,
12      }
13      resp, err := client.Get("https://google.com")
14      if err != nil {
15          fmt.Println(err)
16          return
17      }
18      defer resp.Body.Close()
19      io.Copy(os.Stdout, resp.Body)
20  }
```

Властивість **Timeout** представляє об'єкт **time.Duration**, і в наведеному прикладі вона дорівнює 6 секундам.

9.6.2. Керування запитами за допомогою об’єкту http.Request

Для керування запитами та їх параметрами у мові Go використовується об'єкт **http.Request**. Він дозволяє встановити різні параметри, додати кукі, заголовки та визначити тіло запиту. Для створення об'єкта **http.Request** застосовується функція **http.NewRequest()**, формальне визначення якої наступне:

```
1   func NewRequest(method, url string, body io.Reader)
    (*Request, error)
```

Функція приймає три параметри. Перший параметр – це тип запиту у вигляді рядка ("**GET**", "**POST**"). Другий параметр – це адреса ресурсу. Третій параметр – це тіло запиту.

Для надсилання об'єкта **Request** можна застосовувати метод **Do()**, наприклад:

```
1   Do(req *http.Request) (*http.Response, error)
```

Повний код виглядатиме наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "io"
6       "os"
7   )
8   func main() {
9       client := &http.Client{}
10      req, err := http.NewRequest(
11          "GET", "https://google.com", nil,
12      )
13      // Додаємо заголовки
14      req.Header.Add("Accept", "text/html")    //
    Додаємо заголовок Accept
15      req.Header.Add("User-Agent", "MSIE/15.0")    //
    Додаємо заголовок User-Agent
16
17      resp, err := client.Do(req)
18      if err != nil {
19          fmt.Println(err)
20          return
```

```
21      }
22      defer resp.Body.Close()
23      io.Copy(os.Stdout, resp.Body)
24  }
```

Резюме

У наведеному розділі розглядається мережеве програмування на Go за допомогою пакета **net**. Описується функція **net.Dial()**, яка дозволяє встановлювати з'єднання через протоколи TCP, UDP, IP та Unix-сокети. Наведено приклад коду, що демонструє створення TCP-з'єднання з вебресурсом, надсилання HTTP-запиту та отримання відповіді. Пояснюється, як об'єкт **net.Conn** використовується для читання і запису даних, оскільки реалізує інтерфейси **io.Reader** та **io.Writer**. Також зазначено, що для роботи з HTTP-запитами краще використовувати пакет **net/http**, який спрощує роботу з протоколом HTTP.

У прикладах сервер приймає підключення, відправляє повідомлення клієнту та закриває з'єднання. Для забезпечення одночасної роботи з кількома клієнтами сервер використовує горутини. Також пояснюється, як встановлювати тайм-аути на читання та запис даних для запобігання блокуванню у випадках довгого очікування. Завдяки цим механізмам забезпечується надійна та ефективна взаємодія між клієнтом і сервером. Наведено код клієнта, який підключається до сервера через **net.Dial**, отримує повідомлення і виводить його в консоль. Матеріал демонструє основні методи роботи з мережевими підключеннями, зокрема **Accept**, **Write** і **Close**. Для тестування потрібно спочатку запустити сервер, а потім клієнт, який отримає від нього відповідь.

Описані основні функції, такі як **Get**, **Post**, **PostForm**, які дозволяють надсилати запити різних типів. Показано, як обробляти відповіді серверу через структуру **http.Response**, читаючи тіло запиту та закриваючи потік. Також розглянуто використання структури **http.Client** для налаштування запитів, наприклад, встановлення тайм-ауту. Окрему увагу приділено створенню власних запитів через об’єкт **http.Request**, де можна задавати заголовки, параметри та тіло запиту. Наведено практичні приклади коду для кожного випадку.

Наступні розділи об’єднають всі отримані знання та навички з мережевого програмування, інтеграції вебзастосунків з базами даних та створення динамічних вебзастосунків.

Контрольні запитання і завдання

1. Який пакет у Go використовується для роботи з мережевими з'єднаннями?
2. Яка основна функція використовується для встановлення мережевого з'єднання?
3. Які параметри приймає функція **net.Dial()**?
4. Які основні протоколи підтримує функція **net.Dial()**?
5. Який об'єкт повертає функція **net.Dial()** при успішному з'єднанні?
6. Що повертає функція **net.Listen** у разі успішного виконання?
7. Як правильно завершити роботу сервера після використання **net.Listener**?
8. Що таке горутина, і яку роль вона відіграє у серверній частині програми?
9. Як сервер обробляє запити від декількох клієнтів одночасно?
10. Що відбувається, якщо при прийнятті підключення на сервері виникає помилка?
11. Яка різниця між методами **SetReadDeadline** і **SetWriteDeadline**?
12. Як уникнути блокування при одночасному очікуванні клієнта і сервера?
13. Що таке структура **http.Response**, і які її основні поля?
14. Чим функція **http.Client.Get** відрізняється від **http.Get**?
15. Що таке **http.Request**, і для чого він використовується?
16. Яка функція використовується для надсилання об'єкта **http.Request**?
17. Які основні переваги використання **http.Client** порівняно з простими функціями пакета **net/http**?
18. Поясніть, як функція **io.Copy()** використовується для отримання даних із з'єднання та виводу їх на консоль.
19. Використовуючи пакет **net**, напишіть програму, яка підключається до сервера IPv6-адреси [2001:db8::1]:8080 і надсилає простий рядок.
20. Що робити, якщо під час виконання функції net.Dial() виникає помилка підключення? Як правильно обробити цю ситуацію?
21. Напишіть програму, яка підключається до Unix-сокета **/tmp/mysocket** та надсилає команду "**status**", отримуючи відповідь від сервера.
22. Реалізуйте клієнтську програму, яка зчитує дані від сервера та записує їх у файл замість консолі. Використайте функції пакета **os** для роботи з файлами.

23. Напишіть програму-клієнт, яка виконує автоматичні спроби підключення до сервера кожні 5 секунд, якщо сервер недоступний. Використайте таймери та обробку помилок.
24. Реалізуйте логування підключень до сервера, щоб у консоль виводився IP-адрес клієнта та час підключення. Скористайтеся функціями пакета **time** для роботи з датою й часом.
25. Додайте до серверного коду функцію, яка зчитує дані від клієнта та зберігає їх у локальний файл на сервері. Переконайтеся, що сервер працює правильно з кількома клієнтами.
26. Модифікуйте сервер так, щоб він підтримував шифроване підключення за допомогою TLS. Додайте пакет **crypto/tls** і налаштуйте необхідні сертифікати.
27. Реалізуйте логування на сервері, щоб усі запити клієнтів записувалися в текстовий файл із зазначенням часу.
28. Створіть програму для стрес-тестування сервера, яка одночасно відкриває 100 з’єднань і надсилає запити.
29. Додайте функціонал автентифікації до сервера, де клієнт повинен спочатку відправити ім’я користувача і пароль для отримання доступу.
30. Модифікуйте сервер, щоб він повертав клієнту список усіх слів зі словника, якщо отримує спеціальну команду (наприклад, "**LIST**").

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Який пакет у Go використовується для роботи з мережевими з'єднаннями?

a) os
b) fmt
c) net
d) io

Правильна відповідь: c).

Яка функція використовується для встановлення з'єднання з мережею?

a) net.Listen()
b) net.Dial()
c) net.Connect()

d) net.Open()

Правильна відповідь: b).

Яка функція використовується для надсилання GET-запиту?

a) http.Post

b) http.Get

c) http.Do

d) http.Head

Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які протоколи підтримує функція net.Dial()? (Оберіть кілька)

a) tcp

b) udp

c) http

d) unix

Правильна відповідь: a), b), d).

Що необхідно для встановлення TCP-з'єднання в Go?

a) Адреса сервера

b) Словник перекладів

c) Номер порту

d) Мережевий протокол UDP

Правильна відповідь: a), c).

Завдання на відповідність

Співвіднесіть протоколи з їх призначенням:

a) tcp → 1. Надійна передача даних

b) udp → 2. Швидка передача даних без перевірки

c) unix → 3. Локальне з'єднання

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Встановіть відповідність між методами і їх використанням у клієнті:

a) fmt.Scanln → 1. Отримання введення від користувача

b). conn.Write → 2. Надсилання даних серверу

c) conn.Read → 3. Отримання відповіді від сервера

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

Вставити пропущені слова

Функція net.Dial("tcp", "google.com:80") повертає об'єкт типу _____.

Правильна відповідь: …net.Conn…

Для встановлення підключення клієнт використовує функцію _____.

Правильна відповідь: …net.Dial…

Завдання з вибором істинності

Чи можна використовувати функцію net.Dial() для підключення до HTTP-сервера?

a) Так

b) Ні

Правильна відповідь: a).

Чи потрібно явно закривати з'єднання, отримане через net.Dial()?

a) Так

b) Ні

Правильна відповідь: a).

Завдання з кодом

Що відбудеться, якщо у наведеному коді сервер не викликає listener.Close()?

```
listener, err := net.Listen("tcp", ":4545")
if err != nil {
    return
}
defer listener.Close()
```

a) Сервер продовжить працювати після завершення програми.

b) Порт залишиться зайнятим.

c) Сервер автоматично завершить роботу.

d) Клієнти не зможуть підключатися.

Правильна відповідь: b).

Відкриті запитання

Виправте помилку в коді:

```
listener, err := net.Dial("tcp", ":4545")
if err != nil {
    fmt.Println("Error:", err)
}
```

Правильна відповідь:

```
listener, err := net.Listen("tcp", ":4545")
if err != nil {
    fmt.Println("Error:", err)
}
```

Напишіть код, який дозволяє клієнту автоматично повторно підключатися до сервера у разі втрати з'єднання.

Правильна відповідь:

```
for {
    conn, err := net.Dial("tcp", "127.0.0.1:4545")
    if err != nil {
        fmt.Println("Не вдалось підключитись, спроба знову...")
        time.Sleep(2 * time.Second)
        continue
    }
    defer conn.Close()
    break
}
```

10. БАЗИ ДАНИХ

В цьому розділі розглянемо механізми та можливості взаємодії програм, що написані на мові Go, з базами даних.

10.1. Робота з реляційними базами даних

Для роботи з реляційними базами даних у мові Go використовується вбудований пакет **database/sql**, документація на який доступна за посиланням **https://golang.org/pkg/database/sql/** [10]. Однак він не використовується самостійно. Він лише надає універсальний інтерфейс до роботи з базами даних. Для роботи з конкретною СКБД потрібний відповідний драйвер. Список доступних драйверів можна знайти за посиланням **https://go.dev/wiki/SQLDrivers/** [11]. Однак оскільки драйвера реалізують одні й ті самі інтерфейси, то робота з різними СКБД буде майже ідентична.

Для того, щоб розпочати роботу з базою даних, необхідно відкрити підключення за допомогою функції **Open()**, формальний опис якої наступний:

```
1   func Open(driverName, dataSourceName string) (*DB,
    error)
```

Ця функція приймає у якості параметри ім'я драйвера та ім'я джерела даних, до якого треба підключатися. Повертає функція об'єкт **DB**, зо насправді є базою даних, з якою можна взаємодіяти. Якщо не вдалося підключити до джерела даних, то в об'єкті **error** можна знайти відомості про помилку.

Потім взаємодія з базою даних здійснюється за допомогою методів об'єкта **DB**, наприклад.

```
1   func (db *DB) Exec(query string, args ...interface{})
    (Result, error)
2   func (db *DB) Query(query string, args
    ...interface{}) (*Rows, error)
3   func (db *DB) QueryRow(query string, args
    ...interface{}) *Row
4   func (db *DB) Close() error          // Закриває
    підключення
```

Розглянемо методи об'єкта **DB** детальніше.

Метод **Exec()** виконує деякий **sql-вираз**, який передається через перший параметр, не повертаючи жодного результату. Метод також приймає додаткові параметри, за допомогою яких можна передати значення у виконуваний **sql-**

вираз. Абстрактна операція додавання даних до СКБД, яка передбачає виконання команди **INSERT** виглядатиме наступним чином:

```
result, err := db.Exec("INSERT INTO Products (model, company, price) VALUES ('iPhone X', 'Apple', 72000)")
```

Спосіб вставки додаткових параметрів у **SQL-вираз** залежить від конкретного драйвера. Також цей метод підходить для виконання команд **UPDATE** (оновлення) та **DELETE** (видалення).

Метод **Exec()** повертає об'єкт типу **Result**. Цей інтерфейс визначає два методи, а саме:

```
LastInsertId() (int64, error)    // Повертає id останнього рядка, який був доданий/оновлений/видалений
RowsAffected() (int64, error)    // Повертає кількість задіяних рядків
```

Метод **Query()** призначений для виконання запиту, який повертає будь-які дані. Зазвичай, це запити, які містять команду **SELECT**, наприклад.

```
rows, err := db.Query("SELECT name FROM users WHERE age=23")
```

Результатом запиту є об'єкт ***Rows**, який є набором рядків. За допомогою низки його методів можна видобути отримані дані, наприклад:

```
func (rs *Rows) Columns() ([]string, error)      // Повертає назви стовпців набору
func (rs *Rows) Next() bool                      // Повертає true, якщо в наборі є ще один рядок і переходить до нього
func (rs *Rows) Scan(dest ...interface{}) error      // Зчитує отримані рядки в змінні
func (rs *Rows) Close() error                    // Закриває об'єкт Rows для подальшого читання
```

Загальний принцип читання набору рядків виглядає наступним чином:

```
rows, err := db.Query("SELECT ...")
...
defer rows.Close()
for rows.Next() {
    var id int
    var name string
    rows.Scan(&id, &name)
    fmt.Println(id, name)
}
```

Спочатку виконується запит до СКБД за допомогою метода **db.Query**, потім за допомогою метода **Next()** зчитуються послідовно всі рядки з набору. Якщо рядків у наборі немає, то метод повертає **false**, і відбувається вихід із циклу. Якщо рядки ще є, то покажчик ***Rows** переходить до наступного рядка. І потім можна зчитати у змінні за допомогою метода **Scan()** дані з поточного рядка.

Метод **QueryRow()** повертає один рядок у вигляді об'єкта ***Row**. Як правило, цей метод застосовується для отримання одиничного об'єкта, наприклад, **id**. Цей об'єкт має метод **Scan()**, який дозволяє видобути дані з рядка, наприклад:

```
1   func (r *Row) Scan(dest ...interface{}) error
```

Також варто відзначити, що мова **Go** підтримує створення запитів за допомогою об'єкта **Stmt**, в який можна вводити різні дані та який підвищує продуктивність. І також у мові **Go** є підтримка транзакцій у вигляді об'єкта **Tx**.

Всі ці речі по-різному реалізуються у різних драйверах для конкретних СКБД. Але загальні принципи будуть одні й ті самі. Тобто загальна структура роботи з різними базами даних завдяки єдиному інтерфейсу збігатиметься.

10.2. Робота з MySQL

Для роботи з **MySQL** використовуватимемо драйвер **Go MySQL Driver**, який можна знайти за посиланням **https://github.com/go-sql-driver/mysql/** [12]. Спочатку додамо цей драйвер до змінної **$GOPATH**. Для цього виконаємо в командному рядку/терміналі наступну команду:

```
1   go get github.com/go-sql-driver/mysql
```

Створимо на сервері **MySQL** базу даних **productdb** і в ній таблицю **products**. Для цього можна використовувати наступні команди **SQL**:

```
1   create database productdb;
2   use productdb;
3   create table products (
4       id int auto_increment primary key,
5       model varchar(30) not null,
6       company varchar(30) not null,
7       price int not null
8   )
```

В наведеному прикладі створена база даних **productdb**, в якій є таблиця **products**, яка зберігатиме інформацію про товари. Таблиця **products** буде

складатись з чотирьох стовпців: **id** - ідентифікатор кожного запису; **model** - назва товару; **company** - виробник товару; **price** - ціна товару.

10.2.1. Додавання даних

Створена база даних порожня, тому додамо до неї якісь дані, наприклад:

```
1  package main
2  import (
3      "database/sql"
4      "fmt"
5      _ "github.com/go-sql-driver/mysql"
6  )
7
8  func main() {
9      db, err := sql.Open("mysql",
10 "root:password@/productdb")

11     if err != nil {
12         panic(err)
13     }
14     defer db.Close()
15
16     result, err := db.Exec("insert into
17 productdb.Products (model, company, price) values (?,
   ?, ?)",
           "iPhone X", "Apple", 72000)
18     if err != nil{
19         panic(err)
20     }
21     fmt.Println(result.LastInsertId())  // id
   доданого об'єкта
22     fmt.Println(result.RowsAffected())  // Кількість
   задіяних рядків
23 }
```

Потім необхідно підключити всі необхідні пакети. Для роботи з реляційною базою даних потрібний пакет **database/sql**. І тому що буде використовуватись **mysql**, то також підключаємо пакет **github.com/go-sql-driver/mysql**, причому варто звернути увагу, що перед ним стоїть знак

підкреслення (_). Цей знак дозволяє при завантаженні пакета ініціалізувати його за допомогою виклику функції **init**.

Далі відкривається підключення за допомогою функції **Open**, а саме:

```
1   sql.Open("mysql", "root:password@/productdb")
```

Першим аргументом функції є назва драйвера, у наведеному прикладі це **mysql**. Другий параметр має формат **логін: пароль@/база_даних**. Логін та пароль повинні бути такі, які були вказані для **mysql** під час її встановлення. У наведеному прикладі логін має значення **root**, а пароль має значення **password**. Назвою бази даних є **productdb**, яка була створена раніше.

Функція **Open** повертає об'єкт бази даних **DB**. І для додавання даних у цього об'єкта викликається метод **Exec()**, а саме:

```
1   result, err := db.Exec("insert into productdb.Products
    (model, company, price) values (?, ?, ?)",
2           "iPhone X", "Apple", 72000)
```

Першим аргументом функції є **sql-вираз**, який додає рядок до таблиці **Products**. Знаки питання у цьому виразі представляють плейсхолдери, замість яких вставляються значення, що передаються через другий, третій та наступні параметри. Тобто в таблиці **Products** чотири стовпці, але один з них генерується автоматично, а саме **id**. Тому необхідно передати тільки три значення для стовпців **model**, **company** та **price**. Для цього у виразі міститься три знаки питання і відповідно функція приймає три додаткові параметри. Всі додаткові параметри передаються в **sql-вираз** на місце плейсхолдерів за позицією. Тобто перший параметр замість першого плейсхолдера і так далі.

Результат виконання функції потрапляє у змінну **result**, яка зберігає результат виконання операції у базі даних. Зокрема, за допомогою метода **result.LastInsertId()** можна отримати **id** останнього доданого об'єкта, а за допомогою метода **result.RowsAffected()** кількість доданих рядків.

При виконанні наведеного коду отримаємо результати, які наведені на рисунку 10.1.

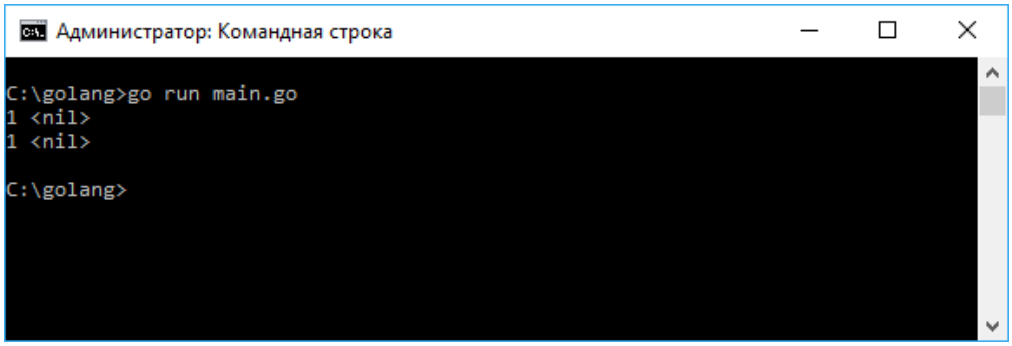

Рисунок 10.1. Результат роботи програми

При цьому необов'язково визначати всі дані, що додаються через параметри, можна ввести їх безпосередньо за допомогою **sql-вираз,** наприклад:

```
result, err := db.Exec("insert into productdb.Products
(model, company, price) values ('Pixel 2', 'Google',
64000)")
```

10.2.2. Отримання даних

Розглянемо механізм отримання раніше доданих даних, наприклад:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)

type product struct{
    id int
    model string
    company string
    price int
}
func main() {
    db, err := sql.Open("mysql",
"root:password@/productdb")

    if err != nil {
        panic(err)
    }
    defer db.Close()
    rows, err := db.Query("select * from
productdb.Products")
    if err != nil {
        panic(err)
    }
    defer rows.Close()
    products := []product{}

    for rows.Next(){
```

```
29          p := product{}
30          err := rows.Scan(&p.id, &p.model, &p.company,
    &p.price)
31          if err != nil{
32              fmt.Println(err)
33              continue
34          }
35          products = append(products, p)
36      }
37      for _, p := range products{
38          fmt.Println(p.id, p.model, p.company, p.price)
39      }
40  }
```

Для роботи з даними в наведеному прикладі визначена структура **product**, яка відповідає даним в таблиці **Products**.

Для отримання даних викликається метод **Query()**, а саме:

```
1   rows, err := db.Query("select * from
    productdb.Products")
```

Цей метод як параметр приймає **sql-вираз SELECT** для отримання всіх даних з таблиці **Products**. Результат вибірки потрапляє у змінну **rows**, що представляє покажчик на структуру **Rows**. Потім за допомогою метода **rows.Next()** можна послідовно перебрати всі рядки в отриманому наборі, наприклад:

```
1   for rows.Next(){
2       p := product{}
3       err := rows.Scan(&p.id, &p.model, &p.company,
    &p.price)
4       if err != nil{
5           fmt.Println(err)
6           continue
7       }
8       products = append(products, p)
9   }
```

Тип **Rows** визначає метод **Scan**, за допомогою якого можна зчитати всі отримані дані змінні. В наведеному прикладі зчитуються дані у структуру **Product** і потім додаються у зріз. Оскільки були отримані всі дані, тобто всі чотири стовпці, то відповідно у **Scan** передаються адреси чотирьох змінних.

Після зчитування даних у зріз можна робити з ними все, що завгодно, наприклад, вивести на консоль, як показано на рисунку 10.2.

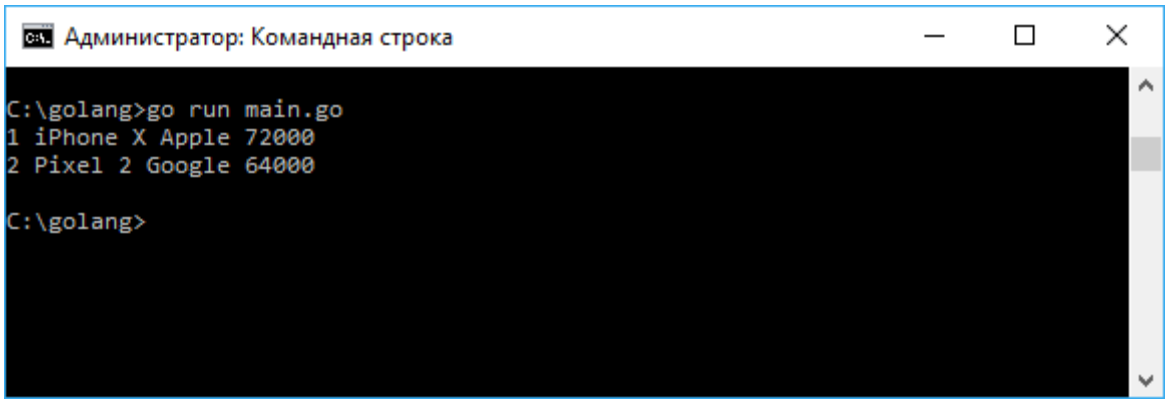

Рисунок 10.2. Результат роботи програми

У методі **Query** можна вказувати додаткові параметри. Приклад отримання товарів, у яких ціна більша за 70000 наступний:

```
rows, err := db.Query("select * from
productdb.Products where price > ?", 70000)
```

Якщо потрібно отримати лише один рядок, то можна використовувати метод **QueryRow()**, наприклад:

```
row := db.QueryRow("select * from productdb.Products
where id = ?", 2)
prod := product{}
err = row.Scan(&prod.id, &prod.model, &prod.company,
&prod.price)
if err != nil{
    panic(err)
}
fmt.Println(prod.id, prod.model, prod.company,
prod.price)
```

10.2.3. Оновлення

Для оновлення даних застосовується метод **Exec**, наприклад:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
)
```

```
7
8   func main() {
9       db, err := sql.Open("mysql",
    "root:password@/productdb")
10
11      if err != nil {
12          panic(err)
13      }
14      defer db.Close()
15      // Оновлюємо рядок з id=1
16      result, err := db.Exec("update productdb.Products
    set price = ? where id = ?", 69000, 1)
17      if err != nil{
18          panic(err)
19      }
20      fmt.Println(result.LastInsertId())
21      fmt.Println(result.RowsAffected())
22  }
```

10.2.4. Видалення

Для видалення даних також застосовується метод **Exec**, наприклад:

```
1   package main
2   import (
3       "database/sql"
4       "fmt"
5       _ "github.com/go-sql-driver/mysql"
6   )
7
8   func main() {
9       db, err := sql.Open("mysql",
    "root:password@/productdb")
10
11      if err != nil {
12          panic(err)
13      }
14      defer db.Close()
15      result, err := db.Exec("delete from
    productdb.Products where id = 1")
```

```
16      if err != nil{
17          panic(err)
18      }
19      fmt.Println(result.LastInsertId())  // id останнього видаленого об'єкта
20      fmt.Println(result.RowsAffected())  // Кількість задіяних рядків
21  }
```

10.3. Робота з PostgreSQL

Для роботи з СКБД **PostgreSQL** у мові Go можна використовувати різні драйвери, але для прикладу розглядатимемо **Pure Go Postgres driver**, який можна знайти за посиланням **https://github.com/lib/pq/** [13].

Спочатку перейдемо в командний рядок або термінал і встановимо цей драйвер за допомогою наступної команди, яка наведено на рисунку 10.3.

```
go get github.com/lib/pq
```

Рисунок 10.3. Зовнішній вигляд вікна введення команди

Нехай на сервері **PostgreSQL** буде база даних **productdb**, в якій є таблиця **Products**, яка описується наступним скриптом:

```
1   CREATE TABLE Products (
2       id    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
3       model  varchar(30) NOT NULL,
4       company varchar(30) NOT NULL,
5       price integer NOT NULL
6   );
```

Тобто в таблиці буде чотири стовпчики: **id**; **model**; **company**; **price**.

10.3.1. Відкриття підключення

Для відкриття з'єднання з базою даних у функцію **sql.Open()** передається ім'я драйвера **postgres** та рядок підключення, а саме:

```
1   connStr := "user=postgres password=mypass dbname=productdb sslmode=disable"
2   db, err := sql.Open("postgres", connStr)
```

Рядок підключення містить наступні параметри: **user** - логін на сервері **PostgreSQL**; **password** - пароль цього користувача; **dbname** - ім'я бази даних; **sslmode** - режим роботи з **SSL**. У наведеному прикладі логін має значення **postgres** (логін за замовчуванням на сервері), пароль має значення mypass, ім'я бази даних має значення **productdb**, а **ssl** відключений та має значення **disable**.

В результаті налаштування підключення метод **db.Open** повертає об'єкт ***DB**, через який можна буде взаємодіяти з СКБД.

Варто зазначити, що щойно розглянуті параметри не єдині. Повний список параметрів для рядка підключення до СКБД та їх значення можна переглянути в документації за посиланням **https://godoc.org/github.com/lib/pq/** [14].

10.3.2. Додавання даних

Для додавання даних використовується метод **Exec()**, наприклад:

```
1   package main
2   import (
3       "database/sql"
4       "fmt"
5       _ "github.com/lib/pq"
6   )
7
8   func main() {
9
10      connStr := "user=postgres password=mypass
    dbname=productdb sslmode=disable"
11      db, err := sql.Open("postgres", connStr)
12      if err != nil {
13          panic(err)
14      }
15      defer db.Close()
16
17      result, err := db.Exec("insert into Products
    (model, company, price) values ('iPhone X', $1, $2)",
18          "Apple", 72000)
19      if err != nil{
20          panic(err)
21      }
```

```
22      fmt.Println(result.LastInsertId())  // Не
    підтримується
23      fmt.Println(result.RowsAffected())  // Кількість
    доданих рядків
24  }
```

Sql-вираз, що виконується, може отримувати значення через додаткові параметри метода **db.Exec()**. У самому **sql-виразі** такі значення представлені плейсхолдерами **$1**, **$2** і так далі, замість яких вставляються значення додаткових параметрів метода **db.Exec**.

Варто звернути увагу, що цей драйвер не підтримує метод **result.LastInsertId()**, який повертав би значення **id** останнього доданого об'єкта, як показано на рисунку 10.4.

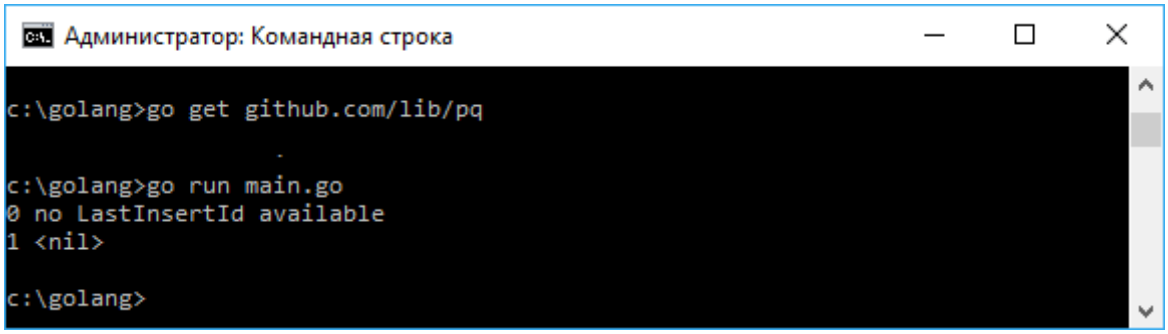

Рисунок 10.4. Результат роботи програми

Якщо ж обов'язково потрібно отримати **id** доданого об'єкта, то можна використовувати метод **db.QueryRow()**, який виконує запит і повертає певний об'єкт, наприклад:

```
1   var id int
2   db.QueryRow("insert into Products (model, company,
    price) values ('Mate 10 Pro', $1, $2) returning id",
3       "Huawei", 35000).Scan(&id)
4   fmt.Println(id)
```

У сам **sql-вираз** вводиться підвираз "**returning id**". А за допомогою метода **Scan()** отримане значення зчитується у змінну **id**.

10.3.3. Отримання даних

Для отримання даних застосовується метод **db.Query()**, який повертає набір рядків, або **db.QueryRow()**, який повертає один рядок, наприклад:

```
1   package main
2   import (
```

```
3       "database/sql"
4       "fmt"
5       _ "github.com/lib/pq"
6   )
7
8   type product struct{
9       id int
10      model string
11      company string
12      price int
13  }
14  func main() {
15
16      connStr := "user=postgres password=mypass
    dbname=productdb sslmode=disable"
17      db, err := sql.Open("postgres", connStr)
18      if err != nil {
19          panic(err)
20      }
21      defer db.Close()
22
23      rows, err := db.Query("select * from Products")
24      if err != nil {
25          panic(err)
26      }
27      defer rows.Close()
28      products := []product{}
29
30      for rows.Next(){
31          p := product{}
32          err := rows.Scan(&p.id, &p.model, &p.company,
    &p.price)
33          if err != nil{
34              fmt.Println(err)
35              continue
36          }
37          products = append(products, p)
38      }
```

```
39      for _, p := range products{
40          fmt.Println(p.id, p.model, p.company,
    p.price)
41      }
42  }
```

Для роботи з даними в цьому прикладі визначена структура **product**, яка відповідає даним в таблиці **Products**.

Для отримання даних викликається метод **Query()**, а саме:

```
1   rows, err := db.Query("select * from Products")
```

Цей метод як параметр приймає **sql-вираз SELECT** для отримання всіх даних з таблиці **Products**. Результат вибірки потрапляє у змінну **rows**, що представляє покажчик на структуру **Rows**. І за допомогою метода **rows.Next()** можна послідовно перебрати всі рядки в отриманому наборі, наприклад:

```
1   for rows.Next(){
2       p := product{}
3       err := rows.Scan(&p.id, &p.model, &p.company,
    &p.price)
4       if err != nil{
5           fmt.Println(err)
6           continue
7       }
8       products = append(products, p)
9   }
```

Тип **Rows** визначає метод **Scan**, за допомогою якого можна зчитати всі отримані дані у змінні. Наведеному прикладі зчитуються дані у структуру **Product** і потім додаються у зріз. Оскільки отримуються всі дані, тобто всі чотири стовпці, то відповідно в **Scan** передаються адреси чотирьох змінних.

Після зчитування даних у зріз можемо робити з ними все, що завгодно, наприклад, вивести на консоль, як показано на рисунку 10.5.

У методі **Query** можна вказувати додаткові параметри. Наприклад, отримання товарів, у яких ціна більша за 70000, виглядатиме наступним чином:

```
1   rows, err := db.Query("select * from Products where
    price > $1", 70000)
```

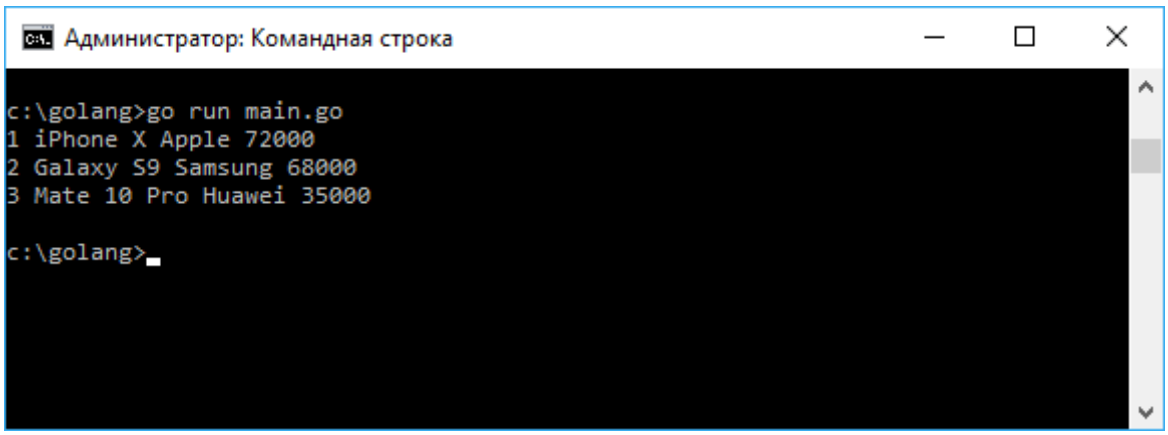

Рисунок 10.5. Результат роботи програми

Якщо потрібно отримати лише один рядок, то можна використовувати метод **QueryRow()**, наприклад:

```
row := db.QueryRow("select * from Products where id = $1", 2)
prod := product{}
err = row.Scan(&prod.id, &prod.model, &prod.company, &prod.price)
if err != nil{
    panic(err)
}
fmt.Println(prod.id, prod.model, prod.company, prod.price)
```

10.3.4. Оновлення

Для оновлення даних застосовується метод **Exec**, наприклад:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
)
func main() {

    connStr := "user=postgres password=mypass dbname=productdb sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
```

```
12          panic(err)
13      }
14      defer db.Close()
15
16      // Оновлюємо рядок з id=1
17      result, err := db.Exec("update Products set price
    = $1 where id = $2", 69000, 1)
18      if err != nil{
19          panic(err)
20      }
21      fmt.Println(result.RowsAffected())  // Кількість
    задіяних рядків
22  }
```

10.3.5. Видалення

Видалення даних із застосуванням метода **Exec** виглядатиме наступним чином:

```
1   package main
2   import (
3       "database/sql"
4       "fmt"
5       _ "github.com/lib/pq"
6   )
7   func main() {
8
9       connStr := "user=postgres password=mypass
    dbname=productdb sslmode=disable"
10      db, err := sql.Open("postgres", connStr)
11      if err != nil {
12          panic(err)
13      }
14      defer db.Close()
15
16      // Видаляємо рядок з id=2
17      result, err := db.Exec("delete from Products where
    id = $1", 2)
18      if err != nil{
19          panic(err)
```

```
20      }
21      fmt.Println(result.RowsAffected())  // Кількість
    видалених рядків
22  }
```

10.4. Робота з SQLite

Для роботи з **SQLite** у мові **Go** знадобиться драйвер **go-sqlite3**, який можна знайти за посиланням **https://github.com/mattn/go-sqlite3/** [15]. Для використання драйвера спочатку необхідно встановити його, виконавши в командному рядку або терміналі наступну команду, як наведено на рисунку 10.6.

```
go get github.com/mattn/go-sqlite3
```

Рисунок 10.6 Зовнішній вигляд вікна введення команди

Нехай у теці з кодом програми на мові **Go** у буде створено базу даних **SQLite**, яка називається **store.db**, в якій буде розташована таблиця **products**, яка визначається наступним чином:

```
1   CREATE TABLE products(
2     id INTEGER PRIMARY KEY AUTOINCREMENT,
3     model TEXT,
4     company TEXT,
5     price INTEGER
6   );
```

Вікно **DB Browser for SQLite** наведено на рисунку 10.7.

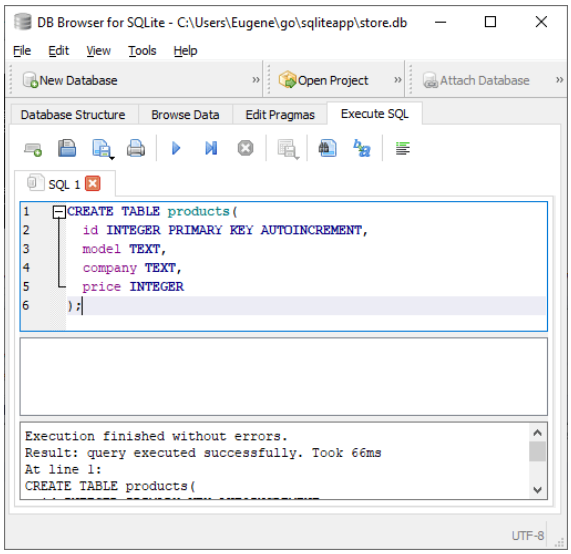

Рисунок 10.7. Вигляд головного вікна DB Browser for SQLite

10.4.1. Відкриття підключення

Для налаштування з'єднання з базою даних у функцію **sql.Open()** передається ім'я драйвера "**sqlite3**" і шлях до файлу бази даних, а саме:

```
db, err := sql.Open("sqlite3", "store.db")
```

В результаті налаштування з'єднання метод **db.Open** повинен повернути об'єкт ***DB**, через який можна буде взаємодіяти з базою даних.

10.4.2. Додавання даних

Для додавання даних використовується метод **Exec()** об'єкта **DB**, наприклад:

```
package main
import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
)
func main() {

    db, err := sql.Open("sqlite3", "store.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()
    result, err := db.Exec("insert into products (model, company, price) values ('iPhone X', $1, $2)",
        "Apple", 72000)
    if err != nil{
        panic(err)
    }
    fmt.Println(result.LastInsertId())  // id останнього доданого об'єкта
    fmt.Println(result.RowsAffected())  // Кількість доданих рядків

}
```

Через додаткові параметри метода **db.Exec()** можна передавати значення виконуваному **sql-виразу** через плейсхолдери **$1**, **$2** і так далі, замість яких вставлятимуться значення додаткових параметрів метода **db.Exec**.

У результату операції є метод **result.LastInsertId()**, який повертає id останнього доданого об'єкта, і метод **result.RowsAffected()**, який дозволяє отримати кількість доданих рядків, як показано на рисунку 10.8.

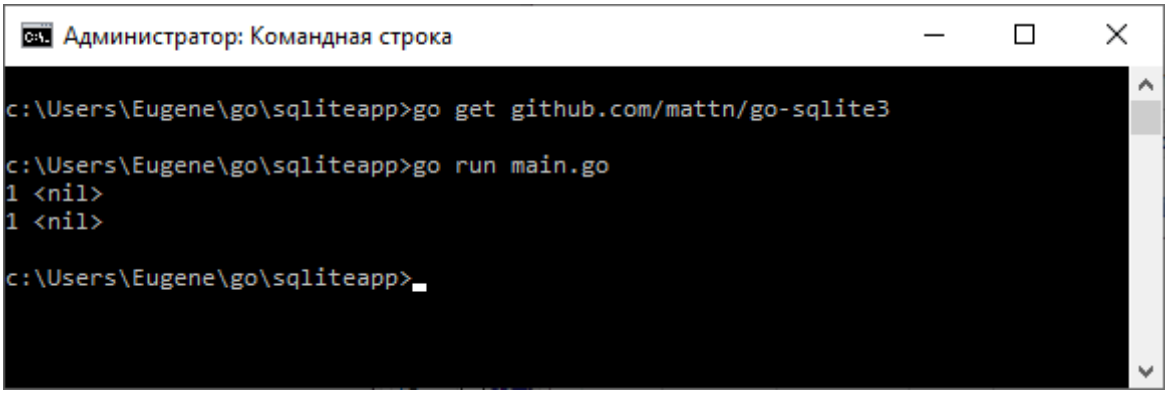

Рисунок 10.8. Результат роботи програми

10.4.3. Отримання даних

Для отримання даних застосовується метод **db.Query()**, який повертає набір рядків, або **db.QueryRow()**, який повертає один рядок, наприклад:

```
1   package main
2   import (
3       "database/sql"
4       "fmt"
5       _ "github.com/mattn/go-sqlite3"
6   )
7
8   type product struct{
9       id int
10      model string
11      company string
12      price int
13  }
14  func main() {
15
16      db, err := sql.Open("sqlite3", "store.db")
17      if err != nil {
```

```
18          panic(err)
19      }
20      defer db.Close()
21      rows, err := db.Query("select * from Products")
22      if err != nil {
23          panic(err)
24      }
25      defer rows.Close()
26      products := []product{}
27
28      for rows.Next(){
29          p := product{}
30          err := rows.Scan(&p.id, &p.model, &p.company,
    &p.price)
31          if err != nil{
32              fmt.Println(err)
33              continue
34          }
35          products = append(products, p)
36      }
37      for _, p := range products{
38          fmt.Println(p.id, p.model, p.company, p.price)
39      }
40  }
```

Для роботи з даними в наведеному прикладі визначена структура **product**, яка відповідає даним у таблиці **Products**.

Для отримання даних викликається метод **Query()**, а саме:

```
1   rows, err := db.Query("select * from Products")
```

Цей метод як параметр приймає **sql-вираз SELECT** для отримання всіх даних з таблиці **Products**. Результат вибірки потрапляє у змінну **rows**, що представляє покажчик на структуру **Rows**. І за допомогою метода **rows.Next()** можна послідовно перебрати всі рядки в отриманому наборі, наприклад:

```
1   for rows.Next(){
2       p := product{}
3       err := rows.Scan(&p.id, &p.model, &p.company,
    &p.price)
4       if err != nil{
5           fmt.Println(err)
```

```
            continue
        }
        products = append(products, p)
    }
```

В наведеному прикладі тип **Rows** визначає метод **Scan**, за допомогою якого можна зчитати всі отримані дані у змінні. У прикладі зчитуються дані до структури **Product** і потім додаються у зріз. Оскільки отримуються всі дані, тобто всі чотири стовпці, то відповідно до **Scan** передається адреси чотирьох змінних.

Після зчитування даних у зріз з ними можна робити все, що завгодно, наприклад, вивести на консоль, як показано на рисунку 10.9.

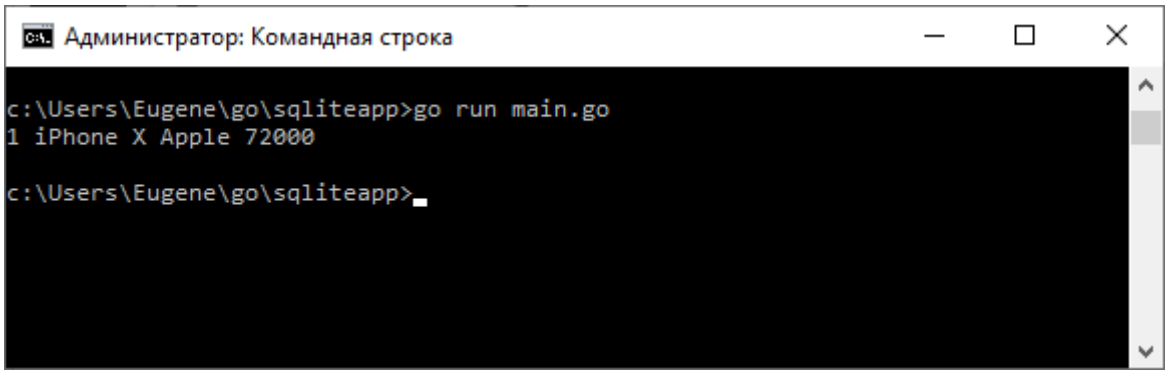

Рисунок 10.9. Результат роботи програми

У методі **Query** можна вказувати додаткові параметри. Приклад отримання товарів, у яких ціна більша за 70000, виглядатиме наступним чином:

```
rows, err := db.Query("select * from Products where price > $1", 70000)
```

Якщо потрібно отримати лише один рядок, то можна використати метод **QueryRow()**, а саме:

```
row := db.QueryRow("select * from Products where id = $1", 2)
prod := product{}
err = row.Scan(&prod.id, &prod.model, &prod.company, &prod.price)
if err != nil{
    panic(err)
}
fmt.Println(prod.id, prod.model, prod.company, prod.price)
```

10.4.4. Оновлення

Для оновлення даних можна застосовувати метод **Exec**, наприклад:

```
1  package main
2  import (
3      "database/sql"
4      "fmt"
5      _ "github.com/mattn/go-sqlite3"
6  )
7  func main() {
8
9      db, err := sql.Open("sqlite3", "store.db")
10     if err != nil {
11         panic(err)
12     }
13     defer db.Close()
14
15     // Оновлюємо рядок з id=1
16     result, err := db.Exec("update Products set price
   = $1 where id = $2", 69000, 1)
17     if err != nil{
18         panic(err)
19     }
20     fmt.Println(result.RowsAffected())  // Кількість
   оновлених рядків
21 }
```

10.4.5. Видалення

Для видалення також можна застосовувати метод **Exec**, наприклад:

```
1  package main
2  import (
3      "database/sql"
4      "fmt"
5      _ "github.com/mattn/go-sqlite3"
6  )
7  func main() {
```

```
8
9       db, err := sql.Open("sqlite3", "store.db")
10      if err != nil {
11          panic(err)
12      }
13      defer db.Close()
14
15      // Видаляємо рядок з id=1
16      result, err := db.Exec("delete from Products where
    id = $1", 1)
17      if err != nil{
18          panic(err)
19      }
20      fmt.Println(result.RowsAffected())  // Кількість
    видалених рядків
21  }
```

10.5. Робота з MongoDB

СКБД **MongoDB** не є реляційною, але це теж досить поширена система управління базами даних, яку можна використовувати в мові **Go**.

Для роботи з **MongoDB** знадобиться драйвер **mgo**, який можна знайти за посиланням **https://labix.org/mgo/** [16].

Спочатку необхідно встановити драйвер, виконавши в командному рядку/терміналі команду, як показано на рисунку 10.10.

```
go get gopkg.in/mgo.v2
```

Рисунок 10.10. Зовнішній вигляд вікна введення команди

10.5.1. Підключення

Для підключення до сервера **MongoDB** необхідно використовувати функцію **mgo.Dial()**, в яку передається адреса сервера, а саме:

```
1   func Dial(url string) (*Session, error)
```

Приклад підключення до сервера на локальному комп'ютері виглядатиме наступним чином:

```
1   session, err := mgo.Dial("mongodb://127.0.0.1")
```

В наведеному прикладі функція повертає покажчик на об'єкт **Session**, який представляє поточну сесію.

Використовуючи метод DB цього об'єкта, можна отримати покажчик на об'єкт **Database**, який представляє конкретну базу даних на сервері, наприклад:

```
func (s *Session) DB(name string) *Database
```

Усі дані в базах даних **MongoDB** структуровані за колекціями. Фактично колекція є аналогом таблиці у реляційних базах даних, яка представлена типом **Collection**. І щоб звернутися до потрібної колекції, необхідно використовувати метод **C()**, а саме:

```
func (db *Database) C(name string) *Collection
```

Приклад отримання колекції "**products**", яка розташована в базі даних "**productdb**" наступний:

```
// Відкриваємо з'єднання
session, err := mgo.Dial("mongodb://127.0.0.1")
if err != nil {
    panic(err)
}
defer session.Close()

// Отримуємо колекцію products у базі даних productdb
productCollection :=
session.DB("productdb").C("products")
```

Після отримання колекції, стає можливим додавання та отримання даних, а також проводити з ними інші операції. Після завершення роботи із сервером обов’язково необхідно закрити підключення за допомогою метода **Close()**.

10.5.2. Додавання даних

Для додавання даних до колекції застосовується метод **Insert()** об'єкта **Collection**, а саме:

```
func (c *Collection) Insert(docs ...interface{})
error
```

Цей метод приймає невизначену кількість об'єктів, що додаються до колекції, наприклад:

```
package main
import (
    "fmt"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)
```

```
7  type Product struct{
8      Id bson.ObjectId `bson:"_id"`
9      Model string `bson:"model"`
10     Company string `bson:"company"`
11     Price int `bson:"price"`
12 }
13 func main() {
14
15     // Відкриваємо з'єднання
16     session, err := mgo.Dial("mongodb://127.0.0.1")
17     if err != nil {
18         panic(err)
19     }
20     defer session.Close()
21
22     // Отримуємо колекцію
23     productCollection :=
   session.DB("productdb").C("products")
24
25     p1 := &Product{Id:bson.NewObjectId(),
   Model:"iPhone 8", Company:"Apple", Price:64567}
26     // Додаємо один об'єкт
27     err = productCollection.Insert(p1)
28     if err != nil{
29         fmt.Println(err)
30     }
31
32     p2 := &Product{Id:bson.NewObjectId(), Model:"Pixel
   2", Company:"Google", Price:58000}
33     p3 := &Product{Id:bson.NewObjectId(),
   Model:"Xplay7", Company:"Vivo", Price:49560}
34     // Додаємо два об'єкти
35     err = productCollection.Insert(p2, p3)
36     if err != nil{
37         fmt.Println(err)
38     }
39 }
```

В наведеному прикладі, насамперед спочатку імпортуються два пакети драйвера, які містять весь необхідний нам функціонал, а саме:

```
1   "gopkg.in/mgo.v2"
2   "gopkg.in/mgo.v2/bson"
```

Для представлення даних визначено структуру **Product**. Визначення кожної змінної структури, крім назви та типу даних, містить назву поля в колекції, з якою змінна буде зіставлятися, а саме,

```
1   Model string `bson:"model"`
```

Змінна **Model** зіставлятиметься із полем "**model**" у колекції. Причому між назвою змінної та полем колекції необов'язково має бути відповідність.

Також варто зазначити, що ідентифікатор об'єкта **MongoDB** представляє спеціальний тип **bson.ObjectId**, а в базі даних йому відповідає поле "**_id**".

Для додавання створюються три об'єкти, фактично три покажчики на об'єкти **Product**. Для створення унікального ідентифікатора використовується функція **bson.NewObjectId()**. Потім додаємо об'єкти до колекції, а саме:

```
1   err = productCollection.Insert(p1)
2   err = productCollection.Insert(p2, p3)
```

10.5.3. Отримання даних

Для отримання даних із колекції застосовується метод **Find()**, а саме:

```
1   func (c *Collection) Find(query interface{}) *Query
```

Як параметр він приймає критерій вибірки та повертає об'єкт ***Query**. Серед методів цього об'єкта слід виділити методи **All()** та **One**, які повертають відповідно всі об'єкти вибірки та дин об'єкт із вибірки, а саме:

```
1   func (q *Query) All(result interface{}) error
2   func (q *Query) One(result interface{}) (err error)
```

В якості параметра обидва методи приймають покажчик на об'єкт, який зберігатиметься результат вибірки.

Приклад отримання раніше збережених об'єктів наступний:

```
1   package main
2   import (
3       "fmt"
4       "gopkg.in/mgo.v2"
5       "gopkg.in/mgo.v2/bson"
6   )
7   type Product struct{
8       Id bson.ObjectId `bson:"_id"`
```

```
9       Model string `bson:"model"`
10      Company string `bson:"company"`
11      Price int `bson:"price"`
12  }
13  func main() {
14
15      // Відкриваємо з'єднання
16      session, err := mgo.Dial("mongodb://127.0.0.1")
17      if err != nil {
18          panic(err)
19      }
20      defer session.Close()
21
22      // Отримуємо колекцію
23      productCollection :=
    session.DB("productdb").C("products")
24      // Критерій вибірки
25      query := bson.M{}
26      // Об'єкт для збереження результату
27      products := []Product{}
28      productCollection.Find(query).All(&products)
29
30      for _, p := range products{
31
32          fmt.Println(p.Model, p.Company, p.Price)
33      }
34  }
```

Критерієм вибірки є об'єкт **bson.M{}**. Порожній об'єкт **bson.M{}** охоплює всі документи в колекції. Усі отримані документи передаються в об'єкт **products**. І потім дані виводяться на консоль, як показано на рисунку 10.11.

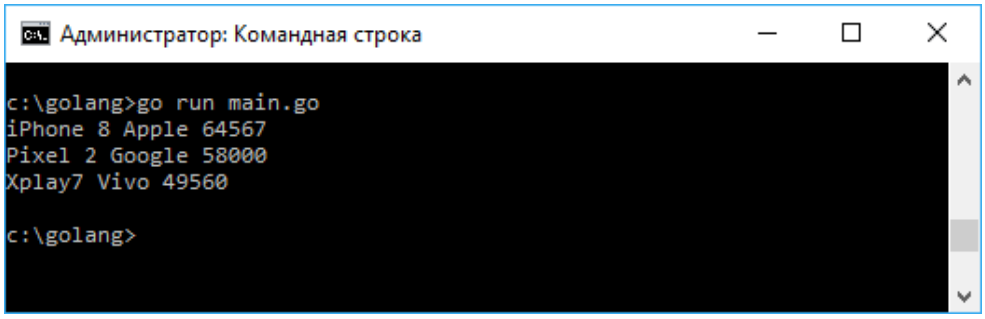

Рисунок 10.11. Результат роботи програми

Також можна встановлювати критерії вибірки, наприклад:

```
query := bson.M{
    "price" : bson.M{
        "$gt":50000,
    },
}
products := []Product{}
productCollection.Find(query).All(&products)

for _, p := range products{

    fmt.Println(p.Model, p.Company, p.Price)
}
```

В наведеному прикладі виконується пошук всіх документів, у яких поле "**price**" має значення більше за 50000.

10.5.4. Оновлення даних

Для оновлення даних застосовуються методи **Update()** або **UpdateAll()** об'єкта **Collection**, а саме:

```
func (c *Collection) Update(selector interface{},
update interface{}) error
func (c *Collection) UpdateAll(selector interface{},
update interface{}) (info *ChangeInfo, err error)
```

Першим параметром методів вибірки є критерій вибірки документів, які будуть оновлюватися. Другий параметр вказує, як ці документи оновлюватимуться. Обидва параметри задаються об'єктом **bson.M**. Однак, якщо метод **Update** оновлює лише один документ, який відповідає першому параметру, то метод **UpdateAll** оновлює всі елементи.

Приклад зміни ціну смартфона "**iPhone 8**" виглядатиме наступним чином:

```
package main
import (
    "fmt"
    "gopkg.in/mgo.v2"
    "gopkg.in/mgo.v2/bson"
)
type Product struct{
    Id bson.ObjectId `bson:"_id"`
```

```
9       Model string `bson:"model"`
10      Company string `bson:"company"`
11      Price int `bson:"price"`
12  }
13  func main() {
14
15      // Відкриваємо з'єднання
16      session, err := mgo.Dial("mongodb://127.0.0.1")
17      if err != nil {
18          panic(err)
19      }
20      defer session.Close()
21
22      // Отримуємо колекцію
23      productCollection :=
    session.DB("productdb").C("products")
24
25      // Оновлюємо дані
26      err = productCollection.Update(bson.M{"model":
    "iPhone 8"}, bson.M{"$set":bson.M{"price":45000}})
27      if err != nil{
28          fmt.Println(err)
29      }
30
31      products := []Product{}
32      productCollection.Find(bson.M{}).All(&products)
33
34      for _, p := range products{
35
36          fmt.Println(p.Model, p.Company, p.Price)
37      }
38  }
```

В наведеному прикладі перший аргумент метода **Update**, а саме **bson.M{"model": "iPhone 8"}** вказує, що вибираються всі елементи, у яких поле "**model**" дорівнює "**iPhone 8**". Другий аргумент **bson.M{"$set":bson.M{"price":45000}}** за допомогою параметра **$set** встановлює, які значення будуть мати ті чи інші поля (у наведеному прикладі це поле "price").

10.5.5. Видалення документів

Для видалення документів з колекції застосовується метод **Remove()** або **RemoveAll()** об'єкта **Collection**, а саме:

```
1   func (c *Collection) Remove(selector interface{})
    error
2   func (c *Collection) RemoveAll(selector interface{})
    (info *ChangeInfo, err error)
```

Обидва методи як параметр приймають критерій вибірки документів, які видалятимуться. При цьому метод **Remove** видаляє лише один документ із вибірки, а метод **RemoveAll** видаляє всі елементи вибірки.

Приклад видалення всіх смартфонів компанії **Vivo** виглядатиме наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "gopkg.in/mgo.v2"
5       "gopkg.in/mgo.v2/bson"
6   )
7   type Product struct{
8       Id bson.ObjectId `bson:"_id"`
9       Model string `bson:"model"`
10      Company string `bson:"company"`
11      Price int `bson:"price"`
12  }
13  func main() {
14
15      // Відкриваємо з'єднання
16      session, err := mgo.Dial("mongodb://127.0.0.1")
17      if err != nil {
18          panic(err)
19      }
20      defer session.Close()
21
22      // Отримуємо колекцію
23      productCollection :=
    session.DB("productdb").C("products")
24
```

```
25      // Видаляємо всі документи з company = "Vivo"
26      _, err =
    productCollection.RemoveAll(bson.M{"company":
    "Vivo"})
27      if err != nil{
28          fmt.Println(err)
29      }
30
31      products := []Product{}
32      productCollection.Find(bson.M{}).All(&products)
33
34      for _, p := range products{
35
36          fmt.Println(p.Model, p.Company, p.Price)
37      }
38  }
```

Резюме

У наведеному розділі розглянуто основи роботи з базами даних у мові програмування Go. Пояснюється використання стандартного пакета **database/sql**, який надає універсальний інтерфейс для взаємодії з різними СКБД через драйвери. Докладно описано роботу з MySQL, PostgreSQL, SQLite, та MongoDB, включаючи підключення, додавання, отримання, оновлення та видалення даних. Наведено приклади використання методів, таких як **Exec**, **Query**, **QueryRow**, а також механізм транзакцій і роботи зі структурами для зберігання даних. Особлива увага приділена особливостям драйверів кожної СКБД, наприклад, плейсхолдерам у запитах або методам для роботи з ідентифікаторами. Загалом, матеріал акцентує увагу на універсальності підходу до роботи з базами даних у Go.

Наступні розділи об’єднають всі отримані знання та навички з мережевого програмування, інтеграції вебзастосунків з базами даних та створення динамічних вебзастосунків на прикладі повноцінного вебзастосунку.

Контрольні запитання і завдання

1. Який пакет використовується в Go для роботи з базами даних?
2. Яку функцію використовують для відкриття з'єднання з базою даних?
3. Що повертає функція **sql.Open** у разі успішного підключення?

4. Для чого використовується метод **Exec** у **database/sql**?
5. Яку роль відіграють плейсхолдери у SQL-запитах?
6. Як отримати кількість рядків, задіяних у запиті, використовуючи метод **Exec**?
7. Який метод використовують для виконання запиту, що повертає набір рядків?
8. Як закрити з'єднання з базою даних у Go?
9. Який метод використовується для отримання одного рядка з бази даних?
10. Як реалізувати транзакції в Go?
11. Які особливості драйвера **github.com/go-sql-driver/mysql**?
12. Як створити таблицю в MySQL для роботи з Go?
13. Чому перед ім'ям пакета-драйвера вказується « _ », наприклад, **_ "github.com/go-sql-driver/mysql"**?
14. Як у Go працювати з базою даних PostgreSQL через **github.com/lib/pq**?
15. Як виглядає рядок підключення до PostgreSQL у Go?
16. Як працювати з базою даних SQLite у Go?
17. Які методи драйвера SQLite дозволяють отримувати та додавати дані?
18. Як у Go реалізується робота з MongoDB через **mgo**?
19. Який тип даних використовується в MongoDB для ідентифікаторів документів?
20. Як виконати оновлення даних у колекції MongoDB, використовуючи **mgo**?
21. Напишіть програму на Go, яка підключається до бази даних MySQL, створює таблицю **users** з полями **id**, **name**, **email**, а потім додає один запис у таблицю.
22. Реалізуйте функцію на Go, яка виконує запит до бази даних PostgreSQL, отримує всі записи з таблиці **products** і виводить їх у консоль.
23. Створіть програму, яка підключається до SQLite, створює таблицю **books** і додає до неї кілька записів. Потім реалізуйте функцію, яка шукає книги за автором.
24. Реалізуйте програму на Go для MongoDB, яка додає кілька записів у колекцію **orders**, а потім знаходить усі замовлення з ціною вище 5000.

25. Напишіть програму на Go, яка використовує транзакції для додавання запису в таблицю **accounts** у базі даних MySQL. Якщо додавання не вдалося, транзакція має бути відхилена.
26. Реалізуйте програму, яка підключається до PostgreSQL, оновлює ціну товару в таблиці **products** за вказаним **id**, а потім виводить оновлену інформацію.
27. Напишіть програму на Go, яка отримує дані з таблиці **employees** в SQLite і записує їх у файл у форматі CSV.
28. Створіть програму на Go, яка видаляє всі записи зі значенням **status =** "**inactive**" у таблиці users в MongoDB.
29. Напишіть функцію, яка підраховує кількість записів у таблиці orders у базі даних MySQL, де сума замовлення перевищує 10000.
30. Реалізуйте програму на Go, яка підключається до PostgreSQL, створює таблицю **students**, додає записи та реалізує функцію пошуку студентів за віком.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Який пакет використовується для роботи з базами даних у Go?

a) fmt
b) database/sql
c) os
d) io

Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які методи використовуються для взаємодії з базою даних через об'єкт *DB?

a) Exec
b) Query
c) Scan
d) QueryRow

Правильна відповідь: a), b), d).

Завдання на відповідність

Співвіднесіть методи з їх призначенням:

a) Exec → 1. Використовується для виконання запиту, який не повертає результати.

b) Query → 2. Використовується для виконання запиту, який повертає набір рядків.

c) QueryRow → 3. Використовується для отримання одного рядка.

d) Close → 4. Закриває підключення до бази даних.

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

d) → 4.

Вставити пропущені слова

Функція для відкриття з'єднання з базою даних називається _____.

Правильна відповідь: …sql.Open…

Завдання з вибором істинності

Метод Rows.Next() повертає true, якщо є ще рядки у результаті запиту.

a) Так

b) Ні

Правильна відповідь: a).

Завдання з кодом

Що буде некоректним у MongoDB-запиті?

a) bson.M{"price": bson.M{"$gt": 5000}}

b) bson.M{"model": "iPhone 8"}

c) bson.M{"company": "Google"}

d) bson.M{"price": {"gt": 5000}}

Правильна відповідь: d).

11. ВЕБПРОГРАМУВАННЯ В GO

Особливу сферу розробки на мові Go займає вебпрограмування, яке передбачає створення різних вебзастосунків, у тому числі різних вебсайтів та вебсервісів. Спочатку мова Go не призначалась для вебпрограмування і навіть не розглядалась розробниками у цій ролі, і знадобився деякий час, щоб мова Go стала застосовуватися у цій галузі. У той же час, це не означає, що мова Go підійде для всіх вебпроєктів. Багато вебфреймворків містять багато готового функціоналу з коробки, який полегшує створення програм. У мові Go такого немає. Мова Go надає в основному лише базові речі, на підставі яких можна будувати складніші конструкції. Мова Go в плані веброзробки підійде насамперед для таких проєктів, які вимагають дуже високої продуктивності, або коли важливі певні можливості мови Go, наприклад, паралельна обробка запитів.

11.1. Простий вебзастосунок

Основний функціонал для створення вебзастосунків в мові Go розташований у пакеті **net/http**. Зокрема, щоб запустити вебзастосунок, який міг би приймати вхідні запити, достатньо викликати функцію **http.ListenAndServe**, формальний опис якої наступний:

```
1   func ListenAndServe(addr string, handler Handler)
    error
```

Перший параметр вказує, запити за яким шляхом будуть обслуговуватися вебзастосунком. Другий параметр визначає обробник запиту у вигляді інтерфейсу **Handler**. Цей інтерфейс визначає функцію **ServeHTTP**, а саме:

```
1   type Handler interface {
2       ServeHTTP(ResponseWriter, *Request)
3   }
```

Функція **ServeHTTP** приймає два параметри. Першим параметром є об'єкт **ResponseWriter**, який є потоком відповіді, у якому можна записати будь-які дані, які необхідно відправити у відповідь користувачу. Другим параметром є **Request**, який інкапсулює всю інформацію про запит.

Наприклад, визначимо найпростіший вебзастосунок. Для цього створимо на жорсткому диску теку для зберігання файлів з вихідним кодом на мові Go і назвемо його **golang**. У цій же теці визначимо файл **server.go** з наступним кодом:

```
1   package main
```

```
2   import (
3       "fmt"
4       "net/http"
5   )
6   type msg string
7   func (m msg) ServeHTTP(resp http.ResponseWriter, req
    *http.Request) {
8      fmt.Fprint(resp, m)
9   }
10  func main() {
11      msgHandler := msg("Hello from Web Server in Go")
12      fmt.Println("Server is listening...")
13      http.ListenAndServe("localhost:8181",
    msgHandler)
14  }
```

В наведеному прикладі визначено додатковий власний тип **msg** на основі типу **string**, який реалізує метод **ServeHTTP** інтерфейсу **Handler**. У самому методі за допомогою виклику **fmt.Fprint(resp, m)** у потік відповіді **resp** записується повідомлення, яке зберігається у рядку **m**. Таким чином, користувачеві надсилається відповідь.

У функції **main** визначається об'єкт **msgHandler**, наступним чином:

```
1   msgHandler := msg("Hello from Web Server in Go")
```

На вигляд це рядок, проте цей об'єкт реалізує інтерфейс **Handler**.

Далі для обробки запитів цей об'єкт як другий параметр передається у функцію **http.ListenAndServe**, а саме:

```
1   http.ListenAndServe("localhost:8181", msgHandler)
```

Перший параметр вказує, що вебзастосунок запускатиметься за адресою **localhost:8181**. Номер порту необов'язково має бути **8181**. Це може бути будь-який незайнятий порт. Запустимо вебзастосунок, як показано на рисунку 11.1.

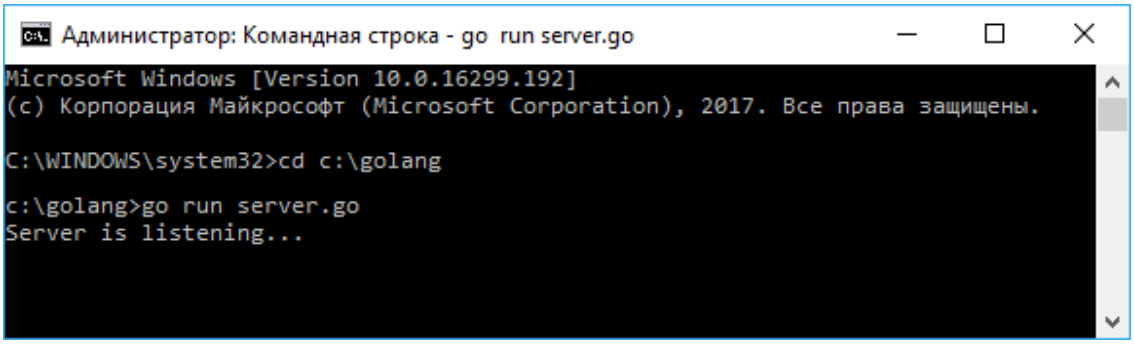

Рисунок 11.1. Запуск вебзастосунку на виконання

І потім при зверненні за адресою **http://localhost:8181/** у будь-якому браузері отримаємо результат, як показано на рисунку 11.2.

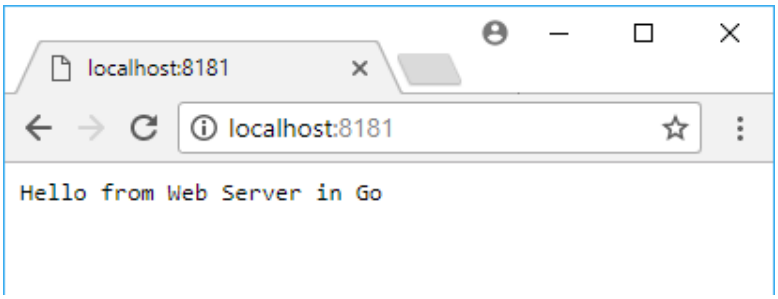

Рисунок 11.2. Результат роботи програми

11.2. Маршрутизація

11.2.1. Функція HandleFunc

Система маршрутизації у мові Go дозволяє співставити певні запити з певними ресурсами всередині вебзастосунків. Для створення найпростішої системи маршрутизації в застосунку може застосовуватись функція **HandleFunc()**, формальний опис якої наступний:

```
1   func HandleFunc(pattern string, handler
    func(ResponseWriter, *Request))
```

Її перевага полягає в тому, що вона дає змогу вказати маршрути для обробки. Першим параметром функції є маршрут, який оброблятиметься даною функцією. А другим параметром є функція **handler**, яка оброблятиме запит. Вона також приймає два параметри, а саме: **ResponseWriter** - потік відповіді; ***Request** - інформація про запит.

Розглянемо наступний код у файлі сервера:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5   )
6
7   func main() {
8
9       http.HandleFunc("/about", func(w
    http.ResponseWriter, r *http.Request){
10          fmt.Fprint(w, "About Page")
11      })
```

```
12      http.HandleFunc("/contact", func(w
    http.ResponseWriter, r *http.Request){
13          fmt.Fprint(w, "Contact Page")
14      })
15      http.HandleFunc("/", func(w http.ResponseWriter,
    r *http.Request){
16          fmt.Fprint(w, "Index Page")
17      })
18      fmt.Println("Server is listening...")
19      http.ListenAndServe("localhost:8181", nil)
20  }
```

Перший аргумент функції **HandleFunc**, а саме "**/about**", вказує, що ця функція оброблятиме запити зі шляхом "**/about**", тобто за адресою **http://localhost:8181/about**. Другий параметр вказує, що у відповідь на запит за цим шляхом користувачеві буде надсилатися рядок "**About Page**", як показано на рисунку 11.3.

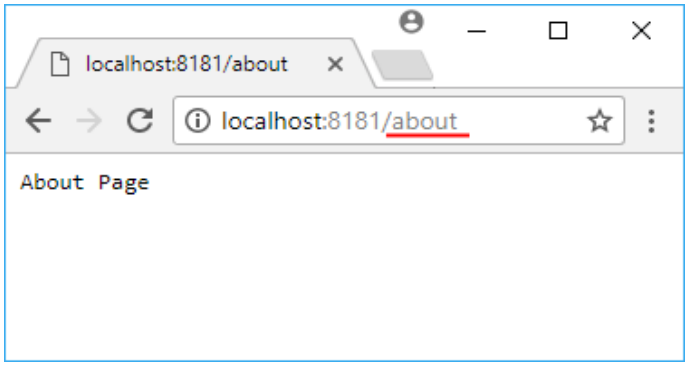

Рисунок 11.3. Результат роботи програми

Відповідно запит зі шляхом "**/contact**" буде оброблятися функцією **http.HandleFunc("/contact",..**, а запит до кореня вебсайту буде оброблятися функцією **http.HandleFunc("/",..**, як показано на рисунку 11.4.

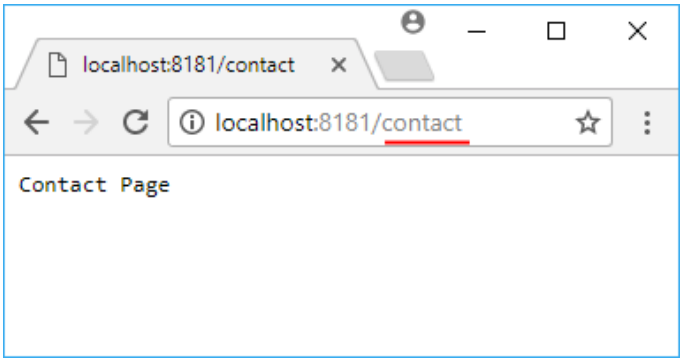

Рисунок 11.4. Результат роботи програми

Подібним чином можна зіставляти маршрути не тільки з функціями, які повертають деякий вміст у вигляді рядка, але також можна зіставляти маршрути зі статичними файлами. Наприклад, визначимо в теці з файлом сервера html-сторінку **hello.html** з наступним кодом:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Index</title>
    </head>
    <body>
        <h1>Index</h1>
    </body>
</html>
```

Додатково необхідно змінити файл сервера наступним чином:

```
package main
import (
    "fmt"
    "net/http"
)

func main() {

    http.HandleFunc("/hello", func(w
http.ResponseWriter, r *http.Request){
        http.ServeFile(w, r, "hello.html")
    })
    http.HandleFunc("/about", func(w
http.ResponseWriter, r *http.Request){
        fmt.Fprint(w, "About Page")
    })
    http.HandleFunc("/", func(w http.ResponseWriter,
r *http.Request){
        fmt.Fprint(w, "Index Page")
    })
    fmt.Println("Server is listening...")
    http.ListenAndServe(":8181", nil)
}
```

Після цього за допомогою функції **http.ServeFile()** при запиті зі шляхом "**/hello**" буде відправлятися файл **hello.html**, як показано на рисунку 11.5.

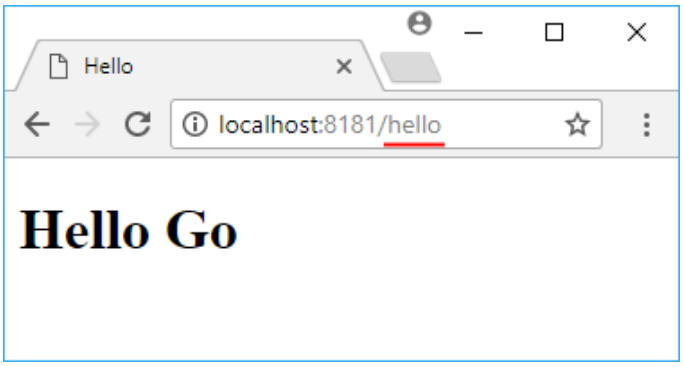

Рисунок 11.5. Результат роботи програми

11.2.2. Функція Handle

Існує ще один спосіб визначення маршрутів та зіставлення їх з обробниками за допомогою функції **http.Handle**, формальний опис якої наступний:

```
1   func Handle(pattern string, handler Handler)
```

У якості прикладу визначимо у файлі сервера наступний код:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5   )
6
7   type httpHandler struct{
8       message string
9   }
10  func (h httpHandler) ServeHTTP(resp
    http.ResponseWriter, req *http.Request) {
11      fmt.Fprint(resp, h.message)
12   }
13
14  func main() {
15
16      h1 := httpHandler{ message:"Index"}
17      h2 := httpHandler{ message:"About"}
18
```

```
19      http.Handle("/", h1)
20      http.Handle("/about", h2)
21
22      fmt.Println("Server is listening...")
23      http.ListenAndServe(":8181", nil)
24  }
```

В наведеному прикладі в ролі інтерфейсу **Handler**, який обробляє запит, виступає структура **httpHandler**, а результат роботи програми наведено на рисунку 11.6.

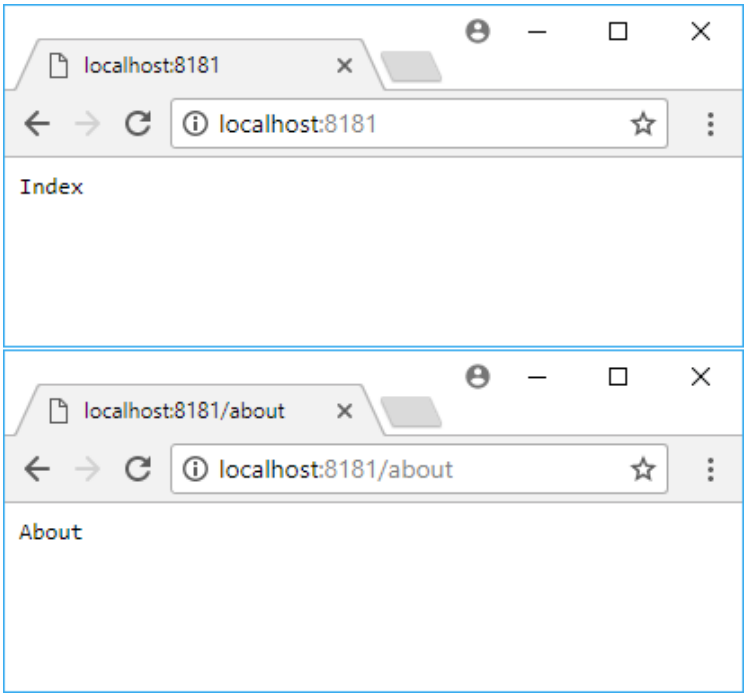

Рисунок 11.6. Результат роботи програми

11.3. Статичні файли

11.3.1. Функція http.FileServer

Вміст вебзастосунку або вебсайту нерідко визначається у вигляді статичних html-сторінок. Їм не потрібна якась додаткова обробка на стороні сервера. Для прямого надсилання статичних файлів у пакеті **http** [9] визначено функцію **FileServer**, яка повертає об'єкт **Handler**, формальний опис якої має наступний вигляд:

```
1   func FileServer(root FileSystem) Handler
```

У якості параметра функція **FileServer** приймає шлях до теки зі статичними файлами.

У якості прикладу визначимо в теці з вихідним файлом Go теку, яку назвемо **static**. Створимо в ній два статичні файли: **index.html** та **about.html**, як показано на рисунку 11.7.

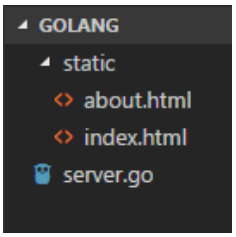

Рисунок 11.7. Вікно менеджера тек і файлів

Напишемо у файлі **index.html** наступний код:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Index</title>
    </head>
    <body>
        <h1>Index</h1>
    </body>
</html>
```

А у файлі **about.html** наступний код:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>About</title>
    </head>
    <body>
        <h1>About</h1>
    </body>
</html>
```

У головному файлі **server.go** визначимо наступний код:

```
package main
import (
    "fmt"
    "net/http"
)
```

```
6   func main() {
7       fmt.Println("Server is listening...")
8       http.ListenAndServe(":8181",
    http.FileServer(http.Dir("static")))
9   }
```

В наведеному прикладі у функцію **http.FileServer()** передається шлях до файлів, що визначається функцією **http.Dir()**.

Після запуску програми на виконання звернемося за адресою **http://localhost:8181** і отримаємо результат, який наведено на рисунку 11.8.

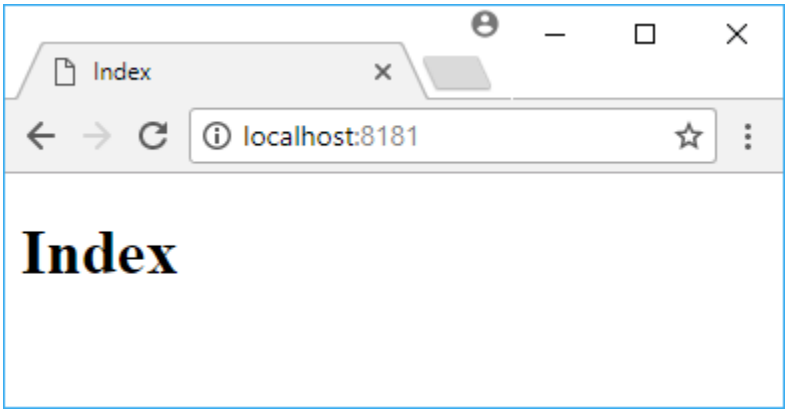

Рисунок 11.8. Результат роботи програми

Варто звернути увагу на те, що шлях до кореня вебсайту автоматично зіставляється з файлом **index.html.** Це рівнозначно зверненню за адресою **http://localhost:8181/index.html**.

Додатково звернемося за адресою **http://localhost:8181/about.html**. У цьому випадку буде отримано вміст файлу **about.html**, як показано на рисунку 11.9.

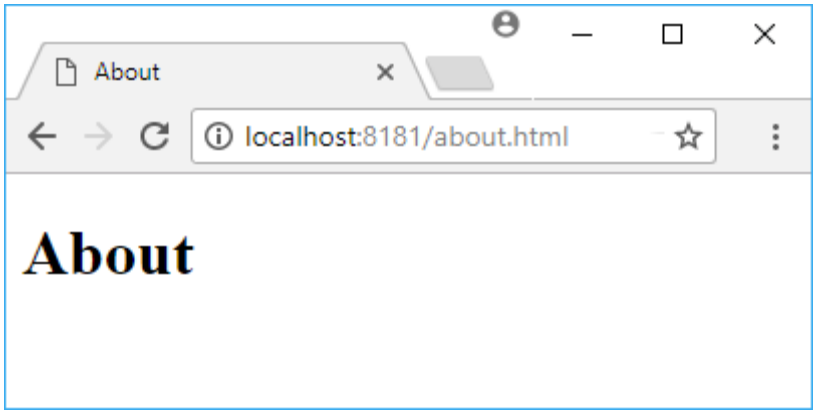

Рисунок 11.9. Результат роботи програми

У той самий час такий підхід досить обмежений, оскільки у цьому разі сервер віддає лише статичні файли. Однак, як правило, виникає необхідність

саме у генерації динамічного контенту. І в цьому випадку необхідно вчинити наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5   )
6
7   func main() {
8
9       fs := http.FileServer(http.Dir("static"))
10      http.Handle("/", fs)
11
12      http.HandleFunc("/about", func(w
    http.ResponseWriter, r *http.Request){
13          fmt.Fprint(w, "About Page")
14      })
15      http.HandleFunc("/contact", func(w
    http.ResponseWriter, r *http.Request){
16          fmt.Fprint(w, "Contact Page")
17      })
18      fmt.Println("Server is listening...")
19      http.ListenAndServe(":8181", nil)
20  }
```

В наведеному прикладі за допомогою функції **http.Handle("/", fs)** файловий сервер монтується по шляху "/", тобто до кореня сайту. І водночас стає можливим визначення обробники для інших маршрутів. Таким чином, буде забезпечено функціонал роботи як з динамічною генерацією контенту, так і зі статичними файлами, як показано на рисунках 11.10 та 11.11.

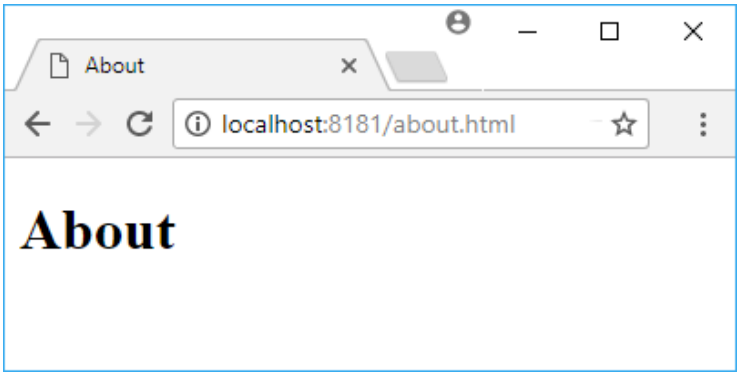

Рисунок 11.10. Результат роботи програми

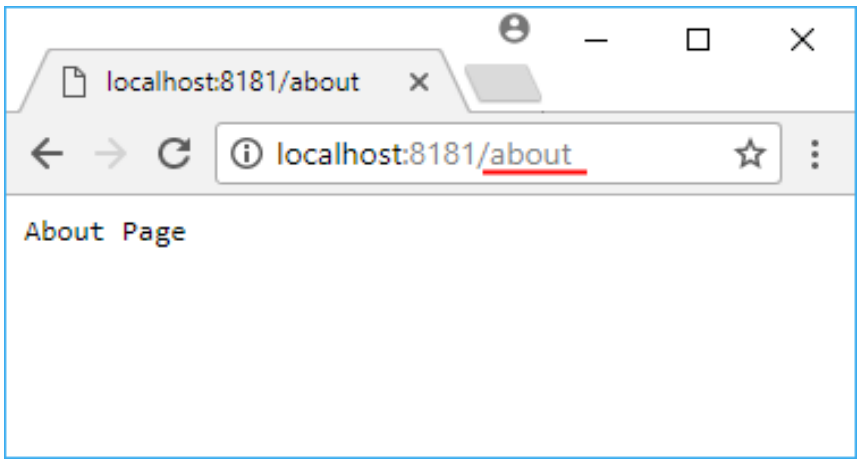

Рисунок 11.11. Результат роботи програми

11.3.2. Надсилання файлів за допомогою функції http.ServeFile

Для надсилання файлів також можна використовувати функцію **http.ServeFile()**. Вона відправляє одиничний файл за певним шляхом. Реалізувати це можна, наприклад, скориставшись раніше створеними файлами **index.html** та **about.html,** модифікацією коду головної програми наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5   )
6
7   func main() {
8
9       http.HandleFunc("/about", func(w
10  http.ResponseWriter, r *http.Request){
11          http.ServeFile(w, r, "static/about.html")
12      })
        http.HandleFunc("/", func(w http.ResponseWriter,
13  r *http.Request){
14          http.ServeFile(w, r, "static/index.html")
15      })
        fmt.Println("Server is listening...")
16      http.ListenAndServe(":8181", nil)
17  }
```

11.4. Маршрутизація та gorilla/mux

Мова Go надає базові можливості маршрутизації. Однак цих можливостей, як правило, було недостатньо, особливо в тих випадках, коли необхідно виділяти сегменти із запитаної URL-адреси і якимось чином обробляти їх. У цьому випадку можна скористатися низкою існуючих інструментів, одним з яких є **Gorilla**, офіційний сайт якої можна знайти за посиланням **http://www.gorillatoolkit.org/** [17]. Це пакет розробника призначено спеціально для спрощення створення вебзастосунків мовою Go, який, у свою чергу, включає наступну низку пакетів:

- **gorilla/context** : призначений для створення глобальних змінних із тіла запиту;
- **gorilla/rpc** : представляє реалізацію протоколу **RPC-JSON**;
- **gorilla/websocket** : реалізує протокол **WebSocket**;
- **gorilla/schema** : дозволяє створювати із значень форми єдину структуру;
- **gorilla/securecookie** : дозволяє створювати зашифровані кукі, які застосовуються під час автентифікації;
- **gorilla/session s**: забезпечує підтримку сесій;
- **gorilla/mux** : дозволяє визначати складні маршрути, які можуть використовувати регулярні вирази;
- **gorilla/reverse** : використовується для створення регулярних виразів для маршрутів.

В якості прикладу розглянемо можливості створення маршрутів за допомогою **gorilla/mux** і для встановлення даного пакета виконаємо в командному рядку або терміналі наступну команду, як наведено на рисунку 11.12.

```
go get github.com/gorilla/mux
```

Рисунок 11.12. Зовнішній вигляд вікна введення команди

Код сервера визначимо у наступний спосіб:

```
1  package main
2  import (
3      "fmt"
4      "net/http"
5      "github.com/gorilla/mux"
6  )
```

```
7
8   func productsHandler(w http.ResponseWriter, r
    *http.Request) {
9       vars := mux.Vars(r)
10      id := vars["id"]
11      response := fmt.Sprintf("Product %s", id)
12      fmt.Fprint(w, response)
13  }
14
15  func main() {
16
17      router := mux.NewRouter()
18      router.HandleFunc("/products/{id:[0-9]+}",
    productsHandler)
19      http.Handle("/",router)
20
21      fmt.Println("Server is listening...")
22      http.ListenAndServe(":8181", nil)
23  }
```

В наведеному прикладі для визначення маршрутів за допомогою **gorilla/mux** використовується функція **mux.NewRouter()**. У об'єкта, що повертається цією функцією, можна викликати метод **HandleFunc**, який зіставляє маршрут з певним обробником.

Першим параметром є шаблон шляху запиту. У фігурних дужках можна визначити параметр у наступному форматі:

```
1   Ім'я_параметра : регулярний_вираз
```

Регулярний вираз визначати необов'язково, але якщо він визначений, то параметр повинен відповідати цьому виразу. Тобто в даному випадку параметр **id** повинен представляти числове значення.

Другим параметром є функція оброблювача запитів за вказаним у першому параметрі маршрутом. Подібна функція повинна мати два параметри, а саме:

```
1   func (w http. Response Writer, r * http. Request)
```

В неведеному прикладі функція обробника **productshandler** отримує параметри шляху запиту через функцію **mux.vars**. Потім з отриманого об'єкта можна отримати назву потрібного нам параметра, а саме: **id:=vars["id"]**. Назва параметра тут те саме, що і у визначенні маршруту.

У результаті при зверненні до додатку за запитом **localhost:8181/products/2** буде отримано результат, який наведено на рисунку 11.13.

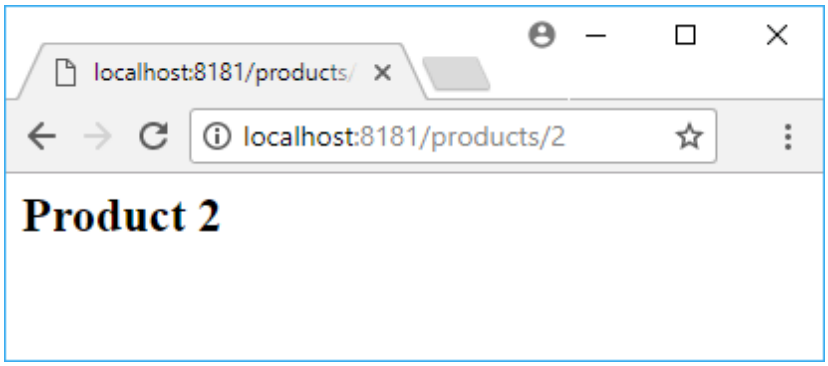

Рисунок 11.13. Результат роботи програми

Щоб зробити те ж саме, але штатними засобами, які є в мові Go без **gorilla/mux**, то довелося б писати додатковий код для парсингу запитаного шляху.

Подібним чином можна використовувати декілька параметрів, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "github.com/gorilla/mux"
6   )
7
8   func productsHandler(w http.ResponseWriter, r
    *http.Request) {
9       vars := mux.Vars(r)
10      id := vars["id"]
11      cat := vars["category"]
12      response := fmt.Sprintf("Product category=%s
    id=%s", cat, id)
13      fmt.Fprint(w, response)
14  }
15
16  func main() {
17
18      router := mux.NewRouter()
19      router.HandleFunc("/products/{category}/{id:[0-
    9]+}", productsHandler)
```

```
20      http.Handle("/",router)
21
22      fmt.Println("Server is listening...")
23      http.ListenAndServe(":8181", nil)
24  }
```

В наведеному прикладі визначено два параметри: **id** та **category**, як наведено на рисунку 11.14.

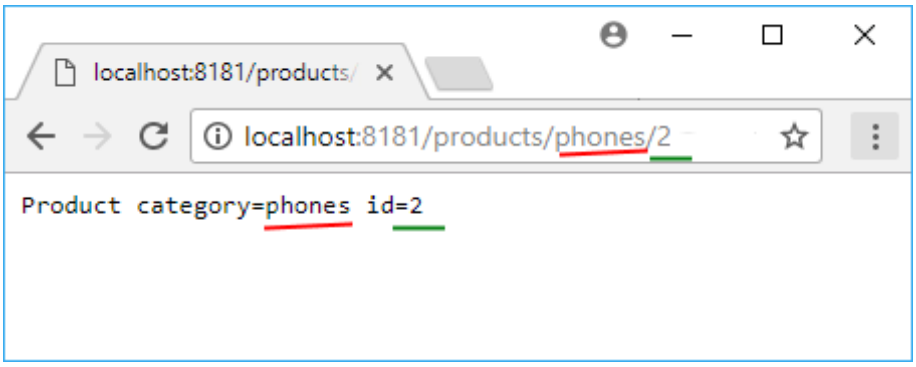

Рисунок 11.14. Результат роботи програми

Крім того, можна визначати кілька маршрутів, які можуть використовувати або різні, або одні й самі обробники, наприклад:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "github.com/gorilla/mux"
6   )
7
8   func productsHandler(w http.ResponseWriter, r
    *http.Request) {
9       vars := mux.Vars(r)
10      id := vars["id"]
11      response := fmt.Sprintf("id=%s", id)
12      fmt.Fprint(w, response)
13  }
14  func indexHandler(w http.ResponseWriter, r
    *http.Request) {
15      fmt.Fprint(w, "Index Page")
16  }
17
```

```
18 func main() {
19
20     router := mux.NewRouter()
21     router.HandleFunc("/products/{id:[0-9]+}",
   productsHandler)
22     router.HandleFunc("/articles/{id:[0-9]+}",
   productsHandler)
23     router.HandleFunc("/", indexHandler)
24     http.Handle("/",router)
25
26     fmt.Println("Server is listening...")
27     http.ListenAndServe(":8181", nil)
28 }
```

11.5. Рядок запиту та відправка форм

11.5.1. Рядок запиту

Рядком запиту є набір параметрів, які розташовуються у адресі після знаку запитання (**?**). При цьому кожен параметр визначає назву та значення, наприклад:

```
1   localhost:8181/user?name=Sam&age=21
```

В наведеному прикладі частина **?name=Sam&age=21** визначає рядок запиту, де є два параметри: **name** та **age**. Для кожного параметра визначено ім'я та значення, що відокремлюються знаком дорівнює (**=**). Параметр **name** має значення "**Sam**", а параметр **age** має значення **21**. Один від одного параметри відокремлюються знаком амперсанду (**&**).

Для отримання рядка запиту у об'єкта **Request** спочатку потрібно отримати запитану адресу через змінну **URL**. Далі у адресі викликається метод **Query()**, який і повертає рядок запиту.

Розглянемо приклад отримання даних з рядка запиту:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5   )
6
7   func main() {
```

```
8
9       http.HandleFunc("/user", func(w
    http.ResponseWriter, r *http.Request){
10
11          name := r.URL.Query().Get("name")
12          age := r.URL.Query().Get("age")
13          fmt.Fprintf(w, "Им'я: %s Вік: %s", name, age)
14      })
15      fmt.Println("Server is listening...")
16      http.ListenAndServe(":8181", nil)
17  }
```

В наведеному прикладі для отримання значення окремого параметра, застосовується метод **Get()**, у який передається ім'я параметра, як показано на рисунку 11.15.

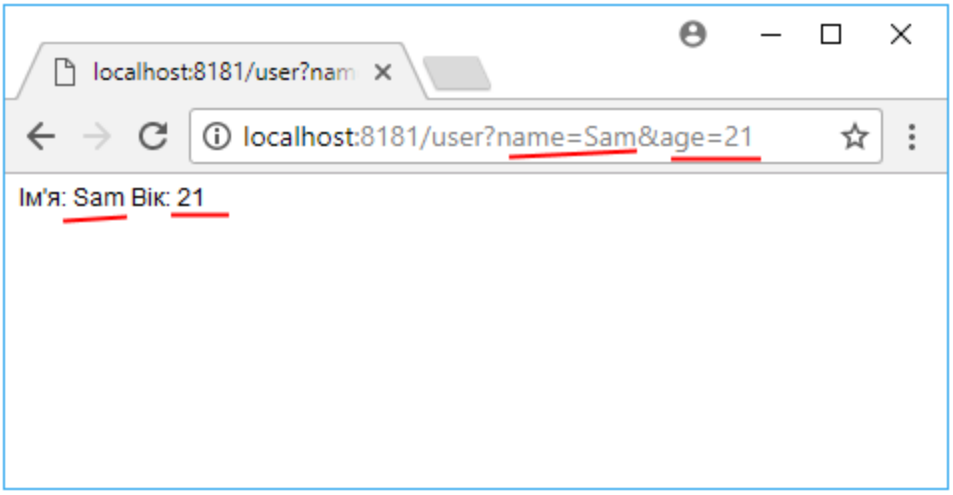

Рисунок 11.15. Результат роботи програми

11.5.2. Обробка форм

Розглянемо як можна отримати у мові Go значення відправлених форм.

У мові Go всі дані запиту інкапсулюються в об'єкті **http.Request**. Через його метод **FormValue()** можна отримати певні дані, надіслані через форму, а саме:

```
1   func (r *Request) FormValue(key string) string
```

Метод **FormValue()** видобуває дані по ключу із запиту **POST** та **PUT**, а також із рядка запиту. При цьому він завжди повертає рядок.

Визначимо поруч із файлом сервера файл **user.html** з наступним кодом:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
```

```
4           <meta charset="UTF-8">
5           <title>Введення даних</title>
6       </head>
7       <body>
8           <h2>Введення даних</h2>
9           <form method="POST" action="postform">
10              <label>Ім’я</label><br>
11              <input type="text" name="username"
    /><br><br>
12              <label>Вік</label><br>
13              <input type="number" name="userage"
    /><br><br>
14              <input type="submit" value="Надіслати" />
15          </form>
16      </body>
17  </html>
```

В наведеному прикладі у формі створюється два поля: **username** та **userage**. При натисканні на кнопку дані надсилатимуться запитом **POST** на адресу "**/postform**".

У коді сервера додамо інструкції для отримання даних наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5   )
6
7   func main() {
8
9       http.HandleFunc("/", func(w http.ResponseWriter,
    r *http.Request){
10          http.ServeFile(w, r, "user.html")
11      })
12      http.HandleFunc("/postform", func(w
    http.ResponseWriter, r *http.Request){
13
14          name := r.FormValue("username")
15          age := r.FormValue("userage")
16
```

```
17          fmt.Fprintf(w, "Ім'я: %s Вік: %s", name, age)
18      })
19      fmt.Println("Server is listening...")
20      http.ListenAndServe(":8181", nil)
21  }
```

Тоді за запитом до кореня сайту, застосунок буде відправляти користувачеві файл **user.html** для введення даних, а при надсиланні форми на адресу "**/postform**" програма отримуватиме дані, як показано на рисунку 11.16. Так як поля на формі називаються **username** та **userage**, то щоб отримати ці дані, потрібно використовувати ці імена наступним чином:

```
1   name: = r.FormValue ("username")
```

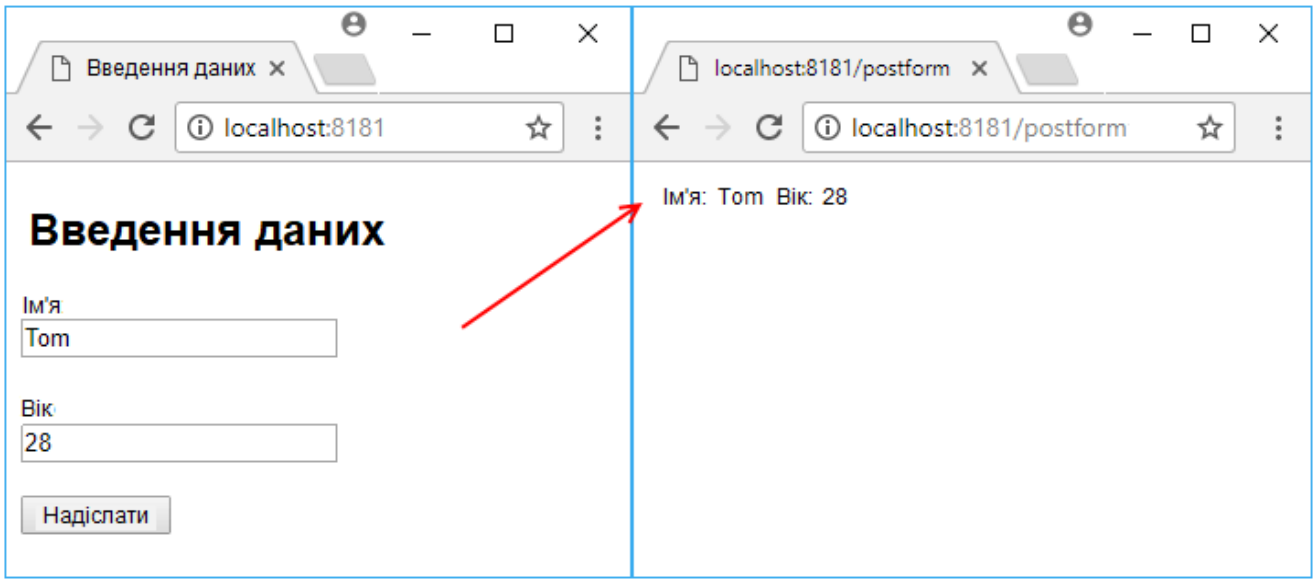

Рисунок 11.16. Результат роботи програми

При цьому **FormValue** також дозволяє отримати дані з рядка запиту, тобто, наприклад, можна передати значення для **username** і **userage** через рядок запиту, як показано на рисунку 11.17.

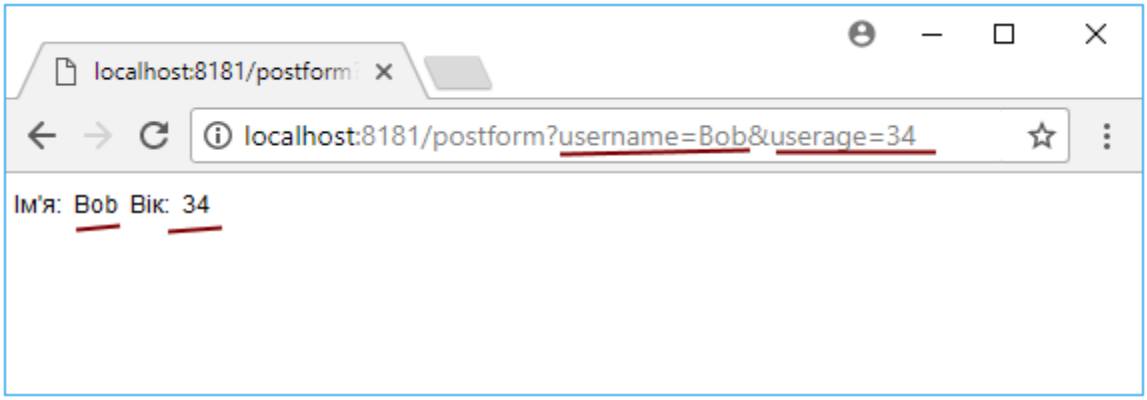

Рисунок 11.17. Результат роботи програми

Резюме

У наведеному матеріалі було розглянуто принципи вебпрограмування мовою Go. Використовуючи стандартний пакет **net/http** [9], можна створювати прості вебзастосунки, визначати маршрути, обробляти запити та відповідати на них. Для розширення можливостей маршрутизації пропонується використовувати фреймворк Gorilla Mux, який підтримує параметри в URL-адресах та регулярні вирази. Також пояснюється, як працювати зі статичними файлами за допомогою функцій **http.FileServer** і **http.ServeFile**. Особливу увагу приділено обробці рядків запиту та форм для передачі даних. Матеріал демонструє, що Go підходить для створення високопродуктивних вебсервісів, хоча потребує більше ручного кодування порівняно з іншими фреймворками.

Наступні розділи призначені для реалізації динамічних вебзастосунків з використанням шаблонів і з'єднанням із зовнішніми базами даних.

Контрольні запитання і завдання

1. Що таке пакет **net/http** і яку роль він відіграє у вебпрограмуванні на Go?
2. Як працює функція **http.ListenAndServe** і які параметри вона приймає?
3. Який інтерфейс повинен реалізувати обробник запитів у Go?
4. Як реалізується метод ServeHTTP у користувацькому типі?
5. Як створити простий вебсервер, що відповідає повідомленням "**Hello from Web Server in Go**"?
6. Яка різниця між **http.Handle** та **http.HandleFunc**?
7. Як у Go організувати маршрутизацію запитів для різних URL-адрес?
8. Що таке статичні файли, і як їх можна обслуговувати через функцію **http.FileServer**?
9. Як працює функція **http.ServeFile**? У яких випадках її доцільно використовувати?
10. Які можливості для маршрутизації пропонує Gorilla Mux?
11. Як встановлювати пакет Gorilla Mux через команду **go get**?
12. Як у Gorilla Mux можна визначити маршрут із параметрами?
13. Що означає використання регулярних виразів у маршрутах Gorilla Mux? Наведіть приклад.
14. Як у Go можна отримати значення параметрів рядка запиту (**query string**)?

15. Як обробляються HTML-форми в Go, і яка функція використовується для отримання даних із форм?
16. Чим відрізняється метод **FormValue** від метода **Query** у об’єкті **http.Request**?
17. Як організувати одночасну роботу зі статичними файлами та динамічними маршрутами?
18. Які обмеження є у стандартній маршрутизації Go і як їх вирішує Gorilla Mux?
19. Що таке метод **mux.Vars**, і як його використовують для отримання даних із маршруту?
20. Як за допомогою Gorilla Mux визначити декілька маршрутів із різними обробниками?
21. Напишіть програму, яка створює вебсервер у Go, що відповідає на всі запити повідомленням "**Hello, World!**".
22. Реалізуйте вебзастосунок із трьома маршрутами:
    / – відповідає "**Welcome to the homepage**!".
    /about – відповідає "**About us**".
    /contact – відповідає "**Contact information**".
23. Створіть вебсервер, який обслуговує статичні файли з теки **static** (наприклад, HTML-документи або зображення).
24. Реалізуйте сервер, який приймає запит із параметрами рядка запиту (наприклад, **?name=John&age=30**) і повертає повідомлення: "**Ім’я: John, Вік: 30**".
25. Створіть форму з полями для імені та віку. Після відправлення форми сервер повинен повернути введені дані.
26. Використовуючи Gorilla Mux, створіть маршрут **/product/{id:[0-9]+}**, який приймає числовий параметр **id** і повертає повідомлення: "**Product ID: [id]**".
27. Додайте маршрут /**category/{category}/product/{id:[0-9]+}**, який відповідає повідомленням: "**Category: [category], Product ID: [id]**".
28. Реалізуйте сервер, який:
- обслуговує статичні файли з теки **public**;
- має маршрут **/api/data**, що повертає JSON-об’єкт із ключами **name** і **age**.
29. Налаштуйте сервер так, щоб він повертав користувацьке повідомлення "**Page not found**" для будь-якого невизначеного маршруту.

30. Створіть API, яке приймає POST-запит із JSON-даними, що містять **username** і **email**, та повертає відповідь: "**User [username] with email [email] was successfully added!**".

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Який пакет у Go використовується для створення вебзастосунків?

a) fmt
b) os
c) net/http
d) io
Правильна відповідь: c).

Яка функція використовується для запуску вебсервера в Go?
a) http.RunServer
b) http.ListenAndServe
c) http.StartServer
d) http.ServeHTTP
Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Які параметри приймає функція http.ListenAndServe?
a) Адресу сервера
b) Обробник запитів
c) Рядок запиту
d) Файл конфігурації
Правильна відповідь: a), b).

Вставити пропущені слова

Завершіть код для створення маршруту в Go:

```
package main

import (
    "fmt"
    "net/http"
```

```
)

func main() {
    http.HandleFunc("/about", func(w
http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "_____________")
    })
    http.ListenAndServe(":8181", nil)
}
```

Правильна відповідь: …About Page…

Завдання на відповідність

Встановіть відповідність між функціями і їх призначенням:

a) http.HandleFunc → 1. Створює маршрут для запиту в Go.

b) http.ServeFile → 2. Обслуговує один статичний файл.

Правильна відповідь:

a) → 1.

b) → 2.

Завдання з вибором істинності

Статичні файли завжди обслуговуються через функцію http.ServeFile.

a) Так

b) Ні

Правильна відповідь: b).

Відкриті запитання

Вкажіть, що не так у цьому коді, і виправте:

http.ListenAndServe("8080", nil)

Правильна відповідь: Адреса повинна містити номер порту разом із localhost або IP.

Виправлений код:

```
http.ListenAndServe(":8080", nil)
```

12. ШАБЛОНИ, ЯК ЗАСІБ РЕАЛІЗАЦІЇ ДИНАМІЧНИХ ВЕБЗАСТОСУНКІВ

12.1. Створення та використання шаблонів

Раніше було розглянуто, яким чином у мові Go організовано відправку статичних файлів, зокрема, html-сторінки. Визначення контенту у вигляді html-сторінок досить зручне, адже можна використовувати переваги стеку технологій **html+css+javascript** та відокремити вигляд від основної логіки, що пишеться мовою Go. Однак статичні сторінки малокорисні, коли необхідно динамічно генерувати певний контент на основі різних факторів, наприклад параметрів, переданих через рядок запиту. І тут стають в нагоді шаблони.

Мова Go надає зручну функціональність шаблонів за замовчуванням у вигляді **html/template**.

Для прикладу застосуємо простий шаблон з наступним кодом:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "html/template"
6   )
7
8   func main() {
9
10      http.HandleFunc("/", func(w http.ResponseWriter, r
    *http.Request) {
11
12          data := "Go Template"
13          tmpl, _ := template.New("data").Parse("<h1>{{
    .}}</h1>")
14          tmpl.Execute(w, data)
15      })
16
17      fmt.Println("Server is listening...")
18      http.ListenAndServe(":8181", nil)
19  }
```

В наведеному прикладі за допомогою функції **template.New("data")** визначається ім'я шаблону. Потім для визначення самого шаблону

використовується функція **Parse("<h1>{{.}}</h1>")**. В цьому прикладі шаблон фактично представляє заголовок **h1**, але ключовим елементом тут є подвійна пара фігурних дужок **{{.}}**. Подвійна пара фігурних дужок **{{.}}** дозволяє вводити в розмітку HTML різні дані. У якості даних в цьому прикладі зазначена крапка (.). Крапка (.) вказує на контекст шаблону, тобто на всі дані, котрі передані шаблону.

Варто відзначити, що функція **Parse** повертає два значення: власне шаблон, який у наведеному прикладі представлено змінною **tmpl**) та об'єкт помилки при її виникненні. У наведеному прикладі об'єкт помилки не використовується, тому замість нього вказано порожній ідентифікатор (_). Присвоювання значення порожньому ідентифікатору фактично призводить до того, що воно втрачається і робиться це програмістом свідомо.

Для передачі шаблону даних, генерації підсумкової html-розмітки та відправлення її у відповідь на запит, застосовується функція **Execute**, а саме:

```
1   tmpl.Execute(w, data)
```

В наведеному прикладі змінна **data** представляє рядок, що якраз і є тими даними, які вставлятимуться в шаблон замість крапки **{{.}}**. Першим параметром є об'єкт **http.ResponseWriter**, через який надсилаються дані.

Результат роботи програми наведено на рисунку 12.1.

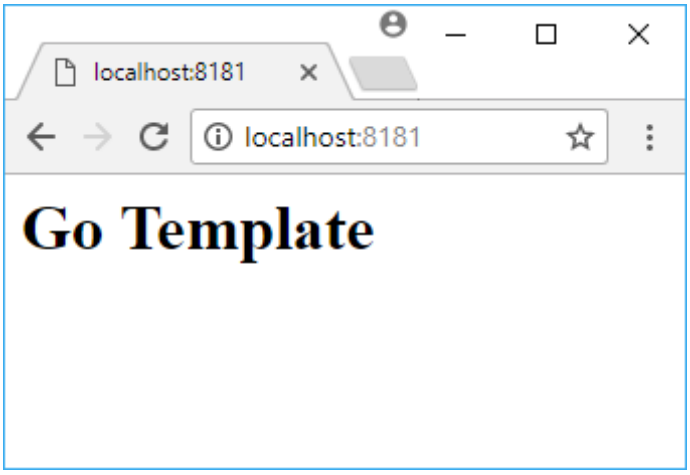

Рисунок 12.1. Результат роботи програми

Шаблони можуть приймати складніші дані, які описуються структурами. У якості прикладу розглянемо наступний код:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
```

```
5       "html/template"
6   )
7   type ViewData struct{
8
9       Title string
10      Message string
11  }
12  func main() {
13
14      http.HandleFunc("/", func(w http.ResponseWriter,
    r *http.Request) {
15
16          data := ViewData{
17              Title: "World Cup",
18              Message: "FIFA will never regret it",
19          }
20          tmpl :=
    template.Must(template.New("data").Parse(`<div>
21              <h1>{{ .Title}}</h1>
22              <p>{{ .Message}}</p>
23          </div>`))
24          tmpl.Execute(w, data)
25      })
26
27      fmt.Println("Server is listening...")
28      http.ListenAndServe(":8181", nil)
29  }
```

В цьому прикладі дані, що передаються у шаблон, описуються структурою **ViewData**, і ця структура буде представляти контекст шаблону. Тому, щоб звернутися до окремих її змінних, необхідно після крапки вказати назву змінної, а саме **{{.Title}}**.

Варто зазначити, що назви змінних необхідно визначати з великої літери.

Оскільки у наведеному прикладі використовуються складні дані, то їх треба обернути за допомогою функції **template.Must()**. Сам код шаблону можна переносити на кілька рядків, у цьому разі код розташовується між косими лапками. Якщо код шаблону розташовується в одному рядку, то можна використовувати звичайні лапки.

Результат роботи програми наведено на рисунку 12.2.

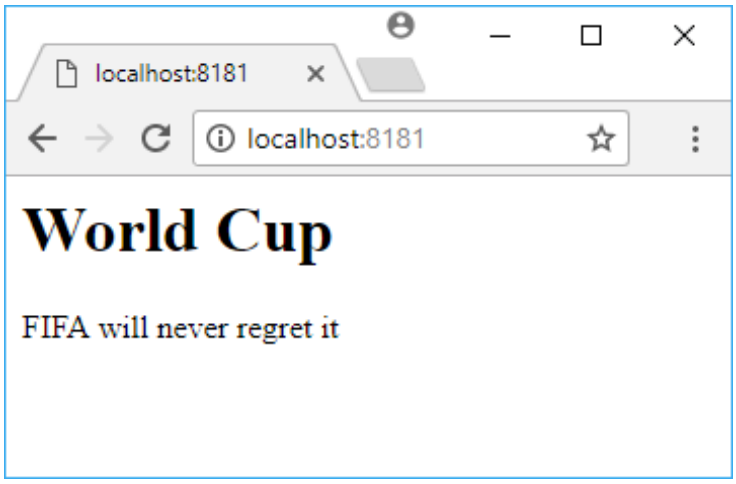

Рисунок 12.2. Результат роботи програми

Тим не менше, визначення шаблону всередині коду на мові Go є не найкращим варіантом, особливо коли шаблон містить багато складної html-розмітки, вкраплення стилів та скриптів JavaScript. Тому оптимально визначати шаблони у окремих файлах. Наприклад, визначимо у проєкті теку **templates**, а всередині її створимо файл **index.html**, як показано на рисунку 12.3.

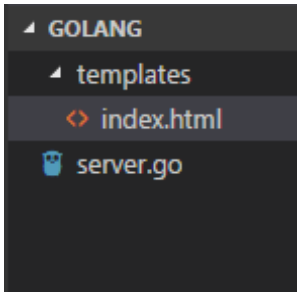

Рисунок 12.3. Вікно менеджера тек і файлів

Код файлу index.html визначимо в наступний спосіб:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>{{ .Title}}</title>
    </head>
    <body>
        <h1>{{ .Title}}</h1>
        <p>{{ .Message}}</p>
    </body>
</html>
```

Використаємо цей шаблон у коді сервера, а саме:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "html/template"
6   )
7   type ViewData struct{
8
9       Title string
10      Message string
11  }
12  func main() {
13
14      http.HandleFunc("/", func(w http.ResponseWriter,
    r *http.Request) {
15
16          data := ViewData{
17              Title: "World Cup",
18              Message: "FIFA will never regret it",
19          }
20          tmpl, _ :=
    template.ParseFiles("templates/index.html")
21          tmpl.Execute(w, data)
22      })
23
24      fmt.Println("Server is listening...")
25      http.ListenAndServe(":8181", nil)
26  }
```

В наведеному прикладі для отримання коду з файлу застосовується функція **template.ParseFiles()**, якій передається шлях до файлу. Підсумковий результат буде майже таким самим, як і в попередньому випадку, і наведено на рисунку 12.4.

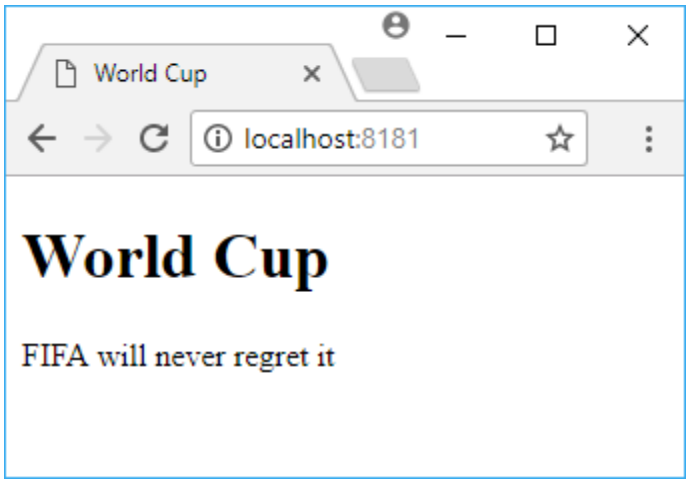

Рисунок 12.4. Результат роботи програми

12.2. Синтаксис шаблонів

Розглянемо деякі базові елементи синтаксису шаблонів, наприклад, умовні конструкції та цикли.

12.2.1. Цикли

Розглянемо приклад передачі шаблона на стороні сервера, визначений наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "html/template"
6   )
7   type ViewData struct{
8
9       Title string
10      Users []string
11  }
12  func main() {
13
14      data := ViewData{
15          Title : "Users List",
16          Users : []string{ "Tom", "Bob", "Sam", },
17      }
18      http.HandleFunc("/", func(w http.ResponseWriter, r
```

```
    *http.Request) {
19
20          tmpl, _ :=
    template.ParseFiles("templates/index.html")
21          tmpl.Execute(w, data)
22      })
23
24      fmt.Println("Server is listening...")
25      http.ListenAndServe(":8181", nil)
26  }
```

В наведеному прикладі для виведення масиву в шаблоні використовується конструкція **{{range масив}} {{end}}**. Після слова **range** вказується масив, що перебирається, а саме:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>{{ .Title}}</title>
6       </head>
7       <body>
8           <h1>{{ .Title}}</h1>
9           <ul>
10              {{range .Users}}
11                  <li><b>{{ . }}</b></li>
12              {{end}}
13          </ul>
14      </body>
15  </html>
```

Всередині конструкції **range** можна звертатися до поточного об'єкта, що перебирається, за допомогою символу крапки **{{ . }}**.

Результат роботи програми наведено на рисунку 12.5.

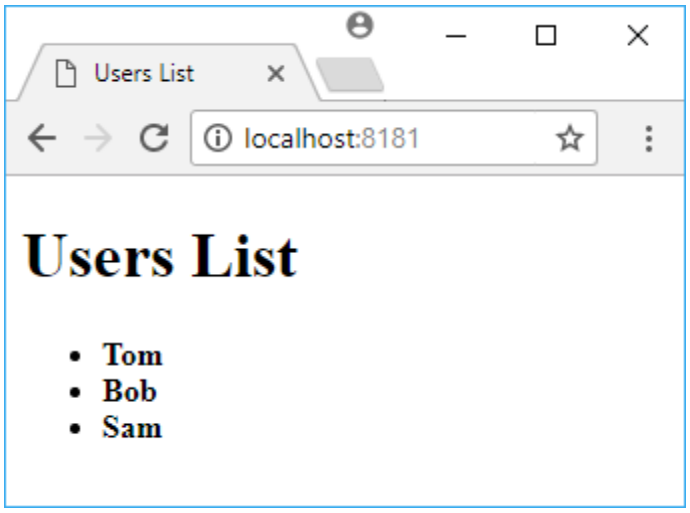

Рисунок 12.5. Результат роботи програми

Масив може не мати даних. Якщо необхідно визначити поведінку на цей випадок, то можна використовувати підконструкцію **{{else}}**, а саме:

```
<ul>
    {{range .Users}}
        <li><b>{{ . }}</b></li>
    {{else}}
        <li><b>no rows</b></li>
    {{end}}
</ul>
```

Дані в масиві можуть представляти складні дані, наприклад:

```
package main
import (
    "fmt"
    "net/http"
    "html/template"
)
type ViewData struct{

    Title string
    Users []User
}
type User struct{
    Name string
    Age int
}
func main() {
```

```
17
18      data := ViewData{
19          Title : "Users List",
20          Users : []User{
21              User{Name: "Tom", Age: 21},
22              User{Name: "Kate", Age: 23},
23              User{Name: "Alice", Age: 25},
24          },
25      }
26      http.HandleFunc("/", func(w http.ResponseWriter,
    r *http.Request) {
27
28          tmpl, _ :=
    template.ParseFiles("templates/index.html")
29          tmpl.Execute(w, data)
30      })
31
32      fmt.Println("Server is listening...")
33      http.ListenAndServe(":8181", nil)
34  }
```

Виведення цих даних у шаблоні можна організувати наступним чином:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>{{ .Title}}</title>
6       </head>
7       <body>
8           <h1>{{ .Title}}</h1>
9           <ul>
10              {{range .Users}}
11                  <li>
12                      <div><b>{{ .Name }}</b>: {{ .Age
    }}</div>
13                  </li>
14              {{end}}
15          </ul>
16      </body>
```

```
17  </html>
```

А результат роботи програми матиме вигляд, який наведено на рисунку 12.6.

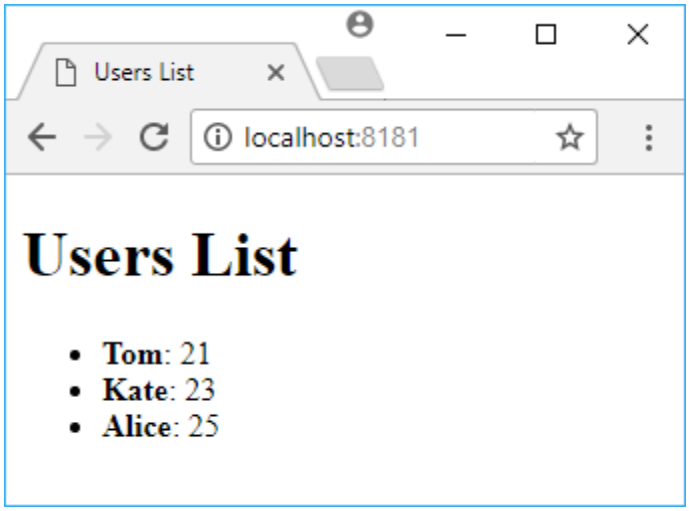

Рисунок 12.6. Результат роботи програми

12.2.2. Умовні конструкції

Для виведення у шаблоні деякої розмітки у залежності від певної умови можна використовувати конструкцію **{{if умова}}{{end}}**. Після ключового слова **if** вказується умова, яка повинна повертати значення типу **bool**: **true** або **false**.

Розглянемо приклад передачі з сервера у шаблон дані, які містять логічний вираз:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "html/template"
6   )
7   type ViewData struct{
8
9       Available bool
10  }
11  func main() {
12
13      data := ViewData{
14          Available: true,
15      }
16      http.HandleFunc("/", func(w http.ResponseWriter,
```

```
    r *http.Request) {
17
18          tmpl, _ :=
    template.ParseFiles("templates/index.html")
19          tmpl.Execute(w, data)
20      })
21
22      fmt.Println("Server is listening...")
23      http.ListenAndServe(":8181", nil)
24  }
```

Визначимо у файлі шаблону наступний код:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>Available</title>
6       </head>
7       <body>
8           <div>
9               {{if .Available}}
10              <p>Available</p>
11              {{end}}
12          </div>
13      </body>
14  </html>
```

В наведеному прикладі, якщо змінна **Available** дорівнюватиме **true**, то виводитиметься розмітка ** Available **. Якщо змінна **Available** дорівнюватиме **false**, то нічого не виводитиметься.

За допомогою конструкції **{{else}}** можна визначити розмітку **html**, яка виводиметься, якщо умова в **if** дорівнює **false**, наприклад:

```
1   <div>
2       {{if .Available}}
3       <p>Available</p>
4       {{else}}
5       <p>Not Available</p>
6       {{end}}
7   </div>
```

Також можна в умові **if** порівнювати значення. Розглянемо приклад передачі сервером у шаблон поточний час:

```
1   package main
2   import (
3       "fmt"
4       "net/http"
5       "html/template"
6       "time"
7   )
8   type ViewData struct{
9
10      Hour int
11  }
12  func main() {
13
14      data := ViewData{
15          Hour: time.Now().Hour(),
16      }
17      http.HandleFunc("/", func(w http.ResponseWriter,
    r *http.Request) {
18
19          tmpl, _ :=
    template.ParseFiles("templates/index.html")
20          tmpl.Execute(w, data)
21      })
22
23      fmt.Println("Server is listening...")
24      http.ListenAndServe(":8181", nil)
25  }
```

В наведеному прикладі за допомогою метода **time.Now().Hour()** отримується поточна година.

У шаблоні визначимо наступни конструкцію:

```
1   <div>
2       {{if lt .Hour 12 }}
3       <p>Доброго ранку</p>
4       {{else}}
5       <p>Добрий день</p>
6       {{end}}
```

```
7   </div>
```

В наведеному прикладі порівнюється значення поточної години з числом 12, і в залежності від значення виводиться той чи інший текст.

Оператор **lt** можна розшифрувати як "**less than**", тобто менше, ніж. Тобто фактично це аналог операції. Він порівнює два значення і повертає **true**, якщо перше значення менше другого. Інакше повертається значення **false**.

Подібним чином можна використовувати ще низку операторів, які аналогічні стандартним операторам порівняння:

- **eq** : повертає **true**, якщо два значення рівні;
- **ne** : повертає **true**, якщо два значення НЕ рівні;
- **le** : повертає **true**, якщо перше значення менше або дорівнює другому;
- **gt** : повертає **true**, якщо перше значення більше за друге;
- **ge** : повертає **true**, якщо перше значення більше або дорівнює другому.

Крім того, є низка операторів, які аналогічні логічним операторам:

- **and** : повертає **true**, якщо два вирази дорівнюють **true;**
- **or** : повертає **true**, якщо хоча б один із двох виразів дорівнює **true;**
- **not** : повертає **true**, якщо вираз повертає **false.**

Наведемо приклад застосування деяких операторів, а саме:

```
1   <div>
2       {{if or (gt 2 1) (lt 5 7)}}
3       <p>Перший варіант</p>
4       {{else}}
5       <p>Другий варіант</p>
6       {{end}}
7   </div>
```

Резюме

У матеріалі розділу розглянуто використання шаблонів у мові Go для створення динамічних вебзастосунків. Шаблони дозволяють генерувати HTML-код на основі даних, що передаються з серверу, розділяючи логіку і представлення. Пояснюється, як працювати з пакетом **html/template**, включаючи функції **Parse**, **Execute** та **Must**. Наведено приклади роботи з простими даними, структурами та масивами, а також використання умовних конструкцій і циклів у шаблонах. Особлива увага приділяється організації шаблонів у зовнішніх файлах для зручності роботи зі складною HTML-розміткою. Також описуються базові оператори та їх застосування для

порівнянь і логічних умов. Матеріал демонструє, як ефективно будувати гнучкі вебзастосунки з використанням шаблонів.

Наступні розділи будуть призначені реалізації динамічних вебзастосунків з використанням шаблонів і з'єднанням із зовнішніми базами.

Контрольні запитання і завдання

1. Що таке шаблони в мові Go і для чого вони використовуються?
2. Яка бібліотека використовується для роботи з шаблонами в Go?
3. Як створити новий шаблон за допомогою функції template.New?
4. Що означає конструкція **{{ . }}** у шаблонах?
5. Як передати дані в шаблон для динамічного відображення?
6. Що повертає функція **template.Parse**?
7. У яких випадках краще використовувати функцію **template.Must**?
8. Як відрізняється запис шаблону, якщо він містить кілька рядків?
9. Чому рекомендується зберігати шаблони в окремих файлах?
10. Як зчитати шаблон із файлу у Go?
11. Як реалізувати цикл у шаблоні Go для роботи з масивами?
12. Що відображатиметься в шаблоні, якщо масив, переданий у **range**, порожній?
13. Як використовувати конструкцію **{{else}}** у шаблонах?
14. Як організувати шаблон для роботи зі складними структурами даних?
15. Яке значення повинна повертати умова в конструкції **{{if ...}}**?
16. Які оператори порівняння доступні в шаблонах Go?
17. Як використовуються логічні оператори (**and**, **or**, **not**) у шаблонах?
18. Як передати на сервер поточну годину та використати її у шаблоні?
19. У яких випадках використовується метод **template.Execute**?
20. Чому змінні в структурах для шаблонів повинні починатися з великої літери?
21. Створіть сервер на Go, який генерує сторінку з привітанням типу **<h1>Welcome to Go templates</h1>** за допомогою пакета **html/template**.
22. Реалізуйте програму, яка відображає поточну дату та час на веб-сторінці, використовуючи шаблон і функцію **time.Now()**.
23. Створіть структуру **Product** з полями **Name** і **Price**. Використовуйте шаблон, щоб відобразити список продуктів у вигляді таблиці.
24. Реалізуйте веб-сервер, який передає в шаблон масив імен користувачів і відображає їх у вигляді нумерованого списку.

25. Створіть сторінку, яка відображає "**Good morning**" або "**Good evening**" залежно від поточного часу.
26. Реалізуйте програму, яка перевіряє, чи масив імен користувачів порожній, і виводить повідомлення "**No users available**", якщо даних немає.
27. Перенесіть HTML-розмітку у зовнішній файл, підключіть його до програми Go та відобразіть сторінку зі статичним заголовком і повідомленням.
28. Створіть структуру **User** з полями **Name** і **Profile**, де **Profile** є вкладеною структурою з полем **Bio**. Використовуйте шаблон для відображення цих даних.
29. Передайте в шаблон поточну годину і виведіть повідомлення залежно від часу: "**Morning**", "**Afternoon**", або "**Evening**".
30. Реалізуйте шаблон, який перевіряє два умови (наприклад, **isAdmin** і **isVerified**) і відображає текст "**Access granted**" тільки якщо обидві умови виконуються.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

Який пакет використовується для роботи з шаблонами в Go?

a) text/template
b) html/template
c) go/template
d) web/template

Правильна відповідь: b).

Що означає конструкція {{ . }} у шаблоні?

a) Початок блоку коду
b) Доступ до контексту шаблону
c) Умова виконання коду
d) Масив даних

Правильна відповідь: b).

Закриті запитання з кількома правильними відповідями

Що повертає функція template.ParseFiles()?

a) Зчитаний шаблон

b) Помилку, якщо вона виникла

c) Веб-сторінку

d) Масив шаблонів

Правильна відповідь: a), b).

Завдання з вибором істинності

Чи обов'язково використовувати template.Must для обробки шаблону?

a) Так

b) Ні

Правильна відповідь: b).

Вставити пропущені слова

Для передачі масиву в шаблон використовується конструкція {{range _____}}.

Правильна відповідь:…Users…

Відкриті запитання

Маємо шаблон:

```
{{if .Available}}<p>Available</p>{{else}}<p>Not Available</p>{{end}}
```

Значення .Available = false. Який HTML буде згенеровано?

***Правильна відповідь: Not Available ***

13. ІНТЕГРАЦІЯ БАЗ ДАНИХ У ВЕБЗАСТОСУНКАХ

13.1. Підключення до БД та отримання даних

Розглянемо прийоми та способи взаємодії з базами даних у вебзастосунках. Основні моменти роботи з БД за допомогою мови програмування Go були розглянуті у Розділі 10. У цьому розділі розглянемо лише безпосередньо застосування цих моментів у межах вебзастосунків.

У якості прикладу СКБД будемо використовувати MySQL. Спочатку створимо на сервері MySQL базу даних **productdb** з таблицею **products**. Для цього можна використовувати наступні SQL-вирази:

```
create database productdb;
use productdb;
create table products (
    id int auto_increment primary key,
    model varchar(30) not null,
    company varchar(30) not null,
    price int not null
)
```

Отже створена база даних **productdb**, в ній розташована таблиця **products**, яка зберігатиме інформацію про товари, і в якій буде чотири стовпці:

- **id** : ідентифікатор кожного запису
- **model** : назва товару;
- **company** : виробник товару;
- **price** : ціна товару.

Додамо до неї якісь початкові дані, наприклад, за допомогою наступного коду:

```
insert into productdb.Products (model, company,
price)
values ('iPhone X', 'Apple', 74000),
('Pixel 2', 'Google', 62000),
('Galaxy S9', 'Samsung', 65000)
```

Перш ніж почати працювати з MySQL, необхідно додати драйвер для Go до змінної **$GOPATH**, якщо він раніше не був доданий. Для цього потрібно виконати в командному рядку/терміналі наступну команду, як наведено на рисунку 13.1.

```
go get github.com/go-sql-driver/mysql
```

Рисунок 13.1. Зовнішній вигляд вікна введення команди

Після цього визначимо на сервері наступний код:

```
package main
import (
    "fmt"
    "database/sql"
    _ "github.com/go-sql-driver/mysql"
    "net/http"
    "html/template"
    "log"
)
type Product struct{
    Id int
    Model string
    Company string
    Price int
}
var database *sql.DB

func IndexHandler(w http.ResponseWriter, r
*http.Request) {

    rows, err := database.Query("select * from
productdb.Products")
    if err != nil {
        log.Println(err)
    }
    defer rows.Close()
    products := []Product{}

    for rows.Next(){
        p := Product{}
        err := rows.Scan(&p.Id, &p.Model, &p.Company,
&p.Price)
        if err != nil{
            fmt.Println(err)
```

```
32              continue
33          }
34          products = append(products, p)
35      }
36
37      tmpl, _ :=
    template.ParseFiles("templates/index.html")
38      tmpl.Execute(w, products)
39  }
40
41  func main() {
42
43      db, err := sql.Open("mysql",
    "root:password@/productdb")
44
45      if err != nil {
46          log.Println(err)
47      }
48      database = db
49      defer db.Close()
50      http.HandleFunc("/", IndexHandler)
51
52      fmt.Println("Server is listening...")
53      http.ListenAndServe(":8181", nil)
54  }
```

В наведеному прикладі визначено структуру **Product**, яка відповідає визначенню таблиці **products** у базі даних. А за взаємодію з базою даних відповідає змінна **database**.

Для надсилання користувачеві списку об'єктів з БД визначено функцію **IndexHandler**. У ній за допомогою метода **database.Query** виконується запит "**select * from productdb.Products**", який видобуває всі об'єкти з таблиці. Потім з отриманого набору створюється масив структур **Product**, який потім передається у шаблон **index.html**, код якого буде наведено нижче.

У функції **main** відкривається підключення до бази даних за допомогою функції **sql.Open**, а саме:

```
1   db, err := sql.Open("mysql",
    "root:password@/productdb")
```

У цій функції в перший параметр передається назва драйвера "**mysql**". Другий параметр описує налаштування підключення, де **root** є ім'ям користувача MySQL, **password** є паролем цього користувача (як правило тим, який використовувався при встановленні MySQL), а **productdb** є назвою бази даних. Відповідно, у кожному конкретному випадку пароль може відрізнятися.

Після відкриття підключення встановлюється значення змінної database, а саме:

```
database = db
```

Потім у якості обробника запитів за кореневою адресою призначається функція **IndexHandler**, а саме:

```
http.HandleFunc("/", IndexHandler)
```

Додамо в проєкті теку **templates**, а в ній створимо новий файл **index.html**, який буде шаблоном для виведення масиву об'єктів і матиме наступний код:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Products</title>
    </head>
    <body>
        <table>
            <thead><th>Id</th><th>Model</th><th>Compan
y</th><th>Price</th></thead>
            {{range . }}
            <tr>
                <td>{{.Id}}</td>
                <td>{{.Model}}</td>
                <td>{{.Company}}</td>
                <td>{{.Price}}</td>
            </tr>
            {{end}}
        </table>
    </body>
</html>
```

В результаті запуску проєкту та звернення до кореня сайту буде відкрито підключення до бази даних, програма отримає всі необхідні дані з БД та передасть їх у шаблон, як наведено на рисунку 13.2.

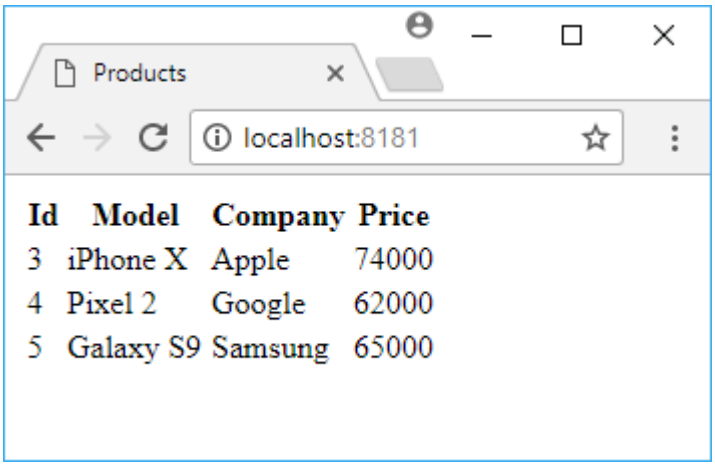

Рисунок 13.2. Результат роботи програми

13.2. Додавання даних

Процес додавання даних складається з кількох частин. Спочатку потрібно показати користувачеві форму для введення даних. Потім, коли користувач надішле введені дані, потрібно їх отримати та додати до БД.

Удосконалимо проєкт із минулого матеріалу. І перш за все додамо до теки **templates** новий файл **create.html** , як показано на рисунку 13.3, який міститиме форму для додавання даних, а саме:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>New Product</title>
    </head>
    <body>
        <h3>Add Product</h3>
        <form method="POST">
            <label>Model</label><br>
            <input type="text" name="model"
/><br><br>
            <label>Company</label><br>
            <input type="text" name="company"
/><br><br>
            <label>Price</label><br>
            <input type="number" name="price"
/><br><br>
            <input type="submit" value="Send" />
```

```
17          </form>
18      </body>
19  </html>
```

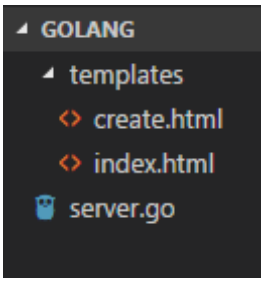

Рисунок 13.3. Вікно менеджера тек і файлів

Змінимо файл сервера наступним чином:

```
1   package main
2   import (
3       "fmt"
4       "database/sql"
5       _ "github.com/go-sql-driver/mysql"
6       "net/http"
7       "html/template"
8       "log"
9   )
10  type Product struct{
11      Id int
12      Model string
13      Company string
14      Price int
15  }
16  var database *sql.DB
17
18  // Функція додавання даних
19  func CreateHandler(w http.ResponseWriter, r
    *http.Request) {
20      if r.Method == "POST" {
21
22          err := r.ParseForm()
23          if err != nil {
24              log.Println(err)
25          }
```

```
26          model := r.FormValue("model")
27          company := r.FormValue("company")
28          price := r.FormValue("price")
29
30          _, err = database.Exec("insert into
    productdb.Products (model, company, price) values (?,
    ?, ?)",
31              model, company, price)
32
33          if err != nil {
34              log.Println(err)
35            }
36          http.Redirect(w, r, "/", 301)
37      }else{
38          http.ServeFile(w,r, "templates/create.html")
39      }
40  }
41
42  func IndexHandler(w http.ResponseWriter, r
    *http.Request) {
43
44      rows, err := database.Query("select * from
    productdb.Products")
45      if err != nil {
46          log.Println(err)
47      }
48      defer rows.Close()
49      products := []Product{}
50
51      for rows.Next(){
52          p := Product{}
53          err := rows.Scan(&p.Id, &p.Model, &p.Company,
    &p.Price)
54          if err != nil{
55              fmt.Println(err)
56              continue
57          }
58          products = append(products, p)
```

```
59      }
60
61      tmpl, _ :=
    template.ParseFiles("templates/index.html")
62      tmpl.Execute(w, products)
63  }
64
65  func main() {
66
67      db, err := sql.Open("mysql",
    "root:password@/productdb")
68
69      if err != nil {
70          log.Println(err)
71      }
72      database = db
73      defer db.Close()
74      http.HandleFunc("/", IndexHandler)
75      http.HandleFunc("/create", CreateHandler)
76
77      fmt.Println("Server is listening...")
78      http.ListenAndServe(":8181", nil)
79  }
```

В наведеному прикладі для додавання даних визначено функцію **CreateHandler()**. Оскільки вимагається з одного боку, відображати користувачеві форму для додавання, а, з іншого боку, отримувати і додавати дані БД, то ця функція умовно розділена на дві частини. У ній перевіряється тип запиту. Якщо запит має тип "**GET**", то повертатиметься форма для додавання. Якщо запит має тип "**POST**", то виконуватиметься парсинг даних отриманої форми для видобування з них потрібних елементів.

Для отримання даних з отриманих форм застосовується метод **r.FormValue()**. У якості параметрів цього метода передаються назви даних. Тобто, наприклад, якщо на формі є наступне поле:

```
1   <input type="text" name="model" />
```

То атрибут **name** вказує, що назва цього поля "**model**". Отже, щоб отримати введені у нього дані, необхідно використовувати наступний вираз:

```
1   model := r.FormValue("model")
```

Після отримання всіх даних вони додаються до БД за допомогою метода **database.Exec**, а саме:

```
database.Exec("insert into productdb.Products (model, company, price) values (?, ?, ?)", model, company, price)
```

Після цього виконується переадресація у корінь сайту за допомогою функції **http.Redirect**, а саме:

```
http.Redirect(w, r, "/", 301)
```

Третій параметр вказує шлях переадресації. У наведеному прикладі це "/", який виводить список об'єктів з БД. Четвертим параметром є статусний код переадресації. У наведеному прикладі код 301 вказує, що переадресація тимчасова.

Якщо запит до сервера має тип **GET**, то користувачеві просто повертається вебсторінка **create.html**, а саме:

```
http.ServeFile(w,r, "templates/create.html")
```

Функція **IndexHandler**, яка повертає список об'єктів, залишається такою самою, як і в попередньому розділі.

У функції **main** функція **CreateHandler** встановлюється як обробник по шляху "**/create**", а саме:

```
http.HandleFunc("/create", CreateHandler)
```

Після запуску програми і переході по шляху "**http://localhost:8181/create**" відобразиться форма, зовнішній вигляд якої наведено на рисунку 13.4, і в яку можна вводити якісь дані.

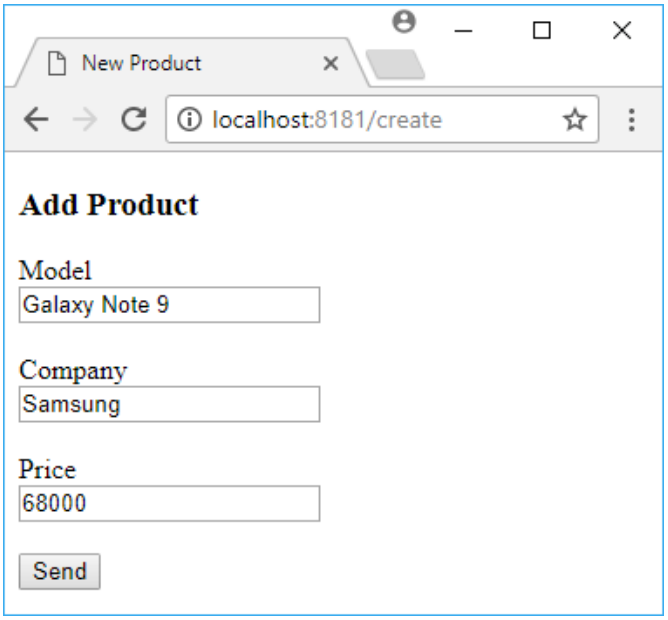

Рисунок 13.4. Форма для введення даних

Після відправлення форми застосунок отримає дані, додасть їх у БД і переадресує на головну сторінку, як показано на рисунку 13.5.

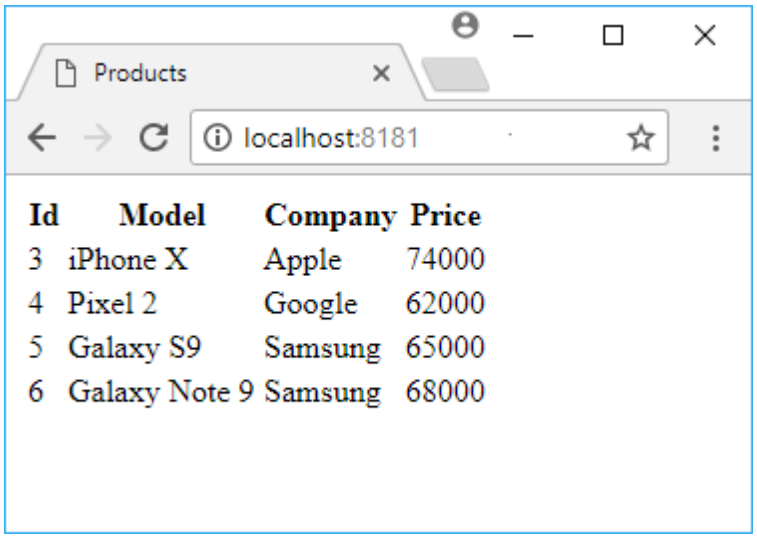

Рисунок 13.5. Результат роботи програми

13.3. Редагування даних

Продовжимо роботу з проєктом з минулого розділу та додамо до нього можливість редагування даних.

Редагування даних, як і додавання, розбивається на кілька частин. Спочатку необхідно відобразити користувачеві форму зміни вибраного об'єкта. Потім необхідно отримати надіслані дані та зберегти їх у БД.

Насамперед визначимо форму для редагування. Для цього у теці **templates** створимо файл **edit.html**, як показано на рисунку 13.6.

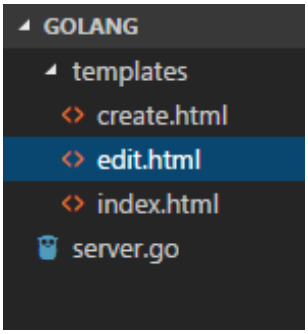

Рисунок 13.6. Вікно менеджера тек і файлів

Визначимо у файлі **edit.html** наступний код:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
```

```
4           <meta charset="UTF-8">
5           <title>Edt Product</title>
6       </head>
7       <body>
8           <h3>Edit Product</h3>
9           <form method="POST">
10              <input type="hidden" name="id"
    value="{{.Id}}" />
11              <label>Model</label><br>
12              <input type="text" name="model"
    value="{{.Model}}" /><br><br>
13              <label>Company</label><br>
14              <input type="text" name="company"
    value="{{.Company}}" /><br><br>
15              <label>Price</label><br>
16              <input type="number" name="price"
    value="{{.Price}}" /><br><br>
17              <input type="submit" value="Send" />
18          </form>
19      </body>
20  </html>
```

Створений файл буде виконувати роль шаблону, в який з коду сервера будуть передаватися відредаговані дані.

Тепер змінимо код сервера, додавши можливість редагування, а саме:

```
1   package main
2   import (
3       "fmt"
4       "database/sql"
5       _ "github.com/go-sql-driver/mysql"
6       "net/http"
7       "html/template"
8       "log"
9       "github.com/gorilla/mux"
10  )
11  type Product struct{
12      Id int
13      Model string
14      Company string
```

```
15      Price int
16  }
17  var database *sql.DB
18
19  // Повертаємо користувачеві сторінку для редагування
    об'єкта
20  func EditPage(w http.ResponseWriter, r *http.Request)
    {
21      vars := mux.Vars(r)
22      id := vars["id"]
23
24      row := database.QueryRow("select * from
    productdb.Products where id = ?", id)
25      prod := Product{}
26      err := row.Scan(&prod.Id, &prod.Model,
    &prod.Company, &prod.Price)
27      if err != nil{
28          log.Println(err)
29          http.Error(w, http.StatusText(404),
    http.StatusNotFound)
30      }else{
31          tmpl, _ :=
    template.ParseFiles("templates/edit.html")
32          tmpl.Execute(w, prod)
33      }
34  }
35  // Отримуємо змінені дані та зберігаємо їх у БД
36  func EditHandler(w http.ResponseWriter, r
    *http.Request) {
37      err := r.ParseForm()
38      if err != nil {
39          log.Println(err)
40      }
41      id := r.FormValue("id")
42      model := r.FormValue("model")
43      company := r.FormValue("company")
44      price := r.FormValue("price")
45
```

```
46      _, err = database.Exec("update productdb.Products
    set model=?, company=?, price = ? where id = ?",
47          model, company, price, id)
48
49      if err != nil {
50          log.Println(err)
51      }
52      http.Redirect(w, r, "/", 301)
53  }
54
55  func CreateHandler(w http.ResponseWriter, r
    *http.Request) {
56      if r.Method == "POST" {
57
58          err := r.ParseForm()
59          if err != nil {
60              log.Println(err)
61          }
62          model := r.FormValue("model")
63          company := r.FormValue("company")
64          price := r.FormValue("price")
65
66          _, err = database.Exec("insert into
    productdb.Products (model, company, price) values (?,
    ?, ?)",
67              model, company, price)
68
69          if err != nil {
70              log.Println(err)
71          }
72          http.Redirect(w, r, "/", 301)
73      }else{
74          http.ServeFile(w,r, "templates/create.html")
75      }
76  }
77
78  func IndexHandler(w http.ResponseWriter, r
    *http.Request) {
```

```
79
80      rows, err := database.Query("select * from
    productdb.Products")
81      if err != nil {
82          log.Println(err)
83      }
84      defer rows.Close()
85      products := []Product{}
86
87      for rows.Next(){
88          p := Product{}
89          err := rows.Scan(&p.Id, &p.Model, &p.Company,
    &p.Price)
90          if err != nil{
91              fmt.Println(err)
92              continue
93          }
94          products = append(products, p)
95      }
96
97      tmpl, _ :=
    template.ParseFiles("templates/index.html")
98      tmpl.Execute(w, products)
99  }
100
101 func main() {
102
103     db, err := sql.Open("mysql",
    "root:password@/productdb")
104
105     if err != nil {
106         log.Println(err)
107     }
108     database = db
109     defer db.Close()
110
111     router := mux.NewRouter()
112     router.HandleFunc("/", IndexHandler)
```

```
113     router.HandleFunc("/create", CreateHandler)
114     router.HandleFunc("/edit/{id:[0-9]+}",
    EditPage).Methods("GET")
115     router.HandleFunc("/edit/{id:[0-9]+}",
    EditHandler).Methods("POST")
116
117     http.Handle("/",router)
118
119     fmt.Println("Server is listening...")
120     http.ListenAndServe(":8181", nil)
121 }
```

В порівняні з попереднім матеріалом в наведеному прикладі додано функції **EditPage** та **EditHandler**, а також змінено функцію **main**.

Щоб вказати, який об'єкт буде редагуватися, використовується адреса **id** цього об'єкта. Для спрощення маршрутизації використовується пакет **gorilla/mux**.

У функції **IndexPage** отримується **id** об'єкта, який необхідно змінити. Потім вибираємо з БД дані цього об'єкта та передаємо їх у шаблон **edit.html**, а саме:

```
1   func EditPage(w http.ResponseWriter, r *http.Request)
    {
2       vars := mux.Vars(r)
3       id := vars["id"]
4
5       row := database.QueryRow("select * from
    productdb.Products where id = ?", id)
6       prod := Product{}
7       err := row.Scan(&prod.Id, &prod.Model,
    &prod.Company, &prod.Price)
8       if err != nil{
9           log.Println(err)
10          http.Error(w, http.StatusText(404),
    http.StatusNotFound)
11      }else{
12          tmpl, _ :=
    template.ParseFiles("templates/edit.html")
13          tmpl.Execute(w, prod)
14      }
```

```
15 }
```

На випадок, якщо в БД не виявиться об'єкт з подібним **id**, за допомогою функції **http.Error()** повертаємо статусний код 404, який вказує, що об'єкт не знайдено.

У функції **EditHandler** отримуються дані з відправленої форми та з їх допомогою змінюється об'єкт у БД за певним **id**, а саме.

```
1   func EditHandler(w http.ResponseWriter, r
    *http.Request) {
2       err := r.ParseForm()
3       if err != nil {
4           log.Println(err)
5       }
6       id := r.FormValue("id")
7       model := r.FormValue("model")
8       company := r.FormValue("company")
9       price := r.FormValue("price")
10
11      _, err = database.Exec("update productdb.Products
    set model=?, company=?, price = ? where id = ?",
12          model, company, price, id)
13
14      if err != nil {
15          log.Println(err)
16      }
17      http.Redirect(w, r, "/", 301)
18  }
```

Після оновлення БД виконується перенаправлення на головну сторінку.

У функції **main** функції **EditPage** та **EditHandler** пов'язуються з певними маршрутами. По суті вони прив'язані до одного і того ж маршруту, проте для різного типу запитів. **EditPage** призначено для **GET** запитів, а **EditHandler** призначено для запитів POST, а саме:

```
1   router.HandleFunc("/edit/{id:[0-9]+}",
    EditPage).Methods("GET")
2   router.HandleFunc("/edit/{id:[0-9]+}",
    EditHandler).Methods("POST")
```

Варто зазначити, що додавання даних, яке в наведеному прикладі представлене **CreateHandler**, також фактично виконує дві дії залежно від типу запиту: відображення сторінки додавання та, власне, додавання даних.

Організацію додавання можна виконати так само, як і редагування, розділивши на дві функції для кожного типу запитів.

Для спрощення керуванням об'єктами змінимо файл **index.html**, додавши до нього посилання на редагування, наприклад:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Products</title>
    </head>
    <body>
        <h2>Перелік товарів</h2>
        <p><a href="/create">Додати</a></p>
        <table>
            <thead><th>Id</th><th>Model</th><th>Company
</th><th>Price</th><th></th></thead>
            {{range . }}
            <tr>
                <td>{{.Id}}</td>
                <td>{{.Model}}</td>
                <td>{{.Company}}</td>
                <td>{{.Price}}</td>
                <td><a href="/edit/{{.Id}}">Змінити</a>
                </td>
            </tr>
            {{end}}
        </table>
    </body>
</html>
```

І після запуску програми стає можливим редагування об'єкти, як показано на рисунку 13.7.

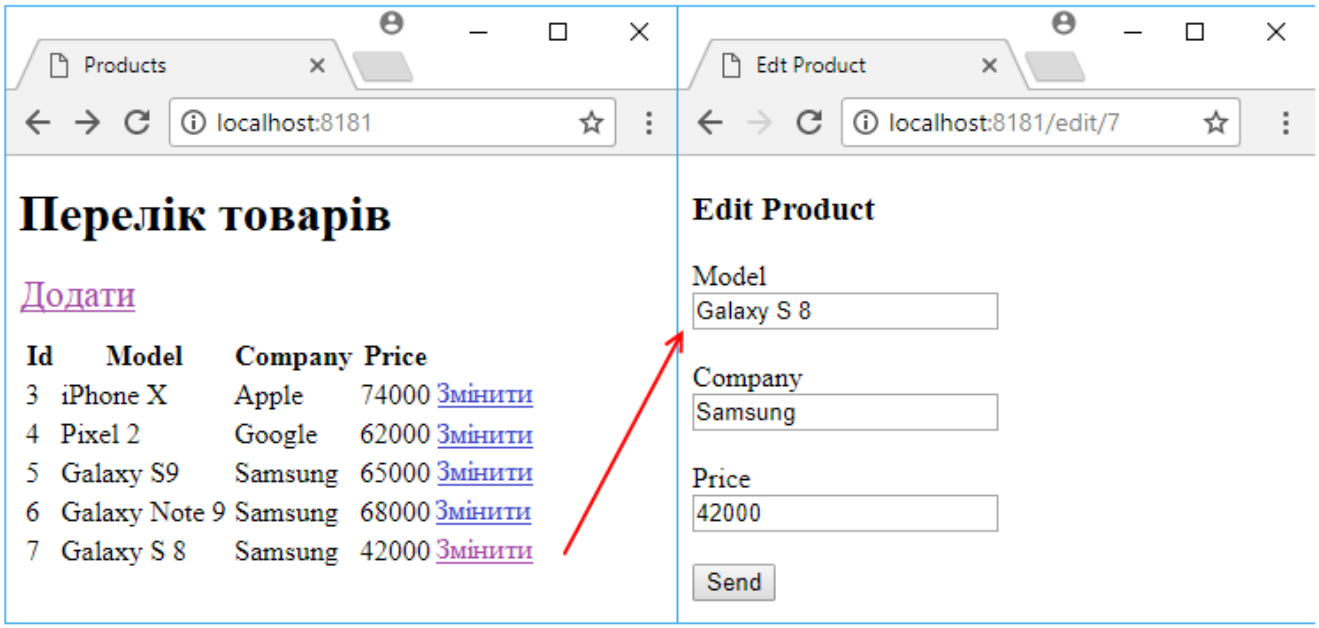

Рисунок 13.7. Результат роботи програми

13.4. Видалення даних

Продовжимо роботу з проєктом та додамо до нього можливість видалення об'єктів. Видалення даних можна організувати у різний спосіб. Спочатку розглянемо найпростіший приклад, коли у GET-запиті передається **id** об'єкта, а у БД відбувається видалення рядка за цим **id**.

Розглянемо змінений код сервера, а саме:

```
1   package main
2   import (
3       "fmt"
4       "database/sql"
5       _ "github.com/go-sql-driver/mysql"
6       "net/http"
7       "html/template"
8       "log"
9       "github.com/gorilla/mux"
10  )
11  type Product struct{
12      Id int
13      Model string
14      Company string
15      Price int
16  }
17  var database *sql.DB
```

```
18
19 func DeleteHandler(w http.ResponseWriter, r
   *http.Request) {
20     vars := mux.Vars(r)
21     id := vars["id"]
22
23     _, err := database.Exec("delete from
   productdb.Products where id = ?", id)
24     if err != nil{
25         log.Println(err)
26     }
27
28     http.Redirect(w, r, "/", 301)
29 }
30
31 func EditPage(w http.ResponseWriter, r *http.Request)
   {
32     vars := mux.Vars(r)
33     id := vars["id"]
34
35     row := database.QueryRow("select * from
   productdb.Products where id = ?", id)
36     prod := Product{}
37     err := row.Scan(&prod.Id, &prod.Model,
   &prod.Company, &prod.Price)
38     if err != nil{
39         log.Println(err)
40         http.Error(w, http.StatusText(404),
   http.StatusNotFound)
41     }else{
42         tmpl, _ :=
   template.ParseFiles("templates/edit.html")
43         tmpl.Execute(w, prod)
44     }
45 }
46
47 func EditHandler(w http.ResponseWriter, r
   *http.Request) {
```

```
48      err := r.ParseForm()
49      if err != nil {
50          log.Println(err)
51      }
52      id := r.FormValue("id")
53      model := r.FormValue("model")
54      company := r.FormValue("company")
55      price := r.FormValue("price")
56
57      _, err = database.Exec("update productdb.Products
    set model=?, company=?, price = ? where id = ?",
58          model, company, price, id)
59
60      if err != nil {
61          log.Println(err)
62      }
63      http.Redirect(w, r, "/", 301)
64  }
65
66  func CreateHandler(w http.ResponseWriter, r
    *http.Request) {
67      if r.Method == "POST" {
68
69          err := r.ParseForm()
70          if err != nil {
71              log.Println(err)
72          }
73          model := r.FormValue("model")
74          company := r.FormValue("company")
75          price := r.FormValue("price")
76
77          _, err = database.Exec("insert into
    productdb.Products (model, company, price) values (?,
    ?, ?)",
78              model, company, price)
79
80          if err != nil {
81              log.Println(err)
```

```
82          }
83          http.Redirect(w, r, "/", 301)
84      }else{
85          http.ServeFile(w,r, "templates/create.html")
86      }
87  }
88
89  func IndexHandler(w http.ResponseWriter, r
    *http.Request) {
90
91      rows, err := database.Query("select * from
    productdb.Products")
92      if err != nil {
93          log.Println(err)
94      }
95      defer rows.Close()
96      products := []Product{}
97
98      for rows.Next(){
99          p := Product{}
100         err := rows.Scan(&p.Id, &p.Model, &p.Company,
    &p.Price)
101         if err != nil{
102             fmt.Println(err)
103             continue
104         }
105         products = append(products, p)
106     }
107
108     tmpl, _ :=
    template.ParseFiles("templates/index.html")
109     tmpl.Execute(w, products)
110 }
111
112 func main() {
113
114     db, err := sql.Open("mysql",
    "root:password@/productdb")
```

```
115
116     if err != nil {
117         log.Println(err)
118     }
119     database = db
120     defer db.Close()
121
122     router := mux.NewRouter()
123     router.HandleFunc("/", IndexHandler)
124     router.HandleFunc("/create", CreateHandler)
125     router.HandleFunc("/edit/{id:[0-9]+}",
    EditPage).Methods("GET")
126     router.HandleFunc("/edit/{id:[0-9]+}",
    EditHandler).Methods("POST")
127     router.HandleFunc("/delete/{id:[0-9]+}",
    DeleteHandler)
128
129     http.Handle("/",router)
130
131     fmt.Println("Server is listening...")
132     http.ListenAndServe(":8181", nil)
133 }
```

У порівнянні з минулим матеріалом в наведеному прикладі додана функція **DeleteHandler**, яка отримує **id** об'єкта, що видаляється і виконує DELETE-запит до БД. Після вилучення відбувається переадресація на головну сторінку.

Для спрощення видалення визначимо у файлі **index.html** посилання на видалення поряд з кожним об'єктом, а саме:

```
1   <!DOCTYPE html>
2   <html>
3       <head>
4           <meta charset="UTF-8">
5           <title>Products</title>
6       </head>
7       <body>
8           <h2>Перелік товарів</h2>
9           <p><a href="/create">Додати</a></p>
10          <table>
```

```
11              <thead><th>Id</th><th>Model</th><th>Company
    </th><th>Price</th><th></th></thead>
12              {{range . }}
13              <tr>
14                  <td>{{.Id}}</td>
15                  <td>{{.Model}}</td>
16                  <td>{{.Company}}</td>
17                  <td>{{.Price}}</td>
18                  <td><a href="/edit/{{.Id}}">Змінити</a>
    |
19                      <a
    href="/delete/{{.Id}}">Видалити</a>
20                  </td>
21              </tr>
22              {{end}}
23          </table>
24      </body>
25  </html>
```

Для перевірки видалення необхідно натиснути на відповідне посилання поряд з якимсь об'єктом, як проказано на рисунку 13.8.

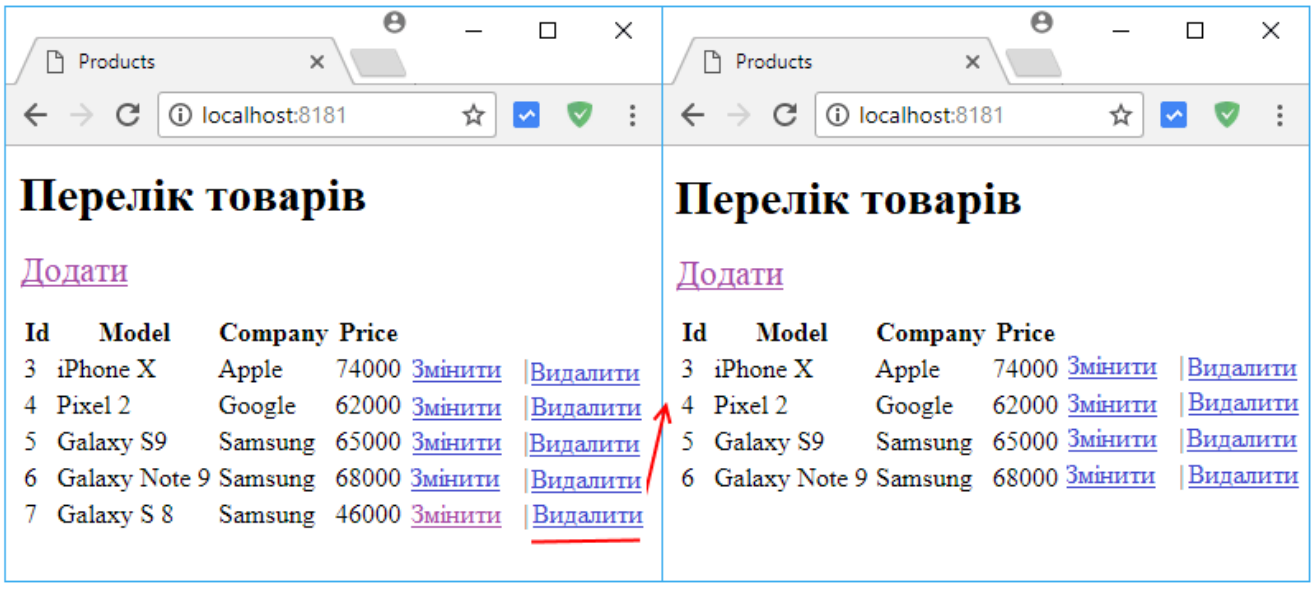

Рисунок 13.8. Результат роботи програми

Резюме

У наведеному матеріалі було розглянуто інтеграцію баз даних у вебзастосунках на мові програмування Go з використанням MySQL.

Демонструється створення бази даних, підключення до неї за допомогою драйвера Go-SQL-Driver, а також взаємодія з таблицею **products**. Представлено приклади обробки HTTP-запитів для відображення, додавання, редагування та видалення даних. Застосовуються HTML-шаблони для динамічного формування вебсторінок, а маршрутизація здійснюється через пакет **Gorilla/Mux** [17]. В результаті розроблено простий CRUD-додаток, що реалізує базові операції роботи з даними.

Для поглибленого вивчення як мови Go, так і вебпрограмування рекомендуємо звернутись до «Переліку посилань» та «Списку рекомендованої літератури».

Контрольні запитання і завдання

1. Що таке база даних і як вона використовується у вебзастосунках?
2. Який SQL-запит використовується для створення бази даних у MySQL?
3. Як створити таблицю в базі даних для зберігання інформації про товари?
4. Які поля є в таблиці **products**, створеній у прикладі?
5. Як додати початкові дані до таблиці **products**?
6. Яка команда використовується для завантаження драйвера MySQL у Go?
7. Як відкрити підключення до бази даних у Go?
8. Що означає рядок підключення "**root:password@/productdb**" у Go?
9. Що таке структура **Product**, і як вона використовується у коді?
10. Як реалізується запит до бази даних для отримання всіх записів у таблиці?
11. Як у Go передати дані з бази даних у HTML-шаблон?
12. Яка роль функції **IndexHandler** у наведеному коді?
13. Як створюється форма для додавання даних у файл **create.html**?
14. Як у Go обробляти дані, надіслані через POST-запит?
15. Як у коді реалізовано редагування даних у базі?
16. Які маршрути використовуються для редагування записів, і як їх створити?
17. Як видалити запис із бази даних за його **id**?
18. Для чого використовується пакет **Gorilla/Mux** у цьому проєкті?
19. Як динамічно формуються посилання для редагування та видалення записів у **index.html**?

20. Які основні кроки створення CRUD-додатка на основі Go і MySQL?
21. Напишіть SQL-запити для створення бази даних **storedb** із таблицею **items**, яка містить такі поля: **id**, **name**, **category**, **price**, **quantity**.
22. Реалізуйте у Go підключення до MySQL-бази даних **storedb** та перевірте підключення на наявність помилок.
23. Реалізуйте функцію у Go, яка отримує всі записи з таблиці **items** і виводить їх у консоль.
24. Створіть HTML-шаблон для відображення даних про товари у вигляді таблиці, а також реалізуйте обробник HTTP-запиту для його заповнення даними.
25. Створіть форму для додавання нового товару до таблиці **items**. Реалізуйте обробник у Go, який додає новий запис на основі введених даних.
26. Додайте функціональність для редагування даних про товари. Реалізуйте форму редагування, заповнену поточними даними, і відповідний обробник для збереження змін.
27. Реалізуйте функцію видалення запису з таблиці **items** за **id**, створивши відповідний маршрут і обробник у Go.
28. Змініть файл **index.html**, щоб додати динамічні посилання на редагування та видалення записів.
29. Розширте таблицю **items**, додавши поле **description**, і реалізуйте функціонал для відображення та редагування цього поля.
30. Додайте новий маршрут, який відображає тільки товари, ціна яких перевищує 1000. Реалізуйте відповідний SQL-запит і обробник для цієї функції.

Огляд тестових завдань

Закриті запитання з одним варіантом відповіді

1. Який SQL-запит використовується для створення бази даних?

a) create database products;
b) new database products;
c) add database products;
d) init database products;

Правильна відповідь: а).

Закриті запитання з кількома правильними відповідями

Які кроки потрібні для інтеграції бази даних у Go-додаток?

a) Встановити драйвер бази даних

b) Відкрити підключення до бази

c) Додати таблицю в шаблон HTML

d) Виконати SQL-запити через код

Правильна відповідь: a), b), d).

Завдання з вибором істинності

Для роботи з MySQL у Go потрібно встановити зовнішній драйвер.

a) Так

b) Ні

Правильна відповідь: a).

Завдання на відповідність

Співставте термін із його визначенням:

a) sql.Open → 1. Підключення до бази даних.

b) template.ParseFiles → 2. Парсинг HTML-шаблона.

c) database.Query → 3. Виконання SQL-запиту до бази.

d) http.Redirect → 4. Перенаправлення на інший маршрут.

Правильна відповідь:

a) → 1.

b) → 2.

c) → 3.

d) → 4.

Вставити пропущені слова

Для отримання даних із бази даних використовується функція _____.

Правильна відповідь:…Query…

Завдання з кодом

Напишіть Go-функцію, яка отримує всі записи з таблиці orders та повертає їх у вигляді JSON.

Правильна відповідь:

```
    func GetOrders(w http.ResponseWriter, r
*http.Request) {
        rows, err := database.Query("select * from
orders")
        if err != nil {
            http.Error(w, err.Error(),
http.StatusInternalServerError)
            return
        }
        defer rows.Close()

        orders := []Order{}
        for rows.Next() {
            o := Order{}
            rows.Scan(&o.Id, &o.Name, &o.Amount)
            orders = append(orders, o)
        }
        json.NewEncoder(w).Encode(orders)
    }
```

Відкриті запитання

Як можна обробити помилку під час виконання SQL-запиту в Go?

Правильна відповідь: Перевіривши значення err після виконання запиту.

ПЕРЕЛІК ПОСИЛАНЬ

1. The Go Programming Language.htm [Електронний ресурс]. - Режим доступу: https://go.dev. - Назва з екрана.
2. All releases - The Go Programming Language.htm [Електронний ресурс]. - Режим доступу: https://go.dev/dl. - Назва з екрана.
3. Google Code Archive - Long-term storage for Google Code Project Hosting.htm [Електронний ресурс]. - Режим доступу: https://code.google.com/archive/p/liteide. - Назва з екрана.
4. GitHub - visualfc_liteide_ LiteIDE is a simple, open source, cross-platform Go IDE.htm [Електронний ресурс]. - Режим доступу: https://github.com/visualfc/liteide. - Назва з екрана.
5. Visual Studio Code - Code Editing. Redefined.htm [Електронний ресурс]. - Режим доступу: https://code.visualstudio.com. - Назва з екрана.
6. Standard library - Go Packages.htm [Електронний ресурс]. - Режим доступу: https://golang.org/pkg. - Назва з екрана.
7. Fmt package - fmt - Go Packages.htm [Електронний ресурс]. - Режим доступу: https://golang.org/pkg/fmt. - Назва з екрана.
8. Net package - net - Go Packages.htm [Електронний ресурс]. - Режим доступу: https://golang.org/pkg/net. - Назва з екрана.
9. Http package - net_http - Go Packages.htm [Електронний ресурс]. - Режим доступу: https://golang.org/pkg/net/http. - Назва з екрана.
10. Sql package - database_sql - Go Packages [Електронний ресурс]. - Режим доступу: https://golang.org/pkg/database/sql/. - Назва з екрана.
11. Go Wiki_ SQL Database Drivers - The Go Programming Language.htm [Електронний ресурс]. - Режим доступу: https://go.dev/wiki/SQLDrivers. - Назва з екрана.
12. GitHub - go-sql-driver_mysql_ Go MySQL Driver is a MySQL driver for Go's (golang) database_sql package.htm [Електронний ресурс]. - Режим доступу: https://github.com/go-sql-driver/mysql. - Назва з екрана.
13. GitHub - lib_pq_ Pure Go Postgres driver for database_sql.htm [Електронний ресурс]. - Режим доступу: https://github.com/lib/pq. - Назва з екрана.
14. Pq package - github.com_lib_pq - Go Packages.htm [Електронний ресурс]. - Режим доступу: https://godoc.org/github.com/lib/pq. - Назва з екрана.
15. GitHub - mattn_go-sqlite3_ sqlite3 driver for go using database_sql.htm [Електронний ресурс]. - Режим доступу: https://github.com/mattn/go-sqlite3. - Назва з екрана.

16. Mgo - Rich MongoDB driver for Go.htm [Електронний ресурс]. - Режим доступу: https://labix.org/mgo. - Назва з екрана.
17. Gorilla, the golang web toolkit.htm [Електронний ресурс]. - Режим доступу: http://www.gorillatoolkit.org. - Назва з екрана.

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ І ЕЛЕКТРОННИХ РЕСУРСІВ

1. Jay McGavren. Head First Go. «O'Reilly Media», 2022 – 556 p.
2. Head First HTML and CSS: A Learner's Guide to Creating Standards-Based Web Pages. O'Reilly Media. 2019. – 762 p.
3. Head First PHP & MySQL. O'Reilly Media. 2018. – 812 p.
4. Head First. Програмування на JavaScript. Фабула. 2022. – 672 p.
5. Donovan A. A., Kernighan B. W. The Go Programming Language. – Addison-Wesley, 2015. – 380 p.
6. Bodner J. Learning Go: An Idiomatic Approach to Real-World Go Programming. – O'Reilly Media, 2021. – 390 p.
7. Youngman N. Get Programming with Go. – Manning Publications, 2018. – 432 p.
8. Edwards A. Let’s Go: Learn to Build Professional Web Applications with Go. – 2nd Edition, 2022. – 380 p.
9. Edwards A. Let’s Go Further! – Self-published, 2022. – 410 p.
10. Bates M., LaNou C. Go Web Programming. – Manning Publications, 2016. – 320 p.
11. Ryer M. Go Programming Blueprints. – Packt Publishing, 2016. – 268 p.
12. Ryer M. Go Design Patterns. – Packt Publishing, 2016. – 280 p.
13. Kennedy W., Ketelsen B., St. Martin J. Go in Action. – Manning Publications, 2015. – 300 p.
14. Kennedy W. Ultimate Go Notebook. – Self-published, 2021. – 260 p.
15. Chang S. S. Go Web Programming. – Manning Publications, 2016. – 300 p.
16. Tsoukalos M. Mastering Go. – Packt Publishing, 2021. – 540 p.
17. Woodbeck A. Network Programming with Go. – No Starch Press, 2021. – 450 p.
18. McMillan M. Data Structures and Algorithms with Go. – O'Reilly Media, 2019. – 350 p.
19. Go на Udemy – Онлайн-курси [Електронний ресурс]. - Режим доступу: https://www.udemy.com/courses/search/?q=golang. - Назва з екрана.
20. Go на Coursera – Курси від провідних університетів [Електронний ресурс]. - Режим доступу: https://www.coursera.org/courses?query=golang. - Назва з екрана.

21. Golang на Pluralsight – Платформа з курсами програмування [Електронний ресурс]. - Режим доступу: https://www.pluralsight.com/courses?query=golang. - Назва з екрана.
22. Golang на Educative – Інтерактивні курси [Електронний ресурс]. - Режим доступу: https://www.educative.io/courses?q=golang. - Назва з екрана.
23. Golang на Codecademy – Початкові курси з Go [Електронний ресурс]. - Режим доступу: https://www.codecademy.com/catalog/language/go. - Назва з екрана.
24. Go: The Complete Developer’s Guide (Udemy [Електронний ресурс]. - Режим доступу: https://www.udemy.com/course/go-the-complete-developers-guide/. - Назва з екрана.
25. Learn Go with Tests – Безкоштовний навчальний курс [Електронний ресурс]. - Режим доступу: https://quii.gitbook.io/learn-go-with-tests/. - Назва з екрана.
26. Gophercises – Практичні завдання [Електронний ресурс]. - Режим доступу: https://gophercises.com/. - Назва з екрана.
27. Golang Forum – Спільнота розробників [Електронний ресурс]. - Режим доступу: https://forum.golangbridge.org/. - Назва з екрана.
28. Go на Stack Overflow – Відповіді на питання [Електронний ресурс]. - Режим доступу: https://stackoverflow.com/questions/tagged/go. - Назва з екрана.
29. R/golang на Reddit – Дискусійний форум [Електронний ресурс]. - Режим доступу: https://www.reddit.com/r/golang/. - Назва з екрана.
30. Go Community Slack – Чат-спільнота [Електронний ресурс]. - Режим доступу: https://gophers.slack.com/. - Назва з екрана.
31. Dev.to – Go Topics – Статті та блоги [Електронний ресурс]. - Режим доступу: https://dev.to/t/golang. - Назва з екрана.