

Rusz głową!

Go



Naucz się pisać
prosty i łatwy
w konserwacji kod

Unikaj
żenujących
błędów
związanych
z typami



Wysil głowę,
rozwiązując
ponad
40 ćwiczeń
z zakresu
języka Go



Przewodnik
po programowaniu
w Go dla uczących się
tego języka



Skoncentruj się
na funkcjach,
dzięki którym
zmaksyma-
lizujesz swoją
produktywność

Uruchamiaj
funkcje
współbieżnie
za pomocą
wątków
goroutine



Tytuł oryginału: Head First Go

Tłumaczenie: Radosław Lesisz i Tomasz Walczak

ISBN: 978-83-283-6152-2

© 2020 Helion SA

Authorized Polish translation of the English edition of Head First Go

ISBN 9781491969557 © 2019 Jay McGavren

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/gorugl.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/gorugl>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści (podsumowanie)

	Wprowadzenie	xxv
1	Zaczynamy. <i>Podstawy składni</i>	1
2	Jaki kod uruchomić w następnej kolejności? <i>Instrukcje warunkowe i pętle</i>	31
3	Wywołaj mnie. <i>Funkcje</i>	79
4	Pakiety kodu. <i>Pakiety</i>	113
5	Na liście. <i>Tablice</i>	149
6	Problem dołączania elementów. <i>Wycinki</i>	175
7	Nazywanie danych. <i>Mapy</i>	205
8	Tworzenie struktur do przechowywania danych. <i>Struktury</i>	231
9	Jesteś w moim typie. <i>Typy zdefiniowane</i>	265
10	Zachowaj to dla siebie. <i>Hermetyzacja i zagnieżdżanie</i>	289
11	Co potrafisz zrobić? <i>Interfejsy</i>	321
12	Znów stań na nogi. <i>Przywracanie stanu po awarii</i>	349
13	Udostępnianie kodu. <i>Wątki goroutine i kanały</i>	379
14	Kontrola jakości kodu. <i>Testy zautomatyzowane</i>	401
15	Reagowanie na żądania. <i>Aplikacje internetowe</i>	425
16	Stosowanie szablonów. <i>Szablony HTML</i>	445
A	Zrozumieć funkcję os.OpenFile. <i>Otwieranie plików</i>	481
B	Sześć kwestii, które nie zostały opisane. <i>Pozostałości</i>	495

Spis treści (z prawdziwego zdarzenia)

Wprowadzenie

Twój mózg myśli o Go. Choć Ty starasz się czegoś *nauczyć*, Twój mózg robi Ci przysługę, dbając o to, by te informacje się *nie utrwały*. Twój mózg myśli sobie: „Lepiej zostawię miejsce na coś naprawdę ważnego, na przykład: jakich dzikich zwierząt lepiej unikać albo dlaczego jeżdżenie na snowboardzie nago nie jest najlepszym pomysłem”. A zatem jak możesz *oszukać* swój mózg, by myślał, że Twoje życie zależy od nauczenia się programowania w Go?

	Dla kogo jest przeznaczona ta książka?	xxvi
	Wiemy, co sobie myślisz	xxvii
	Wiemy, co sobie myśli Twój mózg	xxvii
	Metapoznanie — myślenie o myśleniu	xxix
	Oto co zrobiliśmy	xxx
	Przeczytaj to	xxxii
	Podziękowania	xxxiii


Zaczynamy

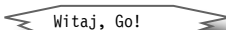
Podstawy składni


1

Czy jesteś gotów na turbodoładowanie swojego oprogramowania? Czy szukasz prostego języka programowania, który **szybko się kompiluje** i **szybko działa**? Który umożliwia **łatwą dystrybucję** programów do użytkowników? Jeśli tak, to **jesteś gotów na Go!**

Go to język programowania, w którym nacisk jest położony na **prostotę** i **szybkość**. Jest on prostszy od innych języków, dlatego jego nauka przebiega szybciej. Ponadto Go pozwala wykorzystać możliwości nowoczesnych procesorów wielordzeniowych, dzięki czemu programy działają szybciej. W tym rozdziale poznasz wszystkie mechanizmy języka Go, które **ułatwiają pracę programistom i zwiększają zadowolenie użytkowników**.

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

 Witaj, Go!

 Witaj, Go!

 1 + 2

 3

 4 < 6

 true

 1 1

 1174

Do biegu, gotowi, Go!	2
Narzędzie Go Playground	3
Co to wszystko znaczy?	4
A jeśli coś się nie powiedzie?	5
Wywoływanie funkcji	7
Funkcja Println	7
Używanie funkcji z innych pakietów	8
Wartości zwracane przez funkcje	9
Szablon programu w Go	11
Łańcuchy znaków	11
Runy	12
Wartości logiczne	12
Liczby	13
Operacje matematyczne i porównania	13
Typy	14
Deklarowanie zmiennych	16
Wartości zerowe	17
Krótkie deklaracje zmiennych	19
Reguły tworzenia nazw	21
Konwersje	22
Instalowanie Go na komputerze	25
Kompilowanie kodu w języku Go	26
Narzędzia języka Go	27
Szybkie sprawdzanie działania kodu za pomocą polecenia go run	27
Twój przyborek do Go	28

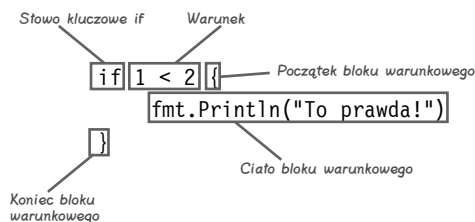
Jaki kod uruchomić w następnej kolejności?

2

Instrukcje warunkowe i pętle

Każdy program zawiera fragmenty wykonywane tylko w określonych sytuacjach. „Ten kod należy uruchomić, *jeśli* wystąpi błąd. W przeciwnym razie należy uruchomić inny kod”. Prawie każdy program zawiera kod, który należy wykonywać tylko wtedy, gdy spełniony jest określony *warunek*. Dlatego prawie każdy język programowania udostępnia **instrukcje warunkowe**, pozwalające ustalić, czy uruchamiać dane fragmenty kodu. Go nie jest tu wyjątkiem.

Możliwe też, że zechcesz *wielokrotnie* wykonywać określone porcje kodu. Go, podobnie jak większość języków, udostępnia **pętle** wykonujące fragmenty kodu więcej niż raz. W tym rozdziale nauczysz się korzystać zarówno z instrukcji warunkowych, jak i z pętli.



Wywoływanie metod	32
Obliczanie oceny	34
Funkcje i metody zwracające wiele wartości	36
Rozwiązanie 1. Zignorować wartość błędu za pomocą pustego identyfikatora	37
Rozwiązanie 2. Obsługa błędu	38
Instrukcje warunkowe	39
Warunkowe rejestrowanie błędów krytycznego	42
Unikaj zakrywania nazw	44
Przekształcanie łańcuchów znaków na liczby	46
Bloki	49
Bloki i zasięg zmiennych	50
Program do wystawiania ocen jest gotowy!	52
Tylko jedna zmienna w krótkiej deklaracji zmiennej musi być nowa	54
Napiszmy grę	55
Nazwy pakietów a ścieżki importowania	56
Generowanie liczby losowej	57
Pobieranie liczby całkowitej z klawiatury	59
Porównywanie wytypowanej liczby z docelową	60
Pętle	61
Instrukcje inicjalizacji i instrukcje wykonywane po iteracji są opcjonalne	63
Używanie pętli w grze w zgadywanie liczb	66
Wychodzenie z pętli zgadywania liczby	69
Ujawnianie docelowej liczby	70
Gratulacje, gra jest kompletna!	72
Twój przybornik do Go	74

Wywołaj mnie

3

Funkcje

Pewnie tego nie zauważyłeś, ale wywoływałeś funkcje jak zawodowiec.

Jednak mogłeś używać jedynie funkcji, które są zdefiniowane w języku Go. Teraz kolej na Ciebie. Zobaczysz, jak tworzyć własne funkcje. Nauczysz się deklarować funkcje z parametrami i bez nich. Zadeklarujesz funkcje zwracające pojedyncze wartości, a także dowiesz się, jak zwracać kilka wartości, aby informować o błędach. Poznasz też **wskaźniki** umożliwiające wywoływanie funkcji w sposób oszczędzający pamięć.



Powtarzający się kod	80
Formatowanie danych wyjściowych z użyciem funkcji Printf i Sprintf	81
Instrukcje formatowania	82
Formatowanie długości wartości	83
Formatowanie długości liczb ułamkowych	84
Używanie funkcji Printf w kalkulatorze ilości farby	85
Deklarowanie funkcji	86
Deklarowanie parametrów funkcji	87
Używanie funkcji w kalkulatorze ilości farby	88
Funkcje i zasięg zmiennych	90
Wartości zwracane przez funkcje	91
Używanie zwracanej wartości w kalkulatorze ilości farby	93
W funkcji paintNeeded potrzebna jest obsługa błędów	95
Wartości błędów	96
Deklarowanie wielu zwracanych wartości	97
Używanie wielu zwracanych wartości w funkcji paintNeeded	98
Zawsze obsługuj błędy!	99
W parametrach funkcji zapisywane są kopie argumentów	102
Wskaźniki	103
Typy wskaźnikowe	104
Pobieranie lub modyfikowanie wartości wskazywanej przez wskaźnik	105
Używanie wskaźników w funkcjach	107
Poprawianie funkcji double z użyciem wskaźników	108
Twój przybornik do Go	110

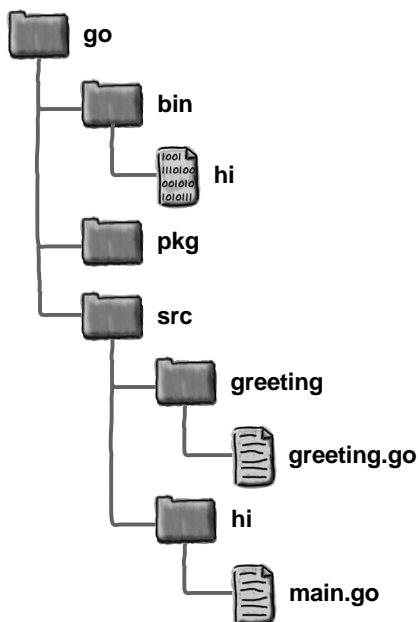
Pakiety kodu

4

Pakiety

Pora zadbać o porządek. Do tej pory zapisywałeś cały kod w jednym pliku. Gdy zaczniesz pisać większe i bardziej złożone programy, szybko doprowadzi to do chaosu.

W tym rozdziale zobaczysz, jak tworzyć własne **pakiety**, które pomagają przechowywać powiązany ze sobą kod w jednym miejscu. Pakiety są przydatne nie tylko do porządkowania kodu. Są też łatwym sposobem na *współużytkowanie kodu między programami*. Są też *programami* i wygodnym narzędziem do *udostępniania kodu innym programistom*.



Różne programy, ta sama funkcja	114
Współdzielenie kodu w programach z użyciem pakietów	116
Kod pakietów jest przechowywany w obszarze roboczym języka Go	117
Tworzenie nowego pakietu	118
Importowanie pakietów do programu	119
W pakietach używany jest ten sam układ plików	120
Konwencje tworzenia nazw pakietów	123
Kwalifikatory w postaci nazw pakietów	123
Przenoszenie wspólnego kodu do pakietu	124
Stałe	126
Zagnieżdżone katalogi pakietów i ścieżki importowania	128
Instalowanie programów wykonywalnych za pomocą instrukcji go install	130
Modyfikowanie obszaru roboczego za pomocą zmiennej środowiskowej GOPATH	131
Ustawianie zmiennej GOPATH	132
Publikowanie pakietów	133
Pobieranie i instalowanie pakietów z użyciem polecenia go get	137
Wczytywanie dokumentacji pakietu za pomocą polecenia go doc	139
Dokumentowanie pakietów z użyciem komentarzy narzędzia doc	141
Wyświetlanie dokumentacji w przeglądarce internetowej	143
Udostępnianie dokumentacji w formacie HTML na swoje potrzeby za pomocą narzędzia godoc	144
Serwer narzędzia godoc udostępnia też TWOJE pakiety!	145
Twój przybornik do Go	146

Na liście

5

Tablice

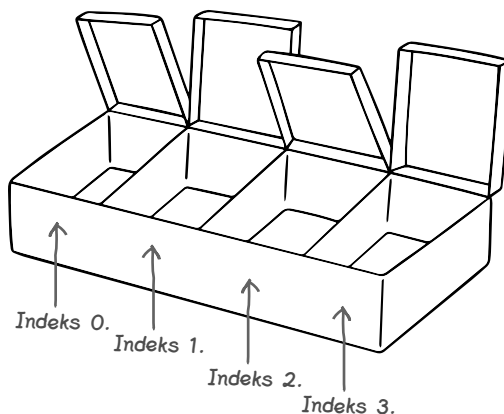
Wiele programów korzysta z list różnych rzeczy: listy adresów, listy numerów telefonów, listy produktów. Go udostępnia *dwa* wbudowane mechanizmy przechowywania list. W tym rozdziale poznasz pierwszy z nich — **tablice**. Dowiesz się, jak je tworzyć, jak wypełniać danymi i jak ponownie pobierać dane. Dalej dowiesz się, jak przetwarzać wszystkie elementy tablicy — najpierw w *trudny* sposób, z użyciem pętli `for`, a następnie *łatwą* metodą, za pomocą pętli `for...range`.

Tablice przechowują kolekcje wartości	150
Wartości zerowe w tablicach	152
Literały tablicowe	153
Funkcje z pakietu <code>fmt</code> potrafią obsługiwać tablice	154
Dostęp do elementów tablicy w pętli	155
Sprawdzanie długości tablicy za pomocą funkcji <code>len</code>	156
Bezpieczne przetwarzanie tablic w pętli za pomocą instrukcji <code>for...range</code>	157
Używanie pustego identyfikatora w pętlach <code>for...range</code>	158
Obliczanie sumy liczb z tablicy	159
Pobieranie średniej liczb z tablicy	161
Wczytywanie pliku tekstowego	163
Wczytywanie pliku tekstowego do tablicy	166
Modyfikowanie programu <code>average</code> , aby czytywał plik tekstowy	168
Nasz program potrafi przetwarzać tylko trzy wartości!	170
Twój przybornik do Go	172

Liczba elementów przechowywanych w tablicy

Typ elementów zapisywanych w tablicy

```
var myArray [4]string
```



Problem dołączania elementów

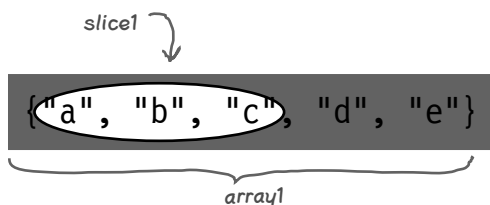
6

Wycinki

Wiesz już, że nie można dołączać dodatkowych elementów do tablicy. Stanowi to poważny problem w programie, ponieważ nie wiadomo z góry, ile elementów danych będzie on zawierać. W tym kontekście pomocne są **wycinki** z języka Go. Wycinki to kolekcje, które mogą się rozrastać, aby pomieścić dodatkowe elementy. Właśnie to jest potrzebne, aby ulepszyć bieżący program! Zobaczysz też, że wycinki ułatwiają użytkownikom przekazywanie danych do *wszystkich* programów i pomagają programistom pisać łatwiejsze w wywoływaniu funkcje.



Wycinki	176
Literały wycinków	177
Operator wycinka	180
Tablice podstawowe	182
Modyfikowanie podstawowych tablic i wycinków	183
Dodawanie elementów do wycinka za pomocą funkcji append	184
Wycinki i wartości zerowe	186
Wczytywanie dodatkowych wierszy plików z użyciem wycinków i funkcji append	187
Sprawdzanie poprawionego programu	189
Zwracanie wycinka nil po wystąpieniu błędu	190
Argumenty wiersza poleceń	191
Pobieranie argumentów wiersza poleceń z wycinka os.Args	192
Operator wycinka można stosować do innych wycinków	193
Modyfikowanie programu, aby używał argumentów wiersza poleceń	194
Funkcje wariadyczne	195
Stosowanie funkcji wariadycznych	197
Użycie funkcji wariadycznej do obliczania średnich	198
Przekazywanie wycinków do funkcji wariadycznych	199
Wycinki uratowały sytuację!	201
Twój przybornik do Go	202



7

Nazywanie danych

Mapy

Rzucanie rzeczy na stertę sprawdza się, dopóki nie musisz czegoś znaleźć.

Zobaczyłeś już, jak tworzyć listy wartości z użyciem *tablic* i *wycinków*. Wiesz też, jak zastosować tę samą operację do *każdej wartości* tablicy lub wycinka. Co jednak zrobić, jeśli chcesz użyć *określonej wartości*? Aby ją znaleźć, musisz zacząć od początku tablicy lub wycinka i *sprawdzić każdą jedną wartość*.

A gdyby utworzyć kolekcję, w której każda wartość ma nazwę? Można byłoby szybko znaleźć tę wartość, która jest potrzebna! W tym rozdziale poznasz **mapy**, które służą właśnie do tego.

Klucze umożliwiają
szybkie odnajdowanie
danych!



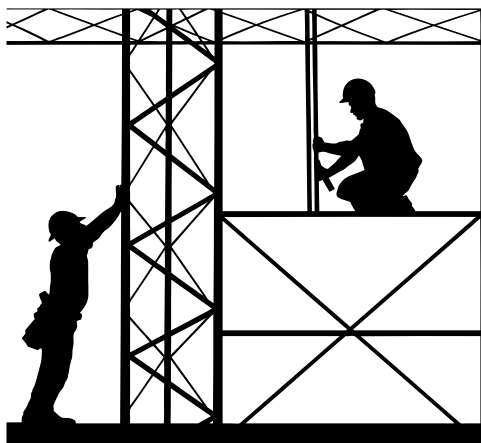
Zliczanie głosów	206
Wczytywanie nazwisk z pliku	207
Zliczanie nazwisk w trudny sposób, z użyciem wycinków	209
Mapy	212
Literały map	214
Wartości zerowe w mapach	215
Wartość zerowa zmiennej reprezentującej mapę to nil	215
Jak odróżnić wartości zerowe od przypisanych?	216
Usuwanie par klucz – wartość za pomocą funkcji delete	218
Modyfikowanie programu zliczającego głosy, aby użyć map	219
Używanie pętli for...range do map	221
Pętla for...range przetwarza mapy w losowej kolejności!	223
Modyfikowanie programu zliczającego głosy — zastosowanie pętli for...range	224
Program do zliczania głosów jest kompletny!	225
Twój przybornik do Go	227

Tworzenie struktur do przechowywania danych

Struktury

8

Czasem chcesz przechowywać wartości różnych typów danych. Znasz już wycinki, które przechowują listy wartości. Następnie poznałeś mapy, które łączą listy kluczy z listami wartości. Jednak obie te struktury danych potrafią przechowywać tylko wartości *jednego* typu. Czasem trzeba połączyć wartości *kilku* typów. Pomyśl o rachunkach, na których podawane są nazwy produktów (łańcuchy znaków) i ich ceny (liczby całkowite), albo o wynikach studentów, gdzie obok nazwisk (łańcuchów znaków) przechowywane są średnie ocen (liczby zmiennoprzecinkowe). W wycinkach i mapach nie można łączyć wartości różnych typów. *Można* jednak łączyć różne typy w innym typie — w **strukturze**. W tym rozdziale dowiesz się wszystkiego na temat struktur!



Wycinki i mapy przechowują wartości JEDNEGO typu	232
Struktury są tworzone z wartości WIELU typów	233
Dostęp do pól struktury za pomocą operatora kropki	234
Zapisywanie danych prenumeratorów w strukturze	235
Typy zdefiniowane i struktury	236
Używanie typu zdefiniowanego na dane prenumeratorów magazynu	238
Używanie typów zdefiniowanych razem z funkcjami	239
Modyfikowanie struktury z użyciem funkcji	242
Dostęp do pól struktur za pomocą wskaźnika	244
Przekazywanie dużych struktur z użyciem wskaźników	246
Przenoszenie typu struktury do innego pakietu	248
Nazwa zdefiniowanego typu musi zaczynać się wielką literą, aby została wyeksportowana	249
Nazwy pól struktury muszą zaczynać się wielką literą, aby zostały wyeksportowane	250
Literały struktur	251
Tworzenie struktury typu Employee	253
Tworzenie struktury typu Address	254
Dodawanie struktury jako pola w innym typie	255
Podawanie wartości struktury w innej strukturze	255
Anonimowe pola struktur	258
Zagnieżdżanie struktur	259
Typy zdefiniowane są gotowe!	260
Twój przybornik do Go	261

Jesteś w moim typie

9

Typy zdefiniowane

Nie wiesz jeszcze wszystkiego o typach zdefiniowanych. W poprzednim rozdziale pokazałem, jak zdefiniować typ, którego typem bazowym jest struktura. *Nie* pokazałem jednak, że jako typ bazowy można wykorzystać *dowolny* typ.

Pamiętasz metody — specjalne funkcje powiązane z wartościami określonego typu?

Wywoływaliśmy już metody dla różnych wartości, nie wiesz jednak, jak definiować *własne* metody.

W tym rozdziale nadrobisz te zaległości. Pora zaczynać!

Błędne typy w rzeczywistym życiu	266
Typy zdefiniowane z prostymi typami bazowymi	267
Typy zdefiniowane i operatory	269
Przekształcenia między typami z użyciem funkcji	271
Rozwiązywanie kolizji nazw funkcji z użyciem metod	274
Definiowanie metod	275
Parametr odbiorcy metody jest (prawie) identyczny jak inne parametry	276
Metoda działa (prawie) jak funkcja	277
Wskaźniki jako parametry odbiorcy metody	279
Przeliczanie litrów i mililitrów na galony z użyciem metod	283
Przeliczanie galonów na litry i mililitry z użyciem metod	284
Twój przyborek do Go	285

Steve myślał,
że kupił tyle
benzyny...



10 galonów

... ale w rzeczywistości
kupił tyle!



10 litrów

Zachowaj to dla siebie

10

Hermetyzacja i zagnieżdżanie

Błędy się zdarzają. Czasem program otrzymuje nieprawidłowe dane wejściowe od użytkownika, z wczytywanego pliku lub innych miejsc. W tym rozdziale zapoznasz się z **hermetyzacją**. Jest to sposób ochrony pól typów strukturalnych przed błędnymi danymi. Dzięki temu będziesz wiedzieć, że można bezpiecznie używać danych z pola!

Zobaczysz tu także, jak **zagnieżdżać** inne typy we własnych typach strukturalnych. Jeśli typ strukturalny potrzebuje metod dostępnych już w innym typie, nie trzeba kopiować i wklejać kodu metody. Można zagnieżdżyć ten inny typ w typie strukturalnym, a następnie używać metod typu zagnieżdżonego w taki sam sposób, jakby były zdefiniowane w Twoim typie!

Sprawdzanie poprawności w setterach jest świetne, gdy użytkownicy z nich korzystają. Jednak część osób ustawia wartości pól struktury bezpośrednio i nadal wpisuje błędne dane!



Tworzenie typu strukturalnego Date	290
Użytkownicy przypisują do pól struktury Date nieprawidłowe wartości!	291
Settery	292
W setterze jako odbiorcę trzeba podać wskaźnik	293
Dodawanie pozostałych setterów	294
Dodawanie sprawdzania poprawności danych do metod	296
Pola nadal mogą zostać ustawione na błędne wartości!	298
Przenoszenie typu Date do innego pakietu	299
Jak sprawić, by pola z typu Date nie były eksportowane?	301
Dostęp do nieeksportowanych pól za pomocą eksportowanych metod	302
Gettery	304
Hermetyzacja	305
Zagnieżdżanie typu Date w typie Event	308
Nieeksportowane pola nie są promowane	309
Eksportowane metody są promowane w taki sam sposób jak pola	310
Hermetyzowanie pola Title z typu Event	312
Promowane metody są dostępne w taki sam sposób jak metody typu zewnętrznego	313
Pakiet calendar jest gotowy!	314
Twój przyborek do Go	316

11

Co potrafisz zrobić?

Interfejsy

Czasem typ wartości nie jest istotny. Nie ma znaczenia, czym *jest* dana wartość. Musisz jedynie wiedzieć, że potrafi ona *wykonywać* potrzebne zadania i że możesz wywoływać dla niej *określone metody*. Nie jest istotne, czy masz `O1`owek, czy `D1`ugopis — potrzebujesz jedynie czegoś z metodą `Rysuj`. Nie jest ważne, czy masz `Samochod`, czy `Statek` — potrzebujesz jednostki z metodą `Kieruj`.

Potrzebny efekt można uzyskać w Go za pomocą **interfejsów**. Pozwalają one definiować zmienne i parametry funkcji, które mogą przechowywać wartość *dowolnego* typu, pod warunkiem że zdefiniowane są w nim określone metody.

Dwa różne typy mające te same metody	322
Parametr metody, który przyjmuje wartości tylko jednego typu	323
Interfejsy	325
Definiowanie typu implementującego interfejs	326
Typy konkretne i typy interfejsowe	327
Przypisz wartość dowolnego typu implementującego interfejs	328
Wywoływać można tylko metody zdefiniowane w interfejsie	329
Poprawianie funkcji <code>playlist</code> z użyciem interfejsu	331
Asercje typów	334
Nieudane asercje typów	336
Unikanie paniki po niepowodzeniu asercji	337
Testowanie typów <code>TapePlayer</code> i <code>TapeRecorder</code> z użyciem asercji typów	338
Interfejs <code>error</code>	340
Interfejs <code>Stringer</code>	342
Pusty interfejs	344
Twój przyborek do Go	347



12

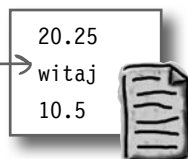
Znów stań na nogi

Przywracanie stanu po awarii

W każdym programie występują błędy. Trzeba mieć plan na takie sytuacje. Czasem obsługa błędów jest prosta — polega tylko na wyświetleniu błędu i zamknięciu programu. Jednak inne usterki mogą wymagać dodatkowych działań. Możliwe, że musisz zamknąć otwarte pliki i połączenia sieciowe lub wykonać inne operacje porządkujące, aby program nie pozostawił po sobie bałaganu. W tym rozdziale zobaczysz, jak **odracać** operacje porządkujące, by były wykonywane nawet po wystąpieniu błędu. Dowiesz się też, jak uruchamiać w programie procedurę **paniki** w tych (rzadkich) sytuacjach, gdy jest to wskazane, a także jak później **przywracać stan** programu.

Jeszcze o wczytywaniu liczb z pliku	350
Błędy powodują, że plik nie zostanie zamknięty	352
Odraczanie wywołań funkcji	353
Przywracanie stanu po błędach z użyciem odroczonego wywołania funkcji	354
Używanie odroczonego wywołania do gwarantowania zamknięcia plików	355
Wyświetlanie plików z katalogu	358
Wyświetlanie plików z podkatalogów (bardziej skomplikowane)	359
Wywołania funkcji rekurencyjnych	360
Rekurencyjne wyświetlanie zawartości katalogu	362
Obsługa błędów w funkcji rekurencyjnej	364
Uruchamianie procedury paniki	365
Ślad stosu	366
Odroczone wywołania ukończone przed awarią	366
Stosowanie funkcji panic w funkcji scanDirectory	367
Kiedy wywoływać panikę?	368
Funkcja recover	370
Funkcja recover zwraca wartość z wywołania panic	371
Przywracanie stanu po wywołaniu funkcji panic w funkcji scanDirectory	373
Ponowne wywołanie funkcji panic	374
Twój przybornik do Go	376

Nie można
przekształcić
na typ float64!



bad-data.txt

13

Udostępnianie kodu

Wątki goroutine i kanały

Praca nad tylko jedną operacją w danym momencie nie zawsze jest najszybszym sposobem na ukończenie zadania. Niektóre rozbudowane problemy można podzielić na mniejsze zadania. **Wątki goroutine** umożliwiają programowi jednoczesne wykonywanie kilku różnych zadań. Działanie wątków goroutine można koordynować za pomocą **kanałów**, które pozwalają przekazywać dane między takimi wątkami *oraz* synchronizować ich działanie, aby jeden wątek nie był wykonywany przed innym. Wątki goroutine umożliwiają pełne wykorzystanie możliwości komputerów wieloprocesorowych, dzięki czemu programy mogą działać jeszcze szybciej.

Pobieranie stron internetowych	380
Wielozadaniowość	382
Współbieżność z użyciem wątków goroutine	383
Używanie wątków goroutine	384
Używanie wątków goroutine w funkcji responseSize	386
Brak bezpośredniej kontroli nad wykonywaniem wątków goroutine	388
Instrukcji go nie można używać razem ze zwracaniem wartości	389
Wysyłanie i przyjmowanie wartości z użyciem kanałów	391
Synchronizowanie wątków goroutine za pomocą kanałów	392
Obserwowanie synchronizacji wątków goroutine	393
Zastosowanie kanałów do poprawienia programu zwracającego wielkość stron	396
Modyfikowanie kanału, aby przesyłał strukturę	398
Twój przybornik do Go	399

Wątek goroutine odbiorcy oczekuje na przestanie wartości przez inny wątek



14

Kontrola jakości kodu

Testy zautomatyzowane

Czy jesteś pewien, że oprogramowanie działa? Zupełnie pewien? Zanim przesałaś nową wersję kodu do użytkowników, zapewne wypróbowałaś nowe funkcje, aby mieć pewność, że działają. Ale czy sprawdziłaś *starsze* funkcje, aby się upewnić, że nie uszkodziłaś żadnej z nich? *Wszystkie* starsze funkcje? Jeśli te pytania rodzą w Tobie niepokój, to potrzebujesz **testów zautomatyzowanych**. Gwarantują one, że komponenty programu działają prawidłowo — nawet po modyfikacjach kodu. Pakiet `test` i narzędzie `go test` umożliwiają łatwe pisanie testów zautomatyzowanych z wykorzystaniem umiejętności, które już opanowałaś.

Testy zautomatyzowane wykrywają błędy, zanim zrobi to ktoś inny	402
Funkcja, dla której <u>należało</u> przygotować testy zautomatyzowane	403
Spowodowaliśmy błąd	405
Pisanie testów	406
Uruchamianie testów za pomocą polecenia <code>go test</code>	407
Testowanie zwracanych wartości	408
Tworzenie szczegółowych komunikatów o niepowodzeniu testów za pomocą funkcji <code>Errorf</code>	410
Funkcje pomocnicze w testach	411
Sprawianie, by testy kończyły się powodzeniem	412
Programowanie sterowane testami	413
Następny błąd do naprawienia	414
Uruchamianie określonego zbioru testów	417
Testy sterowane tabelami	418
Stosowanie testów do poprawiania kodu powodującego panikę	420
Twój przyborek do <code>Go</code>	422



Powodzenie



Dla wycinka `[]string{"jabłko", "pomarańcza", "gruszka"}` funkcja `JoinWithCommas` powinna zwrócić "jabłko, pomarańcza, a także gruszka".

Niepowodzenie



Dla wycinka `[]string{"jabłko", "pomarańcza"}` funkcja `JoinWithCommas` powinna zwrócić "jabłko i pomarańcza".

15

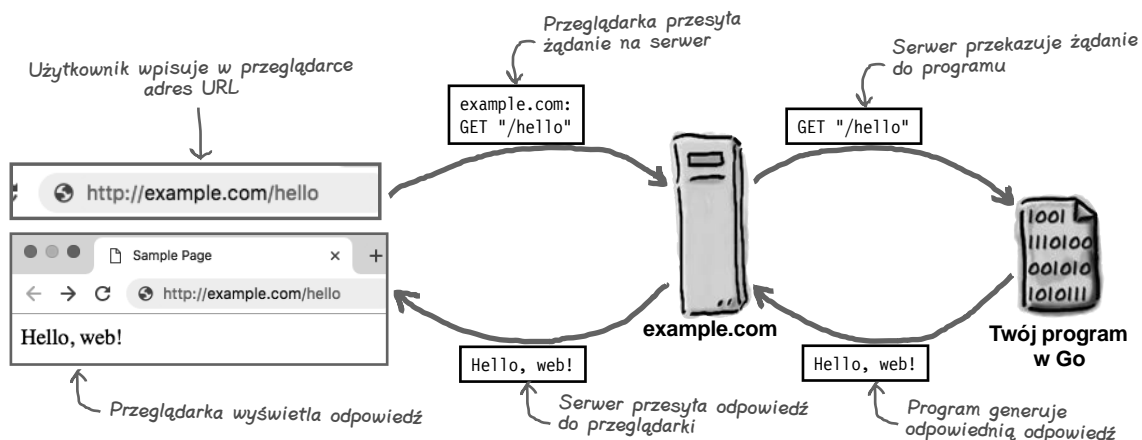
Reagowanie na żądania

Aplikacje internetowe

Mamy XXI wiek. Użytkownicy oczekują aplikacji internetowych. Go umożliwia Ci pracę także w tym obszarze. Biblioteka standardowa języka Go obejmuje pakiety, które pomagają hostować własne aplikacje internetowe i udostępniać je w przeglądarkach. W dwóch ostatnich rozdziałach tej książki zobaczysz, jak pisać takie aplikacje.

Pierwszą rzeczą, jakiej aplikacja internetowa potrzebuje, jest możliwość reagowania, gdy przeglądarka prześle żądanie. W tym rozdziale nauczysz się używać pakietu `net/http`, który służy właśnie do tego.

Pisanie aplikacji internetowych w języku Go	426
Przeglądarki, żądania, serwery i odpowiedzi	427
Prosta aplikacja internetowa	428
Twój komputer rozmawia sam ze sobą	429
Omówienie prostej aplikacji internetowej	430
Ścieżki do zasobów	432
Reagowanie w inny sposób na różne ścieżki do zasobów	433
Funkcje pierwszoklasowe	435
Przekazywanie funkcji do innych funkcji	436
Funkcje jako typy	436
Co dalej?	440
Twój przybornik do Go	441



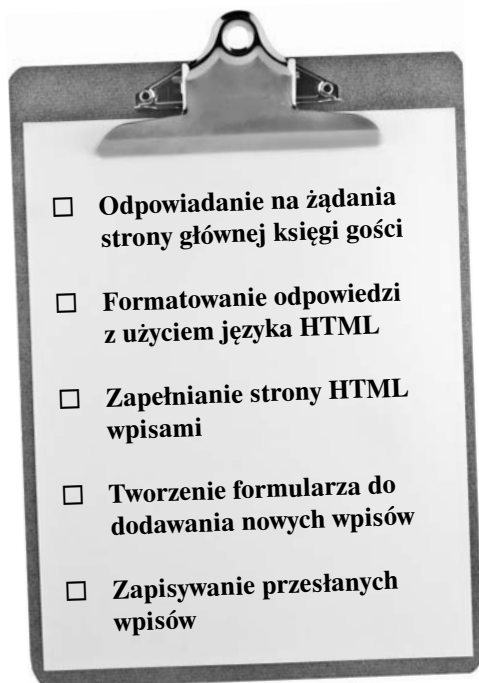
16

Stosowanie szablonów

Szablony HTML

Aplikacja internetowa musi przysyłać odpowiedzi w HTML-u, a nie w formie zwykłego tekstu. Zwykły tekst nadaje się do poczty elektronicznej i wpisów w mediach społecznościowych. Natomiast strony muszą być sformatowane. Potrzebne są nagłówki i akapity. Potrzebne są formularze pozwalające użytkownikom przekazywać dane do aplikacji. Wszystko to wymaga kodu w HTML-u.

Będziesz też musiał wstawiać dane do takiego kodu. Dlatego Go udostępnia pakiet `html/template`. Jest to rozbudowane narzędzie do dodawania danych do HTML-owych odpowiedzi od aplikacji. Szablony są kluczem do pisania większych i lepszych aplikacji internetowych, a w niniejszym, ostatnim rozdziale książki zobaczysz, jak używać takich szablonów.



Księga gości	446
Funkcje do obsługi żądań i sprawdzanie błędów	447
Tworzenie katalogu projektu i wypróbowanie aplikacji	448
Tworzenie listy wpisów w HTML-u	449
Zwracanie stron HTML w aplikacji	450
Pakiet <code>text/template</code>	451
Stosowanie interfejsu <code>io.Writer</code> razem z metodą <code>Execute</code> szablonu	452
Typy <code>ResponseWriter</code> i <code>os.Stdout</code> implementują interfejs <code>io.Writer</code>	453
Wstawianie danych do szablonów z użyciem akcji	454
Tworzenie opcjonalnych części szablonu za pomocą akcji <code>if</code>	455
Powtarzanie sekcji szablonu za pomocą akcji <code>range</code>	456
Wstawianie pól struktury do szablonu za pomocą akcji	457
Wczytywanie wycinka z wpisami z pliku	458
Struktura do przechowywania wpisów i ich liczby	460
Modyfikowanie szablonu w celu umieszczenia w nim wpisów	461
Umożliwianie użytkownikom dodawania danych za pomocą formularzy HTML-owych	464
Żądania przesłania formularza	466
Ścieżki i metody HTTP do przesyłania formularzy	467
Pobieranie z żądania wartości pól formularza	468
Zapisywanie danych z formularza	470
Przekierowania HTTP	472
Kompletny kod aplikacji	474
Twój przybornik do Go	477

Zrozumieć funkcję `os.OpenFile`**Otwieranie plików**

A

Niektóre programy muszą zapisywać dane w plikach, a nie tylko je wczytywać. W tej książce, gdy chciałeś pracować z plikami, musiałeś utworzyć je w edytorze tekstu, aby programy mogły je wczytywać. Jednak niektóre programy *generują* dane, a wtedy potrzebna jest możliwość *zapisywania* danych w pliku.

Wcześniej w niniejszej książce, aby otworzyć plik do zapisu, używaliśmy funkcji `os.OpenFile`, ale nie było tam miejsca na kompletne omówienie jej działania. Z tego dodatku dowiesz się wszystkiego, czego potrzebujesz, aby skutecznie korzystać z funkcji `os.OpenFile`.

Zrozumieć funkcję <code>os.OpenFile</code>	482
Przekazywanie do funkcji <code>os.OpenFile</code> stałych reprezentujących opcje	483
Notacja dwójkowa	485
Operatory bitowe	485
Bitowy operator I	486
Bitowy operator LUB	487
Stosowanie bitowego operatora LUB dla stałych z pakietu <code>os</code>	488
Stosowanie bitowego operatora LUB do poprawienia opcji funkcji <code>os.OpenFile</code>	489
Uniksowe uprawnienia do plików	490
Reprezentowanie uprawnień za pomocą typu <code>os.FileMode</code>	491
Notacja ósemkowa	492
Przekształcanie wartości ósemkowych na wartości typu <code>FileMode</code>	493
Objaśnienie wywołań funkcji <code>os.OpenFile</code>	494

Tym razem
nowy tekst jest
dotaczany na
końcu

Mrówniki są...
milusie!



aardvark.txt

Sześć kwestii, które nie zostały opisane

B

Pozostałości

Przerobiliśmy dużo materiału i prawie skończyłeś lekturę tej książki. Będzie mi Ciebie brakowało, ale byłoby nie w porządku, gdybym pozwolił Ci ruszyć w świat bez *odrobiny* dodatkowych przygotowań. Do omówienia w tym dodatku pozostawiłem sześć ważnych zagadnień.

Numer 1. Inicjalizacja w instrukcji if	496
Numer 2. Instrukcja switch	498
Numer 3. Inne typy proste	499
Numer 4. Jeszcze o runach	499
Numer 5. Kanały buforowane	503
Numer 6. Dalsza lektura	506

Wszystkie znaki są drukowalne

```
0: A
1: B
2: C
3: D
4: E
0: Б
2: Г
4: Д
6: Ж
8: И
```

Instrukcja inicjalizacji Warunek

```
if count := 5; count > 4 {
    fmt.Println("Zmienna count jest równa", count)
}
```

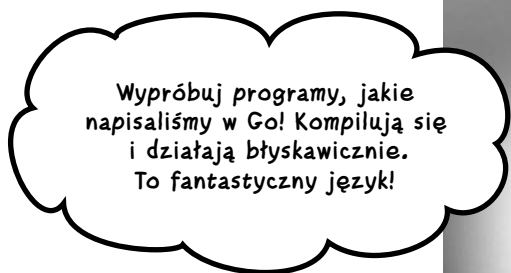
Przesyłanie wartości, gdy bufor jest pełny, powoduje zablokowanie wątku goroutine nadawcy

```
"d"
"c"
"b"
"a"
```

Kolejne przesyłane wartości są dodawane do bufora do momentu jego zapełnienia

1. Zaczynamy

Podstawy składni



Czy jesteś gotów na turbodoładowanie swojego oprogramowania? Czy szukasz prostego języka programowania, który **szybko się kompiluje** i **szybko działa**? Który umożliwia **łatwą dystrybucję** programów do użytkowników? Jeśli tak, to **jesteś gotów na Go!**

Go to język programowania, w którym nacisk jest położony na **prostotę** i **szybkość**. Jest on prostszy od innych języków, dlatego jego nauka przebiega szybciej. Ponadto Go pozwala wykorzystać możliwości nowoczesnych procesorów wielordzeniowych, dzięki czemu programy działają szybciej. W tym rozdziale poznasz wszystkie mechanizmy języka Go, które **ułatwiają pracę programistom** i **zwiększają zadowolenie użytkowników**.

Do biegu, gotowi, Go!

W 2007 r. z wyszukiwarką Google związany był pewien problem. Programiści musieli konserwować programy składające się z milionów wierszy kodu. Zanim mogli przetestować zmiany, musieli skompilować kod do postaci, w jakiej można go uruchomić. Zajmowało to wówczas prawie godzinę. Nie trzeba chyba tłumaczyć, że odbijało się to na produktywności programistów.

Dlatego inżynierowie z Google'a — Robert Griesemer, Rob Pike i Ken Thompson — zarysowali cele stawiane nowemu językowi:

- szybka kompilacja;
- mniej skomplikowany kod;
- automatyczne zwalnianie nieużywanej pamięci (przywracanie nieużytków);
- łatwe pisanie oprogramowania wykonującego jednocześnie różne operacje (współbieżność);
- dobra obsługa procesorów wielordzeniowych.

Po kilku latach pracy firma Google opracowała Go — język umożliwiający szybkie pisanie kodu oraz tworzenie programów, które można szybko skompilować i uruchomić. W 2009 r. projekt udostępniono na licencji open source. Obecnie język Go jest bezpłatny dla każdego. I warto z niego korzystać! Go błyskawicznie zyskuje popularność dzięki swojej prostocie i możliwościom.

Jeśli piszesz narzędzie uruchamiane w wierszu poleceń, Go pozwala na podstawie tego samego kodu źródłowego wygenerować pliki wykonywalne dla systemów Windows, macOS i Linux. Jeżeli tworzysz serwer WWW, Go pomoże Ci w obsłudze wielu użytkowników jednocześnie nawiązujących połączenie. Ponadto bez względu na to, *co* piszesz, Go pomoże Ci tworzyć kod łatwy w konserwacji i rozbudowywaniu.

Gotów do dalszej nauki? Ruszajmy!



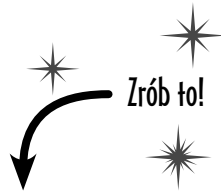
Narzędzie Go Playground

Najłatwiejszy sposób na wypróbowanie Go to odwiedzenie strony <https://play.golang.org> w przeglądarce. Zespół rozwijający Go udostępnia tam prosty edytor, w którym można wpisać kod w języku Go i uruchomić go na serwerach zespołu. Wynik jest wyświetlany bezpośrednio w przeglądarce.



Jest to możliwe oczywiście tylko wtedy, jeśli masz stabilne połączenie internetowe. W przeciwnym razie otwórz książkę na stronie 25, aby się dowiedzieć, jak pobrać kompilator języka Go i uruchomić go bezpośrednio na swoim komputerze. Następnie wykonaj opisane przykłady za pomocą tego kompilatora.

Wypróbuj teraz język Go!



- 1 Otwórz w przeglądarce stronę <https://play.golang.org>. Nie martw się, jeśli zobaczysz stronę, która nie jest identyczna ze zrzutem. Oznacza to tylko tyle, że witryna została usprawniona od czasu wydania książki.
- 2 Usuń cały kod z pola tekstowego i wpisz zamiast niego te instrukcje:

```
package main

import "fmt"

func main() {
    fmt.Println("Witaj, Go!")
}
```

Nie martw się — cały ten kod zostanie objaśniony na następnej stronie!

- 3 Kliknij przycisk *Format*, co spowoduje automatyczne sformatowanie kodu zgodnie z konwencjami stosowanymi w Go.
- 4 Kliknij przycisk *Run*.

Powinieneś zobaczyć napis „Witaj, Go!” wyświetlony w dolnej części ekranu. Gratulacje — właśnie uruchomiłeś swój pierwszy program w Go!

Przewróć stronę, a dowiesz się, co właśnie zrobiłeś.

Dane wyjściowe

Witaj, Go!

Z czego składa się program?

Co to wszystko znaczy?

Właśnie uruchomiłeś swój pierwszy program w Go! Pora przyjrzeć się kodowi i ustalić, co oznacza.

Każdy plik w Go rozpoczyna się od klauzuli `package`. **Pakiet** (ang. *package*) to zbiór kodu przeznaczonego do podobnych zadań, np. do formatowania łańcuchów znaków lub wyświetlania grafiki. Klauzula `package` służy do podawania nazwy pakietu, którego częścią stanie się dany plik z kodem. Tu używany jest specjalny pakiet `main`, wymagany, jeśli kod ma być uruchamiany bezpośrednio (zwykle w oknie terminala).

Dalej w prawie wszystkich plikach w języku Go znajdują się instrukcje `import`. W każdym pliku trzeba **zaimportować** inne pakiety, aby kod z danego pliku mógł używać kodu z podanych pakietów. Wczytanie do komputera całego kodu języka Go spowodowałoby powstanie dużego i powolnego programu. Dlatego importowane są tylko potrzebne pakiety.

Ostatni fragment każdego pliku w języku Go to wykonywany kod, często podzielony na funkcje. **Funkcja** to grupa wierszy kodu, które można **wywoływać** (uruchamiać) w innych miejscach programu. Gdy program w języku Go jest wykonywany, szuka funkcji `main` i uruchamia ją w pierwszej kolejności. Stąd nazwa `main`, czyli „główna”.

Typowy układ pliku w języku Go

Szybko przyzwyczaisz się do trzech sekcji występujących w podanej tu kolejności w prawie każdym pliku w Go, z jakim się zetkniesz:

- 1) klauzula `package`,
- 2) instrukcje `import`,
- 3) wykonywany kod.

Można powiedzieć, że jest miejsce na wszystko i wszystko jest na swoim miejscu. Go jest bardzo *spójnym* językiem. To dobrze. Zobaczysz, że często intuicyjnie będziesz *wiedzieć*, gdzie szukać danego fragmentu w kodzie!

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

Ten wiersz informuje, że reszta kodu z tego pliku należy do pakietu `main`

Ten informuje, że używany będzie dostępny w pakiecie `fmt` kod do formatowania tekstu

Funkcja `main` ma specjalne znaczenie — jest wykonywana jako pierwsza w momencie uruchomienia programu

Ten wiersz wyświetla (instrukcja `print`) tekst „Witaj, Go!” w terminalu (lub w przeglądarce, jeśli używasz narzędzia Go Playground)

W tym celu wywoływana jest funkcja `Println` z pakietu `fmt`



Spokojnie

Nie martw się, jeśli nie wszystko jest dla Ciebie zrozumiałe!

Na kilku kolejnych stronach przyjrzymy się szczegółowo całemu kodowi.

Klauzula `package` { `package main`

Sekcja instrukcji `import` { `import "fmt"`

Podstawowy kod { `func main() {`
 `fmt.Println("Witaj, Go!")`
`}`

nie istnieją
grupie pytań

P: W innym języku programowania, z jakiego korzystam, każda instrukcja musi kończyć się średnikiem. Czy w Go jest inaczej?

O: Możesz używać średników do rozdzielania instrukcji w Go, jednak nie jest to wymagane (a wręcz zwykle nie jest to dobrze widziane).

P: Czym jest przycisk Format? Dlaczego należało go kliknąć przed uruchomieniem kodu?

O: Kompilator języka Go jest dostępny razem ze standardowym narzędziem formatującym kod — `go fmt`. Przycisk *Format* uruchamia internetową wersję tego narzędzia.

Gdy udostępniasz kod, inni programiści języka Go oczekują, że używasz standardowego formatowania. Oznacza to, że wcięcia i odstępy powinny być sformatowane w standardowy sposób, co poprawia czytelność kodu. W innych językach wymaga to polegania na tym, że programiści będą ręcznie formatować kod pod kątem wytycznych z zakresu stylu.

W Go wystarczy uruchomić polecenie `go fmt`, a automatycznie wprowadzi ono wszystkie poprawki.

Uruchomiłem narzędzie formatujące dla każdego przykładu utworzonego na potrzeby tej książki. Ty także powinieneś uruchamiać to narzędzie dla całego kodu, jaki piszesz!

A jeśli coś się nie powiedzie?

Programy w języku Go muszą być zgodne z określonymi regułami, aby nie dezorientować kompilatora. Naruszenie którejs z tych reguł skutkuje wyświetleniem komunikatu o błędzie.

Załóżmy, że zapomniałeś dodać nawias w wywołaniu funkcji `Println` w wierszu 6.

Po próbie uruchomienia tej wersji programu wystąpi błąd:

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

Załóżmy, że zapomniałeś nawiasów, które znajdowały się w tych miejscach

Nazwa pliku używana przez narzędzie Go Playground

Numer wiersza, w którym wystąpił błąd

Opis błędu

```
prog.go:6:14: syntax error: unexpected literal "Witaj, Go!" at end of statement
```

Numer znaku w wierszu, gdzie wystąpił błąd

Język Go informuje o pliku z kodem źródłowym i o numerze wiersza, gdzie trzeba przejść, aby rozwiązać problem (narzędzie Go Playground zapisuje kod w tymczasowym pliku przed uruchomieniem; stąd pochodzi nazwa pliku `prog.go`). Dalej wyświetlany jest opis błędu. Tu zostały usunięte nawiasy, dlatego Go nie potrafi wykryć, że należy wywołać funkcję `Println`. Nie potrafi więc stwierdzić, dlaczego na końcu wiersza 6. znajduje się kod "Witaj, Go!".



Psucie uczy!

Aby zrozumieć reguły, jakich trzeba przestrzegać w programach w Go, możesz celowo popełniać różne błędy w kodzie. Przyjrzyj się poniższemu przykładowi. Spróbuj wprowadzić jedną z opisanych zmian i uruchomić kod. Następnie wycofaj zmianę i wypróbuj następną modyfikację. Zobacz, co się stanie!

```
package main

import "fmt"

func main() {
    fmt.Println("Witaj, Go!")
}
```

Spróbuj zrobić błąd w przykładowym kodzie i zobacz, co się stanie!

Jeśli to zrobisz...		... wystąpi błąd, ponieważ...
Usunięcie klauzuli package	package main	Każdy plik w Go musi zaczynać się od klauzuli package.
Usunięcie instrukcji import	import "fmt"	W każdym pliku w Go trzeba importować wszystkie używane pakiety.
Zaimportowanie drugiego (nieużywanego) pakietu	import "fmt" import "strings"	W plikach w Go trzeba importować <i>wyłącznie</i> używane pakiety. Dzięki temu kompilacja przebiegnie szybko.
Zmiana nazwy funkcji main	func main hello	Go szuka funkcji o nazwie main, aby uruchomić ją jako pierwszą.
Zmiana pierwszej litery w wywołaniu Println na małą	fmt.Pprintln("Witaj, Go!")	W Go wielkość liter jest ważna, dlatego wywołanie fmt.Println jest poprawne, ale nie istnieje funkcja fmt.println.
Usunięcie nazwy pakietu przed wywołaniem Println	fmt .Println("Witaj, Go!")	Funkcja Println nie należy do pakietu main, dlatego Go wymaga nazwy pakietu przed wywołaniem.

W ramach przykładu spróbuj wprowadzić pierwszą zmianę.

Usuń klauzulę package →

```
import "fmt"

func main() {
    fmt.Println("Witaj, Go!")
}
```

Zobaczysz błąd! →

```
can't load package: package main:
prog.go:1:1: expected 'package', found 'import'
```

Wywołanie funkcji

W przykładzie wywoływana jest funkcja `Println` z pakietu `fmt`. Aby wywołać tę funkcję, wpisz jej nazwę (tu jest to `Println`) i nawias.

Objaśnienie znajdziesz dalej

Nazwa funkcji

Nawias

```
fmt.Println()
```

```
package main

import "fmt"

func main() {
    fmt.Println("Witaj, Go!")
}
```

Wywołanie funkcji `Println`

`Println`, podobnie jak wiele funkcji, może przyjmować **argumenty**, czyli używane w funkcji wartości. Argumenty podaje się w nawiasie po nazwie funkcji.

W nawiasie znajdują się argumenty rozdzielone przecinkami

```
fmt.Println("Pierwszy argument", "Drugi argument")
```

Dane wyjściowe → **Pierwszy argument Drugi argument**

Funkcję `Println` można wywołać bez argumentów lub z kilkoma argumentami. Jednak gdy przyjrzesz się innym funkcjom, zauważysz, że większość z nich wymaga określonej liczby argumentów. Jeśli podasz ich za mało lub za dużo, zobaczysz komunikat o błędzie informujący, ile argumentów oczekiwano, i będziesz musiał poprawić kod.

Funkcja `Println`

Funkcji `Println` używaj wtedy, gdy chcesz zobaczyć, co program robi. Przekazane do niej argumenty są wyświetlane w oknie terminala. Wyświetlane argumenty są rozdzielone spacją.

Po wyświetleniu wszystkich argumentów funkcja `Println` przechodzi do następnego wiersza okna terminala. Stąd człon `ln` na końcu nazwy (od *ang. line*, czyli wiersz).

```
fmt.Println("Pierwszy argument", "Drugi argument")
fmt.Println("Inny wiersz")
```

Dane wyjściowe → **Pierwszy argument Drugi argument
Inny wiersz**

Używanie funkcji z innych pakietów

Kod z pierwszego programu jest częścią pakietu `main`, jednak funkcja `Println` znajduje się w pakiecie `fmt` (od ang. *format*). Aby móc wywołać funkcję `Println`, trzeba zaimportować zawierający ją pakiet.

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

Przed uzyskaniem dostępu do funkcji `Println` trzeba zaimportować pakiet `fmt`

Informuje, że wywoływana jest funkcja należąca do pakietu `fmt`

Po zaimportowaniu pakietu można uzyskać dostęp do wszystkich zawartych w nim funkcji. W tym celu należy wpisać nazwę pakietu, kropkę i nazwę potrzebnej funkcji.

Nazwa pakietu Nazwa funkcji

`fmt.Println()`

Oto przykładowy kod, który wywołuje funkcje z kilku innych pakietów. Ponieważ trzeba zaimportować kilka pakietów, stosowany jest inny format instrukcji `import`, pozwalający podać w nawiasie wiele pakietów (po jednym w wierszu).

```
package main
import (
    "math"
    "strings"
)
func main() {
    math.Floor(2.75)
    strings.Title("rusz głową. go")
}
```

Ten inny format instrukcji `import` umożliwia jednoczesne importowanie wielu pakietów

Importowanie pakietu `math`, co daje dostęp do funkcji `math.Floor`

Importowanie pakietu `strings`, co daje dostęp do funkcji `strings.Title`

Wywołanie funkcji `Floor` z pakietu `math`.

Wywołanie funkcji `Title` z pakietu `strings`

Ten program nie wyświetla danych wyjściowych (objaśnienie znajdziesz dalej)

Po zaimportowaniu pakietów `math` i `strings` możesz uzyskać dostęp do funkcji `Floor` z pakietu `math` za pomocą wywołania `math.Floor` i do funkcji `Title` z pakietu `strings` przy użyciu wywołania `strings.Title`.

Może zauważyłeś, że mimo wywołań dwóch wymienionych funkcji ten przykład nie wyświetla żadnych danych wyjściowych. Zaraz zobaczysz, jak rozwiązać ten problem.

Wartości zwracane przez funkcje

W poprzednim przykładowym kodzie wywoływane są funkcje `math.Floor` i `strings.Title`, jednak ten kod nie wyświetla żadnych danych wyjściowych:

```
package main

import (
    "math"
    "strings"
)

func main() {
    math.Floor(2.75)
    strings.Title("rusz głową. go")
}
```

Ten program nie wyświetla żadnych danych wyjściowych!

W momencie wywołania funkcji `fmt.Println` nie trzeba robić nic więcej. Wystarczy przekazać jedną lub kilka wartości do funkcji `Println`, by zostały wyświetlone, i można przyjąć, że funkcja to zrobi. Jednak czasem program potrzebuje wywołać funkcję i pobrać od niej dane. Dlatego w większości języków programowania funkcje mogą **zwracać wartości**. Zwracana wartość to dane obliczane przez funkcję i zwracane do jednostki wywołującej.

Funkcje `math.Floor` i `strings.Title` to przykładowe funkcje zwracające wartość. Funkcja `math.Floor` przyjmuje liczbę zmiennoprzecinkową, zaokrągla ją w dół do najbliższej liczby całkowitej i zwraca tę liczbę całkowitą. Funkcja `strings.Title` przyjmuje łańcuch znaków, zmienia pierwszą literę każdego słowa z tego łańcucha na wielką i zwraca tekst z wielkimi pierwszymi literami.

Aby zobaczyć wyniki takich wywołań, trzeba pobrać zwrócone wartości i przekazać je do funkcji `fmt.Println`:

```
package main

import (
    "fmt"
    "math"
    "strings"
)

func main() {
    fmt.Println(math.Floor(2.75))
    fmt.Println(strings.Title("rusz głową. go"))
}
```

Importowanie także pakietu `fmt`

Przyjmuje łańcuch znaków i zwraca nowy łańcuch z wielkimi pierwszymi literami

Dane wyjściowe

Wywołanie `fmt.Println` dla wartości zwróconej przez funkcję `math.Floor`

Wywołanie `fmt.Println` dla wartości zwróconej przez funkcję `strings.Title`

Przyjmuje liczbę, zaokrągla ją w dół i zwraca wynik

```
2
Rusz Głową. Go
```

Po wprowadzeniu tych zmian zwracane wartości są wyświetlane i możesz zobaczyć wyniki.

Basenowa układanka



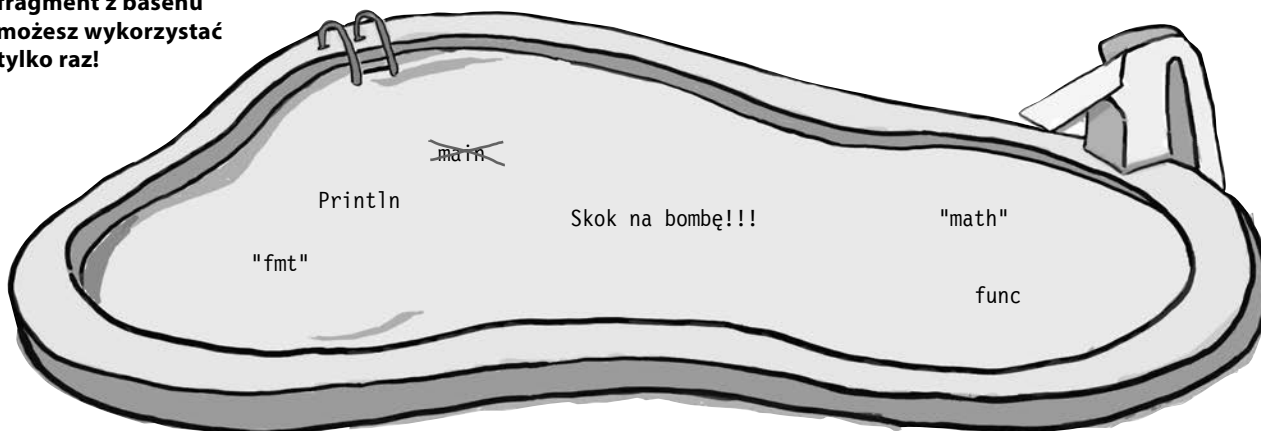
Twoje **zadanie** polega na tym, by wziąć fragmenty kodu z basenu i umieścić je w pustych wierszach w kodzie. **Nie** używaj tych samych fragmentów więcej niż raz. Nie musisz wykorzystać wszystkich fragmentów. Twoim **celem** jest opracowanie kodu, który będzie działał i zwróci pokazane dane wyjściowe.

```
package main ← Pierwsza luka jest już  
                uzupełniona!  
  
import (  
    _____  
)  
  
_____ main() {  
    fmt.Println(_____)  
}
```

Dane wyjściowe

Skok na bombę!!!

Uwaga: każdy fragment z basenu możesz wykorzystać tylko raz!



→ Odpowiedź znajdziesz na stronie 29.


Szablon programu w Go

Czytając kolejne fragmenty kodu, wyobraź sobie, że są one umieszczane w następującym kompletnym programie w Go:

Jeszcze lepszym rozwiązaniem jest wpisanie tego programu w narzędziu Go Playground i wstawianie po kolei prezentowanych fragmentów kodu, aby samemu zobaczyć, jak działają.

```
package main




import "fmt"

func main() {
    fmt.Println()
}
```



Tu wstaw swój kod!



Łańcuchy znaków


Jako argumenty funkcji `Println` przekazywaliśmy **łańcuchy znaków**. Łańcuch znaków to seria bajtów, które zwykle reprezentują znaki tekstowe. Łańcuchy znaków można definiować bezpośrednio w kodzie, używając **literalów tekstowych** — tekstu między cudzysłowami (jest on traktowany przez Go jak łańcuch znaków).


Cudzysłów otwierający  "Witaj, Go!"  Cudzysłów zamykający  Dane wyjściowe
 "Witaj, Go!"


W łańcuchach znaków symbole nowego wiersza, tabulacji i podobne, które trudno byłoby przedstawić w kodzie programu, można zapisywać w formie **sekwencji ucieczki**, czyli lewego ukośnika, po którym następują znaki reprezentujące inny symbol.


Nowy wiersz w łańcuchu znaków  "Witaj, \nGo!"  Dane wyjściowe
 Witaj,
 Go!

 "Witaj, \tGo!"  Witaj, Go!

 "Cudzysłowy: \"\""

 "Lewy ukośnik: \\""

 Cudzysłowy: ""

 Lewy ukośnik: \

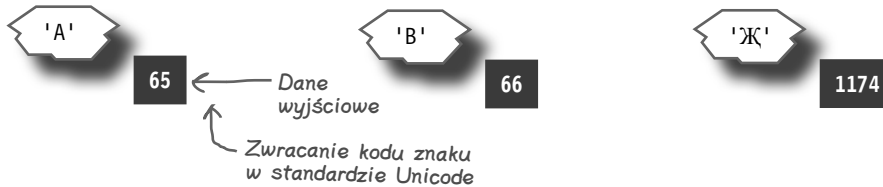
Sekwencja ucieczki	Wartość
\n	Znak nowego wiersza
\t	Znak tabulacji
\"	Cudzysłów
\\	Lewy ukośnik

Runy

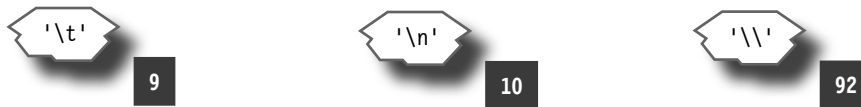
Choć łańcuchy znaków zwykle służą do reprezentowania całych sekwencji znaków tekstowych, **runy** w Go oznaczają pojedyncze znaki.

Literały tekstowe zapisuje się w cudzysłowach ("), natomiast **literały run** są umieszczane w apostrofach (').

W programach w Go można używać niemal dowolnych znaków z prawie każdego języka świata, ponieważ do zapisywania run używany jest tu standard Unicode. Runy są przechowywane jako kody liczbowe, a nie jako znaki, a jeśli przekażesz runę do funkcji `fmt.Println`, zobaczysz w danych wyjściowych ten kod, a nie sam znak.

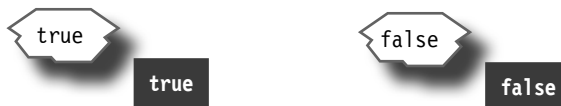


W literałach run, podobnie jak w literałach tekstowych, można stosować sekwencje ucieczki do reprezentowania znaków, które trudno jest umieszczać w kodzie programów:



Wartości logiczne

Wartości **logiczne** przyjmują tylko dwie formy: `true` i `false`. Są one przydatne przede wszystkim w instrukcjach warunkowych, powodujących, że fragmenty kodu są uruchamiane tylko wtedy, gdy warunek jest prawdziwy lub fałszywy (instrukcje warunkowe są omówione w następnym rozdziale).



```
package main    To znowu nasz szablon
import "fmt"
func main() {
    fmt.Println(Tu wstaw swój kod!)
}
```


Liczby

Możesz też bezpośrednio definiować liczby w kodzie. Są one jeszcze prostsze niż literały tekstowe — wystarczy wpisać liczbę.



```
package main    To znowu nasz szablon

import "fmt"

func main() {
    fmt.Println()
}
```

Tu wstaw swój kod!

Wkrótce zobaczysz, że Go traktuje liczby całkowite i zmiennoprzecinkowe jako wartości różnych typów. Zapamiętaj, że do odróżniania liczb całkowitych od zmiennoprzecinkowych możesz używać kropki dziesiętnej.

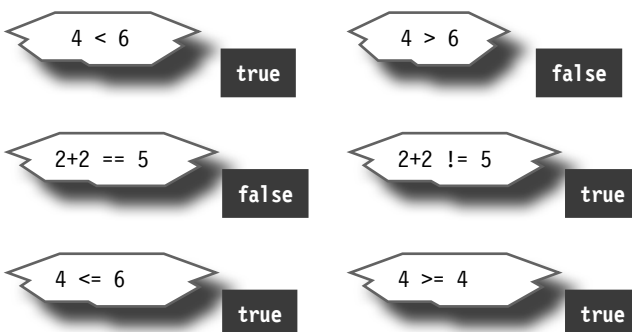
Operacje matematyczne i porównania

Podstawowe operatory matematyczne w Go działają jak w większości innych języków. Symbol `+` służy do dodawania, symbol `-` oznacza odejmowanie, `*` reprezentuje mnożenie, a `/` jest przeznaczony do dzielenia.



Do porównywania dwóch wartości i sprawdzania, czy jedna z nich jest mniejsza czy większa od drugiej, służą operatory `<` i `>`. Do sprawdzania, czy dwie wartości są równe, służy operator `==` (dwa znaki równości), a do określania, czy są różne, przeznaczony jest operator `!=` (wykrzyknik i znak równości; ten operator należy czytać jako „nie jest równe”). Operator `<=` sprawdza, czy pierwsza wartość jest mniejsza lub równa względem drugiej, a operator `>=` informuje, czy pierwsza wartość jest większa lub równa względem drugiej.

Wynikiem porównania jest wartość logiczna — `true` lub `false`.



Typy

W poprzednim przykładowym kodzie używane były funkcja `math.Floor` (zaokrąglą liczbę zmiennoprzecinkową w dół do najbliższej liczby całkowitej) i `strings.Title` (przekształca pierwsze litery słów na wielkie). Sensowne jest przekazywanie liczby jako argumentu funkcji `Floor` i łańcucha znaków jako argumentu funkcji `Title`. Co się jednak stanie, jeśli przekażesz łańcuch znaków do funkcji `Floor` i liczbę do funkcji `Title`?

```
package main

import (
    "fmt"
    "math"
    "strings"
)

func main() {
    fmt.Println(math.Floor("rusz głową. go"))
    fmt.Println(strings.Title(2.75))
}
```

Standardowo przyjmuje liczbę zmiennoprzecinkową

Standardowo przyjmuje łańcuch znaków

Błędy

```
cannot use "rusz głową.go" (type string) as type float64 in argument to math.Floor
cannot use 2.75 (type float64) as type string in argument to strings.Title
```

Go wyświetli dwa komunikaty o błędach (po jednym dla każdej funkcji), a program nie zostanie nawet uruchomiony.

Przedmioty w świecie dookoła nas często można poklasyfikować na różne typy na podstawie ich zastosowań. Nie jesz samochodów osobowych ani ciężarówek na śniadanie (ponieważ są to pojazdy) i nie jeździsz omletem lub talerzem owsianki do pracy (ponieważ są to produkty śniadaniowe).

Podobnie w Go wartości są poklasyfikowane na różne **typy**, określające, do czego można używać tych wartości. Liczby całkowite można stosować w operacjach matematycznych, jednak łańcuchy znaków tego nie umożliwiają. W łańcuchach znaków można zmieniać pierwsze litery słów na wielkie, co jest niemożliwe w przypadku liczb itd.

Go jest językiem ze **statyczną kontrolą typów**. To oznacza, że jeszcze przed uruchomieniem programu wie, jakiego typu będą wartości. Funkcje oczekują argumentów określonych typów, a także wartości zwracane przez funkcje mają ustalone typy (które mogą, ale nie muszą być identyczne z typami argumentów). Jeśli przypadkowo zastosujesz wartość niewłaściwego typu, Go wyświetli komunikat o błędzie. Jest to korzystne, ponieważ pozwala Ci wykryć problem, zanim zauważą go użytkownicy.

W Go działa statyczna kontrola typów. Jeśli użyjesz wartości niewłaściwego typu, Go Cię o tym poinformuje.

Typy (ciąg dalszy)

Możesz wyświetlić typ każdej wartości, przekazując ją do funkcji `TypeOf` z pakietu `reflect`. Sprawdź, jakiego typu są niektóre wartości, z którymi już się zetknąłeś:

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    fmt.Println(reflect.TypeOf(42))
    fmt.Println(reflect.TypeOf(3.1415))
    fmt.Println(reflect.TypeOf(true))
    fmt.Println(reflect.TypeOf("Witaj, Go!"))
}
```

Importowanie pakietu `reflect`, aby można było użyć funkcji `TypeOf`

Zwracanie typu argumentu

Dane wyjściowe

```
int
float64
bool
string
```

Oto zastosowania wymienionych typów:

Typ	Opis
<code>int</code>	Liczba całkowita; przechowuje liczby całkowite.
<code>float64</code>	Liczba zmiennoprzecinkowa; przechowuje liczby z częścią ułamkową. Człon 64 w nazwie typu oznacza 64 bity danych używane do przechowywania liczby, dlatego wartości typu <code>float64</code> przed zaokrągleniem mogą być całkiem precyzyjne (choć nie nieskończenie dokładne).
<code>bool</code>	Wartość logiczna — tylko <code>true</code> lub <code>false</code> .
<code>string</code>	Łańcuch znaków; seria danych reprezentujących zwykle znaki tekstowe.



Ćwiczenie

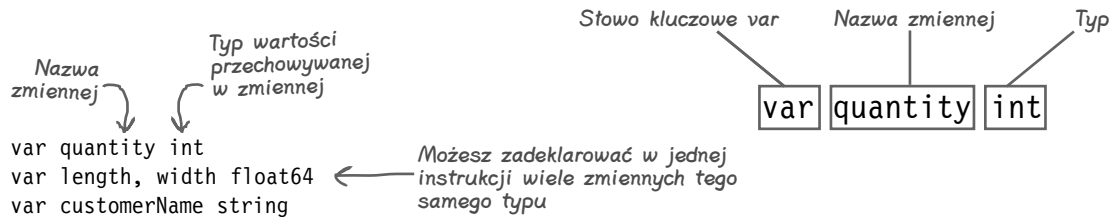
Narysuj linie, aby dopasować każdy fragment kodu do typu. Do niektórych typów prowadzić będzie kilka linii.

```
reflect.TypeOf(25)           int
reflect.TypeOf(true)
reflect.TypeOf(5.2)         float64
reflect.TypeOf(1)
reflect.TypeOf(false)      bool
reflect.TypeOf(1.0)
reflect.TypeOf(witaj")     string
```

Odpowiedzi znajdziesz na stronie 29.

Deklarowanie zmiennych

W Go **zmienna** to fragment pamięci zawierający wartość. Możesz nadać zmiennej nazwę, używając **deklaracji zmiennej**. Wystarczy podać słowo kluczowe `var`, a następnie nazwę zmiennej i typ przechowywanych wartości.



Po zadeklarowaniu zmiennej możesz przypisać do niej dowolną wartość danego typu, używając operatora `=` (jednego znaku równości):

```
quantity = 2
customerName = "Joanna Legutko"
```

W jednej instrukcji możesz przypisać kilka wartości do kilku zmiennych. Wystarczy umieścić kilka nazw zmiennych po lewej stronie operatora `=` i tę samą liczbę wartości po prawej stronie. Nazwy i wartości należy rozdzielić przecinkami.

```
length, width = 1.2, 2.4
```

Jednoczesne przypisywanie wartości do kilku zmiennych

Po przypisaniu wartości do zmiennych te ostatnie można stosować w dowolnym miejscu, gdzie poprawne byłyby pierwotne wartości:

```
package main

import "fmt"

func main() {
    Deklaracje zmiennych {
        var quantity int
        var length, width float64
        var customerName string
    }

    Przypisywanie wartości do zmiennych {
        quantity = 4
        length, width = 1.2, 2.4
        customerName = "Joanna Legutko"
    }

    Używanie zmiennych {
        fmt.Println(customerName)
        fmt.Println("zamówiła", quantity, "arkusze, ")
        fmt.Println("każdy o powierzchni")
        fmt.Println(length*width, "metra kwadratowego.")
    }
}
```

```
Joanna Legutko
zamówiła 4 arkusze,
każdy o powierzchni
2.88 metra kwadratowego.
```

Deklarowanie zmiennych (ciąg dalszy)

Jeśli z góry znasz wartość zmiennej, możesz zadeklarować zmienną i przypisać im wartości w tym samym wierszu:

```

Deklarowanie zmiennych ORAZ przypisywanie im wartości
{
  var quantity int = 4
  var length, width float64 = 1.2, 2.4
  var customerName string = "Joanna Legutko"
}

```

Umieść przypisanie na końcu

Jeśli deklarujesz kilka zmiennych, podaj kilka wartości

Do istniejących zmiennych możesz przypisać nowe wartości, jednak muszą to być wartości tego samego typu. Statyczna kontrola typów w Go gwarantuje, że nie przypiszesz przypadkowo do zmiennej niewłaściwego rodzaju wartości.

Typy przypisanych wartości nie pasują do typów z deklaracji!

```

{
  quantity = "Joanna Legutko"
  customerName = 4
}

```

Błędy

```

cannot use "Joanna Legutko" (type string) as type int in assignment
cannot use 4 (type int) as type string in assignment

```

Jeśli przypisujesz wartość do zmiennej w miejscu deklaracji, zwykle możesz pominąć typ zmiennej. Jako typ zmiennej używany jest wtedy typ przypisywanej wartości.

```

var quantity = 4
var length, width = 1.2, 2.4
var customerName = "Joanna Legutko"
fmt.Println(reflect.TypeOf(quantity))
fmt.Println(reflect.TypeOf(length))
fmt.Println(reflect.TypeOf(width))
fmt.Println(reflect.TypeOf(customerName))

```

Pomijanie typów zmiennych

```

int
float64
float64
string

```

Wartości zerowe

Jeśli zadeklarujesz zmienną bez przypisywania do niej wartości, ta zmienna będzie zawierać **wartość zerową** dla danego typu. Dla typów liczbowych wartością zerową jest 0:

```

var myInt int
var myFloat float64
fmt.Println(myInt, myFloat)

```

Wartość zerowa dla zmiennych typu int to 0 → 0 0 ← Wartość zerowa dla zmiennych typu float64 to 0

Jednak w innych typach wartość 0 byłaby nieprawidłowa, dlatego wartość zerowa dla takich typów może być inna. Na przykład dla zmiennych typu string jest to pusty łańcuch znaków, a dla zmiennych logicznych (typ bool) jest to wartość false.

```

var myString string
var myBool bool
fmt.Println(myString, myBool)

```

Wartość zerowa dla zmiennych typu string to pusty łańcuch znaków → false ← Wartość zerowa dla zmiennych typu bool to false



Magnesiki z kodem

Program w Go jest rozsypany po całej lodówce. Czy potrafisz odtworzyć fragmenty kodu, aby uzyskać działający program, który wygeneruje pokazane dane wyjściowe?

Dane wyjściowe

```
Miałem 10 jabłek.
Jakiś łobuz zjadł 4 jabłka.
Zostało mi 6 jabłek.
```

Fragmenty kodu Go do odtworzenia:

- , "jabłek.")
- , "jabłka.")
- , "jabłek.")
- func main() { }
- fmt.Println("Miałem",
- fmt.Println("Jakiś łobuz zjadł",
- fmt.Println("Zostało mi",
- package main
- originalCount-eatenCount
- import ("fmt")
- var
- var
- int
- int
- =
- =
- 10
- 4
- originalCount
- originalCount
- eatenCount
- eatenCount

→ Odpowiedź znajdziesz na stronie 30.

Krótkie deklaracje zmiennych

Wspomniałem już, że można zadeklarować zmienne i przypisać do nich wartości w tym samym wierszu:

```

Deklarowanie zmiennych ORAZ przypisywanie im wartości {
    var quantity int = 4
    var length, width float64 = 1.2, 2.4
    var customerName string = "Joanna Legutko"
}

```

Umieść przypisanie na końcu

Jeśli deklarujesz kilka zmiennych, podaj kilka wartości

Jeśli jednak zaraz po zadeklarowaniu zmiennej znana jest jej początkowa wartość, częściej stosuje się **krótkie deklaracje zmiennych**. Zamiast jawnie deklarować typ zmiennej i później przypisywać do niej wartość za pomocą operatora =, można wykonać oba te zadania jednocześnie, używając operatora :=.

Zmodyfikuj wcześniejszy przykład za pomocą krótkich deklaracji zmiennych:

```

package main

import "fmt"

func main() {
    Deklarowanie zmiennych ORAZ przypisywanie im wartości {
        quantity := 4
        length, width := 1.2, 2.4
        customerName := "Joanna Legutko"

        fmt.Println(customerName)
        fmt.Println("zamówiła", quantity, "arkusze,")
        fmt.Println("każdy o powierzchni")
        fmt.Println(length*width, "metra kwadratowego.")
    }
}

```

```

Joanna Legutko
zamówiła 4 arkusze,
każdy o powierzchni
2.88 metra kwadratowego.

```

Nie trzeba tu jawnie deklarować typu zmiennych. Typ wartości przypisywanej do zmiennej staje się jej typem.

Ponieważ krótkie deklaracje zmiennych są tak wygodne i zwarte, stosuje się je częściej niż zwykłe deklaracje. Możesz jednak zetknąć się z obiema formami, dlatego ważne jest, aby znać każdą z nich.



Psucie uczy!

Weź program używający zmiennych, po czym spróbuj wprowadzić jedną z przedstawionych niżej zmian i uruchomić kod. Następnie cofnij zmiany i wypróbuj inne modyfikacje. Zobacz, co się stanie!

```
package main

import "fmt"

func main() {
    quantity := 4
    length, width := 1.2, 2.4
    customerName := "Joanna Legutko"

    fmt.Println(customerName)
    fmt.Println("zamówiła", quantity, "arkusze,")
    fmt.Println("każdy o powierzchni")
    fmt.Println(length*width, "metra kwadratowego.")
}
```

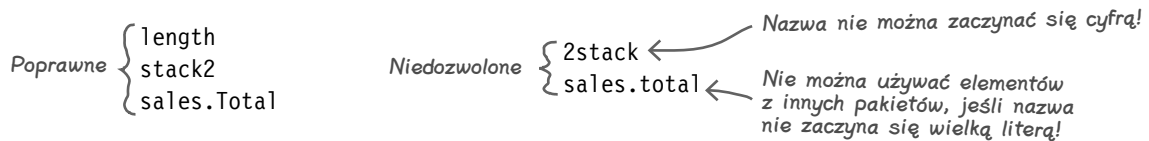
```
Joanna Legutko
zamówiła 4 arkusze,
każdy o powierzchni
2.88 metra kwadratowego.
```

Jeśli to zrobisz...		... wystąpi błąd, ponieważ...
Dodanie drugiej deklaracji tej samej zmiennej	<pre>quantity := 4 quantity := 4</pre>	Zmienną można zadeklarować tylko raz. Można jednak dowolnie często przypisywać do niej nowe wartości. Możesz też zadeklarować inne zmienne o tej samej nazwie, ale muszą się one znajdować w innym zasięgu. Omówienie zasięgów znajdziesz w następnym rozdziale.
Usunięcie : z krótkiej deklaracji zmiennej	<pre>quantity = 4</pre>	Jeśli zapomnisz znaku :, instrukcja zostanie uznana za przypisanie, nie za deklarację, a nie można przypisywać wartości do niezadeklarowanych zmiennych.
Przypisanie wartości typu string do zmiennej typu int	<pre>quantity := 4 quantity = a</pre>	Do zmiennych można przypisywać wyłącznie wartości tego samego typu.
Niedopasowana liczba zmiennych i wartości	<pre>length, width := 1.2</pre>	Musisz podać wartość dla każdej zmiennej i zmienną dla każdej wartości.
Usunięcie kodu korzystającego ze zmiennej	<pre>fmt.Println(customerName)</pre>	Wszystkie zadeklarowane zmienne muszą być używane w programie. Jeśli usuniesz kod używający zmiennej, musisz też wyeliminować deklarację.

Reguły tworzenia nazw

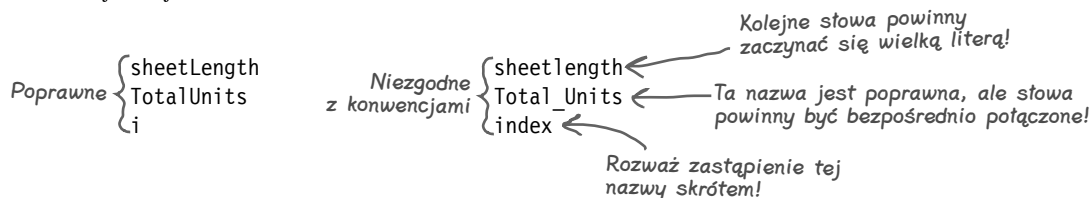
W Go obowiązują prosty zestaw reguł tworzenia nazw zmiennych, funkcji i typów:

- Nazwa musi zaczynać się literą, a dalej może obejmować dowolną liczbę liter i cyfr.
- Jeśli nazwa zmiennej, funkcji lub typu rozpoczyna się wielką literą, dany element jest **eksportowany** i można go używać także w pakietach innych niż bieżący. To dlatego w nazwie `fmt.Println` stosowane jest wielkie P. Dzięki temu funkcji można używać w `main` i dowolnym innym pakiecie. Jeśli nazwa zmiennej, funkcji lub typu rozpoczyna się małą literą, dany element **nie jest eksportowany** i można go stosować tylko w bieżącym pakiecie.



Są to jedyne reguły wymuszane przez język. Jednak społeczność użytkowników języka Go przestrzega też dodatkowych konwencji:

- Jeśli nazwa składa się z wielu słów, każde słowo po pierwszym powinno być zapisane wielką literą. Ponadto słowa należy łączyć bez stosowania spacji. Oto przykłady: `topPrice`, `RetryConnect` i `ion` itd. Pierwsza litera nazwy powinna być wielka tylko wtedy, jeśli chcesz eksportować dany element poza pakiet. Ten styl jest często nazywany *notacją wielbłądzą*, ponieważ wielkie litery wyglądają jak garby wielbłąda.
- Gdy znaczenie nazwy w danym kontekście jest oczywiste, społeczność użytkowników Go zwyczajowo skraca nazwę, używając `i` zamiast `index`, `max` zamiast `maximum` itd. Jednak autorzy serii *Rusz głową* uważają, że w trakcie nauki nowego języka nic nie jest oczywiste, dlatego w tej książce ta konwencja *nie* jest stosowana.



Tylko zmienne, funkcje i typy, których nazwy rozpoczynają się wielką literą, są eksportowane i stają się dostępne także w pakietach innych niż bieżący.

Konwersje

Operacje matematyczne i porównania w Go wymagają używania wartości tego samego typu. Użycie wartości różnych typów spowoduje błąd przy próbie uruchomienia kodu.

Tworzenie zmiennej typu float64 → var length float64 = 1.2

Tworzenie zmiennej typu int → var width int = 2

fmt.Println("Powierzchnia wynosi", length*width)

fmt.Println("length > width?", length > width) ← ... lub w porównaniu...

Jeśli użyjesz jednocześnie typów float64 i int w operacji matematycznej...

zobaczysz błędy!

Błędy → `invalid operation: length * width (mismatched types float64 and int)`
`invalid operation: length > width (mismatched types float64 and int)`

To samo dotyczy przypisywania nowych wartości do zmiennych. Jeśli typ przypisywanej wartości nie pasuje do typu z deklaracji zmiennej, zgłoszony zostanie błąd.

Tworzenie zmiennej typu float64 → var length float64 = 1.2

Tworzenie zmiennej typu int → var width int = 2

length = width ←

fmt.Println(length)

Jeśli przypiszesz wartość typu int do zmiennej typu float64...

...zobaczysz błąd!

Błąd → `cannot use width (type int) as type float64 in assignment`

Rozwiązaniem jest zastosowanie **konwersji**, pozwalających przekształcać wartości jednego typu na inny typ. Wystarczy podać typ, na jaki chcesz przekształcić wartość, a następnie modyfikowaną wartość w nawiasie.

var myInt int = 2

float64 (myInt)

Typ, na który przeprowadzana jest konwersja

Przekształcana wartość

Wynikiem jest nowa wartość oczekiwanego typu. Oto co otrzymasz po wywołaniu funkcji TypeOf dla zmiennej całkowitoliczbowej i ponownie dla tej samej wartości po konwersji na typ float64:

var myInt int = 2
fmt.Println(reflect.TypeOf(myInt))
fmt.Println(reflect.TypeOf(float64(myInt)))

Bez konwersji

int
float64

← Typ został zmieniony

Z konwersją

Konwersje (ciąg dalszy)

Zaktualizujmy teraz przykładowy błędny kod, aby przekształcał wartość typu `int` na typ `float64` przed zastosowaniem tej wartości w operacjach matematycznych lub porównaniach z użyciem innych wartości typu `float64`.

```
var length float64 = 1.2
var width int = 2
fmt.Println("Powierzchnia wynosi", length*float64(width))
fmt.Println("length > width?", length > float64(width))
```

Konwersja wartości typu `int` na typ `float64` przed pomnożeniem jej przez inną wartość typu `float64`

Konwersja wartości typu `int` na typ `float64` przed porównaniem jej z inną wartością typu `float64`

```
Powierzchnia wynosi 2.4
length > width? false
```

Operacja matematyczna i porównanie przebiegają teraz prawidłowo!

Teraz spróbuj przekształcić wartość typu `int` na typ `float64` przed przypisaniem jej do zmiennej typu `float64`:

```
var length float64 = 1.2
var width int = 2
length = float64(width)
fmt.Println(length)
```

Konwersja wartości typu `int` na typ `float64` przed przypisaniem jej do zmiennej typu `float64`

2

Także tu po konwersji przypisanie kończy się sukcesem.

W trakcie konwersji zwróć uwagę na to, że może ona zmieniać wynikową wartość. Na przykład zmienne typu `float64` mogą przechowywać wartości ułamkowe, natomiast zmienne typu `int` tego nie robią. Gdy przekształcasz wartość typu `float64` na typ `int`, część ułamkowa jest pomijana! Może to zakłócać operacje wykonywane na wynikowej wartości.

```
var length float64 = 3.75
var width int = 5
width = int(length)
fmt.Println(width)
```

Ta konwersja powoduje pominięcie części ułamkowej!

3

Wynikowa wartość jest o 0,75 mniejsza!

Jednak o ile zachowasz ostrożność, znajdziesz konwersje niezbędne do pracy w języku Go. Umożliwiają one współdziałanie niekompatybilnych w innym scenariuszu typów.



Napisałeś w Go pokazany poniżej kod, który oblicza łączną cenę z podatkiem i ustala, czy masz wystarczająco dużo pieniędzy na zakup. Jednak próba zastosowania tego kodu w kompletnym programie kończy się błędem!

Ćwiczenie

```
var price int = 100
fmt.Println("Cena wynosi", price, "złotych.")

var taxRate float64 = 0.08
var tax float64 = price * taxRate
fmt.Println("Podatek wynosi", tax, "złotych.")

var total float64 = price + tax
fmt.Println("Łączna cena to", total, "złotych.")

var availableFunds int = 120
fmt.Println("Dostępne środki:", availableFunds, ".")
fmt.Println("Mieścisz się w budżecie?", total <= availableFunds)
```

↙ Błędy

```
invalid operation: price * taxRate (mismatched types int and float64)
invalid operation: price + tax (mismatched types int and float64)
invalid operation: total <= availableFunds (mismatched types float64 and int)
```

Uzupełnij luki, aby poprawić ten kod. Wyeliminuj błędy, by uzyskać oczekiwane dane wyjściowe. Wskazówka: przed operacjami matematycznymi lub porównaniami musisz przeprowadzić konwersję, dzięki czemu typy staną się kompatybilne.

```
var price int = 100
fmt.Println("Cena wynosi", price, "złotych.")

var taxRate float64 = 0.08
var tax float64 = _____
fmt.Println("Podatek wynosi", tax, "złotych.")

var total float64 = _____
fmt.Println("Łączna cena to", total, "złotych.")

var availableFunds int = 120
fmt.Println("Dostępne środki:", availableFunds, "złotych.")
fmt.Println("Mieścisz się w budżecie?", _____)
```

Oczekiwane dane wyjściowe



```
Cena wynosi 100 złotych.
Podatek wynosi 8 złotych.
Łączna cena to 108 złotych.
Dostępne środki: 120 złotych.
Mieścisz się w budżecie? true
```



Odpowiedź znajdziesz na stronie 30.

Instalowanie Go na komputerze

Narzędzie Go Playground jest świetnym sposobem na wypróbowanie języka. Jednak praktyczne zastosowania tego narzędzia są ograniczone. Nie możesz używać go na przykład do pracy z plikami. Nie istnieje też sposób na pobieranie w nim danych wejściowych od użytkownika w oknie terminala, co będzie potrzebne w następnym programie.

Dlatego na zakończenie tego rozdziału pobierz i zainstaluj Go na komputerze. Nie martw się, zespół odpowiedzialny za ten język sprawił, że jest to naprawdę proste. W większości systemów operacyjnych wystarczy uruchomić program instalacyjny i zadanie będzie wykonane!



- 1 Otwórz w przeglądarce stronę <https://golang.org>.
- 2 Kliknij odsyłacz uruchamiający pobieranie.
- 3 Wybierz pakiet instalacyjny dla swojego systemu operacyjnego. Pobieranie powinno rozpocząć się automatycznie.
- 4 Otwórz stronę z instrukcjami instalacji dla Twojego systemu operacyjnego (możliwe, że zostaniesz tam automatycznie przeniesiony po rozpoczęciu pobierania) i zastosuj się do przedstawionych tam wskazówek.
- 5 Otwórz nowe okno terminala lub wiersza poleceń.
- 6 Upewnij się, że język Go został zainstalowany. W tym celu wpisz **go version** po znaku zachęty i wciśnij klawisz *Return* lub *Enter*. Powinieneś zobaczyć komunikat z informacją o zainstalowanej wersji Go.



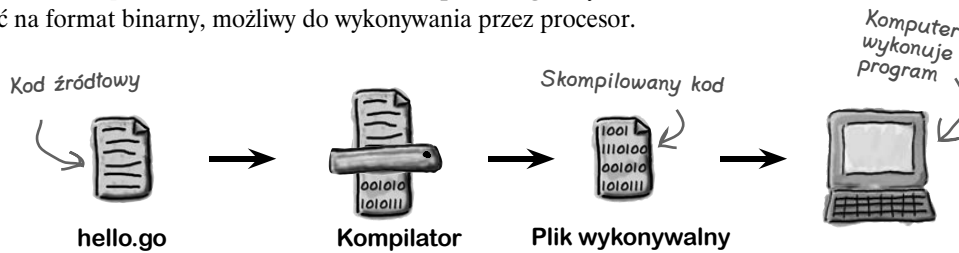
Obejrzyj to! **Witryny stale się zmieniają.**

Możliwe, że po wydaniu tej książki witryna <https://golang.org> lub instalator języka Go zostaną zmodyfikowane, a zaprezentowane tu kroki przestaną być w pełni precyzyjne. Wtedy odwiedź stronę: <http://headfirstgo.com> aby znaleźć pomoc i wskazówki związane z rozwiązywaniem problemów.

Kompilowanie kodu w języku Go

Interakcja z narzędziem Go Playground polegała na wpisywaniu kodu, który w magiczny sposób był wykonywany. Teraz, po zainstalowaniu Go w komputerze, pora bliżej przyjrzeć się jego działaniu.

Komputery nie potrafią bezpośrednio wykonywać kodu w języku Go. Najpierw trzeba wziąć plik z kodem źródłowym i **skompilować go**, czyli przekształcić na format binarny, możliwy do wykonywania przez procesor.



Wypróbuj teraz nowo zainstalowany język Go, aby skompilować i uruchomić wcześniejszy przykład wyświetlający tekst „Witaj, Go!”.



W wybranym edytorze tekstu zapisz kod programu „Witaj, Go!” w zwykłym pliku tekstowym o nazwie *hello.go*.


Otwórz nowe okno terminala lub wiersza poleceń.

W oknie terminala przejdź do katalogu, w którym zapisałeś plik *hello.go*.

Uruchom instrukcję `go fmt hello.go`, aby poprawić formatowanie kodu. Ten krok nie jest konieczny, ale warto go wykonać.

Uruchom instrukcję `go build hello.go`, aby skompilować kod źródłowy. Spowoduje to dodanie pliku wykonywalnego do bieżącego katalogu. W systemach macOS i Linux ten plik wykonywalny będzie miał nazwę *hello*. W systemie Windows ten plik to *hello.exe*.

Uruchom plik wykonywalny. W systemach macOS i Linux wpisz polecenie `./hello` (oznacza ono „uruchom program o nazwie *hello* z bieżącego katalogu”). W systemie Windows wpisz `hello.exe`.

Zapisz ten kod w pliku  **hello.go**

```
package main

import "fmt"

func main() {
    fmt.Println("Witaj, Go!")
}
```

Przejdźcie do katalogu, gdzie zapisany jest plik *hello.go*
Formatowanie kodu
Kompilowanie kodu
Uruchamianie pliku wykonywalnego

```
System Edycja Widok Okno Pomoc
$ cd try_go
$ go fmt hello.go
$ go build hello.go
$ ./hello
Witaj, Go!
$
```

Kompilowanie i uruchamianie pliku *hello.go* w systemach macOS i Linux

Przejdźcie do katalogu, gdzie zapisany jest plik *hello.go*
Formatowanie kodu
Kompilowanie kodu
Uruchamianie pliku wykonywalnego

```
Wiersz poleceń
>cd try_go
>go fmt hello.go
>go build hello.go
>hello.exe
Witaj, Go!
>
```

Kompilowanie i uruchamianie pliku *hello.go* w systemie Windows

Narzędzia języka Go

Po zainstalowaniu Go do wiersza poleceń dodawany jest program wykonywalny *go*. Ten plik wykonywalny zapewnia dostęp do różnych poleceń. Oto niektóre z nich:

Polecenie	Opis
<code>go build</code>	Kompiluje pliki z kodem źródłowym do postaci plików binarnych.
<code>go run</code>	Kompiluje i uruchamia program bez zapisywania pliku wykonywalnego.
<code>go fmt</code>	Formatuje pliki źródłowe zgodnie ze standardowym formatowaniem języka Go.
<code>go version</code>	Wyświetla zainstalowaną wersję języka Go.

Wypróbowałeś już polecenie `go fmt`, formatujące kod zgodnie ze standardowym formatowaniem języka Go. Jest to odpowiednik kliknięcia przycisku *Format* w witrynie Go Playground. Zachęcam do tego, by uruchamiać polecenie `go fmt` dla każdego tworzonego pliku z kodem źródłowym.

Użyłeś też polecenia `go build`, aby skompilować kod do postaci pliku wykonywalnego. Pliki wykonywalne tego rodzaju można udostępniać użytkownikom, którzy mogą uruchamiać je nawet bez zainstalowanego języka Go.

Nie wypróbowałeś jednak jeszcze polecenia `go run`. Zrób to teraz.


Większość edytorów można tak skonfigurować, by automatycznie uruchamiały polecenie `go fmt` przy każdym zapisie pliku. Zobacz stronę <https://blog.golang.org/go-fmt-your-code>

Szybkie sprawdzanie działania kodu za pomocą polecenia `go run`

Polecenie `go run` kompiluje i uruchamia plik z kodem źródłowym bez zapisywania pliku wykonywalnego w bieżącym katalogu. To polecenie doskonale się nadaje, by szybko sprawdzić działanie prostych programów. Użyj `go` do uruchomienia przykładowego programu *hello.go*.

- 1 Otwórz nowe okno terminala lub wiersza poleceń.
- 2 W oknie terminala przejdź do katalogu, w którym zapisałeś program *hello.go*.
- 3 Wpisz `go run hello.go` i wciśnij klawisz *Enter* lub *Return*. Polecenie wygląda tak samo we wszystkich systemach operacyjnych.

Natychmiast zobaczysz dane wyjściowe programu. Jeśli wprowadzisz zmiany w kodzie źródłowym, nie będziesz musiał wykonywać odrębnego kroku kompilacji. Wystarczy uruchomić kod za pomocą polecenia `go run`, a od razu zobaczysz wyniki. Gdy pracujesz nad małymi programami, polecenie `go run` jest przydatnym narzędziem.



```
package main

import "fmt"

func main() {
    fmt.Println("Witaj, Go!")
}
```

hello.go

Przejdźcie do katalogu, gdzie zapisany jest plik *hello.go*

Uruchamianie pliku z kodem źródłowym

```
System Edycja Widok Okno Pomoc
$ cd try_go
$ go run hello.go
Witaj, Go!
$
```

Uruchamianie pliku *hello.go* za pomocą polecenia `go run` (działa w każdym systemie operacyjnym)



Twój przyborek do Go

To już koniec rozdziału 1.!
Do swojego przyborenika dodałeś
wywołania funkcji i typy.

Wywołania funkcji

Funkcja to fragment kodu,
który można wywoływać
w innych miejscach programu.

Gdy wywołujesz funkcję,
możesz używać argumentów,
aby przekazywać funkcji dane.

Typy

Wartości w Go mają przypisane
różne typy, określające, do czego
można używać poszczególnych
wartości.

Operacje matematyczne
i porównania z użyciem wartości
różnych typów są niedozwolone,
jednak w razie potrzeby możesz
dokonać konwersji wartości na
nowy typ.

Zmienne w Go mogą
przechowywać wyłącznie wartości
zadeklarowanych typów.



CELNE SPOSTRZEŻENIA

- **Pakiet** to grupa powiązanych funkcji i innego kodu.
- Zanim będziesz mógł użyć funkcji z pakietu w pliku Go, musisz **zaimportować** ten pakiet.
- Łańcuch znaków (typ `string`) to seria bajtów reprezentujących zwykle znaki tekstowe.
- Runy (typ `rune`) reprezentują pojedyncze znaki tekstowe.
- Dwa najczęściej stosowane typy liczbowe w Go to `int` (przechowuje liczby całkowite) i `float64` (przechowuje liczby zmiennoprzecinkowe).
- Typ logiczny (`bool`) przechowuje wartości logiczne — `true` lub `false`.
- **Zmienna** to fragment pamięci, który może przechowywać wartości określonego typu.
- Jeśli do zmiennej nie przypisano żadnej wartości, będzie ona zawierać **wartość zerową** dla danego typu. Przykładowe wartości zerowe to `0` dla zmiennych typów `int` i `float64` oraz `""` dla zmiennych typu `string`.
- Można w jednej instrukcji zadeklarować zmienną i przypisać do niej wartość, używając **krótkiej deklaracji zmiennej** (`:=`).
- Zmienne, funkcje i typy mogą być używane w kodzie innych pakietów tylko wtedy, jeśli ich nazwy rozpoczynają się wielką literą.
- Polecenie `go fmt` automatycznie zmienia formatowanie plików źródłowych zgodnie ze standardowym formatowaniem języka Go. Powinieneś wywoływać polecenie `go fmt` dla każdego kodu, który zamierzasz udostępnić innym.
- Polecenie `go build` **kompiluje** kod źródłowy w Go do postaci binarnej, która może być wykonywana w komputerach.
- Polecenie `go run` kompiluje i uruchamia program bez zapisywania pliku wykonywalnego w bieżącym katalogu.

Basenowa układanka — rozwiązanie



Twoje **zadanie** polega na tym, by wziąć fragmenty kodu z basenu i umieścić je w pustych wierszach w kodzie. **Nie** używaj tych samych fragmentów więcej niż raz. Nie musisz wykorzystać wszystkich fragmentów. Twoim **celem** jest opracowanie kodu, który będzie działał i zwrócił pokazane dane wyjściowe.

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Skok na bombę!!!")
}
```

Dane wyjściowe

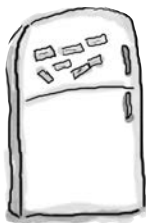
Skok na bombę!!!



Narysuj linie, aby dopasować każdy fragment kodu do typu.
Do niektórych typów prowadzić będzie kilka linii.

Ćwiczenie Rozwiązanie

```
reflect.TypeOf(25) ————— int
reflect.TypeOf(true) ————— float64
reflect.TypeOf(5.2) ————— bool
reflect.TypeOf(1) ————— string
reflect.TypeOf(false) ————— int
reflect.TypeOf(1.0) ————— float64
reflect.TypeOf("hello") ————— bool
```



Magnesiki z kodem — rozwiązanie

Program w Go jest rozsypany po całej lodówce. Czy potrafisz odtworzyć fragmenty kodu, aby uzyskać działający program, który wygeneruje pokazane dane wyjściowe?

```

package main

import (
    "fmt"
)

func main() {
    var originalCount int = 10
    fmt.Println("Miałem", originalCount, "jabłek.")
    var eatenCount int = 4
    fmt.Println("Jakiś łobuz zjadł", eatenCount, "jabłka.")
    fmt.Println("Zostało mi", originalCount-eatenCount, "jabłek.")
}

```

Dane wyjściowe

```

Miałem 10 jabłek.
Jakiś łobuz zjadł 4 jabłka.
Zostało mi 6 jabłek.

```



Ćwiczenie Rozwiązanie

Uzupełnij luki, aby poprawić kod. Wyeliminuj błędy, by uzyskać oczekiwane dane wyjściowe. Wskazówka: przed operacjami matematycznymi lub porównaniami musisz przeprowadzić konwersję, dzięki czemu typy staną się kompatybilne.

```

var price int = 100
fmt.Println("Cena wynosi", price, "złotych.")

var taxRate float64 = 0.08
var tax float64 = float64(price) * taxRate
fmt.Println("Podatek wynosi", tax, "złotych.")

var total float64 = float64(price) + tax
fmt.Println("Łączna cena to", total, "złotych.")

var availableFunds int = 120
fmt.Println("Dostępne środki:", availableFunds, "złotych.")
fmt.Println("Mieścisz się w budżecie?", total <= float64(availableFunds) )

```

Oczekiwane dane wyjściowe

```

Cena wynosi 100 złotych.
Podatek wynosi 8 złotych.
Łączna cena to 108 złotych.
Dostępne środki: 120 złotych.
Mieścisz się w budżecie? true

```

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

Rusz głową!

Go



Język Go, zwany także golangiem, został opracowany w firmie Google i zaprezentowany światu w 2009 roku. Zaprojektowano go pod kątem wydajności przetwarzania sieciowego i wieloprocesorowego. Autorzy chcieli, aby łączył łatwość pisania aplikacji z wydajnością języków kompilowanych. Podobnie jak JavaScript czy Python, Go jest językiem, który można szybko zrozumieć, a dzięki temu bezzwłocznie zacząć tworzyć funkcjonalny kod. Niemniej, aby zyskać uznanie potencjalnego pracodawcy i swojego nowego zespołu, poza wiedzą o składni i instrukcjach sterujących oraz praktyczną umiejętnością kodowania trzeba poznać określone konwencje i techniki.

Książka została przygotowana zgodnie z najnowszymi odkryciami nauk poznawczych, teorii uczenia się i neurofizjologii. Oznacza to tyle, że dzięki niej będziesz się uczyć zgodnie z zasadami pracy swojego mózgu: zaangażujesz umysł, wykorzystasz wiele zmysłów i niepostrzeżenie przyswoisz język programowania Go. Innymi słowy: w naturalny sposób zaczniesz programować! Niecodzienny wygląd i struktura książki sprawiają, że zamiast klasycznego podręcznika otrzymujesz polisensoryczne doświadczenie poznawcze, zaprojektowane tak, aby uzyskać umiejętności przydatne każdemu deweloperowi! Nawet jeśli musisz posługiwać się innymi językami programowania, dzięki tej pozycji nauczysz się technik i praktyk, które będziesz stale wykorzystywać podczas kodowania!

W tej książce między innymi:

- solidne podstawy tworzenia kodu, który będzie przejrzysty i łatwy w utrzymaniu
- metody, funkcje, pakiety...
- testowanie kodu i obsługa błędów
- dynamiczne aplikacje internetowe
- szablony HTML

Go:
rusz głową
i programuj!

Jay McGavren – jest trenerem programowania w serwisie Treehouse i autorem kilku innych książek z serii *Rusz głową!* Ma talent do prostego wyjaśniania skomplikowanych zagadnień. Występował na takich konferencjach jak RubyConf czy OSCON.

	<p>Sprawdź nasze szkolenia!</p>  <p>AKADEMIA IT & BUSINESS</p>	<p>KOD KORZYŚCI Sięgnij po więcej! ▶</p> 
<p> helion.pl</p>	<p>WWW.SZKOLENIA.HELION.PL</p>	<p>ISBN 978-83-283-6152-2</p>  <p>9 788328 361522</p>
<p> HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl</p>		<p>Cena: 99,00 zł</p>
<p>INFORMATYKA W NAJLEPSZYM WYDANIU</p>		