WYDANIE II

PROGRAMOWANIE W JĘZYKU RUST

OFICJALNY PODRĘCZNIK

STEVE KLABNIK, CAROL NICHOLS

OD WERSJI 1.62

Tytuł oryginału: The Rust Programming Language, 2nd Edition

Tłumaczenie: Lech Lachowski

ISBN: 978-83-289-1010-2

Copyright © 2023 by the Rust Foundation and the Rust Project Developers.
Title of English-language original: The Rust Programming Language, 2nd Edition, ISBN 9781718503106, published by No Starch Press Inc. 245 8th Street, San Francisco, California United States 94103. The Polish-language 2nd edition Copyright © 2024 by Helion S.A. under license by No Starch Press Inc. All rights reserved.

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 jakiejkolwiek 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 wydawca 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 wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/prruo2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

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)

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

PRZEDMOWA **17**

WSTĘP **19**

PODZIĘKOWANIA **20**

WPROWADZENIE **21**

1
PIERWSZE KROKI **27**

Instalacja 27
- Instalowanie narzędzia rustup w systemie Linux lub macOS 28
- Instalowanie narzędzia rustup w systemie Windows 29
- Rozwiązywanie problemów 29
- Aktualizowanie i odinstalowywanie 30
- Lokalna dokumentacja 30

Witaj, świecie! 30
- Tworzenie katalogu projektu 31
- Pisanie i uruchamianie programu Rusta 31
- Anatomia programu Rusta 32
- Kompilowanie i uruchamianie to oddzielne kroki 33

Witaj, Cargo! 34
- Tworzenie projektu za pomocą Cargo 35
- Kompilowanie i uruchamianie projektu Cargo 37
- Kompilowanie w celu wydania projektu 38
- Cargo jako konwencja 39

Podsumowanie 39

2
PROGRAMOWANIE GRY W ZGADYWANIE **40**

Konfigurowanie nowego projektu 41

Przetwarzanie zgadywania 42
- Przechowywanie wartości za pomocą zmiennych 43
- Odbieranie danych wprowadzanych przez użytkownika 44

Kup ksi k Pole ksi k

Obsługa ewentualnego niepowodzenia za pomocą typu Result 44
Wypisywanie wartości za pomocą symboli zastępczych println! 46
Testowanie pierwszej części 46
Generowanie sekretnej liczby 47
Korzystanie ze skrzynki w celu uzyskania większej funkcjonalności 47
Generowanie liczby losowej 50
Porównywanie próby odgadnięcia z sekretną liczbą 51
Umożliwianie wielokrotnego zgadywania dzięki zastosowaniu pętli 55
Zakończenie gry po odgadnięciu prawidłowej liczby 56
Obsługa nieprawidłowych danych wejściowych 56
Podsumowanie 59

3
POWSZECHNE KONCEPCJE PROGRAMOWANIA 60
Zmienne i mutowalność 61
Stałe 63
Przysłanianie 63
Typy danych 65
Typy skalarne 66
Typy złożone 70
Funkcje 74
Parametry 75
Instrukcje i wyrażenia 76
Funkcje z wartościami zwracanymi 78
Komentarze 80
Przepływ sterowania 81
Wyrażenia if 81
Powtarzanie za pomocą pętli 85
Podsumowanie 90

4
WŁASNOŚĆ 91
Czym jest własność? 91
Reguły własności 93
Zakres zmiennych 93
Typ String 94
Pamięć i alokacja 95
Własność i funkcje 101
Wartości zwracane i zakres 101
Referencje i pożyczanie 103
Referencje mutowalne 105
Referencje wiszące 108
Reguły referencji 109

Kup ksi k Pole ksi k

Typ wycinka 110
Wycinki łańcucha znaków 111
Inne wycinki 116
Podsumowanie 116

5
WYKORZYSTANIE STRUKTUR DO ORGANIZOWANIA POWIĄZANYCH ZE SOBĄ DANYCH 117
Definiowanie struktur i tworzenie ich instancji 118
Użycie skrótu inicjowania pól 119
Tworzenie instancji z innych instancji za pomocą składni aktualizacji struktury 120
Tworzenie różnych typów za pomocą struktur krotkowych bez nazwanych pól 121
Struktury jednostkowe bez żadnych pól 122
Przykładowy program wykorzystujący struktury 124
Refaktoryzacja z wykorzystaniem krotek 125
Refaktoryzacja z wykorzystaniem struktur: dodanie znaczenia 126
Dodanie użytecznej funkcjonalności za pomocą cech pochodnych 127
Składnia metod 130
Definiowanie metod 130
Metody z większą liczbą parametrów 133
Funkcje powiązane 134
Wiele bloków impl 135
Podsumowanie 136

6
TYPY WYLICZENIOWE I DOPASOWYWANIE WZORCÓW 137
Definiowanie typu wyliczeniowego 138
Wartości wyliczenia 138
Enumeracja Option i jej zalety w porównaniu z wartościami zerowymi 142
Konstrukcja match przepływu sterowania 145
Wzorce, które powodują powiązanie z wartościami 146
Dopasowywania za pomocą Option<T> 147
Dopasowywania są wyczerpujące 149
Wzorce catch-all i symbol zastępczy _ 149
Zwięzły przepływ sterowania z wykorzystaniem składni if let 151
Podsumowanie 152

Kup ksi k Pole ksi k

7
ZARZĄDZANIE ROZWIJAJĄCYMI SIĘ PROJEKTAMI ZA POMOCĄ PAKIETÓW, SKRZYNEK I MODUŁÓW 154

Pakiety i skrzynki 155
Definiowanie modułów w celu kontrolowania zakresu i prywatności 159
Ścieżki odwoływania się do elementów w drzewie modułów 160
Udostępnianie ścieżek za pomocą słowa kluczowego pub 163
Rozpoczynanie ścieżek względnych od słowa kluczowego super 166
Upublicznianie struktur i wyliczeń 167
Wprowadzanie ścieżek do zakresu za pomocą słowa kluczowego use 168
Tworzenie idiomatycznych ścieżek use 170
Wprowadzanie nowych nazw za pomocą słowa kluczowego as 171
Ponowne eksportowanie nazw za pomocą pub use 172
Korzystanie z pakietów zewnętrznych 173
Skracanie długich list instrukcji use za pomocą zagnieżdżonych ścieżek 174
Operator glob 175
Rozdzielanie modułów na różne pliki 175
Podsumowanie 177

8
POWSZECHNIE UŻYWANE KOLEKCJE 179

Przechowywanie list wartości za pomocą wektorów 180
Tworzenie nowego wektora 180
Aktualizowanie wektora 181
Odczytywanie elementów wektorów 181
Iterowanie przez wartości w wektorze 183
Przechowywanie wielu typów za pomocą wyliczenia 184
Porzucenie wektora powoduje porzucenie jego elementów 185
Wykorzystywanie typu String do przechowywania tekstu zakodowanego w UTF-8 185
Czym jest String? 186
Tworzenie nowej instancji String 186
Aktualizowanie instancji String 187
Indeksowanie wartości String 190
Tworzenie wycinków wartości String 192
Metody iterowania przez wartości String 192
Typy String nie są takie proste 193
Przechowywanie kluczy z powiązanymi wartościami w mapach mieszających 193
Tworzenie nowej instancji HashMap 194
Uzyskiwanie dostępu do wartości z mapy mieszającej 194
Mapy mieszające i własność 195
Aktualizowanie instancji HashMap 196
Funkcje mieszające 198
Podsumowanie 198

Kup ksi k Pole ksi k

9
OBSŁUGA BŁĘDÓW ... 200
Obsługa błędów nieodwracalnych za pomocą wywołania panic! ... 201
Obsługa błędów odwracalnych z wykorzystaniem typu Result ... 204
Dopasowywanie na podstawie różnych błędów ... 206
Propagacja błędów ... 209
Panikować czy nie panikować? ... 215
Przykłady, kod prototypów i testy ... 216
Przypadki, w których mamy więcej informacji niż kompilator ... 216
Wskazówki dotyczące obsługi błędów ... 217
Tworzenie typów niestandardowych do celów walidacji ... 218
Podsumowanie ... 220

10
TYPY SPARAMETRYZOWANE, CECHY I CZASY ŻYCIA ... 221
Redukowanie duplikacji przez wyodrębnianie funkcji ... 222
Sparametryzowane typy danych ... 225
Typy sparametryzowane w definicjach funkcji ... 225
Typy sparametryzowane w definicjach struktur ... 227
Typy sparametryzowane w definicjach wyliczeń ... 229
Typy sparametryzowane w definicjach metod ... 230
Wydajność kodu wykorzystującego typy sparametryzowane ... 232
Cechy — definiowanie współdzielonego zachowania ... 233
Definiowanie cechy ... 233
Implementowanie cechy w typie ... 234
Implementacje domyślne ... 236
Cechy jako parametry ... 238
Zwracanie typów, które implementują cechy ... 240
Używanie wiązań cech do warunkowego implementowania metod ... 241
Walidacja referencji za pomocą czasów życia ... 243
Używanie czasów życia do zapobiegania powstawaniu wiszących referencji ... 243
Kontrola pożyczania ... 244
Sparametryzowane czasy życia w funkcjach ... 245
Składnia adnotacji czasów życia ... 247
Adnotacje czasów życia w sygnaturach funkcji ... 247
Myślenie w kategoriach czasów życia ... 250
Adnotacje czasów życia w definicjach struktur ... 251
Elizja czasów życia ... 252
Adnotacje czasów życia w definicjach metod ... 254
Statyczny czas życia ... 255
Parametry typów sparametryzowanych, wiązania cech i czasy życia razem wzięte ... 256
Podsumowanie ... 256

Kup książkę Poleć książkę

11
PISANIE ZAUTOMATYZOWANYCH TESTÓW

PISANIE ZAUTOMATYZOWANYCH TESTÓW ... 258
Jak pisać testy? ... 259
  Anatomia funkcji testowej ... 259
  Sprawdzenie wyników za pomocą makra assert! ... 263
  Testowanie równości za pomocą makr assert_eq! i assert_ne! ... 266
  Dodawanie niestandardowych komunikatów o błędach ... 268
  Sprawdzanie paniki za pomocą atrybutu should_panic ... 270
  Wykorzystanie w testach typu Result<T, E> ... 273
Kontrolowanie sposobu uruchamiania testów ... 274
  Uruchamianie testów równolegle lub po kolei ... 274
  Wyświetlanie danych wyjściowych funkcji ... 275
  Uruchamianie podzbioru testów według nazwy ... 277
  Ignorowanie niektórych testów i uruchamianie ich na specjalne żądanie ... 279
Organizowanie testów ... 280
  Testy jednostkowe ... 280
  Testy integracyjne ... 282
Podsumowanie ... 286

12

PROJEKT Z WYKORZYSTANIEM OPERACJI WE-WY — BUDOWANIE PROGRAMU WIERSZA POLECEŃ ... 287
Przyjmowanie argumentów wiersza poleceń ... 288
  Odczytywanie wartości argumentów ... 289
  Zapisywanie wartości argumentów w zmiennych ... 290
Odczyt pliku ... 291
Refaktoryzacja w celu poprawy modułowości i obsługi błędów ... 293
  Separacja zagadnień projektów binarnych ... 293
  Naprawianie obsługi błędów ... 297
  Wyodrębnianie logiki z funkcji main ... 301
  Wydzielanie kodu do skrzynki biblioteki ... 303
Rozwijanie funkcjonalności biblioteki za pomocą programowania opartego na testach ... 305
  Pisanie testu kończącego się niepowodzeniem ... 305
  Pisanie kodu w celu zapewnienia pozytywnego wyniku testu ... 308
Praca ze zmiennymi środowiskowymi ... 311
  Pisanie niepomyślnego testu dla funkcji search nieuwzględniającej wielkości liter ... 311
  Implementacja funkcji search_case_insensitive ... 313
Wypisywanie komunikatów o błędach do standardowego strumienia błędów, a nie do standardowego strumienia wyjściowego ... 317
  Sprawdzanie, gdzie są zapisywane błędy ... 317
  Wypisywanie błędów do standardowego strumienia błędów ... 318
Podsumowanie ... 319

13
CECHY JĘZYKA FUNKCYJNEGO — ITERATORY I DOMKNIĘCIA 320

Domknięcia — funkcje anonimowe, które przechwytują swoje środowisko 321
    Przechwytywanie środowiska za pomocą domknięć 321
    Inferowanie i adnotowanie typu domknięcia 323
    Przechwytywanie referencji lub przenoszenie własności 325
    Przenoszenie przechwyconych wartości z domknięć i cechy Fn 327
Przetwarzanie serii elementów za pomocą iteratorów 331
    Cecha Iterator i metoda next 332
    Metody, które konsumują iterator 333
    Metody, które wytwarzają inne iteratory 334
    Używanie domknięć, które przechwytują swoje środowisko 335
Ulepszenie projektu operacji we-wy 337
    Usuwanie clone przy użyciu iteratora 337
    Zwiększanie czytelności kodu dzięki adapterom iteratora 340
    Wybór pomiędzy pętlami i iteratorami 341
Porównanie wydajności pętli i iteratorów 341
Podsumowanie 343

14
NARZĘDZIE CARGO I REJESTR CRATES.IO 344

Dostosowywanie kompilacji za pomocą profili wydania 345
Publikowanie skrzynki w rejestrze Crates.io 346
    Tworzenie przydatnych komentarzy dokumentujących 346
    Eksportowanie wygodnego publicznego API za pomocą pub use 350
    Zakładanie konta w rejestrze Crates.io 353
    Dodawanie metadanych do nowej skrzynki 354
    Publikowanie w rejestrze Crates.io 355
    Publikowanie nowej wersji istniejącej skrzynki 356
    Dezaktualizowanie wersji z Crates.io za pomocą polecenia cargo yank 356
Obszary robocze Cargo 357
    Tworzenie obszaru roboczego 357
    Tworzenie drugiego pakietu w obszarze roboczym 358
Instalowanie plików binarnych za pomocą polecenia cargo install 363
Rozszerzanie Cargo przy użyciu niestandardowych poleceń 364
Podsumowanie 364

15
INTELIGENTNE WSKAŹNIKI 365

Używanie Box<T> do wskazywania danych na stercie 366
    Używanie Box<T> do przechowywania danych na stercie 367
    Zastosowanie boksów do używania typów rekurencyjnych 367

Wykorzystanie cechy Deref do traktowania inteligentnych wskaźników jak regularnych referencji 372
Podążanie za wskaźnikiem do wartości 372
Używanie Box<T> jak referencji 373
Definiowanie własnego inteligentnego wskaźnika 374
Implementacja cechy Deref 375
Domyślne wymuszenie wyłuskania w funkcjach i metodach 376
Jak wymuszenie wyłuskania współdziała z mutowalnością? 378
Wykorzystanie cechy Drop do uruchamiania określonego kodu przy czyszczeniu 378
Rc<T> — inteligentny wskaźnik zliczania referencji 382
Używanie Rc<T> do udostępniania danych 382
Klonowanie Rc<T> zwiększa liczbę referencji 385
RefCell<T> i wzorzec mutowalności wewnętrznej 386
Egzekwowanie reguł pożyczania w czasie wykonywania za pomocą RefCell<T> 386
Mutowalność wewnętrzna — mutowalne pożyczanie niemutowalnej wartości 388
Wykorzystanie typów Rc<T> i RefCell<T>, aby mutowalne dane mogły mieć wielu właścicieli 394
Cykle referencji mogą powodować wycieki pamięci 395
Tworzenie cyklu referencji 396
Zapobieganie cyklom referencji za pomocą typu Weak<T> 399
Podsumowanie 404

16
NIEUSTRASZONA WSPÓŁBIEŻNOŚĆ 405
Używanie wątków do jednoczesnego uruchomienia kodu 406
Tworzenie nowego wątku za pomocą funkcji spawn 407
Wykorzystanie uchwytów metody join w celu zaczekania na zakończenie wszystkich wątków 408
Używanie domknięć move z wątkami 410
Używanie przekazywania komunikatów do przesyłania danych między wątkami 413
Kanały i przeniesienie własności 416
Wysyłanie wielu wartości i obserwowanie czekającego odbiornika 417
Tworzenie wielu producentów przez klonowanie nadajnika 418
Współbieżność stanu współdzielonego 420
Używanie muteksów w celu umożliwienia dostępu do danych z jednego wątku na raz 420
Podobieństwa między typami RefCell<T> i Rc<T> a Mutex<T> i Arc<T> 426
Rozszerzanie funkcjonalności współbieżności za pomocą cech Send i Sync 426
Umożliwienie przenoszenia własności między wątkami za pomocą cechy Send 427
Umożliwianie dostępu z wielu wątków za pomocą cechy Sync 427
Ręczne implementowanie cech Send i Sync jest niezabezpieczone 428
Podsumowanie 428

Kup ksi k Pole ksi k

17
FUNKCJONALNOŚCI PROGRAMOWANIA OBIEKTOWEGO 429

Cechy języków obiektowych 430
Obiekty zawierają dane i zachowania 430
Hermetyzacja, która ukrywa szczegóły implementacji 430
Dziedziczenie jako system typów i współdzielenie kodu 432
Używanie obiektów cech, które umożliwiają stosowanie wartości różnych typów 433
Definiowanie cech typowego zachowania 434
Implementowanie cechy 436
Obiekty cech wykonują dynamiczną dyspozycję 439
Implementacja obiektowego wzorca projektowego 439
Definiowanie struktury Post i tworzenie nowej instancji w stanie Draft 441
Przechowywanie tekstu treści wpisu 442
Upewnianie się, że treść wersji roboczej wpisu jest pusta 442
Żądanie korekty zmienia stan wpisu 443
Dodanie metody approve w celu zmiany zachowania metody content 445
Kompromisy wzorca stanu 447
Podsumowanie 452

18
WZORCE I DOPASOWYWANIE 453

Wszystkie miejsca, w których można używać wzorców 454
Ramiona wyrażenia match 454
Wyrażenia warunkowe if let 455
Pętle warunkowe while let 456
Pętle for 457
Instrukcje let 457
Parametry funkcji 458
Odrzucalność — czy dopasowywanie wzorca może się nie powieść? 459
Składnia wzorców 461
Dopasowywanie literałów 462
Dopasowywanie zmiennych nazwanych 462
Wiele wzorców 463
Dopasowywanie przedziałów wartości za pomocą ..= 464
Destrukturyzacja w celu rozkładania wartości na poszczególne elementy 464
Ignorowanie wartości we wzorcu 469
Dodatkowe wyrażenia warunkowe ze strażnikami dopasowywania 473
Wiązanie za pomocą operatora @ 476
Podsumowanie 477

Kup ksi k Pole ksi k

19
FUNKCJONALNOŚCI ZAAWANSOWANE

FUNKCJONALNOŚCI ZAAWANSOWANE **478**
Niezabezpieczony Rust 479
Niezabezpieczone supermoce 479
Wyłuskiwanie surowego wskaźnika 480
Wywoływanie niezabezpieczonej funkcji lub metody 482
Uzyskiwanie dostępu do mutowalnej zmiennej statycznej lub modyfikowanie jej 487
Implementowanie niezabezpieczonej cechy 488
Uzyskiwanie dostępu do pól typu unii 489
Kiedy korzystać z niezabezpieczonego kodu? 489
Cechy zaawansowane 489
Typy powiązane 489
Domyślne parametry typu sparametryzowanego i przeciążanie operatora 491
Rozróżnianie metod o tej samej nazwie 493
Korzystanie z supercech 497
Używanie wzorca newtype do implementowania cech zewnętrznych 499
Typy zaawansowane 500
Używanie wzorca newtype
dla zapewnienia bezpieczeństwa typów i warstwy abstrakcji 500
Tworzenie synonimów typów za pomocą aliasów typów 501
Typ never, który nigdy nie zwraca wartości 503
Typy o dynamicznie ustalanych rozmiarach i cecha Sized 505
Zaawansowane funkcje i domknięcia 507
Wskaźniki funkcji 507
Zwracanie domknięć 509
Makra 510
Różnica między makrami i funkcjami 510
Makra deklaratywne z konstrukcją macro_rules!
dla metaprogramowania ogólnego 511
Makra proceduralne do generowania kodu z atrybutów 513
Jak napisać niestandardowe makro derive? 514
Makra atrybutowe 519
Makra funkcyjne 519
Podsumowanie 520

20
PROJEKT KOŃCOWY — BUDOWA WIELOWĄTKOWEGO SERWERA WWW

PROJEKT KOŃCOWY — BUDOWA WIELOWĄTKOWEGO SERWERA WWW **521**
Tworzenie jednowątkowego serwera WWW 522
Nasłuchiwanie połączenia TCP 523
Odczytywanie żądania 525
Analiza żądania HTTP 527
Pisanie odpowiedzi 527
Zwracanie prawdziwego HTML-a 529

Kup ksi k Pole ksi k

Walidacja żądania i selektywne odpowiadanie ... 530
Odrobina refaktoryzacji ... 532
Przekształcenie serwera jednowątkowego w wielowątkowy ... 533
Symulowanie powolnego żądania ... 533
Zwiększenie przepustowości przy użyciu puli wątków ... 534
Eleganckie zamykanie i czyszczenie ... 551
Implementacja cechy Drop w typie ThreadPool ... 551
Sygnalizowanie wątkom, aby przestały nasłuchiwać zadań ... 554
Podsumowanie ... 557

A
SŁOWA KLUCZOWE ... 559
Obecnie używane słowa kluczowe ... 559
Słowa kluczowe zarezerwowane do wykorzystania w przyszłości ... 561
Surowe identyfikatory ... 561

B
OPERATORY I SYMBOLE ... 563
Operatory ... 563
Symbole niezwiązane z operatorami ... 566

C
CECHY POCHODNE (ZAPOŻYCZONE) ... 571
Cecha Debug do uzyskiwania programistycznych danych wyjściowych ... 572
Cechy PartialEq i Eq do porównania równości ... 572
Cechy PartialOrd i Ord do porównywania kolejności ... 573
Cechy Clone i Copy do duplikowania wartości ... 573
Cecha Hash do mapowania wartości na wartość o stałym rozmiarze ... 574
Cecha Default dla wartości domyślnych ... 575

D
PRZYDATNE NARZĘDZIA PROGRAMISTYCZNE ... 577
Automatyczne formatowanie za pomocą narzędzia rustfmt ... 577
Naprawianie kodu za pomocą narzędzia rustfix ... 578
Lintery narzędzia Clippy ... 579
Integracja z IDE przy użyciu narzędzia rust-analyzer ... 580

E
EDYCJE ... 581

Kup ksi k Pole ksi k

2

Programowanie gry w zgadywanie

Wskoczmy na głęboką wodę języka Rust i popracujmy wspólnie nad praktycznym projektem! W tym rozdziale wprowadzimy kilka typowych koncepcji Rusta, pokazując, jak używać ich w prawdziwym programie. Poznasz m.in. instrukcję `let`, wyrażenie `match`, metody, powiązane funkcje i zewnętrzne skrzynki! W kolejnych rozdziałach omówimy te koncepcje szerzej. W tym rozdziale poćwiczysz po prostu podstawy.

Zaimplementujemy klasyczny problem programowania dla początkujących: grę w zgadywanie. Działa to następująco. Program generuje losową liczbę całkowitą od 1 do 100. Potem prosi gracza o wpisanie typowanej liczby. Po wprowadzeniu liczby program wskazuje, czy jest ona zbyt niska, czy za wysoka. Jeśli gracz poprawnie odgadnie wygenerowaną liczbę, gra wyświetla wiadomość z gratulacjami i kończy działanie.

Kup ksi k

Pole ksi k

Konfigurowanie nowego projektu

Aby skonfigurować nowy projekt, przejdź do katalogu *projects* utworzonego w rozdziale 1. i utwórz nowy projekt przy użyciu Cargo w następujący sposób:

```
$ cargo new guessing_game
$ cd guessing_game
```

Pierwsze polecenie, `cargo new`, jako pierwszy argument przyjmuje nazwę projektu (*guessing_game*). Drugie polecenie powoduje przejście do katalogu nowego projektu.

Przyjrzyjmy się wygenerowanemu plikowi *Cargo.toml*:

Cargo.toml

```
[package]
name = "guessing_game"
version = "0.1.0"
edition = "2021"

# Więcej kluczy wraz z definicjami znajdziesz na stronie https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
```

Jak pokazaliśmy w rozdziale 1., `cargo new` generuje za nas program *Witaj, świecie!*. Sprawdź zawartość pliku *src/main .rs*:

src/main.rs

```
fn main() {
    println!("Witaj, świecie!");
}
```

Teraz w tym samym kroku skompilujmy i uruchommy program *Witaj, świecie!* za pomocą polecenia `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
     Running `target/debug/guessing_game`
Witaj, świecie!
```

Polecenie `run` przydaje się, gdy trzeba przeprowadzać szybkie iteracje projektu, tak jak zrobimy w tej grze, ekspresowo testując każdą iterację przed przejściem do następnej.

Otwórz ponownie plik *src/main.rs*. W tym pliku zostanie napisany cały kod.

Kup ksi k Pole ksi k

Przetwarzanie zgadywania

W pierwszej części programu gry w zgadywanie użytkownik jest proszony o wprowadzenie danych, następnie dane te są przetwarzane i następuje sprawdzenie, czy mają oczekiwany format. Na początek pozwalamy graczowi wprowadzić jego typowaną liczbę. Wpisz w pliku *src/main.rs* kod z listingu 2.1.

Listing 2.1. Kod, który pobiera i wpisuje liczbę wpisywaną przez użytkownika

src/main.rs
```
use std::io;

fn main() {
    println!("Zgadnij liczbę!");

    println!("Wpisz swój typ.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Nie udało się odczytać linii");

    println!("Twój typ to: {guess}");
}
```

Ten kod zawiera wiele informacji, przeanalizujmy go więc linia po linii. Aby uzyskać dane wejściowe użytkownika, a następnie wypisać wynik jako dane wyjściowe, musimy wprowadzić do zakresu bibliotekę we-wy `io`. Pochodzi ona ze standardowej biblioteki znanej jako `std`:

```
use std::io;
```

Domyślnie Rust ma zbiór elementów zdefiniowanych w standardowej bibliotece, które wprowadza do zakresu każdego programu. Zbiór ten nazywa się **prelude**, a informacje na temat wszystkiego, co się w nim znajduje, znajdziesz na stronie *https://doc.rust-lang.org/std/prelude/index.html*.

Jeżeli typ, którego chcemy użyć, nie znajduje się w `prelude`, musimy wprowadzić go do zakresu bezpośrednio za pomocą instrukcji `use`. Korzystanie z biblioteki `std::io` zapewnia wiele przydatnych funkcjonalności, w tym możliwość przyjmowania danych wprowadzanych przez użytkownika.

Jak pokazaliśmy w rozdziale 1., funkcja `main` jest punktem wejścia do programu:

```
fn main() {
```

Składnia `fn` deklaruje nową funkcję; nawiasy okrągłe (`()`) wskazują na brak parametrów, a nawias klamrowy (`{`) rozpoczyna ciało funkcji.

Kup ksi k Pole ksi k

W rozdziale 1. wspomnieliśmy również, że `println!` to makro, które wypisuje łańcuch znaków na ekranie:

```
println!("Zgadnij liczbę!");

println!("Wpisz swój typ.");
```

Ten kod wypisuje komunikat informujący, na czym polega gra, i prosi użytkownika o wprowadzenie danych.

Przechowywanie wartości za pomocą zmiennych

Następnie utworzymy **zmienną** do przechowywania danych wprowadzanych przez użytkownika:

```
let mut guess = String::new();
```

Teraz program zaczyna się robić interesujący! W tej krótkiej linii dużo się dzieje. Do utworzenia zmiennej używamy instrukcji `let`. Oto kolejny przykład:

```
let apples = 5;
```

Ta linia tworzy nową zmienną o nazwie `apples` i wiąże ją z wartością `5`. W języku Rust zmienne są domyślnie niemutowalne, co oznacza, że wartość nadana zmiennej nigdy się nie zmieni. Omówimy tę koncepcję szczegółowo w rozdziale 3., w podrozdziale „Zmienne i mutowalność". Aby uczynić zmienną mutowalną, dodajemy `mut` przed nazwą zmiennej:

```
let apples = 5; // Niemutowalna
let mut bananas = 5; // Mutowalna
```

Uwaga

Składnia `//` rozpoczyna komentarz, który ciągnie się do końca linii. Rust ignoruje wszystko, co zawierają komentarze. Komentarze omówimy szerzej w rozdziale 3.

Wróćmy do programu gry w zgadywanie. Wiesz już, że instrukcja `let mut guess` wprowadza mutowalną zmienną o nazwie `guess`. Znak równości (`=`) instruuje Rusta, że chcemy teraz ze zmienną powiązać jakąś wartość. Po prawej stronie znaku równości znajduje się wartość, z którą powiązana jest zmienna `guess`, czyli wynik wywołania `String::new` — funkcji zwracającej nową instancję typu `String`. Typ `String` to łańcuch znaków dostarczany przez standardową bibliotekę; jest to możliwy do rozwijania fragment tekstu zakodowany w formacie UTF-8.

Kup ksi k Pole ksi k

Składnia :: w linii ::new wskazuje, że new jest funkcją powiązaną z typem String. **Funkcja powiązana** to funkcja zaimplementowana w określonym typie, w tym przypadku typie String. Ta funkcja new tworzy nowy, pusty łańcuch znaków. Funkcję new można znaleźć w wielu typach, ponieważ jest to powszechna nazwa funkcji, która tworzy nową wartość określonego rodzaju.

W całości linia let mut guess = String::new(); spowodowała utworzenie zmiennej mutowalnej, która została powiązana z nową, pustą instancją typu String. Uff!

Odbieranie danych wprowadzanych przez użytkownika

Przypomnijmy, że za pomocą use std::io; w pierwszej linii programu wprowadziliśmy funkcjonalność we-wy ze standardowej biblioteki. Teraz wywołamy funkcję stdin z modułu io, która pozwoli nam na obsługę danych wprowadzanych przez użytkownika:

```
io::stdin()
    .read_line(&mut guess)
```

Gdybyśmy nie importowali biblioteki io za pomocą use std::io; na początku programu, nadal moglibyśmy używać tej funkcji, zapisując ją jako std::io::stdin. Funkcja stdin zwraca instancję std::io::Stdin, która jest typem reprezentującym uchwyt dla standardowego wejścia terminala.

Następnie linia .read_line(&mut guess) wywołuje metodę read_line w standardowym uchwycie wejścia, aby uzyskać dane wejściowe od użytkownika. Przekazujemy również &mut guess do metody read_line jako argument, aby poinstruować ją, w jakim łańcuchu znaków zapisać dane wejściowe użytkownika. Pełne zadanie read_line polega na pobraniu danych, które użytkownik wprowadzi do standardowego wejścia, i dołączeniu ich do łańcucha znaków (bez nadpisywania jego zawartości), więc przekazujemy ten łańcuch znaków jako argument. Argument łańcucha znaków musi być mutowalny, by metoda mogła zmieniać zawartość tego łańcucha.

Symbol & wskazuje, że ten argument jest **referencją**, która umożliwia dostęp wielu części kodu do jednego fragmentu danych bez konieczności wielokrotnego kopiowania tych danych do pamięci. Referencje są złożoną funkcjonalnością, a jedną z głównych zalet języka Rust jest bezpieczne i łatwe korzystanie z nich. Nie musisz znać wielu tych szczegółów, aby ukończyć ten program. Na razie wystarczy wiedzieć, że podobnie jak zmienne, referencje są domyślnie niemutowalne. W związku z tym musisz napisać &mut guess zamiast &guess, aby uczynić tę zmienną mutowalną. (Referencje omówimy szerzej w rozdziale 4.).

Obsługa ewentualnego niepowodzenia za pomocą typu Result

Nadal pracujemy nad tą linią kodu. Omówimy teraz trzecią linię tekstu, ale zwróć uwagę, że wciąż jest ona częścią pojedynczej logicznej linii kodu. Kolejny fragment to następująca metoda:

Kup książkę Poleć książkę

```
.expect("Nie udało się odczytać linii");
```

Mogliśmy napisać ten kod w taki sposób:

```
io::stdin().read_line(&mut guess).expect("Nie udało się odczytać linii");
```

Jednak jedna długa linia jest trudna do odczytania, więc najlepiej ją podzielić. Podczas wywoływania metody za pomocą składni `.nazwa_metody()` często rozsądnie jest wprowadzić znak nowej linii i kolejne znaki niedrukowalne w celu podzielenia długich linii. Omówmy teraz, co robi ta linia.

Jak wspomnieliśmy wcześniej, `read_line` wstawia do przekazywanego do niej łańcucha znaków wszystko, co wprowadza użytkownik, ale także zwraca wartość `Result`. Jest to typ **wyliczeniowy**, zwany często **enum**, który może znajdować się w jednym z wielu możliwych stanów. Każdy możliwy stan nazywamy **wariantem**.

Typy wyliczeniowe omówimy szerzej w rozdziale 6. Celem typów `Result` jest kodowanie informacji dotyczących obsługi błędów.

Wariantami `Result` są `Ok` i `Err`. Wariant `Ok` wskazuje, że operacja zakończyła się powodzeniem, a wewnątrz `Ok` znajduje się pomyślnie wygenerowana wartość. Wariant `Err` oznacza, że operacja się nie powiodła, a `Err` zawiera informacje o tym, w jaki sposób lub dlaczego do tego doszło.

Wartości typu `Result`, podobnie jak wartości każdego innego typu, mają zdefiniowane metody. Instancja `Result` ma metodę `expect`, którą można wywoływać. Jeśli ta instancja `Result` jest wartością `Err`, metoda `expect` powoduje awarię programu i wyświetlenie komunikatu przekazanego jako argument do tej metody. Jeżeli metoda `read_line` zwraca `Err`, prawdopodobnie jest to wynik błędu pochodzącego z bazowego systemu operacyjnego. Gdy dana instancja `Result` jest wartością `Ok`, `expect` pobiera wartość zwracaną, przechowywaną przez `Ok`, i zwraca nam tylko tę wartość, byśmy mogli jej użyć. W tym przypadku tą wartością jest liczba bajtów w danych wprowadzonych przez użytkownika.

Jeśli nie wywołamy `expect`, program skompiluje się, ale pojawi się ostrzeżenie:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
warning: unused `Result` that must be used
  --> src/main.rs:10:5
   |
10 |     io::stdin().read_line(&mut guess);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be handled

warning: `guessing_game` (bin "guessing_game") generated 1 warning
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
```


Kup ksi k Pole ksi k

Rust ostrzega, że nie użyto wartości `Result` zwróconej z `read_line`, wskazując, że program nie obsłużył możliwego błędu.

Właściwym sposobem na pozbycie się tego ostrzeżenia jest tak naprawdę napisanie kodu obsługi błędów, ale w naszym przypadku chcemy, aby ten program po prostu uległ awarii, gdy wystąpi problem, więc możemy użyć `expect`. O odzyskiwaniu sprawności po błędach napiszemy w rozdziale 9.

Wypisywanie wartości za pomocą symboli zastępczych println!

Oprócz zamykającego nawiasu klamrowego w kodzie pozostała do omówienia jeszcze tylko jedna linia:

```
println!("Twój typ to: {guess}");
```

Ta linia wypisuje łańcuch znaków, który zawiera teraz dane wprowadzone przez użytkownika. Zestaw nawiasów klamrowych (`{}`) to symbol zastępczy: potraktuj te nawiasy jak szczypce małego kraba, które utrzymują wartość w miejscu. Gdy wypisujemy wartość zmiennej, nazwę zmiennej można umieścić w nawiasach klamrowych. Podczas wypisywania wyniku wartościowania wyrażenia w łańcuchu znaków formatowania należy umieścić puste nawiasy klamrowe, a za tym łańcuchem wstawić listę oddzielonych przecinkami wyrażeń do wypisania w każdym pustym symbolu zastępczym (nawiasie klamrowym) w tej samej kolejności. Wypisanie zmiennej i wyniku wyrażenia w jednym wywołaniu `println!` może wyglądać tak:

```
let x = 5;
let y = 10;

println!("x = {x} oraz y + 2 = {}", y + 2);
```

Ten kod wypisuje `x = 5 oraz y + 2 = 12`.

Testowanie pierwszej części

Przetestujmy pierwszą część gry w zgadywanie. Uruchom ją za pomocą polecenia `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 6.44s
     Running `target/debug/guessing_game`
Zgadnij liczbę!
Wpisz swój typ.
6
Twój typ to: 6
```


Kup książkę Poleć książkę

W tym momencie pierwsza część gry jest gotowa: pobieramy dane wejściowe z klawiatury, a potem je wypisujemy.

Generowanie sekretnej liczby

Następnie musimy wygenerować sekretną liczbę, którą użytkownik będzie próbował odgadnąć. Ta sekretna liczba powinna być za każdym razem inna, aby można było zagrać więcej niż raz. Użyjemy losowej liczby z przedziału od 1 do 100, by gra nie była zbyt trudna. Rust nie zawiera jeszcze w swojej standardowej bibliotece funkcjonalności generowania liczb losowych. Zespół Rusta udostępnia jednak skrzynkę `rand` ze wspomnianą funkcjonalnością. Dokumentację tej skrzynki znajdziesz na stronie *https://crates.io/crates/rand*.

Korzystanie ze skrzynki w celu uzyskania większej funkcjonalności

Należy pamiętać, że skrzynka jest zbiorem plików z kodem źródłowym Rusta. Projekt, który budujemy, to **skrzynka binarna**, która jest wykonywalna. Skrzynka `rand` jest **skrzynką biblioteki**, która zawiera kod przeznaczony do użycia w innych programach i nie może być wykonywana samodzielnie.

Do koordynacji zewnętrznych skrzynek naprawdę przydatnym narzędziem okazuje się Cargo. Zanim będziemy mogli napisać kod korzystający z `rand`, musimy zmodyfikować plik *Cargo.toml* tak, aby zawierał skrzynkę `rand` jako zależność. Otwórz teraz ten plik i dodaj poniższą linię na dole, pod nagłówkiem sekcji `[dependencies]`, którą utworzyło dla nas Cargo. Pamiętaj, aby określić `rand` dokładnie tak samo, jak pokazaliśmy tutaj, z tym numerem wersji; w przeciwnym razie przykłady kodu z tego tutorialu mogą nie działać:

Cargo.toml

```
[dependencies]
rand = "0.8.5"
```

W pliku *Cargo.toml* wszystko, co następuje po nagłówku, jest częścią danej sekcji, która ciągnie się do momentu rozpoczęcia kolejnej sekcji. W sekcji `[dependencies]` informujemy Cargo, od których zewnętrznych skrzynek zależy nasz projekt i jakich wersji tych skrzynek potrzebujemy. W tym przypadku określamy skrzynkę `rand` z semantycznym specyfikatorem wersji `0.8.5`. Cargo „rozumie" wersjonowanie semantyczne (ang. *Semantic Versioning*; czasami nazywane *SemVer*), które jest standardem zapisywania numerów wersji. Specyfikator `0.8.5` jest tak naprawdę skrótem od `^0.8.5`, co oznacza dowolną wersję z przedziału lewostronnie domkniętego od 0.8.5 do 0.9.

Cargo uznaje, że te wersje mają publiczne interfejsy API zgodne z wersją 0.8.5, a ta specyfikacja zapewnia otrzymanie najnowszej wersji poprawki, która będzie nadal kompilowana z kodem prezentowanym w tym rozdziale. Nie ma gwarancji, że którakolwiek wersja 0.9.0 lub nowsza będzie miała takie samo API jak to użyte w poniższych przykładach.

Nie zmieniając żadnego kodu, skompilujemy teraz projekt, jak pokazaliśmy w listingu 2.2.

Kup ksi k Pole ksi k

Listing 2.2. Dane wyjściowe z uruchomienia polecenia cargo build po dodaniu skrzynki rand jako zależności

```
$ cargo build
    Updating crates.io index
  Downloaded rand v0.8.5
  Downloaded libc v0.2.127
  Downloaded getrandom v0.2.7
  Downloaded cfg-if v1.0.0
  Downloaded ppv-lite86 v0.2.16
  Downloaded rand_chacha v0.3.1
  Downloaded rand_core v0.6.3
   Compiling rand_core v0.6.3
   Compiling libc v0.2.127
   Compiling getrandom v0.2.7
   Compiling cfg-if v1.0.0
   Compiling ppv-lite86 v0.2.16
   Compiling rand_chacha v0.3.1
   Compiling rand v0.8.5
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
```

Możesz zobaczyć różne numery wersji (ale dzięki SemVer! wszystkie będą zgodne z kodem) i różne linie (w zależności od systemu operacyjnego), które mogą zostać wypisane w innej kolejności.

Kiedy dołączamy zewnętrzną zależność, Cargo pobiera z **rejestru** najnowsze wersje wszystkiego, czego dana zależność potrzebuje, czyli kopie danych z Crates.io (*https://crates.io*). Crates.io to repozytorium, w którym członkowie społeczności języka Rust publikują swoje projekty *open source* napisane w tym języku, aby mogli z nich korzystać inni użytkownicy.

Po aktualizacji rejestru Cargo sprawdza sekcję `[dependencies]` i pobiera wszystkie wymienione skrzynki, które nie zostały jeszcze pobrane. W tym przypadku, chociaż jako zależność wymieniliśmy tylko `rand`, Cargo pobrało również inne skrzynki, od których zależy działanie `rand`. Po pobraniu skrzynek Rust kompiluje je, a następnie kompiluje projekt z dostępnymi zależnościami.

Jeśli zaraz po tym uruchomisz ponownie `cargo build` bez wprowadzania jakichkolwiek zmian, nie otrzymasz żadnych danych wyjściowych poza linią `Finished`. Narzędzie Cargo „widzi”, że pobrało już i skompilowało zależności i że nie zmieniliśmy w nich niczego w pliku *Cargo.toml*. Cargo rozpoznaje również, że nie zmieniliśmy niczego w naszym kodzie, więc nie kompiluje go ponownie. Ponieważ nie ma nic do roboty, po prostu kończy działanie.

Jeśli otworzysz plik *src/main.rs*, dokonasz jakiejś trywialnej zmiany, a następnie zapiszesz go i ponownie skompilujesz, zobaczysz tylko dwie linie danych wyjściowych:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53 secs
```

Te linie pokazują, że Cargo aktualizuje kompilację tylko o niewielką zmianę w pliku *src/main.rs*. Twoje zależności nie uległy zmianie, Cargo rozpoznaje więc, że może dla nich ponownie wykorzystać to, co już pobrało i skompilowało.

Kup ksi k Pole ksi k

Zapewnienie powtarzalnych kompilacji za pomocą pliku Cargo.lock

Cargo ma mechanizm, który zapewnia rekompilację tego samego artefaktu za każdym razem, gdy ktokolwiek skompiluje dany kod: dopóki nie wskażesz inaczej, Cargo będzie używać tylko wersji określonych przez Ciebie zależności. Załóżmy na przykład, że w przyszłym tygodniu pojawi się wersja 0.8.6 skrzynki `rand` i będzie ona zawierała ważną poprawkę błędu, ale także regresję, która popsułaby Twój kod. Aby sobie z tym poradzić, przy pierwszym uruchomieniu `cargo build` Rust tworzy plik *Cargo.lock*, więc mamy go teraz w katalogu *guessing_game*.

Kiedy kompilujesz projekt po raz pierwszy, Cargo oblicza wszystkie wersje zależności, które pasują do podanych kryteriów, a następnie zapisuje je w pliku *Cargo.lock*. Gdy za jakiś czas ponownie skompilujesz swój projekt, Cargo „zobaczy", że istnieje plik *Cargo.lock*, i użyje określonych w nim wersji, zamiast ponownie wykonywać całą pracę związaną z ustalaniem wersji zależności. Umożliwia to automatyczne tworzenie powtarzalnych kompilacji. Innymi słowy, dzięki plikowi *Cargo.lock* projekt pozostanie przy wersji 0.8.5, dopóki nie dokonasz bezpośredniego uaktualnienia. Ponieważ plik *Cargo.lock* jest ważny dla powtarzalnych kompilacji, jest często zatwierdzany w systemie kontroli wersji wraz z resztą kodu projektu.

Aktualizacja skrzynki w celu uzyskania nowej wersji

Jeśli *chcesz* zaktualizować jakąś skrzynkę, Cargo udostępnia polecenie `update`, które ignoruje plik *Cargo.lock* i ustala wszystkie najnowsze wersje, które pasują do specyfikacji z pliku *Cargo.toml*. Następnie Cargo zapisuje te wersje w pliku *Cargo.lock*. W przeciwnym razie domyślnie narzędzie Cargo będzie szukało tylko wersji 0.8.5 i nowszych, ale wcześniejszych niż 0.9.0. Jeśli zostaną wydane dwie nowe wersje skrzynki `rand`, 0.8.6 i 0.9.0, po uruchomieniu polecenia `cargo update` zobaczysz poniższe dane wyjściowe:

```
$ cargo update
    Updating crates.io index
    Updating rand v0.8.5 -> v0.8.6
```

Cargo zignoruje wydanie 0.9.0. W tym momencie zauważysz również zmianę w pliku *Cargo.lock*, który będzie wskazywał, że używasz teraz wersji 0.8.6 skrzynki `rand`. Aby użyć `rand` w wersji 0.9.0 lub dowolnej wersji z serii 0.9.*x*, należy zaktualizować plik *Cargo.toml*, aby miał taką postać:

```
[dependencies]
rand = "0.9.0"
```

Następnym razem, gdy uruchomisz `cargo build`, Cargo zaktualizuje rejestr dostępnych skrzynek i dokona ponownej ewaluacji Twoich wymagań dotyczących `rand` zgodnie z nową wersją, którą określiłeś.

Kup ksi k Pole ksi k

Nie wyczerpuje to tematu narzędzia Cargo i jego ekosystemu, który omówimy w rozdziale 14., ale na razie to wszystko, co musisz wiedzieć. Cargo bardzo ułatwia ponowne wykorzystywanie bibliotek, dzięki czemu Rustanie są w stanie pisać mniejsze projekty, które są złożone z wielu pakietów.

Generowanie liczby losowej

Zacznijmy używać skrzynki `rand` do generowania liczby do odgadnięcia. Następnym krokiem jest aktualizacja pliku *src/main.rs*, jak pokazaliśmy w listingu 2.3.

Listing 2.3. Dodawanie kodu w celu generowania liczby losowej

src/main.rs

```
use std::io;
use rand::Rng; ❶

fn main() {
    println!("Zgadnij liczbę!");

    let secret_number = rand::thread_rng().gen_range(1..=100); ❷

    println!("Sekretna liczba to: {secret_number}"); ❸

    println!("Wpisz swój typ.");

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Nie udało się odczytać linii");

    println!("Twój typ to: {guess}");
}
```

Najpierw w punkcie ❶ dodajemy linię `use rand::Rng;`. Cecha `Rng` definiuje metody, które implementują generatory liczb losowych, i byśmy mogli korzystać z tych metod, ta cecha musi być dostępna w zakresie. Cechy omówimy szczegółowo w rozdziale 10.

Następnie pośrodku dodajemy dwie linie. W pierwszej z nowych linii w punkcie ❷ wywołujemy funkcję `rand::thread_rng`, dającą nam konkretny generator liczb losowych, którego zamierzamy użyć: lokalny dla bieżącego wątku wykonywania i dostarczany przez system operacyjny. Następnie w generatorze liczb losowych wywołujemy metodę `gen_range`. Metoda ta jest definiowana przez cechę `Rng`, którą wprowadziliśmy do zakresu za pomocą instrukcji `use rand::Rng;`. Metoda `gen_range` przyjmuje jako argument wyrażenie określające przedział i generuje z niego losową liczbę. Rodzaj używanego tutaj wyrażenia przedziału liczb ma postać *`początek..=koniec`* i jest to przedział obustronnie domknięty, musimy więc określić `1..=100`, aby zażądać liczby z zakresu od 1 do 100.

Kup ksi k Pole ksi k

Uwaga

Abyś wiedział, jakich cech skrzynki użyć oraz jakie jej metody i funkcje wywoływać, każda skrzynka ma dokumentację z instrukcjami użycia. Kolejną ciekawą funkcją Cargo jest to, że uruchomienie polecenia `cargo doc --open` powoduje skompilowanie lokalnie dokumentacji dostarczanej przez wszystkie zależności i otwarcie jej w przeglądarce. Jeśli jesteś zainteresowany innymi funkcjami skrzynki `rand`, uruchom polecenie `cargo doc --open` i kliknij `rand` w panelu po lewej stronie.

Druga nowa linia w punkcie ❸ wypisuje sekretną liczbę. Jest to przydatne podczas opracowywania programu, aby móc go testować, ale usuniemy to z ostatecznej wersji. Gra nie miałaby sensu, gdyby program wypisywał odpowiedź zaraz po uruchomieniu!

Spróbuj uruchomić program kilka razy:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 2.53s
     Running `target/debug/guessing_game`
Zgadnij liczbę!
Sekretna liczba to: 7
Wpisz swój typ.
4
Twój typ to: 4

$ cargo run
   Finished dev [unoptimized + debuginfo] target(s) in 0.02s
    Running `target/debug/guessing_game`
Zgadnij liczbę!
Sekretna liczba to: 83
Wpisz swój typ.
5
Twój typ to: 5
```

Powinieneś otrzymać różne liczby losowe i wszystkie powinny być liczbami z przedziału od 1 do 100. Świetna robota!

Porównywanie próby odgadnięcia z sekretną liczbą

Skoro mamy już dane wprowadzone przez użytkownika i liczbę losową, możemy je porównać. Ten krok pokazaliśmy w listingu 2.4. Pamiętaj, że ten kod jeszcze się nie skompiluje, co wyjaśnimy poniżej.

Listing 2.4. Obsługa możliwych wartości zwracanych dla porównywania dwóch liczb

src/main.rs

```
use rand::Rng;
use std::cmp::Ordering; ❶
```

```
use std::io;

fn main() {
    --fragment pominięty--

    println!("You guessed: {guess}");

    match guess.❷cmp(&secret_number) { ❸
        Ordering::Less => println!("Za mała liczba!"),
        Ordering::Greater => println!("Za duża liczba!"),
        Ordering::Equal => println!("Zgadłeś!"),
    }
}
```

Najpierw w punkcie ❶ dodajemy kolejną instrukcję `use`, wprowadzając do zakresu typ `std::cmp::Ordering` ze standardowej biblioteki. Typ `Ordering` jest kolejnym typem wyliczeniowym i posiada warianty `Less` (mniejsze), `Greater` (większe) i `Equal` (równe). Są to trzy możliwe wyniki porównywania dwóch wartości.

Następnie na końcu dodajemy pięć nowych linii, które korzystają z typu `Ordering`. Metoda `cmp` w punkcie ❸ porównuje dwie wartości i może być wywołana dla wszystkiego, co może być porównywane. Przyjmuje ona referencję do tego, z czym chcemy dokonać porównania: tutaj porównujemy `guess` z `secret_number`. Następnie zwraca ona wariant typu wyliczeniowego `Ordering`, który wprowadziliśmy do zakresu za pomocą instrukcji `use`. W punkcie ❷ używamy wyrażenia `match`, aby zdecydować, co zrobić dalej, na podstawie tego, który wariant `Ordering` został zwrócony z wywołania `cmp` z wartościami `guess` i `secret_number`.

Wyrażenie `match` składa się z **ramion**. Ramię to **wzorzec** do dopasowania i kod, który powinien zostać uruchomiony, jeśli wartość przekazana do `match` będzie pasować do wzorca tego ramienia. Rust przyjmuje wartość podaną dla `match` i przegląda po kolei wzorce poszczególnych ramion. Wzorce i konstrukcja `match` są potężnymi funkcjonalnościami Rusta: pozwalają na wyrażanie różnych sytuacji, które może napotkać kod, i umożliwiają ich obsłużenie. Funkcje te omówimy szczegółowo, odpowiednio, w rozdziałach 6. i 18.

Przeanalizujmy przykład z wyrażeniem `match`, którego używamy tutaj. Przyjmijmy, że użytkownik wytypował liczbę 50, a tym razem losowo wygenerowana sekretna liczba to 38.

Gdy kod porówna 50 z 38, metoda `cmp` zwróci `Ordering::Greater`, ponieważ 50 jest większe niż 38. Wyrażenie `match` pobiera wartość `Ordering::Greater` i rozpoczyna sprawdzanie wzorców wszystkich ramion. Weryfikuje wzorzec pierwszego ramienia, `Ordering::Less`, i „widzi”, że wartość `Ordering::Greater` nie pasuje do `Ordering::Less`, więc ignoruje kod tego ramienia i przechodzi do następnego. Wzorzec następnego ramienia to `Ordering::Greater`, który *pasuje* do `Ordering::Greater`! Kod powiązany z tym ramieniem jest wykonywany i wypisuje na ekranie `Za duża liczba!`. Po pierwszym udanym dopasowaniu wyrażenie `match` kończy porównywanie, więc w tym scenariuszu nie będzie sprawdzać ostatniego ramienia.

Jednak kod pokazany w listingu 2.4 jeszcze się nie skompiluje. Spróbujmy:

Kup ksi k Pole ksi k

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
error[E0308]: mismatched types
  --> src/main.rs:22:21
   |
22 |     match guess.cmp(&secret_number) {
   |                     ^^^^^^^^^^^^^^ expected struct `String`, found integer
   |
   = note: expected reference `&String`
              found reference `&{integer}`
```

Główny komunikat błędu informuje, że istnieją **niedopasowane typy**. Rust ma silny, statyczny system typowania. Ma jednak również funkcjonalność inferencji typów. Kiedy napisaliśmy `let mut guess = String::new()`, Rust był w stanie wywnioskować, że `guess` powinien być typem `String`, i nie kazał nam określać typu. Natomiast `secret_number` jest typem liczbowym. Wartość od 1 do 100 może mieć kilka typów liczbowych Rusta, m.in. `i32` (liczba 32-bitowa), `u32` (liczba 32-bitowa bez znaku) i `i64` (liczba 64-bitowa). Jeśli nie określono inaczej, Rust domyślnie wybiera `i32`, który jest typem `secret_number`, chyba że w innym miejscu dodamy informacje o typie, które spowodują, iż Rust wywnioskuje inny typ liczbowy. Przyczyną błędu jest to, że Rust nie może porównać łańcucha znaków i typu liczbowego.

W związku z tym musimy przekonwertować `String`, który program odczytuje jako dane wejściowe, na typ liczby rzeczywistej, byśmy mogli porównać go z sekretną liczbą. W tym celu do ciała funkcji `main` dodajemy linię pokazaną w poniższym listingu:

src/main.rs

```
    --fragment pominięty--

    let mut guess = String::new();

    io::stdin()
        .read_line(&mut guess)
        .expect("Nie udało się odczytać linii");

    let guess: u32 = guess
        .trim()
        .parse()
        .expect("Wpisz swój typ!");

    println!("Twój typ to: {guess}");

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Za mała liczba!"),
        Ordering::Greater => println!("Za duża liczba!"),
        Ordering::Equal => println!("Zgadłeś!"),
    }
```

Tworzymy zmienną o nazwie `guess`. Ale chwileczkę, czy program nie ma już zmiennej o nazwie `guess`? Tak, ale na szczęście Rust pozwala nam przysłaniać poprzednią wartość `guess` nową wartością. **Przysłanianie** (ang. *shadowing*) pozwala nam ponownie użyć nazwy

Kup książkę Poleć książkę

zmiennej `guess`, dzięki czemu nie musimy tworzyć dwóch unikatowych zmiennych, takich jak na przykład `guess_str` i `guess`. Omówimy to szerzej w rozdziale 3., a na razie wystarczy wiedzieć, że ta funkcjonalność jest często używana, gdy trzeba przekonwertować wartość z jednego typu na drugi.

Tę nową zmienną wiążemy z wyrażeniem `guess.trim().parse()`. `guess` w tym wyrażeniu odwołuje się do pierwotnej zmiennej `guess`, która zawiera dane wejściowe jako łańcuch znaków. Metoda `trim` wywołana na instancji `String` usuwa wszelkie znaki niedrukowalne na początku i końcu; musimy to zrobić, aby móc porównać ten łańcuch znaków z typem `u32`, który może zawierać tylko dane liczbowe. Użytkownik musi nacisnąć *Enter*, aby przekazać argumenty do `read_line` i wprowadzić swój typ liczbowy, co dodaje do łańcucha znaków znak nowej linii. Jeśli użytkownik wpisze na przykład `5` i naciśnie *Enter*, wartość `guess` będzie wyglądała tak: `5\n`. Znaki `\n` reprezentują „nową linię”. (W systemie Windows naciśnięcie klawisza *Enter* powoduje powrót karetki i nową linię: `\r\n`). Metoda `trim` eliminuje `\n` lub `\r\n`, dając w rezultacie tylko `5`.

Metoda `parse` wywoływana na łańcuchu znaków konwertuje go na inny typ. Tutaj stosujemy ją do konwersji z łańcucha znaków na liczbę. Używając `let guess: u32`, musimy poinstruować Rusta, jaki dokładnie typ liczby chcemy uzyskać. Dwukropek (`:`) po `guess` informuje Rusta, że będziemy adnotować typ zmiennej. Rust ma kilka wbudowanych typów liczbowych; widoczny tutaj `u32` jest 32-bitową liczbą całkowitą bez znaku. Jest to dobry domyślny wybór dla małej liczby dodatniej. O innych typach liczbowych napiszemy w rozdziale 3.

Dodatkowo w tym przykładowym programie adnotacja `u32` i porównanie z `secret_number` oznacza, że Rust „wywnioskuje”, iż `secret_number` również powinien być typem `u32`. Teraz porównanie będzie odbywać się między dwiema wartościami tego samego typu!

Metoda `parse` będzie działać tylko na znakach, które można logicznie przekonwertować na liczby, a zatem może łatwo powodować błędy. Gdyby łańcuch zawierał na przykład `A👍%`, nie byłoby sposobu, aby przekonwertować go na liczbę. Ponieważ może się to nie powieść, metoda `parse` zwraca typ `Result`, podobnie jak metoda `read_line` (co omówiliśmy wcześniej w punkcie „Obsługa ewentualnego niepowodzenia za pomocą typu Result”). Potraktujemy typ `Result` w ten sam sposób przez ponowne użycie metody `expect`. Jeżeli metoda `parse` zwróci wariant `Err` typu `Result`, ponieważ nie będzie mogła utworzyć liczby z łańcucha znaków, wywołanie `expect` spowoduje awarię gry i wypisanie komunikatu, który jej podaliśmy. Jeśli metoda `parse` będzie mogła pomyślnie przekonwertować łańcuch znaków na liczbę, zwróci wariant `Ok` typu `Result`, a `expect` zwróci liczbę, którą chcemy uzyskać z wartości `Ok`.

Uruchommy teraz ten program:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running `target/debug/guessing_game`
Zgadnij liczbę!
Sekretna liczba to: 58
Wpisz swój typ.
  76
Twój typ to: 76
Za duża liczba!
```


Kup ksi k Pole ksi k

Nieźle! Mimo dodania spacji przed zgadywaną liczbą program i tak ustalił, że użytkownik wytypował 76. Uruchom program kilka razy, aby zweryfikować różne zachowania przy różnych rodzajach danych wejściowych: wpisz liczbę poprawną, za dużą i za małą.

Większość gry już działa, ale użytkownik może zgadywać tylko raz. Zmieńmy to przez dodanie pętli!

Umożliwianie wielokrotnego zgadywania dzięki zastosowaniu pętli

Słowo kluczowe `loop` tworzy nieskończoną pętlę. Dodamy pętlę, aby dać użytkownikom więcej szans na odgadnięcie liczby.

src/main.rs

```
--fragment pominięty--

println!("Sekretna liczba to: {secret_number}");

loop {
    println!("Wpisz swój typ.");

    --fragment pominięty--

    match guess.cmp(&secret_number) {
        Ordering::Less => println!("Za mała liczba!"),
        Ordering::Greater => println!("Za duża liczba!"),
        Ordering::Equal => println!("Zgadłeś!"),
    }
}
```

Jak widać, wszystko od monitu o wpisanie typowanej liczby przenieśliśmy dalej do pętli. Pamiętaj, aby linie wewnątrz pętli wciąć o kolejne cztery spacje i uruchomić program ponownie. Program będzie teraz w nieskończoność prosił o kolejne odgadnięcie, co w rzeczywistości wprowadza nowy problem. Wygląda na to, że użytkownik nie może zakończyć gry!

Użytkownik zawsze ma wprawdzie opcję przerwania działania programu za pomocą skrótu klawiaturowego *Ctrl+C*. Istnieje jednak inny sposób ucieczki przed tym nienasyconym potworem, o czym wspomnieliśmy wcześniej podczas omawiania parsowania w podrozdziale „Porównywanie próby odgadnięcia z sekretną liczbą”: jeśli użytkownik wprowadzi odpowiedź inną niż liczba, program ulegnie awarii. Możemy to wykorzystać, aby umożliwić użytkownikowi zakończenie gry. Pokazaliśmy to w poniższym listingu:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 1.50s
     Running `target/debug/guessing_game`
Zgadnij liczbę!
Sekretna liczba to: 59
```


Kup ksi k Pole ksi k

```
Wpisz swój typ.
45
Twój typ to: 45
Za mała liczba!
Wpisz swój typ.
60
Twój typ to: 60
Za duża liczba!
Wpisz swój typ.
59
Twój typ to: 59
Zgadłeś!
Wpisz swój typ.
zakończ
thread 'main' panicked at 'Proszę wpisać liczbę!: ParseIntError
{ kind: InvalidDigit }', src/main.rs:28:47
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Wpisanie `zakończ` powoduje wyjście z gry, ale jak zauważysz, to samo dzieje się po wprowadzeniu jakichkolwiek innych danych niebędących liczbami. Jest to co najmniej nieoptymalne; chcemy, aby gra zatrzymała się również po odgadnięciu prawidłowej liczby.

Zakończenie gry po odgadnięciu prawidłowej liczby

Zaprogramujmy grę tak, aby zakończyła się, gdy użytkownik odgadnie poprawną liczbę. W tym celu dodajmy instrukcję `break`:

src/main.rs

```
--fragment pominięty--

match guess.cmp(&secret_number) {
    Ordering::Less => println!("Za mała liczba!"),
    Ordering::Greater => println!("Za duża liczba!"),
    Ordering::Equal => {
        println!("Zgadłeś!");
        break;
    }
}
```

Dodanie linii `break` po `Zgadłeś!` sprawi, że program wyjdzie z pętli, gdy użytkownik poprawnie odgadnie sekretną liczbę. Wyjście z pętli oznacza również wyjście z programu, ponieważ pętla jest ostatnią częścią `main`.

Obsługa nieprawidłowych danych wejściowych

Aby jeszcze bardziej udoskonalić zachowanie gry, zamiast zawieszać program, gdy użytkownik wprowadzi nie-liczbę, sprawmy, by gra ignorowała takie dane wejściowe, a użytkownik mógł kontynuować zgadywanie. W tym celu możemy zmienić linię, w której wartość `guess` jest konwertowana z typu `String` na `u32`, jak pokazaliśmy w listingu 2.5.

Listing 2.5. Ignorowanie wpisywania nie-liczby i umożliwienie użytkownikowi dalszego zgadywania zamiast powodowania awarii programu

src/main.rs

```
--fragment pominięty--

io::stdin()
    .read_line(&mut guess)
    .expect("Nie udało się odczytać linii");
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

println!("Twój typ to: {guess}");

--fragment pominięty--
```

Przełączamy się z wywołania `expect` na wyrażenie `match`, aby przejść od awarii w przypadku błędu do obsługi błędu. Pamiętaj, że `parse` zwraca typ wyliczeniowy `Result`, który ma warianty `Ok` i `Err`. Używamy tutaj wyrażenia `match`, podobnie jak w przypadku wyniku `Ordering` wywołania metody `cmp`.

Jeśli `parse` będzie w stanie pomyślnie przekonwertować łańcuch znaków na liczbę, zwróci wartość `Ok` zawierającą wynikową liczbę. Ta wartość `Ok` będzie pasować do wzorca pierwszego ramienia, a wyrażenie `match` zwróci po prostu wartość `num` wygenerowaną przez `parse` i umieszczoną w wartości `Ok`. Ta liczba znajdzie się dokładnie tam, gdzie chcemy, czyli w nowej zmiennej `guess`, którą tworzymy.

Jeśli `parse` nie będzie w stanie przekształcić łańcucha znaków w liczbę, zwróci wartość `Err` zawierającą więcej informacji o błędzie. Wartość `Err` nie pasuje do wzorca `Ok(num)` w pierwszym ramieniu `match`, ale pasuje do wzorca `Err(_)` drugiego ramienia. Podkreślenie (_) jest wartością uniwersalną; w tym przykładzie instruujemy Rusta, że chcemy dopasować wszystkie wartości `Err`, bez względu na to, jakie informacje zawierają. Program wykona więc kod `continue` drugiego ramienia, który każe mu przejść do następnej iteracji pętli `loop` i poprosić o kolejne typowanie liczby. W ten sposób program zignoruje wszystkie błędy, które może napotkać `parse`!

Teraz wszystko w programie powinno działać zgodnie z oczekiwaniami. Spróbujmy:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///projects/guessing_game)
    Finished dev [unoptimized + debuginfo] target(s) in 4.45s
     Running `target/debug/guessing_game`
Zgadnij liczbę!
Sekretna liczba to: 61
Wpisz swój typ.
10
Twój typ to: 10
Za mała liczba!
Wpisz swój typ.
99
```


Kup ksi k Pole ksi k

```
Twój typ to: 99
Za duża liczba!
Wpisz swój typ.
foo
Wpisz swój typ.
61
Twój typ to: 61
Zgadłeś!
```

Niesamowite! Ostatnią drobną poprawką dokończymy grę w zgadywanie. Przypomnijmy, że program nadal wypisuje sekretną liczbę. Sprawdziło się to w testach, ale rujnuje rzeczywistą grę. Usuńmy instrukcję `println!`, która wypisuje sekretną liczbę. Ostateczny kod pokazaliśmy w listingu 2.6.

Listing 2.6. Kompletny kod gry w zgadywanie

src/main.rs

```
use rand::Rng;
use std::cmp::Ordering;
use std::io;

fn main() {
    println!("Zgadnij liczbę!");

    let secret_number = rand::thread_rng().gen_range(1..=100);

    loop {
        println!("Wpisz swój typ.");

        let mut guess = String::new();

        io::stdin()
            .read_line(&mut guess)
            .expect("Nie udało się odczytać linii");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("Twój typ to: {guess}");

        match guess.cmp(&secret_number) {
            Ordering::Less => println!("Za mała liczba!"),
            Ordering::Greater => println!("Za duża liczba!"),
            Ordering::Equal => {
                println!("Zgadłeś!");
                break;
            }
        }
    }
}
```

Udało Ci się zbudować grę w zgadywanie losowej liczby. Gratulacje!

Kup ksi k Pole ksi k

Podsumowanie

Ten projekt był praktycznym sposobem na zapoznanie Cię z wieloma nowymi koncepcjami języka Rust, takimi jak `let`, `match`, funkcje i wykorzystanie zewnętrznych skrzynek. W następnych kilku rozdziałach omówimy te koncepcje szerzej. W rozdziale 3. przybliżymy koncepcje, które ma większość języków programowania, takie jak zmienne, typy danych i funkcje, oraz pokażemy, jak stosować je w języku Rust. Rozdział 4. będzie poświęcony własności — funkcjonalności, która odróżnia Rusta od innych języków. W rozdziale 5. omówimy struktury i składnię metod, a w rozdziale 6. wyjaśnimy, jak działają typy wyliczeniowe.

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

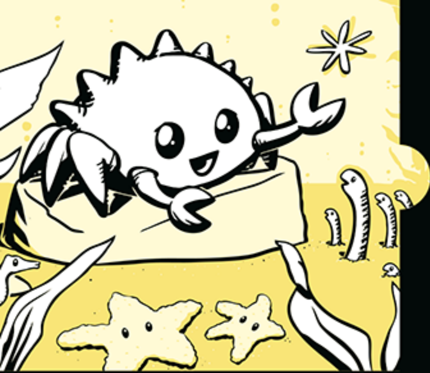
GRUPY HELION

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



RUST:
JĘZYK PRZYSZŁOŚCI
PROGRAMOWANIA!

Rust świetnie się sprawdza na poziomie systemowym, czyli z niskopoziomowymi szczegółami zarządzania pamięcią, reprezentacji danych i współbieżności. Jest zaprojektowany tak, aby naturalnie pisać niezawodny i wydajny kod. Język ten jest również wystarczająco ekspresyjny i ergonomiczny, aby umożliwiać tworzenie aplikacji CLI czy serwerów WWW. Łatwo dostrzec, że praca z Rustem pozwala budować umiejętności, które przydają się w wielu dziedzinach programowania.

Ta książka jest oficjalnym przewodnikiem po języku programowania systemów Rust, udostępnianym na licencji *open source*. Dzięki niej nauczysz się pisać szybsze i bardziej niezawodne oprogramowanie. Dowiesz się również, jak zapewnić sobie kontrolę nad niskopoziomowymi szczegółami wraz z wysokopoziomową ergonomią, co pozwoli Ci na zwiększenie produktywności i uniknięcie trudności związanych z językami niskiego poziomu. Oprócz przystępnie przekazanej wiedzy i niezliczonych przykładów kodu w książce znalazły się trzy rozdziały poświęcone budowaniu kompletnych projektów: gry w zgadywanie liczb, rustowej implementacji narzędzia wiersza poleceń i serwera wielowątkowego.

W książce między innymi:

- tworzenie funkcji, wybieranie typów danych i wiązanie zmiennych
- własność i pożyczanie, czasy życia, typy sparametryzowane
- przekazywanie kompilatorowi ograniczeń programu
- bezstresowe stosowanie współbieżności
- Cargo – wbudowany menedżer pakietów Rusta
- testowanie, obsługa błędów, refaktoryzacja i ekspresyjne dopasowywanie wzorców

Steve Klabnik był kierownikiem zespołu dokumentacji języka Rust i jednym z jego głównych programistów Wcześniej pracował nad takimi projektami jak Ruby i Ruby on Rails.

Carol Nichols jest członkinią zespołu Crates.io i byłą członkinią podstawowego zespołu Rusta. Organizowała konferencję Rust Belt Rust.

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 978-83-289-1010-2

Cena: 129,00 zł