O'REILLY®

Helion

Język Go

Tworzenie kodu z wykorzystaniem najlepszych konwencji i praktyk

Wydanie II

Jon Bodner

Tytuł oryginału: Learning Go: An Idiomatic Approach to Real-world Go Programming, 2nd Edition

Tłumaczenie: Robert Górczyński, z wykorzystaniem fragmentów „Język Go. Tworzenie idiomatycznego kodu w praktyce" w tłumaczeniu Piotra Cieślaka

ISBN: 978-83-289-1476-6

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/jegot2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Typy podstawowe i deklaracje

Masz już skonfigurowane środowisko programistyczne, przyszedł więc czas na zapoznanie się z cechami języka Go i możliwościami ich jak najlepszego wykorzystania. A jeśli już mowa o dążeniu do „najlepszego", kieruj się jedną nadrzędną zasadą: pisz programy w sposób jasno określający Twoje intencje. Omawiając poszczególne aspekty języka, przedstawię różne możliwości postępowania i wyjaśnię, dlaczego dane podejście pozwala uzyskać czystszy kod.

Zaczniemy od przyjrzenia się typom wbudowanym w Go oraz sposobom deklarowania zmiennych tych typów. Choć każdy programista zna podstawy tych zagadnień, język Go pod pewnymi względami jest wyjątkowy — także w tym przypadku dzielą go od innych subtelne różnice.

Typy wbudowane

Język Go oferuje wiele typów wbudowanych w język, nazywanych typami *podstawowymi*. Są one podobne do typów znajdujących się w wielu innych językach: wartości boolowskie (logiczne), całkowite, zmiennoprzecinkowe i łańcuchy znaków. Stosowanie tych typów w sposób idiomatyczny, a zatem charakterystyczny dla języka Go, bywa jednak wyzwaniem dla programistów „przesiadających" się na Go z innych języków. Za chwilę zapoznasz się z tymi typami oraz optymalnymi sposobami korzystania z nich w języku Go. Najpierw jednak omówię kilka koncepcji mających zastosowanie w odniesieniu do wszystkich typów.

Wartość zerowa

Jak większość nowoczesnych języków programowania, Go przypisuje domyślną *wartość zerową* dowolnej zmiennej, która została zadeklarowana, lecz nie przypisano jej żadnej konkretnej wartości. Jawne przypisanie wartości zerowej przyczynia się do większej przejrzystości kodu i pozwala uniknąć błędów charakterystycznych dla programów w językach C i C++. Opowiadając o poszczególnych typach, będę zarazem nawiązywał do wartości zerowej dla każdego z nich. Więcej informacji o wartości zerowej znajdziesz w specyfikacji języka programowania Go (*https://go.dev/ref/spec#The_zero_value*).

Literały

Literał w języku Go reprezentuje wartość liczbową, znakową lub tekstową (łańcuch znaków). W programach napisanych w Go najczęściej występują cztery rodzaje literałów (jest też, rzadko używany, piąty rodzaj, o którym opowiem przy okazji omawiania liczb zespolonych).

Literały całkowite to sekwencje liczb; są to zwykle liczby dziesiętne (o podstawie 10), lecz przy użyciu prefiksów można wskazać inne rodzaje podstaw: `0b` dla liczb binarnych (podstawa 2), `0o` dla liczb ósemkowych (podstawa 8) oraz `0x` dla wartości heksadecymalnych, czyli szesnastkowych (podstawa 16). Prefiks może być zapisany wielką lub małą literą. Zapis z cyfrą `0` bez następującej po niej litery jest alternatywnym sposobem oznaczenia literału ósemkowego, odradzam jednak stosowanie go, bo jest bardzo mylący.

Aby ułatwić odczytywanie długich literałów całkowitych, Go umożliwia wstawianie w nich podkreśleń. Pozwala to na przykład grupować cyfry w liczbach dziesiętnych według tysięcy (1_234). Podkreślenia nie mają wpływu na wartość liczby. Jedyne ograniczenie dotyczące stosowania podkreśleń polega na tym, że nie mogą się one znaleźć na początku ani na końcu wartości i nie wolno wstawiać ich obok siebie. Nic nie stoi na przeszkodzie, by każdą cyfrę literału oddzielić podkreśleniem (1_2_3_4), lecz nie zalecam takiego zapisu. Korzystaj z podkreśleń jedynie w celu poprawienia czytelności liczby — jako separatora tysięcy w liczbach dziesiętnych albo jako separatora grup jedno-, dwu- albo czterobajtowych w wartościach binarnych, ósemkowych i szesnastkowych.

Literały zmiennoprzecinkowe mają znak dziesiętny oddzielający część ułamkową wartości od całkowitej. Mogą też zawierać wykładnik potęgi oznaczony literą `e` oraz wartością dodatnią lub ujemną (na przykład `6.03e23`). Istnieje możliwość zapisywania liczb szesnastkowych z prefiksem `0x` oraz literą `p` oznaczającą wykładnik (np. wartość szesnastkowa `0x12.34p5` jest odpowiednikiem wartości dziesiętnej `582.5`). Tak jak w przypadku literałów całkowitych, do formatowania literałów zmiennoprzecinkowych możesz użyć podkreśleń.

Literały typu rune (ang. *rune literals*; dosł. literały runiczne albo po prostu runy) reprezentują znaki i są ujęte w pojedyncze cudzysłowy. W odróżnieniu od wielu innych języków pojedyncze i podwójne cudzysłowy w Go *nie mogą* być używane zamiennie. Runy można zapisywać jako pojedyncze znaki w standardzie Unicode (`'a'`), ósemkowe wartości 8-bitowe (`'\141'`), szesnastkowe wartości 8-bitowe (`'\x61'`), wartości 16-bitowe (`'\u0061'`) albo 32-bitowe wartości zgodne z Unicode (`'\U00000061'`). Istnieje kilka literałów runicznych z ukośnikiem odwrotnym (\). Najprzydatniejsze z nich to znak nowego wiersza (`'\n'`), tabulator (`'\t'`), cudzysłów pojedynczy (`'\''`), cudzysłów podwójny (`'\"'`) oraz ukośnik odwrotny (`'\\'`).

Od strony praktycznej literały liczbowe najlepiej jest zapisywać w postaci wartości dziesiętnych. Wartości ósemkowe są stosowane rzadko, najczęściej do reprezentowania wartości uprawnień POSIX (na przykład `0o777` dla uprawnień `rwxrwxrwx`). Wartości szesnastkowe i binarne są też niekiedy używane w filtrach bitowych oraz aplikacjach sieciowych i infrastrukturalnych. Unikaj szesnastkowych znaków modyfikacji w przypadku literałów runicznych, chyba że dzięki temu kod będzie czytelniejszy

Literały łańcuchowe można oznaczać na dwa sposoby. W większości przypadków należy użyć w tym celu podwójnych cudzysłowów, które umożliwiają utworzenie *interpretowanego literału łańcuchowego*

(na przykład `"Serdeczne pozdrowienia"`). Mogą one zawierać dowolną liczbę literałów runicznych, w którejkolwiek spośród dopuszczalnych dla nich postaci. Są one nazywane „interpretowanymi", ponieważ mają możliwość interpretacji literałów runicznych (zarówno liczbowych, jak i zawierających ukośnik odwrotny) jako pojedynczych znaków.

Jeden literał runiczny z ukośnikiem odwrotnym jest niedozwolony w literale łańcuchowym: grawis. Zostanie on zastąpiony literałem cudzysłowu podwójnego.

Jedyne znaki, które nie mogą w nich występować, to niepoprzedzone znakiem modyfikacji ukośniki odwrotne, znaki nowego wiersza i podwójne cudzysłowy. Jeśli chcesz skorzystać z interpretowanego literału łańcuchowego i sformatować go tak, by słowo „pozdrowienia" znalazło się w drugim wierszu oraz zostało ujęte w podwójny cudzysłów, napisz: **`"Serdeczne\n\"pozdrowienia\""`**.

Jeśli chcesz umieścić w łańcuchu znaków ukośniki odwrotne, podwójne cudzysłowy albo znaki nowego wiersza, użyj *zwykłego literału łańcuchowego*, zwanego też *surowym*. Taki literał wyodrębnia się przy użyciu grawisów (`` ` ``) i można w nim zawrzeć dowolny znak oprócz grawisu. W literale surowym nie ma żadnych znaków modyfikacji, a wszystkie znaki są interpretowane dosłownie. Korzystając ze zwykłego literału łańcuchowego, podzielone na dwa wiersze pozdrowienia można zapisać tak:

```
`Serdeczne
"pozdrowienia"`
```

Literały w Go *nie są typowane* (do tego zagadnienia jeszcze powrócę w dalszej części rozdziału). W części poświęconej przypisywaniu zmiennych przeczytasz o sytuacjach, w których typ nie jest jawnie zadeklarowany. W takich przypadkach język Go nadaje literałowi *typ domyślny* — zmierzam do tego, że typ domyślny jest stosowany wtedy, gdy wyrażenie w żaden sposób nie określa typu literału. O domyślnych typach literałów będzie mowa później, przy okazji opisywania typów wbudowanych.

Zmienne boolowskie

Typ `bool` oznacza zmienne boolowskie (logiczne). Zmienne typu `bool` mogą przyjmować jedną z dwóch wartości: `true` lub `false`. Wartość zerowa dla zmiennej typu `bool` oznacza `false`:

```
var flag bool // Brak przypisanej wartości, false
var isAwesome = true
```

Trudno opisać typy zmiennych bez uprzedniego zaprezentowania deklaracji zmiennych i na odwrót. Zaczniemy od deklaracji zmiennych, które zostaną opisane w dalszej części tego rozdziału (podrozdział „Słowo kluczowe var a operator :=").

Typy numeryczne

Język Go oferuje dużą liczbę typów numerycznych: aż 12 (i kilka nazw specjalnych), pogrupowanych w trzy kategorie. Jeśli znasz inne języki programowania, takie jak JavaScript, w którym występuje tylko jeden typ numeryczny, możesz uznać, że to bardzo wiele. W praktyce niektóre

z nich są używane bardzo często, inne zaś można potraktować jako ciekawostkę. Zacznę od omówienia typów całkowitych, a po nich przedstawię typy zmiennoprzecinkowe oraz bardzo specyficzny typ zespolony.

Typy całkowite

Go umożliwia działania na liczbach całkowitych ze znakiem i bez znaku, w różnych rozmiarach — od 1 do 8 bajtów. Zostały one zebrane w tabeli 2.1.

Tabela 2.1. Typy całkowite w Go

Nazwa typu	Zakres wartości
`int8`	od –128 do 127
`int16`	od –32768 do 32767
`int32`	od –2147483648 do 2147483647
`int64`	od –9223372036854775808 do 9223372036854775807
`uint8`	od 0 do 255
`uint16`	od 0 do 65535
`uint32`	od 0 do 4294967295
`uint64`	od 0 do 18446744073709551615

Być może jest to oczywiste, lecz wartość zerowa dla wszystkich typów całkowitych to `0`.

Specjalne typy całkowite

Go umożliwia stosowanie nazw specjalnych dla niektórych typów całkowitych. Nazwa `byte` jest odpowiednikiem typu `uint8`; oznacza to, że dopuszczalne jest przypisywanie, porównywanie i wykonywanie działań matematycznych na obu tych typach w sposób zamienny. W kodzie napisanym w Go rzadko spotyka się typ `uint8`; programiści zwykle posługują się nazwą `byte`.

Drugą nazwą specjalną jest `int`. W przypadku architektury 32-bitowej `int` jest 32-bitową wartością całkowitą ze znakiem, analogiczną jak `int32`. W większości architektur 64-bitowych `int` jest 64-bitową wartością całkowitą ze znakiem, będącą odpowiednikiem typu `int64`. Ponieważ specyfika typu `int` nie jest spójna i różni się między platformami, przypisywanie, porównywanie i wykonywanie działań arytmetycznych między typami `int`, `int32` i `int64` bez konwersji typów jest niedozwolone (zob. punkt „Jawna konwersja typów” w dalszej części tego rozdziału). Domyślnym typem literałów całkowitych jest `int`.

Istnieją niestandardowe architektury 64-bitowe, w których typ `int` jest równoważny 32-bitowej wartości całkowitej ze znakiem. Go obsługuje trzy takie architektury: amd64p32, mips64p32 i mips64p32le.

Trzecią nazwą specjalną jest `uint`. Dotyczące jej zasady są takie same jak w przypadku `int`, z tą różnicą, że reprezentuje ona wartości bez znaku (są to wyłącznie wartości dodatnie lub 0).

Istnieją jeszcze dwie nazwy specjalne dotyczące typów całkowitych: `rune` i `uintptr`. O literałach runicznych była mowa wcześniej; typ `rune` omówię w dalszej części tego rozdziału (punkt „Przedsmak informacji o łańcuchach i runach”), a typ `uintptr` w rozdziale 16.

Wybór typu całkowitego

Go oferuje więcej typów całkowitych niż wiele innych języków. Biorąc pod uwagę bogactwo dostępnych możliwości, można się zastanowić, w jakich sytuacjach należy używać każdego z nich. Warto przy tym kierować się trzema prostymi zasadami:

- Jeśli korzystasz z plików w formacie binarnym lub protokołu sieciowego, który opiera się na wartości całkowitej o określonym rozmiarze (ze znakiem lub bez), użyj pasującego typu całkowitego.
- Jeśli piszesz funkcję biblioteczną, która powinna działać z dowolnym typem całkowitym, wykorzystaj oferowaną przez Go obsługę typów sparametryzowanych i użyj parametru generycznego do przedstawienia dowolnego typu całkowitego. (O funkcjach i ich argumentach przeczytasz więcej w rozdziale 5., natomiast o typach sparametryzowanych — w rozdziale 8.).
- We wszystkich pozostałych przypadkach po prostu używaj typu `int`.

Prawdopodobnie natkniesz się na stary kod zawierający parę funkcji wykonujących to samo zadanie, przy czym jedna używa dla parametrów typu `int64`, podczas gdy w drugiej typem parametrów jest `uint64`. Wynika to stąd, że API zostało utworzone jeszcze przed dodaniem obsługi typów sparametryzowanych do języka Go. Ze względu na brak obsługi typów sparametryzowanych w celu zaimplementowania potrzebnego algorytmu konieczne było napisanie wielu podobnych funkcji różniących się głównie nazwami. Zastosowanie typów `int64` i `uint64` umożliwia napisanie kodu funkcji tylko raz i wywoływanie jej z wykorzystaniem mechanizmów konwersji typów — przy przekazywaniu wartości oraz przy przekształcaniu zwróconych danych.

Rozwiązanie to zostało zastosowane w bibliotece standardowej języka Go, w odniesieniu do funkcji `FormatInt/FormatUint` z pakietu `strconv`.

Operatory wartości całkowitych

Na liczbach całkowitych w Go można wykonywać działania przy użyciu zwykłych operatorów arytmetycznych: `+`, `-`, `*`, `/` oraz `%` dla operacji modulo. Rezultat dzielenia liczb całkowitych jest liczbą całkowitą; jeśli chcesz otrzymać wartość zmiennoprzecinkową, musisz dokonać konwersji typów całkowitych na wartości zmiennoprzecinkowe. Uważaj też, by nie dzielić wartości całkowitej przez 0, gdyż wywołuje to alarm typu *panic* (o funkcjach `panic` i `recover` możesz przeczytać w rozdziale 9.).

Dzielenie liczb całkowitych podlega zasadzie „obcinania wyniku w stronę zera”; więcej informacji na ten temat możesz znaleźć w dokumentacji języka Go, w części dotyczącej operatorów matematycznych (*https://oreil.ly/zp3OJ*).

Dowolny z operatorów arytmetycznych możesz połączyć ze znakiem `=`, aby zmodyfikować wartość zmiennej; konstrukcje te wyglądają następująco: `+=`, `-=`, `*=`, `/=` oraz `%=`. Na przykład poniższy kod powoduje przypisanie zmiennej `x` wartości 20:

```
var x int = 10
x *= 2
```

Do porównywania wartości całkowitych służą operatory ==, !=, >, >=, < oraz <=.

Go oferuje ponadto operatory umożliwiające wykonywanie działań na bitach wartości całkowitych. Możesz skorzystać z operatorów przesunięcia w lewo i w prawo (<< i >>) oraz masek bitowych: & (logiczne AND), | (logiczne OR), ^ (logiczne XOR) i &^ (logiczne AND NOT). Podobnie jak w przypadku operatorów arytmetycznych, możesz łączyć operatory logiczne ze znakiem =, aby zmodyfikować wartość zmiennej. Konstrukcje te mają postać: &=, |=, ^=, &^=, <<= oraz >>=.

Typy zmiennoprzecinkowe

Go oferuje dwa typy zmiennoprzecinkowe, wymienione w tabeli 2.2.

Tabela 2.2. Typy zmiennoprzecinkowe w Go

Nazwa typu	Największa wartość bezwzględna	Najmniejsza (niezerowa) wartość bezwzględna
`float32`	3.40282346638528859811704183484516925440e+38	1.40129846432481707092372958328991613128 0e-45
`float64`	1.79769313486231570814527423731704356798 1e+308	4.94065645841246544176568792868221372365 1e-324

Tak jak w przypadku typów całkowitych, wartością zerową dla typów zmiennoprzecinkowych jest 0.

Obsługa wartości zmiennoprzecinkowych w Go jest podobna jak w przypadku analogicznych wartości w innych językach. Go opiera się na specyfikacji IEEE 754, co oznacza duży zakres wartości i ograniczoną dokładność. Wybór właściwego typu zmiennoprzecinkowego jest prosty: jeśli nie musisz zadbać o zachowanie zgodności z istniejącym formatem, użyj typu `float64`. Ponadto, ponieważ domyślnym typem literałów zmiennoprzecinkowych jest właśnie `float64`, stosowanie go jest najprostszym wyjściem. Pomaga ono też złagodzić problemy z dokładnością obliczeń zmiennoprzecinkowych, gdyż typ `float32` oferuje jedynie sześć albo siedem cyfr dokładności po przecinku. Nie przejmuj się różnicami dotyczącymi wymagań związanych z pamięcią operacyjną, chyba że przy użyciu profilera ustaliłeś, że stanowią one przyczynę istotnych problemów. (O testowaniu i profilowaniu możesz przeczytać w rozdziale 15.).

Ważniejsze pytanie dotyczy stosowania wartości zmiennoprzecinkowych w ogóle. W większości przypadków nie jest to konieczne. Tak jak w innych językach programowania, wartości zmiennoprzecinkowe w Go obejmują bardzo duży zakres, lecz nie każda z nich może być precyzyjnie przechowywana; używane są jedynie ich przybliżenia. Ze względu na nieprecyzyjność tych wartości mogą one być stosowane jedynie w tych przypadkach, gdy niedokładności nie stanowią problemu, a programista dobrze zna ograniczenia związane z posługiwaniem się nimi. W rezultacie zakres ich użyteczności kończy się na zastosowaniach graficznych, statystycznych i naukowych.

Literał zmiennoprzecinkowy nie odzwierciedla dokładnej wartości dziesiętnej. Nie używaj takich literałów do obliczeń związanych z finansami lub innych, wymagających absolutnej precyzji! W rozdziale 10. przedstawię moduł zewnętrzny przeznaczony do obsługi dokładnych wartości dziesiętnych.

IEEE 754

Jak już wspomniałem, Go (tak jak większość innych języków programowania) przechowuje wartości zmiennoprzecinkowe zgodnie z normą IEEE 754.

Kwestie zasad zawartych w tej normie wykraczają poza merytoryczny zakres tej książki i nie są proste. Więcej informacji o standardzie IEEE 754 znajdziesz na stronie *https://floating-point-gui.de/formats/fp/*.

W odniesieniu do wartości zmiennoprzecinkowych możesz używać wszystkich standardowych operatorów działań matematycznych i porównań, z wyjątkiem `%`. Dzielenie liczb zmiennoprzecinkowych ma kilka interesujących właściwości. Dzielenie niezerowej liczby zmiennoprzecinkowej przez 0 daje rezultat `+Inf` albo `-Inf` (dodatnia albo ujemna nieskończoność, zależnie od znaku liczby). Dzielenie zmiennej zmiennoprzecinkowej o wartości 0 przez 0 zwraca wartość `NaN` (nieliczba; ang. *not a number*).

Wprawdzie Go umożliwia porównywanie liczb zmiennoprzecinkowych przy użyciu operatorów `==` i `!=`, nie zalecam jednak takiego postępowania. Ze względu na typową dla tych liczb nieprecyzyjność dwie wartości zmiennoprzecinkowe mogą nie być równe, nawet jeśli masz wrażenie, że powinny. W takich przypadkach lepiej jest zdefiniować maksymalną dozwoloną wariancję i sprawdzić, czy różnica między dwiema wartościami zmiennoprzecinkowymi jest mniejsza od niej. Wartość ta (określana niekiedy jako *epsilon*) jest uzależniona od wymaganej dokładności; nie da się tu podać jednoznacznej zasady. Jeśli nie masz pewności co do sposobu postępowania, poradź się znajomego matematyka. Jeżeli nie masz znajomego matematyka, w witrynie internetowej The Floating-Point Guide znajduje się strona *Comparison* (*https://floating-point-gui.de/errors/comparison/*), która może Ci pomóc w rozwiązaniu problemu. (Ewentualnie skłoni Cię do unikania liczb zmiennoprzecinkowych, o ile ich użycie nie jest absolutnie konieczne).

Typy zespolone (zapewne nie będziesz się nimi posługiwać)

Go oferuje jeszcze jeden typ numeryczny, o dość niestandardowych właściwościach. Właściwości te decydują o wyjątkowych możliwościach języka Go pod względem obsługi liczb zespolonych. Jeśli nie wiesz, co to takiego, raczej nie będziesz mieć potrzeby się nimi posługiwać — możesz zatem bez przeszkód przejść do dalszej części tego rozdziału.

Obsługa liczb zespolonych w Go nie wymaga wielu komentarzy. W języku tym zostały zdefiniowane dwa typy obsługujące tego rodzaju wartości. Do odzwierciedlenia części rzeczywistej i urojonej w typie `complex64` są wykorzystywane wartości `float32`, a w typie `complex128` — wartości `float64`. Obydwa typy są deklarowane przy użyciu wbudowanej funkcji `complex`:

```
var complexNum = complex(20.3, 10.2)
```

Rezultat działania tej funkcji w języku Go podlega kilku zasadom:

- Jeśli do obu argumentów funkcji przekażesz nietypowane stałe lub literały, uzyskasz nietypowany literał zespolony, którego typ domyślny to `complex128`.
- Jeśli do obu argumentów funkcji `complex` przekażesz wartości typu `float32`, uzyskasz typ `complex64`.

- Jeśli jedna z wartości jest typu `float32`, a druga jest nietypowaną stałą albo literałem, który nie mieści się w ograniczeniach typu `float32`, uzyskasz typ `complex64`.
- W innych przypadkach uzyskasz typ `complex128`.

W odniesieniu do liczb zespolonych możesz używać wszystkich standardowych operatorów arytmetycznych. Podobnie jak w przypadku wartości zmiennoprzecinkowych, możesz je porównywać przy użyciu operatorów `==` i `!=`, lecz podlegają one tym samym ograniczeniom dotyczącym precyzji, lepiej więc posłużyć się wspomnianą wcześniej metodą bazującą na dopuszczalnej różnicy (epsilon). Ponadto przy użyciu wbudowanych funkcji `real` i `imag` możesz wyodrębnić część rzeczywistą i część urojoną liczby zespolonej. W pakiecie `math/cmplx` są też dostępne dodatkowe funkcje, ułatwiające działania na wartościach typu `complex128`.

Przypisanie wartości zerowej w przypadku obydwu typów liczb zespolonych powoduje nadanie wartości 0 zarówno rzeczywistej, jak i urojonej części tych liczb.

Listing 2.1 to prosty program demonstrujący obsługę liczb zespolonych. Możesz go uruchomić w serwisie Go Playground (*https://oreil.ly/fuyIu*). Kod tego programu znajdziesz także w katalogu *sample_code/complex_numbers* w repozytorium zawierającym materiały do książki (*https://github.com/learning-go-book-2e/ch02*).

Listing 2.1. Liczby zespolone

```
func main() {
    x := complex(2.5, 3.1)
    y := complex(10.2, 2)
    fmt.Println(x + y)
    fmt.Println(x - y)
    fmt.Println(x * y)
    fmt.Println(x / y)
    fmt.Println(real(x))
    fmt.Println(imag(x))
    fmt.Println(cmplx.Abs(x))
}
```

Uruchomienie tego kodu da następujący efekt:

```
(12.7+5.1i)
(-7.699999999999999+1.1i)
(19.3+36.62i)
(0.2934098482043688+0.246390225842280 65i)
2.5
3.1
3.982461550347975
```

Przykład ten ilustruje też nieprecyzyjność obliczeń zmiennoprzecinkowych.

Jeśli zastanawiasz się nad piątym rodzajem literałów prostych, podpowiem, że Go obsługuje literały urojone, reprezentujące urojoną część liczby zespolonej. Wyglądają one tak samo jak literały zmiennoprzecinkowe, tylko są zaopatrzone w przyrostek `i`.

Pomimo wbudowanego typu do obsługi liczb zespolonych Go nieczęsto wykorzystuje się do obliczeń naukowych. Zastosowanie Go na tym polu jest ograniczone ze względu na brak innych cech (na przykład obsługi macierzy), a dodające obsługę tych cech biblioteki muszą bazować na

nieefektywnych zamiennikach, takich jak wycinki wycinków. O wycinkach (ang. *slices*) będzie mowa w rozdziale 3. oraz w rozdziale 6., gdzie zapoznasz się z ich implementacją. Ale jeśli chcesz dokonać obliczeń na zbiorze Mandelbrota w ramach większego programu lub opracować narzędzie do rozwiązywania równań kwadratowych, możesz bez przeszkód skorzystać z możliwości Go w zakresie obliczeń na liczbach zespolonych.

Być może się zastanawiasz, dlaczego zdecydowano się zaimplementować w Go obsługę liczb zespolonych. Odpowiedź jest prosta: Ken Thompson, jeden z twórców Go (i Uniksa) uznał ją za interesujący dodatek (*https://oreil.ly/eBmkq*). Jakiś czas później dyskutowano o usunięciu tej funkcji w przyszłych wersjach Go (*https://oreil.ly/Q76EV*), lecz łatwiej jest ją po prostu zignorować.

Jeśli zależy Ci na pisaniu w Go aplikacji do obliczeń numerycznych, możesz użyć zewnętrznego pakietu Gonum (*https://www.gonum.org*). Wykorzystuje on obsługę liczb zespolonych i zawiera przydatne biblioteki ułatwiające wykonywanie obliczeń w zakresie algebry liniowej, macierzy, integracji i statystyki. Pomimo niewątpliwej przydatności tego pakietu w tego rodzaju przypadkach polecam rozważenie innych języków programowania.

Przedsmak informacji o łańcuchach i runach

W taki oto sposób dotarliśmy do informacji o łańcuchach znaków. Jak większość nowoczesnych języków programowania, Go oferuje wbudowany typ łańcuchowy. Wartość zerowa dla literału łańcuchowego to pusty łańcuch. Język Go obsługuje standard Unicode; jak już pokazywałem wcześniej, w części poświęconej literałom łańcuchowym, w łańcuchu można umieścić dowolny znak Unicode. Tak jak wartości całkowite i zmiennoprzecinkowe, łańcuchy można porównywać pod kątem identyczności przy użyciu operatora `==` oraz pod kątem różnic przy użyciu operatora `!=`. Do zadań związanych z szeregowaniem łańcuchów w określonej kolejności służą operatory `>`, `>=`, `<` i `<=`, a do konkatenacji (łączenia) — operator `+`.

Łańcuchy w Go są niemodyfikowalne (ang. *immutable*) w tym znaczeniu, że o ile da się przypisać nową wartość zmiennej łańcuchowej, o tyle już przypisanej wartości nie można zmienić.

Go oferuje też typ reprezentujący pojedynczą współrzędną kodową znaku (ang. *code point*). Typ `rune` jest odpowiednikiem typu `int32` na tej samej zasadzie, na jakiej typ `byte` jest odpowiednikiem `uint8`. Jak zapewne zgadujesz, domyślnym typem dla literału runicznego jest runa, a domyślnym typem dla literału łańcuchowego — łańcuch.

Jeśli odwołujesz się do znaku, użyj typu runicznego, a nie typu `int32`. Dla kompilatora mogą one oznaczać to samo, lecz w kodzie warto stosować typy, które jasno określają intencje programisty:

```
var myFirstInitial rune = 'J' // Dobrze - nazwa typu odpowiada jego zastosowaniu
var myLastInitial int32 = 'B' // Źle - pomimo poprawnej składni polecenie jest dezorientujące
```

O wiele więcej informacji o łańcuchach znajdziesz w następnym rozdziale — będzie w nim mowa o szczegółach ich implementacji, związkach z bajtami i runami oraz zaawansowanych cechach i często popełnianych błędach.

Jawna konwersja typów

Większość języków programowania oferujących wiele typów numerycznych dokonuje automatycznej konwersji między nimi w razie potrzeby. Proces ten nazywa się czasami *automatycznym rzutowaniem* (albo angielskim terminem *automatic type promotion*), a choć sprawia on wrażenie bardzo wygodnego, w praktyce okazuje się, że reguły umożliwiające prawidłową konwersję między typami bywają skomplikowane i niekiedy dają nieoczekiwane rezultaty. Jako język, którego twórcom przyświecały wartości takie jak przejrzystość intencji i czytelność kodu, Go nie pozwala na automatyczne rzutowanie typów między zmiennymi. W razie niezgodności typów musisz dokonać ich *jawnej konwersji*. Nawet w przypadku liczb całkowitych albo zmiennoprzecinkowych, które po prostu różnią się wielkością, trzeba dokonać konwersji ujednolicającej, aby dało się na nich działać. Pozwala to jasno pokazać, jakiego typu oczekujesz, bez konieczności zapamiętywania reguł konwersji (zob. listing 2.2).

Listing 2.2. Konwersje typów

```
var x int = 10
var y float64 = 30.2
var sum1 float64 = float64(x) + y
var sum2 int = x + int(y)
fmt.Println(sum1, sum2)
```

W tym przykładowym kodzie definiujemy cztery zmienne. Zmienna `x` jest typu `int` i ma wartość 10, a zmienna `y` jest typu `float64` i ma wartość 30,2. Ze względu na to, że typy te nie są tożsame, w celu ich dodania należy dokonać konwersji. W przypadku zmiennej `sum1` konwertujemy `x` na typ `float64` przy użyciu konwersji `float64`, a w przypadku zmiennej `sum2` konwertujemy `y` na typ `int` przy użyciu konwersji `int`. Uruchomienie tego kodu da wyniki 40,2 i 40.

Ten sam sposób działania zachodzi w przypadku typów całkowitych o odmiennej wielkości, jak pokazałem na listingu 2.3.

Listing 2.3. Konwersje typów całkowitych

```
var x int = 10
var b byte = 100
var sum3 int = x + int(b)
var sum4 byte = byte(x) + b
fmt.Println(sum3, sum4)
```

Możesz go uruchomić w serwisie Go Playground (*https://go.dev/play/p/498B-cz0CRv*). Kod tego programu znajdziesz także w katalogu *sample_code/type_conversion* w repozytorium zawierającym materiały do książki (*https://github.com/learning-go-book-2e/ch02*).

Ta specyficzna sztywność przestrzegania typów ma jeszcze inne następstwa. Ponieważ wszystkie konwersje typów w Go są jawne, nie da się potraktować dowolnego typu danych jako wartości boolowskiej. Mam tu na myśli fakt, że w wielu językach programowania wartość niezerowa albo niepusty łańcuch znaków są interpretowane jako logiczna prawda (`true`). Tak jak automatyczne rzutowanie typów, reguły dotyczące interpretowania wartości boolowskich różnią się w różnych językach i mogą być mylące. Nie powinno Cię zatem zaskoczyć, że Go nie dopuszcza takiego sprawdzania prawdziwości. Co więcej, *na typ boolowski nie da się skonwertować żadnego innego*

typu, jawnie czy nie. Jeśli chcesz dokonać pewnej formy przekształcenia innego typu danych na boolowski, użyj jednego z operatorów porównania (==, !=, >, <, <= lub >=). Na przykład aby sprawdzić, czy zmienna x jest równa 0, użyj wyrażenia `x == 0`. Jeśli z kolei chcesz sprawdzić, czy łańcuch jest pusty, użyj wyrażenia w rodzaju `s == ""`.

Konwersje typów są jednym z tych aspektów Go, w których twórcy języka postawili na przejrzystość i prostotę kosztem nieco większej rozwlekłości kodu. Takie kompromisy napotkasz jeszcze wielokrotnie. Podczas pisania idiomatycznego kodu w Go przedkłada się zrozumiałość nad zwięzłość.

Literały nie mają typu

Nie da się nawet dodać dwóch zmiennych całkowitych, jeśli zadeklarowano dla nich różne rozmiary. Język Go umożliwia jednak używanie literałów całkowitych w wyrażeniach zmiennoprzecinkowych, a nawet przypisywanie takich literałów zmiennej zmiennoprzecinkowej.

```
var x float64 = 10
var y float64 = 200.3 * 5
```

Dzieje się tak, ponieważ jak już wcześniej wspomniałem, literały w Go cechuje brak typowania. Literały w Go nie są typowane, gdyż język ten stawia na praktyczność rozwiązań. Unikanie wymuszania typu, zanim zadeklaruje go programista, jest logicznym wyjściem. Dlatego literały mogą wchodzić w interakcje z dowolną zmienną zgodną z danym literałem. W rozdziale 7., w którym zapoznasz się z typami definiowanymi przez użytkownika, przekonasz się, że możliwe jest stosowanie literałów wraz z typami zdefiniowanymi przez użytkownika na podstawie typów podstawowych. Brak typowania nie sięga jednak dalej — nie da się na przykład przypisać literału łańcuchowego zmiennej typu numerycznego (albo literału numerycznego do zmiennej łańcuchowej), niedopuszczalne jest też przypisywanie literału zmiennoprzecinkowego zmiennej całkowitej. Wszystkie takie operacje są traktowane przez kompilator jako błędy. Należy zarazem pamiętać o ograniczeniach dotyczących rozmiaru typów: choć da się wprowadzić w kodzie literały numeryczne większe niż można przechować w dowolnej zmiennej całkowitej, próba przypisania literału, którego wartość wykracza poza pojemność wskazanej zmiennej — na przykład przypisanie wartości 1000 zmiennej typu `byte` — spowoduje błąd na etapie kompilacji.

Słowo kluczowe var a operator :=

Jak na tak prosty język Go oferuje bardzo wiele form deklarowania zmiennych. Nie bez powodu: każdy styl deklaracji komunikuje coś na temat sposobu wykorzystania danej zmiennej. Przyjrzyj się wariantom deklarowania zmiennych w Go i zwróć uwagę, w jakich sytuacjach dany sposób jest odpowiedni.

Najpełniejszy sposób zadeklarowania zmiennej w Go opiera się na słowie kluczowym `var`, jawnym określeniu typu i przypisaniu wartości. Wygląda on tak:

```
var x int = 10
```

Jeśli typ wartości po prawej stronie znaku = jest oczekiwanym typem zmiennej, możesz pominąć nazwę typu po lewej stronie tego znaku. Ponieważ domyślnym typem literału całkowitego jest `int`, poniższy zapis deklaruje `x` jako zmienną typu `int`:

```
var x = 10
```

Na podobnej zasadzie, jeśli chcesz zadeklarować zmienną i przypisać jej wartość zerową, możesz zachować nazwę typu, a pominąć znak = oraz to, co po nim następuje:

```
var x int
```

Przy użyciu słowa kluczowego `var` da się zadeklarować kilka zmiennych naraz. Mogą one być tego samego typu, jak w następującym przypadku:

```
var x, y int = 10, 20
```

Mogą być tego samego typu i mieć wartość zerową:

```
var x, y int
```

Albo mogą być różnych typów:

```
var x, y = 10, "hello"
```

Słowa kluczowego `var` można użyć w jeszcze jeden sposób. Jeśli deklarujesz wiele zmiennych jednocześnie, możesz umieścić je na *liście deklaracji*:

```
var (
    x    int
    y        = 20
    z    int = 30
    d, e     = 40, "hello"
    f, g string
)
```

Go obsługuje też skrócony format deklaracji. W ciele funkcji możesz zastosować operator `:=` i w ten sposób pominąć słowo `var` oraz zastosować mechanizm inferencji typów. Poniższe dwie deklaracje mają identyczne znaczenie, a mianowicie deklarują zmienną `x` typu `int` o wartości 10:

```
var x = 10
x := 10
```

Podobnie jak w przypadku słowa `var`, przy użyciu operatora `:=` możesz zadeklarować kilka zmiennych. Poniższe dwa wiersze kodu przypisują wartość 10 zmiennej `x` oraz wartość `hello` zmiennej `y`:

```
var x, y = 10, "hello"
x, y := 10, "hello"
```

W odróżnieniu od słowa kluczowego `var` operator `:=` umożliwia wykonanie jeszcze jednej czynności, a mianowicie przypisania wartości istniejącej zmiennej. Jeśli tylko po lewej stronie operatora `:=` znajduje się co najmniej jedna nowa zmienna, pozostałe mogą być zmiennymi zadeklarowanymi już wcześniej:

```
x := 10
x, y := 30, "hello"
```

Stosowanie operatora `:=` wiąże się z jednym ograniczeniem. Otóż jeśli deklarujesz zmienną na poziomie pakietu, musisz użyć słowa kluczowego `var`, ponieważ użycie `:=` jest niedopuszczalne poza obrębem funkcji.

Jakimi zasadami należy się kierować przy wyborze stylu deklaracji? Jak zawsze, tak i w tym przypadku należy mieć na uwadze przede wszystkim jasność intencji. Najczęstszym stylem deklaracji w ciele funkcji jest użycie operatora `:=`. Poza funkcjami, w rzadkich przypadkach deklarowania wielu zmiennych na poziomie pakietu, użyj list deklaracji.

W pewnych sytuacjach należy unikać operatora `:=` nawet w ramach funkcji:

- W przypadku inicjalizowania zmiennej o zerowej wartości, użyj deklaracji takiej jak `var x int`. Dzięki temu jasne jest, iż wartość zerowa była zamierzona.
- W przypadku przypisywania nietypowanej stałej lub literału zmiennej, jeśli domyślny typ stałej lub literału jest niezgodny z oczekiwanym typem tej zmiennej, użyj rozszerzonej formy ze słowem `var` oraz jawną nazwą typu. Choć zasadniczo dopuszcza się w takich przypadkach stosowanie konwersji typu oraz operatora `:=`, co oznacza, że zapis `x := byte(20)` zadziała, prawidłowa forma deklaracji w takim przypadku ma postać `var x byte = 20`.
- Ponieważ operator `:=` umożliwia przypisywanie wartości nowym i istniejącym zmiennym, czasami powoduje on utworzenie nowych zmiennych w sytuacji, gdy wydaje Ci się, że wykorzystujesz te, które zadeklarowałeś już wcześniej (więcej informacji na ten temat znajdziesz w rozdziale 4., w punkcie „Przesłanianie zmiennych”). W takich przypadkach jawnie zadeklaruj wszystkie nowe zmienne przy użyciu słowa `var`, aby nie było wątpliwości, które z nich są nowe, a następnie użyj operatora przypisania (`=`), aby nadać wartości zarówno starym, jak i nowym zmiennym.

Choć zarówno `var`, jak i `:=` umożliwiają deklarowanie wielu zmiennych w jednym wierszu, styl ten należy stosować jedynie w przypadku przypisywania kilku wartości zwróconych przez funkcję bądź w ramach konstrukcji nazywanej *comma OK* (dosł. przecinek OK). Więcej informacji na ten temat znajdziesz w rozdziale 5. oraz w rozdziale 3., w punkcie „Idiom »comma OK«”.

Unikaj deklarowania zmiennych poza obrębem funkcji w ramach tak zwanego *bloku pakietu* (zob. podrozdział „Bloki” w rozdziale 4.). Deklarowanie na poziomie pakietu zmiennych, których wartość może ulegać modyfikacjom, nie jest najlepszym pomysłem. Zmienne zadeklarowane poza funkcjami przysparzają bowiem problemów ze śledzeniem wprowadzanych w nich modyfikacji, to zaś utrudnia zrozumienie przepływu danych w programie i stanowi żyzny grunt dla subtelnych błędów. Zasadniczo zmienne deklarowane w bloku pakietu powinny być niemodyfikowalne.

Unikaj deklarowania zmiennych poza funkcjami, bo komplikują one analizę przepływu danych.

Biorąc pod uwagę powyższe rozważania, możesz się zastanawiać, czy język Go oferuje sposób na *zagwarantowanie* niemodyfikowalności literału. Owszem, lecz mechanizm ten działa nieco inaczej niż sposoby być może znane Ci z innych języków programowania. Przyszła pora na zapoznanie się z konstrukcją `const`.

Zastosowanie słowa kluczowego const

Programiści uczący się nowego języka zwykle starają się odnieść doń znane im już koncepcje. Wiele języków oferuje rozwiązania dające gwarancję, że dana wartość będzie niemodyfikowalna. W języku Go służy do tego słowo kluczowe `const`. Na pierwszy rzut oka wydaje się ono działać identycznie jak w innych językach. Wypróbuj kod z listingu 2.4 w serwisie Go Playground (*https://oreil.ly/FdG-W*). Kod tego programu znajdziesz także w katalogu *sample_code/const_declaration* w repozytorium zawierającym materiały do książki (*https://github.com/learning-go-book-2e/ch02*).

Listing 2.4. Deklaracje const

```
package main

import "fmt"

const x int64 = 10

const (
    idKey   = "id"
    nameKey = "name"
)

const z = 20 * 10

func main() {
    const y = "hello"

    fmt.Println(x)
    fmt.Println(y)

    x = x + 1 // To polecenie się nie skompiluje!
    y = "bye" // To polecenie się skompiluje!
    fmt.Println(x)
    fmt.Println(y)
}
```

Próba uruchomienia tego kodu zakończy się niepowodzeniem, sygnalizowanym przez następujące błędy kompilacji:

```
./const.go:20:2: cannot assign to x (constant 10 of type int64)
./const.go:21:2: cannot assign to y (untyped string constant "hello")
```

Jak widać, stałe można deklarować na poziomie pakietu albo w ramach funkcji. Tak jak w przypadku słowa `var`, grupy powiązanych zmiennych można (i trzeba) deklarować na liście ujętej w nawiasy.

Możliwości deklaracji `const` w języku Go są jednak bardzo ograniczone. Stałe w Go są sposobem nadawania nazw literałom. Mogą one przechowywać wartości, które zostaną wykorzystane podczas kompilacji. Oznacza to, że da się im przypisywać:

- literały numeryczne,
- wartości logiczne (`true` i `false`),

- łańcuchy,
- runy,
- wartości zwracane przez wbudowane funkcje `complex`, `real`, `imag`, `len` i `cap`,
- wyrażenia składające się z operatorów i poprzedzających je wartości.

Funkcje `len` i `cap` omówię w następnym rozdziale. Jest jeszcze jedna wartość, której można użyć w deklaracji `const` — nosi ona nazwę `iota`. Będzie o niej mowa w rozdziale 7., przy okazji opisywania sposobów tworzenia własnych typów danych.

Go nie oferuje sposobu deklarowania niemodyfikowalności wartości obliczonej w czasie działania programu. Na przykład zamieszczony tutaj fragment kodu nie zostanie skompilowany i spowoduje wygenerowanie komunikatu błędu `x + y (value of type int) is not constant`:

```
x := 5
y := 10
const z = x + y // To polecenie się nie skompiluje!
```

W następnym rozdziale przeczytasz o tym, że nie ma niemodyfikowalnych tablic, wycinków, map ani struktur; nie da się też zadeklarować niemodyfikowalności pola w strukturze. Wymienione ograniczenia nie są jednak tak drastyczne, jak się zdaje. W obrębie funkcji jasne jest, czy dana zmienna ulega modyfikacji, kwestia jej niemodyfikowalności jest więc mniej ważna. W rozdziale 5. (podrozdział „W Go obowiązuje wywołanie przez wartość") możesz przeczytać o tym, jak Go zapobiega modyfikowaniu zmiennych przekazywanych do funkcji jako argumenty.

Stałe w języku Go stanowią sposób nadawania nazw literałom. W języku tym *nie da się* zadeklarować niemodyfikowalności zmiennej.

Stałe typowane i nietypowane

Stałe mogą być typowane albo nietypowane. Nietypowana stała działa dokładnie tak jak literał: nie ma ona własnego typu, lecz ma typ domyślny, wykorzystywany w sytuacji, gdy żaden inny nie wynika z konstrukcji kodu. Stała typowana może być bezpośrednio przypisana jedynie zmiennej tego samego typu.

Decyzja o zadeklarowaniu typu stałej jest uzależniona od przyczyny deklaracji. Jeśli chcesz nadać nazwę stałej matematycznej, której będzie można następnie używać z różnymi typami numerycznymi, nie deklaruj typu. Zasadniczo pozostawienie stałej jako nietypowanej daje większą elastyczność działania. Są jednak przypadki, w których wymuszenie typu stałej ma sens. Z typowanych stałych będziemy korzystać przy tworzeniu typów wyliczeniowych z użyciem słowa `iota` (zob. punkt „Iota służy do tworzenia typów wyliczeniowych… czasami" w rozdziale 7.)

Tak wygląda przykład deklaracji stałej nietypowanej:

```
const x = 10
```

Wszystkie poniższe przypisania są dozwolone:

```
var y int = x
var z float64 = x
var d byte = x
```

Tak wygląda przykład deklaracji stałej typowanej:

```
const typedX int = 10
```

Taką stałą można przypisywać tylko do typu `int`. Przypisanie jej do dowolnego innego typu spowoduje na etapie kompilacji błąd podobny do poniższego:

```
cannot use typedX (type int) as type float64 in assignment
```

Niewykorzystane zmienne

Jednym z celów języka Go jest ułatwienie dużym zespołom wspólnej pracy nad programami. W związku z tym w Go określono kilka zasad rzadko spotykanych w innych językach programowania. W rozdziale 1. napisałem, że kod w Go musi być sformatowany w konkretny sposób przy użyciu polecenia `go fmt`, aby ułatwić pisanie narzędzi przetwarzających ten kod i zachować zgodność ze standardami. Inny wymóg polega na tym, by *każda zadeklarowana zmienna została odczytana*. Zadeklarowanie zmiennej i niewykorzystanie jej wartości powoduje *błąd kompilacji*.

Weryfikacja wykorzystania zmiennej podczas kompilacji nie jest wyczerpująca. Wystarczy, by zmienna została odczytana raz — kompilatorowi to wystarczy, nawet jeśli późniejsze modyfikacje tej zmiennej nie zostaną faktycznie użyte w programie. Poniższy program w Go jest więc prawidłowy i da się go uruchomić w serwisie Go Playground (*https://oreil.ly/8JLA6*). Kod tego programu znajdziesz także w katalogu *sample_code/assignments_not_read* w repozytorium zawierającym materiały do książki (*https://github.com/learning-go-book-2e/ch02*).

```
func main() {
    x := 10  // To przypisanie nie zostało odczytane!
    x = 20
    fmt.Println(x)
    x = 30 // To przypisanie nie zostało odczytane!
}
```

Ani kompilator, ani polecenie `go vet` nie wychwycą niewykorzystanych działań na zmiennej `x`, polegających na przypisaniu jej wartości 10 i 30. Więcej informacji na temat wymienionych narzędzi znajdziesz w rozdziale 11.

Kompilator Go nie przeszkodzi Ci w tworzeniu niewykorzystanych zmiennych, jeśli zostaną one zadeklarowane na poziomie pakietu. To następny powód, dla którego warto unikać zmiennych tego rodzaju.

Niewykorzystane stałe

Pewnym zaskoczeniem może być fakt, że kompilator Go umożliwia tworzenie niewykorzystanych stałych przy użyciu słowa kluczowego `const`. Dzieje się tak dlatego, że stałe w Go są obliczane na etapie kompilacji, a ich istnienie nie ma żadnych skutków ubocznych. Dzięki temu łatwo je wyeliminować: jeśli jakaś stała nie została użyta, po prostu nie trafia do skompilowanego pliku binarnego.

Nazywanie zmiennych i stałych

Istnieje różnica między obowiązującymi w Go zasadami nazywania zmiennych oraz sposobami nazywania zmiennych i stałych stosowanymi przez wielu programistów. Podobnie jak w większości języków programowania, w Go obowiązuje wymóg zaczynania nazw od litery lub podkreślenia; sama nazwa może zawierać cyfry, podkreślenia i litery. Definicja „litery" i „cyfry" w Go jest jednak nieco szersza: dopuszcza ona stosowanie dowolnego znaku Unicode, który jest uważany za literę lub cyfrę. Oznacza to, że w Go prawidłowe są wszystkie definicje zmiennych z listingu 2.5:

Listing 2.5. Nadawanie zmiennym nazw, których... nigdy nie należy używać

```
_0 := 0_0
_𝟙 := 20
π := 3
 a  := "hello" // Unicode U+FF41
__ := "double underscore" // Dwa znaki podkreślenia
fmt.Println(_0)
fmt.Println(_𝟙)
fmt.Println(π)
fmt.Println( a )
fmt.Println(__)
```

Ten okropny kod możesz przetestować w serwisie Go Playground (*https://go.dev/play/p/ySKtwUtPdk6*). Choć kod jest prawidłowy, *nie zalecam* stosowania tego rodzaju nazewnictwa. O takich nazwach mówi się, że są nieidiomatyczne, ponieważ łamią podstawową zasadę dbałości o czytelność kodu. Nazwy te są mylące albo trudne do wprowadzenia na większości klawiatur. „Podstępne" są zwłaszcza te znaki Unicode, które swoim wyglądem bardzo przypominają znaki alfabetu łacińskiego — nawet jeśli wydaje się nam, że mamy do czynienia z tym samym znakiem, może się okazać, że chodzi o zupełnie różne zmienne. Kod z listingu 2.6 możesz uruchomić w Go Playground (*https://oreil.ly/hrvb6*). Znajdziesz go również w katalogu *sample_code/look_alike_code_points* w repozytorium zawierającym materiały do książki (*https://github.com/learning-go-book-2e/ch02*).

Listing 2.6. Zastosowanie znaków o podobnym wyglądzie w nazwach zmiennych

```
func main() {
     a  := "hello"    // Unicode U+FF41
    a := "goodbye" // Standardowa mała litera a (Unicode U+0061)
    fmt.Println( a )
    fmt.Println(a)
}
```

Po uruchomieniu tego programu uzyskasz następujący efekt:

```
hello
goodbye
```

Chociaż dopuszcza się używanie znaku podkreślenia w nazwach zmiennych, jest on rzadko stosowany, bo pisząc idiomatyczny kod w Go, należy unikać zapisów w rodzaju `licznik_pierwszy` czy `liczba_prob` (tzw. notacja *snake case*). W przypadku identyfikatorów składających się z kilku słów należy stosować zapis typu *camel case* — `licznikPierwszy` czy `liczbaProb`.

Sam znak podkreślenia (_) jest specjalnym identyfikatorem w Go; będzie o nim mowa w rozdziale 5., poświęconym funkcjom.

W wielu językach nazwy stałych zapisuje się wersalikami (samymi wielkimi literami), a poszczególne słowa oddziela się podkreśleniem (`LICZNIK_PIERWSZY` czy `LICZBA_PROB`). Unikaj stosowania takiego zapisu, ponieważ Go na podstawie wielkości pierwszej litery w nazwie deklaracji umieszczonej na poziomie pakietu określa, czy dany element ma być dostępny poza tym pakietem. Wrócimy do tego tematu w rozdziale 10., przy okazji omawiania pakietów.

W ramach funkcji warto stosować jak najkrótsze nazwy zmiennych. *Im mniejszy zasięg jakiejś zmiennej, tym krótsza powinna być jej nazwa.* W kodzie w Go bardzo często spotyka się nazwy jednoznakowe. Na przykład w pętlach `for-range` powszechnie stosuje się nazwy `k` oraz `v`, pochodzące od angielskich słów *key* i *value*. Z kolei liczniki w typowych pętlach `for` bardzo często nazywa się `i` oraz `j`. Są inne idiomatyczne metody nazywania często używanych zmiennych; będę o nich wspominał przy okazji omawiania kolejnych aspektów biblioteki standardowej.

W niektórych językach ze słabszym typowaniem zachęca się programistów do umieszczania w nazwie zmiennej jej oczekiwanego typu. Ponieważ język Go jest silnie typowany, nie musisz stosować takich trików, aby ułatwić sobie dostrzeganie typu zmiennych. Funkcjonują jednak pewne ogólne zasady dotyczące jednoliterowych nazw zmiennych różnych typów. Programiści często stosują na przykład pierwszą literę nazwy typu w charakterze nazwy zmiennej tego typu — na przykład `i` w przypadku zmiennych całkowitych (*integer*), `f` w przypadku wartości zmiennoprzecinkowych (*floats*) czy `b` dla wartości boolowskich. Podobne rozwiązania dotyczą typów własnych, głównie w zakresie nadawania nazw zmiennym odbiorców (to zagadnienie dokładnie omówię w rozdziale 7.).

Te krótkie nazwy służą dwóm celom. Po pierwsze, pozwalają uniknąć uciążliwego pisania i sprawiają, że kod jest krótszy. Po drugie, pozwalają ocenić poziom komplikacji kodu. Jeśli zauważysz, że zaczynasz z trudem panować nad znaczeniem zmiennych o krótkich nazwach, to niewykluczone, że dany blok kodu po prostu robi za dużo.

W przypadku zmiennych i stałych w bloku pakietu lepiej używać opisowych nazw. Nie zalecam umieszczania w nich nazwy typu, lecz ze względu na szerszy zakres takich zmiennych i stałych warto posłużyć się obszernymi nazwami, nie pozostawiającymi wątpliwości co do ich znaczenia.

Więcej informacji na temat zaleceń związanych z nadawaniem nazw w Go znajdziesz w sekcji *Naming* opracowanego przez Google dokumentu *Go Style Decisions* (*https://google.github.io/styleguide/go/decisions#naming*).

Ćwiczenia

Zamieszczone tutaj ćwiczenia pozwalają wypróbować koncepcje omówione w rozdziale. Odpowiedzi do tych ćwiczeń znajdziesz w repozytorium pod adresem *https://ftp.helion.pl/przyklady/jegot2.zip*.

1. Utwórz program deklarujący zmienną `i` typu liczby całkowitej o wartości `20`. Przypisz ją zmiennej `f` typu liczby zmiennoprzecinkowej. Następnie wyświetl wartości obu zmiennych.
2. Utwórz program deklarujący stałą o nazwie `value`, którą można przypisać zmiennej typu liczby całkowitej lub liczby zmiennoprzecinkowej. Przypisz ją zmiennej liczby całkowitej o nazwie `i` oraz zmiennej liczby zmiennoprzecinkowej o nazwie `f`. Następnie wyświetl wartości obu zmiennych.
3. Utwórz program deklarujący trzy zmienne: `b` typu `byte`, `smallI` typu `int32` i `bigI` typu `uint64`. Poszczególnym zmiennym przypisz maksymalną wartość dozwoloną dla danego typu, a następnie dodaj do niej `1`. Wyświetl wartości wszystkich zmiennych.

Podsumowanie

W tym rozdziale omówiłem obszerny zakres zagadnień: jak używać wbudowanych typów danych, deklarować zmienne oraz stosować przypisania i operatory. W następnym rozdziale przyjrzysz się dostępnym w Go typom złożonym: tablicom, wycinkom, mapom i strukturom. Wrócę w nim też do kwestii zmiennych łańcuchowych i runicznych oraz poruszę kwestię kodowania znaków.

Książka rzeczowo wyjaśnia najważniejsze cechy języka i omawia dobre wzorce projektowe

Aaron Schlesinger, starszy inżynier, Microsoft

Go bardzo szybko upowszechnił się wśród twórców usług sieciowych. Jednak zaznajomienie się z jego składnią nie wystarczy programistom, którzy używają innych języków. Poprzestanie na takiej pobieżnej nauce powoduje nieświadome stosowanie wzorców, które w kontekście Go nie mają sensu. Aby skorzystać w pełni z potencjału tego języka, trzeba się nauczyć pisać idiomatyczny kod.

Niezależnie od poziomu doświadczenia ten praktyczny przewodnik ułatwi Ci opanowanie Go. Znajdziesz tu kluczowe informacje, dzięki którym nauczysz się tworzyć przejrzysty, idiomatyczny kod w Go i myśleć jak programista Go. To wydanie uwzględnia nowości, które zostały udostępnione w ciągu ostatnich trzech lat: nowe funkcjonalności, narzędzia i biblioteki. Wyjaśniono tu stosowanie takich usprawnień jak strukturalne rejestrowanie danych, fuzzing, przestrzenie robocze i sprawdzanie pod kątem luk w zabezpieczeniach. Dokładniej opisano ekosystem narzędzi Go i wzbogacono to omówienie ćwiczeniami i przykładami. Jeśli chcesz pisać idiomatyczny kod Go, który będzie niezawodny, trwały i łatwy w późniejszej obsłudze technicznej — to książka dla Ciebie!

W książce:

- idiomatyczny kod Go
- przygotowanie środowiska programistycznego Go
- stosowanie refleksji oraz modułów unsafe i cgo
- zapewnianie efektywnego działania kodu
- optymalizacja użycia pamięci
- zaawansowane narzędzia programistyczne Go

Jon Bodner jest inżynierem oprogramowania i architektem z dwudziestokilkuletnim doświadczeniem. Zajmował się tworzeniem i rozwijaniem aplikacji w wielu dziedzinach, między innymi w finansach, handlu, ochronie zdrowia i administracji. Obecnie pracuje w firmie Datadog. Często występuje na konferencjach dotyczących języka Go. Jest też autorem kilku bibliotek i aplikacji.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1476-6
9 788328 914766
Cena: 99,00 zł