# Comprehensive Rust 🦀

Martin Geisler

# Contents

**Comprehensive Rust'a Hoş Geldiniz 🦀** **12**

**1 Kursun Çalıştırılması** **14**
1.1 Kurs Yapısı . . . 15
1.2 Klavye Kısayolları . . . 18
1.3 Çeviriler . . . 19

**2 Cargo'yu Kullanmak** **20**
2.1 Rust Ekosistemi . . . 20
2.2 Bu Eğitimdeki Kod Örnekleri . . . 21
2.3 Cargo'nun Yerel (Local) Olarak Çalıştırılması . . . 22

**I Gün 1: Sabah** **24**

**3 1. Gün'e Hoş Geldiniz** **25**

**4 Merhaba, Dünya** **27**
4.1 Rust Nedir? . . . 27
4.2 Rust'ın Faydaları . . . 28
4.3 Deneme Alanı (Playground) . . . 28

**5 Türler ve Değerler** **30**
5.1 Merhaba, Dünya . . . 30
5.2 Değişkenler . . . 31
5.3 Değerler . . . 31
5.4 Aritmetik . . . 32
5.5 Tür Çıkarımı (Type Inference) . . . 33
5.6 Alıştırma: Fibonacci . . . 33
5.6.1 Çözüm . . . 34

**6 Kontrol Akışı Temelleri** **35**
6.1 Bloklar ve Kapsamlar . . . 35
6.2 `if` ifadeleri . . . 36
6.3 `match` İfadeleri . . . 36
6.4 Döngüler . . . 38
6.4.1 `for` . . . 38
6.4.2 `loop` . . . 38
6.5 `break` ve `continue` . . . 39

6.5.1 Etiketler . . . 39
6.6 Fonksiyonlar . . . 40
6.7 Makrolar . . . 40
6.8 Alıştırma: Collatz Sekansı . . . 41
6.8.1 Çözüm . . . 42

**II Gün 1: Öğleden Sonra 43**

**7 Tekrar Hoş Geldiniz 44**

**8 Demetler (Tuples) ve Diziler (Arrays) 45**
8.1 Diziler . . . 45
8.2 Demetler . . . 46
8.3 Dizi Adımlama (Iteration) . . . 47
8.4 Desenler ve Çözümlenme (Destructuring) . . . 47
8.5 Alıştırma: İçiçe Diziler . . . 48
8.5.1 Çözüm . . . 48

**9 Referanslar 50**
9.1 Paylaşılan (Shared) Referanslar . . . 50
9.2 Özel (Exclusive) Referanslar . . . 51
9.3 Dilimler (Slices) . . . 52
9.4 Dizeler (Strings) . . . 53
9.5 Referans Geçerliliği (Validity) . . . 54
9.6 Alıştırma: Geometri . . . 54
9.6.1 Çözüm . . . 55

**10 Kullanıcı Tanımlı Türler 57**
10.1 İsimli Yapılar (Named Structs) . . . 57
10.2 Demet Yapıları (Tuple Structs) . . . 58
10.3 Enumlar . . . 59
10.4 Tür Eş İsimleri (Type Aliases) . . . 61
10.5 `const` . . . 62
10.6 `static` . . . 62
10.7 Alıştırma: Asansör Olayları . . . 63
10.7.1 Çözüm . . . 64

**III Gün 2: Sabah 67**

**11 2. Gün'e Hoş Geldiniz 68**

**12 Desen Eşleştirme 69**
12.1 Reddedilemez Desenler (Irrefutable Patterns) . . . 69
12.2 Değerleri Eşleştirmek . . . 70
12.3 Yapılar (Structs) . . . 72
12.4 Enumlar . . . 73
12.5 Let'li Kontrol Akışı . . . 74
12.5.1 `if let` İfadeleri . . . 74
12.5.2 `while let` Deyimleri . . . 74
12.5.3 `let else` Deyimleri . . . 75

12.6 Alıştırma: İfade Değerlendirmesi . . . 76
12.6.1 Çözüm . . . 79

**13 Metotlar ve Özellikler (Traits) 82**
13.1 Metotlar . . . 82
13.2 Özellikler (Traits) . . . 84
13.2.1 Özelliklerin (Traits) Gerçekleştirimi . . . 84
13.2.2 Üstözellikler (Supertraits) . . . 85
13.2.3 İlişkili Türler . . . 86
13.3 Türetme (Deriving) . . . 86
13.4 Alıştırma: Kaydedici Özelliği (Logger Trait) . . . 87
13.4.1 Çözüm . . . 88

**14 Genelleştirmeler (Generics) 90**
14.1 Genelleştirilmiş (Generic) Fonksiyonlar . . . 90
14.2 Özellik (Trait) Sınırları . . . 91
14.3 Genelleştirilmiş (Generic) Veri Türleri . . . 92
14.4 Genelleştirilmiş (Generic) Özellikler (Traits) . . . 93
14.5 `impl Trait` . . . 94
14.6 `dyn Trait` . . . 95
14.7 Alıştırma: Genelleştirilmiş (Generic) `min` . . . 96
14.7.1 Çözüm . . . 97

**IV Gün 2: Öğleden Sonra 98**

**15 Tekrar Hoş Geldiniz 99**

**16 Çevreleyiciler (Closures) 100**
16.1 Çevreleyici Sözdizimi (Closure Syntax) . . . 100
16.2 Yakalama (Capturing) . . . 101
16.3 Çevreleyici özellikleri (Closure traits) . . . 102
16.4 Alıştırma: Kaydedici (Log) Filtresi . . . 103
16.4.1 Çözüm . . . 103

**17 Standart Kütüphane Türleri 105**
17.1 Standart Kütüphane . . . 105
17.2 Dokümantasyon . . . 105
17.3 Option . . . 106
17.4 Result . . . 107
17.5 String . . . 108
17.6 `Vec` . . . 109
17.7 `HashMap` . . . 109
17.8 Alıştırma: Sayıcı . . . 111
17.8.1 Çözüm . . . 112

**18 Standart Kütüphanedeki Özellikler (Traits) 114**
18.1 Karşılaştırmalar . . . 114
18.2 Operatörler . . . 116
18.3 `From` ve `Into` . . . 117
18.4 Tür Dönüştürme (Casting) . . . 117
18.5 `Read` ve `Write` . . . 118

18.6 `Default` Özelliği (Trait) . . . 119
18.7 Alıştırma: ROT13 . . . 119
18.7.1 Çözüm . . . 120

**V Gün 3: Sabah** **122**

**19 Welcome to Day 3** **123**

**20 Bellek Yönetimi** **124**
20.1 Program Belleğinin Gözden Geçirilmesi . . . 124
20.2 Bellek Yönetimine Yaklaşımlar . . . 125
20.3 Sahiplik . . . 126
20.4 Taşıma Semantiği . . . 127
20.5 Clone . . . 129
20.6 Kopyalanan Türler . . . 130
20.7 The `Drop` Trait . . . 131
20.8 Alıştırma: İnşa Edici / Yapıcı Tür (Builder Type) . . . 132
20.8.1 Çözüm . . . 134

**21 Akıllı Göstericiler** **136**
21.1 `Box<T>` . . . 136
21.2 `Rc` . . . 138
21.3 Sahipli Özellik Nesneleri (Owned Trait Objects) . . . 138
21.4 Alıştırma: İkili Ağaç . . . 140
21.4.1 Çözüm . . . 142

**VI Gün 3: Öğleden Sonra** **146**

**22 Tekrar Hoş Geldiniz** **147**

**23 Ödünç Alma** **148**
23.1 Bir Değeri Ödünç Alma . . . 148
23.2 Ödünç Alma Kontrolü . . . 149
23.3 Ödünç Alma Hataları . . . 151
23.4 İç Değişebilirlik (Interior Mutability) . . . 151
23.4.1 `Cell` . . . 151
23.4.2 `RefCell` . . . 152
23.5 Alıştırma: Sağlık İstatistikleri . . . 153
23.5.1 Çözüm . . . 154

**24 Ömürler (Lifetimes)** **156**
24.1 Ömür için Ek Açıklamalar (Annotations) . . . 156
24.2 Fonksiyon Çağrılarında Ömürler . . . 157
24.3 Veri Yapılarında Ömürler . . . 158
24.4 Alıştırma: Protobuf Ayrıştırma . . . 159
24.4.1 Çözüm . . . 164

**VII Gün 4: Sabah** **170**

**25 4. Gün'e Hoş Geldiniz** **171**

**26 Adımlayıcılar (Iterators)** **172**
26.1 Adımlayıcılara (Iterators) Motive Edici Şeyler . . . 172
26.2 `Iterator` Özelliği (Trait) . . . 173
26.3 `Iterator` Yardımcı Metotları . . . 174
26.4 `collect` . . . 175
26.5 `IntoIterator` . . . 175
26.6 Alıştırma: Adımlayıcı Metot Zincirleme . . . 177
26.6.1 Çözüm . . . 178

**27 Modüller** **179**
27.1 Modüller . . . 179
27.2 Dosya Sistemi Hiyerarşisi . . . 180
27.3 Görünürlük . . . 181
27.4 Visibility and Encapsulation . . . 182
27.5 use, super, self . . . 183
27.6 Alıştırma: Bir GUI Kütüphanesi için Modüller . . . 184
27.6.1 Çözüm . . . 186

**28 Test Etme** **190**
28.1 Birim Testleri . . . 190
28.2 Diğer Test Türleri . . . 191
28.3 Derleyici Tüyleri (Lint) ve Clippy Aracı . . . 192
28.4 Alıştırma: Luhn Algrotiması . . . 192
28.4.1 Çözüm . . . 193

**VIII Gün 4: Öğleden Sonra** **196**

**29 Tekrar Hoş Geldiniz** **197**

**30 Hata İşleme** **198**
30.1 Panikler (Panics) . . . 198
30.2 `Result` . . . 199
30.3 Try Operatörü . . . 200
30.4 Try Dönüşümleri . . . 201
30.5 Dinamik Hata Türleri . . . 203
30.6 `thiserror` . . . 203
30.7 `anyhow` . . . 204
30.8 Alıştırma: Result ile Yeniden Yazma . . . 205
30.8.1 Çözüm . . . 207

**31 Emniyetsiz (Unsafe) Rust** **209**
31.1 Emniyetsiz (Unsafe) Rust . . . 209
31.2 Ham Göstericilerinin İçeriği (Dereferencing) . . . 210
31.3 Değişebilir Statik Değişkenler . . . 211
31.4 Birlikler (Unions) . . . 212
31.5 Emniyetsiz (Unsafe) Fonksiyonlar . . . 212
31.5.1 Emniyetsiz (Unsafe) Rust Fonksiyonlar . . . 212

31.5.2 Emniyetsiz (Unsafe) Harici Fonksiyonlar . . . 213
31.5.3 Emniyetsiz (Unsafe) Fonksiyonları Çağırma . . . 214
31.6 Emniyetsiz Özelliklerin (Unsafe Traits) Gerçekleştirilmesi . . . 215
31.7 Emniyetli FFI Sarıcı (Wrapper) . . . 215
31.7.1 Çözüm . . . 218

**IX Android 222**

**32 Welcome to Rust in Android 223**

**33 Kurulum (Setup) 224**

**34 İnşa (Build) Kuralları 225**
34.1 Rust Binaries . . . 226
34.2 Rust Libraries . . . 226

**35 AIDL 228**
35.1 Doğum Günü Servisi Eğitimi . . . 228
35.1.1 AIDL Interfaces . . . 228
35.1.2 Generated Service API . . . 229
35.1.3 Service Implementation . . . 229
35.1.4 AIDL Server . . . 230
35.1.5 Dağıtmak (Deploy) . . . 231
35.1.6 AIDL Client . . . 232
35.1.7 API'yi Değiştirme . . . 233
35.1.8 Updating Client and Service . . . 233
35.2 Working With AIDL Types . . . 234
35.2.1 İlkel (Primitive) Türler . . . 234
35.2.2 Dizi Türleri . . . 234
35.2.3 Nesnelerin Gönderilmesi . . . 235
35.2.4 Parsellenebilir Gönderme . . . 236
35.2.5 Dosyaların Gönderilmesi . . . 237

**36 Android'de Test Etme 239**
36.1 GoogleTest . . . 240
36.2 Taklit Etme (Mocking) . . . 242

**37 Kayıt Tutma (Logging) 244**

**38 Birlikte Çalışabilirlik (Interoperability) 246**
38.1 Interoperability with C . . . 246
38.1.1 Basit bir C Kütüphanesi . . . 247
38.1.2 Using Bindgen . . . 247
38.1.3 İkili Dosyamızı Çalıştırmak . . . 248
38.1.4 Basit bir Rust Kütüphanesi . . . 249
38.1.5 Calling Rust . . . 249
38.2 C++ ile . . . 250
38.2.1 Köprü Modülü . . . 250
38.2.2 Rust Bridge Declarations . . . 251
38.2.3 Oluşturulan (Generated) C++ . . . 252
38.2.4 C++ Bridge Declarations . . . 252

38.2.5 Paylaşılan (Shared) Türler . . . 253
38.2.6 Paylaşılan (Shared) Enum'lar . . . 254
38.2.7 Rust Hata İşleme . . . 255
38.2.8 C++ Hata İşleme . . . 255
38.2.9 Ek Türler . . . 255
38.2.10 Android'de İnşa Etme (Build) . . . 256
38.2.11 Android'de İnşa Etme (Build) . . . 257
38.2.12 Android'de İnşa Etme (Build) . . . 257
38.3 Interoperability with Java . . . 257

**X Chromium 260**

**39 Chromium'daki Rust'a Hoş Geldiniz 261**

**40 Kurulum (Setup) 262**

**41 Chromium ve Cargo Ekosistemlerinin Karşılaştırılması 264**

**42 Chromium Rust policy 266**

**43 İnşa (Build) Kuralları 268**
43.1 Including `unsafe` Rust Code . . . 268
43.2 Chromium C++'dan Rust Koduna Bağımlılık . . . 269
43.3 Visual Studio Code . . . 269
43.4 İnşa (build) kuralları alıştırması . . . 270

**44 Test Etme 272**
44.1 `rust_gtest_interop` Kütüphanesi . . . 273
44.2 Rust Testleri için GN Kuralları . . . 273
44.3 `chromium::import!` Makrosu . . . 273
44.4 Testing exercise . . . 274

**45 C++ ile Birlikte Çalışabilirlik (Interoperability) 275**
45.1 Bağlamalara Örnek . . . 276
45.2 CXX'in Sınırlamaları . . . 276
45.3 CXX Hata İşleme . . . 277
45.3.1 CXX Error Handling: QR Example . . . 277
45.3.2 CXX Error Handling: PNG Example . . . 278
45.4 Using cxx in Chromium . . . 279
45.5 Alıştırma: C++ ile Birlikte Çalışabilirlik . . . 279

**46 Üçüncü Taraf Kasalarnı Eklenmesi 281**
46.1 Configuring the `Cargo.toml` file to add crates . . . 281
46.2 `gnrt_config.toml`'yi Yapılandırma . . . 282
46.3 Kasaların İndirilmesi . . . 282
46.4 gn İnşa Kurallarının Oluşturulması . . . 283
46.5 Problemlerin Çözülmesi . . . 283
46.5.1 Kod Oluşturan İnşa Betikleri (Build Scripts ) . . . 283
46.5.2 C++ Kodunu İnşa Eden (Build) veya Keyfi Eylemler Gerçekleştiren İnşa Betikleri . . . 284
46.6 Bir Kasaya Bağımlılık . . . 284

46.7 Auditing Third Party Crates . . . 284
46.8 Checking Crates into Chromium Source Code . . . 285
46.9 Kasaları Güncel Tutmak . . . 285
46.10 Alıştırma . . . 285

**47 Bringing It Together --- Exercise** **287**

**48 Alıştırma Çözümleri** **289**

**XI Yalın Sistem (Bare Metal): Sabah** **290**

**49 Welcome to Bare Metal Rust** **291**

**50 `no_std`** **293**
50.1 A minimal `no_std` program . . . 294
50.2 `alloc` . . . 294

**51 Mikrodenetleyiciler** **296**
51.1 Ham MMIO . . . 296
51.2 Peripheral Access Crates . . . 298
51.3 HAL crates . . . 299
51.4 Board support crates . . . 300
51.5 The type state pattern . . . 300
51.6 `embedded-hal` . . . 301
51.7 `probe-rs` ve `cargo-embed` . . . 302
51.7.1 Hata Ayıklama . . . 302
51.8 Other projects . . . 303

**52 Alıştırmalar** **304**
52.1 Pusula . . . 304
52.2 Bare Metal Rust Morning Exercise . . . 306

**XII Yalın Sistem (Bare Metal): Öğleden Sonra** **310**

**53 Application processors** **311**
53.1 Rust'a Hazırlanmak . . . 311
53.2 Inline assembly . . . 314
53.3 Volatile memory access for MMIO . . . 315
53.4 Let's write a UART driver . . . 315
53.4.1 More traits . . . 316
53.4.2 Using it . . . 317
53.5 A better UART driver . . . 318
53.5.1 Bitflags Kasası . . . 318
53.5.2 Multiple registers . . . 319
53.5.3 Sürücü . . . 320
53.6 safe-mmio . . . 321
53.6.1 Sürücü . . . 322
53.6.2 Onu Kullan . . . 323
53.7 Kayıt Tutma (Logging) . . . 324
53.7.1 Using it . . . 325

53.8 İstisnalar . . . 326
53.9 aarch64-rt . . . 328
53.10Other projects . . . 329

**54 Kullanışlı kasalar (crates) 330**
54.1 zerocopy . . . 330
54.2 aarch64-paging . . . 331
54.3 buddy_system_allocator . . . 331
54.4 tinyvec . . . 332
54.5 spin . . . 332

**55 Android'de Yalın Sistem (Bare Metal) 334**
55.1 vmbase . . . 335

**56 Alıştırmalar 336**
56.1 RTC driver . . . 336
56.2 Yalın (Bare) Metal Rust Öğleden Sonra . . . 343

**XIII Eşzamanlılık: Sabah 348**

**57 Welcome to Concurrency in Rust 349**

**58 İş Parçacıkları (Threads) 350**
58.1 Temel İş Parçacıkları (Plain Threads) . . . 350
58.2 Kapsamlı İş Parçacıkları (Threads) . . . 351

**59 Kanallar 353**
59.1 Vericiler ve Alıcılar . . . 353
59.2 Sınırsız Kanallar . . . 354
59.3 Sınırlı Kanallar . . . 354

**60 Send ve Sync 356**
60.1 Marker Özellikleri (Traits) . . . 356
60.2 Send . . . 356
60.3 Sync . . . 357
60.4 Örnekler . . . 357

**61 Paylaşımlı Durum (State) 359**
61.1 Arc . . . 359
61.2 Mutex . . . 360
61.3 Örnek . . . 361

**62 Alıştırmalar 363**
62.1 Filozofların Akşam Yemeği . . . 363
62.2 Çok İş Parçacıklı Link Denetleyicisi . . . 364
62.3 Çözümler . . . 367

**XIV Eşzamanlılık: Öğleden Sonra 372**

**63 Hoş Geldiniz 373**

**64 Async Temelleri** **374**
64.1 `async`/`await` . . . 374
64.2 Future Özellikleri (Traits) . . . 375
64.3 Durum Makinesi . . . 376
64.4 Çalışma Zamanları . . . 378
64.4.1 Tokio Kasası . . . 378
64.5 Görevler . . . 379

**65 Kanallar ve Kontrol Akışı** **380**
65.1 Async Kanalları . . . 380
65.2 Join . . . 381
65.3 Select . . . 382

**66 Tuzaklar** **383**
66.1 Blocking the executor . . . 383
66.2 `Pin` . . . 384
66.3 Async Özellikleri (Traits) . . . 386
66.4 İptal (Cancellation) . . . 388

**67 Alıştırmalar** **391**
67.1 Filozofların Akşam Yemeği --- Async . . . 391
67.2 Yayımlamalı Sohbet Uygulaması . . . 392
67.3 Çözümler . . . 395

## XV Idiomatic Rust 400

**68 Welcome to Idiomatic Rust** **401**

**69 Leveraging the Type System** **404**
69.1 Tür Durum Deseni . . . 405
69.1.1 Semantic Confusion . . . 405
69.1.2 Parse, Don't Validate . . . 406
69.1.3 Is It Truly Encapsulated? . . . 407

## XVI Emniyetsiz 409

**70 Welcome to Unsafe Rust** **410**

**71 Setting Up** **411**

**72 Motivasyon** **412**
72.1 Birlikte Çalışabilirlik (Interoperability) . . . 412
72.2 Veri Yapılarında Ömürler . . . 416
72.3 Performance . . . 416

**73 Fonksiyonlar** **417**
73.1 What is “unsafety”? . . . 417
73.2 When is unsafe used? . . . 419
73.3 Data structures are safe ... . . . 419
73.4 ... but actions on them might not be . . . 420
73.5 Less powerful than it seems . . . 420

**XVII Son sözler** **422**

**74 Teşekkürler!** **423**

**75 Sözlük** **424**

**76 Other Rust Resources** **428**

**77 Emekler** **430**

# Comprehensive Rust'a Hoş Geldiniz 


Bu, Google'daki Android ekibi tarafından geliştirilen ücretsiz bir Rust kursudur. Kurs, temel sözdiziminden genelleştirmeler (generics) ve hata işleme gibi ileri düzey konulara kadar Rust'ın tüm yelpazesini kapsar.

> Kursun en son sürümüne https://google.github.io/comprehensive-rust/ adresinden ulaşılabilir. Başka bir yerde okuyorsanız, lütfen güncellemeleri kontrol edin.
>
> Kurs diğer dillerde de mevcuttur. Sayfanın sağ üst köşesinden tercih ettiğiniz dili seçin veya tüm mevcut çevirilerin listesi için Çeviriler sayfasını kontrol edin.
>
> Kurs ayrıca PDF olarak da mevcuttur.

Kursun amacı size Rust'ı öğretmektir. Rust hakkında hiçbir şey bilmediğinizi varsayıyoruz ve şunu umuyoruz:

- Rust sözdizimi ve dili hakkında kapsamlı bir anlayış sunmayı.
- Rust'ta mevcut programları değiştirmenize ve yeni programlar yazmanıza olanak sağlamayı.
- Yaygın Rust kod kalıplarını göstermeyi.

İlk dört kurs gününe Rust Esasları (Rust Fundamentals) adını veriyoruz.

Buna dayanarak, bir veya daha fazla uzmanlık konusuna dalmaya davetlisiniz:

- Android: Android platformu geliştirme (AOSP) için Rust'ın kullanımına ilişkin yarım günlük bir kurs. Buna C, C++ ve Java ile birlikte çalışabilirlik (interoperability) de dahildir.
- Chromium: Rust'ın Chromium tabanlı tarayıcılarda kullanılmasına ilişkin yarım günlük bir kurs. Bu, C++ ile birlikte çalışabilirliği ve üçüncü taraf kasaların Chromium'a nasıl dahil edileceğini içerir.
- Yalın-metal: Yalın sistem (gömülü) geliştirme için Rust'ın kullanımına ilişkin tam günlük bir ders. Hem mikrodenetleyiciler hem de uygulama işlemcileri kapsanmaktadır.
- Eşzamanlılık: Rust'ta eşzamanlılık üzerine tam günlük bir ders. Hem klasik eşzamanlılığı (iş parçacıkları ve muteksleri kullanarak kesintili/geçişli zamanlama) hem de async/await eşzamanlılığını (future özelliklerini kullanarak işbirlikçi çoklu görev) ele alıyoruz.

## Hedef Dışı

Rust geniş bir dil ve birkaç gün içinde tamamını ele almamız mümkün olmayacak. Aşağıdakiler bu kursun amaçları dışında kalır:

- Makroların nasıl geliştirileceğini öğrenmek: lütfen bunun yerine Rust Kitabı'na ve Örneklerle Rust'a bakınız.

## Varsayımlar

Kurs, nasıl programlanacağını zaten bildiğinizi varsayar. Rust, statik tür sistemine sahip (statically-typed) bir dildir ve Rust yaklaşımını daha iyi açıklamak veya karşılaştırmak için bazen C ve C++ ile karşılaştırmalar yapacağız.

Python veya JavaScript gibi dinamik olarak yazılmış bir dilde nasıl programlanacağını biliyorsanız, o zaman da gayet iyi takip edebileceksiniz.

Bu bir *konuşmacı notu* örneğidir. Bunları slaytlara ek bilgi eklemek için kullanacağız. Bunlar, eğitmenin ele alması gereken önemli noktaların yanı sıra sınıfta ortaya çıkan tipik soruların yanıtları da olabilir.

# Chapter 1

# Kursun Çalıştırılması

Bu sayfa kurs eğitmenine yöneliktir.

Kursu Google'da dahili olarak nasıl kullandığımıza dair biraz arka plan bilgisini burada bulabilirsiniz.

Dersleri genellikle sabah 9:00'dan akşam 16:00'ya kadar yürütüyoruz ve ortasında 1 saatlik öğle yemeği arası veriyoruz. Geriye sabah dersi için 3saat, öğleden sonra dersi için ise 3 saat kalıyor. Her iki oturum da öğrencilerin alıştırmalar üzerinde çalışmaları için birden fazla ara ve zaman içerir.

Kursu çalıştırmadan önce şunları yapmak isteyeceksiniz:

1. Ders materyaline aşina olun. Önemli noktaların vurgulanmasına yardımcı olmak için konuşmacı notları ekledik (lütfen daha fazla konuşmacı notuna katkıda bulunarak bize yardımcı olun!). Sunum yaparken, konuşmacı notlarını açılır pencerede açtığınızdan emin olmalısınız (”Konuşmacı Notları”nın yanındaki küçük oklu bağlantıya tıklayın). Bu şekilde sınıfa sunacağınız temiz bir ekrana sahip olursunuz.
2. Tarihlere karar verin. Kurs en az dört tam gün sürdüğünden, günleri iki haftaya yaymanızı öneririz. Kurs katılımcıları, kursta boşluk kalmasının, onlara verdiğimiz tüm bilgileri işlemelerine yardımcı olması açısından yararlı bulduklarını belirtmişlerdir.
3. Katılımcılarınız için yeterince büyük bir oda bulun. Sınıf mevcudunun 15-25 kişilik olmasını öneriyoruz. Bu, insanların rahatça soru sorabileceği kadar küçüktür --- aynı zamanda bir eğitmenin soruları yanıtlamak için zamanının olacağı kadar da küçüktür. Odada kendiniz ve öğrencileriniz için *masaların* olduğundan emin olun: hepinizin oturup dizüstü bilgisayarlarınızla çalışabilmesi gerekir. Özellikle eğitmen olarak çok sayıda canlı kodlama yapacaksınız, bu nedenle kürsü size pek yardımcı olmayacaktır.
4. Kurs gününüzde, işleri ayarlamak için odaya biraz erken gelin. Doğrudan dizüstü bilgisayarınızda çalışan `mdbook serve` komutunu kullanarak sunum yapmanızı öneririz (kurulum talimatlarına bakın). Bu, sayfaları değiştirirken gecikme olmadan en iyi performansı sağlar. Dizüstü bilgisayarınızı kullanmak, siz veya kurs katılımcılarının tespit ettiği yazım hatalarını düzeltmenize de olanak tanır.
5. İnsanların alıştırmaları kendi başlarına veya küçük gruplar halinde çözmelerine izin verin. Genellikle sabah ve öğleden sonra alıştırmalara 30-45 dakika harcıyoruz (çözümleri gözden geçirme (review) zamanı da dahil). İnsanlara takılıp kalmadıklarını

veya yardımcı olabileceğiniz bir konu olup olmadığını sorduğunuzdan emin olun. Birkaç kişinin aynı sorunu yaşadığını gördüğünüzde bunu sınıfa söyleyin ve bir çözüm önerin; örneğin insanlara standart kütüphanede ilgili bilgiyi nerede bulabileceklerini gösterin.

Hepsi bu kadar, kursta iyi şanslar! Umarız bizim için olduğu kadar sizin için de eğlenceli olur!

Kursu geliştirmeye devam edebilmemiz için lütfen sonrasında geri bildirimde bulunun. Sizin için neyin işe yaradığını ve neyin daha iyi hale getirilebileceğini duymak isteriz. Öğrencileriniz de bize geri bildirim gönderebilir!

## 1.1 Kurs Yapısı

Bu sayfa kurs eğitmenine yöneliktir.

### Rust Esasları

İlk dört gün Rust Temelleri'ni oluşturur. Günler hızlı geçiyor ve çok fazla yol kat ediyoruz!

Kurs programı:

- 1. Gün Sabah (2 saat 10 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 5 dakika |
| Merhaba, Dünya | 15 dakika |
| Türler ve Değerler | 40 dakika |
| Kontrol Akışı Temelleri | 45 dakika |

- 1. Gün Öğleden Sonra (2 saat 45 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Demetler (Tuples) ve Diziler (Arrays) | 35 dakika |
| Referanslar | 55 dakika |
| Kullanıcı Tanımlı Türler | 1 saat |

- 2. Gün Sabah (2 saat 45 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 3 dakika |
| Desen Eşleştirme | 50 dakika |
| Metotlar ve Özellikler (Traits) | 45 dakika |
| Genelleştirmeler (Generics) | 45 dakika |

- 2. Gün Öğleden Sonra (2 saat 50 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Çevreleyiciler (Closures) | 30 dakika |
| Standart Kütüphane Türleri | 1 saat |
| Standart Kütüphanedeki Özellikler (Traits) | 1 saat |

- 3. Gün Sabah (2 saat 20 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 3 dakika |
| Bellek Yönetimi | 1 saat |
| Akıllı Göstericiler | 55 dakika |

- 3. Gün Öğleden Sonra (1 saat 55 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Ödünç Alma | 55 dakika |
| Ömürler (Lifetimes) | 50 dakika |

- 4. Gün Sabah (2 saat 50 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 3 dakika |
| Adımlayıcılar (Iterators) | 55 dakika |
| Modüller | 45 dakika |
| Test Etme | 45 dakika |

- 4. Gün Öğleden Sonra (2 saat 20 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Hata İşleme | 55 dakika |
| Emniyetsiz (Unsafe) Rust | 1 saat 15 dakika |

## Derin Dalışlar

Rust'ın Esasları hakkındaki 4 günlük derse ek olarak, daha özel konuları da ele alıyoruz:

### Android'de Rust

Rust in Android ayrıntılı incelemesi, Android platformu geliştirme için Rust'ı kullanma konusunda yarım günlük bir kurstur. Buna C, C++ ve Java ile birlikte çalışabilirlik (interoperability) de dahildir.

Bir AOSP kasasına ihtiyacınız olacak. Aynı makinede kurs deposunu kontrol edin ve `src/android/` dizinini AOSP kasasının kök dizinine taşıyın. Bu, Android inşa (build) sisteminin `src/android/` içindeki `Android.bp` dosyalarını görmesini sağlayacaktır.

`adb sync` emülatörünüzle veya gerçek cihazınızla çalıştığından emin olun ve tüm Android örneklerini `src/android/build_all.sh` kullanarak önceden inşa (build) edin. Çalıştırdığı komutları görmek için betiği okuyun ve bunları elle çalıştırdığınızda çalıştıklarından emin olun.

### Chromium'da Rust

Chromium'da Rust'ın ayrıntılı incelemesi, Rust'ın Chromium tarayıcısının bir parçası olarak kullanılmasına ilişkin yarım günlük bir kurstur. Bu, Chromium'un gn yapı sisteminde Rust'ın kullanılmasını, üçüncü taraf kitaplıkların (”kasalar/crates”) getirilmesini ve C++ ile birlikte çalışabilirliğini içerir.

You will need to be able to build Chromium --- a debug, component build is recommended for speed but any build will work. Ensure that you can run the Chromium browser that you've built.

### Yalın-Sistem (Bare-metal) Rust

Yalın-Sistem Rust ayrıntılı incelemesi, yalın sistem (gömülü) geliştirme için Rust kullanımına ilişkin tam günlük bir derstir. Hem mikrodenetleyiciler hem de uygulama işlemcileri kapsanmaktadır.

Mikrodenetleyici kısmı için BBC micro:bit v2 geliştirme kartını önceden satın almanız gerekecek. Herkesin karşılama sayfasında açıklandığı gibi bir dizi paketi yüklemesi gerekecektir.

### Rust'ta Eşzamanlılık

Rust'ta Eşzamanlılık ayrıntılı incelemesi, klasik ve aynı zamanda `async`/`await` eşzamanlılığı üzerine tam günlük bir derstir.

Yeni bir kasa (crate) kurulumuna ve bağımlılıkların indirilip kullanıma hazır hale getirilmesine ihtiyacınız olacak. Daha sonra örnekleri denemek için kopyalayıp `src/main.rs` dosyasına yapıştırabilirsiniz:

```
cargo init concurrency
cd concurrency
cargo add tokio --features full
cargo run
```

Kurs programı:

- Sabah (3 saat 20 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| İş Parçacıkları (Threads) | 30 dakika |
| Kanallar | 20 dakika |
| Send ve Sync | 15 dakika |
| Paylaşımlı Durum (State) | 30 dakika |
| Alıştırmalar | 1 saat 10 dakika |

- Öğleden Sonra (3 saat 30 dakika, aralar dahil)

| Bölüm | Süre |
|---|---|
| Async Temelleri | 40 dakika |
| Kanallar ve Kontrol Akışı | 20 dakika |
| Tuzaklar | 55 dakika |
| Alıştırmalar | 1 saat 10 dakika |

#### Idiomatic Rust

The Idiomatic Rust deep dive is a 2-day class on Rust idioms and patterns.

You should be familiar with the material in Rust Fundamentals before starting this course.

Kurs programı:

- Morning (25 minutes, including breaks)

| Bölüm | Süre |
|---|---|
| Leveraging the Type System | 25 minutes |

#### Unsafe (Work in Progress)

The Unsafe deep dive is a two-day class on the *unsafe* Rust language. It covers the fundamentals of Rust's safety guarantees, the motivation for `unsafe`, review process for `unsafe` code, FFI basics, and building data structures that the borrow checker would normally reject.

Kurs programı:

- Day 1 Morning (1 hour, including breaks)

| Bölüm | Süre |
|---|---|
| Kurulum (Setup) | 2 dakika |
| Motivasyon | 20 dakika |
| Fonksiyonlar | 25 minutes |

### Format

Kursun çok etkileşimli olması amaçlanıyor ve soruların Rust'ın keşfini yönlendirmesine izin vermenizi öneririz!

## 1.2 Klavye Kısayolları

Bu mdBook'ta birkaç kullanışlı klavye kısayolu vardır:

- Arrow-Left: Önceki sayfaya gidin.
- Arrow-Right: Sonraki sayfaya gidin.
- Ctrl + Enter: Odaklı kod örneğini yürütün.
- s: Arama çubuğunu etkinleştirin.

## 1.3 Çeviriler

Kurs bir grup harika gönüllüler tarafından diğer dillere çevrildi:

- Brezilya Portekizcesi yazanlar: @rastringer, @hugojacob, @joaovicmendes ve @henrif75.
- Çince (Basitleştirilmiş) yazanlar: @suetfei, @wnghl, @anlunx, @kongy, @noahdragon, @superwhd, @SketchK ve @nodmp.
- Çince (Geleneksel) yazanlar: @hueich, @victorhsieh, @mingyc, @kuanhungchen ve @johnathan79717.
- Farsça yazanlar: @DannyRavi, @javad-jafari, @Alix1383, @moaminsharifi, @hamidrezakp ve @mehrad77.
- Japonca yazanlar: @CoinEZ-JPN, @momotaro1105, @HidenoriKobayashi ve @kantasv.
- Korece yazanlar: @keispace, @jiyongp, @jooyunghan ve @namhyung.
- Bengalce yazanlar: @raselmandol.
- Ukraynaca yazanlar: @git-user-cpp, @yaremam ve @reta.

Diller arasında geçiş yapmak için sağ üst köşedeki dil seçiciyi kullanın.

### Tamamlanmamış Çeviriler

Devam eden çok sayıda çeviri var. En son güncellenen çevirilere bağlantı veriyoruz:

- Arapça yazanlar: @younies
- Bengalce yazanlar: @raselmandol.
- Fransızca yazanlar: @KookaS, @vcaen ve @AdrienBaudemont.
- Almanca yazanlar: @Throvn ve @ronaldfw.
- İtalyanca yazanlar: @henrythebuilder ve @detro.

Çevirilerin tam listesi ve güncel durumları son güncelleme tarihi itibarıyla veya kursun en son sürümüyle senkronize edilmiş olarak da mevcuttur.

Bu çabaya yardımcı olmak istiyorsanız nasıl başlayacağınızı öğrenmek için lütfen talimatlarımıza bakın. Çeviriler sorun izleyicide üzerinde koordine edilir.

# Chapter 2

# Cargo'yu Kullanmak

Rust hakkında okumaya başladığınızda, Rust ekosisteminde Rust uygulamalarını inşa etmek (build) ve çalıştırmak için kullanılan standart araç Cargo ile yakında tanışacaksınız. Burada Cargo'nun ne olduğuna, daha geniş ekosisteme nasıl uyum sağladığına ve bu eğitime nasıl uyduğuna dair kısa bir genel bakış sunmak istiyoruz.

## Kurulum

> **Lütfen https://rustup.rs/ adresindeki talimatları izleyin.**

Bu size Cargo oluşturma aracını (`cargo`) ve Rust derleyicisini (`rustc`) sağlayacaktır. Ayrıca, araç zincirlerini (toolchain) kurmak/değiştirmek, çapraz derlemeyi ayarlamak vb. için kullanabileceğiniz bir komut satırı yardımcı programı olan `rustup`'a sahip olacaksınız.

Rust'u kurduktan sonra editörünüzü/düzenleyicinizi veya IDE'nizi Rust ile çalışacak şekilde yapılandırmalısınız. Çoğu editör bunu, VS Code, Emacs, Vim/Neovim ve diğerleri için otomatik tamamlama ve tanıma atlama işlevselliği sağlayan rust-analyzer ile haberleşerek yapar. Ayrıca RustRover adında farklı bir IDE de mevcuttur.

- On Debian/Ubuntu, you can install `rustup` via `apt`:

  ```
  sudo apt install rustup
  ```

- Mac işletim sisteminde Rust'ı yüklemek için Homebrew'i kullanabilirsiniz, ancak bu eski bir sürüm sağlayabilir. Bu nedenle Rust'ı resmi sitesinden kurmanız önerilir.

## 2.1 Rust Ekosistemi

Rust ekosistemi bir dizi araçtan oluşur; bunların başlıcaları şunlardır:

- `rustc`: `.rs` dosyalarını ikili (binary) dosyalara ve diğer ara biçimlere (format) dönüştüren Rust derleyicisi.
- `cargo`: Rust bağımlılık yöneticisi ve inşa (build) aracı. Cargo, genellikle https://crates.io üzerinde barındırılan bağımlılıkları nasıl indireceğini bilir ve projenizi oluştururken bunları `rustc` derleyicisine aktarır. Cargo ayrıca birim testlerini yürütmek için kullanılan yerleşik bir test çalıştırıcısıyla birlikte gelir.

- `rustup`: Rust araç zinciri (toolchain) yükleyicisi ve güncelleyicisi. Bu araç, Rust'ın yeni sürümleri yayınlandığında `rustc` ve `cargo` programlarını yüklemek ve güncellemek için kullanılır. Ek olarak, `rustup` standart kitaplık için belgeleri de indirebilir. Aynı anda birden fazla Rust sürümünü yükleyebilirsiniz ve `rustup` gerektiğinde bunlar arasında geçiş yapmanıza olanak tanır.

Anahtar noktalar:

- Rust'un her altı haftada bir yeni sürümünün çıktığı hızlı bir sürüm programı var. Yeni sürümler, eski sürümlerle geriye dönük uyumluluğu korur ve ayrıca yeni işlevsellik sağlar.
- Üç sürüm kanalı vardır: ”kararlı (stable)”, ”beta” ve ”gecelik (nightly)”.
- Yeni özellikler ”gecelik (nightly)” kanalında test ediliyor, ”beta” her altı haftada bir ”kararlı (stable)” hale geliyor.
- Bağımlılıklar ayrıca alternatif olarak kayıtlardan, git'ten, klasörlerden ve daha fazlasından çözülebilir.
- Rust'ın ayrıca yayınları (editions) vardır: mevcut yayın (edition) Rust 2024'dür. Önceki yayınlar Rust 2015, Rust 2018 ve Rust 2021'di.
  - Yayınların dilde geriye dönük olarak uyumsuz değişiklikler yapmasına izin verilir.
  - Kodun kırılmasını (breaking code) önlemek için yayınlar isteğe bağlıdır: `Cargo.toml` dosyası aracılığıyla kasanızın (crate) sürümünü seçersiniz.
  - Ekosistemin bölünmesini önlemek için Rust derleyicileri farklı sürümler için yazılan kodları karıştırabilir.
  - Derleyiciyi `cargo` yoluyla değil de doğrudan kullanmanın oldukça nadir olduğunu unutmayın (çoğu kullanıcı asla kullanmaz).
  - Cargo'nun kendisinin son derece güçlü ve kapsamlı bir araç olduğunu belirtmekte fayda var. Aşağıdakiler dahil olmakla ve bunlarla sınırlı olmamak üzere birçok gelişmiş özelliğe sahiptir:
    * Proje/paket yapısı
    * çalışma alanları
    * Geliştirici Bağımlılıkları ve Çalışma Zamanı Bağımlılığı yönetimi/önbelleğe alma
    * build scripting
    * genel kurulum
    * Ayrıca alt komut eklentileriyle de genişletilebilir (cargo Clippy gibi).
  - Resmi Cargo Kitabı'ndan daha fazlasını okuyun

## 2.2 Bu Eğitimdeki Kod Örnekleri

Bu eğitimde çoğunlukla Rust dilini tarayıcınız üzerinden yürütebileceğiniz örnekler üzerinden inceleyeceğiz. Bu, kurulumu çok daha kolay hale getirir ve herkes için tutarlı bir deneyim sağlar.

Cargo'nun kurulumu hala teşvik edilmektedir: alıştırmaları yapmanızı kolaylaştıracaktır. Son gün, bağımlılıklarla nasıl çalışacağınızı ve bunun için Cargo'ya ihtiyacınız olduğunu gösteren daha büyük bir alıştırma yapacağız.

Bu kurstaki kod blokları tamamen etkileşimlidir:

```
fn main() {
    println!("Düzenle beni!");
}
```

Metin kutusu odaklanmışken kodu çalıştırmak için Ctrl + Enter tuşlarını kullanabilirsiniz.

Çoğu kod örneği yukarıda gösterildiği gibi düzenlenebilir. Birkaç kod örneği çeşitli nedenlerden dolayı düzenlenemez:

- Gömülü deneme alanları (playgrounds) birim testlerini yürütemez. Birim testlerini göstermek için kodu kopyalayıp yapıştırın ve gerçek Deneme Alanında (Playground) açın.
- Gömülü deneme alanları, sayfadan uzaklaştığınız anda durumlarını kaybeder! Öğrencilerin alıştırmaları yerel bir Rust kurulumu kullanarak veya Deneme Alanı aracılığıyla çözmelerinin nedeni budur.

## 2.3 Cargo'nun Yerel (Local) Olarak Çalıştırılması

Kodu kendi sisteminizde denemek istiyorsanız, önce Rust'u yüklemeniz gerekecektir. Bunu Rust Book'taki talimatları uygulayarak yapın. Bu size çalışan bir `rustc` ve `cargo`vermelidir. Bu yazının yazıldığı sırada en son kararlı Rust sürümü şu sürüm numaralarına sahiptir:

```
% rustc --version
rustc 1.69.0 (84c898d65 2023-04-16)
% cargo --version
cargo 1.69.0 (6e9a83356 2023-04-12)
```

Rust geriye dönük uyumluluğu koruduğu için daha sonraki herhangi bir sürümü de kullanabilirsiniz.

Bunu yaptıktan sonra, bu eğitimdeki örneklerden birinden bir Rust ikili (binary) programı oluşturmak için şu adımları izleyin:

1. Kopyalamak istediğiniz örneğin üzerindeki ”Panoya kopyala” düğmesine tıklayın.
2. Kodunuz için yeni bir `exercise/` dizini oluşturmak için `cargo new exercise` komutunu kullanın:

   ```
   $ cargo new exercise
        Created binary (application) `exercise` package
   ```

3. İkili (binary) dosyanızı inşa etmek (build) ve çalıştırmak için `exercise/` dizinine gidin ve `cargo run` komutunu kullanın:

   ```
   $ cd exercise
   $ cargo run
      Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
       Finished dev [unoptimized + debuginfo] target(s) in 0.75s
        Running `target/debug/exercise`
   Hello, world!
   ```

4. `src/main.rs` dosyasındaki hazır kodu kendi kodunuzla değiştirin. Örneğin, önceki sayfadaki örneği kullanarak `src/main.rs` dosyasının şu şekilde görünmesini sağlayın

```rust
fn main() {
    println!("Düzenle beni!");
}
```

5. Güncellenmiş ikili (binary) dosyanızı inşa etmek (build) ve çalıştırmak (run) için `cargo run` komutunu kullanın:

```
$ cargo run
   Compiling exercise v0.1.0 (/home/mgeisler/tmp/exercise)
    Finished dev [unoptimized + debuginfo] target(s) in 0.24s
     Running `target/debug/exercise`
Edit me!
```

6. Projenizi hatalara karşı hızlı bir şekilde kontrol etmek için `cargo check` komutunu kullanın, çalıştırmadan derlemek için `cargo build` komutunu kullanın. Normal bir hata ayıklama inşasında (build) çıktıyı `target/debug/` içinde bulacaksınız. `target/release/` içinde optimize edilmiş bir sürüm inşası oluşturmak için `cargo build --release` komutunu kullanın.

7. `Cargo.toml` dosyasını düzenleyerek projeniz için bağımlılıklar ekleyebilirsiniz. `cargo` komutlarını çalıştırdığınızda, eksik bağımlılıkları sizin için otomatik olarak indirip derleyecektir.

Sınıf katılımcılarını Cargo'yu yüklemeye ve yerel bir metin düzenleyici kullanmaya teşvik etmeye çalışın. Normal bir gelişim ortamına sahip olacakları için hayatları kolaylaşacaktır.

# Part I

# Gün 1: Sabah

# Chapter 3

# 1. Gün'e Hoş Geldiniz

Rust Esasları'nın ilk günü. Bugün çok fazla konuyu ele alacağız:

- Temel Rust sözdizimi: değişkenler, skaler ve bileşik türleri, numaralandırma türleri (enums), yapılar (structs), referanslar, fonksiyonlar ve metotlar.
- Türler ve tür çıkarımı (inference).
- Kontrol akışı yapıları (constructs): döngüler, koşullar vb.
- Kullanıcı tanımlı türler: yapılar (structs) ve numaralandırmalar (enums)

## Zamanlama (Schedule)

Bu oturum 10 dakikalık aralar dahil yaklaşık 2 saat 10 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 5 dakika |
| Merhaba, Dünya | 15 dakika |
| Türler ve Değerler | 40 dakika |
| Kontrol Akışı Temelleri | 45 dakika |

This slide should take about 5 minutes.

Lütfen öğrencilere şunu hatırlatın:

- Sorular akıllarına geldiğinde sormalılar, sona saklamamalılar.
- Sınıfın etkileşimli olması amaçlanıyor ve tartışmalar çok teşvik ediliyor!
  - Bir eğitmen olarak tartışmaları güncel tutmaya çalışmalısınız, yani Rust'ın işi nasıl yaptığı ve başka bir dilin işi nasıl yaptığıyla ilgili tartışmaları sürdürün. Doğru dengeyi bulmak zor olabilir, ancak insanları tek yönlü iletişimden çok daha fazla meşgul ettiği için tartışmalara izin vermeyi tercih edin.
- Sorular çoğunlukla slaytlardan önce bazı şeyler hakkında konuşacağımız anlamına gelecektir.
  - Bu kesinlikle sorun değil! Tekrar etmek, öğrenmenin önemli bir parçasıdır. Slaytların sadece bir destek olduğunu ve istediğiniz gibi atlamakta özgür olduğunuzu unutmayın.

İlk günün amacı, Rust'ta diğer dillerle hemen hemen paralel olması gereken ”temel” şeyleri göstermektir. Rust'ın daha gelişmiş kısımları ilerleyen günlerde gelecek.

Eğer bunu bir sınıfta öğretiyorsanız, burası programın üzerinden geçmek için iyi bir yerdir. Her bölümün sonunda bir alıştırma ve ardından bir ara olduğunu unutmayın. Aradan sonra alıştırma çözümünü tamamlamayı planlayın. Burada listelenen zamanlar kursun programa uygun devam etmesi için bir öneridir. Esnek olmaktan ve gerektiği gibi ayarlamaktan çekinmeyin!

# Chapter 4

# Merhaba, Dünya

Bu bölüm yaklaşık 15 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Rust Nedir? | 10 dakika |
| Rust'ın Faydaları | 3 dakika |
| Deneme Alanı (Playground) | 2 dakika |

## 4.1 Rust Nedir?

Rust, 2015'te 1.0 sürümü çıkan yeni bir programlama dilidir:

- Rust, C++ ile benzer role sahip statik olarak derlenen bir dildir
  - `rustc` arka tarafta (backend) LLVM'yi kullanır.
- Rust birçok platformu ve mimariyi destekler:
  - x86, ARM, WebAssembly, ...
  - Linux, Mac, Windows, ...
- Rust çok çeşitli cihazlarda kullanılır:
  - donanım yazılımlarında (firmwares) ve önyükleyicilerde (boot loaders),
  - akıllı ekranlarda,
  - cep telefonlarında,
  - masaüstü bilgisayarlarda,
  - sunucularda.

This slide should take about 10 minutes.

Rust, C++ ile aynı alanda yer almaktadır:

- Yüksek esneklik.
- Yüksek düzeyde kontrol.
- Mikrodenetleyiciler gibi çok kısıtlı cihazlara ölçeklendirilebilir.
- Çalışma zamanı veya çöp toplama özelliği yoktur.
- Performanstan ödün vermeden güvenilirlik ve emniyete (safety) odaklanır.

## 4.2 Rust'ın Faydaları

Rust'ın bazı benzersiz yok satan noktaları:

- *Derleme zamanı bellek emniyeti* - derleme zamanında tüm bellek hataları sınıfları önlenir
  - İlklendirilmemiş değişken yok.
  - Adresi iki kez serbest bırakma (double free) yok.
  - Serbest bıraktıktan sonra kullanma (use after free) yok.
  - `NULL` göstericileri yok.
  - Kilitli halde unutulmuş hiçbir mutex yok.
  - İş parçacıkları (threads) arasında veri yarışları (data races) yok.
  - Adımlayıcıyının (iterator) geçersiz kılınması (invalidation) yok.
- *Tanımsız çalışma zamanı davranışı yok* - Bir Rust deyiminin yaptığı şey hiçbir zaman belirtilmemiş (unspecified) olarak bırakılmaz
  - Dizi erişiminde sınırlar kontrol edilir.
  - Tamsayı taşması tanımlıdır (panic veya wrap-around).
- *Modern dil özellikleri* - üst seviye diller kadar etkileyici ve ergonomik
  - Numaralandırmalar (Enums) ve desen eşleştirme.
  - Genelleştirmeler (Generics).
  - FFI ek yükü yok.
  - Sıfır maliyetli soyutlamalar.
  - Mükemmel derleyici hata mesajları.
  - Yerleşik bağımlılık yöneticisi.
  - Test için yerleşik destek.
  - Mükemmel Dil Sunucusu Protokolü (Language Server Protocol) desteği.

This slide should take about 3 minutes.

Burada fazla zaman geçirmeyin. Bu noktaların tümü ileride daha derinlemesine ele alınacaktır.

Sınıfa hangi dillerde deneyime sahip olduklarını mutlaka sorun. Cevabınıza bağlı olarak Rust'ın farklı özelliklerini vurgulayabilirsiniz:

- C veya C++ deneyimi: Rust, ödünç alma denetleyicisi (borrow checker) aracılığıyla tüm *çalışma zamanı hatalarını(runtime errors)* ortadan kaldırır. C ve C++'daki gibi performans elde edersiniz bununla birlikte bellek emniyetsizliği sorunları yaşamazsınız. Ayrıca desen eşleştirme (pattern matching) ve yerleşik bağımlılık yönetimi (dependency management) gibi yapılara sahip modern bir dil elde edersiniz.
- Java, Go, Python, JavaScript... ile deneyim: Bu dillerdekiyle aynı bellek emniyetine (safety) ve benzer bir yüksek seviyeli dil hissini elde edersiniz. Ayrıca C ve C++ gibi hızlı ve öngörülebilir performans (çöp toplayıcı yok) ve düşük seviyeli donanıma erişim (ihtiyaç duymanız halinde) elde edersiniz.

## 4.3 Deneme Alanı (Playground)

Rust Deneme Alanı kısa Rust programlarını çalıştırmanın kolay bir yolunu sunar ve bu kurstaki örnekler ve alıştırmaların temelini oluşturur. Deneme alanında ”merhaba-dünya”

ile başlayan programı çalıştırmayı deneyin. Deneme alanı birkaç kullanışlı özellik ile birlikte gelir:

- Kodunuzu ”standart” şekilde biçimlendirmek için ”Tools” altında `rustfmt` seçeneğini kullanın.
- Rust kodu oluşturmak için iki ana ”profil” (profile) vardır: Hata ayıklama (ekstra çalışma zamanı kontrolleri, daha az optimizasyon) ve Sürüm (release) (daha az çalışma zamanı kontrolü, çok fazla optimizasyon). Bunlara üst kısımdaki ”Hata Ayıklama” bölümünden erişilebilir.
- Eğer ilgileniyorsanız, oluşturulan asm kodunu görmek için ”...” altındaki ”ASM”yi kullanın.

This slide should take about 2 minutes.

Öğrenciler araya girerken onları deneme alanını açmaya ve biraz deney yapmaya teşvik edin. Kursun geri kalanında sekmeyi açık tutmaya ve bir şeyler denemeye teşvik edin. Bu özellikle Rust'ın optimizasyonları veya oluşturulan asm kodu hakkında daha fazla bilgi edinmek isteyen ileri düzey öğrenciler için faydalıdır.

# Chapter 5

# Türler ve Değerler

Bu bölüm yaklaşık 40 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Merhaba, Dünya | 5 dakika |
| Değişkenler | 5 dakika |
| Değerler | 5 dakika |
| Aritmetik | 3 dakika |
| Tür Çıkarımı (Type Inference) | 3 dakika |
| Alıştırma: Fibonacci | 15 dakika |

## 5.1 Merhaba, Dünya

Mümkün olan en basit Rust programına, klasik bir Merhaba Dünya programına geçelim:

```
fn main() {
    println!("Merhaba 🌍!");
}
```

Ne görüyorsunuz:

- Fonksiyonlar `fn` ile tanıtılır.
- `main` fonksiyon programın giriş noktasıdır (entry point).
- Bloklar, C ve C++'daki gibi küme parantezleriyle sınırları belirlenmiştir.
- Deyimler (Statements) `;` ile biter.
- Rust arınmış/pak/izole (hygienic) makrolara sahiptir, `println!` buna bir örnektir.
- Rust dilinindeki dizeler (strings) UTF-8 olarak kodlanmıştır ve herhangi bir Unicode karakteri içerebilir.

This slide should take about 5 minutes.

Bu slayt öğrencilerin Rust kodu konusunda rahat olmalarını sağlamaya çalışmaktadır. Önümüzdeki dört gün içinde çok şey görecekler, bu yüzden aşina bir şeyle küçükten başlıyoruz.

Anahtar noktalar:

- Rust, C/C++/Java geleneğindeki diğer dillere çok benzer. Rust emirli bir dildir (imperative) ve kesinlikle gerekmedikçe bir şeyleri yeniden keşfetmeye çalışmaz.
- Rust, Unicode gibi şeyleri tam olarak destekleyen modern bir yazılımdır.
- Rust, değişken sayıda argümana sahip olmak istediğiniz durumlar için makroları kullanır (fonksiyon yüklemesi (overloading) yoktur).
- Makroların 'arınmış/pak' olması, kullanıldıkları kapsamdaki (scope) tanımlayıcıları (identifiers) yanlışlıkla yakalamadıkları anlamına gelir. Rust makroları aslında yalnızca kısmi paktır (hygienic).
- Rust çok paradigmalı bir dildir. Örneğin, güçlü nesne yönelimli programlama özelliklerine sahiptir ve çeşitli fonksiyonel kavramları (fonksiyonel bir dil olmasa da) içerir.

## 5.2 Değişkenler

Rust, statik tür sistemine sahip olmasıyla tür emniyeti sağlar. Değişken bağlamaları (binding) `let` ile yapılır:

```
fn main() {
    let x: i32 = 10;
    println!("x: {x}");
    // x = 20;
    // println!("x: {x}");
}
```

This slide should take about 5 minutes.

- Değişkenlerin varsayılan olarak değiştirilemez (immutable) olduğunu göstermek için `x = 20`'in yorum satırını kaldırın. Değişikliklere izin vermek için `mut` anahtar kelimesini ekleyin.
- Bu slayt için kullanılmayan değişkenler veya gereksiz `mut` gibi uyarılar (warnings) etkinleştirildi. Dikkat dağıtan uyarılardan kaçınmak için çoğu slaytta bunlar atlanır. Atamayı içeren yorum satırını kaldırmayı deneyin, ancak `mut` anahtar sözcüğünü yerinde kalmaya devam etsin.
- Buradaki `i32` değişkenin türüdür. Bunun derleme zamanında bilinmesi gerekir, ancak tür çıkarımı (type inference) (daha sonra ele alınacaktır), programcının birçok durumda bunu atlamasına olanak tanır.

## 5.3 Değerler

Burada bazı temel yerleşik türler ve her türün değişmez değerlerinin (literal values) sözdizimi (syntax) verilmiştir.

| Türler | Değişmezler |
|---|---|
| İşaretli tamsayılar | `i8`, `i16`, `i32`, `i64`, `i128`, `isize` | `-10`, `0`, `1_000`, `123_i64` |

| Türler | Değişmezler |
|---|---|
| İşaretsiz tamsayılar | `u8`, `u16`, `u32`, `u64`, `u128`, `usize` | `0`, `123`, `10_u16` |
| Ondalıklı (floating point) sayılar | `f32`, `f64` | `3.14`, `-10.0e20`, `2_f32` |
| Ünikod skaler değerler | `char` | `'a'`, `'α'`, `'∞'` |
| Boole'sal | `bool` | `true`, `false` |

Türlerin genişlikleri/boyutları aşağıdaki gibidir:

- `iN`, `uN` ve `fN` türleri *N* bit genişliğinde,
- `isize` ve `usize` türleri göstericinin genişliğinde,
- `char` türü 32 bit genişliğinde,
- `bool` türü 8 bit genişliğinde.

This slide should take about 5 minutes.

Yukarıda gösterilmeyen birkaç sözdizimi (syntax) vardır:

- Rakamlardaki tüm alt çizgiler atlanabilir, bunlar yalnızca okunabilirlik içindir. Yani `1_000`, `1000` (veya `10_00`) olarak ve `123_i64`, `123i64` olarak yazılabilir.

## 5.4 Aritmetik

```
fn interproduct(a: i32, b: i32, c: i32) -> i32 {
    return a * b + b * c + c * a;
}

fn main() {
    println!("sonuç: {}", interproduct(120, 100, 248));
}
```

This slide should take about 3 minutes.

İlk defa `main` dışında bir fonksiyon görüyoruz, ancak anlamı açık: üç tamsayı alır ve bir tamsayı geri döndürür. Fonksiyonlar daha sonra daha detaylı olarak ele alınacaktır.

Aritmetik, benzer öncelik (precedence) kurallarıyla birlikte diğer dillerdeki aritmetiğe oldukça benzerdir.

Peki, tamsayı taşması ne olacak? C ve C++'da *işaretli* tamsayıların taşması aslında tanımsızdır (undefined) ve çalışma zamanında bilinmeyen şeyler olabilir. Rust'ta ise bu tanımlıdır.

Bir hata ayıklama (debug) inşasında paniğe (kontrollü) neden olan ve bir sürüm (release) inşasında sarmalanan (wrap) bir tamsayı taşmasını görmek için `i32`'leri `i16` olarak değiştirin. Taşma (overflow), doyum (saturating), elde (carrying) gibi işlemlerinin kontrolü için başka seçenekler de vardır. Bunlara metot sözdizimi ile erişilir, örneğin `(a * b).saturating_add(b * c).saturating_add(c * a)`.

Aslında derleyici sabit ifadelerin (constant expression) taşmasını algılayacaktır, bu nedenle örnek ayrı bir fonksiyon gerektirir.

## 5.5 Tür Çıkarımı (Type Inference)

Rust, türü belirlemek için değişkenin nasıl *kullanıldığına* bakacaktır:

```
fn takes_u32(x: u32) {
    println!("u32: {x}");
}

fn takes_i8(y: i8) {
    println!("i8: {y}");
}

fn main() {
    let x = 10;
    let y = 20;

    takes_u32(x);
    takes_i8(y);
    // takes_u32(y);
}
```

This slide should take about 3 minutes.

Bu slayt, Rust derleyicisinin değişken bildirimleri ve kullanımları tarafından verilen kısıtlamalara (constraints) dayanarak türleri nasıl çıkarım (inference) yapabildiğini gösterir.

Bu şekilde bildirilen değişkenlerin herhangi bir veriyi tutabilecek dinamik ”herhangi bir tür” olmadığını vurgulamak çok önemlidir. Böyle bir bildirim tarafından oluşturulan makine kodu, bir türün açık (explicit) bildirimiyle aynıdır. Derleyici bizim için işi yapar ve daha kısa kod yazmamıza yardımcı olur.

Bir tamsayı değişmezinin (integer literal) türünü hiçbir şeykısıtlamadığında, Rust varsayılan olarak türü `i32` yapar. Bu bazen hata mesajlarında ”{integer}” olarak görünür. Benzer şekilde, ondalıklı sayı değişmezleri (floating-point literals) için varsayılan tür `f64`'tür.

```
fn main() {
    let x = 3.14;
    let y = 20;
    assert_eq!(x, y);
    // HATA: `{float} == {integer}` için gerçekleştirim-uyarlama (implementation) yok
}
```

## 5.6 Alıştırma: Fibonacci

Fibonacci dizisi `[0,1]` ile başlar. n>1 için, n'inci Fibonacci sayısı yinelemeli olarak n-1'inci ve n-2'inci Fibonacci sayılarının toplamı olarak hesaplanır.

N'inci Fibonacci sayısını hesaplayan bir `fib(n)` fonksiyonu yazın. Bu fonksiyon ne zaman paniğe (panic) uğrar?

```
fn fib(n: u32) -> u32 {
    if n < 2 {
        // Temel durum.
        return todo!("Bunu gerçekleştirin (implement)");
    } else {
        // Özyinelemeli (recursive) durum.
        return todo!("Bunu gerçekleştirin (implement)");
    }
}

fn main() {
    let n = 20;
    println!("fib({n}) = {}", fib(n));
}
```

### 5.6.1 Çözüm

```
fn fib(n: u32) -> u32 {
    if n < 2 {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}

fn main() {
    let n = 20;
    println!("fib({n}) = {}", fib(n));
}
```

# Chapter 6

# Kontrol Akışı Temelleri

Bu bölüm yaklaşık 45 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Bloklar ve Kapsamlar | 5 dakika |
| if İfadeleri | 4 dakika |
| match İfadeleri | 5 dakika |
| Döngüler | 5 dakika |
| break ve continue | 4 dakika |
| Fonksiyonlar | 3 dakika |
| Makrolar | 2 dakika |
| Alıştırma: Collatz Sekansı | 15 dakika |

- Şimdi, Rust'ta bulunan birçok akış kontrol (flow control) çeşitlerini ele alacağız.
- Bunların çoğu diğer programlama dillerinde gördüklerinize çok tanıdık gelecektir.

## 6.1 Bloklar ve Kapsamlar

Rust dilindeki bir blok, `{}` parantezleri içine alınmış bir ifadeler dizisi (sequence of expressions) içerir. Her bloğun bir değeri ve bir türü vardır, bunlar, bloğun son ifadesinin değer ve türüdür:

```
fn main() {
    let z = 13;
    let x = {
        let y = 10;
        dbg!(y);
        z - y
    };
    dbg!(x);
    // dbg!(y);
}
```

Son ifade `;` ile bitiyorsa, ortaya çıkan değer ve tür `()` olur.

Bir değişkenin kapsamı (scope) onu çevreleyen blokla (enclosing block) sınırlıdır.

This slide should take about 5 minutes.

- ”dbg!”'nin, hızlı ve basitçe hata ayıklama için verilen bir ifadenin değerini yazdırmaya ve geri döndürmeye yarayan bir Rust makrosu olduğunu açıklayabilirsiniz.
- Bloktaki son satırı değiştirerek bloğun değerinin nasıl değiştiğini gösterebilirsiniz. Örneğin, noktalı virgül eklemek/kaldırmak veya bir `return` kullanmak.
- y değişkenine kapsamı (scope) dışından erişmeye çalışmanın derleme hatası olacağını gösterin.
- Değerler, yığındaki (stack) verileri hala orada olsa bile, kapsamlarının (scope) dışına çıktıklarında etkili bir şekilde ”tahsisleri geri verilir (deallocated)”.

## 6.2 `if` ifadeleri

`if` ifadelerini tam olarak diğer dillerdeki `if` deyimleri (statement) gibi kullanabilirsiniz:

```
fn main() {
    let x = 10;
    if x == 0 {
        println!("sıfır!");
    } else if x < 100 {
        println!("büyükçe");
    } else {
        println!("aşırı büyük");
    }
}
```

Ayrıca, `if`'i ifade (expresssion) olarak da kullanabilirsiniz. Her bloğun son ifadesi `if` ifadesinin değeri olur:

```
fn main() {
    let x = 10;
    let size = if x < 20 { "küçük" } else { "büyük" };
    println!("sayının boyutu: {}", size);
}
```

This slide should take about 4 minutes.

`if` bir ifade (expression) olduğundan ve belirli bir türe sahip olması gerektiğinden, her iki dal bloğunun da aynı türde olması gerekir. İkinci örnekte `"küçük"`ten sonra `;` eklerseniz ne olacağını gösterin.

Bir `if` ifadesi diğer ifadelerle aynı şekilde kullanılmalıdır. Örneğin, bir `let` ifadesinde kullanıldığında, ifadenin `;` ile de sonlandırılması gerekir. Derleyici hatasını görmek için `println!`'den önceki `;`'yi kaldırın.

## 6.3 `match` İfadeleri

`match` bir değeri bir veya daha fazla seçeneğe göre kontrol etmek için kullanılabilir:

```
fn main() {
    let val = 1;
    match val {
        1 => println!("bir"),
        10 => println!("on"),
        100 => println!("yüz"),
        _ => {
            println!("başka bir şey");
        }
    }
}
```

`if` ifadeleri gibi `match` ifadesi de bir değeri geri döndürebilir (return);

```
fn main() {
    let flag = true;
    let val = match flag {
        true => 1,
        false => 0,
    };
    println!("{flag}'in değeri = {val}");
}
```

This slide should take about 5 minutes.

- `match` kolları (arms) yukarıdan aşağıya doğru değerlendirilir ve eşleşen ilk kolun karşılık gelen gövdesi (body) çalıştırılır.
- `switch`'in diğer dillerdeki çalışma şekli gibi durumlar (cases) arasında aşağıya geçiş (fall-through) yoktur.
- Bir `match` kolunun gövdesi tek bir ifade (expression) veya bir blok olabilir. Teknik olarak bu aynı şeydir, çünkü bloklar da bir ifadedir, ancak öğrenciler bu noktadaki denk yapıyı (symmetry) tam olarak anlamayabilirler.
- `match` ifadelerinin kapsamlı (exhaustive) olması gerekir, yani ya tüm olası değerleri kapsamaları ya da `_` gibi varsayılan bir duruma (case) sahip olmaları gerekir. Kapsamlılık (Exhaustiveness) enumlarla gösterilmesi en kolay olanıdır, ancak enumlar henüz tanıtılmadı. Bunun yerine, en basit ilkel tür olan `bool` üzerinde eşleştirmeyi (matching) gösteriyoruz.
- Bu slayt, desen eşleştirmesinden (pattern matching) bahsetmeden `match`'i tanıtıyor ve öğrencilere çok fazla bilgi yüklemeden sözdizimine (syntax) aşina olma şansı veriyor. Desen eşleştirmesi hakkında yarın daha detaylı konuşacağız, bu yüzden burada çok fazla detaya girmemeye çalışın.

## Daha Fazlasını Keşfedin

- `match` kullanımını daha da motive etmek için, örnekleri `if` ile yazılmış eşdeğerleriyle karşılaştırabilirsiniz. İkinci durumda, bir `bool` üzerinde eşleme (matching) yapmak, `if {} else {}` bloğuna oldukça benzer. Ancak birden fazla durumu kontrol eden ilk örnekte, bir `match` ifadesi `if {} else if {} else if {} else` ifadesinden daha az ve öz olabilir.

- `match` ayrıca eşleşme filtrelerini (match guards) da destekler, bu da eşleşme kolunun (match arm) alınıp alınmaması gerektiğini belirlemek için değerlendirilecek keyfi bir mantıksal koşul (condition) eklemenize olanak tanır. Ancak eşleşme filtreleri hakkında konuşmak, bu slaytta kaçınmaya çalıştığımız desen eşleştirmesi (pattern matching) hakkında açıklama gerektirir.

## 6.4 Döngüler

Rust dilinde üç döngüsel anahtar kelime vardır: `while`, `loop`, ve `for`:

### `while`

`while` anahtar kelimesi diğer dillerdeki gibi işler ve koşul doğru olduğu sürece döngü gövdesini yürütür (execute).

```
fn main() {
    let mut x = 200;
    while x >= 10 {
        x = x / 2;
    }
    dbg!(x);
}
```

### 6.4.1 `for`

`for` döngüsü değer aralıkları (ranges of values) veya bir koleksiyondaki (collection) öğeler üzerinde adımlama yapar (iterate):

```
fn main() {
    for x in 1..5 {
        dbg!(x);
    }

    for elem in [2, 4, 8, 16, 32] {
        dbg!(elem);
    }
}
```

- `for` döngülerinin başlığının altında, farklı türdeki aralıklar/koleksiyonlar üzerinde adımlamayı gerçekleştirmek için ”adımlayıcılar” adı verilen bir kavram kullanılır. Adımlayıcılar (iterators) daha sonra daha ayrıntılı olarak tartışılacaktır.
- İlk `for` döngüsünün yalnızca 4'e kadar adımladığını unutmayın. Kapalı bir aralık için `1..=5` sözdizimini gösterin.

### 6.4.2 `loop`

`loop` deyimi, bir `break`'e kadar sonsuz döner.

```
fn main() {
    let mut i = 0;
    loop {
```

```
        i += 1;
        dbg!(i);
        if i > 100 {
            break;
        }
    }
}
```

- `loop` deyimi `while true` döngüsü gibi çalışır. Bağlantılara sonsuza kadar hizmet edecek sunucular gibi şeyler için kullanın.

## 6.5 `break` ve `continue`

Bir sonraki adıma hemen başlamak istiyorsanız `continue` kullanın.

Herhangi bir döngüden erken çıkmak istiyorsanız, `break` kullanın. `loop` ile, isteğe bağlı bir ifade yazılabilir ve bu ifade, `loop` ifadesinin değeri olur.

```
fn main() {
    let mut i = 0;
    loop {
        i += 1;
        if i > 5 {
            break;
        }
        if i % 2 == 0 {
            continue;
        }
        dbg!(i);
    }
}
```

This slide and its sub-slides should take about 4 minutes.

`loop`'un önemsiz olmayan (non-trivial) bir değer geri döndüren tek döngüsel yapı olduğunu unutmayın. Bunun nedeni, en az bir `break` ifadesinde sonlanmasının garantili olmasıdır (koşul başarısız olduğunda da sonlanabilen `while` ve `for` döngülerinin aksine).

### 6.5.1 Etiketler

Hem `continue` ve `break` isteğe bağlı olarak iç içe geçmiş döngülerden çıkmak için kullanılan bir etiket (label) argümanını alabilir:

```
fn main() {
    let s = [[5, 6, 7], [8, 9, 10], [21, 15, 32]];
    let mut elements_searched = 0;
    let target_value = 10;
    'outer: for i in 0..=2 {
        for j in 0..=2 {
            elements_searched += 1;
            if s[i][j] == target_value {
                break 'outer;
            }
```

```
        }
    }
    dbg!(elements_searched);
}
```

- Etiketli break, keyfi olarak yazılan bloklarda da çalışır, örneğin.
  ```
  'label: {
      break 'label;
      println!("Bu satır geçilece");
  }
  ```

## 6.6 Fonksiyonlar

```
fn gcd(a: u32, b: u32) -> u32 {
    if b > 0 { gcd(b, a % b) } else { a }
}

fn main() {
    dbg!(gcd(143, 52));
}
```

This slide should take about 3 minutes.

- Bildirim (declaration) parametrelerinin ardından bir tür (bazı programlama dillerinin tersi) ve ardından bir geri dönüş türü gelir.
- Bir fonksiyon gövdesindeki (veya herhangi bir bloktaki) son ifade (expression), geri dönüş değeri (return value) olur. İfadenin sonundaki `;` işaretini silmeniz yeterlidir. `return` anahtar kelimesi erken geri dönüş (early return) için kullanılabilir, ancak fonksiyonun sonunda "yalın (bare) değer" kullanmak daha yaygındır (idiomatic) (`return`kullanmak için `gcd`'yi yeniden düzenleyin - refactor).
- Bazı fonksiyonların geri dönüş değeri (return value) yoktur ve 'birim türü (unit type)', `()` geri döndürürler. Eğer geri dönüş türü yazılmazsa, derleyici birim türünü çıkarım (infer) yapacaktır.
- Fonksiyon yüklemesi (overloading) desteklenmez -- her fonksiyonun tek bir uygulaması (implementation) vardır.
  - Her zaman sabit sayıda parametre alır. Varsayılan argümanlar (default arguments) desteklenmez. Makrolar değişken sayıda argüman alan fonksiyonları (variadic functions) desteklemek için kullanılabilir.
  - Her zaman tek bir parametre türü kümesi alır. Bu türler genelleştirilmiş (generic) olabilir, bu konuya daha sonra değinilecektir.

## 6.7 Makrolar

Makrolar, derleme sırasında Rust koduna genişletilir ve değişken sayıda argüman alabilir. Sonda olan `!` işaretiyle ayırt edilirler. Rust standart kütüphanesi çeşitli yararlı makrolar içerir.

- `println!(format, ..)`, `std::fmt`'de açıklanan biçimi (formatting) uygulayarak standart çıktıya bir satır yazdırır.
- `format!(format, ..)` tıpkı `println!` gibi işler ancak sonucu bir dize (string) olarak geri döndürür.

- `dbg!(expression)` ifadenin değerini kaydeder (loggging) ve onu geri döndürür.
- `todo!()` kodun bir kısmını henüz uygulanmamış (not-yet-implemented) olarak işaretler. Eğer yürütülürse (execute), paniğe neden olur.

```
fn factorial(n: u32) -> u32 {
    let mut product = 1;
    for i in 1..=n {
        product *= dbg!(i);
    }
    product
}

fn fizzbuzz(n: u32) -> u32 {
    todo!()
}

fn main() {
    let n = 4;
    println!("{n}! = {}", factorial(n));
}
```

This slide should take about 2 minutes.

Bu bölümden çıkarılacak sonuç, bu ortak kolaylıkların mevcut olduğu ve bunların nasıl kullanılacağıdır. Bunların neden makro olarak tanımlandığı ve neye genişledikleri özellikle kritik değildir.

Kurs, makroların tanımlanmasını (define) kapsamaz, ancak daha sonraki bir bölümde `derive` makrolarının kullanımı anlatılacaktır.

### Daha Fazlasını Keşfedin

Standart kütüphane tarafından sağlanan birçok başka kullanışlı makro vardır. Öğrenciler daha fazlasını öğrenmek isterse onlarla paylaşabileceğiniz bazı diğer örnekler:

- `assert!` ve ilgili makrolar, kodunuza doğrulamalar (assertions) eklemek için kullanılabilir. Bu makrolar, test yazımında yoğun şekilde kullanılır.
- `unreachable!`, hiçbir zaman ulaşılmaması gereken bir kontrol akışı dalını işaretlemek için kullanılır.
- `eprintln!` stderr'e yazdırmanıza olanak tanır.

## 6.8 Alıştırma: Collatz Sekansı

Collatz sanısı, sıfırdan büyük herhangi bir $n_1$ için aşağıdaki şekilde tanımlanır:

- Eğer $n_i$ 1 ise, dizi $n_i$'de sonlanır.
- Eğer $n_i$ çift ise, o zaman $n_{i+1} = n_i / 2$.
- Eğer $n_i$ tek sayı ise $n_{i+1} = 3 * n_i + 1$.

Örneğin, $n_1 = 3$ ile başalrsak:

- 3 tek sayıdır, dolayısıyla $n_2 = 3 * 3 + 1 = 10$;
- 10 çift sayıdır, dolayısıyla $n_3 = 10 / 2 = 5$;

- 5 tek sayıdır, dolayısıyla $n_4$ = 3 * 5 + 1 = 16;
- 16 çift sayıdır, dolayısıyla $n_5$ = 16 / 2 = 8;
- 8 çift sayıdır, dolayısıyla $n_6$ = 8 / 2 = 4;
- 4 çift sayıdır, dolayısıyla $n_7$ = 4 / 2 = 2;
- 2 çifttir, bu yüzden $n_8$ = 1; ve
- sekans sonlandırılır.

Belirli bir başlangıç n için collatz sekansının uzunluğunu hesaplayan bir fonksiyon yazın.

```
/// `n`'den başlayan collatz sekansının uzunluğunu belirle.
fn collatz_length(mut n: i32) -> u32 {
  todo!("Bunu gerçekleştirin (implement)")
}

fn main() {
    println!("Uzunluk: {}", collatz_length(11)); // 15 olmalı
}
```

### 6.8.1 Çözüm

```
/// `n`'den başlayan collatz sekansının uzunluğunu belirle.
fn collatz_length(mut n: i32) -> u32 {
    let mut len = 1;
    while n > 1 {
        n = if n % 2 == 0 { n / 2 } else { 3 * n + 1 };
        len += 1;
    }
    len
}

fn main() {
    println!("Uzunluk: {}", collatz_length(11)); // 15 olmalı
}
```

# Part II

# Gün 1: Öğleden Sonra

# Chapter 7

# Tekrar Hoş Geldiniz

Bu oturum 10 dakikalık aralar dahil yaklaşık 2 saat 45 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Demetler (Tuples) ve Diziler (Arrays) | 35 dakika |
| Referanslar | 55 dakika |
| Kullanıcı Tanımlı Türler | 1 saat |

# Chapter 8

# Demetler (Tuples) ve Diziler (Arrays)

Bu bölüm yaklaşık 35 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Diziler | 5 dakika |
| Demetler | 5 dakika |
| Dizi Adımlama (Iteration) | 3 dakika |
| Desenler ve Çözümlenme (Destructuring) | 5 dakika |
| Alıştırma: İçiçe Diziler | 15 dakika |

- Rust'ta (primitive) ilkel türlerin nasıl çalıştığını gördük. Şimdi yeni bileşik (composite) türler oluşturmaya başlamanın zamanı geldi.

## 8.1 Diziler

```
fn main() {
    let mut a: [i8; 5] = [5, 4, 3, 2, 1];
    a[2] = 0;
    println!("a: {a:?}");
}
```

This slide should take about 5 minutes.

- Diziler ayrıca kısaltma sözdizimi (shorthand syntax) kullanılarak da ilklendirilebilir, örneğin `[0; 1024]`. Bu, tüm elemanları aynı değere başlatmak istediğinizde veya manuel olarak başlatılması zor olan büyük bir diziniz varsa yararlı olabilir.
- Bir `[T; N]` dizi türünün değeri , aynı `T` türdeki `N` (derleme zamanı sabiti) öğelerini/elemanlarını tutar. Dizinin uzunluğunun *türünün parçası* olduğuna dikkat edin, bu da `[u8; 3]' ve '[u8; 4]` dizilerinin iki farklı tür olarak kabul edildiği anlamına gelir. Dilimler (slices), ki çalışma zamanında belirlenen bir boyuta sahiptir, daha sonra ele alınacaktır.

- Sınırların dışında (out-of-bounds) bir dizi (array) elemanına erişmeyi deneyin. Derleyici, indeksin emniyetsiz (unsafe) olduğunu belirleyebilir ve kodu derlemez:

```
fn main() {
    let mut a: [i8; 5] = [5, 4, 3, 2, 1];
    a[6] = 0;
    println!("a: {a:?}");
}
```

- Dizi erişimleri çalışma zamanında kontrol edilir. Rust genellikle bu kontrolleri optimize edebilir; yani, derleyici erişimin güvenli olduğunu kanıtlayabilirse, daha iyi performans için çalışma zamanı kontrolünü kaldırır. Bu kontroller, unsafe Rust kullanılarak da atlanabilir. Optimizasyon o kadar iyidir ki, çalışma zamanı kontrollerinin başarısız olduğu bir örnek vermek zordur. Aşağıdaki kod derlenecektir ancak çalışma zamanında panik (panic) ile sonuçlanacaktır:

```
fn get_index() -> usize {
    6
}

fn main() {
    let mut a: [i8; 5] = [5, 4, 3, 2, 1];
    a[get_index()] = 0;
    println!("a: {a:?}");
}
```

- Dizilere değer atamak için değişmezleri (literals) kullanabiliriz.
- `println!` makrosu `?` biçim (format) parametresiyle hata ayıklama gerçekleştirimini/uyarlamasını (implementation) ister: `{}` varsayılan çıktıyı verir, `{:?}` hata ayıklama çıktısını verir. Tamsayılar (integer) ve dizeler (string) gibi türler varsayılan çıktıyı uygular (default output), ancak diziler (arrays) yalnızca hata ayıklama çıktısını (debug output) uygular. Bu, bahsi geçen kodda hata ayıklama çıktısını kullanmamız gerektiği anlamına gelir.
- `#` eklemek, örneğin `{a:#?}`, okunması daha kolay olabilecek "güzel yazdırma (pretty printing)" biçimini (format) çağırır.
- Arrays are not heap-allocated. They are regular values with a fixed size known at compile time, meaning they go on the stack. This can be different from what students expect if they come from a garbage collected language, where arrays may be heap allocated by default.

## 8.2 Demetler

```
fn main() {
    let t: (i8, bool) = (7, true);
    dbg!(t.0);
    dbg!(t.1);
}
```

This slide should take about 5 minutes.

- Diziler gibi, demetlerin (tuple) de sabit bir uzunluğu vardır.
- Demetler farklı türdeki değerleri bir bileşik tür (compound type) halinde gruplandırır.

- Bir demetin alanlarına (fields) nokta ve değerin indeksi ile erişilebilir, örneğin `t.0`, `t.1`.
- Boş demet `()`, "birim türü (unit type)" olarak anılır ve, diğer dillerdeki `void`'e benzer şekilde, bir geri dönüş değerinin (return value) bulunmadığını belirtir.

## 8.3 Dizi Adımlama (Iteration)

'for' deyimi (statement) diziler üzerinde adımlamayı/dolaşmayı destekler (ancak demetler (tuples) üzerinde desteklemez).

```
fn main() {
    let primes = [2, 3, 5, 7, 11, 13, 17, 19];
    for prime in primes {
        for i in 2..prime {
            assert_ne!(prime % i, 0);
        }
    }
}
```

This slide should take about 3 minutes.

Bu fonksiyonellik `IntoIterator` özelliğini (trait) kullanıyor, ancak bunu henüz ele almadık.

`assert_ne!` makrosu burada yenidir. Ayrıca `assert_eq!` ve `assert!` makroları da vardır. Bunlar her zaman kontrol edilirken, `debug_assert!` gibi yalnızca hata ayıklama varyantları sürüm inşasında (release build) hiçbir şeye derlenmez.

## 8.4 Desenler ve Çözümlenme (Destructuring)

Rust, bir demet (tuple) gibi daha büyük bir değeri, onu oluşturan parçalara çözümlemek (destructure) için desen eşleştirmeyi (pattern matching) kullanmayı destekler:

```
fn check_order(tuple: (i32, i32, i32)) -> bool {
    let (left, middle, right) = tuple;
    left < middle && middle < right
}

fn main() {
    let tuple = (1, 5, 3);
    println!(
        "{tuple:?}: {}",
        if check_order(tuple) { "sıralanmış" } else { "sıralı de" }
    );
}
```

This slide should take about 5 minutes.

- Burada kullanılan desenler (patterns) "reddedilemez (irrefutable)"dir, yani derleyici = sağındaki değerin desenle aynı yapıya sahip olduğunu statik olarak doğrulayabilir.
- Bir değişken adı, her zaman herhangi bir değerle eşleşen reddedilemez (irrefutable) bir desendir, bu nedenle tek bir değişkeni bildirmek için `let`'i kullanabiliriz.
- Rust ayrıca koşullu ifadelerde desenlerin (patterns) kullanılmasını destekleyerek eşitlik karşılaştırmasının (comparison) ve çözümlenmenin (destructuring) aynı anda

gerçekleşmesine olanak tanır. Bu desen eşleştirme stili daha sonra daha ayrıntılı olarak tartışılacaktır.
- Desenin (pattern) eşleştirilen değerle eşleşmediğinde oluşacak derleyici hatasını göstermek için yukarıdaki örnekleri düzenleyin.

## 8.5 Alıştırma: İçiçe Diziler

Diziler başka diziler de içerebilir:

```
let array = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];
```

Bu değişkenin türü nedir?

Bir matrisin devrini (transpose) yapacak (satırları sütunlara çevirecek) `transpose` fonksiyonu yazmak için yukarıdaki gibi bir dizi kullanın:

```
           ⎛⎡1 2 3⎤⎞      ⎡1 4 7⎤
"transpose"⎜⎢4 5 6⎥⎟  "=="⎢2 5 8⎥
           ⎝⎣7 8 9⎦⎠      ⎣3 6 9⎦
```

Aşağıdaki kodu https://play.rust-lang.org/ adresine kopyalayın ve fonksiyonu gerçekleştirin (implement). Bu fonksiyon yalnızca 3x3'lük matrislerde çalışır.

```
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    todo!()
}

fn main() {
    let matrix = [
        [101, 102, 103], // <-- bu yorum, rustfmt'nin yeni satır eklemesini sağlar
        [201, 202, 203],
        [301, 302, 303],
    ];

    dbg!(matrix);
    let transposed = transpose(matrix);
    dbg!(transposed);
}
```

### 8.5.1 Çözüm

```
fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
    let mut result = [[0; 3]; 3];
    for i in 0..3 {
        for j in 0..3 {
            result[j][i] = matrix[i][j];
        }
    }
    result
}

fn main() {
    let matrix = [
```

```
        [101, 102, 103], // <-- bu yorum, rustfmt'nin yeni satır eklemesini sağlar
        [201, 202, 203],
        [301, 302, 303],
    ];

    dbg!(matrix);
    let transposed = transpose(matrix);
    dbg!(transposed);
}
```

# Chapter 9

# Referanslar

Bu bölüm yaklaşık 55 dakika sürmelidir. İçerir:

| Slayt | Süre |
|---|---|
| Paylaşılan (Shared) Referanslar | 10 dakika |
| Özel (Exclusive) Referanslar | 5 dakika |
| Dilimler (Slices) | 10 dakika |
| Dizeler (Strings) | 10 dakika |
| Referans Geçerliliği (Validity) | 3 dakika |
| Alıştırma: Geometri | 20 dakika |

## 9.1 Paylaşılan (Shared) Referanslar

Bir referans, değerin sahipliğini almadan başka bir değere erişmenin bir yolunu sağlar ve ”ödünç alma (borrowing)” olarak da adlandırılır. Paylaşılan referanslar (shared reference) salt okunurdur (read-only) ve referans edilen veriler değiştirilemez.

```
fn main() {
    let a = 'A';
    let b = 'B';

    let mut r: &char = &a;
    dbg!(r);

    r = &b;
    dbg!(r);
}
```

`T` türüne yönelik paylaşılan referansın türü `&T`'dir. `&` operatörüyle bir referans değeri (reference value) oluşturulur. `*` operatörü bir referansın içeriğine eriştirir (dereference) ve onun değerini verir.

This slide should take about 7 minutes.

- Rust'ta referanslar hiçbir zaman null olamaz, bu yüzden null kontrolü (null checking) gerekli değildir.

- Bir referansın, atıfta bulunduğu (refer) değeri ”ödünç aldığı (barrow)” söylenir ve bu, göstericilere aşina olmayan öğrenciler için iyi bir modeldir: kod, değere erişmek için referansı kullanabilir, ancak değer yine de orijinal değişken tarafından ”sahiplidir (owned)”. Kurs, 3. günde sahiplik (ownership) konusunda daha ayrıntılı bilgi verecektir.
- Referanslar göstericiler (pointers) olarak gerçekleştirilir (implement) ve önemli bir avantaj şudur ki, gösterdikleri şeyin boyutundan çok daha küçük olabilmeleridir. C veya C++'a aşina olan öğrenciler referansları göstericiler olarak tanıyacaklardır. Kursun ilerleyen bölümlerinde Rust'ın, ham göstericilerin (raw pointers) kullanılmasından kaynaklanan bellek emniyeti hatalarını (memory-safety bugs) nasıl önlediği ele alınacaktır.
- Açık bir şekilde (explicit) `&` ile referans vermek genellikle gereklidir. Ancak, Rust metotları çağırırken referans verme ve referanstan çıkarma (dereferencing) işlemlerini otomatik olarak gerçekleştirir.
- Rust bazı durumlarda, özellikle de metotları çağırırken (`r.is_ascii()`'i deneyin) otomatik olarak referansın içeriğine erişecektir (auto-dereference). C++'daki gibi `->` operatörüne gerek yoktur.
- Bu örnekte, `r` değiştirilebilir (mutable) olduğundan ona tekrar atama yapılabilir (reassigned) (`r = &b`). Bunun `r`'yi yeniden bağladığını (re-bind) ve böylece başka bir şeye atıfta bulunduğunu (refer) unutmayın. Bu C++ dilinden, bir referansa atamanın (assignment) referans verilenin değerini değiştirmesinden, farklıdır.
- Paylaşılan bir referans (shared reference), atıfta bulunduğu (refer) değerin değiştirilmesine (modify), bu değer değiştirilebilir (mutable) olsa bile, izin vermez. `*r = 'X'`'i deneyin.
- Rust, yeterince uzun süre hayatta olmalarını sağlamak için tüm referansların ömrülerini (lifetime) takip eder. Emniyetli (safe) Rust'ta boşa çıkan referanslar (dangling reference) oluşamaz.
- Sahipliğe (ownership) geçince ödünç alma (borrowing) konusunu daha detaylı konuşacağız. Sahiplik (ownership) konusuna geldiğimizde ödünç alma ve boşa çıkan referansları (dangling references) daha detaylı konuşacağız.

## 9.2 Özel (Exclusive) Referanslar

Değiştirilebilir referanslar (mutable reference) olarak da bilinen özel (exclusive) referanslar, atıfta bulundukları (refer) değerin değiştirilmesine olanak tanır. Aynı zamanda, `&mut T` türüne sahiplerdir.

```
fn main() {
    let mut point = (1, 2);
    let x_coord = &mut point.0;
    *x_coord = 20;
    println!("point: {point:?}");
}
```

This slide should take about 5 minutes.

Anahtar noktalar:

- ”Özel (exclusive)”, değere erişmek için yalnızca bu referansın kullanılabileceği anlamına gelir. Aynı anda başka hiçbir referans (paylaşılan veya özel olsun farketmez) mevcut olamaz ve özel referans (exclusive) varken atıfta bulunan değere erişilemez. `x_coord` hayattayken bir `&point.0` oluşturmayı veya `point.0`'ı değiştirmeyi deneyin.
- `let mut x_coord: &i32` ve `let x_coord: &mut i32` arasındaki farka dikkat ettiğinizden emin olun. Birincisi, farklı değerlere bağlanabilen (bind) paylaşılan bir referansı (shared reference) temsil ederken, ikincisi, değiştirilebilir bir değere (mutable value) özel bir referansı (exclusive reference) temsil eder (represent).

## 9.3 Dilimler (Slices)

Bir dilim (slice) size daha büyük bir koleksiyonun içinden bir görüntü verir:

```
fn main() {
    let a: [i32; 6] = [10, 20, 30, 40, 50, 60];
    println!("a: {a:?}");

    let s: &[i32] = &a[2..4];

    println!("s: {s:?}");
}
```

- Dilimler (slice), dilimlenmiş türden verileri ödünç alır (borrowing).

This slide should take about 7 minutes.

- `a`'yı ödünç alıp (borrowing) başlangıç ve bitiş indekslerini köşeli parantez içinde belirterek bir dilim (slice) oluşturuyoruz.
- Dilim (slice) indeks 0'da başlıyorsa, Rust'ın aralık sözdizimi (range syntax) başlangıç indeksini kullanmamaya izin verir; bu, `&a[0..a.len()]` ve `&a[..a.len()]` dilimlerinin aynı olduğu anlamına gelir.
- Aynı şey bitiş indeksi için de geçerlidir, dolayısıyla `&a[2..a.len()]` ve `&a[2..]` aynıdır.
- Tüm bir dizinin dilimini (slice) kolayca oluşturmak için `&a[..]`'ı kullanabiliriz.
- `s`,`i32`'lerin bir dilimine (slice) referanstır. `s` (`&[i32]`) türünün artık dizi uzunluğundan bahsetmediğine dikkat edin. Bu bize farklı boyutlardaki dilimler üzerinde hesaplama yapmamızı sağlar.
- Dilimler (slice) her zaman başka bir nesneden ödünç alır (borrow). Bu örnekte `a`'nın en azından dilimimiz (slice) kadar 'canlı (alive)' (kapsamında) kalması gerekiyor.
- Bir dilimi (slice) oluşturduktan sonra onu ”büyütemezsiniz”:
  - Dilim (slice), destekleyici arabelleğe (backing buffer) sahip olmadığından, dilimin öğelerine ekleyemezsiniz.
  - Bir dilimi (slice), destekleyici arabelleğin (backing buffer) daha büyük bir bölümünü gösterecek şekilde büyütemezsiniz. Dilim, temel arabellek (underlying buffer) hakkında bilgiyi kaybeder ve bu nedenle ne kadar büyütülebileceğini bilemezsiniz.
  - Daha büyük bir dilim (slice) elde etmek için, orijinal arabelleği (buffer) kullanarak daha büyük bir dilim (slice) oluşturmalısınız.

## 9.4 Dizeler (Strings)

Artık Rust'taki iki dize (string) türünü anlayabiliyoruz:

- `&str`, `&[u8]`e benzer şekilde, UTF-8 olarak kodlanmış baytların bir dilimidir (slice).
- `String`, `Vec<T>`ye benzer şekildedir, UTF-8 olarak kodlanmış baytlardan oluşan sahipli bir arabellektir.

```
fn main() {
    let s1: &str = "Dünya";
    println!("s1: {s1}");

    let mut s2: String = String::from("Merhaba ");
    println!("s2: {s2}");

    s2.push_str(s1);
    println!("s2: {s2}");

    let s3: &str = &s2[2..9];
    println!("s3: {s3}");
}
```

This slide should take about 10 minutes.

- `&str`, bir bellek bloğunda saklanan UTF-8 olarak kodlanmış dize (string) verilerine değiştirilemez (immutable) referans olan bir dize dilimi (string slice) sunar. Dize değişmezleri (string literals) (`"Hello"`), programın ikili (binary) dosyasında saklanır.
- Rust'ın `String` türü, baytlardan oluşan bir vektörün bir sarmalayıcı türüdür. Bir `Vec<T>` gibi, bu türden veri `String` türü tarafından sahiplidir (owned).
- Diğer birçok türde olduğu gibi, `String::from()` bir dize değişmezinden (string literal) bir string oluşturur; `String::new()` yeni bir boş dize oluşturur ve dizeye veriler `push()` ve `push_str()` metotları kullanılarak eklenebilir.
- `format!()` makrosu, dinamik değerlerden sahipli bir dize oluşturmak için uygun bir yoldur. Aynı biçimin `println!()` makrosundaki gibi aynı biçim tanımlamasını kabul eder.
- `String`'den `&str` dilimlerini (slices) & ve isteğe bağlı olarak aralık seçimi (range selection) ile ödünç alabilirsiniz (barrow). Karakter sınırlarının dışını da kapsayacak bir bayt aralığı seçerseniz, ifade (expression) paniğe neden olacaktır. `chars` adımlayıcısı, karakterler üzerinde adımlama yapar ve karakter sınırlarını doğru bulmaya çalışmak yerine tercih edilir.
- C++ programcıları için: `&str`'yi C++'daki `std::string_view` olarak düşünün, ancak her zaman bellekteki geçerli bir dizeye işaret eder. Rust dilindeki `String`, C++'daki `std::string`'in kabaca eşdeğeridir (ana fark: yalnızca UTF-8 olarak kodlanmış baytlar içerebilir ve asla küçük dize optimizasyonu / small-string optimization kullanmaz).
- Bayt dizelerinden değişmezler (Byte strings literals), doğrudan bir `&[u8]` değeri oluşturmanıza olanak tanır:

  ```
  fn main() {
      println!("{:?}", b"abc");
  ```

```
    println!("{:?}", &[97, 98, 99]);
}
```

- Ham (raw) dizeler, kaçış karakterlerininin devre dışı bırakıldığı bir `&str` değeri oluşturmanıza olanak tanır: `r"\n" == "\\n"`. Dizenin içindeki çift tırnakları, dizeyi oluşturan tırnakların her iki tarafınaa eşit sayıda # karakteri koyarak görebilirsiniz:

```
fn main() {
    println!(r#"<a href="link.html">link</a>"#);
    println!("<a href=\"link.html\">link</a>");
}
```

## 9.5 Referans Geçerliliği (Validity)

Rust, referansların her zaman emniyetli (safe) kullanılmasını sağlayan bir dizi kural uygular. Bunlardan biri, referansların asla `null` olamayacağıdır; bu da `null` kontrolü yapmadan emniyetli kullanılmalarını sağlar. Şimdi bakacağımız diğer kural ise, referansların gösterdikleri veriden *daha uzun ömürlü (outlive)* olamayacaklarıdır.

```
fn main() {
    let x_ref = {
        let x = 10;
        &x
    };
    dbg!(x_ref);
}
```

This slide should take about 3 minutes.

- Bu slayt, Rust'ın referanslar için diğer dillerden farklı kuralları olduğundan, öğrencilerin referansların sadece göstericiler (pointers) olmadığını düşünmelerini sağlar.
- Rust'ın ödünç alma (borrowing) kurallarının geri kalanına, 3. günde Rust'ın sahiplik (ownership) sisteminden bahsederken bakacağız.

### Daha Fazlasını Keşfedin

- Rust dilindeki null olabilirliğin karşılığı `Option` türüdür, bu tür herhangi bir türü ”null olabilir (nullable)” hâle getirmek için kullanılabilir (sadece referanslar/göstericiler değil). Ancak henüz enum'ları veya desen eşleştirmeyi (pattern matching) tanıtmadık, bu yüzden burada bu konulara fazla girmemeye çalış.

## 9.6 Alıştırma: Geometri

3 boyutlu geometri için, bir noktayı `[f64;3]` olarak temsil eden birkaç yardımcı fonksiyon oluşturacağız. Fonksiyon imzalarını belirlemek size kalmış.

```
// Bir vektörün büyüklüğünü, koordinatlarının karelerini toplayarak hesaplayın
// ve karekökünü alın. Karekökü hesaplamak için `sqrt()` metodunu kullanın,
// örneğin `v.sqrt()` gibi.
```

```rust
fn magnitude(...) -> f64 {
    todo!()
}

// Bir vektörü, büyüklüğünü hesaplayarak ve tüm koordinatlarını bu büyüklüğe
// bölererek normalize edin.


fn normalize(...) {
    todo!()
}

// Çalışmanızı test etmek için aşağıdaki `main` fonksiyonunu kullanın.

fn main() {
    println!("Birim vektörünün büyüklüğü: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("{v:?} vektörünün büyüklüğü: {}", magnitude(&v));
    normalize(&mut v);
    println!("Normalize edildikten sonra {v:?} vektörünün büyüklüğü: {}", magnitude(&v)
}
```

### 9.6.1 Çözüm

```rust
/// Verilen vektörün büyüklüğünü hesaplayın.
fn magnitude(vector: &[f64; 3]) -> f64 {
    let mut mag_squared = 0.0;
    for coord in vector {
        mag_squared += coord * coord;
    }
    mag_squared.sqrt()
}

/// Vektörün büyüklüğünü, yönünü değiştirmeden 1.0'a değiştirin.
fn normalize(vector: &mut [f64; 3]) {
    let mag = magnitude(vector);
    for item in vector {
        *item /= mag;
    }
}

fn main() {
    println!("Birim vektörünün büyüklüğü: {}", magnitude(&[0.0, 1.0, 0.0]));

    let mut v = [1.0, 2.0, 9.0];
    println!("{v:?} vektörünün büyüklüğü: {}", magnitude(&v));
    normalize(&mut v);
    println!("Normalize edildikten sonra {v:?} vektörünün büyüklüğü: {}", magnitude(&v)
}
```

- `normalize` fonksiyonunda her bir elemanı değiştirmek için `*item /= mag` yapabildiğimize dikkat edin. Bunun sebebi, bir diziye değiştirilebilir (mutable) referans kullanarak döngü de dönmemizdir; bu da `for` döngüsünün her elemana değiştirilebilir (mutable) referanslar vermesini sağlar.
- It is also possible to take slice references here, e.g., `fn magnitude(vector: &[f64]) -> f64`. This makes the function more general, at the cost of a runtime length check.

# Chapter 10

# Kullanıcı Tanımlı Türler

Bu bölüm yaklaşık 1 saat sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| İsimli Yapılar (Named Structs) | 10 dakika |
| Demet Yapıları (Tuple Structs) | 10 dakika |
| Enumlar | 5 dakika |
| Tür Eş İsimleri (Type Aliases) | 2 dakika |
| Const | 10 dakika |
| Static | 5 dakika |
| Alıştırma: Asansör Olayları | 15 dakika |

## 10.1 İsimli Yapılar (Named Structs)

C ve C++ dillerindeki gibi Rust'ın da özel (kullanıcı tanımlı) yapılara desteği vardır:

```
struct Person {
    name: String,
    age: u8,
}

fn describe(person: &Person) {
    println!("{} {} yaşında", person.name, person.age);
}

fn main() {
    let mut peter = Person {
        name: String::from("Peter"),
        age: 27,
    };
    describe(&peter);

    peter.age = 28;
    describe(&peter);
```

```
    let name = String::from("Avery");
    let age = 39;
    let avery = Person { name, age };
    describe(&avery);
}
```

This slide should take about 10 minutes.

Önemli Noktalar:

- Yapılar C veya C++'daki gibi çalışır.
  - C++'daki gibi ve C'den farklı olarak, bir türü tanımlamak için typedef'e gerek yoktur.
  - C++'dan farklı olarak yapılar arasında kalıtım (inheritance) yoktur.
- Bu, insanlara farklı yapı türleri olduğunu duyurmak için iyi bir zaman olabilir.
  - Sıfır boyutlu (zero-sized) yapılar (örn. `struct Foo;`), bir özelliğin (trait) bazı türlere gerçekleştiriminde (implementing) kullanılabilir, ancak değer olarak kendisinde saklamak istenilen herhangi bir veri yoktur.
  - Sonraki slaytta alan adları (field names) önemli olmadığında kullanılan Demet (Tuple) yapıları tanıtılacaktır.
- Eğer zaten doğru adlara sahip değişkenleriniz varsa, o zaman yapıyı bir kısaltma kullanarak oluşturabilirsiniz.
- Yapı alanları (struct fields) varsayılan değerleri desteklemez. Varsayılan değerler, daha sonra ele alacağımız `Default` özelliği (trait) uygulanarak belirtilir.

### Daha Fazlasını Keşfedin

- Burada aynı zamanda yapı güncelleme sözdizimini (struct update syntax) de gösterebilirsiniz:

  ```
  let jackie = Person { name: String::from("Jackie"), ..avery };
  ```

- Bu, eski yapının (struct) alanlarının çoğunu açıkça tek tek yazmak zorunda kalmadan kopyalamamıza olanak tanır. Bu her zaman son öğe olmalıdır.
- Bu, çoğunlukla `Default` özelliği (trait) ile birlikte kullanılır. `Default` özelliği ile ilgili slaytta yapı güncelleme sözdizimini (struct update syntax) daha detaylı ele alacağız, bu yüzden öğrenciler sormadıkça burada bahsetmemize gerek yok.

## 10.2 Demet Yapıları (Tuple Structs)

Alan adları (field names) önemsizse demet (tuple) yapısını kullanabilirsiniz:

```
struct Point(i32, i32);

fn main() {
    let p = Point(17, 23);
    println!("({}, {})", p.0, p.1);
}
```

Bu genellikle tek alanlı sarmalayıcılar (single-field wrappers - newtype deseni olarak adlandırılır) için kullanılır:

```
struct PoundsOfForce(f64);
struct Newtons(f64);

fn compute_thruster_force() -> PoundsOfForce {
    todo!("NASA'daki bir roket bilim adamına sorun")
}

fn set_thruster_force(force: Newtons) {
    // ...
}

fn main() {
    let force = compute_thruster_force();
    set_thruster_force(force);
}
```

This slide should take about 10 minutes.

- Yeni tür (newtype) deseni, ilkel (primitive) bir türdeki değer hakkındaki ek bilgileri kodlamanın harika bir yoludur, örneğin:
  - Sayı bazı birimlerde ölçülür: Yukarıdaki örnekte `Newtons`.
  - Değer, oluşturulduğunda bir miktar doğrulamadan geçti, bu nedenle artık onu her kullanımda yeniden doğrulamanız gerekmiyor: `PhoneNumber(String)` veya `OddNumber(u32)`.
- The newtype pattern is covered extensively in the "Idiomatic Rust" module.
- Yeni türde desenindeki tek alana erişerek bir `Newtons` türüne `f64` değerinin nasıl ekleneceğini gösterin.
  - Rust genellikle bir nesnenin değerinin otomatik çözülmesini (automatic unwrapping) veya boole değerlerini tamsayı olarak kullanmak gibi açık olmayan şeylerden hoşlanmaz.
  - Operatör yüklemesi 3. günde (genelleştirmeler(generics)) tartışıldı.
- Bir demet (tuple) yapısının sıfır alanı varsa, `()` kısmı atlanabilir. Böylece ortaya sıfır boyutlu (zero-sized) bir tür (SBT/ZST) çıkar ve bu türün yalnızca bir değeri vardır (türün adı).
  - Bu, bazı davranışları uygulayan verisi olmayan türler için yaygındır (her zaman EOF döndüren bir `NullReader` hayal edin; okuma davranışını uygular ama veri içermez).
- Örnek, Mars İklim Yörünge Aracı hatasına incelikli bir göndermedir.

## 10.3 Enumlar

`enum` anahtar sözcüğü birkaç farklı varyanta sahip bir türün yaratılmasına olanak sağlar:

```
#[derive(Debug)]
enum Direction {
    Left,
    Right,
}

#[derive(Debug)]
enum PlayerMove {
```

```
    Pass,                         // Basit (Simple) varyant
    Run(Direction),               // Demet (Tuple) varyantı
    Teleport { x: u32, y: u32 }, // Yapı (Struct) varyantı
}

fn main() {
    let dir = Direction::Left;
    let player_move: PlayerMove = PlayerMove::Run(dir);
    println!("Sıradaki hamle: {player_move:?}");
}
```

This slide should take about 5 minutes.

Önemli Noktalar:

- Numaralandırmalar, bir dizi değeri tek bir tür altında toplamanıza olanak tanır.
- `Direction`, varyantları olan bir türdür. `Direction`'ın iki değeri vardır: `Direction::Left` ve `Direction::Right`.
- `PlayerMove` üç varyantı olan bir türdür. Rust, varyant ek yüküne (variant payloads) ek olarak, çalışma zamanında hangi varyantın `PlayerMove` değerinde olduğunu bilmesi için bir ayırıcı (discriminant) da depolar.
- Bu, yapıları ve numaralandırmaları karşılaştırmak için iyi bir zaman olabilir:
  - Her ikisinde de, alanları olmayan basit bir versiyona (birim yapı) veya farklı alan türlerine sahip bir versiyona (varyant ek yükleri) sahip olabilirsiniz.
  - Bir enum'ın farklı varyantlarını ayrı yapılarla bile uygulayabilirsiniz ancak o zaman da hepsi bir enum'da tanımlanmış olduğunda olacağı gibi aynı türde olmazlardı.
- Rust, ayırıcıyı (discriminant) depolamak için minimum alan kullanır.
  - Gerekirse, gereken en küçük boyuttaki bir tam sayıyı depolar
  - İzin verilen varyant değerleri tüm bit desenlerini kapsamıyorsa, ayırıcıyı kodlamak için geçersiz bit desenleri kullanır (”niş/oyuk/niche optimizasyonu”). Örneğin, `Option<&u8>`, `None` varyantı için bir tamsayıya gösterici veya `NULL` depolar.
  - Gerekirse ayırıcıyı kontrol edebilirsiniz (örneğin, C ile uyumluluk için):
    ```
    #[repr(u32)]
    enum Bar {
        A, // 0
        B = 10000,
        C, // 10001
    }

    fn main() {
        println!("A: {}", Bar::A as u32);
        println!("B: {}", Bar::B as u32);
        println!("C: {}", Bar::C as u32);
    }
    ```
    `repr` olmadan, ayırıcı tür 2 baytlık olur, çünkü 10001 2 bayta sığar.

## Daha Fazlasını Keşfedin

Rust, enum'ların daha az yer kaplamasını sağlamak için kullanabileceği çeşitli optimizasyonlara sahiptir.

- Null gösterici optimizasyonu: Bazı türler için, Rust `size_of::<T>()`'nin `size_of::<Option<T>>()`'ye eşit olduğunu garanti eder.

Bitsel gösterimin pratikte nasıl *görünebileceğini* göstermek istiyorsanız örnek kod aşağıdadır. Derleyicinin bu gösterimle (representation) ilgili hiçbir garanti sağlamadığını, dolayısıyla bunun tamamen emniyetsiz olduğunu belirtmek önemlidir.

```
use std::mem::transmute;

macro_rules! dbg_bits {
    ($e:expr, $bit_type:ty) => {
        println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_type>($e));
    };
}

fn main() {
    unsafe {
        println!("bool:");
        dbg_bits!(false, u8);
        dbg_bits!(true, u8);

        println!("Option<bool>:");
        dbg_bits!(None::<bool>, u8);
        dbg_bits!(Some(false), u8);
        dbg_bits!(Some(true), u8);

        println!("Option<Option<bool>>:");
        dbg_bits!(Some(Some(false)), u8);
        dbg_bits!(Some(Some(true)), u8);
        dbg_bits!(Some(None::<bool>), u8);
        dbg_bits!(None::<Option<bool>>, u8);

        println!("Option<&i32>:");
        dbg_bits!(None::<&i32>, usize);
        dbg_bits!(Some(&0i32), usize);
    }
}
```

## 10.4 Tür Eş İsimleri (Type Aliases)

Bir tür eş ismi, başka bir tür için bir isim oluşturur. Her iki tür de birbirinin yerine kullanılabilir.

```
enum CarryableConcreteItem {
    Left,
    Right,
}

type Item = CarryableConcreteItem;

// Eş isimler (aliases) uzun ve karmaşık türlerde daha kullanışlıdır:
use std::cell::RefCell;
use std::sync::{Arc, RwLock};
type PlayerInventory = RwLock<Vec<Arc<RefCell<Item>>>>;
```

This slide should take about 2 minutes.

- Bir yeni tür (newtype) deseni genellikle farklı bir tür oluşturduğu için daha iyi bir seçenektir. `type InventoryCount = usize` yerine `struct InventoryCount(usize)`'ı tercih edin.
- C programcıları bunun, `typedef`'e benzer olduğunu anlayacaklardır.

## 10.5 `const`

Sabitler (constants) derleme zamanında (compile-time) değerlendirilir ve değerleri kullanıldıkları her yerde satır içi (inline) olarak belirtilir:

```
const DIGEST_SIZE: usize = 3;
const FILL_VALUE: u8 = calculate_fill_value();

const fn calculate_fill_value() -> u8 {
    if DIGEST_SIZE < 10 { 42 } else { 13 }
}

fn compute_digest(text: &str) -> [u8; DIGEST_SIZE] {
    let mut digest = [FILL_VALUE; DIGEST_SIZE];
    for (idx, &b) in text.as_bytes().iter().enumerate() {
        digest[idx % DIGEST_SIZE] = digest[idx % DIGEST_SIZE].wrapping_add(b);
    }
    digest
}

fn main() {
    let digest = compute_digest("Hello");
    println!("digest: {digest:?}");
}
```

Rust RFC Book'a göre bunlar kullanım sırasında satır içidir (inline).

Sadece `const` olarak işaretlenen fonksiyonlar derleme zamanında çağrılarak `const` değerler üretilebilir. Ancak `const` fonksiyonlar çalışma zamanında da çağrılabilir.

This slide should take about 10 minutes.

- `const`'un C++'ın `constexpr`'ına anlamsal olarak benzer davrandığını belirtin

## 10.6 `static`

Statik değişkenler programın tüm yürütmesi (execution) boyunca yaşayacak ve bu nedenle taşınmayacaklar (not move):

```
static BANNER: &str = "RustOS 3.14'e Hoş Geldiniz";

fn main() {
    println!("{BANNER}");
}
```

Rust RFC Kitabında belirtildiği gibi, statik kullanımı satır içi (inline) özelliği vermez ve gerçek bir ilişkili bellek konumuna sahiptir. Bu, emniyetsiz (unsafe) ve gömülü (embedded) kod için yararlıdır ve değişken, program yürütmesinin tamamı boyunca yaşar. Global kapsamlı (globally-scoped) bir değerin nesne kimliğine (object identity) ihtiyaç duyması için bir nedeni olmadığında, genellikle `const` tercih edilir.

This slide should take about 5 minutes.

- `statik`, C++'daki değiştirilebilir olan global değişkenlere benzer.
- `static`, `Mutex<T>` gibi iç değişebilirliğe (interior mutability) sahip türlerin gerektirdiği şekilde bellekteki bir adres ve durumu nesneye kimlik olarak sağlar.

## Daha Fazlasını Keşfedin

Çünkü `static` değişkenlere herhangi bir iş parçacığından (thread) erişilebildiğinden, bunların `Sync` olması gerekir. İç değişebilirlik (Interior mutability) `Mutex`, atomik veya benzer yöntemlerle mümkündür.

İlk kullanımda ilklendirmeyi (initialization) desteklemek için `static` içinde `OnceLock` kullanmak yaygındır. `OnceCell`, `Sync` olmadığı için bu bağlamda kullanılamaz.

İş parçacığı için yerel (thread-local) veriler `std::thread_local` makrosu ile oluşturulabilir.

## 10.7 Alıştırma: Asansör Olayları

Bir asansör kontrol sistemindeki bir olayı temsil etmek için bir veri yapısı oluşturacağız. Çeşitli olayları yapılandırmak için türleri ve fonksiyonları tanımlamak size kalmış. Türlerin `{:?}` ile biçimlendirilmesine (format) izin vermek için `#[derive(Debug)]` kullanın.

Bu alıştırma yalnızca, `main`'in hatasız çalışması için veri yapıları oluşturmayı ve doldurmayı gerektirir. Kursun bir sonraki kısmı bu yapılardan veri almayı kapsayacaktır.

```
#![allow(dead_code)]

#[derive(Debug)]
/// Asansör sisteminde kontrolörün tepki vermesi gereken bir olay.
enum Event {
    // TODO: gerekli değişkeleri (variant) ekle
}

/// Bir seyahat yönü.

#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// Kabin verilen kata ulaşmış durumda.
fn car_arrived(floor: i32) -> Event {
    todo!()
}
```

```
/// Kabin kapıları açılmış durumda.
fn car_door_opened() -> Event {
    todo!()
}

/// Kabin kapıları kapanmış durumda.
fn car_door_closed() -> Event {
    todo!()
}

/// Verilen kattaki asansör lobisinde yön düğmesine basıldı.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    todo!()
}

/// Asansör kabininde bir kat düğmesine basıldı.
fn car_floor_button_pressed(floor: i32) -> Event {
    todo!()
}

fn main() {
    println!(
        "Zemin kattaki bir yolcu yukarı düğmesine basmış durumda: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("Kabin zemin kata ulaşmış durumda: {:?}", car_arrived(0));
    println!("Kabin kapısı açıldı: {:?}", car_door_opened());
    println!(
        "Bir yolcu 3. katın düğmesine basmış durumda: {:?}",
        car_floor_button_pressed(3)
    );
    println!("Kabin kapısı kapandı: {:?}", car_door_closed());
    println!("Kabin 3. kata ulaşmış durumda: {:?}", car_arrived(3));
}
```

This slide and its sub-slides should take about 15 minutes.

- Eğer öğrenciler alıştırmanın en üstündeki `#![allow(dead_code)]` satırını sorarlarsa, bunun nedeni `Event` türüyle yaptığımız tek şeyin onu ekrana yazdırmak olmasıdır. Derleyicinin ölü kodu (dead code) kontrol etme biçimindeki bir ayrıntı nedeniyle, bu durum kodun kullanılmadığını sanmasına yol açar. Öğrenciler bu satırı bu alıştırma özelinde görmezden gelebilirler.

### 10.7.1 Çözüm

```
#![allow(dead_code)]

#[derive(Debug)]
/// Asansör sisteminde kontrolörün tepki vermesi gereken bir olay.
enum Event {
    /// Bir düğmeye basıldı.
```

```
    ButtonPressed(Button),

    /// Kabin verilen kata ulaşmış durumda.
    CarArrived(Floor),

    /// Kabin kapıları açılmış durumda.
    CarDoorOpened,

    /// Kabin kapıları kapanmış durumda.
    CarDoorClosed,
}

/// Bir kat tamsayı olarak temsil edilir.
type Floor = i32;

/// Bir seyahat yönü.

#[derive(Debug)]
enum Direction {
    Up,
    Down,
}

/// Kullanıcının erişebileceği bir düğme.
#[derive(Debug)]
enum Button {
    /// Verilem katta asansör lobisinde bulunan bir düğme.
    LobbyCall(Direction, Floor),

    /// Kabinin içindeki bir kat düğmesi.
    CarFloor(Floor),
}

/// Kabin verilen kata ulaşmış durumda.
fn car_arrived(floor: i32) -> Event {
    Event::CarArrived(floor)
}

/// Kabin kapıları açılmış durumda.
fn car_door_opened() -> Event {
    Event::CarDoorOpened
}

/// Kabin kapıları kapanmış durumda.
fn car_door_closed() -> Event {
    Event::CarDoorClosed
}

/// Verilen kattaki asansör lobisinde yön düğmesine basıldı.
fn lobby_call_button_pressed(floor: i32, dir: Direction) -> Event {
    Event::ButtonPressed(Button::LobbyCall(dir, floor))
```

```rust
}

/// Asansör kabininde bir kat düğmesine basıldı.
fn car_floor_button_pressed(floor: i32) -> Event {
    Event::ButtonPressed(Button::CarFloor(floor))
}

fn main() {
    println!(
        "Zemin kattaki bir yolcu yukarı düğmesine basmış durumda: {:?}",
        lobby_call_button_pressed(0, Direction::Up)
    );
    println!("Kabin zemin kata ulaşmış durumda: {:?}", car_arrived(0));
    println!("Kabin kapısı açıldı: {:?}", car_door_opened());
    println!(
        "Bir yolcu 3. katın düğmesine basmış durumda: {:?}",
        car_floor_button_pressed(3)
    );
    println!("Kabin kapısı kapandı: {:?}", car_door_closed());
    println!("Kabin 3. kata ulaşmış durumda: {:?}", car_arrived(3));
}
```

# Part III

# Gün 2: Sabah

# Chapter 11

# 2. Gün'e Hoş Geldiniz

Rust'ı artık yeterince gördüğümüze göre, bugün Rust'ın tür sistemine odaklanacağız:

- Desen eşleştirme (pattern matching): yapılardan veri çıkarma.
- Metotlar: fonksiyonları türlerle ilişkilendirme.
- Özellikler (traits): birden fazla tür tarafından paylaşılan davranışlar.
- Genelleştirmeler (Generics): türleri diğer türler üzerinden parametrelendirme.
- Standart kütüphane türleri ve özellikleri (traits): Rust'ın zengin standart kütüphanesinde bir tur.
- Çevreleyiciler (closures): veriyle birlikte fonksiyon göstericileri.

## Zamanlama (Schedule)

Bu oturum 10 dakikalık aralar dahil yaklaşık 2 saat 45 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 3 dakika |
| Desen Eşleştirme | 50 dakika |
| Metotlar ve Özellikler (Traits) | 45 dakika |
| Genelleştirmeler (Generics) | 45 dakika |

# Chapter 12

# Desen Eşleştirme

Bu bölüm yaklaşık 50 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Reddedilemez Desenler (Irrefutable Patterns) | 5 dakika |
| Değerleri Eşleştirmek | 10 dakika |
| Yapıların Çözümlenmesi (Destructuring Structs) | 4 dakika |
| Enum'ların Çözümlenmesi (Destructuring) | 4 dakika |
| Let'li Kontrol Akışı | 10 dakika |
| Alıştırma: İfade Değerlendirmesi | 15 dakika |

## 12.1 Reddedilemez Desenler (Irrefutable Patterns)

1. günde, desenlerin bileşik değerlerini (compound values) *çözümlemek (destructure)* için nasıl kullanılabileceğini kısaca görmüştük. Şimdi bunu gözden geçirelim ve desenlerin ifade edebileceği birkaç diğer şey hakkında konuşalım:

```
fn takes_tuple(tuple: (char, i32, bool)) {
    let a = tuple.0;
    let b = tuple.1;
    let c = tuple.2;

    // Bu, yukarıdakiyle aynı şeyi yapar.
    let (a, b, c) = tuple;

    // İlk elemanı yoksay, sadece ikinci ve üçüncüyü bağla.
    let (_, b, c) = tuple;

    // Son eleman hariç her şeyi yoksay.
    let (.., c) = tuple;
}

fn main() {
```

```
    takes_tuple(('a', 777, true));
}
```

This slide should take about 5 minutes.

- Gösterilen desenlerin tümü *reddedilemezdir (irrefutable)*, yani her zaman sağ taraftaki değerle eşleşeceklerdir.
- Desenler, reddedilemez (irrefutable) desenler de dahil olmak üzere türe özgüdür (type-specific). Demete (tuple) bir eleman eklemeyi veya çıkarmayı deneyin ve ortaya çıkan derleyici hatalarına bakın.
- Değişken adları her zaman eşleşen ve eşleştikleri değeri bu isimle yeni bir değişkene bağlayan (bind) desenlerdir.
- _ her zaman herhangi bir değerle eşleşen ve eşleşen değeri atan (discarding) bir desendir.
- `..` aynı anda birden fazla değeri yok saymanıza olanak tanır.

### Daha Fazlasını Keşfedin

- Ayrıca, bir demetin (tuple) ortadaki elemanlarını yok saymak gibi `..`'nın daha gelişmiş kullanımlarını da gösterebilirsiniz.

  ```
  fn takes_tuple(tuple: (char, i32, bool, u8)) {
      let (first, .., last) = tuple;
  }
  ```

- Bu desenlerin tümü dizilerle (arrays) de çalışır:

  ```
  fn takes_array(array: [u8; 5]) {
      let [first, .., last] = array;
  }
  ```

## 12.2 Değerleri Eşleştirmek

`match` anahtar kelimesi, bir değeri bir veya daha fazla *desene (pattern)* göre eşleştirmenizi sağlar. Desenler, C ve C++'daki `switch`'e benzer şekilde basit değerler olabilir, ancak daha karmaşık koşulları ifade etmek için de kullanılabilirler:

```
#[rustfmt::skip]
fn main() {
    let input = 'x';
    match input {
        'q'                       => println!("Çıkılıyor"),
        'a' | 's' | 'w' | 'd'     => println!("Etrafta hareket ediliyor"),
        '0'..='9'                 => println!("Sayı girişi"),
        key if key.is_lowercase() => println!("Küçük harf: {key}"),
        _                         => println!("Başka bir şey"),
    }
}
```

Desendeki bir değişken (bu örnekte `key`) eşleşme kolu (match arm) içinde kullanılabilecek bir bağlama (binding) oluşturacaktır. Bunu bir sonraki slaytta daha ayrıntılı öğreneceğiz.

Bir eşleşme filtresi (match guard), kolun yalnızca koşul doğruysa eşleşmesine neden olur. Koşul yanlışsa, eşleşme diğer durumları kontrol etmeye devam edecektir.

This slide should take about 10 minutes.

Önemli Noktalar:

- Bir desendeyken bazı özel karakterlerin nasıl kullanıldığına dikkat edin
  - `|` bir `veya (or)` olarak
  - `..` gerektiği kadar genişleyebilir
  - `1..=5` kapsayıcı bir aralığı (inclusive range) temsil eder
  - `_` bir joker karakterdir
- Ayrı bir sözdizimi özelliği olarak eşleşme filtreleri (match guards), tek başına desenlerin izin verdiğinden daha karmaşık fikirleri kısaca ifade etmek istediğimizde önemli ve gereklidir.
- Match guards are different from `if` expressions after the `=>`. An `if` expression is evaluated after the match arm is selected. Failing the `if` condition inside of that block won't result in other arms of the original `match` expression being considered. In the following example, the wildcard pattern `_ =>` is never even attempted.

```
#[rustfmt::skip]
fn main() {
    let input = 'a';
    match input {
        key if key.is_uppercase() => println!("Uppercase"),
        key => if input == 'q' { println!("Çıkılıyor") },
        _   => println!("Bug: this is never printed"),
    }
}
```

- Filtrede tanımlanan koşul, bir `|` içeren bir desendeki her ifade için geçerlidir.
- Bir eşleşme kolunda (match arm) mevcut bir değişkeni koşul olarak kullanamayacağınızı unutmayın, çünkü bunun yerine bir değişken adı deseni (variable name pattern) olarak yorumlanacaktır, bu da mevcut olanı gölgeleyecek (shadow) yeni bir değişken oluşturur. Örneğin:

  ```
  let expected = 5;
  match 123 {
      expected => println!("Beklenen değer 5, gerçek değer {expected}"),
      _ => println!("Değer başka bir şeydi"),
  }
  ```

  Burada 123 sayısıyla eşleştirmeye çalışıyoruz, ilk durumun değerin 5 olup olmadığını kontrol etmesini istiyoruz. Doğal beklenti, değer 5 olmadığı için ilk durumun eşleşmeyeceğidir, ancak bunun yerine bu, her zaman eşleşen bir değişken deseni olarak yorumlanır, yani her zaman ilk dal alınacaktır. Bunun yerine bir sabit (constant) kullanılırsa bu beklendiği gibi çalışacaktır.

## Daha Fazlasını Keşfedin

- Öğrencilere gösterebileceğiniz başka bir desen sözdizimi parçası, bir desenin bir bölümünü bir değişkene bağlayan `@` sözdizimidir. Örneğin:

  ```
  let opt = Some(123);
  match opt {
      outer @ Some(inner) => {
          println!("dış: {outer:?}, iç: {inner}");
      }
      None => {}
  }
  ```

  Bu örnekte `inner`, çözümleme (destructuring) yoluyla `Option`'dan çektiği `123` değerine sahiptir, `outer` ise tüm `Some(inner)` ifadesini yakalar, bu nedenle tam `Option::Some(123)`'ü içerir. Bu nadiren kullanılır ancak daha karmaşık desenlerde faydalı olabilir.

## 12.3 Yapılar (Structs)

Demetler (tuples) gibi, yapılar (structs) da eşleştirme (matching) yoluyla çözümlenebilir (destructured):

```
struct Foo {
    x: (u32, u32),
    y: u32,
}

#[rustfmt::skip]
fn main() {
    let foo = Foo { x: (1, 2), y: 3 };
    match foo {
        Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
        Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
        Foo { y, .. }        => println!("y = {y}, diğer alanlar yoksayıldı"),
    }
}
```

This slide should take about 4 minutes.

- `foo`'daki değişmez (literal) değerleri diğer desenlerle eşleşecek şekilde değiştirin.
- `Foo`'ya yeni bir alan ekleyin ve desende gerektiği gibi değişiklikler yapın.

### Daha Fazlasını Keşfedin

- `match &foo`'yu deneyin ve yakalanan (captures) değerlerin türünü kontrol edin. Desen sözdizimi aynı kalır, ancak yakalananlar paylaşılan referanslar (shared references) haline gelir. Bu eşleşme ergonomisidir (match ergonomics) ve bir enum üzerinde metotlar uygularken `match self` ile genellikle kullanışlıdır.
  - Aynı etki `match &mut foo` ile de meydana gelir: yakalanan (captures) değerler özel (exclusive) referanslar haline gelir.

- The distinction between a capture and a constant expression can be hard to spot. Try changing the 2 in the first arm to a variable, and see that it subtly doesn't work. Change it to a `const` and see it working again.

## 12.4 Enumlar

Demetler (tuples) gibi, enum'lar da eşleştirme yoluyla çözümlenebilir (destructured):

Desenler, değişkenleri değerlerinizin parçalarına bağlamak için de kullanılabilir. Türlerinizin yapısını bu şekilde inceleyebilirsiniz. Basit bir `enum` türüyle başlayalım:

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("{n} iki eşit parçaya bölünemez"))
    }
}

fn main() {
    let n = 100;
    match divide_in_two(n) {
        Result::Ok(half) => println!("{n} ikiye bölündüğünde {half}"),
        Result::Err(msg) => println!("üzgünüm, bir hata oluştu: {msg}"),
    }
}
```

Burada `Result` değerini *çözümlemek (destructure)* için kolları (arm) kullandık. İlk kolda, `half`, `Ok` varyantının içindeki değere bağlanır. İkinci kolda, `msg` hata mesajına bağlanır.

This slide should take about 4 minutes.

- `if`/`else` ifadesi, daha sonra bir `match` ile açılan (unpacked) bir enum geri döndürüyor.
- Enum tanımına üçüncü bir varyant eklemeyi ve kodu çalıştırırken hataları göstermeyi deneyebilirsiniz. Kodunuzun artık kapsayıcı olmadığı (inexhaustive) yerlere ve derleyicinin size nasıl ipuçları vermeye çalıştığına dikkat çekin.
- Enum varyantlarındaki değerlere yalnızca desen eşleştirmesi yapıldıktan sonra erişilebilir.
- Aramanın kapsayıcı olmadığında (inexhaustive) ne olduğunu gösterin. Rust derleyicisinin tüm durumların ele alındığını onaylayarak sağladığı avantaja dikkat edin.
- Enum tanımına ve `match`'e bir tane ekleyerek yapı stili (struct-style) bir varyant için sözdizimini (syntax) gösterin. Bunun sözdizimsel olarak bir yapı (struct) üzerinde eşleştirmeye (matching) nasıl benzediğine dikkat çekin.

## 12.5 Let'li Kontrol Akışı

Rust'ın diğer dillerden farklı olan birkaç kontrol akışı yapısı vardır. Bunlar desen eşleştirme (pattern matching) için kullanılır:

- `if let` ifadeleri
- `while let` ifadeleri
- `let else` ifadeleri

### 12.5.1 `if let` İfadeleri

`if let` ifadesi, bir değerin bir desenle eşleşip eşleşmediğine bağlı olarak farklı kod çalıştırmanıza olanak tanır:

```
use std::time::Duration;

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("{duration:?} kadar uyunuldu");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

- `match`'in aksine, `if let` tüm dalları kapsamak zorunda değildir. Bu onu `match`'den daha özlü (concise) yapabilir.
- `Option` ile çalışırken yaygın bir kullanım, `Some` değerlerini işlemektir.
- `match`'in aksine, `if let` desen eşleştirme için filtre koşul grubu (guard clauses) desteklemez.
- Bir `else` durumuyla, bu bir ifade (expression) olarak kullanılabilir.

### 12.5.2 `while let` Deyimleri

`if let`'te olduğu gibi, bir değeri tekrar tekrar bir desene göre test eden bir `while let` çeşidi vardır:

```
fn main() {
    let mut name = String::from("Comprehensive Rust 🦀");
    while let Some(c) = name.pop() {
        dbg!(c);
    }
    // (Bir dizeyi tersine çevirmenin daha verimli yolları var!)
}
```

Burada `String::pop` dize boşalana kadar `Some(c)` geri döndürür, ardından `None` geri döndürür. `while let` tüm öğeler arasında adımlamaya (iterating) devam etmemizi sağlar.

- `while let` döngüsünün, değer desenle eşleştiği sürece devam edeceğine dikkat çekin.
- `while let` döngüsünü, `name.pop()` için açılacak bir değer olmadığında kesintiye uğrayan bir if ifadesiyle sonsuz bir döngü olarak yeniden yazabilirsiniz. `while let` yukarıdaki senaryo için sözdizimsel kolaylık sağlar.
- Bu form bir ifade (expression) olarak kullanılamaz, çünkü koşul yanlışsa hiçbir değeri olmayabilir.

### 12.5.3 `let else` Deyimleri

Bir deseni eşleştirip fonksiyondan geri döndürmenin yaygın durumu için `let else` kullanın. "else" durumu farklı olmalıdır (diverge) (`return`, `break` veya `panic` - bloğun sonundan çıkmak dışında her şey).

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let s = if let Some(s) = maybe_string {
        s
    } else {
        return Err(String::from("None alındı"));
    };

    let first_byte_char = if let Some(first) = s.chars().next() {
        first
    } else {
        return Err(String::from("boş dize alındı"));
    };

    let digit = if let Some(digit) = first_byte_char.to_digit(16) {
        digit
    } else {
        return Err(String::from("onaltılık bir basamak değil"));
    };

    Ok(digit)
}

fn main() {
    println!("sonuç: {:?}", hex_or_die_trying(Some(String::from("foo"))));
}
```

The rewritten version is:

```
fn hex_or_die_trying(maybe_string: Option<String>) -> Result<u32, String> {
    let Some(s) = maybe_string else {
        return Err(String::from("None alındı"));
    };

    let Some(first_byte_char) = s.chars().next() else {
        return Err(String::from("boş dize alındı"));
    };

    let Some(digit) = first_byte_char.to_digit(16) else {
        return Err(String::from("onaltılık bir basamak değil"));
```

```
    };

    Ok(digit)
}
```

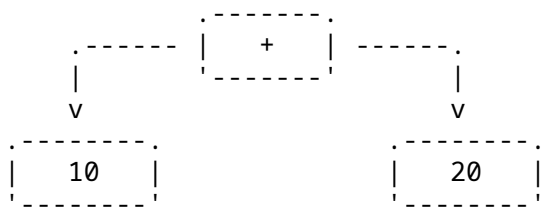
**Daha Fazlasını Keşfedin**

- Bu erken geri döndürme tabanlı kontrol akışı (early return-based control flow), Rust hata işleme (error handling) kodunda yaygındır; burada bir `Result`'tan bir değer almaya çalışırsınız ve `Result` `Err` ise bir hata geri döndürürsünüz.
- Öğrenciler sorarsa, gerçek hata işleme kodunun `?` ile nasıl yazılacağını da gösterebilirsiniz.

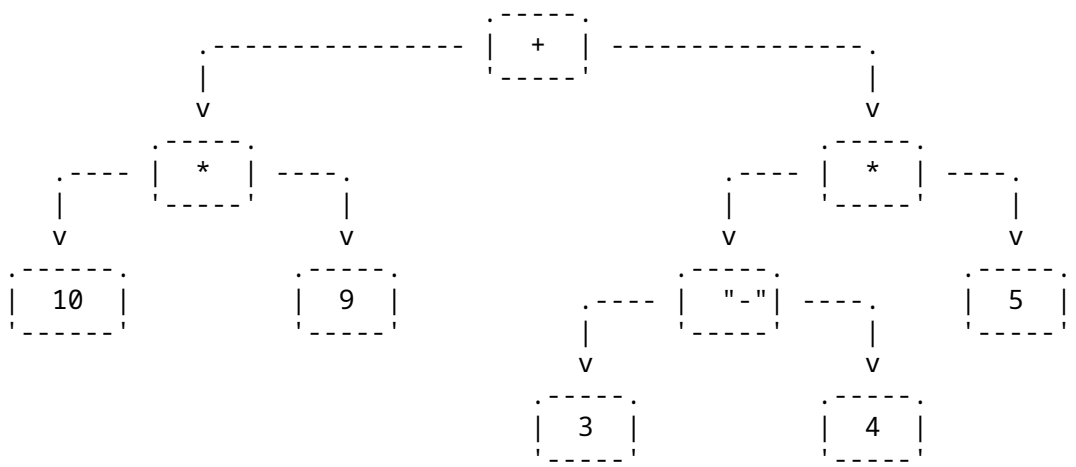
## 12.6 Alıştırma: İfade Değerlendirmesi

Aritmetik ifadeler için basit bir özyinelemeli değerlendirici (recursive evaluator) yazalım.

Küçük bir aritmetik ifadenin bir örneği, `30` olarak değerlendirilen `10 + 20` olabilir. İfadeyi bir ağaç olarak temsil edebiliriz:




Daha büyük ve daha karmaşık bir ifade, `85` olarak değerlendirilen `(10 * 9) + ((3 - 4) * 5)` olacaktır. Bunu çok daha büyük bir ağaç olarak temsil ediyoruz:




Kodda, ağacı iki türle temsil (represent) edeceğiz:

```
/// İki alt ifade (subexpression) üzerinde gerçekleştirilecek bir işlem.
#[derive(Debug)]
enum Operation {
    Add,
```

```
    Sub,
    Mul,
    Div,
}

/// Ağaç şeklinde bir ifade (expression).
#[derive(Debug)]
enum Expression {
    /// İki alt ifade üzerinde bir işlem.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Bir değişmez (literal) değer
    Value(i64),
}
```

Box türü burada bir akıllı göstericidir (smart pointer) ve kursun ilerleyen bölümlerinde ayrıntılı olarak ele alınacaktır. Bir ifade, testlerde görüldüğü gibi `Box::new` ile "kutulanabilir (boxed)". Kutulanmış bir ifadeyi değerlendirmek için, onu "kutudan çıkarmak (unbox)" için referans kaldırma (deref) operatörünü (`*`) kullanın: `eval(*boxed_expr)`.

Kodu Rust deneme alanına (playground) kopyalayıp yapıştırın ve `eval`'i uygulamaya başlayın. Son ürün testleri geçmelidir. `todo!()` kullanmak ve testleri birer birer geçmek yardımcı olabilir. Bir testi geçici olarak `#[ignore]` ile de atlayabilirsiniz:

```
#[test]
#[ignore]
fn test_value() { .. }
```

```
/// İki alt ifade (subexpression) üzerinde gerçekleştirilecek bir işlem.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Ağaç şeklinde bir ifade (expression).
#[derive(Debug)]
enum Expression {
    /// İki alt ifade üzerinde bir işlem.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Bir değişmez (literal) değer
    Value(i64),
}

fn eval(e: Expression) -> i64 {
    todo!()
}

#[test]
fn test_value() {
```

```
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        30
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        85
    );
}

#[test]
fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
```

```
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
}

#[test]
fn test_div() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(2)),
        }),
        5
    )
}
```

### 12.6.1 Çözüm

```
/// İki alt ifade (subexpression) üzerinde gerçekleştirilecek bir işlem.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Ağaç şeklinde bir ifade (expression).
#[derive(Debug)]
enum Expression {
    /// İki alt ifade üzerinde bir işlem.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Bir değişmez (literal) değer
    Value(i64),
}
```

```
fn eval(e: Expression) -> i64 {
    match e {
        Expression::Op { op, left, right } => {
            let left = eval(*left);
            let right = eval(*right);
            match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => left / right,
            }
        }
        Expression::Value(v) => v,
    }
}

#[test]
fn test_value() {
    assert_eq!(eval(Expression::Value(19)), 19);
}

#[test]
fn test_sum() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(20)),
        }),
        30
    );
}

#[test]
fn test_recursion() {
    let term1 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Value(10)),
        right: Box::new(Expression::Value(9)),
    };
    let term2 = Expression::Op {
        op: Operation::Mul,
        left: Box::new(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(3)),
            right: Box::new(Expression::Value(4)),
        }),
        right: Box::new(Expression::Value(5)),
    };
    assert_eq!(
        eval(Expression::Op {
```

```
            op: Operation::Add,
            left: Box::new(term1),
            right: Box::new(term2),
        }),
        85
    );
}

#[test]
fn test_zeros() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Add,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Mul,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(0)),
            right: Box::new(Expression::Value(0))
        }),
        0
    );
}

#[test]
fn test_div() {
    assert_eq!(
        eval(Expression::Op {
            op: Operation::Div,
            left: Box::new(Expression::Value(10)),
            right: Box::new(Expression::Value(2)),
        }),
        5
    )
}
```

# Chapter 13

# Metotlar ve Özellikler (Traits)

Bu bölüm yaklaşık 45 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Metotlar | 10 dakika |
| Özellikler (Traits) | 15 dakika |
| Türetme (Deriving) | 3 dakika |
| Alıştırma: Genel Kaydedici | 15 dakika |

## 13.1 Metotlar

Rust, fonksiyonları yeni türlerinizle ilişkilendirmenize (associate) olanak tanır. Bunu bir `impl` bloğu ile yaparsınız:

```
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // Alıcı (receiver) yok, statik bir metot
    fn new(name: &str) -> Self {
        Self { name: String::from(name), laps: Vec::new() }
    }

    // self'e özel (exclusive) ödünç alınmış (borrowed) okuma-yazma erişimi
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }

    // self'e paylaşılan ve salt okunur (read-only) ödünç alınmış (borrowed) erişim
    fn print_laps(&self) {
        println!("{} tur kaydedildi, {} için:", self.laps.len(), self.name);
```

```
        for (idx, lap) in self.laps.iter().enumerate() {
            println!("Tur {idx}: {lap} sn");
        }
    }

    // self'in özel (exclusive) sahipliği (daha sonra ele alınacak)
    fn finish(self) {
        let total: i32 = self.laps.iter().sum();
        println!("Yarış {} bitti, toplam tur süresi: {}", self.name, total);
    }
}

fn main() {
    let mut race = CarRace::new("Türkiye Grand Prix");
    race.add_lap(70);
    race.add_lap(68);
    race.print_laps();
    race.add_lap(71);
    race.print_laps();
    race.finish();
    // race.add_lap(42);
}
```

`self` argümanları ”alıcıyı (receiver)” ( metodun üzerinde işlem yaptığı nesne) belirtir. Bir metot için birkaç yaygın alıcı vardır:

- `&self`: nesneyi çağırandan paylaşılan ve değiştirilemez (shared and immutable) bir referans kullanarak ödünç alır (borrows). Nesne daha sonra tekrar kullanılabilir.
- `&mut self`: nesneyi çağırandan benzersiz ve değiştirilebilir (unique and mutable) bir referans kullanarak ödünç alır (borrows). Nesne daha sonra tekrar kullanılabilir.
- `self`: nesnenin sahipliğini (ownership) alır ve onu çağırandan uzağa taşır. Metot, nesnenin sahibi olur. Sahipliği açıkça (explicitly) aktarılmadığı sürece, metot geri döndüğünde nesne düşürülür (deallocated). Tam sahiplik otomatik olarak değiştirilebilirlik (mutability) anlamına gelmez.
- `mut self`: yukarıdakiyle aynı, ancak metot nesneyi değiştirebilir.
- Alıcı (receiver) yok: bu, yapı (struct) üzerinde statik bir metot haline gelir. Genellikle geleneksel olarak `new` olarak adlandırılan yapıcılar (constructors) oluşturmak için kullanılır.

This slide should take about 8 minutes.

Önemli Noktalar:

- Metotları fonksiyonlarla karşılaştırarak tanıtmak faydalı olabilir.
  - Metotlar bir türün (bir yapı veya enum gibi) bir örneği üzerinden çağrılır; ilk parametre, örneği (instance) `self` olarak temsil (represents) eder.
  - Geliştiriciler, metot alıcı sözdiziminden (method receiver syntax) yararlanmak ve onları daha düzenli tutmaya yardımcı olmak için metotları kullanmayı seçebilirler. Metotları kullanarak tüm gerçekleştirim (implementation) kodunu tek bir öngörülebilir yerde tutabiliriz.
  - Metotların, alıcıyı (receiver) açıkça ileterek ilişkili fonksiyonlar gibi de çağrılabileceğini unutmayın, örn. `CarRace::add_lap(&mut race, 20)`.
- Bir metot alıcısı (receiver) olan `self` anahtar kelimesinin kullanımına dikkat çekin.

- Bunun `self: Self` için kısaltılmış bir terim olduğunu gösterin ve belki de yapı (struct) adının nasıl kullanılabileceğini de gösterin.
- `Self`'in, `impl` bloğunun içinde olduğu tür için bir tür takma adı (type alias) olduğunu ve bloğun başka yerlerinde de kullanılabileceğini açıklayın.
- `self`'in diğer yapılar gibi nasıl kullanıldığına ve bireysel alanlara (individual fields) atıfta bulunmak için nokta (.) notasyonunun nasıl kullanılabileceğine dikkat edin.
- `finish`'i iki kez çalıştırmayı deneyerek `&self`'in `self`'ten nasıl farklı olduğunu göstermek için bu iyi bir zaman olabilir.
- `self` üzerindeki varyantların ötesinde, `Box<Self>` gibi alıcı (receiver) türleri olmasına izin verilen özel sarmalayıcı türleri de vardır.

## 13.2 Özellikler (Traits)

Rust, özellikler (traits) ile türler üzerinde soyutlama yapmanıza olanak tanır. Arayüzlere (interfaces) benzerler:

```
trait Pet {
    /// Bu evcil hayvandan bir cümle geri döndürün.
    fn talk(&self) -> String;

    /// Bu evcil hayvanı selamlayan bir dizeyi terminale yazdırın.
    fn greet(&self);
}
```

This slide and its sub-slides should take about 15 minutes.

- Bir özellik (trait), türlerin özelliği gerçekleştirmek (implement) için sahip olması gereken bir dizi metodu tanımlar.
- Bir sonraki ”Genelleştirmeler (Generics)” bölümünde, bir özelliği (trait) gerçekleştiren tüm türler üzerinde jenerik olan fonksiyonelliğin nasıl oluşturulacağını göreceğiz.

### 13.2.1 Özelliklerin (Traits) Gerçekleştirimi

```
trait Pet {
    fn talk(&self) -> String;

    fn greet(&self) {
        println!("Ah sen ne şirinsin! Adın ne? {}", self.talk());
    }
}

struct Dog {
    name: String,
    age: i8,
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Hav, benim adım {}!", self.name)
    }
}
```

```
fn main() {
    let fido = Dog { name: String::from("Fido"), age: 5 };
    dbg!(fido.talk());
    fido.greet();
}
```

- Type için `Trait`'i gerçekleştirmek (implement) üzere bir `impl Trait for Type { .. }` bloğu kullanırsınız.
- Go arayüzlerinin (interfaces) aksine, sadece eşleşen metotlara sahip olmak yeterli değildir: `talk()` metoduna sahip bir `Cat` türü, bir `impl Pet` bloğunda olmadığı sürece `Pet`'i otomatik olarak karşılamaz.
- Özellikler (traits), bazı metotların varsayılan gerçekleştirmelerini (implementations) sağlayabilir. Varsayılan gerçekleştirmeler, özelliğin tüm metotlarına dayanabilir. Bu durumda, `greet` sağlanır ve `talk`'a dayanır.
- Belirli bir tür için birden fazla `impl` bloğuna izin verilir. Bu hem doğal `impl` bloklarını hem de özellik (trait) `impl` bloklarını içerir. Aynı şekilde belirli bir tür için birden fazla özellik gerçekleştirilebilir (ve genellikle türler birçok özelliği gerçekleştirir!). `impl` blokları birden fazla modüle/dosyaya bile yayılabilir.

### 13.2.2 Üstözellikler (Supertraits)

Bir özellik (trait), onu gerçekleştiren türlerin *üstözellikler (supertraits)* olarak adlandırılan diğer özellikleri de gerçekleştirmesini gerektirebilir. Burada, `Pet`'i gerçekleştiren herhangi bir tür `Animal`'ı da gerçekleştirmelidir.

```
trait Animal {
    fn leg_count(&self) -> u32;
}

trait Pet: Animal {
    fn name(&self) -> String;
}

struct Dog(String);

impl Animal for Dog {
    fn leg_count(&self) -> u32 {
        4
    }
}

impl Pet for Dog {
    fn name(&self) -> String {
        self.0.clone()
    }
}

fn main() {
    let puppy = Dog(String::from("Rex"));
```

```
    println!("{}, {} bacağa sahip", puppy.name(), puppy.leg_count());
}
```

Bu bazen ”özellik kalıtımı (trait inheritance)” olarak adlandırılır ancak öğrencilerin bunun nesne yönelimli kalıtım (OO inheritance) gibi davranmasını beklememesi gerekir. Bu sadece bir özelliğin gerçekleştirmeleri üzerinde ek bir gereksinimi (requirement) belirtir.

### 13.2.3 İlişkili Türler

İlişkili türler (associated types), özellik (trait) gerçekleştirimi tarafından sağlanan yer tutucu (placeholder) türlerdir.

```
#[derive(Debug)]
struct Meters(i32);
#[derive(Debug)]
struct MetersSquared(i32);

trait Multiply {
    type Output;
    fn multiply(&self, other: &Self) -> Self::Output;
}

impl Multiply for Meters {
    type Output = MetersSquared;
    fn multiply(&self, other: &Self) -> Self::Output {
        MetersSquared(self.0 * other.0)
    }
}

fn main() {
    println!("{:?}", Meters(10).multiply(&Meters(20)));
}
```

- İlişkili türler (associated types) bazen ”çıktı türleri (output types)” olarak da adlandırılır. Anahtar gözlem, bu türü çağıranın değil, gerçekleştirenin (implementer) seçmesidir.
- Aritmetik operatörler ve `Iterator` da dahil olmak üzere birçok standart kütüphane özelliğinin (traits) ilişkili türleri vardır.

## 13.3 Türetme (Deriving)

Desteklenen özellikler (traits), özel türleriniz için aşağıdaki gibi otomatik olarak gerçekleştirilebilir (implemented):

```
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
```

```
    let p1 = Player::default(); // Default özelliği (trait) `default` yapıcısını (const
    let mut p2 = p1.clone(); // Clone özelliği (trait) `clone` metodunu ekler.
    p2.name = String::from("EldurScrollz");
    // Debug özelliği (trait) `{:?}` ile yazdırma desteği ekler.
    println!("{p1:?} vs. {p2:?}");
}
```

This slide should take about 3 minutes.

- Türetme (derivation), makrolarla gerçekleştirilir (implemented) ve birçok kasa (crate) kullanışlı fonksiyonellik eklemek için faydalı türetme (derive) makroları sağlar. Örneğin, `serde`, `#[derive(Serialize)]` kullanarak bir yapı (struct) için serileştirme desteği türetebilir.
- Türetme (derivation) genellikle, çoğu durum için doğru olan yaygın bir basmakalıpsal (boilerplate-y) gerçekleştirmeye sahip özellikler (traits) için sağlanır. Örneğin, bir manuel `Clone` gerçekleştiriminin (impl) özelliği (trait) türetmeye kıyasla nasıl tekrarlanabilen bir yapıda olabileceğini gösterin:

  ```
  impl Clone for Player {
      fn clone(&self) -> Self {
          Player {
              name: self.name.clone(),
              strength: self.strength.clone(),
              hit_points: self.hit_points.clone(),
          }
      }
  }
  ```

  Yukarıdaki `.clone()`'ların hepsi bu durumda gerekli değildir, ancak bu, manuel gerçekleştirmelerin (impls) izleyeceği genel basmakalıpsal (boilerplate-y) deseni gösterir, bu da öğrencilere `derive` kullanımını netleştirmeye yardımcı olmalıdır.

## 13.4 Alıştırma: Kaydedici Özelliği (Logger Trait)

Bir `log` metoduna sahip bir `Logger` özelliği (trait) kullanarak basit bir kayıt (logging) aracı tasarlayalım. İlerlemesini kaydedebilecek kod daha sonra bir `&impl Logger` alabilir. Testte bu, mesajları test kayıt dosyasına (logfile) koyabilirken, bir üretim (production) inşasında mesajları bir kayıt sunucusuna (log server) gönderir.

Ancak, aşağıda verilen `StderrLogger` ayrıntı seviyesine (verbosity) bakılmaksızın tüm mesajları kaydeder. Göreviniz, maksimum ayrıntı seviyesi üzerindeki mesajları yoksayacak bir `VerbosityFilter` türü yazmaktır.

This is a common pattern: a struct wrapping a trait implementation and implementing that same trait, adding behavior in the process. In the "Generics" segment, we will see how to make the wrapper generic over the wrapped type.

```
trait Logger {
    /// Belirtilen ayrıntı seviyesinde bir mesaj kaydedin.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;
```

```
impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("ayrıntı seviyesi={verbosity}: {message}");
    }
}

/// Yalnızca belirtilen ayrıntı seviyesine kadar olan mesajları kaydedin.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}

// TODO: `VerbosityFilter` için `Logger` özelliğini (trait) gerçekleştirin.

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "Bilginiz Olsun");
    logger.log(2, "Eyvah");
}
```

### 13.4.1 Çözüm

```
trait Logger {
    /// Belirtilen ayrıntı seviyesinde bir mesaj kaydedin.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("ayrıntı seviyesi={verbosity}: {message}");
    }
}

/// Yalnızca belirtilen ayrıntı seviyesine kadar olan mesajları kaydedin.
struct VerbosityFilter {
    max_verbosity: u8,
    inner: StderrLogger,
}

impl Logger for VerbosityFilter {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
```

```
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "Bilginiz Olsun");
    logger.log(2, "Eyvah");
}
```

# Chapter 14

# Genelleştirmeler (Generics)

Bu bölüm yaklaşık 45 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Genelleştirilmiş (Generic) Fonksiyonlar | 5 dakika |
| Özellik (Trait) Sınırları | 10 dakika |
| Genelleştirilmiş (Generic) Veri Türleri | 10 dakika |
| impl Trait | 5 dakika |
| dyn Trait | 5 dakika |
| Alıştırma: Genelleştirilmiş (Generic) min | 10 dakika |

## 14.1 Genelleştirilmiş (Generic) Fonksiyonlar

Rust, algoritmaları veya veri yapılarını (sıralama veya ikili ağaç gibi) kullanılan veya depolanan türler üzerinde soyutlamanıza olanak tanıyan genelleştirmeleri (generics) destekler.

```
fn pick<T>(cond: bool, left: T, right: T) -> T {
    if cond { left } else { right }
}

fn main() {
    println!("bir sayı seçildi: {:?}", pick(true, 222, 333));
    println!("bir dize seçildi: {:?}", pick(false, 'L', 'R'));
}
```

This slide should take about 5 minutes.

- Genelleştirmelerin (generics) kod tekrarını nasıl azaltabileceğini göstermek için genelleştirilmiş `pick`'ten önce veya genelleştirmelerden sonra tek biçimli hâle getirmenin (monomorphization) nasıl çalıştığını göstermek için `pick`'in tek biçimli hâle getirilmiş (monomorphized) sürümlerini göstermek faydalı olabilir.

  ```
  fn pick_i32(cond: bool, left: i32, right: i32) -> i32 {
      if cond { left } else { right }
  ```

```
}

fn pick_char(cond: bool, left: char, right: char) -> char {
    if cond { left } else { right }
}
```

- Rust, argümanların ve geri dönüş değerinin türlerine dayanarak T için bir tür çıkarır (infers).
- Bu örnekte T için yalnızca ilkel (primitive) `i32` ve `char` türlerini kullanıyoruz, ancak burada kullanıcı tanımlı türler (user-defined types) de dahil olmak üzere herhangi bir türü kullanabiliriz:

```
struct Foo {
    val: u8,
}

pick(false, Foo { val: 7 }, Foo { val: 99 });
```

- Bu, C++ şablonlarına (templates) benzer, ancak Rust, genelleştirilmiş (generic) fonksiyonu kısmen de olsa anında derler, bu nedenle bu fonksiyon kısıtlamalarla (constraints) eşleşen tüm türler için geçerli olmalıdır. Örneğin, `cond` yanlışsa `left + right` geri döndürmek için `pick`'i değiştirmeyi deneyin. Yalnızca tamsayılarla `pick` örneği kullanılsa bile, Rust bunu yine de geçersiz kabul eder. C++ bunu yapmanıza izin verirdi.
- Genelleştirilmiş (generic) kod, çağrıldığı yerlerde (call sites) genelleştirilmemiş (özelleştirilmiş) koda dönüştürülür. Bu, sıfır maliyetli bir soyutlamadır: soyutlama olmadan veri yapılarını elle kodlamış olsaydınız alacağınız sonucun aynısını elde edersiniz.

## 14.2 Özellik (Trait) Sınırları

Genelleştirmelerle (generics) çalışırken, genellikle türlerin bazı özellikleri (trait) gerçekleştirmesini (implement) istersiniz, böylece bu özelliğin metotlarını çağırabilirsiniz.

Bunu `T: Trait` ile yapabilirsiniz:

```
fn duplicate<T: Clone>(a: T) -> (T, T) {
    (a.clone(), a.clone())
}

struct NotCloneable;

fn main() {
    let foo = String::from("foo");
    let pair = duplicate(foo);
    println!("{pair:?}");
}
```

This slide should take about 8 minutes.

- Bir `NotCloneable` yapıp `duplicate`'e geçirmeyi deneyin.
- Birden fazla özellik (trait) gerektiğinde, bunları birleştirmek için + kullanın.

- Öğrencilerin kod okurken karşılaşacakları bir `where` yan tümcesi gösterin.

```
fn duplicate<T>(a: T) -> (T, T)
where
    T: Clone,
{
    (a.clone(), a.clone())
}
```

  - Çok sayıda parametreniz varsa fonksiyon imzasını (signature) sadeleştirir.
  - Daha güçlü kılan ek özelliklere sahiptir.
    * Birisi sorarsa, ekstra özellik, ":" solundaki türün `Option<T>` gibi keyfi olabilmesidir.

- Rust'ın (şimdilik) özelleştirmeyi (specialization) desteklemediğini unutmayın. Örneğin, orijinal `duplicate` verildiğinde, özel bir `duplicate(a: u32)` eklemek geçersizdir.

## 14.3 Genelleştirilmiş (Generic) Veri Türleri

Belirli alan türü (concrete field type) üzerinde soyutlama yapmak için genelleştirmeleri (generics) kullanabilirsiniz. Önceki bölümün alıştırmasına geri dönersek:

```
pub trait Logger {
    /// Belirtilen ayrıntı seviyesinde bir mesaj kaydedin.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("ayrıntı seviyesi={verbosity}: {message}");
    }
}

/// Yalnızca belirtilen ayrıntı seviyesine kadar olan mesajları kaydedin.
struct VerbosityFilter<L> {
    max_verbosity: u8,
    inner: L,
}

impl<L: Logger> Logger for VerbosityFilter<L> {
    fn log(&self, verbosity: u8, message: &str) {
        if verbosity <= self.max_verbosity {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = VerbosityFilter { max_verbosity: 3, inner: StderrLogger };
    logger.log(5, "Bilginiz Olsun");
```

```
    logger.log(2, "Eyvah");
}
```

This slide should take about 10 minutes.

- *S:* Neden `L`, `impl<L: Logger> .. VerbosityFilter<L>`'de iki kez belirtiliyor? Bu gereksiz (redundant) değil mi?
  - Bunun nedeni, genelleştirilmiş (generic) tür için genel bir gerçekleştirme bölümü (generic implementation section) olmasıdır. Onlar, bağımsız olarak genelleştirilmiştir (generic).
  - Bu, bu metotların herhangi bir `L` için tanımlandığı anlamına gelir.
  - `impl VerbosityFilter<StderrLogger> { .. }` yazmak mümkündür.
    * `VerbosityFilter` hala genelleştirilmiştir türdür ve `VerbosityFilter<f64>` türünü kullanabilirsiniz, ancak bu bloktaki metotlar yalnızca `VerbosityFilter<StderrLogger>` için mevcut olacaktır.
- `VerbosityFilter` türünün kendisine bir özellik sınırı (trait bound)koymadığımıza dikkat edin. Oraya da sınırlar koyabilirsiniz, ancak genellikle Rust'ta özellik sınırlarını (trait bounds) yalnızca `impl` bloklarına koyarız.

## 14.4 Genelleştirilmiş (Generic) Özellikler (Traits)

Özellikler (traits), tıpkı türler ve fonksiyonlar gibi genelleştirilmiş (generic) de olabilir. Bir özelliğin parametreleri, kullanıldığında belirli (concrete) türler alır. Örneğin, `From<T>` özelliği tür dönüşümlerini (type conversions) tanımlamak için kullanılır:

```
pub trait From<T>: Sized {
    fn from(value: T) -> Self;
}

#[derive(Debug)]
struct Foo(String);

impl From<u32> for Foo {
    fn from(from: u32) -> Foo {
        Foo(format!("Tamsayıdan dönüştürüldü: {from}"))
    }
}

impl From<bool> for Foo {
    fn from(from: bool) -> Foo {
        Foo(format!("Bool'dan dönüştürüldü: {from}"))
    }
}

fn main() {
    let from_int = Foo::from(123);
    let from_bool = Foo::from(true);
    dbg!(from_int);
    dbg!(from_bool);
}
```

- `From` özelliği (trait) kursun ilerleyen bölümlerinde ele alınacaktır, ancak `std`

belgelerindeki tanımı basittir ve referans için buraya kopyalanmıştır.

- Özelliğin (trait) gerçekleştirmeleri (implementations), olası tüm tür parametrelerini kapsamak zorunda değildir. Burada, `Foo` için `From<&str>` gerçekleştirmesi olmadığından `Foo::from("hello")` derlenmezdi.
- Genelleştirilmiş (generic) özellikler (traits) türleri "girdi" olarak alırken, ilişkili türler (associated types) bir tür "çıktı" türüdür. Bir özelliğin farklı girdi türleri için birden çok gerçekleştirmesi (implementation) olabilir.
- Aslında, Rust herhangi bir T türü için bir özelliğin (trait) en fazla bir gerçekleştiriminin (implementation) eşleşmesini gerektirir. Diğer bazı dillerden farklı olarak, Rust'ın "en spesifik" eşleşmeyi seçmek için bir sezgisi (heuristic) yoktur. Bu desteği eklemek için özelleştirme (specialization) adı verilen bir çalışma vardır.

## 14.5 `impl Trait`

Özellik sınırlarına (trait bounds) benzer şekilde, fonksiyon argümanlarında ve geri dönüş değerlerinde bir `impl Trait` sözdizimi kullanılabilir:

```
// Sözdizimsel kolaylık:
//   fn add_42_millions<T: Into<i32>>(x: T) -> i32 {
fn add_42_millions(x: impl Into<i32>) -> i32 {
    x.into() + 42_000_000
}

fn pair_of(x: u32) -> impl std::fmt::Debug {
    (x + 1, x - 1)
}

fn main() {
    let many = add_42_millions(42_i8);
    dbg!(many);
    let many_more = add_42_millions(10_000_000);
    dbg!(many_more);
    let debuggable = pair_of(27);
    dbg!(debuggable);
}
```

This slide should take about 5 minutes.

`impl Trait`, isimlendiremeyeceğiniz türlerle çalışmanıza olanak tanır. `impl Trait`'in anlamı farklı konumlarda biraz farklıdır.

- Bir parametre için, `impl Trait` bir özellik sınırına (trait bound) sahip anonim bir genelleştirilmiş (generic) parametre gibidir.
- Bir geri dönüş türü için, bu, geri dönüş türünün, türü isimlendirmeden özelliği (trait) gerçekleştiren (implement) bir belirli (concrete) tür olduğu anlamına gelir. Bu, genel bir API'de belirli olan türü açığa çıkarmak istemediğinizde kullanışlı olabilir.

  Geri dönüş konumunda (return position) çıkarım (inference) yapmak zordur. `impl Foo` geri döndüren bir fonksiyon, geri döndürdüğü belirli olan türü kaynakta yazmadan seçer. `collect<B>() -> B` gibi genelleştirilmiş bir tür döndüren bir fonksiyon, `B`'yi sağlayan

herhangi bir türü geri döndürebilir ve çağıranın, `let x: Vec<_> = foo.collect()` veya tür belirteciyle (turbo balığı / turbofish) ile `foo.collect::<Vec<_>>()` gibi birini seçmesi gerekebilir.

debuggable'ın türü nedir? Hata mesajının ne gösterdiğini görmek için `let debuggable: () = ..`'yi deneyin.

## 14.6 `dyn Trait`

Rust, genelleştirmeler (generics) yoluyla statik yönlendirme (static dispatch) için özellikleri (trait) kullanmaya ek olarak, özellik nesneleri (trait objects) yoluyla silinmiş türle (type-erased), dinamik yönlendirmeyi (dynamic dispatch) destekler:

```
struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    lives: i8,
}

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Hav, benim adım {}!", self.name)
    }
}

impl Pet for Cat {
    fn talk(&self) -> String {
        String::from("Miyav!")
    }
}

// Genelleştirmeleri (generics) ve statik yönlendirmeyi (static dispatch) kullanır.
fn generic(pet: &impl Pet) {
    println!("Merhaba, nasılsın? {}", pet.talk());
}

// Tür silmeyi (type-erasure) ve dinamik yönlendirmeyi (dynamic dispatch) kullanır.
fn dynamic(pet: &dyn Pet) {
    println!("Merhaba, nasılsın? {}", pet.talk());
}

fn main() {
    let cat = Cat { lives: 9 };
    let dog = Dog { name: String::from("Fido"), age: 5 };
```

```
    generic(&cat);
    generic(&dog);

    dynamic(&cat);
    dynamic(&dog);
}
```

This slide should take about 5 minutes.

- `impl Trait` de dahil olmak üzere genelleştirmeler (generics), genelleştirmenin oluşturulduğu her farklı tür için fonksiyonun özel bir örneğini oluşturmak üzere tek biçimli hâle getirmeyi (monomorphization) kullanır. Bu, genelleştirilmiş bir fonksiyon içinden bir özellik (trait) metodunu çağırmanın hala statik yönlendirme (static dispatch) kullandığı anlamına gelir, çünkü derleyici tam tür bilgisine sahiptir ve hangi türün özellik gerçekleştiriminin (trait implementation) kullanılacağını çözebilir.
- `dyn Trait` kullanırken, bunun yerine bir sanal metot tablosu (virtual method table) (vtable) aracılığıyla dinamik yönlendirme (dynamic dispatch) kullanır. Bu, hangi türde `Pet` geçirilirse geçirilsin kullanılan tek bir `fn dynamic` sürümü olduğu anlamına gelir.
- `dyn Trait` kullanırken, özellik nesnesinin (trait object) bir tür dolaylı yol (indirection) arkasında olması gerekir. Bu durumda bu bir referanstır, ancak `Box` gibi akıllı gösterici (smart pointer) türleri de kullanılabilir (bu 3. günde gösterilecektir).
- Çalışma zamanında, bir `&dyn Pet`, ”genişletilmiş bir gösterici (fat pointer)” olarak temsil edilir, yani iki göstericiden oluşan bir çift: Bir gösterici, `Pet`'i gerçekleştiren (implement) belirli nesneye işaret eder ve diğeri o tür için özellik gerçekleştiriminin (trait implementation) sanal metot tablosuna (vtable) işaret eder. `&dyn Pet` üzerinde `talk` metodunu çağırırken, derleyici sanal metot tablosunda `talk` için fonksiyon göstericisini (function pointer) arar ve ardından `Dog` veya `Cat`'e göstericiyi o fonksiyona geçirerek fonksiyonu çağırır. Derleyicinin bunu yapmak için `Pet`'in belirli olan türünü (concrete type) bilmesi gerekmez.
- Bir `dyn Trait`, ”silinmiş tür (type-erased)” olarak kabul edilir, çünkü artık somut türün ne olduğuna dair derleme zamanı bilgisine sahip değiliz.

## 14.7 Alıştırma: Genelleştirilmiş (Generic) `min`

Bu kısa alıştırmada, `Ord` özelliğini (trait) kullanarak iki değerin minimumunu belirleyen genelleştirilmiş (generic) bir `min` fonksiyonu gerçekleştireceksiniz (implement).

```
use std::cmp::Ordering;

// TODO: testlerde kullanılan `min` fonksiyonunu gerçekleştirin.

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
```

```
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("merhaba", "hoşçakal"), "hoşçakal");
    assert_eq!(min("yarasa", "armadillo"), "armadillo");
}
```

This slide and its sub-slides should take about 10 minutes.

- Öğrencilere `Ord` özelliğini (trait) ve `Ordering` enum'ını gösterin.

### 14.7.1 Çözüm

```
use std::cmp::Ordering;

fn min<T: Ord>(l: T, r: T) -> T {
    match l.cmp(&r) {
        Ordering::Less | Ordering::Equal => l,
        Ordering::Greater => r,
    }
}

#[test]
fn integers() {
    assert_eq!(min(0, 10), 0);
    assert_eq!(min(500, 123), 123);
}

#[test]
fn chars() {
    assert_eq!(min('a', 'z'), 'a');
    assert_eq!(min('7', '1'), '1');
}

#[test]
fn strings() {
    assert_eq!(min("merhaba", "hoşçakal"), "hoşçakal");
    assert_eq!(min("yarasa", "armadillo"), "armadillo");
}
```

# Part IV

# Gün 2: Öğleden Sonra

# Chapter 15

# Tekrar Hoş Geldiniz

Bu oturum 10 dakikalık aralar dahil yaklaşık 2 saat 50 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Çevreleyiciler (Closures) | 30 dakika |
| Standart Kütüphane Türleri | 1 saat |
| Standart Kütüphanedeki Özellikler (Traits) | 1 saat |

# Chapter 16

# Çevreleyiciler (Closures)

Bu bölüm yaklaşık 30 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Çevreleyici Sözdizimi (Closure Syntax) | 3 dakika |
| Yakalama (Capturing) | 5 dakika |
| Çevreleyici Özellikler (Closure Traits) | 10 dakika |
| Alıştırma: Kaydedici (Log) Filtresi | 10 dakika |

## 16.1 Çevreleyici Sözdizimi (Closure Syntax)

Çevreleyiciler (closures) dikey çubuklarla oluşturulur: `|..|` `...`

```
fn main() {
    // Hafif sözdizimi (lightweight syntax) için, argüman ve geri dönüş türü çıkarımı ya
    let double_it = |n| n * 2;
    dbg!(double_it(50));

    // Veya türleri belirtebilir ve tamamen açık (fully explicit) olmak için gövdeyi pa
    let add_1f32 = |x: f32| -> f32 { x + 1.0 };
    dbg!(add_1f32(50.));
}
```

This slide should take about 3 minutes.

- Argümanlar `|..|` arasına girer. Gövde `{ .. }` içine alınır., ancak tek bir ifade (expression) ise bunlar atlanabilir.
- Argüman türleri isteğe bağlıdır ve eğer verilmezse türler çıkarım (inference) yapılır. Geri dönüş türü (return type) de isteğe bağlıdır, ancak yalnızca gövdenin etrafında `{ .. }` varsa yazılabilir.
- Örneklerin her ikisi de bunun yerine sadece iç içe fonksiyonlar olarak yazılabilir -- çünkü bulundukları sözcüksel ortamdan (lexical environment) herhangi bir değişkeni yakalamazlar. Yakalamayı (capturing) bir sonraki adımda göreceğiz.

### Daha Fazlasını Keşfedin

- Fonksiyonları değişkenlerde saklama yeteneği sadece çevreleyicilere (closures) özgü değildir, düzenli (regular) fonksiyonlar da değişkenlere konulabilir ve ardından çevreleyicilerle aynı şekilde çağrılabilir: Deneme alanında (playground) örnek.
  - Bağlantısı verilen örnek ayrıca hiçbir şey yakalamayan (capture) çevreleyicilerin (closures) de düzenli bir fonksiyon göstericisine (regular function pointer) dönüştürülebileceğini (coerce) göstermektedir.

## 16.2 Yakalama (Capturing)

Bir çevreleyici (closure), tanımlandığı ortamdan değişkenleri yakalayabilir (capture).

```
fn main() {
    let max_value = 5;
    let clamp = |v| {
        if v > max_value { max_value } else { v }
    };

    dbg!(clamp(1));
    dbg!(clamp(3));
    dbg!(clamp(5));
    dbg!(clamp(7));
    dbg!(clamp(10));
}
```

This slide should take about 5 minutes.

- Varsayılan olarak, bir çevreleyici (closure) değerleri referans yoluyla yakalar. Burada `max_value`, `clamp` tarafından yakalanır, ancak yazdırma için `main`'de hala kullanılabilir. `max_value` değişkenini değiştirilebilir (mutable) yapmayı, değiştirmeyi ve değerleri tekrar yazdırmayı deneyin. Bu neden işe yaramıyor?
- Bir çevreleyici (closure) değerleri değiştirirse, onları değiştirilebilir (mutable) referans yoluyla yakalar. `clamp`'e `max_value += 1` eklemeyi deneyin.
- Bir çevreleyiciyi (closure), `move` anahtar kelimesiyle değerleri referans almak yerine taşımaya (move) zorlayabilirsiniz. Bu, ömürlerle (lifetimes) ilgili yardımcı olabilir, örneğin çevreleyici yakalanan değerlerden daha uzun yaşamalıysa (ömürler hakkında daha sonra daha fazla bilgi).

  Bu, `move |v| ..` gibi görünüyor. Bu anahtar kelimeyi eklemeyi deneyin ve `clamp`'i tanımladıktan sonra `main`'in hala `max_value`'e erişip erişemediğini görün.
- Varsayılan olarak, çevreleyiciler (closures) dış kapsamdan (outer scope) her değişkeni, yapabildikleri en az talepkar erişim biçimiyle yakalarlar (mümkünse paylaşılan (shared) referansla, sonra özel (exclusive) referansla, sonra taşıma (move) ile). `move` anahtar kelimesi, değerle yakalamayı (move)zorlar.

## 16.3 Çevreleyici özellikleri (Closure traits)

Çevreleyicilerin (closures) veya lambda ifadelerinin isimlendirilemeyen türleri vardır. Ancak, özel `Fn`, `FnMut` ve `FnOnce` özelliklerini (traits) gerçekleştirirler (implement):

Özel `fn(..) -> T` türleri, fonksiyon göstericilerine (function pointers) referans verir - ya bir fonksiyonun adresi ya da hiçbir şey yakalamayan (capture) bir çevreleyici (closure).

```
fn apply_and_log(
    func: impl FnOnce(&'static str) -> String,
    func_name: &'static str,
    input: &'static str,
) {
    println!("Çağrılıyor {func_name}({input}): {}", func(input))
}

fn main() {
    let suffix = "sendromu";
    let add_suffix = |x| format!("{x} {suffix}");
    apply_and_log(&add_suffix, "add_suffix", "son sınıf");
    apply_and_log(&add_suffix, "add_suffix", "appendix");

    let mut v = Vec::new();
    let mut accumulate = |x| {
        v.push(x);
        v.join("/")
    };
    apply_and_log(&mut accumulate, "accumulate", "kırmızı");
    apply_and_log(&mut accumulate, "accumulate", "yeşil");
    apply_and_log(&mut accumulate, "accumulate", "mavi");

    let take_and_reverse = |prefix| {
        let mut acc = String::from(prefix);
        acc.push_str(&v.into_iter().rev().collect::<Vec<_>>().join("/"));
        acc
    };
    apply_and_log(take_and_reverse, "take_and_reverse", "ters çevrilmiş: ");
}
```

This slide should take about 10 minutes.

Bir `Fn` (ör. `add_suffix`) yakalanan değerleri (captured values) ne tüketir ne de değiştirir. Sadece çevreleyiciye (closure) paylaşılan bir referans (shared reference) gerektirerek çağrılabilir, bu da çevreleyicinin tekrar tekrar ve hatta eş zamanlı olarak (concurrently) yürütülebileceği anlamına gelir.

Bir `FnMut` (ör. `accumulate`) yakalanan değerleri (captured values) değiştirebilir. Çevreleyici (closure) nesnesine özel (exclusive) referans yoluyla erişilir, bu nedenle tekrar tekrar çağrılabilir ancak eş zamanlı olarak (concurrently) çağrılamaz.

Eğer bir `FnOnce`'ınız varsa (ör. `take_and_reverse`), onu yalnızca bir kez çağırabilirsiniz. Bunu yapmak, çevreleyiciyi (closure) ve taşıma (move) ile yakalanan tüm değerleri tüketir.

`FnMut`, `FnOnce`'ın bir alt türüdür. `Fn`, `FnMut` ve `FnOnce`'ın bir alt türüdür. Yani, `FnOnce`'ın

istendiği her yerde bir `FnMut` kullanabilirsiniz ve `FnMut` veya `FnOnce`'ın istendiği her yerde bir `Fn` kullanabilirsiniz.

Bir çevreleyici (closure) alan bir fonksiyon tanımladığınızda, yapabiliyorsanız `FnOnce` (yani onu bir kez çağırırsınız), değilse `FnMut` ve son olarak `Fn` almalısınız. Bu, çağırana (caller) en fazla esnekliği sağlar.

Buna karşılık, bir çevreleyiciniz (closure) olduğunda, sahip olabileceğiniz en esnek olan `Fn`'dir (3 çevreleyici özelliğinin (trait) herhangi birinin tüketicisine geçirilebilir), sonra `FnMut` ve son olarak `FnOnce`.

Derleyici ayrıca, çevreleyicinin (closure) ne yakaladığına bağlı olarak `Copy` (ör. `add_suffix` için) ve `Clone` (ör. `take_and_reverse`) çıkarımı yapar (infers). Fonksiyon göstericileri (`fn` öğelerine referanslar) `Copy`'yi ve `Fn`'yi gerçekleştirir (implement).

## 16.4 Alıştırma: Kaydedici (Log) Filtresi

Bu sabahki genel kaydediciden (logger) yola çıkarak, kayıt mesajlarını filtrelemek için bir çevreleyici (closure) kullanan bir `Filter` gerçekleştirin (implement), filtreleme koşulunu geçenleri bir iç kaydediciye (logger) gönderin.

```
pub trait Logger {
    /// Belirtilen ayrıntı seviyesinde bir mesaj kaydedin.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;

impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("ayrıntı seviyesi={verbosity}: {message}");
    }
}

// TODO: `Filter`'ı tanımlayın ve gerçekleştirin (implement).

fn main() {
    let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("berbat"));
    logger.log(5, "Bilginiz Olsun");
    logger.log(1, "berbat, bir şeyler ters gitti");
    logger.log(2, "eyvah");
}
```

### 16.4.1 Çözüm

```
pub trait Logger {
    /// Belirtilen ayrıntı seviyesinde bir mesaj kaydedin.
    fn log(&self, verbosity: u8, message: &str);
}

struct StderrLogger;
```

```
impl Logger for StderrLogger {
    fn log(&self, verbosity: u8, message: &str) {
        eprintln!("ayrıntı seviyesi={verbosity}: {message}");
    }
}

/// Yalnızca bir filtreleme koşuluyla eşleşen günlük mesajlarını kaydet.
struct Filter<L, P> {
    inner: L,
    predicate: P,
}

impl<L, P> Filter<L, P>
where
    L: Logger,
    P: Fn(u8, &str) -> bool,
{
    fn new(inner: L, predicate: P) -> Self {
        Self { inner, predicate }
    }
}
impl<L, P> Logger for Filter<L, P>
where
    L: Logger,
    P: Fn(u8, &str) -> bool,
{
    fn log(&self, verbosity: u8, message: &str) {
        if (self.predicate)(verbosity, message) {
            self.inner.log(verbosity, message);
        }
    }
}

fn main() {
    let logger = Filter::new(StderrLogger, |_verbosity, msg| msg.contains("berbat"));
    logger.log(5, "Bilginiz Olsun");
    logger.log(1, "berbat, bir şeyler ters gitti");
    logger.log(2, "eyvah");
}
```

- İlk `Filter` impl bloğundaki `P: Fn(u8, &str) -> bool` sınırının kesinlikle gerekli olmadığını, ancak new çağrılırken tür çıkarımına (type inference) yardımcı olduğunu unutmayın. Onu kaldırmayı gösterin ve derleyicinin şimdi new'e geçirilen çevreleyici (closure) için tür ek açıklamalarına (type annotations) nasıl ihtiyaç duyduğunu gösterin.

# Chapter 17

# Standart Kütüphane Türleri

Bu bölüm yaklaşık 1 saat sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Standart Kütüphane | 3 dakika |
| Dokümantasyon | 5 dakika |
| Option | 10 dakika |
| Result | 5 dakika |
| String | 5 dakika |
| Vec | 5 dakika |
| HashMap | 5 dakika |
| Alıştırma: Sayıcı | 20 dakika |

Bu bölümdeki slaytların her biri için, dokümantasyon sayfalarını gözden geçirerek biraz zaman ayırın ve daha yaygın metotlardan bazılarını öne çıkarın.

## 17.1 Standart Kütüphane

Rust, Rust kütüphaneleri ve programları tarafından kullanılan bir dizi ortak türün oluşturulmasına yardımcı olan bir standart kütüphane ile birlikte gelir. Bu şekilde, her hangi iki kütüphane aynı `String` türünü kullandığı için sorunsuzca birlikte çalışabilir.

Aslında Rust, Standart Kütüphanenin birkaç katmanını içerir: `core`, `alloc` ve `std`.

- `core`, `libc`'ye, tahsis ediciye (allocator) ve hatta bir işletim sisteminin varlığına bağlı olmayan en temel türleri ve fonksiyonları içerir.
- `alloc`, `Vec`, `Box` ve `Arc` gibi global bir dinamik bellek tahsis edicisini (heap allocator) gerektiren türleri içerir.
- Gömülü Rust uygulamaları genellikle sadece `core` ve bazen de `alloc` kullanır.

## 17.2 Dokümantasyon

Rust, kapsamlı bir dokümantasyonla birlikte gelir. Örneğin:

- Döngüler hakkındaki tüm ayrıntılar.
- u8 gibi ilkel türler (primitive types).
- Option veya BinaryHeap gibi standart kütüphane türleri.

Dokümantasyonu görüntülemek için `rustup doc --std` veya https://std.rs kullanın.

Aslında, kendi kodunuzu belgeleyebilirsiniz:

```
/// İlk argümanın ikinci argümana bölünüp bölünemeyeceğini belirle.
///
/// İkinci argüman sıfır ise, sonuç yanlıştır.
fn is_divisible_by(lhs: u32, rhs: u32) -> bool {
    if rhs == 0 {
        return false;
    }
    lhs % rhs == 0
}
```

İçerikler Markdown olarak işlenir. Yayınlanan tüm Rust kütüphane kasaları (crates), rustdoc aracı kullanılarak docs.rs adresinde otomatik olarak belgelenir. Bir API'deki tüm genel (public) öğeleri bu deseni kullanarak belgelemek, tercih edilen yaklaşımdır (idiomatic).

Bir öğeyi, öğenin içinden (bir modülün içi gibi) belgelemek için "iç doküman yorumları (inner doc comments)" olarak adlandırılan `//!` veya `/*! .. */` kullanın:

```
//! Bu modül, tamsayıların bölünebilirliği ile ilgili fonksiyonellik içerir.
```

This slide should take about 5 minutes.

- Öğrencilere https://docs.rs/rand adresindeki `rand` kasası (crate) için oluşturulmuş belgeleri gösterin.

## 17.3 Option

Daha önce `Option<T>`'nin bazı kullanımlarını görmüştük. Ya `T` türünde bir değer saklar ya da hiçbir şey saklamaz. Örneğin, `String::find` bir `Option<usize>` geri döndürür.

```
fn main() {
    let name = "Löwe 老虎 Léopard Gepardi";
    let mut position: Option<usize> = name.find('é');
    dbg!(position);
    assert_eq!(position.unwrap(), 14);
    position = name.find('Z');
    dbg!(position);
    assert_eq!(position.expect("Karakter bulunamadı"), 0);
}
```

This slide should take about 10 minutes.

- `Option` sadece standart kütüphanede kullanılır gibi bir şey yoktur, yaygın olarak kullanılır.
- `unwrap`, bir `Option` içindeki değeri geri döndürür veya paniğe (panic) neden olur. `expect` benzerdir ancak bir hata mesajı (error message) alır.

  - None durumunda panik alabilirsiniz, ancak None'ı kontrol etmeyi ”yanlışlıkla” unutamazsınız.
  - Bir şeyler üzerinde çalışırken her yerde `unwrap`/`expect` kullanmak yaygındır, ancak üretim (production) kodu genellikle None'ı daha şık bir şekilde ele alır.
- ”Niş/oyuk optimizasyonu (niche optimization)”, T'nin geçerli bir değeri olmayan bir gösterimi (representation) varsa, `Option<T>`'nin genellikle bellekte `T` ile aynı boyuta sahip olduğu anlamına gelir. Örneğin, bir referans NULL olamaz, bu nedenle `Option<&T>` None varyantını temsil etmek için otomatik olarak NULL kullanır ve bu sayede `&T` ile aynı bellekte saklanabilir.

## 17.4 Result

`Result`, `Option`'a benzer, ancak her biri farklı bir enum varyantına sahip bir işlemin başarısını veya başarısızlığını belirtir. Genelleştirilmiştir (generic): `Result<T, E>`; burada `T`, `Ok` varyantında kullanılır ve `E`, `Err` varyantında görünür.

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("günlük.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Sevgili günlük: {contents} ({bytes} bayt)");
            } else {
                println!("Dosya içeriği okunamadı");
            }
        }
        Err(err) => {
            println!("Günlük açılamadı: {err}");
        }
    }
}
```

This slide should take about 5 minutes.

- `Option`'da olduğu gibi, başarılı değer `Result`'ın içinde yer alır ve geliştiriciyi onu açıkça (explicitly) çıkarmaya zorlar. Bu, hata kontrolünü teşvik eder. Bir hatanın asla olmaması gereken durumlarda, `unwrap()` veya `expect()` çağrılabilir ve bu aynı zamanda geliştiricinin niyetinin de bir işaretidir.
- `Result` dokümantasyonunu okumanız tavsiye edilir. Kurs sırasında değil, ama bahsetmeye değer. Fonksiyonel tarzda programlamaya yardımcı olan birçok kullanışlı metot ve fonksiyon içerir.
- `Result`, 4. Günde göreceğimiz gibi hata işlemeyi (error handling) gerçekleştirmek (implement) için standart türdür.

## 17.5 String

`String` türü, büyüyebilir, UTF-8 olarak kodlanmış bir dizedir (string):

```
fn main() {
    let mut s1 = String::new();
    s1.push_str("Merhaba");
    println!("s1: uzunluk = {}, kapasite = {}", s1.len(), s1.capacity());

    let mut s2 = String::with_capacity(s1.len() + 1);
    s2.push_str(&s1);
    s2.push('!');
    println!("s2: uzunluk = {}, kapasite = {}", s2.len(), s2.capacity());

    let s3 = String::from("🇹🇷");
    println!("s3: uzunluk = {}, karakter sayısı = {}", s3.len(), s3.chars().count());
}
```

`String`, `Deref<Target = str>` özelliğini gerçekleştirir (implement), bu da bir `String` üzerinde tüm `str` metotlarını çağırabileceğiniz anlamına gelir.

This slide should take about 5 minutes.

- `String::new` yeni bir boş dize geri döndürür, dizeye ne kadar veri eklemek istediğinizi bildiğinizde `String::with_capacity` kullanın.
- `String::len`, `String`'in bayt cinsinden boyutunu geri döndürür (bu, karakter cinsinden uzunluğundan farklı olabilir).
- `String::chars`, gerçek karakterler üzerinde bir adımlayıcı (iterator) geri döndürür. yazıbirim (grapheme) kümeleri nedeniyle bir `char`'ın bir insanın ”karakter” olarak kabul edeceği şeyden farklı olabileceğini unutmayın.
- İnsanlar dizelerden (strings) bahsettiğinde, ya `&str` ya da `String`'den bahsediyor olabilirler.
- Bir tür `Deref<Target = T>`'yi gerçekleştirdiğinde (implement), derleyici `T`'den metotları şeffaf bir şekilde çağırmanıza izin verir.
  - `Deref` özelliğini (trait) henüz tartışmadık, bu yüzden bu noktada bu çoğunlukla dokümantasyondaki kenar çubuğunun yapısını (neden String türünün altında str metotlarının da görüldüğünü) açıklar.
  - `String`, `Deref<Target = str>` özelliğini gerçekleştirir (implement) ve bu ona şeffaf bir şekilde `str`'nin metotlarına erişim sağlar.
  - `let s3 = s1.deref();` ve `let s3 = &*s1;` yazıp karşılaştırın.
- `String`, bir bayt vektörü etrafında bir sarmalayıcı (wrapper) olarak gerçekleştirilmiştir (implement), vektörlerde desteklenen birçok işlem `String`'de de desteklenir, ancak bazı ek garantilerle.
- Bir `String`'e erimenin farklı yollarını karşılaştırın:
  - `i`'nin sınırlar içinde veya dışında olduğu `s3.chars().nth(i).unwrap()` kullanarak bir karaktere erişmek.
  - Bu dilimin (slice) karakter sınırlarına denk gelip gelmediği durumlarda `s3[0..4]` kullanarak bir alt dizeye erişmek.
- Birçok tür, `to_string` metodu ile bir dizeye dönüştürülebilir. Bu özellik (trait), `Display`'i gerçekleştiren (implement) tüm türler için otomatik olarak gerçekleştirilir, bu nedenle biçimlendirilebilen (format) her şey bir dizeye de dönüştürülebilir.

## 17.6 Vec

Vec, standart yeniden boyutlandırılabilir dinamik bellekten tahsis edilen arabellektir (heap-allocated buffer):

```
fn main() {
    let mut v1 = Vec::new();
    v1.push(42);
    println!("v1: uzunluk = {}, kapasite = {}", v1.len(), v1.capacity());

    let mut v2 = Vec::with_capacity(v1.len() + 1);
    v2.extend(v1.iter());
    v2.push(9999);
    println!("v2: uzunluk = {}, kapasite = {}", v2.len(), v2.capacity());

    // Bir vektörü elemanlarla ilklendirmek (initialize) için standart makro.
    let mut v3 = vec![0, 0, 1, 2, 3, 4];

    // Sadece çift elemanları tut.
    v3.retain(|x| x % 2 == 0);
    println!("{v3:?}");

    // Ardışık kopyaları sil.
    v3.dedup();
    println!("{v3:?}");
}
```

`Vec`, `Deref<Target = [T]>` özelliğini gerçekleştirir (implement), bu da bir `Vec` üzerinde dilim (slice) metotlarını çağırabileceğiniz anlamına gelir.

This slide should take about 5 minutes.

- `Vec`, `String` ve `HashMap` ile birlikte bir koleksiyon türüdür. İçerdiği veriler dinamik bellekte (heap) saklanır. Bu, veri miktarının derleme zamanında bilinmesi gerekmediği anlamına gelir. Çalışma zamanında büyüyebilir veya küçülebilir.
- `Vec<T>`'nin de genelleştirilmiş (generic) bir tür olduğuna, ancak `T`'yi açıkça belirtmek zorunda olmadığınıza dikkat edin. Rust tür çıkarımında (type inference) her zaman olduğu gibi, `T` ilk `push` çağrısı sırasında belirlenmiştir.
- `vec![...]`, `Vec::new()` yerine kullanılacak standart bir makrodur ve vektöre başlangıç elemanları eklemeyi destekler.
- Vektörün elemanlarına erişmek için `[ ]` kullanırsınız, ancak sınırlar dışındaysa paniğe (panic) neden olurlar. Alternatif olarak, `get` kullanmak bir `Option` geri döndürür. `pop` fonksiyonu son elemanı siler.

## 17.7 HashMap

HashDoS saldırılarına karşı korumalı standart özet haritası (hash map):

```
use std::collections::HashMap;

fn main() {
    let mut page_counts = HashMap::new();
```

```
    page_counts.insert("Fadiş", 207);
    page_counts.insert("Dede Korkut Hikayeleri", 751);
    page_counts.insert("Aşk-ı Memnu", 303);

    if !page_counts.contains_key("İnce Mehmed") {
        println!(
            "{} kitap hakkında bilgimiz var, ama İnce Mehmed hakkında yok.",
            page_counts.len()
        );
    }

    for book in ["Aşk-ı Memnu", "Keloğlan Masalları "] {
        match page_counts.get(book) {
            Some(count) => println!("{book}: {count} sayfa"),
            None => println!("{book} bilinmiyor."),
        }
    }

    // Eğer bir şey bulunmazsa bir değer eklemek için .entry() metodunu kullanın.
    for book in ["Aşk-ı Memnu", "Keloğlan Masalları "] {
        let page_count: &mut i32 = page_counts.entry(book).or_insert(0);
        *page_count += 1;
    }

    dbg!(page_counts);
}
```

This slide should take about 5 minutes.

- `HashMap`, başlangıç (prelude) kısmında tanımlı değildir ve kapsama (scope) alınması gerekir.
- Aşağıdaki kod satırlarını deneyin. İlk satır bir kitabın hash haritasında olup olmadığına bakar ve yoksa alternatif bir değer geri döndürür. İkinci satır, kitap bulunmazsa alternatif değeri hash haritasına ekler.

  ```
  let pc1 = page_counts
      .get("Osmanlı Cadısı")
      .unwrap_or(&336);
  let pc2 = page_counts
      .entry("Gri Gökyüzü ")
      .or_insert(374);
  ```

- `vec!`'in aksine, maalesef standart bir `hashmap!` makrosu yoktur.
  - Ancak, Rust 1.56'dan beri, HashMap `From<[(K, V); N]>` özelliğini (trait) gerçekleştirir (implement), bu da bir özet haritasını (hash map) bir değişmez (literal) diziden kolayca ilklendirmemize (initialize) olanak tanır:

    ```
    let page_counts = HashMap::from([
      ("Osmanlı Cadısı".to_string(), 336),
      ("Gri Gökyüzü ".to_string(), 374),
    ]);
    ```

- Alternatif olarak HashMap, anahtar-değer (key-value) demetleri (tuple) üreten herhangi bir `Iterator`'dan oluşturulabilir.
- Bu tür, `std::collections::hash_map::Keys` gibi birkaç "metoda özgü" geri dönüş türüne sahiptir. Bu türler genellikle Rust belgelerindeki aramalarda görünür. Öğrencilere bu türün belgelerini ve `keys` metoduna geri dönen yardımcı bağlantıyı gösterin.

## 17.8 Alıştırma: Sayıcı

Bu alıştırmada çok basit bir veri yapısı alıp onu genelleştirilmiş (generic) hale getireceksiniz. Hangi değerlerin görüldüğünü ve her birinin kaç kez göründüğünü takip etmek için bir `std::collections::HashMap` kullanır.

`Counter`'ın ilk sürümü sadece `u32` değerleri için çalışacak şekilde sabit kodlanmıştır. Yapıyı ve metotlarını, izlenen değerin türü üzerinde genelleştirme (generic) yapın, bu şekilde `Counter` her tür değeri izleyebilir.

Eğer erken bitirirseniz, `count` metodunu gerçekleştirmek (implement) için gereken özet (hash) arama sayısını yarıya indirmek için `entry` metodunu kullanmayı deneyin.

```
use std::collections::HashMap;

/// Counter, T türündeki her bir değerin kaç kez görüldüğünü sayar.
struct Counter {
    values: HashMap<u32, u64>,
}

impl Counter {
    /// Yeni bir Counter oluştur.
    fn new() -> Self {
        Counter {
            values: HashMap::new(),
        }
    }

    /// Verilen değerin bir tekrarını say.
    fn count(&mut self, value: u32) {
        if self.values.contains_key(&value) {
            *self.values.get_mut(&value).unwrap() += 1;
        } else {
            self.values.insert(value, 1);
        }
    }

    /// Verilen değerin kaç kez görüldüğünü geri döndür.
    fn times_seen(&self, value: u32) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
```

```
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);

    for i in 10..20 {
        println!("{} değer {}`e eşit görüldü", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("elma");
    strctr.count("portakal");
    strctr.count("elma");
    println!("{} elma alındı", strctr.times_seen("elma"));
}
```

### 17.8.1 Çözüm

```
use std::collections::HashMap;
use std::hash::Hash;

/// Counter, T türündeki her bir değerin kaç kez görüldüğünü sayar.
struct Counter<T> {
    values: HashMap<T, u64>,
}

impl<T: Eq + Hash> Counter<T> {
    /// Yeni bir Counter oluştur.
    fn new() -> Self {
        Counter { values: HashMap::new() }
    }

    /// Verilen değerin bir tekrarını say.
    fn count(&mut self, value: T) {
        *self.values.entry(value).or_default() += 1;
    }

    /// Verilen değerin kaç kez görüldüğünü geri döndür.
    fn times_seen(&self, value: T) -> u64 {
        self.values.get(&value).copied().unwrap_or_default()
    }
}

fn main() {
    let mut ctr = Counter::new();
    ctr.count(13);
    ctr.count(14);
    ctr.count(16);
```

```
    ctr.count(14);
    ctr.count(14);
    ctr.count(11);

    for i in 10..20 {
        println!("{} değer {}`e eşit görüldü", ctr.times_seen(i), i);
    }

    let mut strctr = Counter::new();
    strctr.count("elma");
    strctr.count("portakal");
    strctr.count("elma");
    println!("{} elma alındı", strctr.times_seen("elma"));
}
```

# Chapter 18

# Standart Kütüphanedeki Özellikler (Traits)

Bu bölüm yaklaşık 1 saat sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Karşılaştırmalar | 5 dakika |
| Operatörler | 5 dakika |
| From ve Into | 5 dakika |
| Tür Dönüştürme (Casting) | 5 dakika |
| Read ve Write | 5 dakika |
| Default, yapı (struct) güncelleme sözdizimi | 5 dakika |
| Alıştırma: ROT13 | 30 dakika |

Standart kütüphane türlerinde olduğu gibi, her bir özellik (trait) için de dokümantasyonu gözden geçirmeye zaman ayırın.

Bu bölüm uzun. Ortasında bir mola verin.

## 18.1 Karşılaştırmalar

Bu özellikler (traits), değerler arasında karşılaştırmaları destekler. Tüm özellikler, bu özellikleri gerçekleştiren (implement) alanlar içeren türler için türetilebilir.

### `PartialEq` ve `Eq`

`PartialEq`, gerekli metot `eq` ve sağlanan metot `ne` ile kısmi bir denklik ilişkisidir. `==` ve `!=` operatörleri bu metotları çağırır.

```
struct Key {
    id: u32,
    metadata: Option<String>,
}
```

```
impl PartialEq for Key {
    fn eq(&self, other: &Self) -> bool {
        self.id == other.id
    }
}
```

Eq tam bir denklik ilişkisidir (yansımalı, simetrik ve geçişme) (reflexive, symmetric, and transitive) ve `PartialEq` özelliğini kapsar (imply). Tam denklik gerektiren fonksiyonlar, bir özellik sınırı (trait bound) olarak `Eq` kullanır.

### `PartialOrd` ve `Ord`

`PartialOrd`, `partial_cmp` metoduyla kısmi bir sıralama tanımlar. `<`, `<=`, `>=` ve `>` operatörlerini gerçekleştirmek (implement) için kullanılır.

```
use std::cmp::Ordering;
#[derive(Eq, PartialEq)]
struct Citation {
    author: String,
    year: u32,
}
impl PartialOrd for Citation {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        match self.author.partial_cmp(&other.author) {
            Some(Ordering::Equal) => self.year.partial_cmp(&other.year),
            author_ord => author_ord,
        }
    }
}
```

`Ord`, `cmp`'nin `Ordering` geri döndürdüğü tam bir sıralamadır.

This slide should take about 5 minutes.

- `PartialEq` farklı türler arasında gerçekleştirilebilir (implement), ancak `Eq` yansımalı (reflexive) olduğu için gerçekleştirilemez:

  ```
  struct Key {
      id: u32,
      metadata: Option<String>,
  }
  impl PartialEq<u32> for Key {
      fn eq(&self, other: &u32) -> bool {
          self.id == *other
      }
  }
  ```

- Pratikte bu özellikleri (traits) türetmek (derive) yaygındır, ancak onları gerçekleştirmek (implement) yaygın değildir.
- Rust'ta referansları karşılaştırırken, gösterdikleri şeylerin değerini karşılaştırır, referansların kendilerini DEĞİL. Bu, gösterilen değerler aynıysa, iki farklı şeye yapılan referansların eşit olarak karşılaştırılabileceği anlamına gelir:

```
    fn main() {
        let a = "Merhaba";
        let b = String::from("Merhaba");
        assert_eq!(a, b);
    }
```

## 18.2 Operatörler

Operatör yüklemesi (overloading), `std::ops`'deki özellikler (traits) aracılığıyla gerçekleştirilir (implement):

```
#[derive(Debug, Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

impl std::ops::Add for Point {
    type Output = Self;

    fn add(self, other: Self) -> Self {
        Self { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Point { x: 10, y: 20 };
    let p2 = Point { x: 100, y: 200 };
    println!("{p1:?} + {p2:?} = {:?}", p1 + p2);
}
```

This slide should take about 5 minutes.

Tartışma noktaları:

- `&Point` için `Add`'i gerçekleştirebilirsiniz (implement). Bu hangi durumlarda kullanışlıdır?
  - Cevap: `Add:add`, `self`'i tüketir. Operatörü yüklediğiniz `T` türü `Copy` değilse, operatörü `&T` için de yüklemey yapmayı (overloading) düşünmelisiniz. Bu, çağrı yerleride (call site) gereksiz klonlamayı önler.
- `Output` neden bir ilişkili türdür (associated type)? Metodun bir tür parametresi yapılabilir miydi?
  - Kısa cevap: Fonksiyon tür parametreleri çağıran (caller) tarafından kontrol edilir, ancak ilişkili türler (`Output` gibi) bir özelliğin (trait) gerçekleştiricisi (implementer) tarafından kontrol edilir.
- İki farklı tür için `Add` gerçekleştirebilirsiniz (implement), örn. `impl Add<(i32, i32)> for Point`, bu bir `Point`'e bir demet (tuple) eklerdi.

`Not` özelliği (`!` operatörü), C ailesi dillerindeki aynı operatör gibi "boole'laştırmadığı" için dikkat çekicidir; bunun yerine, tamsayı türleri için sayının her bitini olumsuzlar, bu da aritmetik olarak onu -1'den çıkarmaya eşdeğerdir: `!5 == -6`.

## 18.3 `From` ve `Into`

Türler, tür dönüşümlerini kolaylaştırmak için `From` ve `Into` özelliklerini gerçekleştirir (implement). `as`'in aksine, bu özellikler (traits) kayıpsız, hatasız dönüşümlere karşılık gelir.

```
fn main() {
    let s = String::from("merhaba");
    let addr = std::net::Ipv4Addr::from([127, 0, 0, 1]);
    let one = i16::from(true);
    let bigger = i32::from(123_i16);
    println!("{s}, {addr}, {one}, {bigger}");
}
```

`From` gerçekleştirildiğinde (implement) `Into` otomatik olarak gerçekleştirilir:

```
fn main() {
    let s: String = "merhaba".into();
    let addr: std::net::Ipv4Addr = [127, 0, 0, 1].into();
    let one: i16 = true.into();
    let bigger: i32 = 123_i16.into();
    println!("{s}, {addr}, {one}, {bigger}");
}
```

This slide should take about 5 minutes.

- Bu yüzden sadece `From`'u gerçekleştirmek (implement) yaygındır, çünkü türünüz `Into` gerçekleştirmesini de alacaktır.
- "Bir `String`'e dönüştürülebilen herhangi bir şey" gibi bir fonksiyon argümanı girdi türü bildirirken, kural tam tersidir, `Into` kullanmalısınız. Fonksiyonunuz `From`'u gerçekleştiren (implement) türleri ve *sadece* `Into`'yu gerçekleştirenleri kabul edecektir.

## 18.4 Tür Dönüştürme (Casting)

Rust'ta *örtük (implicit)* tür dönüşümleri yoktur, ancak `as` ile açık (explicit) tür dönüştürmelerini (casts) destekler. Bunlar genellikle tanımlandıkları yerlerde C semantiğini takip eder.

```
fn main() {
    let value: i64 = 1000;
    println!("u16 olarak: {}", value as u16);
    println!("i16 olarak: {}", value as i16);
    println!("u8 olarak: {}", value as u8);
}
```

`as`'in sonuçları Rust'ta *her zaman* tanımlıdır ve platformlar arasında tutarlıdır. Bu, işaret (sign) değiştirme veya daha küçük bir türe dönüştürme (cast) konusundaki sezginizle uyuşmayabilir -- belgeleri kontrol edin ve açıklık için yorum yapın.

`as` ile dönüştürme (cast), yanlış kullanılması kolay olan nispeten keskin bir araçtır ve gelecekteki bakım çalışmaları (maintenance work) kullanılan türleri veya türlerdeki değer aralıklarını değiştirdikçe ince hataların kaynağı olabilir. Tür dnüştürmesi en iyi, niyetin koşulsuz olarak kesmeyi belirtmek olduğu durumlarda kullanılır (örneğin, yüksek bitlerde ne olursa olsun bir u64'ün alt 32 bitini `as u32` ile seçmek).

Hatasız tür dönüştürmeleri (cast) için (örneğin u32'den u64'e), dönüşümün gerçekten hatasız olduğunu doğrulamak için `as` yerine `From` veya `Into` kullanmayı tercih edin. Hatalı olabilecek dönüştürmeler (cast) için, uyan ve uymayan dnüştürmeleri (cast) ele almak istediğinizde `TryFrom` ve `TryInto` mevcuttur.

This slide should take about 5 minutes.

Bu slayttan sonra bir mola vermeyi düşünün.

`as`, C++'daki bir statik dönüştürmeye (static cast) benzer. Veri kaybı olabilecek durumlarda `as` kullanımı genellikle önerilmez veya en azından açıklayıcı bir yorumu hak eder.

Bu, tamsayıları bir indeks olarak kullanmak üzere `usize`'a dönüştürürken yaygındır.

## 18.5 `Read` ve `Write`

`Read` ve `BufRead` kullanarak, u8 kaynakları üzerinde soyutlama yapabilirsiniz:

```
use std::io::{BufRead, BufReader, Read, Result};

fn count_lines<R: Read>(reader: R) -> usize {
    let buf_reader = BufReader::new(reader);
    buf_reader.lines().count()
}

fn main() -> Result<()> {
    let slice: &[u8] = b"foo\nbar\nbaz\n";
    println!("dilimdeki (slice) satırlar: {}", count_lines(slice));

    let file = std::fs::File::open(std::env::current_exe()?)?;
    println!("dosyadaki satırlar: {}", count_lines(file));
    Ok(())
}
```

Benzer şekilde, `Write` u8 alıcıları (sinks) üzerinde soyutlama yapmanızı sağlar:

```
use std::io::{Result, Write};

fn log<W: Write>(writer: &mut W, msg: &str) -> Result<()> {
    writer.write_all(msg.as_bytes())?;
    writer.write_all("\n".as_bytes())
}

fn main() -> Result<()> {
    let mut buffer = Vec::new();
    log(&mut buffer, "Merhaba")?;
    log(&mut buffer, "Dünya")?;
    println!("Kaydedildi: {buffer:?}");
    Ok(())
}
```

## 18.6 `Default` Özelliği (Trait)

`Default` özelliği (trait), bir tür için varsayılan bir değer üretir.

```
#[derive(Debug, Default)]
struct Derived {
    x: u32,
    y: String,
    z: Implemented,
}

#[derive(Debug)]
struct Implemented(String);

impl Default for Implemented {
    fn default() -> Self {
        Self("Evliya Çelebi".into())
    }
}

fn main() {
    let default_struct = Derived::default();
    dbg!(default_struct);

    let almost_default_struct =
        Derived { y: "Y ayarlandı!".into(), ..Derived::default() };
    dbg!(almost_default_struct);

    let nothing: Option<Derived> = None;
    dbg!(nothing.unwrap_or_default());
}
```

This slide should take about 5 minutes.

- Doğrudan gerçekleştirilebilir (implement) veya `#[derive(Default)]` aracılığıyla türetilebilir.
- Türetilmiş bir gerçekleştirme (implementation), tüm alanların varsayılan değerlerine ayarlandığı bir değer üretecektir.
  - Bu, yapıdaki (struct) tüm türlerin de `Default`'u gerçekleştirmesi (implement) gerektiği anlamına gelir.
- Standart Rust türleri genellikle `Default`'u makul değerlerle (ör. `0`, `""`, vb.) gerçekleştirir (implement).
- Kısmi yapı ilklendirmesi (partial struct initialization), default özelliği ile güzel çalışır.
- Rust standart kütüphanesi, türlerin `Default`'u gerçekleştirebileceğinin (implement) farkındadır ve onu kullanan kullanışlı metotlar sağlar.
- `..` sözdizimine yapı güncelleme sözdizimi (struct update syntax) denir.

## 18.7 Alıştırma: ROT13

Bu örnekte, klasik "ROT13" şifresini gerçekleştireceksiniz (implement). Bu kodu deneme alanına (playground) kopyalayın ve eksik kısımları gerçekleştirin. Sonucun hala geçerli UTF-8

olduğundan emin olmak için yalnızca ASCII alfabetik karakterleri döndürün.

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

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
}

// `RotDecoder` için `Read` özelliğini (trait) gerçekleştirin.

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn joke() {
        let mut rot =
            RotDecoder { input: "B gnensn ineznx tnlrfvlyr!".as_bytes(), rot: 13 };
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "O tarafa varmak gayesiyle!");
    }

    #[test]
    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13 };
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

Her biri 13 karakter kaydırma (ROT13) yapan iki adet RotDecoder örneğini ardı ardına (biri diğerinin çıktısını okuyarak) çalıştırırsan ne olur?

### 18.7.1 Çözüm

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

struct RotDecoder<R: Read> {
    input: R,
    rot: u8,
}

impl<R: Read> Read for RotDecoder<R> {
```

```
    fn read(&mut self, buf: &mut [u8]) -> std::io::Result<usize> {
        let size = self.input.read(buf)?;
        for b in &mut buf[..size] {
            if b.is_ascii_alphabetic() {
                let base = if b.is_ascii_uppercase() { 'A' } else { 'a' } as u8;
                *b = (*b - base + self.rot) % 26 + base;
            }
        }
        Ok(size)
    }
}

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn joke() {
        let mut rot =
            RotDecoder { input: "B gnensn ineznx tnlrfvlyr!".as_bytes(), rot: 13 };
        let mut result = String::new();
        rot.read_to_string(&mut result).unwrap();
        assert_eq!(&result, "O tarafa varmak gayesiyle!");
    }

    #[test]
    fn binary() {
        let input: Vec<u8> = (0..=255u8).collect();
        let mut rot = RotDecoder::<&[u8]> { input: input.as_slice(), rot: 13 };
        let mut buf = [0u8; 256];
        assert_eq!(rot.read(&mut buf).unwrap(), 256);
        for i in 0..=255 {
            if input[i] != buf[i] {
                assert!(input[i].is_ascii_alphabetic());
                assert!(buf[i].is_ascii_alphabetic());
            }
        }
    }
}
```

# Part V

# Gün 3: Sabah

# Chapter 19

# Welcome to Day 3

Today, we will cover:

- Memory management, lifetimes, and the borrow checker: how Rust ensures memory safety.
- Smart pointers: standard library pointer types.

## Zamanlama (Schedule)

Bu oturum 10 dakikalık aralar dahil yaklaşık 2 saat 20 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 3 dakika |
| Bellek Yönetimi | 1 saat |
| Akıllı Göstericiler | 55 dakika |

# Chapter 20

# Bellek Yönetimi

Bu bölüm yaklaşık 1 saat sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Program Belleğinin Gözden Geçirilmesi | 5 dakika |
| Bellek Yönetimine Yaklaşımlar | 10 dakika |
| Sahiplik | 5 dakika |
| Taşıma Semantiği | 5 dakika |
| Clone | 2 dakika |
| Kopyalanan Türler | 5 dakika |
| Drop | 10 dakika |
| Alıştırma: İnşa Edici / Yapıcı Tür (Builder Type) | 20 dakika |

## 20.1 Program Belleğinin Gözden Geçirilmesi

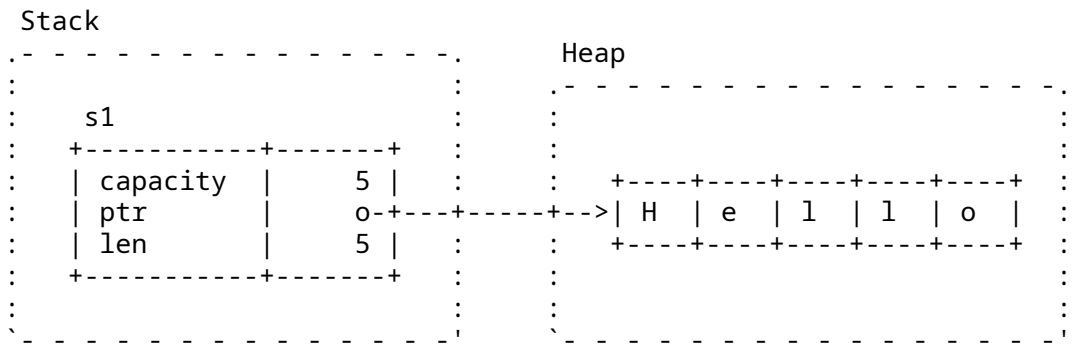
Programs allocate memory in two ways:

- Stack: Continuous area of memory for local variables.
  - Values have fixed sizes known at compile time.
  - Extremely fast: just move a stack pointer.
  - Easy to manage: follows function calls.
  - Great memory locality.
- Heap: Storage of values outside of function calls.
  - Values have dynamic sizes determined at runtime.
  - Slightly slower than the stack: some book-keeping needed.
  - No guarantee of memory locality.

### Örnek

Creating a `String` puts fixed-sized metadata on the stack and dynamically sized data, the actual string, on the heap:

```rust
fn main() {
    let s1 = String::from("Merhaba");
}
```




This slide should take about 5 minutes.

- Mention that a `String` is backed by a `Vec`, so it has a capacity and length and can grow if mutable via reallocation on the heap.
- If students ask about it, you can mention that the underlying memory is heap allocated using the System Allocator and custom allocators can be implemented using the Allocator API

### Daha Fazlasını Keşfedin

We can inspect the memory layout with `unsafe` Rust. However, you should point out that this is rightfully unsafe!

```rust
fn main() {
    let mut s1 = String::from("Merhaba");
    s1.push(' ');
    s1.push_str("dünya");
    // DON'T DO THIS AT HOME! For educational purposes only.
    // String provides no guarantees about its layout, so this could lead to
    // undefined behavior.
    unsafe {
        let (capacity, ptr, len): (usize, usize, usize) = std::mem::transmute(s1);
        println!("capacity = {capacity}, ptr = {ptr:#x}, len = {len}");
    }
}
```

## 20.2 Bellek Yönetimine Yaklaşımlar

Traditionally, languages have fallen into two broad categories:

- Full control via manual memory management: C, C++, Pascal, ...
  - Programmer decides when to allocate or free heap memory.
  - Programmer must determine whether a pointer still points to valid memory.
  - Studies show, programmers make mistakes.
- Full safety via automatic memory management at runtime: Java, Python, Go, Haskell, ...

  - A runtime system ensures that memory is not freed until it can no longer be referenced.
  - Genellikle referans sayımı veya çöp toplama ile uygulanır.

Rust offers a new mix:

> Full control *and* safety via compile time enforcement of correct memory management.

It does this with an explicit ownership concept.

This slide should take about 10 minutes.

This slide is intended to help students coming from other languages to put Rust in context.

- C must manage heap manually with `malloc` and `free`. Common errors include forgetting to call `free`, calling it multiple times for the same pointer, or dereferencing a pointer after the memory it points to has been freed.
- C++ has tools like smart pointers (`unique_ptr`, `shared_ptr`) that take advantage of language guarantees about calling destructors to ensure memory is freed when a function returns. It is still quite easy to mis-use these tools and create similar bugs to C.
- Java, Go, and Python rely on the garbage collector to identify memory that is no longer reachable and discard it. This guarantees that any pointer can be dereferenced, eliminating use-after-free and other classes of bugs. But, GC has a runtime cost and is difficult to tune properly.

Rust's ownership and borrowing model can, in many cases, get the performance of C, with alloc and free operations precisely where they are required -- zero cost. It also provides tools similar to C++'s smart pointers. When required, other options such as reference counting are available, and there are even crates available to support runtime garbage collection (not covered in this class).

## 20.3 Sahiplik

All variable bindings have a *scope* where they are valid and it is an error to use a variable outside its scope:

```
struct Point(i32, i32);

fn main() {
    {
        let p = Point(3, 4);
        dbg!(p.0);
    }
    dbg!(p.1);
}
```

We say that the variable *owns* the value. Every Rust value has precisely one owner at all times.

At the end of the scope, the variable is *dropped* and the data is freed. A destructor can run here to free up resources.

This slide should take about 5 minutes.

Students familiar with garbage-collection implementations will know that a garbage collector starts with a set of ”roots” to find all reachable memory. Rust's ”single owner” principle is a similar idea.

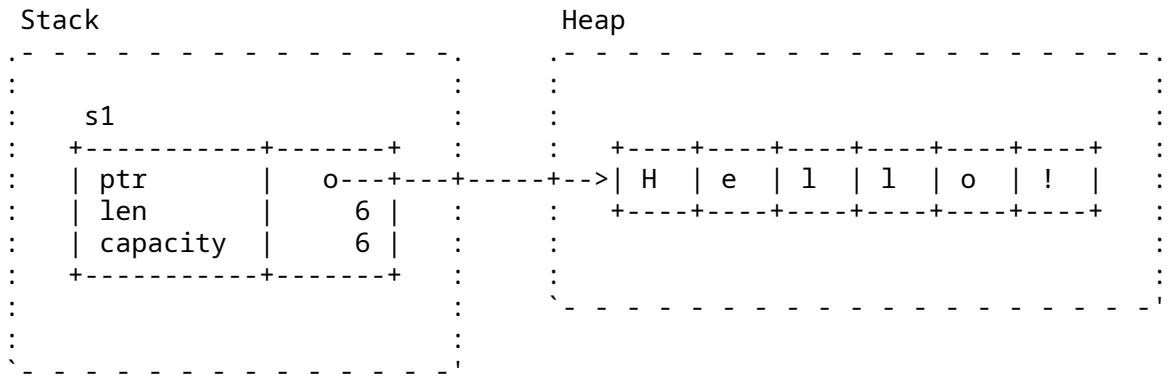
## 20.4 Taşıma Semantiği

An assignment will transfer *ownership* between variables:

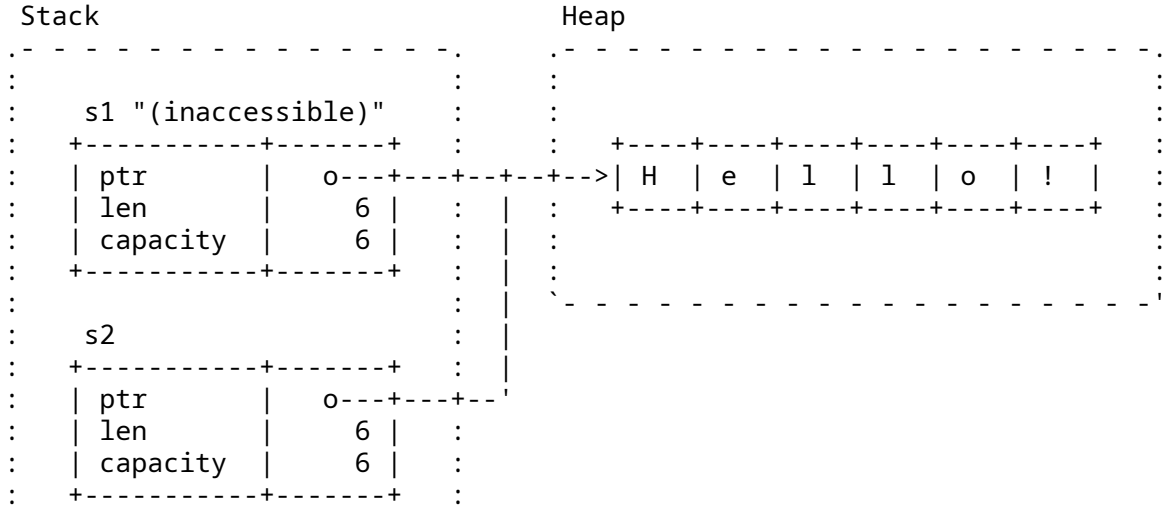
```
fn main() {
    let s1 = String::from("Merhaba!");
    let s2 = s1;
    dbg!(s2);
    // dbg!(s1);
}
```

- The assignment of `s1` to `s2` transfers ownership.
- When `s1` goes out of scope, nothing happens: it does not own anything.
- When `s2` goes out of scope, the string data is freed.

Before move to `s2`:




After move to `s2`:



When you pass a value to a function, the value is assigned to the function parameter. This transfers ownership:

```
fn say_hello(name: String) {
    println!("Merhaba {name}")
}

fn main() {
    let name = String::from("Alice");
    say_hello(name);
    // say_hello(name);
}
```

This slide should take about 5 minutes.

- Mention that this is the opposite of the defaults in C++, which copies by value unless you use `std::move` (and the move constructor is defined!).
- It is only the ownership that moves. Whether any machine code is generated to manipulate the data itself is a matter of optimization, and such copies are aggressively optimized away.
- Simple values (such as integers) can be marked `Copy` (see later slides).
- In Rust, clones are explicit (by using `clone`).

In the `say_hello` example:

- With the first call to `say_hello`, `main` gives up ownership of `name`. Afterwards, `name` cannot be used anymore within `main`.
- The heap memory allocated for `name` will be freed at the end of the `say_hello` function.
- `main` can retain ownership if it passes `name` as a reference (`&name`) and if `say_hello` accepts a reference as a parameter.
- Alternatively, `main` can pass a clone of `name` in the first call (`name.clone()`).
- Rust makes it harder than C++ to inadvertently create copies by making move semantics the default, and by forcing programmers to make clones explicit.

# Daha Fazlasını Keşfedin

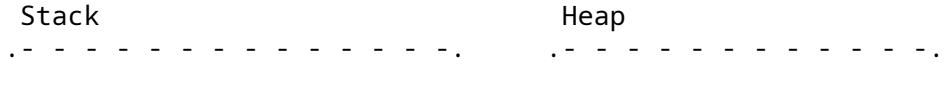
## Defensive Copies in Modern C++

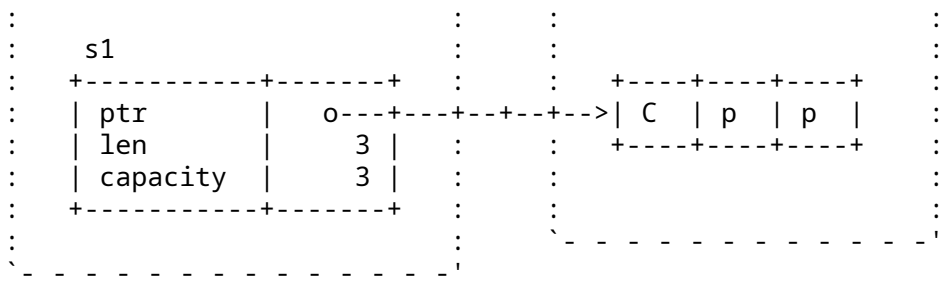
Modern C++ solves this differently:

```
std::string s1 = "Cpp";
std::string s2 = s1;  // Duplicate the data in s1.
```

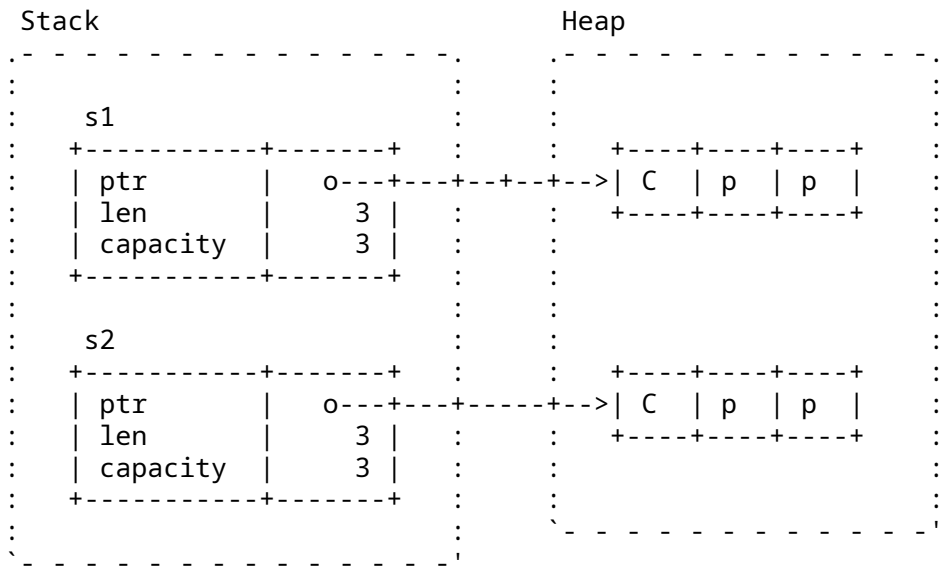
- The heap data from `s1` is duplicated and `s2` gets its own independent copy.
- When `s1` and `s2` go out of scope, they each free their own memory.

Before copy-assignment:






After copy-assignment:




Anahtar noktalar:

- C++ has made a slightly different choice than Rust. Because = copies data, the string data has to be cloned. Otherwise we would get a double-free when either string goes out of scope.
- C++ also has `std::move`, which is used to indicate when a value may be moved from. If the example had been `s2 = std::move(s1)`, no heap allocation would take place. After the move, `s1` would be in a valid but unspecified state. Unlike Rust, the programmer is allowed to keep using `s1`.
- Unlike Rust, = in C++ can run arbitrary code as determined by the type which is being copied or moved.

## 20.5 Clone

Sometimes you *want* to make a copy of a value. The `Clone` trait accomplishes this.

```
fn say_hello(name: String) {
    println!("Merhaba {name}")
}
```

```
fn main() {
    let name = String::from("Alice");
    say_hello(name.clone());
    say_hello(name);
}
```

This slide should take about 2 minutes.

- The idea of `Clone` is to make it easy to spot where heap allocations are occurring. Look for `.clone()` and a few others like `vec!` or `Box::new`.
- It's common to ”clone your way out” of problems with the borrow checker, and return later to try to optimize those clones away.
- `clone` generally performs a deep copy of the value, meaning that if you e.g. clone an array, all of the elements of the array are cloned as well.
- The behavior for `clone` is user-defined, so it can perform custom cloning logic if needed.

## 20.6 Kopyalanan Türler

While move semantics are the default, certain types are copied by default:

```
fn main() {
    let x = 42;
    let y = x;
    dbg!(x); // would not be accessible if not Copy
    dbg!(y);
}
```

These types implement the `Copy` trait.

You can opt-in your own types to use copy semantics:

```
#[derive(Copy, Clone, Debug)]
struct Point(i32, i32);

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;
    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

- After the assignment, both `p1` and `p2` own their own data.
- We can also use `p1.clone()` to explicitly copy the data.

This slide should take about 5 minutes.

Copying and cloning are not the same thing:

- Copying refers to bitwise copies of memory regions and does not work on arbitrary objects.
- Copying does not allow for custom logic (unlike copy constructors in C++).
- Cloning is a more general operation and also allows for custom behavior by implementing the `Clone` trait.

- Copying does not work on types that implement the `Drop` trait.

In the above example, try the following:

- Add a `String` field to `struct Point`. It will not compile because `String` is not a `Copy` type.
- Remove `Copy` from the `derive` attribute. The compiler error is now in the `println!` for `p1`.
- Show that it works if you clone `p1` instead.

## Daha Fazlasını Keşfedin

- Shared references are `Copy`/`Clone`, mutable references are not. This is because Rust requires that mutable references be exclusive, so while it's valid to make a copy of a shared reference, creating a copy of a mutable reference would violate Rust's borrowing rules.

## 20.7 The Drop Trait

Values which implement `Drop` can specify code to run when they go out of scope:

```
struct Droppable {
    name: &'static str,
}

impl Drop for Droppable {
    fn drop(&mut self) {
        println!("Dropping {}", self.name);
    }
}

fn main() {
    let a = Droppable { name: "a" };
    {
        let b = Droppable { name: "b" };
        {
            let c = Droppable { name: "c" };
            let d = Droppable { name: "d" };
            println!("Exiting innermost block");
        }
        println!("Exiting next block");
    }
    drop(a);
    println!("Exiting main");
}
```

This slide should take about 8 minutes.

- Note that `std::mem::drop` is not the same as `std::ops::Drop::drop`.
- Values are automatically dropped when they go out of scope.

- When a value is dropped, if it implements `std::ops::Drop` then its `Drop::drop` implementation will be called.
- All its fields will then be dropped too, whether or not it implements `Drop`.
- `std::mem::drop` is just an empty function that takes any value. The significance is that it takes ownership of the value, so at the end of its scope it gets dropped. This makes it a convenient way to explicitly drop values earlier than they would otherwise go out of scope.
  - This can be useful for objects that do some work on `drop`: releasing locks, closing files, etc.

Tartışma noktaları:

- Why doesn't `Drop::drop` take `self`?
  - Short-answer: If it did, `std::mem::drop` would be called at the end of the block, resulting in another call to `Drop::drop`, and a stack overflow!
- Try replacing `drop(a)` with `a.drop()`.

## 20.8 Alıştırma: İnşa Edici / Yapıcı Tür (Builder Type)

In this example, we will implement a complex data type that owns all of its data. We will use the ”builder pattern” to support building a new value piece-by-piece, using convenience functions.

Fill in the missing pieces.

```
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

#[derive(Clone, Debug)]
struct Dependency {
    name: String,
    version_expression: String,
}

/// A representation of a software package.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Return a representation of this package as a dependency, for use in
    /// building other packages.
    fn as_dependency(&self) -> Dependency {
```

```
        todo!("1")
    }
}

/// A builder for a Package. Use `build()` to create the `Package` itself.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        todo!("2")
    }

    /// Set the package version.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// Set the package authors.
    fn authors(mut self, authors: Vec<String>) -> Self {
        todo!("3")
    }

    /// Add an additional dependency.
    fn dependency(mut self, dependency: Dependency) -> Self {
        todo!("4")
    }

    /// Set the language. If not set, language defaults to None.
    fn language(mut self, language: Language) -> Self {
        todo!("5")
    }

    fn build(self) -> Package {
        self.0
    }
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    dbg!(&base64);
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
    dbg!(&log);
    let serde = PackageBuilder::new("serde")
        .authors(vec!["djmitche".into()])
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    dbg!(serde);
```

```
}
```

### 20.8.1 Çözüm

```
#[derive(Debug)]
enum Language {
    Rust,
    Java,
    Perl,
}

#[derive(Clone, Debug)]
struct Dependency {
    name: String,
    version_expression: String,
}

/// A representation of a software package.
#[derive(Debug)]
struct Package {
    name: String,
    version: String,
    authors: Vec<String>,
    dependencies: Vec<Dependency>,
    language: Option<Language>,
}

impl Package {
    /// Return a representation of this package as a dependency, for use in
    /// building other packages.
    fn as_dependency(&self) -> Dependency {
        Dependency {
            name: self.name.clone(),
            version_expression: self.version.clone(),
        }
    }
}

/// A builder for a Package. Use `build()` to create the `Package` itself.
struct PackageBuilder(Package);

impl PackageBuilder {
    fn new(name: impl Into<String>) -> Self {
        Self(Package {
            name: name.into(),
            version: "0.1".into(),
            authors: Vec::new(),
            dependencies: Vec::new(),
            language: None,
        })
    }
```

```
    /// Set the package version.
    fn version(mut self, version: impl Into<String>) -> Self {
        self.0.version = version.into();
        self
    }

    /// Set the package authors.
    fn authors(mut self, authors: Vec<String>) -> Self {
        self.0.authors = authors;
        self
    }

    /// Add an additional dependency.
    fn dependency(mut self, dependency: Dependency) -> Self {
        self.0.dependencies.push(dependency);
        self
    }

    /// Set the language. If not set, language defaults to None.
    fn language(mut self, language: Language) -> Self {
        self.0.language = Some(language);
        self
    }

    fn build(self) -> Package {
        self.0
    }
}

fn main() {
    let base64 = PackageBuilder::new("base64").version("0.13").build();
    dbg!(&base64);
    let log =
        PackageBuilder::new("log").version("0.4").language(Language::Rust).build();
    dbg!(&log);
    let serde = PackageBuilder::new("serde")
        .authors(vec!["djmitche".into()])
        .version(String::from("4.0"))
        .dependency(base64.as_dependency())
        .dependency(log.as_dependency())
        .build();
    dbg!(serde);
}
```

# Chapter 21

# Akıllı Göstericiler

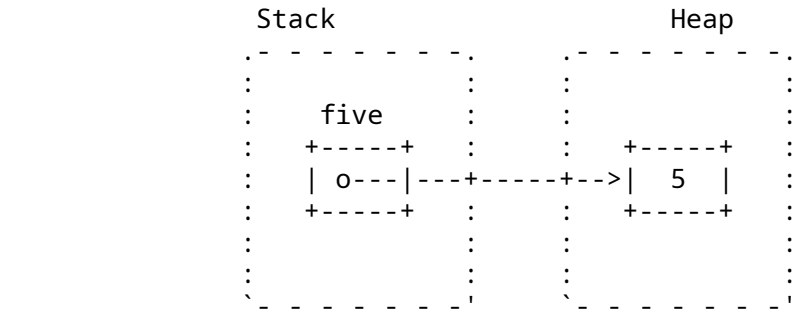
Bu bölüm yaklaşık 55 dakika sürmelidir. İçerir:

| Slayt | Süre |
|---|---|
| Box | 10 dakika |
| Rc | 5 dakika |
| Sahipli Özellik Nesneleri (Owned Trait Objects) | 10 dakika |
| Alıştırma: İkili Ağaç | 30 dakika |

## 21.1 Box<T>

Box is an owned pointer to data on the heap:

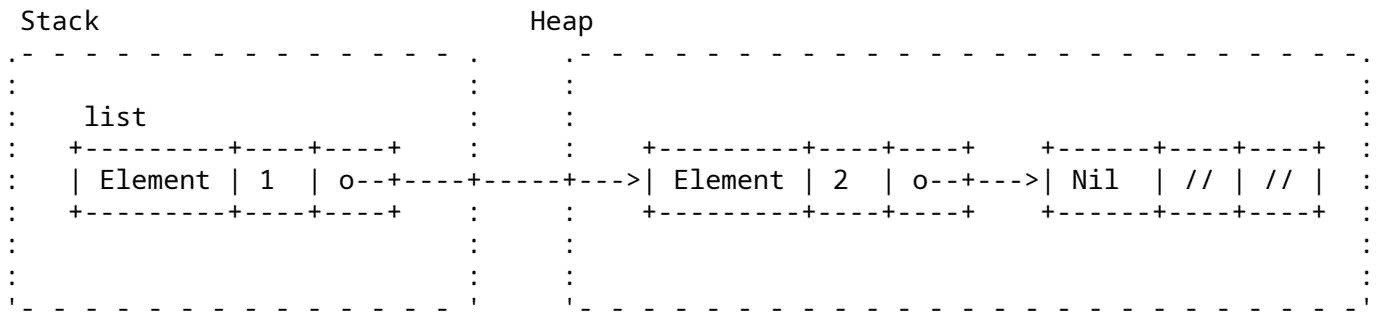
```
fn main() {
    let five = Box::new(5);
    println!("five: {}", *five);
}
```




`Box<T>` implements `Deref<Target = T>`, which means that you can call methods from `T` directly on a `Box<T>`.

Recursive data types or data types with dynamic sizes cannot be stored inline without a pointer indirection. `Box` accomplishes that indirection:

```rust
#[derive(Debug)]
enum List<T> {
    /// A non-empty list: first element and the rest of the list.
    Element(T, Box<List<T>>),
    /// An empty list.
    Nil,
}

fn main() {
    let list: List<i32> =
        List::Element(1, Box::new(List::Element(2, Box::new(List::Nil))));
    println!("{list:?}");
}
```




This slide should take about 8 minutes.

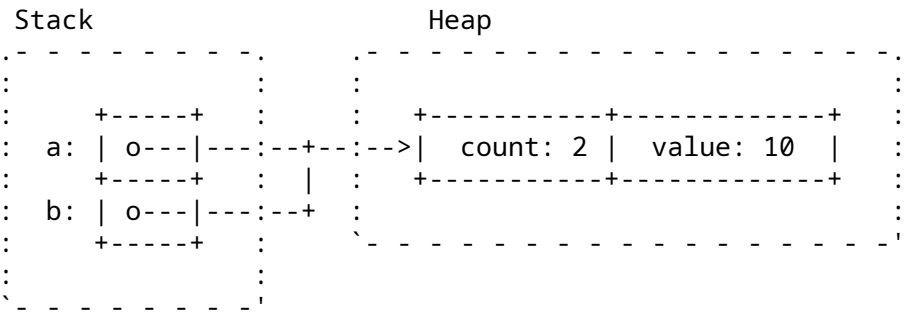
- Box is like `std::unique_ptr` in C++, except that it's guaranteed to be not null.
- A Box can be useful when you:
  - have a type whose size can't be known at compile time, but the Rust compiler wants to know an exact size.
  - want to transfer ownership of a large amount of data. To avoid copying large amounts of data on the stack, instead store the data on the heap in a Box so only the pointer is moved.
- If Box was not used and we attempted to embed a `List` directly into the `List`, the compiler would not be able to compute a fixed size for the struct in memory (the `List` would be of infinite size).
- Box solves this problem as it has the same size as a regular pointer and just points at the next element of the `List` in the heap.
- Remove the Box in the List definition and show the compiler error. We get the message ”recursive without indirection”, because for data recursion, we have to use indirection, a Box or reference of some kind, instead of storing the value directly.
- Though Box looks like `std::unique_ptr` in C++, it cannot be empty/null. This makes Box one of the types that allow the compiler to optimize storage of some enums (the ”niche optimization”).

## 21.2 Rc

Rc is a reference-counted shared pointer. Use this when you need to refer to the same data from multiple places:

```
use std::rc::Rc;

fn main() {
    let a = Rc::new(10);
    let b = Rc::clone(&a);

    dbg!(a);
    dbg!(b);
}
```

Each Rc points to the same shared data structure, containing strong and weak pointers and the value:




- See Arc and Mutex if you are in a multi-threaded context.
- You can *downgrade* a shared pointer into a Weak pointer to create cycles that will get dropped.

This slide should take about 5 minutes.

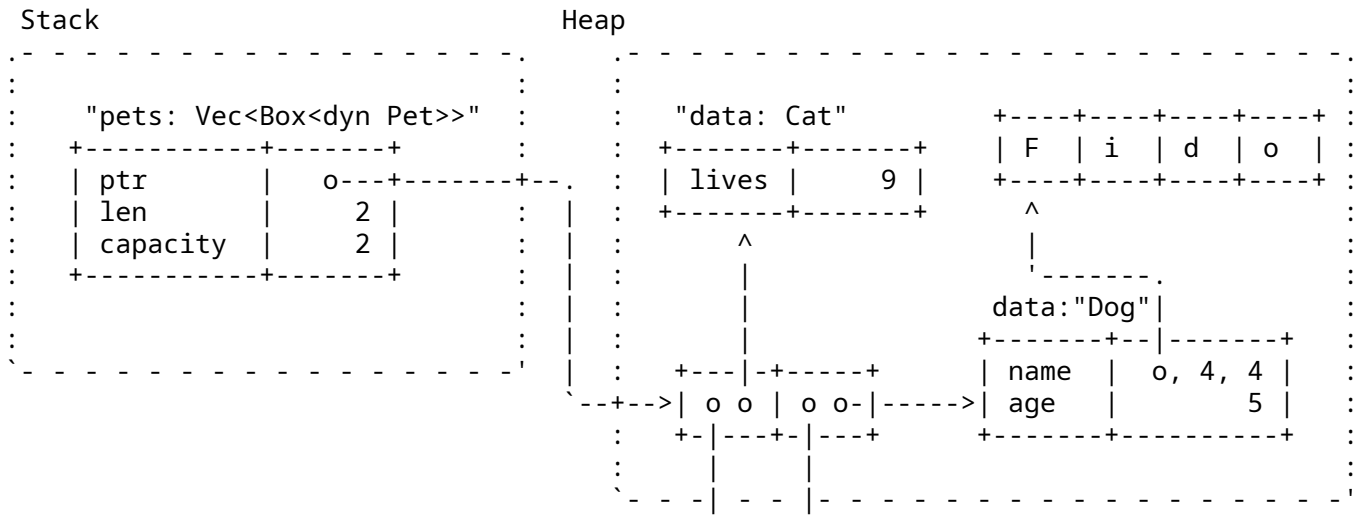
- Rc's count ensures that its contained value is valid for as long as there are references.
- Rc in Rust is like std::shared_ptr in C++.
- Rc::clone is cheap: it creates a pointer to the same allocation and increases the reference count. Does not make a deep clone and can generally be ignored when looking for performance issues in code.
- make_mut actually clones the inner value if necessary ("clone-on-write") and returns a mutable reference.
- Use Rc::strong_count to check the reference count.
- Rc::downgrade gives you a *weakly reference-counted* object to create cycles that will be dropped properly (likely in combination with RefCell).

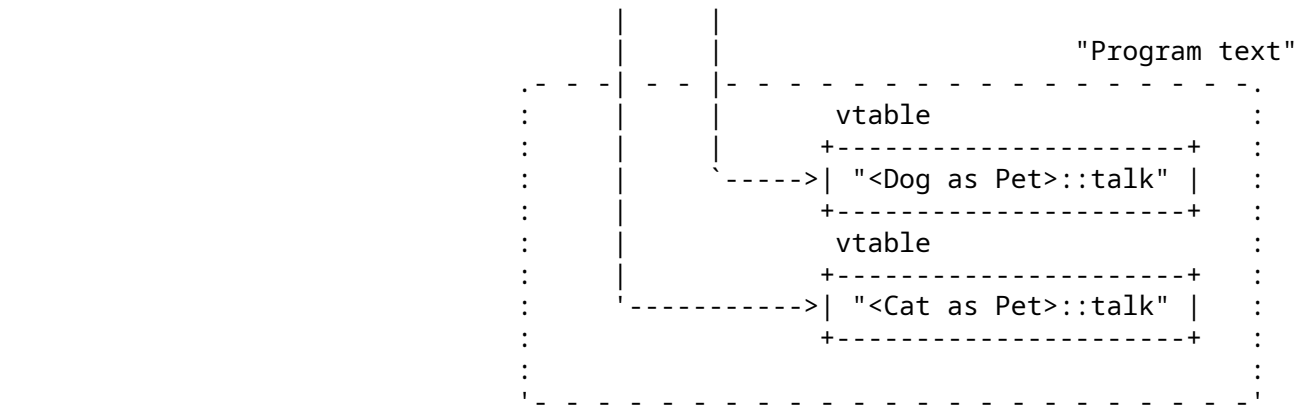
## 21.3 Sahipli Özellik Nesneleri (Owned Trait Objects)

We previously saw how trait objects can be used with references, e.g &dyn Pet. However, we can also use trait objects with smart pointers like Box to create an owned trait object: Box<dyn Pet>.

```rust
struct Dog {
    name: String,
    age: i8,
}
struct Cat {
    lives: i8,
}

trait Pet {
    fn talk(&self) -> String;
}

impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Hav, benim adım {}!", self.name)
    }
}

impl Pet for Cat {
    fn talk(&self) -> String {
        String::from("Miyav!")
    }
}

fn main() {
    let pets: Vec<Box<dyn Pet>> = vec![
        Box::new(Cat { lives: 9 }),
        Box::new(Dog { name: String::from("Fido"), age: 5 }),
    ];
    for pet in pets {
        println!("Merhaba, nasılsın? {}", pet.talk());
    }
}
```

Memory layout after allocating `pets`:






This slide should take about 10 minutes.

- Types that implement a given trait may be of different sizes. This makes it impossible to have things like `Vec<dyn Pet>` in the example above.
- `dyn Pet` is a way to tell the compiler about a dynamically sized type that implements `Pet`.
- In the example, `pets` is allocated on the stack and the vector data is on the heap. The two vector elements are *fat pointers*:
  - A fat pointer is a double-width pointer. It has two components: a pointer to the actual object and a pointer to the virtual method table (vtable) for the `Pet` implementation of that particular object.
  - The data for the `Dog` named Fido is the `name` and `age` fields. The `Cat` has a `lives` field.
- Compare these outputs in the above example:

```
println!("{} {}", std::mem::size_of::<Dog>(), std::mem::size_of::<Cat>());
println!("{} {}", std::mem::size_of::<&Dog>(), std::mem::size_of::<&Cat>());
println!("{}", std::mem::size_of::<&dyn Pet>());
println!("{}", std::mem::size_of::<Box<dyn Pet>>());
```

## 21.4 Alıştırma: İkili Ağaç

A binary tree is a tree-type data structure where every node has two children (left and right). We will create a tree where each node stores a value. For a given node N, all nodes in a N's left subtree contain smaller values, and all nodes in N's right subtree will contain larger values. A given value should only be stored in the tree once, i.e. no duplicate nodes.

Implement the following types, so that the given tests pass.

```
/// A node in the binary tree.
#[derive(Debug)]
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// A possibly-empty subtree.
#[derive(Debug)]
```

```rust
struct Subtree<T: Ord>(Option<Box<Node<T>>>);

/// A container storing a set of values, using a binary tree.
///
/// If the same value is added multiple times, it is only stored once.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}

impl<T: Ord> BinaryTree<T> {
    fn new() -> Self {
        Self { root: Subtree::new() }
    }

    fn insert(&mut self, value: T) {
        self.root.insert(value);
    }

    fn has(&self, value: &T) -> bool {
        self.root.has(value)
    }

    fn len(&self) -> usize {
        self.root.len()
    }
}

// Implement `new`, `insert`, `len`, and `has` for `Subtree`.

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
        tree.insert(2); // not a unique item
        assert_eq!(tree.len(), 2);
        tree.insert(3);
        assert_eq!(tree.len(), 3);
    }

    #[test]
    fn has() {
        let mut tree = BinaryTree::new();
```

```
        fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {
            let got: Vec<bool> =
                (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
            assert_eq!(&got, exp);
        }

        check_has(&tree, &[false, false, false, false, false]);
        tree.insert(0);
        check_has(&tree, &[true, false, false, false, false]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(3);
        check_has(&tree, &[true, false, false, true, true]);
    }

    #[test]
    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
        assert!(tree.has(&50));
    }
}
```

### 21.4.1 Çözüm

```
use std::cmp::Ordering;

/// A node in the binary tree.
#[derive(Debug)]
struct Node<T: Ord> {
    value: T,
    left: Subtree<T>,
    right: Subtree<T>,
}

/// A possibly-empty subtree.
#[derive(Debug)]
struct Subtree<T: Ord>(Option<Box<Node<T>>>);

/// A container storing a set of values, using a binary tree.
///
/// If the same value is added multiple times, it is only stored once.
#[derive(Debug)]
pub struct BinaryTree<T: Ord> {
    root: Subtree<T>,
}
```

```rust
impl<T: Ord> BinaryTree<T> {
    fn new() -> Self {
        Self { root: Subtree::new() }
    }

    fn insert(&mut self, value: T) {
        self.root.insert(value);
    }

    fn has(&self, value: &T) -> bool {
        self.root.has(value)
    }

    fn len(&self) -> usize {
        self.root.len()
    }
}

impl<T: Ord> Subtree<T> {
    fn new() -> Self {
        Self(None)
    }

    fn insert(&mut self, value: T) {
        match &mut self.0 {
            None => self.0 = Some(Box::new(Node::new(value))),
            Some(n) => match value.cmp(&n.value) {
                Ordering::Less => n.left.insert(value),
                Ordering::Equal => {}
                Ordering::Greater => n.right.insert(value),
            },
        }
    }

    fn has(&self, value: &T) -> bool {
        match &self.0 {
            None => false,
            Some(n) => match value.cmp(&n.value) {
                Ordering::Less => n.left.has(value),
                Ordering::Equal => true,
                Ordering::Greater => n.right.has(value),
            },
        }
    }

    fn len(&self) -> usize {
        match &self.0 {
            None => 0,
            Some(n) => 1 + n.left.len() + n.right.len(),
        }
```

```
    }
}

impl<T: Ord> Node<T> {
    fn new(value: T) -> Self {
        Self { value, left: Subtree::new(), right: Subtree::new() }
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn len() {
        let mut tree = BinaryTree::new();
        assert_eq!(tree.len(), 0);
        tree.insert(2);
        assert_eq!(tree.len(), 1);
        tree.insert(1);
        assert_eq!(tree.len(), 2);
        tree.insert(2); // not a unique item
        assert_eq!(tree.len(), 2);
        tree.insert(3);
        assert_eq!(tree.len(), 3);
    }

    #[test]
    fn has() {
        let mut tree = BinaryTree::new();
        fn check_has(tree: &BinaryTree<i32>, exp: &[bool]) {
            let got: Vec<bool> =
                (0..exp.len()).map(|i| tree.has(&(i as i32))).collect();
            assert_eq!(&got, exp);
        }

        check_has(&tree, &[false, false, false, false, false]);
        tree.insert(0);
        check_has(&tree, &[true, false, false, false, false]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(4);
        check_has(&tree, &[true, false, false, false, true]);
        tree.insert(3);
        check_has(&tree, &[true, false, false, true, true]);
    }

    #[test]
    fn unbalanced() {
        let mut tree = BinaryTree::new();
        for i in 0..100 {
```

```
            tree.insert(i);
        }
        assert_eq!(tree.len(), 100);
        assert!(tree.has(&50));
    }
}
```

# Part VI

# Gün 3: Öğleden Sonra

# Chapter 22

# Tekrar Hoş Geldiniz

Bu oturum 10 dakikalık aralar dahil yaklaşık 1 saat 55 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Ödünç Alma | 55 dakika |
| Ömürler (Lifetimes) | 50 dakika |

# Chapter 23

# Ödünç Alma

Bu bölüm yaklaşık 55 dakika sürmelidir. İçerir:

| Slayt | Süre |
|---|---|
| Bir Değeri Ödünç Alma | 10 dakika |
| Ödünç Alma Kontrolü | 10 dakika |
| Ödünç Alma Hataları | 3 dakika |
| İç Değişebilirlik (Interior Mutability) | 10 dakika |
| Alıştırma: Sağlık İstatistikleri | 20 dakika |

## 23.1 Bir Değeri Ödünç Alma

As we saw before, instead of transferring ownership when calling a function, you can let a function *borrow* the value:

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    Point(p1.0 + p2.0, p1.1 + p2.1)
}

fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- The add function *borrows* two points and returns a new point.
- The caller retains ownership of the inputs.

This slide should take about 10 minutes.

This slide is a review of the material on references from day 1, expanding slightly to include function arguments and return values.

## Daha Fazlasını Keşfedin

Notes on stack returns and inlining:

- Demonstrate that the return from add is cheap because the compiler can eliminate the copy operation, by inlining the call to add into main. Change the above code to print stack addresses and run it on the Playground or look at the assembly in Godbolt. In the ”DEBUG” optimization level, the addresses should change, while they stay the same when changing to the ”RELEASE” setting:

```
#[derive(Debug)]
struct Point(i32, i32);

fn add(p1: &Point, p2: &Point) -> Point {
    let p = Point(p1.0 + p2.0, p1.1 + p2.1);
    println!("&p.0: {:p}", &p.0);
    p
}

pub fn main() {
    let p1 = Point(3, 4);
    let p2 = Point(10, 20);
    let p3 = add(&p1, &p2);
    println!("&p3.0: {:p}", &p3.0);
    println!("{p1:?} + {p2:?} = {p3:?}");
}
```

- The Rust compiler can do automatic inlining, that can be disabled on a function level with `#[inline(never)]`.
- Once disabled, the printed address will change on all optimization levels. Looking at Godbolt or Playground, one can see that in this case, the return of the value depends on the ABI, e.g. on amd64 the two i32 that is making up the point will be returned in 2 registers (eax and edx).

## 23.2 Ödünç Alma Kontrolü

Rust's *borrow checker* puts constraints on the ways you can borrow values. We've already seen that a reference cannot *outlive* the value it borrows:

```
fn main() {
    let x_ref = {
        let x = 10;
        &x
    };
    dbg!(x_ref);
}
```

There's also a second main rule that the borrow checker enforces: The *aliasing* rule. For a given value, at any time:

- You can have one or more shared references to the value, *or*
- You can have exactly one exclusive reference to the value.

```
fn main() {
    let mut a = 10;
    let b = &a;

    {
        let c = &mut a;
        *c = 20;
    }

    dbg!(a);
    dbg!(b);
}
```

This slide should take about 10 minutes.

- The ”outlives” rule was demonstrated previously when we first looked at references. We review it here to show students that the borrow checking is following a few different rules to validate borrowing.
- The above code does not compile because `a` is borrowed as mutable (through `c`) and as immutable (through `b`) at the same time.
  - Note that the requirement is that conflicting references not *exist* at the same point. It does not matter where the reference is dereferenced. Try commenting out `*c = 20` and show that the compiler error still occurs even if we never use `c`.
  - Note that the intermediate reference `c` isn't necessary to trigger a borrow conflict. Replace `c` with a direct mutation of `a` and demonstrate that this produces a similar error. This is because direct mutation of a value effectively creates a temporary mutable reference.
- Move the `dbg!` statement for `b` before the scope that introduces `c` to make the code compile.
  - After that change, the compiler realizes that `b` is only ever used before the new mutable borrow of `a` through `c`. This is a feature of the borrow checker called ”non-lexical lifetimes”.

## Daha Fazlasını Keşfedin

- Technically multiple mutable references to a piece of data can exist at the same time via re-borrowing. This is what allows you to pass a mutable reference into a function without invalidating the original reference. This playground example demonstrates that behavior.
- Rust uses the exclusive reference constraint to ensure that data races do not occur in multi-threaded code, since only one thread can have mutable access to a piece of data at a time.
- Rust also uses this constraint to optimize code. For example, a value behind a shared reference can be safely cached in a register for the lifetime of that reference.
- Fields of a struct can be borrowed independently of each other, but calling a method on a struct will borrow the whole struct, potentially invalidating references to individual fields. See this playground snippet for an example of this.

## 23.3 Ödünç Alma Hataları

As a concrete example of how these borrowing rules prevent memory errors, consider the case of modifying a collection while there are references to its elements:

```
fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];
    let elem = &vec[2];
    vec.push(6);
    dbg!(elem);
}
```

Similarly, consider the case of iterator invalidation:

```
fn main() {
    let mut vec = vec![1, 2, 3, 4, 5];
    for elem in &vec {
        vec.push(elem * 2);
    }
}
```

This slide should take about 3 minutes.

- In both of these cases, modifying the collection by pushing new elements into it can potentially invalidate existing references to the collection's elements if the collection has to reallocate.

## 23.4 İç Değişebilirlik (Interior Mutability)

In some situations, it's necessary to modify data behind a shared (read-only) reference. For example, a shared data structure might have an internal cache, and wish to update that cache from read-only methods.

The ”interior mutability” pattern allows exclusive (mutable) access behind a shared reference. The standard library provides several ways to do this, all while still ensuring safety, typically by performing a runtime check.

This slide and its sub-slides should take about 10 minutes.

The main thing to take away from this slide is that Rust provides *safe* ways to modify data behind a shared reference. There are a variety of ways to ensure that safety, and the next sub-slides present a few of them.

### 23.4.1 `Cell`

`Cell` wraps a value and allows getting or setting the value using only a shared reference to the `Cell`. However, it does not allow any references to the inner value. Since there are no references, borrowing rules cannot be broken.

```
use std::cell::Cell;

fn main() {
    // Note that `cell` is NOT declared as mutable.
    let cell = Cell::new(5);
```

```
    cell.set(123);
    dbg!(cell.get());
}
```

- `Cell` is a simple means to ensure safety: it has a `set` method that takes `&self`. This needs no runtime check, but requires moving values, which can have its own cost.

### 23.4.2 `RefCell`

`RefCell` allows accessing and mutating a wrapped value by providing alternative types `Ref` and `RefMut` that emulate `&T`/`&mut T` without actually being Rust references.

These types perform dynamic checks using a counter in the `RefCell` to prevent existence of a `RefMut` alongside another `Ref`/`RefMut`.

By implementing `Deref` (and `DerefMut` for `RefMut`), these types allow calling methods on the inner value without allowing references to escape.

```
use std::cell::RefCell;

fn main() {
    // Note that `cell` is NOT declared as mutable.
    let cell = RefCell::new(5);

    {
        let mut cell_ref = cell.borrow_mut();
        *cell_ref = 123;

        // This triggers an error at runtime.
        // let other = cell.borrow();
        // println!("{}", other);
    }

    println!("{cell:?}");
}
```

- `RefCell` enforces Rust's usual borrowing rules (either multiple shared references or a single exclusive reference) with a runtime check. In this case, all borrows are very short and never overlap, so the checks always succeed.
- The extra block in the example is to end the borrow created by the call to `borrow_mut` before we print the cell. Trying to print a borrowed `RefCell` just shows the message `"{borrowed}"`.

#### Daha Fazlasını Keşfedin

There are also `OnceCell` and `OnceLock`, which allow initialization on first use. Making these useful requires some more knowledge than students have at this time.

## 23.5 Alıştırma: Sağlık İstatistikleri

You're working on implementing a health-monitoring system. As part of that, you need to keep track of users' health statistics.

You'll start with a stubbed function in an `impl` block as well as a `User` struct definition. Your goal is to implement the stubbed out method on the `User` `struct` defined in the `impl` block.

Copy the code below to https://play.rust-lang.org/ and fill in the missing method:

```
#![allow(dead_code)]
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: u32,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        todo!("Update a user's statistics based on measurements from a visit to the doc
    }
}

#[test]
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);
    assert!((report.height_change - 0.9).abs() < 0.00001);
```

```
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
    assert_eq!(report.height_change, 0.0);
}
```

### 23.5.1 Çözüm

```
#![allow(dead_code)]
pub struct User {
    name: String,
    age: u32,
    height: f32,
    visit_count: u32,
    last_blood_pressure: Option<(u32, u32)>,
}

pub struct Measurements {
    height: f32,
    blood_pressure: (u32, u32),
}

pub struct HealthReport<'a> {
    patient_name: &'a str,
    visit_count: u32,
    height_change: f32,
    blood_pressure_change: Option<(i32, i32)>,
}

impl User {
    pub fn new(name: String, age: u32, height: f32) -> Self {
        Self { name, age, height, visit_count: 0, last_blood_pressure: None }
    }

    pub fn visit_doctor(&mut self, measurements: Measurements) -> HealthReport {
        self.visit_count += 1;
        let bp = measurements.blood_pressure;
        let report = HealthReport {
            patient_name: &self.name,
            visit_count: self.visit_count,
            height_change: measurements.height - self.height,
            blood_pressure_change: match self.last_blood_pressure {
                Some(lbp) => {
                    Some((bp.0 as i32 - lbp.0 as i32, bp.1 as i32 - lbp.1 as i32))
                }
                None => None,
            },
```

```
        };
        self.height = measurements.height;
        self.last_blood_pressure = Some(bp);
        report
    }
}

#[test]
fn test_visit() {
    let mut bob = User::new(String::from("Bob"), 32, 155.2);
    assert_eq!(bob.visit_count, 0);
    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (120, 80) });
    assert_eq!(report.patient_name, "Bob");
    assert_eq!(report.visit_count, 1);
    assert_eq!(report.blood_pressure_change, None);
    assert!((report.height_change - 0.9).abs() < 0.00001);

    let report =
        bob.visit_doctor(Measurements { height: 156.1, blood_pressure: (115, 76) });

    assert_eq!(report.visit_count, 2);
    assert_eq!(report.blood_pressure_change, Some((-5, -4)));
    assert_eq!(report.height_change, 0.0);
}
```

# Chapter 24

# Ömürler (Lifetimes)

Bu bölüm yaklaşık 50 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Ömür için Ek Açıklamalar (Annotations) | 10 dakika |
| Ömür Atlaması (Elision) | 5 dakika |
| Veri Yapılarında Ömürler | 5 dakika |
| Alıştırma: Protobuf Ayrıştırma | 30 dakika |

## 24.1 Ömür için Ek Açıklamalar (Annotations)

A reference has a *lifetime*, which must not "outlive" the value it refers to. This is verified by the borrow checker.

The lifetime can be implicit - this is what we have seen so far. Lifetimes can also be explicit: `&'a Point`, `&'document str`. Lifetimes start with `'` and `'a` is a typical default name. Read `&'a Point` as "a borrowed `Point` which is valid for at least the lifetime `a`".

Only ownership, not lifetime annotations, control when values are destroyed and determine the concrete lifetime of a given value. The borrow checker just validates that borrows never extend beyond the concrete lifetime of the value.

Explicit lifetime annotations, like types, are required on function signatures (but can be elided in common cases). These provide information for inference at callsites and within the function body, helping the borrow checker to do its job.

```
#[derive(Debug)]
struct Point(i32, i32);

fn left_most(p1: &Point, p2: &Point) -> &Point {
    if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
    let p1 = Point(10, 10);
```

```
    let p2 = Point(20, 20);
    let p3 = left_most(&p1, &p2); // What is the lifetime of p3?
    dbg!(p3);
}
```

This slide should take about 10 minutes.

In this example, the compiler does not know what lifetime to infer for p3. Looking inside the function body shows that it can only safely assume that p3's lifetime is the shorter of p1 and p2. But just like types, Rust requires explicit annotations of lifetimes on function arguments and return values.

Add `'a` appropriately to `left_most`:

```
fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
```

This says there is some lifetime `'a` which both p1 and p2 outlive, and which outlives the return value. The borrow checker verifies this within the function body, and uses this information in `main` to determine a lifetime for p3.

Try dropping p2 in `main` before printing p3.

## 24.2 Fonksiyon Çağrılarında Ömürler

Lifetimes for function arguments and return values must be fully specified, but Rust allows lifetimes to be elided in most cases with a few simple rules. This is not inference -- it is just a syntactic shorthand.

- Each argument which does not have a lifetime annotation is given one.
- If there is only one argument lifetime, it is given to all un-annotated return values.
- If there are multiple argument lifetimes, but the first one is for `self`, that lifetime is given to all un-annotated return values.

```
#[derive(Debug)]
struct Point(i32, i32);

fn cab_distance(p1: &Point, p2: &Point) -> i32 {
    (p1.0 - p2.0).abs() + (p1.1 - p2.1).abs()
}

fn find_nearest<'a>(points: &'a [Point], query: &Point) -> Option<&'a Point> {
    let mut nearest = None;
    for p in points {
        if let Some((_, nearest_dist)) = nearest {
            let dist = cab_distance(p, query);
            if dist < nearest_dist {
                nearest = Some((p, dist));
            }
        } else {
            nearest = Some((p, cab_distance(p, query)));
        };
    }
    nearest.map(|(p, _)| p)
}
```

```
fn main() {
    let points = &[Point(1, 0), Point(1, 0), Point(-1, 0), Point(0, -1)];
    let nearest = {
        let query = Point(0, 2);
        find_nearest(points, &query)
    };
    println!("{:?}", nearest);
}
```

This slide should take about 5 minutes.

In this example, `cab_distance` is trivially elided.

The `nearest` function provides another example of a function with multiple references in its arguments that requires explicit annotation. In `main`, the return value is allowed to outlive the query.

Try adjusting the signature to ”lie” about the lifetimes returned:

```
fn find_nearest<'a, 'q>(points: &'a [Point], query: &'q Point) -> Option<&'q Point> {
```

This won't compile, demonstrating that the annotations are checked for validity by the compiler. Note that this is not the case for raw pointers (unsafe), and this is a common source of errors with unsafe Rust.

Students may ask when to use lifetimes. Rust borrows *always* have lifetimes. Most of the time, elision and type inference mean these don't need to be written out. In more complicated cases, lifetime annotations can help resolve ambiguity. Often, especially when prototyping, it's easier to just work with owned data by cloning values where necessary.

## 24.3 Veri Yapılarında Ömürler

If a data type stores borrowed data, it must be annotated with a lifetime:

```
#[derive(Debug)]
enum HighlightColor {
    Pink,
    Yellow,
}

#[derive(Debug)]
struct Highlight<'document> {
    slice: &'document str,
    color: HighlightColor,
}

fn main() {
    let doc = String::from("The quick brown fox jumps over the lazy dog.");
    let noun = Highlight { slice: &doc[16..19], color: HighlightColor::Yellow };
    let verb = Highlight { slice: &doc[20..25], color: HighlightColor::Pink };
    // drop(doc);
    dbg!(noun);
```

```
    dbg!(verb);
}
```

This slide should take about 5 minutes.

- In the above example, the annotation on `Highlight` enforces that the data underlying the contained `&str` lives at least as long as any instance of `Highlight` that uses that data. A struct cannot live longer than the data it references.
- If `doc` is dropped before the end of the lifetime of `noun` or `verb`, the borrow checker throws an error.
- Types with borrowed data force users to hold on to the original data. This can be useful for creating lightweight views, but it generally makes them somewhat harder to use.
- When possible, make data structures own their data directly.
- Some structs with multiple references inside can have more than one lifetime annotation. This can be necessary if there is a need to describe lifetime relationships between the references themselves, in addition to the lifetime of the struct itself. Those are very advanced use cases.

## 24.4 Alıştırma: Protobuf Ayrıştırma

In this exercise, you will build a parser for the protobuf binary encoding. Don't worry, it's simpler than it seems! This illustrates a common parsing pattern, passing slices of data. The underlying data itself is never copied.

Fully parsing a protobuf message requires knowing the types of the fields, indexed by their field numbers. That is typically provided in a `proto` file. In this exercise, we'll encode that information into `match` statements in functions that get called for each field.

We'll use the following proto:

```
message PhoneNumber {
  optional string number = 1;
  optional string type = 2;
}

message Person {
  optional string name = 1;
  optional int32 id = 2;
  repeated PhoneNumber phones = 3;
}
```

### Messages

A proto message is encoded as a series of fields, one after the next. Each is implemented as a ”tag” followed by the value. The tag contains a field number (e.g., 2 for the `id` field of a `Person` message) and a wire type defining how the payload should be determined from the byte stream. These are combined into a single integer, as decoded in `unpack_tag` below.

### Varint

Integers, including the tag, are represented with a variable-length encoding called VARINT. Luckily, `parse_varint` is defined for you below.

## Wire Types

Proto defines several wire types, only two of which are used in this exercise.

The `Varint` wire type contains a single varint, and is used to encode proto values of type `int32` such as `Person.id`.

The `Len` wire type contains a length expressed as a varint, followed by a payload of that number of bytes. This is used to encode proto values of type `string` such as `Person.name`. It is also used to encode proto values containing sub-messages such as `Person.phones`, where the payload contains an encoding of the sub-message.

## Alıştırma

The given code also defines callbacks to handle `Person` and `PhoneNumber` fields, and to parse a message into a series of calls to those callbacks.

What remains for you is to implement the `parse_field` function and the `ProtoMessage` trait for `Person` and `PhoneNumber`.

```
/// A wire type as seen on the wire.
enum WireType {
    /// The Varint WireType indicates the value is a single VARINT.
    Varint,
    // The I64 WireType indicates that the value is precisely 8 bytes in
    // little-endian order containing a 64-bit signed integer or double type.
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    // The I32 WireType indicates that the value is precisely 4 bytes in
    // little-endian order containing a 32-bit signed integer or float type.
    //I32,  -- not needed for this exercise
}

#[derive(Debug)]
/// A field's value, typed based on the wire type.
enum FieldValue<'a> {
    Varint(u64),
    //I64(i64),  -- not needed for this exercise
    Len(&'a [u8]),
    //I32(i32),  -- not needed for this exercise
}

#[derive(Debug)]
/// A field, containing the field number and its value.
struct Field<'a> {
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default {
    fn add_field(&mut self, field: Field<'a>);
```

```
}

impl From<u64> for WireType {
    fn from(value: u64) -> Self {
        match value {
            0 => WireType::Varint,
            //1 => WireType::I64,  -- not needed for this exercise
            2 => WireType::Len,
            //5 => WireType::I32,  -- not needed for this exercise
            _ => panic!("Invalid wire type: {value}"),
        }
    }
}

impl<'a> FieldValue<'a> {
    fn as_str(&self) -> &'a str {
        let FieldValue::Len(data) = self else {
            panic!("Expected string to be a `Len` field");
        };
        std::str::from_utf8(data).expect("Invalid string")
    }

    fn as_bytes(&self) -> &'a [u8] {
        let FieldValue::Len(data) = self else {
            panic!("Expected bytes to be a `Len` field");
        };
        data
    }

    fn as_u64(&self) -> u64 {
        let FieldValue::Varint(value) = self else {
            panic!("Expected `u64` to be a `Varint` field");
        };
        *value
    }
}

/// Parse a VARINT, returning the parsed value and the remaining bytes.
fn parse_varint(data: &[u8]) -> (u64, &[u8]) {
    for i in 0..7 {
        let Some(b) = data.get(i) else {
            panic!("Not enough bytes for varint");
        };
        if b & 0x80 == 0 {
            // This is the last byte of the VARINT, so convert it to
            // a u64 and return it.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return (value, &data[i + 1..]);
```

```
        }
    }

    // More than 7 bytes is invalid.
    panic!("Too many bytes for varint");
}

/// Convert a tag into a field number and a WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
    let field_num = tag >> 3;
    let wire_type = WireType::from(tag & 0x7);
    (field_num, wire_type)
}


/// Parse a field, returning the remaining bytes
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
    let (tag, remainder) = parse_varint(data);
    let (field_num, wire_type) = unpack_tag(tag);
    let (fieldvalue, remainder) = match wire_type {
        _ => todo!("Based on the wire type, build a Field, consuming as many bytes as ne
    };
    todo!("Return the field, and any un-consumed bytes.")
}

/// Parse a message in the given data, calling `T::add_field` for each field in
/// the message.
///
/// The entire input is consumed.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> T {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data);
        result.add_field(parsed.0);
        data = parsed.1;
    }
    result
}

#[derive(Debug, Default)]
struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
}

#[derive(Debug, Default)]
struct Person<'a> {
    name: &'a str,
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}
```

```
// TODO: Implement ProtoMessage for Person and PhoneNumber.

#[test]
fn test_id() {
    let person_id: Person = parse_message(&[0x10, 0x2a]);
    assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });
}

#[test]
fn test_name() {
    let person_name: Person = parse_message(&[
        0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
        0x6e, 0x61, 0x6d, 0x65,
    ]);
    assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![] });
}

#[test]
fn test_just_person() {
    let person_name_id: Person =
        parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
    assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });
}

#[test]
fn test_phone() {
    let phone: Person = parse_message(&[
        0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" },],
        }
    );
}

// Put that all together into a single parse.
#[test]
fn test_full_person() {
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
```

```
        0x65,
    ]);
    assert_eq!(
        person,
        Person {
            name: "maxwell",
            id: 42,
            phone: vec![
                PhoneNumber { number: "+1202-555-1212", type_: "home" },
                PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
            ]
        }
    );
}
```

This slide and its sub-slides should take about 30 minutes.

- In this exercise there are various cases where protobuf parsing might fail, e.g. if you try to parse an `i32` when there are fewer than 4 bytes left in the data buffer. In normal Rust code we'd handle this with the `Result` enum, but for simplicity in this exercise we panic if any errors are encountered. On day 4 we'll cover error handling in Rust in more detail.

### 24.4.1 Çözüm

```
/// A wire type as seen on the wire.
enum WireType {
    /// The Varint WireType indicates the value is a single VARINT.
    Varint,
    // The I64 WireType indicates that the value is precisely 8 bytes in
    // little-endian order containing a 64-bit signed integer or double type.
    //I64,  -- not needed for this exercise
    /// The Len WireType indicates that the value is a length represented as a
    /// VARINT followed by exactly that number of bytes.
    Len,
    // The I32 WireType indicates that the value is precisely 4 bytes in
    // little-endian order containing a 32-bit signed integer or float type.
    //I32,  -- not needed for this exercise
}

#[derive(Debug)]
/// A field's value, typed based on the wire type.
enum FieldValue<'a> {
    Varint(u64),
    //I64(i64),  -- not needed for this exercise
    Len(&'a [u8]),
    //I32(i32),  -- not needed for this exercise
}

#[derive(Debug)]
/// A field, containing the field number and its value.
struct Field<'a> {
```

```
    field_num: u64,
    value: FieldValue<'a>,
}

trait ProtoMessage<'a>: Default {
    fn add_field(&mut self, field: Field<'a>);
}

impl From<u64> for WireType {
    fn from(value: u64) -> Self {
        match value {
            0 => WireType::Varint,
            //1 => WireType::I64,  -- not needed for this exercise
            2 => WireType::Len,
            //5 => WireType::I32,  -- not needed for this exercise
            _ => panic!("Invalid wire type: {value}"),
        }
    }
}

impl<'a> FieldValue<'a> {
    fn as_str(&self) -> &'a str {
        let FieldValue::Len(data) = self else {
            panic!("Expected string to be a `Len` field");
        };
        std::str::from_utf8(data).expect("Invalid string")
    }

    fn as_bytes(&self) -> &'a [u8] {
        let FieldValue::Len(data) = self else {
            panic!("Expected bytes to be a `Len` field");
        };
        data
    }

    fn as_u64(&self) -> u64 {
        let FieldValue::Varint(value) = self else {
            panic!("Expected `u64` to be a `Varint` field");
        };
        *value
    }
}

/// Parse a VARINT, returning the parsed value and the remaining bytes.
fn parse_varint(data: &[u8]) -> (u64, &[u8]) {
    for i in 0..7 {
        let Some(b) = data.get(i) else {
            panic!("Not enough bytes for varint");
        };
        if b & 0x80 == 0 {
            // This is the last byte of the VARINT, so convert it to
```

```
            // a u64 and return it.
            let mut value = 0u64;
            for b in data[..=i].iter().rev() {
                value = (value << 7) | (b & 0x7f) as u64;
            }
            return (value, &data[i + 1..]);
        }
    }

    // More than 7 bytes is invalid.
    panic!("Too many bytes for varint");
}

/// Convert a tag into a field number and a WireType.
fn unpack_tag(tag: u64) -> (u64, WireType) {
    let field_num = tag >> 3;
    let wire_type = WireType::from(tag & 0x7);
    (field_num, wire_type)
}

/// Parse a field, returning the remaining bytes
fn parse_field(data: &[u8]) -> (Field, &[u8]) {
    let (tag, remainder) = parse_varint(data);
    let (field_num, wire_type) = unpack_tag(tag);
    let (fieldvalue, remainder) = match wire_type {
        WireType::Varint => {
            let (value, remainder) = parse_varint(remainder);
            (FieldValue::Varint(value), remainder)
        }
        WireType::Len => {
            let (len, remainder) = parse_varint(remainder);
            let len: usize = len.try_into().expect("len not a valid `usize`");
            if remainder.len() < len {
                panic!("Unexpected EOF");
            }
            let (value, remainder) = remainder.split_at(len);
            (FieldValue::Len(value), remainder)
        }
    };
    (Field { field_num, value: fieldvalue }, remainder)
}

/// Parse a message in the given data, calling `T::add_field` for each field in
/// the message.
///
/// The entire input is consumed.
fn parse_message<'a, T: ProtoMessage<'a>>(mut data: &'a [u8]) -> T {
    let mut result = T::default();
    while !data.is_empty() {
        let parsed = parse_field(data);
        result.add_field(parsed.0);
```

```
        data = parsed.1;
    }
    result
}

#[derive(PartialEq)]
#[derive(Debug, Default)]
struct PhoneNumber<'a> {
    number: &'a str,
    type_: &'a str,
}

#[derive(PartialEq)]
#[derive(Debug, Default)]
struct Person<'a> {
    name: &'a str,
    id: u64,
    phone: Vec<PhoneNumber<'a>>,
}

impl<'a> ProtoMessage<'a> for Person<'a> {
    fn add_field(&mut self, field: Field<'a>) {
        match field.field_num {
            1 => self.name = field.value.as_str(),
            2 => self.id = field.value.as_u64(),
            3 => self.phone.push(parse_message(field.value.as_bytes())),
            _ => {} // skip everything else
        }
    }
}

impl<'a> ProtoMessage<'a> for PhoneNumber<'a> {
    fn add_field(&mut self, field: Field<'a>) {
        match field.field_num {
            1 => self.number = field.value.as_str(),
            2 => self.type_ = field.value.as_str(),
            _ => {} // skip everything else
        }
    }
}

#[test]
fn test_id() {
    let person_id: Person = parse_message(&[0x10, 0x2a]);
    assert_eq!(person_id, Person { name: "", id: 42, phone: vec![] });
}

#[test]
fn test_name() {
    let person_name: Person = parse_message(&[
        0x0a, 0x0e, 0x62, 0x65, 0x61, 0x75, 0x74, 0x69, 0x66, 0x75, 0x6c, 0x20,
```

```
        0x6e, 0x61, 0x6d, 0x65,
    ]);
    assert_eq!(person_name, Person { name: "beautiful name", id: 0, phone: vec![] });
}

#[test]
fn test_just_person() {
    let person_name_id: Person =
        parse_message(&[0x0a, 0x04, 0x45, 0x76, 0x61, 0x6e, 0x10, 0x16]);
    assert_eq!(person_name_id, Person { name: "Evan", id: 22, phone: vec![] });
}

#[test]
fn test_phone() {
    let phone: Person = parse_message(&[
        0x0a, 0x00, 0x10, 0x00, 0x1a, 0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x33,
        0x34, 0x2d, 0x37, 0x37, 0x37, 0x2d, 0x39, 0x30, 0x39, 0x30, 0x12, 0x04,
        0x68, 0x6f, 0x6d, 0x65,
    ]);
    assert_eq!(
        phone,
        Person {
            name: "",
            id: 0,
            phone: vec![PhoneNumber { number: "+1234-777-9090", type_: "home" },],
        }
    );
}

// Put that all together into a single parse.
#[test]
fn test_full_person() {
    let person: Person = parse_message(&[
        0x0a, 0x07, 0x6d, 0x61, 0x78, 0x77, 0x65, 0x6c, 0x6c, 0x10, 0x2a, 0x1a,
        0x16, 0x0a, 0x0e, 0x2b, 0x31, 0x32, 0x30, 0x32, 0x2d, 0x35, 0x35, 0x35,
        0x2d, 0x31, 0x32, 0x31, 0x32, 0x12, 0x04, 0x68, 0x6f, 0x6d, 0x65, 0x1a,
        0x18, 0x0a, 0x0e, 0x2b, 0x31, 0x38, 0x30, 0x30, 0x2d, 0x38, 0x36, 0x37,
        0x2d, 0x35, 0x33, 0x30, 0x38, 0x12, 0x06, 0x6d, 0x6f, 0x62, 0x69, 0x6c,
        0x65,
    ]);
    assert_eq!(
        person,
        Person {
            name: "maxwell",
            id: 42,
            phone: vec![
                PhoneNumber { number: "+1202-555-1212", type_: "home" },
                PhoneNumber { number: "+1800-867-5308", type_: "mobile" },
            ]
        }
    );
```

```
}
```

# Part VII

# Gün 4: Sabah

# Chapter 25

# 4. Gün'e Hoş Geldiniz

Today we will cover topics relating to building large-scale software in Rust:

- Iterators: a deep dive on the `Iterator` trait.
- Modules and visibility.
- Testing.
- Error handling: panics, `Result`, and the try operator `?`.
- Unsafe Rust: the escape hatch when you can't express yourself in safe Rust.

## Zamanlama (Schedule)

Bu oturum 10 dakikalık aralar dahil yaklaşık 2 saat 50 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Hoş Geldiniz | 3 dakika |
| Adımlayıcılar (Iterators) | 55 dakika |
| Modüller | 45 dakika |
| Test Etme | 45 dakika |

# Chapter 26

# Adımlayıcılar (Iterators)

Bu bölüm yaklaşık 55 dakika sürmelidir. İçerir:

| Slayt | Süre |
|---|---|
| Motivasyon | 3 dakika |
| Iterator Özelliği (Trait) | 5 dakika |
| Iterator Yardımcı Metotları | 5 dakika |
| collect | 5 dakika |
| IntoIterator | 5 dakika |
| Alıştırma: Adımlayıcı Metot Zincirleme | 30 dakika |

## 26.1 Adımlayıcılara (Iterators) Motive Edici Şeyler

If you want to iterate over the contents of an array, you'll need to define:

- Some state to keep track of where you are in the iteration process, e.g. an index.
- A condition to determine when iteration is done.
- Logic for updating the state of iteration each loop.
- Logic for fetching each element using that iteration state.

In a C-style for loop you declare these things directly:

```
for (int i = 0; i < array_len; i += 1) {
    int elem = array[i];
}
```

In Rust we bundle this state and logic together into an object known as an ”iterator”.

This slide should take about 3 minutes.

- This slide provides context for what Rust iterators do under the hood. We use the (hopefully) familiar construct of a C-style `for` loop to show how iteration requires some state and some logic, that way on the next slide we can show how an iterator bundles these together.
- Rust doesn't have a C-style `for` loop, but we can express the same thing with `while`:

```
    let array = [2, 4, 6, 8];
    let mut i = 0;
    while i < array.len() {
        let elem = array[i];
        i += 1;
    }
```

### Daha Fazlasını Keşfedin

There's another way to express array iteration using `for` in C and C++: You can use a pointer to the front and a pointer to the end of the array and then compare those pointers to determine when the loop should end.

```
for (int *ptr = array; ptr < array + len; ptr += 1) {
    int elem = *ptr;
}
```

If students ask, you can point out that this is how Rust's slice and array iterators work under the hood (though implemented as a Rust iterator).

## 26.2 `Iterator` Özelliği (Trait)

The `Iterator` trait defines how an object can be used to produce a sequence of values. For example, if we wanted to create an iterator that can produce the elements of a slice it might look something like this:

```
struct SliceIter<'s> {
    slice: &'s [i32],
    i: usize,
}

impl<'s> Iterator for SliceIter<'s> {
    type Item = &'s i32;

    fn next(&mut self) -> Option<Self::Item> {
        if self.i == self.slice.len() {
            None
        } else {
            let next = &self.slice[self.i];
            self.i += 1;
            Some(next)
        }
    }
}

fn main() {
    let slice = &[2, 4, 6, 8];
    let iter = SliceIter { slice, i: 0 };
    for elem in iter {
        dbg!(elem);
```

```
    }
}
```

This slide should take about 5 minutes.

- The `SliceIter` example implements the same logic as the C-style `for` loop demonstrated on the last slide.
- Point out to the students that iterators are lazy: Creating the iterator just initializes the struct but does not otherwise do any work. No work happens until the `next` method is called.
- Iterators don't need to be finite! It's entirely valid to have an iterator that will produce values forever. For example, a half open range like `0..` will keep going until integer overflow occurs.

### Daha Fazlasını Keşfedin

- The "real" version of `SliceIter` is the `slice::Iter` type in the standard library, however the real version uses pointers under the hood instead of an index in order to eliminate bounds checks.
- The `SliceIter` example is a good example of a struct that contains a reference and therefore uses lifetime annotations.
- You can also demonstrate adding a generic parameter to `SliceIter` to allow it to work with any kind of slice (not just `&[i32]`).

## 26.3 `Iterator` Yardımcı Metotları

In addition to the `next` method that defines how an iterator behaves, the `Iterator` trait provides 70+ helper methods that can be used to build customized iterators.

```
fn main() {
    let result: i32 = (1..=10) // Create a range from 1 to 10
        .filter(|x| x % 2 == 0) // Keep only even numbers
        .map(|x| x * x) // Square each number
        .sum(); // Sum up all the squared numbers

    println!("The sum of squares of even numbers from 1 to 10 is: {}", result);
}
```

This slide should take about 5 minutes.

- The `Iterator` trait implements many common functional programming operations over collections (e.g. `map`, `filter`, `reduce`, etc). This is the trait where you can find all the documentation about them.
- Many of these helper methods take the original iterator and produce a new iterator with different behavior. These are know as "iterator adapter methods".
- Some methods, like `sum` and `count`, consume the iterator and pull all of the elements out of it.
- These methods are designed to be chained together so that it's easy to build a custom iterator that does exactly what you need.

### Daha Fazlasını Keşfedin

- Rust's iterators are extremely efficient and highly optimizable. Even complex iterators made by combining many adapter methods will still result in code as efficient as equivalent imperative implementations.

## 26.4 `collect`

The `collect` method lets you build a collection from an `Iterator`.

```
fn main() {
    let primes = vec![2, 3, 5, 7];
    let prime_squares = primes.into_iter().map(|p| p * p).collect::<Vec<_>>();
    println!("prime_squares: {prime_squares:?}");
}
```

This slide should take about 5 minutes.

- Any iterator can be collected in to a `Vec`, `VecDeque`, or `HashSet`. Iterators that produce key-value pairs (i.e. a two-element tuple) can also be collected into `HashMap` and `BTreeMap`.

Show the students the definition for `collect` in the standard library docs. There are two ways to specify the generic type `B` for this method:

- With the "turbofish": `some_iterator.collect::<COLLECTION_TYPE>()`, as shown. The `_` shorthand used here lets Rust infer the type of the `Vec` elements.
- With type inference: `let prime_squares: Vec<_> = some_iterator.collect()`. Rewrite the example to use this form.

### Daha Fazlasını Keşfedin

- If students are curious about how this works, you can bring up the `FromIterator` trait, which defines how each type of collection gets built from an iterator.
- In addition to the basic implementations of `FromIterator` for `Vec`, `HashMap`, etc., there are also more specialized implementations which let you do cool things like convert an `Iterator<Item = Result<V, E>>` into a `Result<Vec<V>, E>`.
- The reason type annotations are often needed with `collect` is because it's generic over its return type. This makes it harder for the compiler to infer the correct type in a lot of cases.

## 26.5 `IntoIterator`

The `Iterator` trait tells you how to *iterate* once you have created an iterator. The related trait `IntoIterator` defines how to create an iterator for a type. It is used automatically by the `for` loop.

```
struct Grid {
    x_coords: Vec<u32>,
    y_coords: Vec<u32>,
}
```

```
impl IntoIterator for Grid {
    type Item = (u32, u32);
    type IntoIter = GridIter;
    fn into_iter(self) -> GridIter {
        GridIter { grid: self, i: 0, j: 0 }
    }
}

struct GridIter {
    grid: Grid,
    i: usize,
    j: usize,
}

impl Iterator for GridIter {
    type Item = (u32, u32);

    fn next(&mut self) -> Option<(u32, u32)> {
        if self.i >= self.grid.x_coords.len() {
            self.i = 0;
            self.j += 1;
            if self.j >= self.grid.y_coords.len() {
                return None;
            }
        }
        let res = Some((self.grid.x_coords[self.i], self.grid.y_coords[self.j]));
        self.i += 1;
        res
    }
}

fn main() {
    let grid = Grid { x_coords: vec![3, 5, 7, 9], y_coords: vec![10, 20, 30, 40] };
    for (x, y) in grid {
        println!("point = {x}, {y}");
    }
}
```

This slide should take about 5 minutes.

- `IntoIterator` is the trait that makes for loops work. It is implemented by collection types such as `Vec<T>` and references to them such as `&Vec<T>` and `&[T]`. Ranges also implement it. This is why you can iterate over a vector with `for i in some_vec { .. }` but `some_vec.next()` doesn't exist.

Click through to the docs for `IntoIterator`. Every implementation of `IntoIterator` must declare two types:

- `Item`: the type to iterate over, such as `i8`,
- `IntoIter`: the `Iterator` type returned by the `into_iter` method.

Note that `IntoIter` and `Item` are linked: the iterator must have the same `Item` type, which means that it returns `Option<Item>`

The example iterates over all combinations of x and y coordinates.

Try iterating over the grid twice in `main`. Why does this fail? Note that `IntoIterator::into_iter` takes ownership of `self`.

Fix this issue by implementing `IntoIterator` for `&Grid` and creating a `GridRefIter` that iterates by reference. A version with both `GridIter` and `GridRefIter` is available in this playground.

The same problem can occur for standard library types: `for e in some_vector` will take ownership of `some_vector` and iterate over owned elements from that vector. Use `for e in &some_vector` instead, to iterate over references to elements of `some_vector`.

## 26.6 Alıştırma: Adımlayıcı Metot Zincirleme

In this exercise, you will need to find and use some of the provided methods in the `Iterator` trait to implement a complex calculation.

Copy the following code to https://play.rust-lang.org/ and make the tests pass. Use an iterator expression and `collect` the result to construct the return value.

```
/// Calculate the differences between elements of `values` offset by `offset`,
/// wrapping around from the end of `values` to the beginning.
///
/// Element `n` of the result is `values[(n+offset)%len] - values[n]`.
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
    todo!()
}

#[test]
fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
    assert_eq!(offset_differences(1, empty), vec![]);
}
```

### 26.6.1 Çözüm

```
/// Calculate the differences between elements of `values` offset by `offset`,
/// wrapping around from the end of `values` to the beginning.
///
/// Element `n` of the result is `values[(n+offset)%len] - values[n]`.
fn offset_differences(offset: usize, values: Vec<i32>) -> Vec<i32> {
    let a = values.iter();
    let b = values.iter().cycle().skip(offset);
    a.zip(b).map(|(a, b)| *b - *a).collect()
}

#[test]
fn test_offset_one() {
    assert_eq!(offset_differences(1, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
    assert_eq!(offset_differences(1, vec![1, 3, 5]), vec![2, 2, -4]);
    assert_eq!(offset_differences(1, vec![1, 3]), vec![2, -2]);
}

#[test]
fn test_larger_offsets() {
    assert_eq!(offset_differences(2, vec![1, 3, 5, 7]), vec![4, 4, -4, -4]);
    assert_eq!(offset_differences(3, vec![1, 3, 5, 7]), vec![6, -2, -2, -2]);
    assert_eq!(offset_differences(4, vec![1, 3, 5, 7]), vec![0, 0, 0, 0]);
    assert_eq!(offset_differences(5, vec![1, 3, 5, 7]), vec![2, 2, 2, -6]);
}

#[test]
fn test_degenerate_cases() {
    assert_eq!(offset_differences(1, vec![0]), vec![0]);
    assert_eq!(offset_differences(1, vec![1]), vec![0]);
    let empty: Vec<i32> = vec![];
    assert_eq!(offset_differences(1, empty), vec![]);
}
```

# Chapter 27

# Modüller

Bu bölüm yaklaşık 45 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Modüller | 3 dakika |
| Dosya Sistemi Hiyerarşisi | 5 dakika |
| Görünürlük | 5 dakika |
| Kapsülleme (Encapsulation) | 5 dakika |
| use, super, self | 10 dakika |
| Alıştırma: Bir GUI Kütüphanesi için Modüller | 15 dakika |

## 27.1 Modüller

We have seen how `impl` blocks let us namespace functions to a type.

Similarly, `mod` lets us namespace types and functions:

```
mod foo {
    pub fn do_something() {
        println!("In the foo module");
    }
}

mod bar {
    pub fn do_something() {
        println!("In the bar module");
    }
}

fn main() {
    foo::do_something();
    bar::do_something();
}
```

This slide should take about 3 minutes.

- Packages provide functionality and include a `Cargo.toml` file that describes how to build a bundle of 1+ crates.
- Crates are a tree of modules, where a binary crate creates an executable and a library crate compiles to a library.
- Modules define organization, scope, and are the focus of this section.

## 27.2 Dosya Sistemi Hiyerarşisi

Omitting the module content will tell Rust to look for it in another file:

```
mod garden;
```

This tells Rust that the `garden` module content is found at `src/garden.rs`. Similarly, a `garden::vegetables` module can be found at `src/garden/vegetables.rs`.

The `crate` root is in:

- `src/lib.rs` (for a library crate)
- `src/main.rs` (for a binary crate)

Modules defined in files can be documented, too, using "inner doc comments". These document the item that contains them -- in this case, a module.

```
//! This module implements the garden, including a highly performant germination
//! implementation.

// Re-export types from this module.
pub use garden::Garden;
pub use seeds::SeedPacket;

/// Sow the given seed packets.
pub fn sow(seeds: Vec<SeedPacket>) {
    todo!()
}

/// Harvest the produce in the garden that is ready.
pub fn harvest(garden: &mut Garden) {
    todo!()
}
```

This slide should take about 5 minutes.

- Before Rust 2018, modules needed to be located at `module/mod.rs` instead of `module.rs`, and this is still a working alternative for editions after 2018.
- The main reason to introduce `filename.rs` as alternative to `filename/mod.rs` was because many files named `mod.rs` can be hard to distinguish in IDEs.
- Deeper nesting can use folders, even if the main module is a file:

```
src/
├── main.rs
├── top_module.rs
└── top_module/
    └── sub_module.rs
```

- The place rust will look for modules can be changed with a compiler directive:

  ```
  #[path = "some/path.rs"]
  mod some_module;
  ```

  This is useful, for example, if you would like to place tests for a module in a file named `some_module_test.rs`, similar to the convention in Go.

## 27.3 Görünürlük

Modules are a privacy boundary:

- Module items are private by default (hides implementation details).
- Parent and sibling items are always visible.
- In other words, if an item is visible in module `foo`, it's visible in all the descendants of `foo`.

```
mod outer {
    fn private() {
        println!("outer::private");
    }

    pub fn public() {
        println!("outer::public");
    }

    mod inner {
        fn private() {
            println!("outer::inner::private");
        }

        pub fn public() {
            println!("outer::inner::public");
            super::private();
        }
    }
}

fn main() {
    outer::public();
}
```

This slide should take about 5 minutes.

- Use the pub keyword to make modules public.

Additionally, there are advanced `pub(...)` specifiers to restrict the scope of public visibility.

- See the Rust Reference.
- Configuring `pub(crate)` visibility is a common pattern.
- Less commonly, you can give visibility to a specific path.
- In any case, visibility must be granted to an ancestor module (and all of its descendants).

## 27.4 Visibility and Encapsulation

Like with items in a module, struct fields are also private by default. Private fields are likewise visible within the rest of the module (including child modules). This allows us to encapsulate implementation details of struct, controlling what data and functionality is visible externally.

```
use outer::Foo;

mod outer {
    pub struct Foo {
        pub val: i32,
        is_big: bool,
    }

    impl Foo {
        pub fn new(val: i32) -> Self {
            Self { val, is_big: val > 100 }
        }
    }

    pub mod inner {
        use super::Foo;

        pub fn print_foo(foo: &Foo) {
            println!("{} büyük mü? {}", foo.val, foo.is_big);
        }
    }
}

fn main() {
    let foo = Foo::new(42);
    println!("foo.val = {}", foo.val);
    // let foo = Foo { val: 42, is_big: true };

    outer::inner::print_foo(&foo);
    // println!("Is {} big? {}", foo.val, foo.is_big);
}
```

This slide should take about 5 minutes.

- This slide demonstrates how privacy in structs is module-based. Students coming from object oriented languages may be used to types being the encapsulation boundary, so this demonstrates how Rust behaves differently while showing how we can still achieve encapsulation.
- Note how the `is_big` field is fully controlled by `Foo`, allowing `Foo` to control how it's initialized and enforce any invariants it needs to (e.g. that `is_big` is only `true` if `val > 100`).
- Point out how helper functions can be defined in the same module (including child modules) in order to get access to the type's private fields/methods.
- The first commented out line demonstrates that you cannot initialize a struct with private fields. The second one demonstrates that you also can't directly access private

fields.

- Enums do not support privacy: Variants and data within those variants is always public.

### Daha Fazlasını Keşfedin

- If students want more information about privacy (or lack thereof) in enums, you can bring up `#[doc_hidden]` and `#[non_exhaustive]` and show how they're used to limit what can be done with an enum.
- Module privacy still applies when there are `impl` blocks in other modules (example in the playground).

## 27.5 use, super, self

A module can bring symbols from another module into scope with `use`. You will typically see something like this at the top of each module:

```
use std::collections::HashSet;
use std::process::abort;
```

### Yollar (Paths)

Paths are resolved as follows:

1. As a relative path:
   - `foo` or `self::foo` refers to `foo` in the current module,
   - `super::foo` refers to `foo` in the parent module.
2. As an absolute path:
   - `crate::foo` refers to `foo` in the root of the current crate,
   - `bar::foo` refers to `foo` in the `bar` crate.

This slide should take about 8 minutes.

- It is common to ”re-export” symbols at a shorter path. For example, the top-level `lib.rs` in a crate might have

  ```
  mod storage;

  pub use storage::disk::DiskStorage;
  pub use storage::network::NetworkStorage;
  ```

  making `DiskStorage` and `NetworkStorage` available to other crates with a convenient, short path.
- For the most part, only items that appear in a module need to be `use`'d. However, a trait must be in scope to call any methods on that trait, even if a type implementing that trait is already in scope. For example, to use the `read_to_string` method on a type implementing the `Read` trait, you need to `use std::io::Read`.
- The `use` statement can have a wildcard: `use std::io::*`. This is discouraged because it is not clear which items are imported, and those might change over time.

## 27.6 Alıştırma: Bir GUI Kütüphanesi için Modüller

In this exercise, you will reorganize a small GUI Library implementation. This library defines a `Widget` trait and a few implementations of that trait, as well as a `main` function.

It is typical to put each type or set of closely-related types into its own module, so each widget type should get its own module.

### Cargo Kurulumu (Setup)

The Rust playground only supports one file, so you will need to make a Cargo project on your local filesystem:

```
cargo init gui-modules
cd gui-modules
cargo run
```

Edit the resulting `src/main.rs` to add `mod` statements, and add additional files in the `src` directory.

### Source

Here's the single-module implementation of the GUI library:

```rust
pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}

pub struct Label {
    label: String,
}

impl Label {
    fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

pub struct Button {
    label: Label,
}
```

```
impl Button {
    fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }

    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        // Add 4 paddings for borders
        self.inner_width() + 4
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: Change draw_into to return Result<(), std::fmt::Error>. Then use the
        // ?-operator here instead of .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
```

```
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        self.label.width() + 8 // add a bit of padding
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
    }

    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        writeln!(buffer, "{}", &self.label).unwrap();
    }
}

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
    window.add_widget(Box::new(Button::new("Click me!")));
    window.draw();
}
```

This slide and its sub-slides should take about 15 minutes.

Encourage students to divide the code in a way that feels natural for them, and get accustomed to the required `mod`, `use`, and `pub` declarations. Afterward, discuss what organizations are most idiomatic.

### 27.6.1 Çözüm

```
src
├── main.rs
├── widgets
│   ├── button.rs
```

```
│   ├── label.rs
│   └── window.rs
└── widgets.rs
```

```rust
// ---- src/widgets.rs ----
pub use button::Button;
pub use label::Label;
pub use window::Window;

mod button;
mod label;
mod window;

pub trait Widget {
    /// Natural width of `self`.
    fn width(&self) -> usize;

    /// Draw the widget into a buffer.
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write);

    /// Draw the widget on standard output.
    fn draw(&self) {
        let mut buffer = String::new();
        self.draw_into(&mut buffer);
        println!("{buffer}");
    }
}
```

```rust
// ---- src/widgets/label.rs ----
use super::Widget;

pub struct Label {
    label: String,
}

impl Label {
    pub fn new(label: &str) -> Label {
        Label { label: label.to_owned() }
    }
}

impl Widget for Label {
    fn width(&self) -> usize {
        // ANCHOR_END: Label-width
        self.label.lines().map(|line| line.chars().count()).max().unwrap_or(0)
    }

    // ANCHOR: Label-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Label-draw_into
        writeln!(buffer, "{}", &self.label).unwrap();
    }
```

```
}
// ---- src/widgets/button.rs ----
use super::{Label, Widget};

pub struct Button {
    label: Label,
}

impl Button {
    pub fn new(label: &str) -> Button {
        Button { label: Label::new(label) }
    }
}

impl Widget for Button {
    fn width(&self) -> usize {
        // ANCHOR_END: Button-width
        self.label.width() + 8 // add a bit of padding
    }

    // ANCHOR: Button-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Button-draw_into
        let width = self.width();
        let mut label = String::new();
        self.label.draw_into(&mut label);

        writeln!(buffer, "+{:-<width$}+", "").unwrap();
        for line in label.lines() {
            writeln!(buffer, "|{:^width$}|", &line).unwrap();
        }
        writeln!(buffer, "+{:-<width$}+", "").unwrap();
    }
}
// ---- src/widgets/window.rs ----
use super::Widget;

pub struct Window {
    title: String,
    widgets: Vec<Box<dyn Widget>>,
}

impl Window {
    pub fn new(title: &str) -> Window {
        Window { title: title.to_owned(), widgets: Vec::new() }
    }

    pub fn add_widget(&mut self, widget: Box<dyn Widget>) {
        self.widgets.push(widget);
    }
```

```
    fn inner_width(&self) -> usize {
        std::cmp::max(
            self.title.chars().count(),
            self.widgets.iter().map(|w| w.width()).max().unwrap_or(0),
        )
    }
}

impl Widget for Window {
    fn width(&self) -> usize {
        // ANCHOR_END: Window-width
        // Add 4 paddings for borders
        self.inner_width() + 4
    }

    // ANCHOR: Window-draw_into
    fn draw_into(&self, buffer: &mut dyn std::fmt::Write) {
        // ANCHOR_END: Window-draw_into
        let mut inner = String::new();
        for widget in &self.widgets {
            widget.draw_into(&mut inner);
        }

        let inner_width = self.inner_width();

        // TODO: after learning about error handling, you can change
        // draw_into to return Result<(), std::fmt::Error>. Then use
        // the ?-operator here instead of .unwrap().
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
        writeln!(buffer, "| {:^inner_width$} |", &self.title).unwrap();
        writeln!(buffer, "+={:=<inner_width$}=+", "").unwrap();
        for line in inner.lines() {
            writeln!(buffer, "| {:inner_width$} |", line).unwrap();
        }
        writeln!(buffer, "+-{:-<inner_width$}-+", "").unwrap();
    }
}

// ---- src/main.rs ----
mod widgets;

use widgets::{Button, Label, Widget, Window};

fn main() {
    let mut window = Window::new("Rust GUI Demo 1.23");
    window.add_widget(Box::new(Label::new("This is a small text GUI demo.")));
    window.add_widget(Box::new(Button::new("Click me!")));
    window.draw();
}
```

# Chapter 28

# Test Etme

Bu bölüm yaklaşık 45 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Birim Testleri | 5 dakika |
| Diğer Test Türleri | 5 dakika |
| Derleyici Tüyleri (Lint) ve Clippy Aracı | 3 dakika |
| Alıştırma: Luhn Algrotiması | 30 dakika |

## 28.1 Birim Testleri

Rust and Cargo come with a simple unit test framework. Tests are marked with `#[test]`. Unit tests are often put in a nested `tests` module, using `#[cfg(test)]` to conditionally compile them only when building tests.

```
fn first_word(text: &str) -> &str {
    match text.find(' ') {
        Some(idx) => &text[..idx],
        None => &text,
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_empty() {
        assert_eq!(first_word(""), "");
    }

    #[test]
    fn test_single_word() {
        assert_eq!(first_word("Merhaba"), "Merhaba");
```

```
    }

    #[test]
    fn test_multiple_words() {
        assert_eq!(first_word("Merhaba Dünya"), "Merhaba");
    }
}
```

- This lets you unit test private helpers.
- The `#[cfg(test)]` attribute is only active when you run `cargo test`.

## 28.2 Diğer Test Türleri

### Entegrasyon Testleri

If you want to test your library as a client, use an integration test.

Create a `.rs` file under `tests/`:

```
// tests/my_library.rs
use my_library::init;

#[test]
fn test_init() {
    assert!(init().is_ok());
}
```

These tests only have access to the public API of your crate.

### Dokümantasyon Testleri

Rust has built-in support for documentation tests:

```
/// Shortens a string to the given length.
///
/// ```
/// # use playground::shorten_string;
/// assert_eq!(shorten_string("Hello World", 5), "Hello");
/// assert_eq!(shorten_string("Hello World", 20), "Hello World");
/// ```
pub fn shorten_string(s: &str, length: usize) -> &str {
    &s[..std::cmp::min(length, s.len())]
}
```

- Code blocks in `///` comments are automatically seen as Rust code.
- The code will be compiled and executed as part of `cargo test`.
- Adding # in the code will hide it from the docs, but will still compile/run it.
- Test the above code on the Rust Playground.

## 28.3 Derleyici Tüyleri (Lint) ve Clippy Aracı

The Rust compiler produces fantastic error messages, as well as helpful built-in lints. Clippy provides even more lints, organized into groups that can be enabled per-project.

```
#[deny(clippy::cast_possible_truncation)]
fn main() {
    let mut x = 3;
    while (x < 70000) {
        x *= 2;
    }
    println!("X probably fits in a u16, right? {}", x as u16);
}
```

This slide should take about 3 minutes.

There are compiler lints visible here, but not clippy lints. Run `clippy` on the playground site to show clippy warnings. Clippy has extensive documentation of its lints, and adds new lints (including default-deny lints) all the time.

Note that errors or warnings with `help: ...` can be fixed with `cargo fix` or via your editor.

## 28.4 Alıştırma: Luhn Algrotiması

The Luhn algorithm is used to validate credit card numbers. The algorithm takes a string as input and does the following to validate the credit card number:

- Ignore all spaces. Reject numbers with fewer than two digits. Reject letters and other non-digit characters.
- Moving from **right to left**, double every second digit: for the number 1234, we double 3 and 1. For the number 98765, we double 6 and 8.
- After doubling a digit, sum the digits if the result is greater than 9. So doubling 7 becomes 14 which becomes `1 + 4 = 5`.
- Sum all the undoubled and doubled digits.
- The credit card number is valid if the sum ends with `0`.

The provided code provides a buggy implementation of the luhn algorithm, along with two basic unit tests that confirm that most of the algorithm is implemented correctly.

Copy the code below to https://play.rust-lang.org/ and write additional tests to uncover bugs in the provided implementation, fixing any bugs you find.

```
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
```

```
            } else {
                sum += digit;
            }
            double = !double;
        } else {
            continue;
        }
    }

    sum % 10 == 0
}

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
        assert!(luhn("7992 7398 713"));
    }

    #[test]
    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }
}
```

### 28.4.1 Çözüm

```
pub fn luhn(cc_number: &str) -> bool {
    let mut sum = 0;
    let mut double = false;
    let mut digits = 0;

    for c in cc_number.chars().rev() {
        if let Some(digit) = c.to_digit(10) {
            digits += 1;
            if double {
                let double_digit = digit * 2;
                sum +=
                    if double_digit > 9 { double_digit - 9 } else { double_digit };
            } else {
                sum += digit;
            }
            double = !double;
        } else if c.is_whitespace() {
            // New: accept whitespace.
```

```
            continue;
        } else {
            // New: reject all other characters.
            return false;
        }
    }

    // New: check that we have at least two digits
    digits >= 2 && sum % 10 == 0
}

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn test_valid_cc_number() {
        assert!(luhn("4263 9826 4026 9299"));
        assert!(luhn("4539 3195 0343 6467"));
        assert!(luhn("7992 7398 713"));
    }

    #[test]
    fn test_invalid_cc_number() {
        assert!(!luhn("4223 9826 4026 9299"));
        assert!(!luhn("4539 3195 0343 6476"));
        assert!(!luhn("8273 1232 7352 0569"));
    }

    #[test]
    fn test_non_digit_cc_number() {
        assert!(!luhn("foo"));
        assert!(!luhn("foo 0 0"));
    }

    #[test]
    fn test_empty_cc_number() {
        assert!(!luhn(""));
        assert!(!luhn(" "));
        assert!(!luhn("  "));
        assert!(!luhn("    "));
    }

    #[test]
    fn test_single_digit_cc_number() {
        assert!(!luhn("0"));
    }

    #[test]
    fn test_two_digit_cc_number() {
        assert!(luhn(" 0 0 "));
```

```
        }
    }
```

# Part VIII

# Gün 4: Öğleden Sonra

# Chapter 29

# Tekrar Hoş Geldiniz

Bu oturum 10 dakikalık aralar dahil yaklaşık 2 saat 20 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Hata İşleme | 55 dakika |
| Emniyetsiz (Unsafe) Rust | 1 saat 15 dakika |

# Chapter 30

# Hata İşleme

Bu bölüm yaklaşık 55 dakika sürmelidir. İçerir:

| Slayt | Süre |
|---|---|
| Panikler (Panics) | 3 dakika |
| Result | 5 dakika |
| Try Operatörü | 5 dakika |
| Try Dönüşümleri | 5 dakika |
| Error Özelliği (Trait) | 5 dakika |
| thiserror | 5 dakika |
| anyhow | 5 dakika |
| Alıştırma: Result ile Yeniden Yazma | 20 dakika |

## 30.1 Panikler (Panics)

Rust handles fatal errors with a ”panic”.

Rust will trigger a panic if a fatal error happens at runtime:

```
fn main() {
    let v = vec![10, 20, 30];
    dbg!(v[100]);
}
```

- Panics are for unrecoverable and unexpected errors.
  - Panics are symptoms of bugs in the program.
  - Runtime failures like failed bounds checks can panic
  - Assertions (such as `assert!`) panic on failure
  - Purpose-specific panics can use the `panic!` macro.
- A panic will ”unwind” the stack, dropping values just as if the functions had returned.
- Use non-panicking APIs (such as `Vec::get`) if crashing is not acceptable.

This slide should take about 3 minutes.

By default, a panic will cause the stack to unwind. The unwinding can be caught:

```
use std::panic;

fn main() {
    let result = panic::catch_unwind(|| "No problem here!");
    dbg!(result);

    let result = panic::catch_unwind(|| {
        panic!("oh no!");
    });
    dbg!(result);
}
```

- Catching is unusual; do not attempt to implement exceptions with `catch_unwind`!
- This can be useful in servers which should keep running even if a single request crashes.
- This does not work if `panic = 'abort'` is set in your `Cargo.toml`.

## 30.2 `Result`

Our primary mechanism for error handling in Rust is the `Result` enum, which we briefly saw when discussing standard library types.

```
use std::fs::File;
use std::io::Read;

fn main() {
    let file: Result<File, std::io::Error> = File::open("günlük.txt");
    match file {
        Ok(mut file) => {
            let mut contents = String::new();
            if let Ok(bytes) = file.read_to_string(&mut contents) {
                println!("Sevgili günlük: {contents} ({bytes} bayt)");
            } else {
                println!("Dosya içeriği okunamadı");
            }
        }
        Err(err) => {
            println!("Günlük açılamadı: {err}");
        }
    }
}
```

This slide should take about 5 minutes.

- `Result` has two variants: `Ok` which contains the success value, and `Err` which contains an error value of some kind.
- Whether or not a function can produce an error is encoded in the function's type signature by having the function return a `Result` value.
- Like with `Option`, there is no way to forget to handle an error: You cannot access either the success value or the error value without first pattern matching on the `Result` to check which variant you have. Methods like `unwrap` make it easier to write quick-and-dirty code that doesn't do robust error handling, but means that you can always see in

your source code where proper error handling is being skipped.

## Daha Fazlasını Keşfedin

It may be helpful to compare error handling in Rust to error handling conventions that students may be familiar with from other programming languages.

### İstisnalar

- Many languages use exceptions, e.g. C++, Java, Python.
- In most languages with exceptions, whether or not a function can throw an exception is not visible as part of its type signature. This generally means that you can't tell when calling a function if it may throw an exception or not.
- Exceptions generally unwind the call stack, propagating upward until a `try` block is reached. An error originating deep in the call stack may impact an unrelated function further up.

### Error Numbers

- Some languages have functions return an error number (or some other error value) separately from the successful return value of the function. Examples include C and Go.
- Depending on the language it may be possible to forget to check the error value, in which case you may be accessing an uninitialized or otherwise invalid success value.

## 30.3 Try Operatörü

Runtime errors like connection-refused or file-not-found are handled with the `Result` type, but matching this type on every call can be cumbersome. The try-operator `?` is used to return errors to the caller. It lets you turn the common

```
match some_expression {
    Ok(value) => value,
    Err(err) => return Err(err),
}
```

into the much simpler

```
some_expression?
```

We can use this to simplify our error handling code:

```
use std::io::Read;
use std::{fs, io};

fn read_username(path: &str) -> Result<String, io::Error> {
    let username_file_result = fs::File::open(path);
    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(err) => return Err(err),
```

```
    };

    let mut username = String::new();
    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(err) => Err(err),
    }
}

fn main() {
    //fs::write("config.dat", "alice").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

This slide should take about 5 minutes.

Simplify the `read_username` function to use `?`.

Anahtar noktalar:

- The `username` variable can be either `Ok(string)` or `Err(error)`.
- Use the `fs::write` call to test out the different scenarios: no file, empty file, file with username.
- Note that `main` can return a `Result<(), E>` as long as it implements `std::process::Termination`. In practice, this means that `E` implements `Debug`. The executable will print the `Err` variant and return a nonzero exit status on error.

## 30.4 Try Dönüşümleri

The effective expansion of `?` is a little more complicated than previously indicated:

```
expression?
```

works the same as

```
match expression {
    Ok(value) => value,
    Err(err)  => return Err(From::from(err)),
}
```

The `From::from` call here means we attempt to convert the error type to the type returned by the function. This makes it easy to encapsulate errors into higher-level errors.

### Örnek

```
use std::error::Error;
use std::io::Read;
use std::{fmt, fs, io};

#[derive(Debug)]
enum ReadUsernameError {
    IoError(io::Error),
    EmptyUsername(String),
```

```
}

impl Error for ReadUsernameError {}

impl fmt::Display for ReadUsernameError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            Self::IoError(e) => write!(f, "I/O error: {e}"),
            Self::EmptyUsername(path) => write!(f, "Found no username in {path}"),
        }
    }
}

impl From<io::Error> for ReadUsernameError {
    fn from(err: io::Error) -> Self {
        Self::IoError(err)
    }
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //std::fs::write("config.dat", "").unwrap();
    let username = read_username("config.dat");
    println!("username or error: {username:?}");
}
```

This slide should take about 5 minutes.

The `?` operator must return a value compatible with the return type of the function. For `Result`, it means that the error types have to be compatible. A function that returns `Result<T, ErrorOuter>` can only use `?` on a value of type `Result<U, ErrorInner>` if `ErrorOuter` and `ErrorInner` are the same type or if `ErrorOuter` implements `From<ErrorInner>`.

A common alternative to a `From` implementation is `Result::map_err`, especially when the conversion only happens in one place.

There is no compatibility requirement for `Option`. A function returning `Option<T>` can use the `?` operator on `Option<U>` for arbitrary `T` and `U` types.

A function that returns `Result` cannot use `?` on `Option` and vice versa. However, `Option::ok_or` converts `Option` to `Result` whereas `Result::ok` turns `Result` into `Option`.

## 30.5 Dinamik Hata Türleri

Sometimes we want to allow any type of error to be returned without writing our own enum covering all the different possibilities. The `std::error::Error` trait makes it easy to create a trait object that can contain any error.

```
use std::error::Error;
use std::fs;
use std::io::Read;

fn read_count(path: &str) -> Result<i32, Box<dyn Error>> {
    let mut count_str = String::new();
    fs::File::open(path)?.read_to_string(&mut count_str)?;
    let count: i32 = count_str.parse()?;
    Ok(count)
}

fn main() {
    fs::write("count.dat", "1i3").unwrap();
    match read_count("count.dat") {
        Ok(count) => println!("Count: {count}"),
        Err(err) => println!("Error: {err}"),
    }
}
```

This slide should take about 5 minutes.

The `read_count` function can return `std::io::Error` (from file operations) or `std::num::ParseIntError` (from `String::parse`).

Boxing errors saves on code, but gives up the ability to cleanly handle different error cases differently in the program. As such it's generally not a good idea to use `Box<dyn Error>` in the public API of a library, but it can be a good option in a program where you just want to display the error message somewhere.

Make sure to implement the `std::error::Error` trait when defining a custom error type so it can be boxed.

## 30.6 `thiserror`

The `thiserror` crate provides macros to help avoid boilerplate when defining error types. It provides derive macros that assist in implementing `From<T>`, `Display`, and the `Error` trait.

```
use std::io::Read;
use std::{fs, io};
use thiserror::Error;

#[derive(Debug, Error)]
enum ReadUsernameError {
    #[error("I/O error: {0}")]
    IoError(#[from] io::Error),
    #[error("Found no username in {0}")]
    EmptyUsername(String),
```

```
}

fn read_username(path: &str) -> Result<String, ReadUsernameError> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)?.read_to_string(&mut username)?;
    if username.is_empty() {
        return Err(ReadUsernameError::EmptyUsername(String::from(path)));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err) => println!("Error: {err:?}"),
    }
}
```

This slide should take about 5 minutes.

- The `Error` derive macro is provided by `thiserror`, and has lots of useful attributes to help define error types in a compact way.
- The message from `#[error]` is used to derive the `Display` trait.
- Note that the `(thiserror::)Error` derive macro, while it has the effect of implementing the `(std::error::)Error` trait, is not the same this; traits and macros do not share a namespace.

## 30.7 anyhow

The anyhow crate provides a rich error type with support for carrying additional contextual information, which can be used to provide a semantic trace of what the program was doing leading up to the error.

This can be combined with the convenience macros from thiserror to avoid writing out trait impls explicitly for custom error types.

```
use anyhow::{Context, Result, bail};
use std::fs;
use std::io::Read;
use thiserror::Error;

#[derive(Clone, Debug, Eq, Error, PartialEq)]
#[error("Found no username in {0}")]
struct EmptyUsernameError(String);

fn read_username(path: &str) -> Result<String> {
    let mut username = String::with_capacity(100);
    fs::File::open(path)
        .with_context(|| format!("Failed to open {path}"))?
        .read_to_string(&mut username)
        .context("Failed to read")?;
```

```
    if username.is_empty() {
        bail!(EmptyUsernameError(path.to_string()));
    }
    Ok(username)
}

fn main() {
    //fs::write("config.dat", "").unwrap();
    match read_username("config.dat") {
        Ok(username) => println!("Username: {username}"),
        Err(err) => println!("Error: {err:?}"),
    }
}
```

This slide should take about 5 minutes.

- `anyhow::Error` is essentially a wrapper around `Box<dyn Error>`. As such it's again generally not a good choice for the public API of a library, but is widely used in applications.
- `anyhow::Result<V>` is a type alias for `Result<V, anyhow::Error>`.
- Functionality provided by `anyhow::Error` may be familiar to Go developers, as it provides similar behavior to the Go `error` type and `Result<T, anyhow::Error>` is much like a Go `(T, error)` (with the convention that only one element of the pair is meaningful).
- `anyhow::Context` is a trait implemented for the standard `Result` and `Option` types. `use anyhow::Context` is necessary to enable `.context()` and `.with_context()` on those types.

## Daha Fazlasını Keşfedin

- `anyhow::Error` has support for downcasting, much like `std::any::Any`; the specific error type stored inside can be extracted for examination if desired with `Error::downcast`.

## 30.8 Alıştırma: Result ile Yeniden Yazma

In this exercise we're revisiting the expression evaluator exercise that we did in day 2. Our initial solution ignores a possible error case: Dividing by zero! Rewrite `eval` to instead use idiomatic error handling to handle this error case and return an error when it occurs. We provide a simple `DivideByZeroError` type to use as the error type for `eval`.

```
/// İki alt ifade (subexpression) üzerinde gerçekleştirilecek bir işlem.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}
```

```rust
/// Ağaç şeklinde bir ifade (expression).
#[derive(Debug)]
enum Expression {
    /// İki alt ifade üzerinde bir işlem.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Bir değişmez (literal) değer
    Value(i64),
}

#[derive(PartialEq, Eq, Debug)]
struct DivideByZeroError;

// The original implementation of the expression evaluator. Update this to
// return a `Result` and produce an error when dividing by 0.
fn eval(e: Expression) -> i64 {
    match e {
        Expression::Op { op, left, right } => {
            let left = eval(*left);
            let right = eval(*right);
            match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => if right != 0 {
                    left / right
                } else {
                    panic!("Cannot divide by zero!");
                },
            }
        }
        Expression::Value(v) => v,
    }
}

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn test_error() {
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Div,
                left: Box::new(Expression::Value(99)),
                right: Box::new(Expression::Value(0)),
            }),
            Err(DivideByZeroError)
        );
    }
```

```
    #[test]
    fn test_ok() {
        let expr = Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(20)),
            right: Box::new(Expression::Value(10)),
        };
        assert_eq!(eval(expr), Ok(10));
    }
}
```

This slide and its sub-slides should take about 20 minutes.

- The starting code here isn't exactly the same as the previous exercise's solution: We've added in an explicit panic to show students where the error case is. Point this out if students get confused.

### 30.8.1 Çözüm

```
/// İki alt ifade (subexpression) üzerinde gerçekleştirilecek bir işlem.
#[derive(Debug)]
enum Operation {
    Add,
    Sub,
    Mul,
    Div,
}

/// Ağaç şeklinde bir ifade (expression).
#[derive(Debug)]
enum Expression {
    /// İki alt ifade üzerinde bir işlem.
    Op { op: Operation, left: Box<Expression>, right: Box<Expression> },

    /// Bir değişmez (literal) değer
    Value(i64),
}

#[derive(PartialEq, Eq, Debug)]
struct DivideByZeroError;

fn eval(e: Expression) -> Result<i64, DivideByZeroError> {
    match e {
        Expression::Op { op, left, right } => {
            let left = eval(*left)?;
            let right = eval(*right)?;
            Ok(match op {
                Operation::Add => left + right,
                Operation::Sub => left - right,
                Operation::Mul => left * right,
                Operation::Div => {
                    if right == 0 {
```

```
                        return Err(DivideByZeroError);
                    } else {
                        left / right
                    }
                }
            })
        }
        Expression::Value(v) => Ok(v),
    }
}

#[cfg(test)]
mod test {
    use super::*;

    #[test]
    fn test_error() {
        assert_eq!(
            eval(Expression::Op {
                op: Operation::Div,
                left: Box::new(Expression::Value(99)),
                right: Box::new(Expression::Value(0)),
            }),
            Err(DivideByZeroError)
        );
    }

    #[test]
    fn test_ok() {
        let expr = Expression::Op {
            op: Operation::Sub,
            left: Box::new(Expression::Value(20)),
            right: Box::new(Expression::Value(10)),
        };
        assert_eq!(eval(expr), Ok(10));
    }
}
```

# Chapter 31

# Emniyetsiz (Unsafe) Rust

Bu bölüm yaklaşık 1 saat 15 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Emniyetsiz | 5 dakika |
| Ham Göstericilerinin İçeriği (Dereferencing) | 10 dakika |
| Değişebilir Statik Değişkenler | 5 dakika |
| Birlikler (Unions) | 5 dakika |
| Emniyetsiz (Unsafe) Fonksiyonlar | 15 dakika |
| Emniyetsiz Özellikler (Unsafe Traits) | 5 dakika |
| Alıştırma: FFI Sarmalayıcı | 30 dakika |

## 31.1 Emniyetsiz (Unsafe) Rust

The Rust language has two parts:

- **Safe Rust:** memory safe, no undefined behavior possible.
- **Unsafe Rust:** can trigger undefined behavior if preconditions are violated.

We saw mostly safe Rust in this course, but it's important to know what Unsafe Rust is.

Unsafe code is usually small and isolated, and its correctness should be carefully documented. It is usually wrapped in a safe abstraction layer.

Unsafe Rust gives you access to five new capabilities:

- Dereference raw pointers.
- Access or modify mutable static variables.
- Access `union` fields.
- Call `unsafe` functions, including `extern` functions.
- Implement `unsafe` traits.

We will briefly cover unsafe capabilities next. For full details, please see Chapter 19.1 in the Rust Book and the Rustonomicon.

This slide should take about 5 minutes.

Unsafe Rust does not mean the code is incorrect. It means that developers have turned off some compiler safety features and have to write correct code by themselves. It means the compiler no longer enforces Rust's memory-safety rules.

## 31.2 Ham Göstericilerinin İçeriği (Dereferencing)

Creating pointers is safe, but dereferencing them requires `unsafe`:

```
fn main() {
    let mut x = 10;

    let p1: *mut i32 = &raw mut x;
    let p2 = p1 as *const i32;

    // SAFETY: p1 and p2 were created by taking raw pointers to a local, so they
    // are guaranteed to be non-null, aligned, and point into a single (stack-)
    // allocated object.
    //
    // The object underlying the raw pointers lives for the entire function, so
    // it is not deallocated while the raw pointers still exist. It is not
    // accessed through references while the raw pointers exist, nor is it
    // accessed from other threads concurrently.
    unsafe {
        dbg!(*p1);
        *p1 = 6;
        // Mutation may soundly be observed through a raw pointer, like in C.
        dbg!(*p2);
    }

    // UNSOUND. DO NOT DO THIS.
    /*
    let r: &i32 = unsafe { &*p1 };
    dbg!(r);
    x = 50;
    dbg!(r); // Object underlying the reference has been mutated. This is UB.
    */
}
```

This slide should take about 10 minutes.

It is good practice (and required by the Android Rust style guide) to write a comment for each `unsafe` block explaining how the code inside it satisfies the safety requirements of the unsafe operations it is doing.

In the case of pointer dereferences, this means that the pointers must be *valid*, i.e.:

- The pointer must be non-null.
- The pointer must be *dereferenceable* (within the bounds of a single allocated object).
- The object must not have been deallocated.
- There must not be concurrent accesses to the same location.
- If the pointer was obtained by casting a reference, the underlying object must be live and no reference may be used to access the memory.

In most cases the pointer must also be properly aligned.

The ”UNSOUND” section gives an example of a common kind of UB bug: naïvely taking a reference to the dereference of a raw pointer sidesteps the compiler's knowledge of what object the reference is actually pointing to. As such, the borrow checker does not freeze x and so we are able to modify it despite the existence of a reference to it. Creating a reference from a pointer requires *great care*.

## 31.3 Değişebilir Statik Değişkenler

It is safe to read an immutable static variable:

```
static HELLO_WORLD: &str = "Merhaba, dünya!";

fn main() {
    println!("HELLO_WORLD: {HELLO_WORLD}");
}
```

However, mutable static variables are unsafe to read and write because multiple threads could do so concurrently without synchronization, constituting a data race.

Using mutable statics soundly requires reasoning about concurrency without the compiler's help:

```
static mut COUNTER: u32 = 0;

fn add_to_counter(inc: u32) {
    // SAFETY: There are no other threads which could be accessing `COUNTER`.
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_counter(42);

    // SAFETY: There are no other threads which could be accessing `COUNTER`.
    unsafe {
        dbg!(COUNTER);
    }
}
```

This slide should take about 5 minutes.

- The program here is sound because it is single-threaded. However, the Rust compiler reasons about functions individually so can't assume that. Try removing the `unsafe` and see how the compiler explains that it is undefined behavior to access a mutable static from multiple threads.
- The 2024 Rust edition goes further and makes accessing a mutable static by reference an error by default.
- Using a mutable static is almost always a bad idea, you should use interior mutability instead.

- There are some cases where it might be necessary in low-level `no_std` code, such as implementing a heap allocator or working with some C APIs. In this case you should use pointers rather than references.

## 31.4 Birlikler (Unions)

Unions are like enums, but you need to track the active field yourself:

```
#[repr(C)]
union MyUnion {
    i: u8,
    b: bool,
}

fn main() {
    let u = MyUnion { i: 42 };
    println!("int: {}", unsafe { u.i });
    println!("bool: {}", unsafe { u.b }); // Undefined behavior!
}
```

This slide should take about 5 minutes.

Unions are very rarely needed in Rust as you can usually use an enum. They are occasionally needed for interacting with C library APIs.

If you just want to reinterpret bytes as a different type, you probably want `std::mem::transmute` or a safe wrapper such as the `zerocopy` crate.

## 31.5 Emniyetsiz (Unsafe) Fonksiyonlar

A function or method can be marked `unsafe` if it has extra preconditions you must uphold to avoid undefined behaviour.

Unsafe functions may come from two places:

- Rust functions declared unsafe.
- Unsafe foreign functions in `extern "C"` blocks.

This slide and its sub-slides should take about 15 minutes.

We will look at the two kinds of unsafe functions next.

### 31.5.1 Emniyetsiz (Unsafe) Rust Fonksiyonlar

You can mark your own functions as `unsafe` if they require particular preconditions to avoid undefined behaviour.

```
/// Swaps the values pointed to by the given pointers.
///
/// # Safety
///
/// The pointers must be valid, properly aligned, and not otherwise accessed for
/// the duration of the function call.
```

```
unsafe fn swap(a: *mut u8, b: *mut u8) {
    // SAFETY: Our caller promised that the pointers are valid, properly aligned
    // and have no other access.
    unsafe {
        let temp = *a;
        *a = *b;
        *b = temp;
    }
}

fn main() {
    let mut a = 42;
    let mut b = 66;

    // SAFETY: The pointers must be valid, aligned and unique because they came
    // from references.
    unsafe {
        swap(&mut a, &mut b);
    }

    println!("a = {}, b = {}", a, b);
}
```

We wouldn't actually use pointers for a `swap` function --- it can be done safely with references.

Note that Rust 2021 and earlier allow unsafe code within an unsafe function without an `unsafe` block. This changed in the 2024 edition. We can prohibit it in older editions with `#[deny(unsafe_op_in_unsafe_fn)]`. Try adding it and see what happens.

### 31.5.2 Emniyetsiz (Unsafe) Harici Fonksiyonlar

You can declare foreign functions for access from Rust with `unsafe extern`. This is unsafe because the compiler has to way to reason about their behavior. Functions declared in an `extern` block must be marked as `safe` or `unsafe`, depending on whether they have preconditions for safe use:

```
use std::ffi::c_char;

unsafe extern "C" {
    // `abs` doesn't deal with pointers and doesn't have any safety requirements.
    safe fn abs(input: i32) -> i32;

    /// # Safety
    ///
    /// `s` must be a pointer to a NUL-terminated C string which is valid and
    /// not modified for the duration of this function call.
    unsafe fn strlen(s: *const c_char) -> usize;
}

fn main() {
    println!("Absolute value of -3 according to C: {}", abs(-3));
```

```
    unsafe {
        // SAFETY: We pass a pointer to a C string literal which is valid for
        // the duration of the program.
        println!("Dize uzunluğu: {}", strlen(c"String".as_ptr()));
    }
}
```

- Rust used to consider all extern functions unsafe, but this changed in Rust 1.82 with `unsafe extern` blocks.
- `abs` must be explicitly marked as `safe` because it is an external function (FFI). Calling external functions is usually only a problem when those functions do things with pointers which might violate Rust's memory model, but in general any C function might have undefined behaviour under any arbitrary circumstances.
- The `"C"` in this example is the ABI; other ABIs are available too.
- Note that there is no verification that the Rust function signature matches that of the function definition -- that's up to you!

### 31.5.3 Emniyetsiz (Unsafe) Fonksiyonları Çağırma

Failing to uphold the safety requirements breaks memory safety!

```
#[derive(Debug)]
#[repr(C)]
struct KeyPair {
    pk: [u16; 4], // 8 bytes
    sk: [u16; 4], // 8 bytes
}

const PK_BYTE_LEN: usize = 8;

fn log_public_key(pk_ptr: *const u16) {
    let pk: &[u16] = unsafe { std::slice::from_raw_parts(pk_ptr, PK_BYTE_LEN) };
    println!("{pk:?}");
}

fn main() {
    let key_pair = KeyPair { pk: [1, 2, 3, 4], sk: [0, 0, 42, 0] };
    log_public_key(key_pair.pk.as_ptr());
}
```

Always include a safety comment for each `unsafe` block. It must explain why the code is actually safe. This example is missing a safety comment and is unsound.

Anahtar noktalar:

- The second argument to `slice::from_raw_parts` is the number of *elements*, not bytes! This example demonstrates unexpected behavior by reading past the end of one array and into another.
- This is undefined behavior because we're reading past the end of the array that the pointer was derived from.
- `log_public_key` should be unsafe, because `pk_ptr` must meet certain prerequisites to avoid undefined behaviour. A safe function which can cause undefined behaviour is said to be unsound. What should its safety documentation say?

- The standard library contains many low-level unsafe functions. Prefer the safe alternatives when possible!
- If you use an unsafe function as an optimization, make sure to add a benchmark to demonstrate the gain.

## 31.6 Emniyetsiz Özelliklerin (Unsafe Traits) Gerçekleştirilmesi

Like with functions, you can mark a trait as `unsafe` if the implementation must guarantee particular conditions to avoid undefined behaviour.

For example, the `zerocopy` crate has an unsafe trait that looks something like this:

```
use std::{mem, slice};

/// ...
/// # Safety
/// The type must have a defined representation and no padding.
pub unsafe trait IntoBytes {
    fn as_bytes(&self) -> &[u8] {
        let len = mem::size_of_val(self);
        let slf: *const Self = self;
        unsafe { slice::from_raw_parts(slf.cast::<u8>(), len) }
    }
}

// SAFETY: `u32` has a defined representation and no padding.
unsafe impl IntoBytes for u32 {}
```

This slide should take about 5 minutes.

There should be a `# Safety` section on the Rustdoc for the trait explaining the requirements for the trait to be safely implemented.

The actual safety section for `IntoBytes` is rather longer and more complicated.

The built-in `Send` and `Sync` traits are unsafe.

## 31.7 Emniyetli FFI Sarıcı (Wrapper)

Rust has great support for calling functions through a *foreign function interface* (FFI). We will use this to build a safe wrapper for the `libc` functions you would use from C to read the names of files in a directory.

You will want to consult the manual pages:

- `opendir(3)`
- `readdir(3)`
- `closedir(3)`

You will also want to browse the `std::ffi` module. There you find a number of string types which you need for the exercise:

| Türler | Encoding | Use |
|---|---|---|
| `str` and `String` | UTF-8 | Text processing in Rust |
| `CStr` and `CString` | NUL-terminated | Communicating with C functions |
| `OsStr` and `OsString` | OS-specific | Communicating with the OS |

You will convert between all these types:

- `&str` to `CString`: you need to allocate space for a trailing `\0` character,
- `CString` to `*const i8`: you need a pointer to call C functions,
- `*const i8` to `&CStr`: you need something which can find the trailing `\0` character,
- `&CStr` to `&[u8]`: a slice of bytes is the universal interface for "some unknown data",
- `&[u8]` to `&OsStr`: `&OsStr` is a step towards `OsString`, use `OsStrExt` to create it,
- `&OsStr` to `OsString`: you need to clone the data in `&OsStr` to be able to return it and call `readdir` again.

The Nomicon also has a very useful chapter about FFI.

Copy the code below to https://play.rust-lang.org/ and fill in the missing functions and methods:

```
// TODO: uyarlamanız (implementation) bittiğinde bu yorum satırını silin.
#![allow(unused_imports, unused_variables, dead_code)]

mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "macos"))]
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // Layout according to the Linux man page for readdir(3), where ino_t and
    // off_t are resolved according to the definitions in
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    #[cfg(not(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }
```

```
    // Layout according to the macOS man page for dir(5).
    #[cfg(all(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    unsafe extern "C" {
        pub unsafe fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        // See https://github.com/rust-lang/libc/issues/414 and the section on
        // _DARWIN_FEATURE_64_BIT_INODE in the macOS man page for stat(2).
        //
        // "Platforms that existed before these updates were available" refers
        // to macOS (as opposed to iOS / wearOS / etc.) on Intel and PowerPC.
        #[cfg(all(target_os = "macos", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        pub unsafe fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        todo!()
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
```

```
        // Keep calling readdir until we get a NULL pointer back.
        todo!()
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        todo!()
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}
```

This slide and its sub-slides should take about 30 minutes.

FFI binding code is typically generated by tools like bindgen, rather than being written manually as we are doing here. However, bindgen can't run in an online playground.

### 31.7.1 Çözüm

```
mod ffi {
    use std::os::raw::{c_char, c_int};
    #[cfg(not(target_os = "macos"))]
    use std::os::raw::{c_long, c_uchar, c_ulong, c_ushort};

    // Opaque type. See https://doc.rust-lang.org/nomicon/ffi.html.
    #[repr(C)]
    pub struct DIR {
        _data: [u8; 0],
        _marker: core::marker::PhantomData<(*mut u8, core::marker::PhantomPinned)>,
    }

    // Layout according to the Linux man page for readdir(3), where ino_t and
    // off_t are resolved according to the definitions in
    // /usr/include/x86_64-linux-gnu/{sys/types.h, bits/typesizes.h}.
    #[cfg(not(target_os = "macos"))]
    #[repr(C)]
    pub struct dirent {
        pub d_ino: c_ulong,
        pub d_off: c_long,
        pub d_reclen: c_ushort,
        pub d_type: c_uchar,
        pub d_name: [c_char; 256],
    }

    // Layout according to the macOS man page for dir(5).
    #[cfg(all(target_os = "macos"))]
```

```
    #[repr(C)]
    pub struct dirent {
        pub d_fileno: u64,
        pub d_seekoff: u64,
        pub d_reclen: u16,
        pub d_namlen: u16,
        pub d_type: u8,
        pub d_name: [c_char; 1024],
    }

    unsafe extern "C" {
        pub unsafe fn opendir(s: *const c_char) -> *mut DIR;

        #[cfg(not(all(target_os = "macos", target_arch = "x86_64")))]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        // See https://github.com/rust-lang/libc/issues/414 and the section on
        // _DARWIN_FEATURE_64_BIT_INODE in the macOS man page for stat(2).
        //
        // "Platforms that existed before these updates were available" refers
        // to macOS (as opposed to iOS / wearOS / etc.) on Intel and PowerPC.
        #[cfg(all(target_os = "macos", target_arch = "x86_64"))]
        #[link_name = "readdir$INODE64"]
        pub unsafe fn readdir(s: *mut DIR) -> *const dirent;

        pub unsafe fn closedir(s: *mut DIR) -> c_int;
    }
}

use std::ffi::{CStr, CString, OsStr, OsString};
use std::os::unix::ffi::OsStrExt;

#[derive(Debug)]
struct DirectoryIterator {
    path: CString,
    dir: *mut ffi::DIR,
}

impl DirectoryIterator {
    fn new(path: &str) -> Result<DirectoryIterator, String> {
        // Call opendir and return a Ok value if that worked,
        // otherwise return Err with a message.
        let path =
            CString::new(path).map_err(|err| format!("Invalid path: {err}"))?;
        // SAFETY: path.as_ptr() cannot be NULL.
        let dir = unsafe { ffi::opendir(path.as_ptr()) };
        if dir.is_null() {
            Err(format!("Could not open {path:?}"))
        } else {
            Ok(DirectoryIterator { path, dir })
        }
```

```
    }
}

impl Iterator for DirectoryIterator {
    type Item = OsString;
    fn next(&mut self) -> Option<OsString> {
        // Keep calling readdir until we get a NULL pointer back.
        // SAFETY: self.dir is never NULL.
        let dirent = unsafe { ffi::readdir(self.dir) };
        if dirent.is_null() {
            // We have reached the end of the directory.
            return None;
        }
        // SAFETY: dirent is not NULL and dirent.d_name is NUL
        // terminated.
        let d_name = unsafe { CStr::from_ptr((*dirent).d_name.as_ptr()) };
        let os_str = OsStr::from_bytes(d_name.to_bytes());
        Some(os_str.to_owned())
    }
}

impl Drop for DirectoryIterator {
    fn drop(&mut self) {
        // Call closedir as needed.
        // SAFETY: self.dir is never NULL.
        if unsafe { ffi::closedir(self.dir) } != 0 {
            panic!("Could not close {:?}", self.path);
        }
    }
}

fn main() -> Result<(), String> {
    let iter = DirectoryIterator::new(".")?;
    println!("files: {:#?}", iter.collect::<Vec<_>>());
    Ok(())
}

#[cfg(test)]
mod tests {
    use super::*;
    use std::error::Error;

    #[test]
    fn test_nonexisting_directory() {
        let iter = DirectoryIterator::new("no-such-directory");
        assert!(iter.is_err());
    }

    #[test]
    fn test_empty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
```

```
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Non UTF-8 character in path")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", ".."]);
        Ok(())
    }

    #[test]
    fn test_nonempty_directory() -> Result<(), Box<dyn Error>> {
        let tmp = tempfile::TempDir::new()?;
        std::fs::write(tmp.path().join("foo.txt"), "The Foo Diaries\n")?;
        std::fs::write(tmp.path().join("bar.png"), "<PNG>\n")?;
        std::fs::write(tmp.path().join("crab.rs"), "//! Crab\n")?;
        let iter = DirectoryIterator::new(
            tmp.path().to_str().ok_or("Non UTF-8 character in path")?,
        )?;
        let mut entries = iter.collect::<Vec<_>>();
        entries.sort();
        assert_eq!(entries, &[".", "..", "bar.png", "crab.rs", "foo.txt"]);
        Ok(())
    }
}
```

# Part IX

# Android

# Chapter 32

# Welcome to Rust in Android

Rust is supported for system software on Android. This means that you can write new services, libraries, drivers or even firmware in Rust (or improve existing code as needed).

The speaker may mention any of the following given the increased use of Rust in Android:

- Service example: DNS over HTTP.
- Libraries: Rutabaga Virtual Graphics Interface.
- Kernel Drivers: Binder.
- Firmware: pKVM firmware.

# Chapter 33

# Kurulum (Setup)

We will be using a Cuttlefish Android Virtual Device to test our code. Make sure you have access to one or create a new one with:

```
source build/envsetup.sh
lunch aosp_cf_x86_64_phone-trunk_staging-userdebug
acloud create
```

Please see the Android Developer Codelab for details.

The code on the following pages can be found in the `src/android/` directory of the course material. Please `git clone` the repository to follow along.

Anahtar noktalar:

- Cuttlefish is a reference Android device designed to work on generic Linux desktops. MacOS support is also planned.
- The Cuttlefish system image maintains high fidelity to real devices, and is the ideal emulator to run many Rust use cases.

# Chapter 34

# İnşa (Build) Kuralları

The Android build system (Soong) supports Rust via a number of modules:

| Module Type | Description |
|---|---|
| `rust_binary` | Produces a Rust binary. |
| `rust_library` | Produces a Rust library, and provides both `rlib` and `dylib` variants. |
| `rust_ffi` | Produces a Rust C library usable by `cc` modules, and provides both static and shared variants. |
| `rust_proc_macro` | Produces a `proc-macro` Rust library. These are analogous to compiler plugins. |
| `rust_test` | Produces a Rust test binary that uses the standard Rust test harness. |
| `rust_fuzz` | Produces a Rust fuzz binary leveraging `libfuzzer`. |
| `rust_protobuf` | Generates source and produces a Rust library that provides an interface for a particular protobuf. |
| `rust_bindgen` | Generates source and produces a Rust library containing Rust bindings to C libraries. |

We will look at `rust_binary` and `rust_library` next.

Additional items speaker may mention:

- Cargo is not optimized for multi-language repos, and also downloads packages from the internet.
- For compliance and performance, Android must have crates in-tree. It must also interop with C/C++/Java code. Soong fills that gap.
- Soong has many similarities to Bazel, which is the open-source variant of Blaze (used in google3).
- Fun fact: Data from Star Trek is a Soong-type Android.

## 34.1 Rust Binaries

Let us start with a simple application. At the root of an AOSP checkout, create the following files:

*hello_rust/Android.bp*:

```
rust_binary {
    name: "hello_rust",
    crate_name: "hello_rust",
    srcs: ["src/main.rs"],
}
```

*hello_rust/src/main.rs*:

```
//! Rust demo.

/// Prints a greeting to standard output.
fn main() {
    println!("Rust dilinden merhaba!");
}
```

You can now build, push, and run the binary:

```
m hello_rust
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust" /data/local/tmp
adb shell /data/local/tmp/hello_rust
```

```
Hello from Rust!
```

- Go through the build steps and demonstrate them running in your emulator.
- Notice the extensive documentation comments? The Android build rules enforce that all modules have documentation. Try removing it and see what error you get.
- Stress that the Rust build rules look like the other Soong rules. This is on purpose to make it as easy to use Rust as C++ or Java.

## 34.2 Rust Libraries

You use `rust_library` to create a new Rust library for Android.

Here we declare a dependency on two libraries:

- `libgreeting`, which we define below,
- `libtextwrap`, which is a crate already vendored in `external/rust/android-crates-io/crates/`.

*hello_rust/Android.bp*:

```
rust_binary {
    name: "hello_rust_with_dep",
    crate_name: "hello_rust_with_dep",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libgreetings",
        "libtextwrap",
```

```
    ],
    prefer_rlib: true, // Need this to avoid dynamic link error.
}

rust_library {
    name: "libgreetings",
    crate_name: "greetings",
    srcs: ["src/lib.rs"],
}
```

*hello_rust/src/main.rs*:

```
//! Rust demo.

use greetings::greeting;
use textwrap::fill;

/// Prints a greeting to standard output.
fn main() {
    println!("{}", fill(&greeting("Bob"), 24));
}
```

*hello_rust/src/lib.rs*:

```
//! Greeting library.

/// Greet `name`.
pub fn greeting(name: &str) -> String {
    format!("Hello {name}, it is very nice to meet you!")
}
```

You build, push, and run the binary like before:

```
m hello_rust_with_dep
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_with_dep" /data/local/tmp
adb shell /data/local/tmp/hello_rust_with_dep
```

```
Hello Bob, it is very
nice to meet you!
```

- Go through the build steps and demonstrate them running in your emulator.
- A Rust crate named `greetings` must be built by a rule called `libgreetings`. Note how the Rust code uses the crate name, as is normal in Rust.
- Again, the build rules enforce that we add documentation comments to all public items.

# Chapter 35

# AIDL

The Android Interface Definition Language (AIDL) is supported in Rust:

- Rust code can call existing AIDL servers,
- You can create new AIDL servers in Rust.
- AIDL is what enables Android apps to interact with each other.
- Since Rust is supported as a first-class citizen in this ecosystem, Rust services can be called by any other process on the phone.

## 35.1 Doğum Günü Servisi Eğitimi

To illustrate how to use Rust with Binder, we're going to walk through the process of creating a Binder interface. We're then going to both implement the described service and write client code that talks to that service.

### 35.1.1 AIDL Interfaces

You declare the API of your service using an AIDL interface:

*birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl*:

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years);
}
```

*birthday_service/aidl/Android.bp*:

```
aidl_interface {
    name: "com.example.birthdayservice",
    srcs: ["com/example/birthdayservice/*.aidl"],
    unstable: true,
    backend: {
```

```
        rust: { // Rust is not enabled by default
            enabled: true,
        },
    },
}
```

- Note that the directory structure under the `aidl/` directory needs to match the package name used in the AIDL file, i.e. the package is `com.example.birthdayservice` and the file is at `aidl/com/example/IBirthdayService.aidl`.

### 35.1.2 Generated Service API

Binder generates a trait for each interface definition.

*birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl*:

```
/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years);
}
```

*out/soong/.intermediates/.../com_example_birthdayservice.rs*:

```
trait IBirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String>;
}
```

Your service will need to implement this trait, and your client will use this trait to talk to the service.

- Point out how the generated function signature, specifically the argument and return types, correspond the interface definition.
  - `String` for an argument results in a different Rust type than `String` as a return type.

### 35.1.3 Service Implementation

We can now implement the AIDL service:

*birthday_service/src/lib.rs*:

```
//! Implementation of the `IBirthdayService` AIDL interface.
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService:
use com_example_birthdayservice::binder;

/// The `IBirthdayService` implementation.
pub struct BirthdayService;

impl binder::Interface for BirthdayService {}

impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(&self, name: &str, years: i32) -> binder::Result<String> {
        Ok(format!("Happy Birthday {name}, congratulations with the {years} years!"))
```

```
    }
}
```

*birthday_service/Android.bp*:

```
rust_library {
    name: "libbirthdayservice",
    crate_name: "birthdayservice",
    srcs: ["src/lib.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
    ],
}
```

- Point out the path to the generated `IBirthdayService` trait, and explain why each of the segments is necessary.
- Note that `wishHappyBirthday` and other AIDL IPC methods take `&self` (instead of `&mut self`).
  - This is necessary because binder responds to incoming requests on a thread pool, allowing for multiple requests to be processed in parallel. This requires that the service methods only get a shared reference to `self`.
  - Any state that needs to be modified by the service will have to be put in something like a `Mutex` to allow for safe mutation.
  - The correct approach for managing service state depends heavily on the details of your service.
- TODO: What does the `binder::Interface` trait do? Are there methods to override? Where source?

### 35.1.4 AIDL Server

Finally, we can create a server which exposes the service:

*birthday_service/src/server.rs*:

```
//! Birthday service.
use birthdayservice::BirthdayService;
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService::
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Entry point for birthday service.
fn main() {
    let birthday_service = BirthdayService;
    let birthday_service_binder = BnBirthdayService::new_binder(
        birthday_service,
        binder::BinderFeatures::default(),
    );
    binder::add_service(SERVICE_IDENTIFIER, birthday_service_binder.as_binder())
        .expect("Failed to register service");
    binder::ProcessState::join_thread_pool();
}
```

*birthday_service/Android.bp*:

```
rust_binary {
    name: "birthday_server",
    crate_name: "birthday_server",
    srcs: ["src/server.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
        "libbirthdayservice",
    ],
    prefer_rlib: true, // To avoid dynamic link error.
}
```

The process for taking a user-defined service implementation (in this case the `BirthdayService` type, which implements the `IBirthdayService`) and starting it as a Binder service has multiple steps, and may appear more complicated than students are used to if they've used Binder from C++ or another language. Explain to students why each step is necessary.

1. Create an instance of your service type (`BirthdayService`).
2. Wrap the service object in corresponding `Bn*` type (`BnBirthdayService` in this case). This type is generated by Binder and provides the common Binder functionality that would be provided by the `BnBinder` base class in C++. We don't have inheritance in Rust, so instead we use composition, putting our `BirthdayService` within the generated `BnBinderService`.
3. Call `add_service`, giving it a service identifier and your service object (the `BnBirthdayService` object in the example).
4. Call `join_thread_pool` to add the current thread to Binder's thread pool and start listening for connections.

### 35.1.5 Dağıtmak (Deploy)

We can now build, push, and start the service:

```
m birthday_server
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_server" /data/local/tmp
adb root
adb shell /data/local/tmp/birthday_server
```

In another terminal, check that the service runs:

```
adb shell service check birthdayservice
```

```
Service birthdayservice: found
```

You can also call the service with `service call`:

```
adb shell service call birthdayservice 1 s16 Bob i32 24
```

```
Result: Parcel(
  0x00000000: 00000000 00000036 00610048 00700070 '....6...H.a.p.p.'
  0x00000010: 00200079 00690042 00740072 00640068 'y. .B.i.r.t.h.d.'
  0x00000020: 00790061 00420020 0062006f 0020002c 'a.y. .B.o.b.,. .'
  0x00000030: 006f0063 0067006e 00610072 00750074 'c.o.n.g.r.a.t.u.'
  0x00000040: 0061006c 00690074 006e006f 00200073 'l.a.t.i.o.n.s. .'
  0x00000050: 00690077 00680074 00740020 00650068 'w.i.t.h. .t.h.e.'
  0x00000060: 00320020 00200034 00650079 00720061 ' .2.4. .y.e.a.r.'
  0x00000070: 00210073 00000000                   's.!.....       ')
```

### 35.1.6 AIDL Client

Finally, we can create a Rust client for our new service.

*birthday_service/src/client.rs*:

```
use com_example_birthdayservice::aidl::com::example::birthdayservice::IBirthdayService::
use com_example_birthdayservice::binder;

const SERVICE_IDENTIFIER: &str = "birthdayservice";

/// Call the birthday service.
fn main() -> Result<(), Box<dyn Error>> {
    let name = std::env::args().nth(1).unwrap_or_else(|| String::from("Bob"));
    let years = std::env::args()
        .nth(2)
        .and_then(|arg| arg.parse::<i32>().ok())
        .unwrap_or(42);

    binder::ProcessState::start_thread_pool();
    let service = binder::get_interface::<dyn IBirthdayService>(SERVICE_IDENTIFIER)
        .map_err(|_| "Failed to connect to BirthdayService")?;

    // Call the service.
    let msg = service.wishHappyBirthday(&name, years)?;
    println!("{msg}");
}
```

*birthday_service/Android.bp*:

```
rust_binary {
    name: "birthday_client",
    crate_name: "birthday_client",
    srcs: ["src/client.rs"],
    rustlibs: [
        "com.example.birthdayservice-rust",
    ],
    prefer_rlib: true, // To avoid dynamic link error.
}
```

Notice that the client does not depend on `libbirthdayservice`.

Build, push, and run the client on your device:

```
m birthday_client
adb push "$ANDROID_PRODUCT_OUT/system/bin/birthday_client" /data/local/tmp
adb shell /data/local/tmp/birthday_client Charlie 60
```

```
Happy Birthday Charlie, congratulations with the 60 years!
```

- `Strong<dyn IBirthdayService>` is the trait object representing the service that the client has connected to.
  - `Strong` is a custom smart pointer type for Binder. It handles both an in-process ref count for the service trait object, and the global Binder ref count that tracks how many processes have a reference to the object.

  - Note that the trait object that the client uses to talk to the service uses the exact same trait that the server implements. For a given Binder interface, there is a single Rust trait generated that both client and server use.
- Use the same service identifier used when registering the service. This should ideally be defined in a common crate that both the client and server can depend on.

### 35.1.7 API'yi Değiştirme

Let us extend the API with more functionality: we want to let clients specify a list of lines for the birthday card:

```
package com.example.birthdayservice;

/** Birthday service interface. */
interface IBirthdayService {
    /** Generate a Happy Birthday message. */
    String wishHappyBirthday(String name, int years, in String[] text);
}
```

This results in an updated trait definition for `IBirthdayService`:

```
trait IBirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String>;
}
```

- Note how the `String[]` in the AIDL definition is translated as a `&[String]` in Rust, i.e. that idiomatic Rust types are used in the generated bindings wherever possible:
  - `in` array arguments are translated to slices.
  - `out` and `inout` args are translated to `&mut Vec<T>`.
  - Return values are translated to returning a `Vec<T>`.

### 35.1.8 Updating Client and Service

Update the client and server code to account for the new API.

*birthday_service/src/lib.rs*:

```
impl IBirthdayService for BirthdayService {
    fn wishHappyBirthday(
        &self,
        name: &str,
        years: i32,
        text: &[String],
    ) -> binder::Result<String> {
        let mut msg = format!(
            "Happy Birthday {name}, congratulations with the {years} years!",
        );
```

```
        for line in text {
            msg.push('\n');
            msg.push_str(line);
        }

        Ok(msg)
    }
}
```

*birthday_service/src/client.rs*:

```
let msg = service.wishHappyBirthday(
    &name,
    years,
    &[
        String::from("Habby birfday to yuuuuu"),
        String::from("And also: many more"),
    ],
)?;
```

- TODO: Move code snippets into project files where they'll actually be built?

## 35.2 Working With AIDL Types

AIDL types translate into the appropriate idiomatic Rust type:

- Primitive types map (mostly) to idiomatic Rust types.
- Collection types like slices, `Vec`s and string types are supported.
- References to AIDL objects and file handles can be sent between clients and services.
- File handles and parcelables are fully supported.

### 35.2.1 İlkel (Primitive) Türler

Primitive types map (mostly) idiomatically:

| AIDL Türü | Rust Türü | Note |
|---|---|---|
| `boolean` | `bool` | |
| `byte` | `i8` | Note that bytes are signed. |
| `char` | `u16` | Note the usage of `u16`, NOT `u32`. |
| `int` | `i32` | |
| `long` | `i64` | |
| `float` | `f32` | |
| `double` | `f64` | |
| `String` | `String` | |

### 35.2.2 Dizi Türleri

The array types (`T[]`, `byte[]`, and `List<T>`) get translated to the appropriate Rust array type depending on how they are used in the function signature:

| Position | Rust Türü |
|---|---|
| in argument | `&[T]` |
| out/inout argument | `&mut Vec<T>` |
| Return | `Vec<T>` |

- In Android 13 or higher, fixed-size arrays are supported, i.e. `T[N]` becomes `[T; N]`. Fixed-size arrays can have multiple dimensions (e.g. `int[3][4]`). In the Java backend, fixed-size arrays are represented as array types.
- Arrays in parcelable fields always get translated to `Vec<T>`.

### 35.2.3 Nesnelerin Gönderilmesi

AIDL objects can be sent either as a concrete AIDL type or as the type-erased `IBinder` interface:

*birthday_service/aidl/com/example/birthdayservice/IBirthdayInfoProvider.aidl*:

```
package com.example.birthdayservice;

interface IBirthdayInfoProvider {
    String name();
    int years();
}
```

*birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl*:

```
import com.example.birthdayservice.IBirthdayInfoProvider;

interface IBirthdayService {
    /** The same thing, but using a binder object. */
    String wishWithProvider(IBirthdayInfoProvider provider);

    /** The same thing, but using `IBinder`. */
    String wishWithErasedProvider(IBinder provider);
}
```

*birthday_service/src/client.rs*:

```
/// Rust struct implementing the `IBirthdayInfoProvider` interface.
struct InfoProvider {
    name: String,
    age: u8,
}

impl binder::Interface for InfoProvider {}

impl IBirthdayInfoProvider for InfoProvider {
    fn name(&self) -> binder::Result<String> {
        Ok(self.name.clone())
    }

    fn years(&self) -> binder::Result<i32> {
```

```
        Ok(self.age as i32)
    }
}

fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");

    // Create a binder object for the `IBirthdayInfoProvider` interface.
    let provider = BnBirthdayInfoProvider::new_binder(
        InfoProvider { name: name.clone(), age: years as u8 },
        BinderFeatures::default(),
    );

    // Send the binder object to the service.
    service.wishWithProvider(&provider)?;

    // Perform the same operation but passing the provider as an `SpIBinder`.
    service.wishWithErasedProvider(&provider.as_binder())?;
}
```

- Note the usage of `BnBirthdayInfoProvider`. This serves the same purpose as `BnBirthdayService` that we saw previously.

### 35.2.4 Parsellenebilir Gönderme

Binder for Rust supports sending parcelables directly:

*birthday_service/aidl/com/example/birthdayservice/BirthdayInfo.aidl*:

```
package com.example.birthdayservice;

parcelable BirthdayInfo {
    String name;
    int years;
}
```

*birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl*:

```
import com.example.birthdayservice.BirthdayInfo;

interface IBirthdayService {
    /** The same thing, but with a parcelable. */
    String wishWithInfo(in BirthdayInfo info);
}
```

*birthday_service/src/client.rs*:

```
fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");

    let info = BirthdayInfo { name: "Alice".into(), years: 123 };
```

```
    service.wishWithInfo(&info)?;
}
```

### 35.2.5 Dosyaların Gönderilmesi

Files can be sent between Binder clients/servers using the `ParcelFileDescriptor` type:

*birthday_service/aidl/com/example/birthdayservice/IBirthdayService.aidl*:

```
interface IBirthdayService {
    /** The same thing, but loads info from a file. */
    String wishFromFile(in ParcelFileDescriptor infoFile);
}
```

*birthday_service/src/client.rs*:

```
fn main() {
    binder::ProcessState::start_thread_pool();
    let service = connect().expect("Failed to connect to BirthdayService");

    // Open a file and put the birthday info in it.
    let mut file = File::create("/data/local/tmp/birthday.info").unwrap();
    writeln!(file, "{name}")?;
    writeln!(file, "{years}")?;

    // Create a `ParcelFileDescriptor` from the file and send it.
    let file = ParcelFileDescriptor::new(file);
    service.wishFromFile(&file)?;
}
```

*birthday_service/src/lib.rs*:

```
impl IBirthdayService for BirthdayService {
    fn wishFromFile(
        &self,
        info_file: &ParcelFileDescriptor,
    ) -> binder::Result<String> {
        // Convert the file descriptor to a `File`. `ParcelFileDescriptor` wraps
        // an `OwnedFd`, which can be cloned and then used to create a `File`
        // object.
        let mut info_file = info_file
            .as_ref()
            .try_clone()
            .map(File::from)
            .expect("Invalid file handle");

        let mut contents = String::new();
        info_file.read_to_string(&mut contents).unwrap();

        let mut lines = contents.lines();
        let name = lines.next().unwrap();
        let years: i32 = lines.next().unwrap().parse().unwrap();

        Ok(format!("Happy Birthday {name}, congratulations with the {years} years!"))
```

```
    }
}
```

- `ParcelFileDescriptor` wraps an `OwnedFd`, and so can be created from a `File` (or any other type that wraps an `OwnedFd`), and can be used to create a new `File` handle on the other side.
- Other types of file descriptors can be wrapped and sent, e.g. TCP, UDP, and UNIX sockets.

# Chapter 36

# Android'de Test Etme

Building on Testing, we will now look at how unit tests work in AOSP. Use the `rust_test` module for your unit tests:

*testing/Android.bp*:

```
rust_library {
    name: "libleftpad",
    crate_name: "leftpad",
    srcs: ["src/lib.rs"],
}

rust_test {
    name: "libleftpad_test",
    crate_name: "leftpad_test",
    srcs: ["src/lib.rs"],
    host_supported: true,
    test_suites: ["general-tests"],
}

rust_test {
    name: "libgoogletest_example",
    crate_name: "googletest_example",
    srcs: ["googletest.rs"],
    rustlibs: ["libgoogletest_rust"],
    host_supported: true,
}

rust_test {
    name: "libmockall_example",
    crate_name: "mockall_example",
    srcs: ["mockall.rs"],
    rustlibs: ["libmockall"],
    host_supported: true,
}
```

*testing/src/lib.rs*:

```rust
//! Left-padding library.

/// Left-pad `s` to `width`.
pub fn leftpad(s: &str, width: usize) -> String {
    format!("{s:>width$}")
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn short_string() {
        assert_eq!(leftpad("foo", 5), "  foo");
    }

    #[test]
    fn long_string() {
        assert_eq!(leftpad("foobar", 6), "foobar");
    }
}
```

You can now run the test with

```
atest --host libleftpad_test
```

The output looks like this:

```
INFO: Elapsed time: 2.666s, Critical Path: 2.40s
INFO: 3 processes: 2 internal, 1 linux-sandbox.
INFO: Build completed successfully, 3 total actions
//comprehensive-rust-android/testing:libleftpad_test_host            PASSED in 2.3s
    PASSED  libleftpad_test.tests::long_string (0.0s)
    PASSED  libleftpad_test.tests::short_string (0.0s)
Test cases: finished with 2 passing and 0 failing out of 2 test cases
```

Notice how you only mention the root of the library crate. Tests are found recursively in nested modules.

## 36.1 GoogleTest

The GoogleTest crate allows for flexible test assertions using *matchers*:

```rust
use googletest::prelude::*;

#[googletest::test]
fn test_elements_are() {
    let value = vec!["foo", "bar", "baz"];
    expect_that!(value, elements_are!(eq(&"foo"), lt(&"xyz"), starts_with("b")));
}
```

If we change the last element to `"!"`, the test fails with a structured error message pin-pointing the error:

```
---- test_elements_are stdout ----
Value of: value
Expected: has elements:
  0. is equal to "foo"
  1. is less than "xyz"
  2. starts with prefix "!"
Actual: ["foo", "bar", "baz"],
  where element #2 is "baz", which does not start with "!"
  at src/testing/googletest.rs:6:5
Error: See failure output above
```

This slide should take about 5 minutes.

- GoogleTest is not part of the Rust Playground, so you need to run this example in a local environment. Use `cargo add googletest` to quickly add it to an existing Cargo project.
- The `use googletest::prelude::*;` line imports a number of commonly used macros and types.
- This just scratches the surface, there are many builtin matchers. Consider going through the first chapter of "Advanced testing for Rust applications", a self-guided Rust course: it provides a guided introduction to the library, with exercises to help you get comfortable with `googletest` macros, its matchers and its overall philosophy.
- A particularly nice feature is that mismatches in multi-line strings are shown as a diff:

```rust
#[test]
fn test_multiline_string_diff() {
    let haiku = "Memory safety found,\n\
                 Rust's strong typing guides the way,\n\
                 Secure code you'll write.";
    assert_that!(
        haiku,
        eq("Memory safety found,\n\
            Rust's silly humor guides the way,\n\
            Secure code you'll write.")
    );
}
```

shows a color-coded diff (colors not shown here):

```
    Value of: haiku
Expected: is equal to "Memory safety found,\nRust's silly humor guides the way,\nSecure
Actual: "Memory safety found,\nRust's strong typing guides the way,\nSecure code you'll
  which isn't equal to "Memory safety found,\nRust's silly humor guides the way,\nSecure
Difference(-actual / +expected):
 Memory safety found,
-Rust's strong typing guides the way,
+Rust's silly humor guides the way,
 Secure code you'll write.
  at src/testing/googletest.rs:17:5
```

- The crate is a Rust port of GoogleTest for C++.

## 36.2 Taklit Etme (Mocking)

For mocking, Mockall is a widely used library. You need to refactor your code to use traits, which you can then quickly mock:

```
use std::time::Duration;

#[mockall::automock]
pub trait Pet {
    fn is_hungry(&self, since_last_meal: Duration) -> bool;
}

#[test]
fn test_robot_dog() {
    let mut mock_dog = MockPet::new();
    mock_dog.expect_is_hungry().return_const(true);
    assert!(mock_dog.is_hungry(Duration::from_secs(10)));
}
```

This slide should take about 5 minutes.

- Mockall is the recommended mocking library in Android (AOSP). There are other mocking libraries available on crates.io, in particular in the area of mocking HTTP services. The other mocking libraries work in a similar fashion as Mockall, meaning that they make it easy to get a mock implementation of a given trait.
- Note that mocking is somewhat *controversial*: mocks allow you to completely isolate a test from its dependencies. The immediate result is faster and more stable test execution. On the other hand, the mocks can be configured wrongly and return output different from what the real dependencies would do.

  If at all possible, it is recommended that you use the real dependencies. As an example, many databases allow you to configure an in-memory backend. This means that you get the correct behavior in your tests, plus they are fast and will automatically clean up after themselves.

  Similarly, many web frameworks allow you to start an in-process server which binds to a random port on `localhost`. Always prefer this over mocking away the framework since it helps you test your code in the real environment.
- Mockall is not part of the Rust Playground, so you need to run this example in a local environment. Use `cargo add mockall` to quickly add Mockall to an existing Cargo project.
- Mockall has a lot more functionality. In particular, you can set up expectations which depend on the arguments passed. Here we use this to mock a cat which becomes hungry 3 hours after the last time it was fed:

```
#[test]
fn test_robot_cat() {
    let mut mock_cat = MockPet::new();
    mock_cat
        .expect_is_hungry()
        .with(mockall::predicate::gt(Duration::from_secs(3 * 3600)))
        .return_const(true);
```

```
    mock_cat.expect_is_hungry().return_const(false);
    assert!(mock_cat.is_hungry(Duration::from_secs(5 * 3600)));
    assert!(!mock_cat.is_hungry(Duration::from_secs(5)));
}
```

- You can use `.times(n)` to limit the number of times a mock method can be called to `n` --- the mock will automatically panic when dropped if this isn't satisfied.

# Chapter 37

# Kayıt Tutma (Logging)

You should use the `log` crate to automatically log to `logcat` (on-device) or `stdout` (on-host):

*hello_rust_logs/Android.bp*:

```
rust_binary {
    name: "hello_rust_logs",
    crate_name: "hello_rust_logs",
    srcs: ["src/main.rs"],
    rustlibs: [
        "liblog_rust",
        "liblogger",
    ],
    host_supported: true,
}
```

*hello_rust_logs/src/main.rs*:

```
//! Rust logging demo.

use log::{debug, error, info};

/// Logs a greeting.
fn main() {
    logger::init(
        logger::Config::default()
            .with_tag_on_device("rust")
            .with_max_level(log::LevelFilter::Trace),
    );
    debug!("Starting program.");
    info!("Things are going fine.");
    error!("Something went wrong!");
}
```

Build, push, and run the binary on your device:

```
m hello_rust_logs
adb push "$ANDROID_PRODUCT_OUT/system/bin/hello_rust_logs" /data/local/tmp
```

```
adb shell /data/local/tmp/hello_rust_logs
```

The logs show up in `adb logcat`:

```
adb logcat -s rust
```

```
09-08 08:38:32.454  2420  2420 D rust: hello_rust_logs: Starting program.
09-08 08:38:32.454  2420  2420 I rust: hello_rust_logs: Things are going fine.
09-08 08:38:32.454  2420  2420 E rust: hello_rust_logs: Something went wrong!
```

- The logger implementation in `liblogger` is only needed in the final binary, if you're logging from a library you only need the `log` facade crate.

# Chapter 38

# Birlikte Çalışabilirlik (Interoperability)

Rust has excellent support for interoperability with other languages. This means that you can:

- Call Rust functions from other languages.
- Call functions written in other languages from Rust.

When you call functions in a foreign language we say that you're using a *foreign function interface*, also known as FFI.

- This is a key ability of Rust: compiled code becomes indistinguishable from compiled C or C++ code.
- Technically, we say that Rust can be compiled to the same ABI (application binary interface) as C code.

## 38.1 Interoperability with C

Rust has full support for linking object files with a C calling convention. Similarly, you can export Rust functions and call them from C.

You can do it by hand if you want:

```
unsafe extern "C" {
    safe fn abs(x: i32) -> i32;
}

fn main() {
    let x = -42;
    let abs_x = abs(x);
    println!("{x}, {abs_x}");
}
```

We already saw this in the Safe FFI Wrapper exercise.

This assumes full knowledge of the target platform. Not recommended for production.

We will look at better options next.

- The `"C"` part of the `extern` block tells Rust that `abs` can be called using the C ABI (application binary interface).
- The `safe fn abs` part tells that Rust that `abs` is a safe function. By default, extern functions are considered unsafe, but since `abs(x)` is valid for any `x`, we can declare it safe.

### 38.1.1 Basit bir C Kütüphanesi

Let's first create a small C library:

*interoperability/bindgen/libbirthday.h*:

```
typedef struct card {
  const char* name;
  int years;
} card;

void print_card(const card* card);
```

*interoperability/bindgen/libbirthday.c*:

```
#include <stdio.h>
#include "libbirthday.h"

void print_card(const card* card) {
  printf("+--------------\n");
  printf("| Happy Birthday %s!\n", card->name);
  printf("| Congratulations with the %i years!\n", card->years);
  printf("+--------------\n");
}
```

Add this to your `Android.bp` file:

*interoperability/bindgen/Android.bp*:

```
cc_library {
    name: "libbirthday",
    srcs: ["libbirthday.c"],
}
```

### 38.1.2 Using Bindgen

The bindgen tool can auto-generate bindings from a C header file.

Create a wrapper header file for the library (not strictly needed in this example):

*interoperability/bindgen/libbirthday_wrapper.h*:

```
#include "libbirthday.h"
```

*interoperability/bindgen/Android.bp*:

```
rust_bindgen {
    name: "libbirthday_bindgen",
    crate_name: "birthday_bindgen",
    wrapper_src: "libbirthday_wrapper.h",
    source_stem: "bindings",
    static_libs: ["libbirthday"],
}
```

Finally, we can use the bindings in our Rust program:

*interoperability/bindgen/Android.bp*:

```
rust_binary {
    name: "print_birthday_card",
    srcs: ["main.rs"],
    rustlibs: ["libbirthday_bindgen"],
    static_libs: ["libbirthday"],
}
```

*interoperability/bindgen/main.rs*:

```
//! Bindgen demo.

use birthday_bindgen::{card, print_card};

fn main() {
    let name = std::ffi::CString::new("Peter").unwrap();
    let card = card { name: name.as_ptr(), years: 42 };
    // SAFETY: The pointer we pass is valid because it came from a Rust
    // reference, and the `name` it contains refers to `name` above which also
    // remains valid. `print_card` doesn't store either pointer to use later
    // after it returns.
    unsafe {
        print_card(&card);
    }
}
```

- The Android build rules will automatically call `bindgen` for you behind the scenes.
- Notice that the Rust code in `main` is still hard to write. It is good practice to encapsulate the output of `bindgen` in a Rust library which exposes a safe interface to caller.

### 38.1.3 İkili Dosyamızı Çalıştırmak

Build, push, and run the binary on your device:

```
m print_birthday_card
adb push "$ANDROID_PRODUCT_OUT/system/bin/print_birthday_card" /data/local/tmp
adb shell /data/local/tmp/print_birthday_card
```

Finally, we can run auto-generated tests to ensure the bindings work:

*interoperability/bindgen/Android.bp*:

```
rust_test {
    name: "libbirthday_bindgen_test",
```

```
    srcs: [":libbirthday_bindgen"],
    crate_name: "libbirthday_bindgen_test",
    test_suites: ["general-tests"],
    auto_gen_config: true,
    clippy_lints: "none", // Generated file, skip linting
    lints: "none",
}
```

```
atest libbirthday_bindgen_test
```

### 38.1.4 Basit bir Rust Kütüphanesi

Exporting Rust functions and types to C is easy. Here's a simple Rust library:

*interoperability/rust/libanalyze/analyze.rs*

```
//! Rust FFI demo.
#![deny(improper_ctypes_definitions)]

use std::os::raw::c_int;

/// Analyze the numbers.
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
pub extern "C" fn analyze_numbers(x: c_int, y: c_int) {
    if x < y {
        println!("x ({x}) is smallest!");
    } else {
        println!("y ({y}) is probably larger than x ({x})");
    }
}
```

*interoperability/rust/libanalyze/Android.bp*

```
rust_ffi {
    name: "libanalyze_ffi",
    crate_name: "analyze_ffi",
    srcs: ["analyze.rs"],
    include_dirs: ["."],
}
```

`#[unsafe(no_mangle)]` disables Rust's usual name mangling, so the exported symbol will just be the name of the function. You can also use `#[unsafe(export_name = "some_name")]` to specify whatever name you want.

### 38.1.5 Calling Rust

We can now call this from a C binary:

*interoperability/rust/libanalyze/analyze.h*

```
#ifndef ANALYZE_H
#define ANALYZE_H
```

```
void analyze_numbers(int x, int y);

#endif
```

*interoperability/rust/analyze/main.c*

```
#include "analyze.h"

int main() {
  analyze_numbers(10, 20);
  analyze_numbers(123, 123);
  return 0;
}
```

*interoperability/rust/analyze/Android.bp*

```
cc_binary {
    name: "analyze_numbers",
    srcs: ["main.c"],
    static_libs: ["libanalyze_ffi"],
}
```

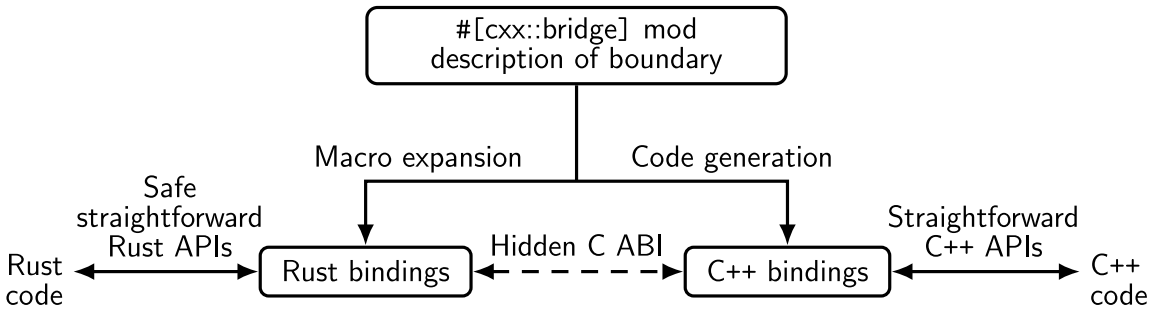
Build, push, and run the binary on your device:

```
m analyze_numbers
adb push "$ANDROID_PRODUCT_OUT/system/bin/analyze_numbers" /data/local/tmp
adb shell /data/local/tmp/analyze_numbers
```

## 38.2 C++ ile

The CXX crate makes it possible to do safe interoperability between Rust and C++.

The overall approach looks like this:




### 38.2.1 Köprü Modülü

CXX relies on a description of the function signatures that will be exposed from each language to the other. You provide this description using extern blocks in a Rust module annotated with the `#[cxx::bridge]` attribute macro.

```
#[allow(unsafe_op_in_unsafe_fn)]
#[cxx::bridge(namespace = "org::blobstore")]
mod ffi {
```

```
    // Shared structs with fields visible to both languages.
    struct BlobMetadata {
        size: usize,
        tags: Vec<String>,
    }

    // Rust types and signatures exposed to C++.
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    // C++ types and signatures exposed to Rust.
    unsafe extern "C++" {
        include!("include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
        fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    }
}
```

- The bridge is generally declared in an `ffi` module within your crate.
- From the declarations made in the bridge module, CXX will generate matching Rust and C++ type/function definitions in order to expose those items to both languages.
- To view the generated Rust code, use cargo-expand to view the expanded proc macro. For most of the examples you would use `cargo expand ::ffi` to expand just the `ffi` module (though this doesn't apply for Android projects).
- To view the generated C++ code, look in `target/cxxbridge`.

### 38.2.2 Rust Bridge Declarations

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type MyType; // Opaque type
        fn foo(&self); // Method on `MyType`
        fn bar() -> Box<MyType>; // Free function
    }
}

struct MyType(i32);

impl MyType {
    fn foo(&self) {
        println!("{}", self.0);
    }
```

```
}

fn bar() -> Box<MyType> {
    Box::new(MyType(123))
}
```

- Items declared in the `extern "Rust"` reference items that are in scope in the parent module.
- The CXX code generator uses your `extern "Rust"` section(s) to produce a C++ header file containing the corresponding C++ declarations. The generated header has the same path as the Rust source file containing the bridge, except with a .rs.h file extension.

### 38.2.3 Oluşturulan (Generated) C++

```
#[cxx::bridge]
mod ffi {
    // Rust types and signatures exposed to C++.
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }
}
```

Results in (roughly) the following C++:

```
struct MultiBuf final : public ::rust::Opaque {
  ~MultiBuf() = delete;

private:
  friend ::rust::layout;
  struct layout {
    static ::std::size_t size() noexcept;
    static ::std::size_t align() noexcept;
  };
};

::rust::Slice<::std::uint8_t const> next_chunk(::org::blobstore::MultiBuf &buf) noexcept
```

### 38.2.4 C++ Bridge Declarations

```
#[cxx::bridge]
mod ffi {
    // C++ types and signatures exposed to Rust.
    unsafe extern "C++" {
        include!("include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: Pin<&mut BlobstoreClient>, parts: &mut MultiBuf) -> u64;
        fn tag(self: Pin<&mut BlobstoreClient>, blobid: u64, tag: &str);
```

```
        fn metadata(&self, blobid: u64) -> BlobMetadata;
    }
}
```

Results in (roughly) the following Rust:

```
#[repr(C)]
pub struct BlobstoreClient {
    _private: ::cxx::private::Opaque,
}

pub fn new_blobstore_client() -> ::cxx::UniquePtr<BlobstoreClient> {
    extern "C" {
        #[link_name = "org$blobstore$cxxbridge1$new_blobstore_client"]
        fn __new_blobstore_client() -> *mut BlobstoreClient;
    }
    unsafe { ::cxx::UniquePtr::from_raw(__new_blobstore_client()) }
}

impl BlobstoreClient {
    pub fn put(&self, parts: &mut MultiBuf) -> u64 {
        extern "C" {
            #[link_name = "org$blobstore$cxxbridge1$BlobstoreClient$put"]
            fn __put(
                _: &BlobstoreClient,
                parts: *mut ::cxx::core::ffi::c_void,
            ) -> u64;
        }
        unsafe {
            __put(self, parts as *mut MultiBuf as *mut ::cxx::core::ffi::c_void)
        }
    }
}

// ...
```

- The programmer does not need to promise that the signatures they have typed in are accurate. CXX performs static assertions that the signatures exactly correspond with what is declared in C++.
- `unsafe extern` blocks allow you to declare C++ functions that are safe to call from Rust.

### 38.2.5 Paylaşılan (Shared) Türler

```
#[cxx::bridge]
mod ffi {
    #[derive(Clone, Debug, Hash)]
    struct PlayingCard {
        suit: Suit,
        value: u8,  // A=1, J=11, Q=12, K=13
    }
```

```
    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

- Only C-like (unit) enums are supported.
- A limited number of traits are supported for `#[derive()]` on shared types. Corresponding functionality is also generated for the C++ code, e.g. if you derive `Hash` also generates an implementation of `std::hash` for the corresponding C++ type.

### 38.2.6 Paylaşılan (Shared) Enum'lar

```
#[cxx::bridge]
mod ffi {
    enum Suit {
        Clubs,
        Diamonds,
        Hearts,
        Spades,
    }
}
```

Oluşturulan (Generated) Rust:

```
#[derive(Copy, Clone, PartialEq, Eq)]
#[repr(transparent)]
pub struct Suit {
    pub repr: u8,
}

#[allow(non_upper_case_globals)]
impl Suit {
    pub const Clubs: Self = Suit { repr: 0 };
    pub const Diamonds: Self = Suit { repr: 1 };
    pub const Hearts: Self = Suit { repr: 2 };
    pub const Spades: Self = Suit { repr: 3 };
}
```

Generated C++:

```
enum class Suit : uint8_t {
  Clubs = 0,
  Diamonds = 1,
  Hearts = 2,
  Spades = 3,
};
```

- On the Rust side, the code generated for shared enums is actually a struct wrapping a numeric value. This is because it is not UB in C++ for an enum class to hold a value different from all of the listed variants, and our Rust representation needs to have the same behavior.

### 38.2.7 Rust Hata İşleme

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn fallible(depth: usize) -> anyhow::Result<String> {
    if depth == 0 {
        return Err(anyhow::Error::msg("fallible1 requires depth > 0"));
    }

    Ok("Success!".into())
}
```

- Rust functions that return `Result` are translated to exceptions on the C++ side.
- The exception thrown will always be of type `rust::Error`, which primarily exposes a way to get the error message string. The error message will come from the error type's `Display` impl.
- A panic unwinding from Rust to C++ will always cause the process to immediately terminate.

### 38.2.8 C++ Hata İşleme

```
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("example/include/example.h");
        fn fallible(depth: usize) -> Result<String>;
    }
}

fn main() {
    if let Err(err) = ffi::fallible(99) {
        eprintln!("Error: {}", err);
        process::exit(1);
    }
}
```

- C++ functions declared to return a `Result` will catch any thrown exception on the C++ side and return it as an `Err` value to the calling Rust function.
- If an exception is thrown from an extern "C++" function that is not declared by the CXX bridge to return `Result`, the program calls C++'s `std::terminate`. The behavior is equivalent to the same exception being thrown through a `noexcept` C++ function.

### 38.2.9 Ek Türler

| Rust Türü | C++ Türü |
|---|---|
| `String` | `rust::String` |
| `&str` | `rust::Str` |
| `CxxString` | `std::string` |
| `&[T]/&mut [T]` | `rust::Slice` |
| `Box<T>` | `rust::Box<T>` |
| `UniquePtr<T>` | `std::unique_ptr<T>` |
| `Vec<T>` | `rust::Vec<T>` |
| `CxxVector<T>` | `std::vector<T>` |

- These types can be used in the fields of shared structs and the arguments and returns of extern functions.
- Note that Rust's `String` does not map directly to `std::string`. There are a few reasons for this:
  - `std::string` does not uphold the UTF-8 invariant that `String` requires.
  - The two types have different layouts in memory and so can't be passed directly between languages.
  - `std::string` requires move constructors that don't match Rust's move semantics, so a `std::string` can't be passed by value to Rust.

### 38.2.10 Android'de İnşa Etme (Build)

Create two genrules: One to generate the CXX header, and one to generate the CXX source file. These are then used as inputs to the `cc_library_static`.

```
// Generate a C++ header containing the C++ bindings
// to the Rust exported functions in lib.rs.
genrule {
    name: "libcxx_test_bridge_header",
    tools: ["cxxbridge"],
    cmd: "$(location cxxbridge) $(in) --header > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.h"],
}

// Generate the C++ code that Rust calls into.
genrule {
    name: "libcxx_test_bridge_code",
    tools: ["cxxbridge"],
    cmd: "$(location cxxbridge) $(in) > $(out)",
    srcs: ["lib.rs"],
    out: ["lib.rs.cc"],
}
```

- The `cxxbridge` tool is a standalone tool that generates the C++ side of the bridge module. It is included in Android and available as a Soong tool.
- By convention, if your Rust source file is `lib.rs` your header file will be named `lib.rs.h` and your source file will be named `lib.rs.cc`. This naming convention isn't enforced, though.

### 38.2.11 Android'de İnşa Etme (Build)

Create a `cc_library_static` to build the C++ library, including the CXX generated header and source file.

```
cc_library_static {
    name: "libcxx_test_cpp",
    srcs: ["cxx_test.cpp"],
    generated_headers: [
        "cxx-bridge-header",
        "libcxx_test_bridge_header"
    ],
    generated_sources: ["libcxx_test_bridge_code"],
}
```

- Point out that `libcxx_test_bridge_header` and `libcxx_test_bridge_code` are the dependencies for the CXX-generated C++ bindings. We'll show how these are setup on the next slide.
- Note that you also need to depend on the `cxx-bridge-header` library in order to pull in common CXX definitions.
- Full docs for using CXX in Android can be found in the Android docs. You may want to share that link with the class so that students know where they can find these instructions again in the future.

### 38.2.12 Android'de İnşa Etme (Build)

Create a `rust_binary` that depends on `libcxx` and your `cc_library_static`.

```
rust_binary {
    name: "cxx_test",
    srcs: ["lib.rs"],
    rustlibs: ["libcxx"],
    static_libs: ["libcxx_test_cpp"],
}
```

## 38.3 Interoperability with Java

Java can load shared objects via Java Native Interface (JNI). The `jni` crate allows you to create a compatible library.

First, we create a Rust function to export to Java:

*interoperability/java/src/lib.rs*:

```
//! Rust <-> Java FFI demo.

use jni::JNIEnv;
use jni::objects::{JClass, JString};
use jni::sys::jstring;

/// HelloWorld::hello method implementation.
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
```

```
pub extern "system" fn Java_HelloWorld_hello(
    mut env: JNIEnv,
    _class: JClass,
    name: JString,
) -> jstring {
    let input: String = env.get_string(&name).unwrap().into();
    let greeting = format!("Merhaba, {input}!");
    let output = env.new_string(greeting).unwrap();
    output.into_raw()
}
```

*interoperability/java/Android.bp*:

```
rust_ffi_shared {
    name: "libhello_jni",
    crate_name: "hello_jni",
    srcs: ["src/lib.rs"],
    rustlibs: ["libjni"],
}
```

We then call this function from Java:

*interoperability/java/HelloWorld.java*:

```
class HelloWorld {
    private static native String hello(String name);

    static {
        System.loadLibrary("hello_jni");
    }

    public static void main(String[] args) {
        String output = HelloWorld.hello("Alice");
        System.out.println(output);
    }
}
```

*interoperability/java/Android.bp*:

```
java_binary {
    name: "helloworld_jni",
    srcs: ["HelloWorld.java"],
    main_class: "HelloWorld",
    jni_libs: ["libhello_jni"],
}
```

Finally, you can build, sync, and run the binary:

```
m helloworld_jni
adb sync  # requires adb root && adb remount
adb shell /system/bin/helloworld_jni
```

- The `unsafe(no_mangle)` attribute instructs Rust to emit the `Java_HelloWorld_hello` symbol exactly as written. This is important so that Java can recognize the symbol as a `hello` method on the `HelloWorld` class.

– By default, Rust will mangle (rename) symbols so that a binary can link in two versions of the same Rust crate.

# Part X

# Chromium

# Chapter 39

# Chromium'daki Rust'a Hoş Geldiniz

Rust is supported for third-party libraries in Chromium, with first-party glue code to connect between Rust and existing Chromium C++ code.

> Today, we'll call into Rust to do something silly with strings. If you've got a corner of the code where you're displaying a UTF8 string to the user, feel free to follow this recipe in your part of the codebase instead of the exact part we talk about.

# Chapter 40

# Kurulum (Setup)

Make sure you can build and run Chromium. Any platform and set of build flags is OK, so long as your code is relatively recent (commit position 1223636 onwards, corresponding to November 2023):

```
gn gen out/Debug
autoninja -C out/Debug chrome
out/Debug/chrome # or on Mac, out/Debug/Chromium.app/Contents/MacOS/Chromium
```

(A component, debug build is recommended for quickest iteration time. This is the default!)

See How to build Chromium if you aren't already at that point. Be warned: setting up to build Chromium takes time.

It's also recommended that you have Visual Studio code installed.

# Alıştırmalar hakkında

This part of the course has a series of exercises which build on each other. We'll be doing them spread throughout the course instead of just at the end. If you don't have time to complete a certain part, don't worry: you can catch up in the next slot.

# Chapter 41

# Chromium ve Cargo Ekosistemlerinin Karşılaştırılması

The Rust community typically uses `cargo` and libraries from crates.io. Chromium is built using `gn` and `ninja` and a curated set of dependencies.

When writing code in Rust, your choices are:

- Use `gn` and `ninja` with the help of the templates from `//build/rust/*.gni` (e.g. `rust_static_library` that we'll meet later). This uses Chromium's audited toolchain and crates.
- Use `cargo`, but restrict yourself to Chromium's audited toolchain and crates
- Use `cargo`, trusting a toolchain and/or crates downloaded from the internet

From here on we'll be focusing on `gn` and `ninja`, because this is how Rust code can be built into the Chromium browser. At the same time, Cargo is an important part of the Rust ecosystem and you should keep it in your toolbox.

## Küçük bir alıştırma

Split into small groups and:

- Brainstorm scenarios where `cargo` may offer an advantage and assess the risk profile of these scenarios.
- Discuss which tools, libraries, and groups of people need to be trusted when using `gn` and `ninja`, offline `cargo`, etc.

Ask students to avoid peeking at the speaker notes before completing the exercise. Assuming folks taking the course are physically together, ask them to discuss in small groups of 3-4 people.

Notes/hints related to the first part of the exercise (”scenarios where Cargo may offer an advantage”):

- It's fantastic that when writing a tool, or prototyping a part of Chromium, one has access to the rich ecosystem of crates.io libraries. There is a crate for almost anything and they are usually quite pleasant to use. (`clap` for command-line parsing, `serde` for

serializing/deserializing to/from various formats, `itertools` for working with iterators, etc.).

  - `cargo` makes it easy to try a library (just add a single line to `Cargo.toml` and start writing code)
  - It may be worth comparing how CPAN helped make `perl` a popular choice. Or comparing with `python` + `pip`.

- Development experience is made really nice not only by core Rust tools (e.g. using `rustup` to switch to a different `rustc` version when testing a crate that needs to work on nightly, current stable, and older stable) but also by an ecosystem of third-party tools (e.g. Mozilla provides `cargo vet` for streamlining and sharing security audits; `criterion` crate gives a streamlined way to run benchmarks).

  - `cargo` makes it easy to add a tool via `cargo install --locked cargo-vet`.
  - It may be worth comparing with Chrome Extensions or VScode extensions.

- Broad, generic examples of projects where `cargo` may be the right choice:

  - Perhaps surprisingly, Rust is becoming increasingly popular in the industry for writing command line tools. The breadth and ergonomics of libraries is comparable to Python, while being more robust (thanks to the rich typesystem) and running faster (as a compiled, rather than interpreted language).
  - Participating in the Rust ecosystem requires using standard Rust tools like Cargo. Libraries that want to get external contributions, and want to be used outside of Chromium (e.g. in Bazel or Android/Soong build environments) should probably use Cargo.

- Examples of Chromium-related projects that are `cargo`-based:

  - `serde_json_lenient` (experimented with in other parts of Google which resulted in PRs with performance improvements)
  - Fontations libraries like `font-types`
  - `gnrt` tool (we will meet it later in the course) which depends on `clap` for command-line parsing and on `toml` for configuration files.
    * Disclaimer: a unique reason for using `cargo` was unavailability of gn when building and bootstrapping Rust standard library when building Rust toolchain.
    * `run_gnrt.py` uses Chromium's copy of `cargo` and `rustc`. `gnrt` depends on third-party libraries downloaded from the internet, but `run_gnrt.py` asks `cargo` that only `--locked` content is allowed via `Cargo.lock`.)

Students may identify the following items as being implicitly or explicitly trusted:

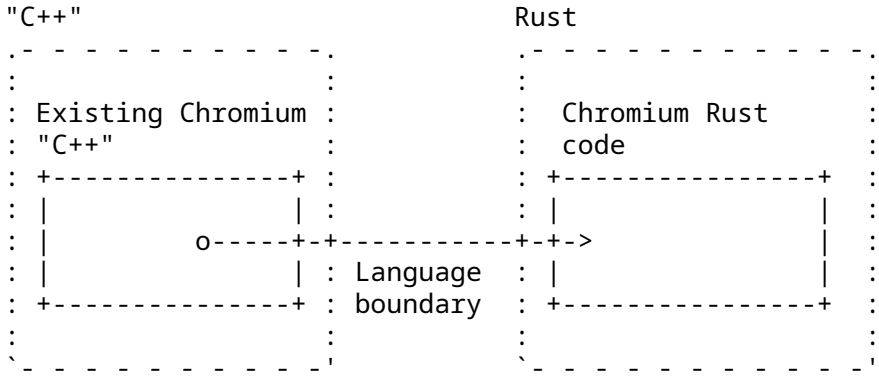
- `rustc` (the Rust compiler) which in turn depends on the LLVM libraries, the Clang compiler, the `rustc` sources (fetched from GitHub, reviewed by Rust compiler team), binary Rust compiler downloaded for bootstrapping
- `rustup` (it may be worth pointing out that `rustup` is developed under the umbrella of the https://github.com/rust-lang/ organization - same as `rustc`)
- `cargo`, `rustfmt`, etc.
- Various internal infrastructure (bots that build `rustc`, system for distributing the prebuilt toolchain to Chromium engineers, etc.)
- Cargo tools like `cargo audit`, `cargo vet`, etc.
- Rust libraries vendored into `//third_party/rust` (audited by security@chromium.org)
- Other Rust libraries (some niche, some quite popular and commonly used)

# Chapter 42

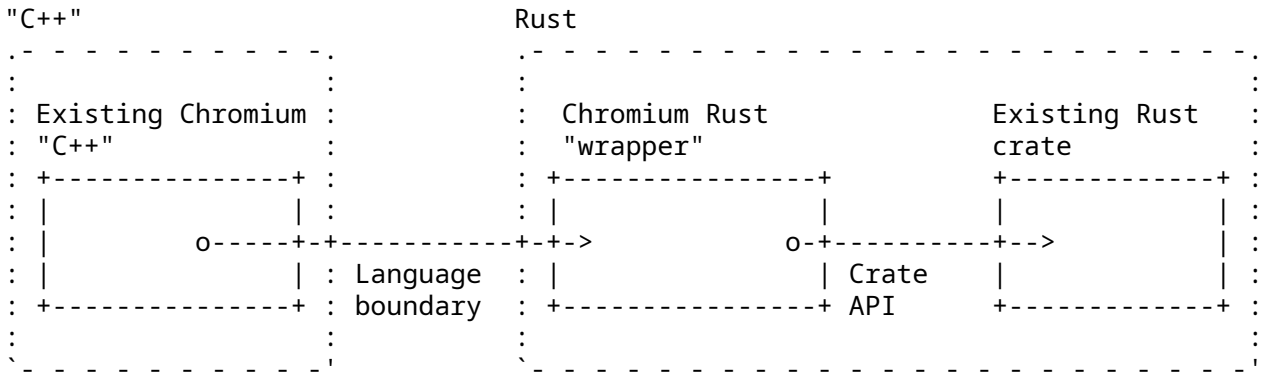
# Chromium Rust policy

Chromium's Rust policy can be found here. Rust can be used for both first-party and third-party code.

Using Rust for pure first-party code looks like this:




The third-party case is also common. It's likely that you'll also need a small amount of first-party glue code, because very few Rust libraries directly expose a C/C++ API.




The scenario of using a third-party crate is the more complex one, so today's course will focus on:

- Bringing in third-party Rust libraries (”crates”)
- Writing glue code to be able to use those crates from Chromium C++. (The same techniques are used when working with first-party Rust code).

# Chapter 43

# İnşa (Build) Kuralları

Rust code is usually built using `cargo`. Chromium builds with gn and `ninja` for efficiency --- its static rules allow maximum parallelism. Rust is no exception.

## Adding Rust code to Chromium

In some existing Chromium `BUILD.gn` file, declare a `rust_static_library`:

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}
```

You can also add `deps` on other Rust targets. Later we'll use this to depend upon third party code.

You must specify *both* the crate root, *and* a full list of sources. The `crate_root` is the file given to the Rust compiler representing the root file of the compilation unit --- typically `lib.rs`. `sources` is a complete list of all source files which `ninja` needs in order to determine when rebuilds are necessary.

(There's no such thing as a Rust `source_set`, because in Rust, an entire crate is a compilation unit. A `static_library` is the smallest unit.)

Students might be wondering why we need a gn template, rather than using gn's built-in support for Rust static libraries. The answer is that this template provides support for CXX interop, Rust features, and unit tests, some of which we'll use later.

## 43.1 Including `unsafe` Rust Code

Unsafe Rust code is forbidden in `rust_static_library` by default --- it won't compile. If you need unsafe Rust code, add `allow_unsafe = true` to the gn target. (Later in the course we'll see circumstances where this is necessary.)

```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [
    "lib.rs",
    "hippopotamus.rs"
  ]
  allow_unsafe = true
}
```

## 43.2 Chromium C++'dan Rust Koduna Bağımlılık

Simply add the above target to the `deps` of some Chromium C++ target.

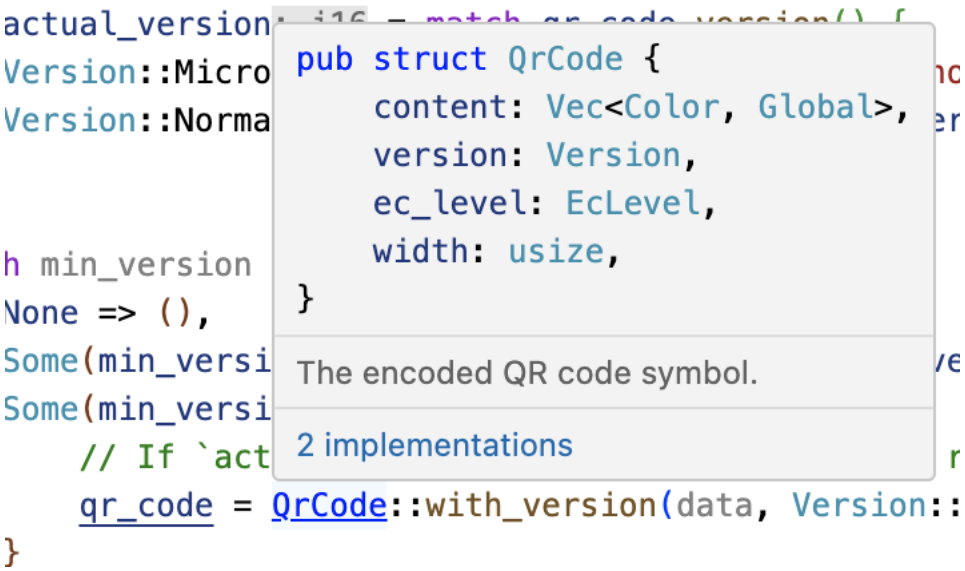
```
import("//build/rust/rust_static_library.gni")

rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
}

# or source_set, static_library etc.
component("preexisting_cpp") {
  deps = [ ":my_rust_lib" ]
}
```

We'll see that this relationship only works if the Rust code exposes plain C APIs which can be called from C++, or if we use a C++/Rust interop tool.

## 43.3 Visual Studio Code

Types are elided in Rust code, which makes a good IDE even more useful than for C++. Visual Studio code works well for Rust in Chromium. To use it,

- Ensure your VSCode has the `rust-analyzer` extension, not earlier forms of Rust support
- `gn gen out/Debug --export-rust-project` (or equivalent for your output directory)
- `ln -s out/Debug/rust-project.json rust-project.json`




A demo of some of the code annotation and exploration features of rust-analyzer might be beneficial if the audience are naturally skeptical of IDEs.

The following steps may help with the demo (but feel free to instead use a piece of Chromium-related Rust that you are most familiar with):

- Open `components/qr_code_generator/qr_code_generator_ffi_glue.rs`
- Place the cursor over the `QrCode::new` call (around line 26) in 'qr_code_generator_ffi_glue.rs
- Demo **show documentation** (typical bindings: vscode = ctrl k i; vim/CoC = K).
- Demo **go to definition** (typical bindings: vscode = F12; vim/CoC = g d). (This will take you to `//third_party/rust/.../qr_code-.../src/lib.rs`.)
- Demo **outline** and navigate to the `QrCode::with_bits` method (around line 164; the outline is in the file explorer pane in vscode; typical vim/CoC bindings = space o)
- Demo **type annotations** (there are quite a few nice examples in the `QrCode::with_bits` method)

It may be worth pointing out that `gn gen ... --export-rust-project` will need to be rerun after editing `BUILD.gn` files (which we will do a few times throughout the exercises in this session).

## 43.4 İnşa (build) kuralları alıştırması

In your Chromium build, add a new Rust target to `//ui/base/BUILD.gn` containing:

```
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
pub extern "C" fn hello_from_rust() {
    println!("Rust dilinden merhaba!")
}
```

**Important**: note that `no_mangle` here is considered a type of unsafety by the Rust compiler, so you'll need to allow unsafe code in your gn target.

Add this new Rust target as a dependency of `//ui/base:base`. Declare this function at the

top of `ui/base/resource/resource_bundle.cc` (later, we'll see how this can be automated by bindings generation tools):

```
extern "C" void hello_from_rust();
```

Call this function from somewhere in `ui/base/resource/resource_bundle.cc` - we suggest the top of `ResourceBundle::MaybeMangleLocalizedString`. Build and run Chromium, and ensure that "Hello from Rust!" is printed lots of times.

If you use VSCode, now set up Rust to work well in VSCode. It will be useful in subsequent exercises. If you've succeeded, you will be able to use right-click "Go to definition" on `println!`.

## Where to find help

- The options available to the `rust_static_library` gn template
- Information about `#[unsafe(no_mangle)]`
- Information about `extern "C"`
- Information about gn's `--export-rust-project` switch
- How to install rust-analyzer in VSCode

It's really important that students get this running, because future exercises will build on it.

This example is unusual because it boils down to the lowest-common-denominator interop language, C. Both C++ and Rust can natively declare and call C ABI functions. Later in the course, we'll connect C++ directly to Rust.

`allow_unsafe = true` is required here because `#[unsafe(no_mangle)]` might allow Rust to generate two functions with the same name, and Rust can no longer guarantee that the right one is called.

If you need a pure Rust executable, you can also do that using the `rust_executable` gn template.

# Chapter 44

# Test Etme

Rust community typically authors unit tests in a module placed in the same source file as the code being tested. This was covered earlier in the course and looks like this:

```
#[cfg(test)]
mod tests {
    #[test]
    fn my_test() {
        todo!()
    }
}
```

In Chromium we place unit tests in a separate source file and we continue to follow this practice for Rust --- this makes tests consistently discoverable and helps to avoid rebuilding `.rs` files a second time (in the `test` configuration).

This results in the following options for testing Rust code in Chromium:

- Native Rust tests (i.e. `#[test]`). Discouraged outside of `//third_party/rust`.
- `gtest` tests authored in C++ and exercising Rust via FFI calls. Sufficient when Rust code is just a thin FFI layer and the existing unit tests provide sufficient coverage for the feature.
- `gtest` tests authored in Rust and using the crate under test through its public API (using `pub mod for_testing { ... }` if needed). This is the subject of the next few slides.

Mention that native Rust tests of third-party crates should eventually be exercised by Chromium bots. (Such testing is needed rarely --- only after adding or updating third-party crates.)

Some examples may help illustrate when C++ `gtest` vs Rust `gtest` should be used:

- QR has very little functionality in the first-party Rust layer (it's just a thin FFI glue) and therefore uses the existing C++ unit tests for testing both the C++ and the Rust implementation (parameterizing the tests so they enable or disable Rust using a `ScopedFeatureList`).
- Hypothetical/WIP PNG integration may need to implement memory-safe implementation of pixel transformations that are provided by `libpng` but missing in the png crate - e.g. RGBA => BGRA, or gamma correction. Such functionality may benefit from separate tests authored in Rust.

## 44.1 `rust_gtest_interop` Kütüphanesi

The rust_gtest_interop library provides a way to:

- Use a Rust function as a `gtest` testcase (using the `#[gtest(...)]` attribute)
- Use `expect_eq!` and similar macros (similar to `assert_eq!` but not panicking and not terminating the test when the assertion fails).

Örnek:

```
use rust_gtest_interop::prelude::*;

#[gtest(MyRustTestSuite, MyAdditionTest)]
fn test_addition() {
    expect_eq!(2 + 2, 4);
}
```

## 44.2 Rust Testleri için GN Kuralları

The simplest way to build Rust `gtest` tests is to add them to an existing test binary that already contains tests authored in C++. For example:

```
test("ui_base_unittests") {
  ...
  sources += [ "my_rust_lib_unittest.rs" ]
  deps += [ ":my_rust_lib" ]
}
```

Authoring Rust tests in a separate `static_library` also works, but requires manually declaring the dependency on the support libraries:

```
rust_static_library("my_rust_lib_unittests") {
  testonly = true
  is_gtest_unittests = true
  crate_root = "my_rust_lib_unittest.rs"
  sources = [ "my_rust_lib_unittest.rs" ]
  deps = [
    ":my_rust_lib",
    "//testing/rust_gtest_interop",
  ]
}

test("ui_base_unittests") {
  ...
  deps += [ ":my_rust_lib_unittests" ]
}
```

## 44.3 `chromium::import!` Makrosu

After adding `:my_rust_lib` to GN deps, we still need to learn how to import and use `my_rust_lib` from `my_rust_lib_unittest.rs`. We haven't provided an explicit `crate_name` for `my_rust_lib` so its crate name is computed based on the full target path

and name. Fortunately we can avoid working with such an unwieldy name by using the `chromium::import!` macro from the automatically-imported `chromium` crate:

```
chromium::import! {
    "//ui/base:my_rust_lib";
}

use my_rust_lib::my_function_under_test;
```

Under the covers the macro expands to something similar to:

```
extern crate ui_sbase_cmy_urust_ulib as my_rust_lib;

use my_rust_lib::my_function_under_test;
```

More information can be found in the doc comment of the `chromium::import` macro.

`rust_static_library` supports specifying an explicit name via `crate_name` property, but doing this is discouraged. And it is discouraged because the crate name has to be globally unique. crates.io guarantees uniqueness of its crate names so `cargo_crate` GN targets (generated by the `gnrt` tool covered in a later section) use short crate names.

## 44.4 Testing exercise

Time for another exercise!

In your Chromium build:

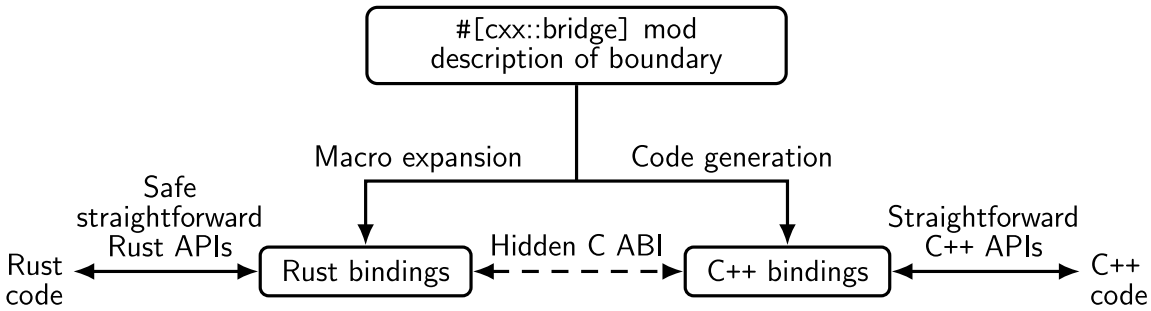
- Add a testable function next to `hello_from_rust`. Some suggestions: adding two integers received as arguments, computing the nth Fibonacci number, summing integers in a slice, etc.
- Add a separate `..._unittest.rs` file with a test for the new function.
- Add the new tests to `BUILD.gn`.
- Build the tests, run them, and verify that the new test works.

# Chapter 45

# C++ ile Birlikte Çalışabilirlik (Interoperability)

The Rust community offers multiple options for C++/Rust interop, with new tools being developed all the time. At the moment, Chromium uses a tool called CXX.

You describe your whole language boundary in an interface definition language (which looks a lot like Rust) and then CXX tools generate declarations for functions and types in both Rust and C++.




See the CXX tutorial for a full example of using this.

Talk through the diagram. Explain that behind the scenes, this is doing just the same as you previously did. Point out that automating the process has the following benefits:

- The tool guarantees that the C++ and Rust sides match (e.g. you get compile errors if the `#[cxx::bridge]` doesn't match the actual C++ or Rust definitions, but with out-of-sync manual bindings you'd get Undefined Behavior)
- The tool automates generation of FFI thunks (small, C-ABI-compatible, free functions) for non-C features (e.g. enabling FFI calls into Rust or C++ methods; manual bindings would require authoring such top-level, free functions manually)
- The tool and the library can handle a set of core types - for example:
  - `&[T]` can be passed across the FFI boundary, even though it doesn't guarantee any particular ABI or memory layout. With manual bindings `std::span<T>` / `&[T]` have to be manually destructured and rebuilt out of a pointer and length - this is error-prone given that each language represents empty slices slightly differently)

  - Smart pointers like `std::unique_ptr<T>`, `std::shared_ptr<T>`, and/or `Box` are natively supported. With manual bindings, one would have to pass C-ABI-compatible raw pointers, which would increase lifetime and memory-safety risks.
  - `rust::String` and `CxxString` types understand and maintain differences in string representation across the languages (e.g. `rust::String::lossy` can build a Rust string from non-UTF8 input and `rust::String::c_str` can NUL-terminate a string).

## 45.1 Bağlamalara Örnek

CXX requires that the whole C++/Rust boundary is declared in `cxx::bridge` modules inside `.rs` source code.

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type MultiBuf;

        fn next_chunk(buf: &mut MultiBuf) -> &[u8];
    }

    unsafe extern "C++" {
        include!("example/include/blobstore.h");

        type BlobstoreClient;

        fn new_blobstore_client() -> UniquePtr<BlobstoreClient>;
        fn put(self: &BlobstoreClient, buf: &mut MultiBuf) -> Result<u64>;
    }
}

// Definitions of Rust types and functions go here
```

Point out:

- Although this looks like a regular Rust `mod`, the `#[cxx::bridge]` procedural macro does complex things to it. The generated code is quite a bit more sophisticated - though this does still result in a `mod` called `ffi` in your code.
- Native support for C++'s `std::unique_ptr` in Rust
- Native support for Rust slices in C++
- Calls from C++ to Rust, and Rust types (in the top part)
- Calls from Rust to C++, and C++ types (in the bottom part)

**Common misconception**: It *looks* like a C++ header is being parsed by Rust, but this is misleading. This header is never interpreted by Rust, but simply `#included` in the generated C++ code for the benefit of C++ compilers.

## 45.2 CXX'in Sınırlamaları

By far the most useful page when using CXX is the type reference.

CXX fundamentally suits cases where:

- Your Rust-C++ interface is sufficiently simple that you can declare all of it.
- You're using only the types natively supported by CXX already, for example `std::unique_ptr`, `std::string`, `&[u8]` etc.

It has many limitations --- for example lack of support for Rust's `Option` type.

These limitations constrain us to using Rust in Chromium only for well isolated "leaf nodes" rather than for arbitrary Rust-C++ interop. When considering a use-case for Rust in Chromium, a good starting point is to draft the CXX bindings for the language boundary to see if it appears simple enough.

In addition, right now, Rust code in one component cannot depend on Rust code in another, due to linking details in our component build. That's another reason to restrict Rust to use in leaf nodes.

You should also discuss some of the other sticky points with CXX, for example:

- Its error handling is based around C++ exceptions (given on the next slide)
- Function pointers are awkward to use.

## 45.3 CXX Hata İşleme

CXX's support for `Result<T,E>` relies on C++ exceptions, so we can't use that in Chromium. Alternatives:

- The `T` part of `Result<T, E>` can be:
  - Returned via out parameters (e.g. via `&mut T`). This requires that `T` can be passed across the FFI boundary - for example `T` has to be:
    - A primitive type (like `u32` or `usize`)
    - A type natively supported by `cxx` (like `UniquePtr<T>`) that has a suitable default value to use in a failure case (*unlike* `Box<T>`).
  - Retained on the Rust side, and exposed via reference. This may be needed when `T` is a Rust type, which cannot be passed across the FFI boundary, and cannot be stored in `UniquePtr<T>`.
- The `E` part of `Result<T, E>` can be:
  - Returned as a boolean (e.g. `true` representing success, and `false` representing failure)
  - Preserving error details is in theory possible, but so far hasn't been needed in practice.

### 45.3.1 CXX Error Handling: QR Example

The QR code generator is an example where a boolean is used to communicate success vs failure, and where the successful result can be passed across the FFI boundary:

```
#[cxx::bridge(namespace = "qr_code_generator")]
mod ffi {
    extern "Rust" {
        fn generate_qr_code_using_rust(
            data: &[u8],
```

```
            min_version: i16,
            out_pixels: Pin<&mut CxxVector<u8>>,
            out_qr_size: &mut usize,
        ) -> bool;
    }
}
```

Students may be curious about the semantics of the `out_qr_size` output. This is not the size of the vector, but the size of the QR code (and admittedly it is a bit redundant - this is the square root of the size of the vector).

It may be worth pointing out the importance of initializing `out_qr_size` before calling into the Rust function. Creation of a Rust reference that points to uninitialized memory results in Undefined Behavior (unlike in C++, when only the act of dereferencing such memory results in UB).

If students ask about `Pin`, then explain why CXX needs it for mutable references to C++ data: the answer is that C++ data can't be moved around like Rust data, because it may contain self-referential pointers.

### 45.3.2 CXX Error Handling: PNG Example

A prototype of a PNG decoder illustrates what can be done when the successful result cannot be passed across the FFI boundary:

```
#[cxx::bridge(namespace = "gfx::rust_bindings")]
mod ffi {
    extern "Rust" {
        /// This returns an FFI-friendly equivalent of `Result<PngReader<'a>,
        /// ()>`.
        fn new_png_reader<'a>(input: &'a [u8]) -> Box<ResultOfPngReader<'a>>;

        /// C++ bindings for the `crate::png::ResultOfPngReader` type.
        type ResultOfPngReader<'a>;
        fn is_err(self: &ResultOfPngReader) -> bool;
        fn unwrap_as_mut<'a, 'b>(
            self: &'b mut ResultOfPngReader<'a>,
        ) -> &'b mut PngReader<'a>;

        /// C++ bindings for the `crate::png::PngReader` type.
        type PngReader<'a>;
        fn height(self: &PngReader) -> u32;
        fn width(self: &PngReader) -> u32;
        fn read_rgba8(self: &mut PngReader, output: &mut [u8]) -> bool;
    }
}
```

`PngReader` and `ResultOfPngReader` are Rust types --- objects of these types cannot cross the FFI boundary without indirection of a `Box<T>`. We can't have an `out_parameter: &mut PngReader`, because CXX doesn't allow C++ to store Rust objects by value.

This example illustrates that even though CXX doesn't support arbitrary generics nor templates, we can still pass them across the FFI boundary by manually specializing / monomorphizing them into a non-generic type. In the example `ResultOfPngReader` is

a non-generic type that forwards into appropriate methods of `Result<T, E>` (e.g. into `is_err`, `unwrap`, and/or `as_mut`).

## 45.4 Using cxx in Chromium

In Chromium, we define an independent `#[cxx::bridge] mod` for each leaf-node where we want to use Rust. You'd typically have one for each `rust_static_library`. Just add

```
cxx_bindings = [ "my_rust_file.rs" ]
   # list of files containing #[cxx::bridge], not all source files
allow_unsafe = true
```

to your existing `rust_static_library` target alongside `crate_root` and `sources`.

C++ headers will be generated at a sensible location, so you can just

```
#include "ui/base/my_rust_file.rs.h"
```

You will find some utility functions in `//base` to convert to/from Chromium C++ types to CXX Rust types --- for example `SpanToRustSlice`.

Students may ask --- why do we still need `allow_unsafe = true`?

The broad answer is that no C/C++ code is "safe" by the normal Rust standards. Calling back and forth to C/C++ from Rust may do arbitrary things to memory, and compromise the safety of Rust's own data layouts. Presence of *too many* `unsafe` keywords in C/C++ interop can harm the signal-to-noise ratio of such a keyword, and is controversial, but strictly, bringing any foreign code into a Rust binary can cause unexpected behavior from Rust's perspective.

The narrow answer lies in the diagram at the top of this page --- behind the scenes, CXX generates Rust `unsafe` and `extern "C"` functions just like we did manually in the previous section.

## 45.5 Alıştırma: C++ ile Birlikte Çalışabilirlik

### Part one

- In the Rust file you previously created, add a `#[cxx::bridge]` which specifies a single function, to be called from C++, called `hello_from_rust`, taking no parameters and returning no value.
- Modify your previous `hello_from_rust` function to remove `extern "C"` and `#[unsafe(no_mangle)]`. This is now just a standard Rust function.
- Modify your gn target to build these bindings.
- In your C++ code, remove the forward-declaration of `hello_from_rust`. Instead, include the generated header file.
- İnşa et (Build) ve çalıştır!

### Part two

It's a good idea to play with CXX a little. It helps you think about how flexible Rust in Chromium actually is.

Some things to try:

- Call back into C++ from Rust. You will need:
  - An additional header file which you can `include!` from your `cxx::bridge`. You'll need to declare your C++ function in that new header file.
  - An `unsafe` block to call such a function, or alternatively specify the `unsafe` keyword in your `#[cxx::bridge]` as described here.
  - You may also need to `#include "third_party/rust/cxx/v1/crate/include/cxx.h"`
- Pass a C++ string from C++ into Rust.
- Pass a reference to a C++ object into Rust.
- Intentionally get the Rust function signatures mismatched from the `#[cxx::bridge]`, and get used to the errors you see.
- Intentionally get the C++ function signatures mismatched from the `#[cxx::bridge]`, and get used to the errors you see.
- Pass a `std::unique_ptr` of some type from C++ into Rust, so that Rust can own some C++ object.
- Create a Rust object and pass it into C++, so that C++ owns it. (Hint: you need a `Box`).
- Declare some methods on a C++ type. Call them from Rust.
- Declare some methods on a Rust type. Call them from C++.

## Part three

Now you understand the strengths and limitations of CXX interop, think of a couple of use-cases for Rust in Chromium where the interface would be sufficiently simple. Sketch how you might define that interface.

## Where to find help

- The `cxx` binding reference
- The `rust_static_library` gn template

As students explore Part Two, they're bound to have lots of questions about how to achieve these things, and also how CXX works behind the scenes.

Some of the questions you may encounter:

- I'm seeing a problem initializing a variable of type X with type Y, where X and Y are both function types. This is because your C++ function doesn't quite match the declaration in your `cxx::bridge`.
- I seem to be able to freely convert C++ references into Rust references. Doesn't that risk UB? For CXX's *opaque* types, no, because they are zero-sized. For CXX trivial types yes, it's *possible* to cause UB, although CXX's design makes it quite difficult to craft such an example.

# Chapter 46

# Üçüncü Taraf Kasalarnı Eklenmesi

Rust libraries are called ”crates” and are found at crates.io. It's *very easy* for Rust crates to depend upon one another. So they do!

| Property | C++ kütüphanesi | Rust crate |
|---|---|---|
| Build system | Lots | Consistent: `Cargo.toml` |
| Typical library size | Large-ish | Small |
| Transitive dependencies | Few | Lots |

For a Chromium engineer, this has pros and cons:

- All crates use a common build system so we can automate their inclusion into Chromium...
- ... but, crates typically have transitive dependencies, so you will likely have to bring in multiple libraries.

We'll discuss:

- How to put a crate in the Chromium source code tree
- How to make gn build rules for it
- How to audit its source code for sufficient safety.

All of the things in the table on this slide are generalizations, and counter-examples can be found. But in general it's important for students to understand that most Rust code depends on other Rust libraries, because it's easy to do so, and that this has both benefits and costs.

## 46.1 Configuring the `Cargo.toml` file to add crates

Chromium has a single set of centrally-managed direct crate dependencies. These are managed through a single `Cargo.toml`:

```
[dependencies]
bitflags = "1"
cfg-if = "1"
cxx = "1"
# lots more...
```

As with any other `Cargo.toml`, you can specify more details about the dependencies --- most commonly, you'll want to specify the `features` that you wish to enable in the crate.

When adding a crate to Chromium, you'll often need to provide some extra information in an additional file, `gnrt_config.toml`, which we'll meet next.

## 46.2 `gnrt_config.toml`'yi Yapılandırma

Alongside `Cargo.toml` is `gnrt_config.toml`. This contains Chromium-specific extensions to crate handling.

If you add a new crate, you should specify at least the `group`. This is one of:

```
#   'safe': The library satisfies the rule-of-2 and can be used in any process.
#   'sandbox': The library does not satisfy the rule-of-2 and must be used in
#              a sandboxed process such as the renderer or a utility process.
#   'test': The library is only used in tests.
```

For instance,

```
[crate.my-new-crate]
group = 'test' # only used in test code
```

Depending on the crate source code layout, you may also need to use this file to specify where its `LICENSE` file(s) can be found.

Later, we'll see some other things you will need to configure in this file to resolve problems.

## 46.3 Kasaların İndirilmesi

A tool called `gnrt` knows how to download crates and how to generate `BUILD.gn` rules.

To start, download the crate you want like this:

```
cd chromium/src
vpython3 tools/crates/run_gnrt.py -- vendor
```

> Although the `gnrt` tool is part of the Chromium source code, by running this command you will be downloading and running its dependencies from `crates.io`. See the earlier section discussing this security decision.

This `vendor` command may download:

- Your crate
- Direct and transitive dependencies
- New versions of other crates, as required by `cargo` to resolve the complete set of crates required by Chromium.

Chromium maintains patches for some crates, kept in `//third_party/rust/chromium_crates_io/patches`. These will be reapplied automatically, but if patching fails you may need to take manual action.

## 46.4 gn İnşa Kurallarının Oluşturulması

Once you've downloaded the crate, generate the `BUILD.gn` files like this:

```
vpython3 tools/crates/run_gnrt.py -- gen
```

Now run `git status`. You should find:

- At least one new crate source code in `third_party/rust/chromium_crates_io/vendor`
- At least one new `BUILD.gn` in `third_party/rust/<crate name>/v<major semver version>`
- An appropriate `README.chromium`

The ”major semver version” is a Rust ”semver” version number.

Take a close look, especially at the things generated in `third_party/rust`.

Talk a little about semver --- and specifically the way that in Chromium it's to allow multiple incompatible versions of a crate, which is discouraged but sometimes necessary in the Cargo ecosystem.

## 46.5 Problemlerin Çözülmesi

If your build fails, it may be because of a `build.rs`: programs which do arbitrary things at build time. This is fundamentally at odds with the design of gn and `ninja` which aim for static, deterministic, build rules to maximize parallelism and repeatability of builds.

Some `build.rs` actions are automatically supported; others require action:

| build script effect | Supported by our gn templates | Work required b |
|---|---|---|
| Checking rustc version to configure features on and off | Yes | None |
| Checking platform or CPU to configure features on and off | Yes | None |
| Generating code | Yes | Yes - specify in g |
| Building C/C++ | No | Patch around it |
| Arbitrary other actions | No | Patch around it |

Fortunately, most crates don't contain a build script, and fortunately, most build scripts only do the top two actions.

### 46.5.1 Kod Oluşturan İnşa Betikleri (Build Scripts )

If `ninja` complains about missing files, check the `build.rs` to see if it writes source code files.

If so, modify `gnrt_config.toml` to add `build-script-outputs` to the crate. If this is a transitive dependency, that is, one on which Chromium code should not directly depend, also add `allow-first-party-usage=false`. There are several examples already in that file:

```
[crate.unicode-linebreak]
allow-first-party-usage = false
build-script-outputs = ["tables.rs"]
```

Now rerun `gnrt.py -- gen` to regenerate `BUILD.gn` files to inform ninja that this particular output file is input to subsequent build steps.

### 46.5.2 C++ Kodunu İnşa Eden (Build) veya Keyfi Eylemler Gerçekleştiren İnşa Betikleri

Some crates use the `cc` crate to build and link C/C++ libraries. Other crates parse C/C++ using `bindgen` within their build scripts. These actions can't be supported in a Chromium context --- our gn, ninja and LLVM build system is very specific in expressing relationships between build actions.

So, your options are:

- Avoid these crates
- Apply a patch to the crate.

Patches should be kept in `third_party/rust/chromium_crates_io/patches/<crate>` - see for example the patches against the `cxx` crate - and will be applied automatically by `gnrt` each time it upgrades the crate.

## 46.6 Bir Kasaya Bağımlılık

Once you've added a third-party crate and generated build rules, depending on a crate is simple. Find your `rust_static_library` target, and add a dep on the `:lib` target within your crate.

Specifically,

```
                      +------------+      +----------------------+
"//third_party/rust" | crate name | "/v" | major semver version | ":lib"
                      +------------+      +----------------------+
```

For instance,

```
rust_static_library("my_rust_lib") {
  crate_root = "lib.rs"
  sources = [ "lib.rs" ]
  deps = [ "//third_party/rust/example_rust_crate/v1:lib" ]
}
```

## 46.7 Auditing Third Party Crates

Adding new libraries is subject to Chromium's standard policies, but of course also subject to security review. As you may be bringing in not just a single crate but also transitive dependencies, there may be a lot of code to review. On the other hand, safe Rust code can have limited negative side effects. How should you review it?

Over time Chromium aims to move to a process based around cargo vet.

Meanwhile, for each new crate addition, we are checking for the following:

- Understand why each crate is used. What's the relationship between crates? If the build system for each crate contains a `build.rs` or procedural macros, work out what they're for. Are they compatible with the way Chromium is normally built?
- Check each crate seems to be reasonably well maintained
- Use `cd third-party/rust/chromium_crates_io; cargo audit` to check for known vulnerabilities (first you'll need to `cargo install cargo-audit`, which ironically involves downloading lots of dependencies from the internet2)
- Ensure any `unsafe` code is good enough for the Rule of Two
- Check for any use of `fs` or `net` APIs
- Read all the code at a sufficient level to look for anything out of place that might have been maliciously inserted. (You can't realistically aim for 100% perfection here: there's often just too much code.)

These are just guidelines --- work with reviewers from `security@chromium.org` to work out the right way to become confident of the crate.

## 46.8 Checking Crates into Chromium Source Code

`git status` should reveal:

- Crate code in `//third_party/rust/chromium_crates_io`
- Metadata (`BUILD.gn` and `README.chromium`) in `//third_party/rust/<crate>/<version>`

Please also add an `OWNERS` file in the latter location.

You should land all this, along with your `Cargo.toml` and `gnrt_config.toml` changes, into the Chromium repo.

**Important**: you need to use `git add -f` because otherwise `.gitignore` files may result in some files being skipped.

As you do so, you might find presubmit checks fail because of non-inclusive language. This is because Rust crate data tends to include names of git branches, and many projects still use non-inclusive terminology there. So you may need to run:

```
infra/update_inclusive_language_presubmit_exempt_dirs.sh > infra/inclusive_language_pres
git add -p infra/inclusive_language_presubmit_exempt_dirs.txt # add whatever changes are
```

## 46.9 Kasaları Güncel Tutmak

As the OWNER of any third party Chromium dependency, you are expected to keep it up to date with any security fixes. It is hoped that we will soon automate this for Rust crates, but for now, it's still your responsibility just as it is for any other third party dependency.

## 46.10 Alıştırma

Add uwuify to Chromium, turning off the crate's default features. Assume that the crate will be used in shipping Chromium, but won't be used to handle untrustworthy input.

(In the next exercise we'll use uwuify from Chromium, but feel free to skip ahead and do that now if you like. Or, you could create a new `rust_executable` target which uses `uwuify`).

Students will need to download lots of transitive dependencies.

The total crates needed are:

- `instant`,
- `lock_api`,
- `parking_lot`,
- `parking_lot_core`,
- `redox_syscall`,
- `scopeguard`,
- `smallvec`, and
- `uwuify`.

If students are downloading even more than that, they probably forgot to turn off the default features.

Thanks to Daniel Liu for this crate!

# Chapter 47

# Bringing It Together --- Exercise

In this exercise, you're going to add a whole new Chromium feature, bringing together everything you already learned.

## The Brief from Product Management

A community of pixies has been discovered living in a remote rainforest. It's important that we get Chromium for Pixies delivered to them as soon as possible.

The requirement is to translate all Chromium's UI strings into Pixie language.

There's not time to wait for proper translations, but fortunately pixie language is very close to English, and it turns out there's a Rust crate which does the translation.

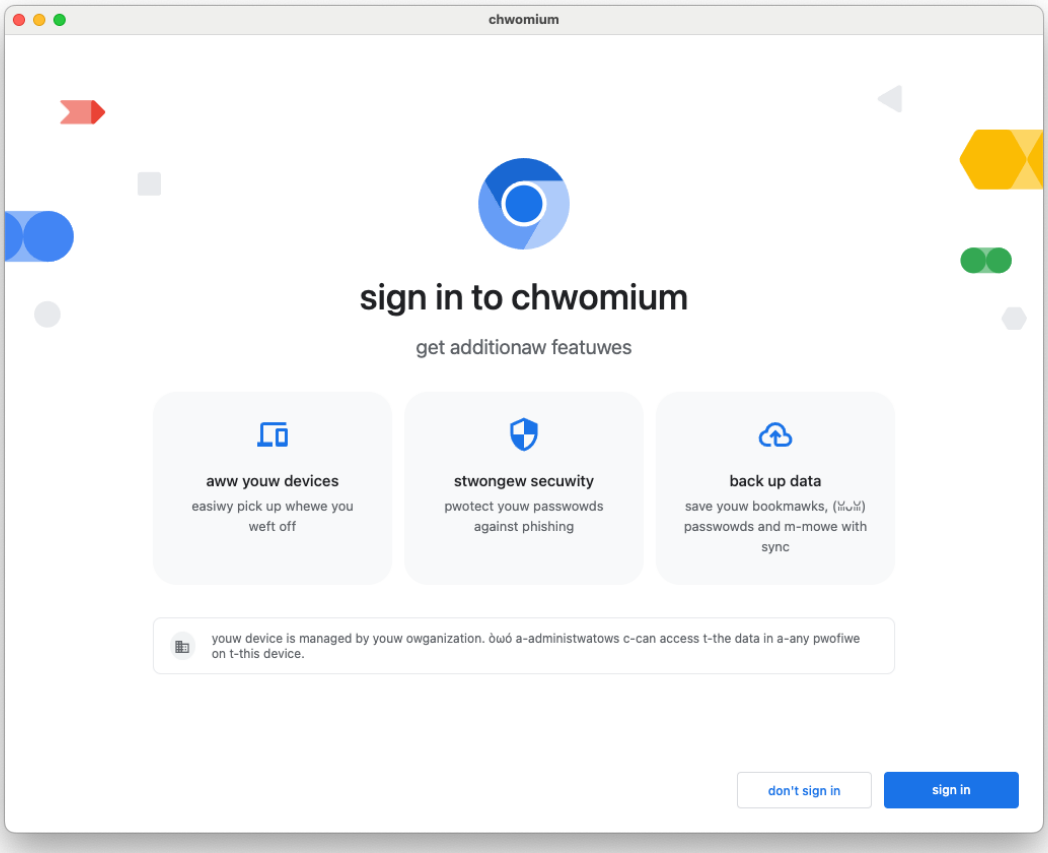
In fact, you already imported that crate in the previous exercise.

(Obviously, real translations of Chrome require incredible care and diligence. Don't ship this!)

## Steps

Modify `ResourceBundle::MaybeMangleLocalizedString` so that it uwuifies all strings before display. In this special build of Chromium, it should always do this irrespective of the setting of `mangle_localized_strings_`.

If you've done everything right across all these exercises, congratulations, you should have created Chrome for pixies!




Students will likely need some hints here. Hints include:

- UTF16 vs UTF8. Students should be aware that Rust strings are always UTF8, and will probably decide that it's better to do the conversion on the C++ side using `base::UTF16ToUTF8` and back again.
- If students decide to do the conversion on the Rust side, they'll need to consider `String::from_utf16`, consider error handling, and consider which CXX supported types can transfer a lot of u16s.
- Students may design the C++/Rust boundary in several different ways, e.g. taking and returning strings by value, or taking a mutable reference to a string. If a mutable reference is used, CXX will likely tell the student that they need to use `Pin`. You may need to explain what `Pin` does, and then explain why CXX needs it for mutable references to C++ data: the answer is that C++ data can't be moved around like Rust data, because it may contain self-referential pointers.
- The C++ target containing `ResourceBundle::MaybeMangleLocalizedString` will need to depend on a `rust_static_library` target. The student probably already did this.
- The `rust_static_library` target will need to depend on `//third_party/rust/uwuify/v0_2:lib`.

# Chapter 48

# Alıştırma Çözümleri

Solutions to the Chromium exercises can be found in this series of CLs.

# Part XI

# Yalın Sistem (Bare Metal): Sabah

# Chapter 49

# Welcome to Bare Metal Rust

This is a standalone one-day course about bare-metal Rust, aimed at people who are familiar with the basics of Rust (perhaps from completing the Comprehensive Rust course), and ideally also have some experience with bare-metal programming in some other language such as C.

Today we will talk about 'bare-metal' Rust: running Rust code without an OS underneath us. This will be divided into several parts:

- What is `no_std` Rust?
- Writing firmware for microcontrollers.
- Writing bootloader / kernel code for application processors.
- Some useful crates for bare-metal Rust development.

For the microcontroller part of the course we will use the BBC micro:bit v2 as an example. It's a development board based on the Nordic nRF52833 microcontroller with some LEDs and buttons, an I2C-connected accelerometer and compass, and an on-board SWD debugger.

To get started, install some tools we'll need later. On gLinux or Debian:

```
sudo apt install gdb-multiarch libudev-dev picocom pkg-config qemu-system-arm build-esse
rustup update
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-rs/releases/late
```

And give users in the `plugdev` group access to the micro:bit programmer:

```
echo 'SUBSYSTEM=="hidraw", ATTRS{idVendor}=="0d28", MODE="0660", GROUP="logindev", TAG+=
  sudo tee /etc/udev/rules.d/50-microbit.rules
sudo udevadm control --reload-rules
```

You should see ”NXP ARM mbed” in the output of `lsusb` if the device is available. If you are using a Linux environment on a Chromebook, you will need to share the USB device with Linux, via `chrome://os-settings/crostini/sharedUsbDevices`.

On MacOS:

```
xcode-select --install
brew install gdb picocom qemu
rustup update
```

```
rustup target add aarch64-unknown-none thumbv7em-none-eabihf
rustup component add llvm-tools-preview
cargo install cargo-binutils
curl --proto '=https' --tlsv1.2 -LsSf https://github.com/probe-rs/probe-rs/releases/late
```

# Chapter 50

# `no_std`

`core`

`alloc`

`std`

- Slices, `&str`, `CStr`
- `NonZeroU8`...
- `Option`, `Result`
- `Display`, `Debug`, `write!`...
- `Iterator`
- `Error`
- `panic!`, `assert_eq!`...
- `NonNull` and all the usual pointer-related functions
- `Future` and `async`/`await`
- `fence`, `AtomicBool`, `AtomicPtr`, `AtomicU32`...
- `Duration`

- `Box`, `Cow`, `Arc`, `Rc`
- `Vec`, `BinaryHeap`, `BtreeMap`, `LinkedList`, `VecDeque`
- `String`, `CString`, `format!`

- `HashMap`
- `Mutex`, `Condvar`, `Barrier`, `Once`, `RwLock`, `mpsc`
- `File` and the rest of `fs`
- `println!`, `Read`, `Write`, `Stdin`, `Stdout` and the rest of `io`
- `Path`, `OsString`
- `net`
- `Command`, `Child`, `ExitCode`
- `spawn`, `sleep` and the rest of `thread`
- `SystemTime`, `Instant`

- `HashMap` depends on RNG.
- `std` re-exports the contents of both `core` and `alloc`.

## 50.1 A minimal `no_std` program

```
#![no_main]
#![no_std]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_panic: &PanicInfo) -> ! {
    loop {}
}
```

- This will compile to an empty binary.
- `std` provides a panic handler; without it we must provide our own.
- It can also be provided by another crate, such as `panic-halt`.
- Depending on the target, you may need to compile with `panic = "abort"` to avoid an error about `eh_personality`.
- Note that there is no `main` or any other entry point; it's up to you to define your own entry point. This will typically involve a linker script and some assembly code to set things up ready for Rust code to run.

## 50.2 `alloc`

To use `alloc` you must implement a global (heap) allocator.

```
#![no_main]
#![no_std]

extern crate alloc;
extern crate panic_halt as _;

use alloc::string::ToString;
use alloc::vec::Vec;
use buddy_system_allocator::LockedHeap;

#[global_allocator]
static HEAP_ALLOCATOR: LockedHeap<32> = LockedHeap::<32>::new();

const HEAP_SIZE: usize = 65536;
static mut HEAP: [u8; HEAP_SIZE] = [0; HEAP_SIZE];

pub fn entry() {
    // SAFETY: `HEAP` is only used here and `entry` is only called once.
    unsafe {
        // Give the allocator some memory to allocate.
        HEAP_ALLOCATOR.lock().init(&raw mut HEAP as usize, HEAP_SIZE);
    }

    // Now we can do things that require heap allocation.
    let mut v = Vec::new();
    v.push("A string".to_string());
```

```
}
```

- `buddy_system_allocator` is a crate implementing a basic buddy system allocator. Other crates are available, or you can write your own or hook into your existing allocator.
- The const parameter of `LockedHeap` is the max order of the allocator; i.e. in this case it can allocate regions of up to 2**32 bytes.
- If any crate in your dependency tree depends on `alloc` then you must have exactly one global allocator defined in your binary. Usually this is done in the top-level binary crate.
- `extern crate panic_halt as _` is necessary to ensure that the `panic_halt` crate is linked in so we get its panic handler.
- This example will build but not run, as it doesn't have an entry point.

# Chapter 51

# Mikrodenetleyiciler

The `cortex_m_rt` crate provides (among other things) a reset handler for Cortex M microcontrollers.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;

use cortex_m_rt::entry;

#[entry]
fn main() -> ! {
    loop {}
}
```

Next we'll look at how to access peripherals, with increasing levels of abstraction.

- The `cortex_m_rt::entry` macro requires that the function have type `fn() -> !`, because returning to the reset handler doesn't make sense.
- Run the example with `cargo embed --bin minimal`

## 51.1 Ham MMIO

Most microcontrollers access peripherals via memory-mapped IO. Let's try turning on an LED on our micro:bit:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

mod interrupts;
```

```
use core::mem::size_of;
use cortex_m_rt::entry;

/// GPIO port 0 peripheral address
const GPIO_P0: usize = 0x5000_0000;

// GPIO peripheral offsets
const PIN_CNF: usize = 0x700;
const OUTSET: usize = 0x508;
const OUTCLR: usize = 0x50c;

// PIN_CNF fields
const DIR_OUTPUT: u32 = 0x1;
const INPUT_DISCONNECT: u32 = 0x1 << 1;
const PULL_DISABLED: u32 = 0x0 << 2;
const DRIVE_S0S1: u32 = 0x0 << 8;
const SENSE_DISABLED: u32 = 0x0 << 16;

#[entry]
fn main() -> ! {
    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let pin_cnf_21 = (GPIO_P0 + PIN_CNF + 21 * size_of::<u32>()) as *mut u32;
    let pin_cnf_28 = (GPIO_P0 + PIN_CNF + 28 * size_of::<u32>()) as *mut u32;
    // SAFETY: The pointers are to valid peripheral control registers, and no
    // aliases exist.
    unsafe {
        pin_cnf_21.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
        pin_cnf_28.write_volatile(
            DIR_OUTPUT
                | INPUT_DISCONNECT
                | PULL_DISABLED
                | DRIVE_S0S1
                | SENSE_DISABLED,
        );
    }

    // Set pin 28 low and pin 21 high to turn the LED on.
    let gpio0_outset = (GPIO_P0 + OUTSET) as *mut u32;
    let gpio0_outclr = (GPIO_P0 + OUTCLR) as *mut u32;
    // SAFETY: The pointers are to valid peripheral control registers, and no
    // aliases exist.
    unsafe {
        gpio0_outclr.write_volatile(1 << 28);
        gpio0_outset.write_volatile(1 << 21);
    }
```

```
    loop {}
}
```

- GPIO 0 pin 21 is connected to the first column of the LED matrix, and pin 28 to the first row.

Run the example with:

```
cargo embed --bin mmio
```

## 51.2 Peripheral Access Crates

svd2rust generates mostly-safe Rust wrappers for memory-mapped peripherals from CMSIS-SVD files.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use nrf52833_pac::Peripherals;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p.P0;

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    gpio0.pin_cnf[21].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });
    gpio0.pin_cnf[28].write(|w| {
        w.dir().output();
        w.input().disconnect();
        w.pull().disabled();
        w.drive().s0s1();
        w.sense().disabled();
        w
    });

    // Set pin 28 low and pin 21 high to turn the LED on.
    gpio0.outclr.write(|w| w.pin28().clear());
    gpio0.outset.write(|w| w.pin21().set());
```

```
    loop {}
}
```

- SVD (System View Description) files are XML files typically provided by silicon vendors which describe the memory map of the device.
  - They are organised by peripheral, register, field and value, with names, descriptions, addresses and so on.
  - SVD files are often buggy and incomplete, so there are various projects which patch the mistakes, add missing details, and publish the generated crates.
- `cortex-m-rt` provides the vector table, among other things.
- If you `cargo install cargo-binutils` then you can run `cargo objdump --bin pac -- -d --no-show-raw-insn` to see the resulting binary.

Run the example with:

```
cargo embed --bin pac
```

## 51.3 HAL crates

HAL crates for many microcontrollers provide wrappers around various peripherals. These generally implement traits from `embedded-hal`.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use nrf52833_hal::gpio::{Level, p0};
use nrf52833_hal::pac::Peripherals;

#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();

    // Create HAL wrapper for GPIO port 0.
    let gpio0 = p0::Parts::new(p.P0);

    // Configure GPIO 0 pins 21 and 28 as push-pull outputs.
    let mut col1 = gpio0.p0_28.into_push_pull_output(Level::High);
    let mut row1 = gpio0.p0_21.into_push_pull_output(Level::Low);

    // Set pin 28 low and pin 21 high to turn the LED on.
    col1.set_low().unwrap();
    row1.set_high().unwrap();

    loop {}
}
```

- `set_low` and `set_high` are methods on the `embedded_hal` `OutputPin` trait.

- HAL crates exist for many Cortex-M and RISC-V devices, including various STM32, GD32, nRF, NXP, MSP430, AVR and PIC microcontrollers.

Run the example with:

```
cargo embed --bin hal
```

## 51.4 Board support crates

Board support crates provide a further level of wrapping for a specific board for convenience.

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use cortex_m_rt::entry;
use embedded_hal::digital::OutputPin;
use microbit::Board;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    board.display_pins.col1.set_low().unwrap();
    board.display_pins.row1.set_high().unwrap();

    loop {}
}
```

- In this case the board support crate is just providing more useful names, and a bit of initialisation.
- The crate may also include drivers for some on-board devices outside of the microcontroller itself.
  - `microbit-v2` includes a simple driver for the LED matrix.

Run the example with:

```
cargo embed --bin board_support
```

## 51.5 The type state pattern

```
#[entry]
fn main() -> ! {
    let p = Peripherals::take().unwrap();
    let gpio0 = p0::Parts::new(p.P0);

    let pin: P0_01<Disconnected> = gpio0.p0_01;

    // let gpio0_01_again = gpio0.p0_01; // Error, moved.
    let mut pin_input: P0_01<Input<Floating>> = pin.into_floating_input();
    if pin_input.is_high().unwrap() {
```

```
        // ...
    }
    let mut pin_output: P0_01<Output<OpenDrain>> = pin_input
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
    pin_output.set_high().unwrap();
    // pin_input.is_high(); // Error, moved.

    let _pin2: P0_02<Output<OpenDrain>> = gpio0
        .p0_02
        .into_open_drain_output(OpenDrainConfig::Disconnect0Standard1, Level::Low);
    let _pin3: P0_03<Output<PushPull>> =
        gpio0.p0_03.into_push_pull_output(Level::Low);

    loop {}
}
```

- Pins don't implement `Copy` or `Clone`, so only one instance of each can exist. Once a pin is moved out of the port struct nobody else can take it.
- Changing the configuration of a pin consumes the old pin instance, so you can't keep use the old instance afterwards.
- The type of a value indicates the state that it is in: e.g. in this case, the configuration state of a GPIO pin. This encodes the state machine into the type system, and ensures that you don't try to use a pin in a certain way without properly configuring it first. Illegal state transitions are caught at compile time.
- You can call `is_high` on an input pin and `set_high` on an output pin, but not vice-versa.
- Many HAL crates follow this pattern.

## 51.6 `embedded-hal`

The `embedded-hal` crate provides a number of traits covering common microcontroller peripherals:

- GPIO
- PWM
- Delay timers
- I2C and SPI buses and devices

Similar traits for byte streams (e.g. UARTs), CAN buses and RNGs are broken out into `embedded-io`, `embedded-can` and `rand_core` respectively.

Other crates then implement drivers in terms of these traits, e.g. an accelerometer driver might need an I2C or SPI device instance.

- The traits cover using the peripherals but not initialising or configuring them, as initialisation and configuration is usually highly platform-specific.
- There are implementations for many microcontrollers, as well as other platforms such as Linux on Raspberry Pi.
- `embedded-hal-async` provides async versions of the traits.
- `embedded-hal-nb` provides another approach to non-blocking I/O, based on the `nb` crate.

## 51.7 `probe-rs` ve `cargo-embed`

probe-rs is a handy toolset for embedded debugging, like OpenOCD but better integrated.

- SWD (Serial Wire Debug) and JTAG via CMSIS-DAP, ST-Link and J-Link probes
- GDB stub and Microsoft DAP (Debug Adapter Protocol) server
- Cargo integration

`cargo-embed` is a cargo subcommand to build and flash binaries, log RTT (Real Time Transfers) output and connect GDB. It's configured by an `Embed.toml` file in your project directory.

- CMSIS-DAP is an Arm standard protocol over USB for an in-circuit debugger to access the CoreSight Debug Access Port of various Arm Cortex processors. It's what the on-board debugger on the BBC micro:bit uses.
- ST-Link is a range of in-circuit debuggers from ST Microelectronics, J-Link is a range from SEGGER.
- The Debug Access Port is usually either a 5-pin JTAG interface or 2-pin Serial Wire Debug.
- probe-rs is a library which you can integrate into your own tools if you want to.
- The Microsoft Debug Adapter Protocol lets VSCode and other IDEs debug code running on any supported microcontroller.
- cargo-embed is a binary built using the probe-rs library.
- RTT (Real Time Transfers) is a mechanism to transfer data between the debug host and the target through a number of ringbuffers.

### 51.7.1 Hata Ayıklama

*Embed.toml*:

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true
```

In one terminal under `src/bare-metal/microcontrollers/examples/`:

```
cargo embed --bin board_support debug
```

In another terminal in the same directory:

On gLinux or Debian:

```
gdb-multiarch target/thumbv7em-none-eabihf/debug/board_support --eval-command="target re
```

On MacOS:

```
arm-none-eabi-gdb target/thumbv7em-none-eabihf/debug/board_support --eval-command="targe
```

In GDB, try running:

```
b src/bin/board_support.rs:29
b src/bin/board_support.rs:30
b src/bin/board_support.rs:32
c
c
c
```

## 51.8 Other projects

- RTIC
  - ”Real-Time Interrupt-driven Concurrency”.
  - Shared resource management, message passing, task scheduling, timer queue.
- Embassy
  - `async` executors with priorities, timers, networking, USB.
- TockOS
  - Security-focused RTOS with preemptive scheduling and Memory Protection Unit support.
- Hubris
  - Microkernel RTOS from Oxide Computer Company with memory protection, unprivileged drivers, IPC.
- Bindings for FreeRTOS.

Some platforms have `std` implementations, e.g. esp-idf.

- RTIC can be considered either an RTOS or a concurrency framework.
  - It doesn't include any HALs.
  - It uses the Cortex-M NVIC (Nested Virtual Interrupt Controller) for scheduling rather than a proper kernel.
  - Cortex-M only.
- Google uses TockOS on the Haven microcontroller for Titan security keys.
- FreeRTOS is mostly written in C, but there are Rust bindings for writing applications.

# Chapter 52

# Alıştırmalar

We will read the direction from an I2C compass, and log the readings to a serial port.

After looking at the exercises, you can look at the solutions provided.

## 52.1 Pusula

We will read the direction from an I2C compass, and log the readings to a serial port. If you have time, try displaying it on the LEDs somehow too, or use the buttons somehow.

Hints:

- Check the documentation for the `lsm303agr` and `microbit-v2` crates, as well as the micro:bit hardware.
- The LSM303AGR Inertial Measurement Unit is connected to the internal I2C bus.
- TWI is another name for I2C, so the I2C master peripheral is called TWIM.
- The LSM303AGR driver needs something implementing the `embedded_hal::i2c::I2c` trait. The `microbit::hal::Twim` struct implements this.
- You have a `microbit::Board` struct with fields for the various pins and peripherals.
- You can also look at the nRF52833 datasheet if you want, but it shouldn't be necessary for this exercise.

Download the exercise template and look in the `compass` directory for the following files.

*src/main.rs*:

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use microbit::{hal::{Delay, uarte::{Baudrate, Parity, Uarte}}, Board};

#[entry]
fn main() -> ! {
```

```
    let mut board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
    );

    // Use the system timer as a delay provider.
    let mut delay = Delay::new(board.SYST);

    // Set up the I2C controller and Inertial Measurement Unit.
    // TODO

    writeln!(serial, "Ready.").unwrap();

    loop {
        // Read compass data and log it to the serial port.
        // TODO
    }
}
```

*Cargo.toml* (you shouldn't need to change this):

```
[workspace]

[package]
name = "compass"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
cortex-m-rt = "0.7.5"
embedded-hal = "1.0.0"
lsm303agr = "1.1.0"
microbit-v2 = "0.15.1"
panic-halt = "1.0.0"
```

*Embed.toml* (you shouldn't need to change this):

```
[default.general]
chip = "nrf52833_xxAA"

[debug.gdb]
enabled = true

[debug.reset]
halt_afterwards = true
```

*.cargo/config.toml* (you shouldn't need to change this):

```
[build]
target = "thumbv7em-none-eabihf" # Cortex-M4F

[target.'cfg(all(target_arch = "arm", target_os = "none"))']
rustflags = ["-C", "link-arg=-Tlink.x"]
```

See the serial output on Linux with:

```
picocom --baud 115200 --imap lfcrlf /dev/ttyACM0
```

Or on Mac OS something like (the device name may be slightly different):

```
picocom --baud 115200 --imap lfcrlf /dev/tty.usbmodem14502
```

Use Ctrl+A Ctrl+Q to quit picocom.

## 52.2 Bare Metal Rust Morning Exercise

### Pusula

(back to exercise)

```
#![no_main]
#![no_std]

extern crate panic_halt as _;

use core::fmt::Write;
use cortex_m_rt::entry;
use embedded_hal::digital::InputPin;
use lsm303agr::{
    AccelMode, AccelOutputDataRate, Lsm303agr, MagMode, MagOutputDataRate,
};
use microbit::Board;
use microbit::display::blocking::Display;
use microbit::hal::twim::Twim;
use microbit::hal::uarte::{Baudrate, Parity, Uarte};
use microbit::hal::{Delay, Timer};
use microbit::pac::twim0::frequency::FREQUENCY_A;

const COMPASS_SCALE: i32 = 30000;
const ACCELEROMETER_SCALE: i32 = 700;

#[entry]
fn main() -> ! {
    let mut board = Board::take().unwrap();

    // Configure serial port.
    let mut serial = Uarte::new(
        board.UARTE0,
        board.uart.into(),
        Parity::EXCLUDED,
        Baudrate::BAUD115200,
```

```
);

// Use the system timer as a delay provider.
let mut delay = Delay::new(board.SYST);

// Set up the I2C controller and Inertial Measurement Unit.
writeln!(serial, "Setting up IMU...").unwrap();
let i2c = Twim::new(board.TWIM0, board.i2c_internal.into(), FREQUENCY_A::K100);
let mut imu = Lsm303agr::new_with_i2c(i2c);
imu.init().unwrap();
imu.set_mag_mode_and_odr(
    &mut delay,
    MagMode::HighResolution,
    MagOutputDataRate::Hz50,
)
.unwrap();
imu.set_accel_mode_and_odr(
    &mut delay,
    AccelMode::Normal,
    AccelOutputDataRate::Hz50,
)
.unwrap();
let mut imu = imu.into_mag_continuous().ok().unwrap();

// Set up display and timer.
let mut timer = Timer::new(board.TIMER0);
let mut display = Display::new(board.display_pins);

let mut mode = Mode::Compass;
let mut button_pressed = false;

writeln!(serial, "Ready.").unwrap();

loop {
    // Read compass data and log it to the serial port.
    while !(imu.mag_status().unwrap().xyz_new_data()
        && imu.accel_status().unwrap().xyz_new_data())
    {}
    let compass_reading = imu.magnetic_field().unwrap();
    let accelerometer_reading = imu.acceleration().unwrap();
    writeln!(
        serial,
        "{},{},{}\t{},{},{}",
        compass_reading.x_nt(),
        compass_reading.y_nt(),
        compass_reading.z_nt(),
        accelerometer_reading.x_mg(),
        accelerometer_reading.y_mg(),
        accelerometer_reading.z_mg(),
    )
    .unwrap();
```

```
        let mut image = [[0; 5]; 5];
        let (x, y) = match mode {
            Mode::Compass => (
                scale(-compass_reading.x_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
                    as usize,
                scale(compass_reading.y_nt(), -COMPASS_SCALE, COMPASS_SCALE, 0, 4)
                    as usize,
            ),
            Mode::Accelerometer => (
                scale(
                    accelerometer_reading.x_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
                scale(
                    -accelerometer_reading.y_mg(),
                    -ACCELEROMETER_SCALE,
                    ACCELEROMETER_SCALE,
                    0,
                    4,
                ) as usize,
            ),
        };
        image[y][x] = 255;
        display.show(&mut timer, image, 100);

        // If button A is pressed, switch to the next mode and briefly blink all LEDs
        // on.
        if board.buttons.button_a.is_low().unwrap() {
            if !button_pressed {
                mode = mode.next();
                display.show(&mut timer, [[255; 5]; 5], 200);
            }
            button_pressed = true;
        } else {
            button_pressed = false;
        }
    }
}

#[derive(Copy, Clone, Debug, Eq, PartialEq)]
enum Mode {
    Compass,
    Accelerometer,
}

impl Mode {
    fn next(self) -> Self {
```

```
        match self {
            Self::Compass => Self::Accelerometer,
            Self::Accelerometer => Self::Compass,
        }
    }
}

fn scale(value: i32, min_in: i32, max_in: i32, min_out: i32, max_out: i32) -> i32 {
    let range_in = max_in - min_in;
    let range_out = max_out - min_out;
    let scaled = min_out + range_out * (value - min_in) / range_in;
    scaled.clamp(min_out, max_out)
}
```

# Part XII

# Yalın Sistem (Bare Metal): Öğleden Sonra

# Chapter 53

# Application processors

So far we've talked about microcontrollers, such as the Arm Cortex-M series. These are typically small systems with very limited resources.

Larger systems with more resources are typically called application processors, built around processors such as the ARM Cortex-A or Intel Atom.

For simplicity we'll just work with QEMU's aarch64 'virt' board.

- Broadly speaking, microcontrollers don't have an MMU or multiple levels of privilege (exception levels on Arm CPUs, rings on x86).
- Application processors have more resources, and often run an operating system, instead of directly executing the target application on startup.
- QEMU supports emulating various different machines or board models for each architecture. The 'virt' board doesn't correspond to any particular real hardware, but is designed purely for virtual machines.
- We will still address this board as bare-metal, as if we were writing an operating system.

## 53.1 Rust'a Hazırlanmak

Before we can start running Rust code, we need to do some initialisation.

```
/**
 * This is a generic entry point for an image. It carries out the
 * operations required to prepare the loaded image to be run.
 * Specifically, it
 *
 * - sets up the MMU with an identity map of virtual to physical
 *   addresses, and enables caching
 * - enables floating point
 * - zeroes the bss section using registers x25 and above
 * - prepares the stack, pointing to a section within the image
 * - sets up the exception vector
 * - branches to the Rust `main` function
 *
 * It preserves x0-x3 for the Rust entry point, as these may contain
```

```
 * boot parameters.
 */
.section .init.entry, "ax"
.global entry
entry:
    /*
     * Load and apply the memory management configuration, ready to
     * enable MMU and caches.
     */
    adrp x30, idmap
    msr ttbr0_el1, x30

    mov_i x30, .Lmairval
    msr mair_el1, x30

    mov_i x30, .Ltcrval
    /* Copy the supported PA range into TCR_EL1.IPS. */
    mrs x29, id_aa64mmfr0_el1
    bfi x30, x29, #32, #4

    msr tcr_el1, x30

    mov_i x30, .Lsctlrval

    /*
     * Ensure everything before this point has completed, then
     * invalidate any potentially stale local TLB entries before they
     * start being used.
     */
    isb
    tlbi vmalle1
    ic iallu
    dsb nsh
    isb

    /*
     * Configure sctlr_el1 to enable MMU and cache and don't proceed
     * until this has completed.
     */
    msr sctlr_el1, x30
    isb

    /* Disable trapping floating point access in EL1. */
    mrs x30, cpacr_el1
    orr x30, x30, #(0x3 << 20)
    msr cpacr_el1, x30
    isb

    /* Zero out the bss section. */
    adr_l x29, bss_begin
    adr_l x30, bss_end
```

```
0:  cmp x29, x30
    b.hs 1f
    stp xzr, xzr, [x29], #16
    b 0b

1:  /* Prepare the stack. */
    adr_l x30, boot_stack_end
    mov sp, x30

    /* Set up exception vector. */
    adr x30, vector_table_el1
    msr vbar_el1, x30

    /* Call into Rust code. */
    bl main

    /* Loop forever waiting for interrupts. */
2:  wfi
    b 2b
```

This code is in `src/bare-metal/aps/examples/src/entry.S`. It's not necessary to understand this in detail -- the takeaway is that typically some low-level setup is needed to meet Rust's expectations of the system.

- This is the same as it would be for C: initialising the processor state, zeroing the BSS, and setting up the stack pointer.
  - The BSS (block starting symbol, for historical reasons) is the part of the object file which containing statically allocated variables which are initialised to zero. They are omitted from the image, to avoid wasting space on zeroes. The compiler assumes that the loader will take care of zeroing them.
- The BSS may already be zeroed, depending on how memory is initialised and the image is loaded, but we zero it to be sure.
- We need to enable the MMU and cache before reading or writing any memory. If we don't:
  - Unaligned accesses will fault. We build the Rust code for the `aarch64-unknown-none` target which sets `+strict-align` to prevent the compiler generating unaligned accesses, so it should be fine in this case, but this is not necessarily the case in general.
  - If it were running in a VM, this can lead to cache coherency issues. The problem is that the VM is accessing memory directly with the cache disabled, while the host has cacheable aliases to the same memory. Even if the host doesn't explicitly access the memory, speculative accesses can lead to cache fills, and then changes from one or the other will get lost when the cache is cleaned or the VM enables the cache. (Cache is keyed by physical address, not VA or IPA.)
- For simplicity, we just use a hardcoded pagetable (see `idmap.S`) which identity maps the first 1 GiB of address space for devices, the next 1 GiB for DRAM, and another 1 GiB higher up for more devices. This matches the memory layout that QEMU uses.
- We also set up the exception vector (`vbar_el1`), which we'll see more about later.
- All examples this afternoon assume we will be running at exception level 1 (EL1). If you need to run at a different exception level you'll need to modify `entry.S` accordingly.

## 53.2 Inline assembly

Sometimes we need to use assembly to do things that aren't possible with Rust code. For example, to make an HVC (hypervisor call) to tell the firmware to power off the system:

```
#![no_main]
#![no_std]

use core::arch::asm;
use core::panic::PanicInfo;

mod asm;
mod exceptions;

const PSCI_SYSTEM_OFF: u32 = 0x84000008;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(_x0: u64, _x1: u64, _x2: u64, _x3: u64) {
    // SAFETY: this only uses the declared registers and doesn't do anything
    // with memory.
    unsafe {
        asm!("hvc #0",
            inout("w0") PSCI_SYSTEM_OFF => _,
            inout("w1") 0 => _,
            inout("w2") 0 => _,
            inout("w3") 0 => _,
            inout("w4") 0 => _,
            inout("w5") 0 => _,
            inout("w6") 0 => _,
            inout("w7") 0 => _,
            options(nomem, nostack)
        );
    }

    loop {}
}
```

(If you actually want to do this, use the smccc crate which has wrappers for all these functions.)

- PSCI is the Arm Power State Coordination Interface, a standard set of functions to manage system and CPU power states, among other things. It is implemented by EL3 firmware and hypervisors on many systems.
- The `0 => _` syntax means initialise the register to 0 before running the inline assembly code, and ignore its contents afterwards. We need to use `inout` rather than `in` because the call could potentially clobber the contents of the registers.
- This `main` function needs to be `#[unsafe(no_mangle)]` and `extern "C"` because it is called from our entry point in `entry.S`.
  - Just `#[no_mangle]` would be sufficient but RFC3325 uses this notation to draw reviewer attention to attributes which might cause undefined behavior if used incorrectly.
- `_x0`–`_x3` are the values of registers `x0`–`x3`, which are conventionally used by the

bootloader to pass things like a pointer to the device tree. According to the standard aarch64 calling convention (which is what `extern "C"` specifies to use), registers `x0`–`x7` are used for the first 8 arguments passed to a function, so `entry.S` doesn't need to do anything special except make sure it doesn't change these registers.
- Run the example in QEMU with `make qemu_psci` under `src/bare-metal/aps/examples`.

## 53.3 Volatile memory access for MMIO

- Use `pointer::read_volatile` and `pointer::write_volatile`.
- Never hold a reference to a location being accessed with these methods. Rust may read from (or write to, for `&mut`) a reference at any time.
- Use `&raw` to get fields of structs without creating an intermediate reference.

```
const SOME_DEVICE_REGISTER: *mut u64 = 0x800_0000 as _;
// SAFETY: Some device is mapped at this address.
unsafe {
    SOME_DEVICE_REGISTER.write_volatile(0xff);
    SOME_DEVICE_REGISTER.write_volatile(0x80);
    assert_eq!(SOME_DEVICE_REGISTER.read_volatile(), 0xaa);
}
```

- Volatile access: read or write operations may have side-effects, so prevent the compiler or hardware from reordering, duplicating or eliding them.
  - Usually if you write and then read, e.g. via a mutable reference, the compiler may assume that the value read is the same as the value just written, and not bother actually reading memory.
- Some existing crates for volatile access to hardware do hold references, but this is unsound. Whenever a reference exist, the compiler may choose to dereference it.
- Use `&raw` to get struct field pointers from a pointer to the struct.
- For compatibility with old versions of Rust you can use the `addr_of!` macro instead.

## 53.4 Let's write a UART driver

The QEMU 'virt' machine has a PL011 UART, so let's write a driver for that.

```
const FLAG_REGISTER_OFFSET: usize = 0x18;
const FR_BUSY: u8 = 1 << 3;
const FR_TXFF: u8 = 1 << 5;

/// Minimal driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    base_address: *mut u8,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device at the
    /// given base address.
    ///
    /// # Safety
```

```
    ///
    /// The given base address must point to the 8 MMIO control registers of a
    /// PL011 device, which must be mapped into the address space of the process
    /// as device memory and not have any other aliases.
    pub unsafe fn new(base_address: *mut u8) -> Self {
        Self { base_address }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register() & FR_TXFF != 0 {}

        // SAFETY: We know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            self.base_address.write_volatile(byte);
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register() & FR_BUSY != 0 {}
    }

    fn read_flag_register(&self) -> u8 {
        // SAFETY: We know that the base address points to the control
        // registers of a PL011 device which is appropriately mapped.
        unsafe { self.base_address.add(FLAG_REGISTER_OFFSET).read_volatile() }
    }
}
```

- Note that `Uart::new` is unsafe while the other methods are safe. This is because as long as the caller of `Uart::new` guarantees that its safety requirements are met (i.e. that there is only ever one instance of the driver for a given UART, and nothing else aliasing its address space), then it is always safe to call `write_byte` later because we can assume the necessary preconditions.
- We could have done it the other way around (making `new` safe but `write_byte` unsafe), but that would be much less convenient to use as every place that calls `write_byte` would need to reason about the safety
- This is a common pattern for writing safe wrappers of unsafe code: moving the burden of proof for soundness from a large number of places to a smaller number of places.

### 53.4.1 More traits

We derived the `Debug` trait. It would be useful to implement a few more traits too.

```
use core::fmt::{self, Write};

impl Write for Uart {
    fn write_str(&mut self, s: &str) -> fmt::Result {
        for c in s.as_bytes() {
            self.write_byte(*c);
```

```
        }
        Ok(())
    }
}

// SAFETY: `Uart` just contains a pointer to device memory, which can be
// accessed from any context.
unsafe impl Send for Uart {}
```

- Implementing `Write` lets us use the `write!` and `writeln!` macros with our `Uart` type.
- `Send` is an auto-trait, but not implemented automatically because it is not implemented for pointers.

### 53.4.2 Using it

Let's write a small program using our driver to write to the serial console.

```
#![no_main]
#![no_std]

mod asm;
mod exceptions;
mod pl011_minimal;

use crate::pl011_minimal::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use log::error;
use smccc::Hvc;
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: *mut u8 = 0x900_0000 as _;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let mut uart = unsafe { Uart::new(PL011_BASE_ADDRESS) };

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    system_off::<Hvc>().unwrap();
}
```

- As in the inline assembly example, this `main` function is called from our entry point code in `entry.S`. See the speaker notes there for details.
- Run the example in QEMU with `make qemu_minimal` under `src/bare-metal/aps/examples`.

## 53.5 A better UART driver

The PL011 actually has a bunch more registers, and adding offsets to construct pointers to access them is error-prone and hard to read. Plus, some of them are bit fields which would be nice to access in a structured way.

| Offset | Register name | Width |
|---|---|---|
| 0x00 | DR | 12 |
| 0x04 | RSR | 4 |
| 0x18 | FR | 9 |
| 0x20 | ILPR | 8 |
| 0x24 | IBRD | 16 |
| 0x28 | FBRD | 6 |
| 0x2c | LCR_H | 8 |
| 0x30 | CR | 16 |
| 0x34 | IFLS | 6 |
| 0x38 | IMSC | 11 |
| 0x3c | RIS | 11 |
| 0x40 | MIS | 11 |
| 0x44 | ICR | 11 |
| 0x48 | DMACR | 3 |

- There are also some ID registers which have been omitted for brevity.

### 53.5.1 Bitflags Kasası

The `bitflags` crate is useful for working with bitflags.

```
use bitflags::bitflags;

bitflags! {
    /// Flags from the UART flag register.
    #[repr(transparent)]
    #[derive(Copy, Clone, Debug, Eq, PartialEq)]
    struct Flags: u16 {
        /// Clear to send.
        const CTS = 1 << 0;
        /// Data set ready.
        const DSR = 1 << 1;
        /// Data carrier detect.
        const DCD = 1 << 2;
        /// UART busy transmitting data.
        const BUSY = 1 << 3;
        /// Receive FIFO is empty.
        const RXFE = 1 << 4;
        /// Transmit FIFO is full.
        const TXFF = 1 << 5;
        /// Receive FIFO is full.
        const RXFF = 1 << 6;
        /// Transmit FIFO is empty.
```

```
        const TXFE = 1 << 7;
        /// Ring indicator.
        const RI = 1 << 8;
    }
}
```

- The `bitflags!` macro creates a newtype something like `struct Flags(u16)`, along with a bunch of method implementations to get and set flags.

### 53.5.2 Multiple registers

We can use a struct to represent the memory layout of the UART's registers.

```
#[repr(C, align(4))]
pub struct Registers {
    dr: u16,
    _reserved0: [u8; 2],
    rsr: ReceiveStatus,
    _reserved1: [u8; 19],
    fr: Flags,
    _reserved2: [u8; 6],
    ilpr: u8,
    _reserved3: [u8; 3],
    ibrd: u16,
    _reserved4: [u8; 2],
    fbrd: u8,
    _reserved5: [u8; 3],
    lcr_h: u8,
    _reserved6: [u8; 3],
    cr: u16,
    _reserved7: [u8; 3],
    ifls: u8,
    _reserved8: [u8; 3],
    imsc: u16,
    _reserved9: [u8; 2],
    ris: u16,
    _reserved10: [u8; 2],
    mis: u16,
    _reserved11: [u8; 2],
    icr: u16,
    _reserved12: [u8; 2],
    dmacr: u8,
    _reserved13: [u8; 3],
}
```

- `#[repr(C)]` tells the compiler to lay the struct fields out in order, following the same rules as C. This is necessary for our struct to have a predictable layout, as default Rust representation allows the compiler to (among other things) reorder fields however it sees fit.

### 53.5.3 Sürücü

Now let's use the new `Registers` struct in our driver.

```
/// Driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart {
    registers: *mut Registers,
}

impl Uart {
    /// Constructs a new instance of the UART driver for a PL011 device with the
    /// given set of registers.
    ///
    /// # Safety
    ///
    /// The given pointer must point to the 8 MMIO control registers of a PL011
    /// device, which must be mapped into the address space of the process as
    /// device memory and not have any other aliases.
    pub unsafe fn new(registers: *mut Registers) -> Self {
        Self { registers }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&mut self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // SAFETY: We know that self.registers points to the control registers
        // of a PL011 device which is appropriately mapped.
        unsafe {
            // Write to the TX buffer.
            (&raw mut (*self.registers).dr).write_volatile(byte.into());
        }

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
    /// received.
    pub fn read_byte(&mut self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            // SAFETY: We know that self.registers points to the control
            // registers of a PL011 device which is appropriately mapped.
            let data = unsafe { (&raw const (*self.registers).dr).read_volatile() };
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
    }
```

```
    fn read_flag_register(&self) -> Flags {
        // SAFETY: We know that self.registers points to the control registers
        // of a PL011 device which is appropriately mapped.
        unsafe { (&raw const (*self.registers).fr).read_volatile() }
    }
}
```

- Note the use of `&raw const` / `&raw mut` to get pointers to individual fields without creating an intermediate reference, which would be unsound.
- The example isn't included in the slides because it is very similar to the `safe-mmio` example which comes next. You can run it in QEMU with `make qemu` under `src/bare-metal/aps/examples` if you need to.

## 53.6 safe-mmio

The `safe-mmio` crate provides types to wrap registers which can be read or written safely.

| Can't read | Read has no side-effects | Read has side-effects |
|---|---|---|
| Can't write | `ReadPure` | `ReadOnly` |
| Can write | `WriteOnly` | `ReadPureWrite` | `ReadWrite` |

```
use safe_mmio::fields::{ReadPure, ReadPureWrite, ReadWrite, WriteOnly};

#[repr(C, align(4))]
pub struct Registers {
    dr: ReadWrite<u16>,
    _reserved0: [u8; 2],
    rsr: ReadPure<ReceiveStatus>,
    _reserved1: [u8; 19],
    fr: ReadPure<Flags>,
    _reserved2: [u8; 6],
    ilpr: ReadPureWrite<u8>,
    _reserved3: [u8; 3],
    ibrd: ReadPureWrite<u16>,
    _reserved4: [u8; 2],
    fbrd: ReadPureWrite<u8>,
    _reserved5: [u8; 3],
    lcr_h: ReadPureWrite<u8>,
    _reserved6: [u8; 3],
    cr: ReadPureWrite<u16>,
    _reserved7: [u8; 3],
    ifls: ReadPureWrite<u8>,
    _reserved8: [u8; 3],
    imsc: ReadPureWrite<u16>,
    _reserved9: [u8; 2],
```

```
    ris: ReadPure<u16>,
    _reserved10: [u8; 2],
    mis: ReadPure<u16>,
    _reserved11: [u8; 2],
    icr: WriteOnly<u16>,
    _reserved12: [u8; 2],
    dmacr: ReadPureWrite<u8>,
    _reserved13: [u8; 3],
}
```

- Reading `dr` has a side effect: it pops a byte from the receive FIFO.
- Reading `rsr` (and other registers) has no side-effects. It is a 'pure' read.
- There are a number of different crates providing safe abstractions around MMIO operations; we recommend the `safe-mmio` crate.
- The difference between `ReadPure` or `ReadOnly` (and likewise between `ReadPureWrite` and `ReadWrite`) is whether reading a register can have side-effects which change the state of the device. E.g. reading the data register pops a byte from the receive FIFO. `ReadPure` means that reads have no side-effects, they are purely reading data.

### 53.6.1 Sürücü

Now let's use the new `Registers` struct in our driver.

```
use safe_mmio::{UniqueMmioPointer, field, field_shared};

/// Driver for a PL011 UART.
#[derive(Debug)]
pub struct Uart<'a> {
    registers: UniqueMmioPointer<'a, Registers>,
}

impl<'a> Uart<'a> {
    /// Constructs a new instance of the UART driver for a PL011 device with the
    /// given set of registers.
    pub fn new(registers: UniqueMmioPointer<'a, Registers>) -> Self {
        Self { registers }
    }

    /// Writes a single byte to the UART.
    pub fn write_byte(&mut self, byte: u8) {
        // Wait until there is room in the TX buffer.
        while self.read_flag_register().contains(Flags::TXFF) {}

        // Write to the TX buffer.
        field!(self.registers, dr).write(byte.into());

        // Wait until the UART is no longer busy.
        while self.read_flag_register().contains(Flags::BUSY) {}
    }

    /// Reads and returns a pending byte, or `None` if nothing has been
```

```
    /// received.
    pub fn read_byte(&mut self) -> Option<u8> {
        if self.read_flag_register().contains(Flags::RXFE) {
            None
        } else {
            let data = field!(self.registers, dr).read();
            // TODO: Check for error conditions in bits 8-11.
            Some(data as u8)
        }
    }

    fn read_flag_register(&self) -> Flags {
        field_shared!(self.registers, fr).read()
    }
}
```

- The driver no longer needs any unsafe code!
- `UniqueMmioPointer` is a wrapper around a raw pointer to an MMIO device or register. The caller of `UniqueMmioPointer::new` promises that it is valid and unique for the given lifetime, so it can provide safe methods to read and write fields.
- Note that `Uart::new` is now safe; `UniqueMmioPointer::new` is unsafe instead.
- These MMIO accesses are generally a wrapper around `read_volatile` and `write_volatile`, though on aarch64 they are instead implemented in assembly to work around a bug where the compiler can emit instructions that prevent MMIO virtualisation.
- The `field!` and `field_shared!` macros internally use `&raw mut` and `&raw const` to get pointers to individual fields without creating an intermediate reference, which would be unsound.
- `field!` needs a mutable reference to a `UniqueMmioPointer`, and returns a `UniqueMmioPointer` which allows reads with side effects and writes.
- `field_shared!` works with a shared reference to either a `UniqueMmioPointer` or a `SharedMmioPointer`. It returns a `SharedMmioPointer` which only allows pure reads.

### 53.6.2 Onu Kullan

Let's write a small program using our driver to write to the serial console, and echo incoming bytes.

```
#![no_main]
#![no_std]

mod asm;
mod exceptions;
mod pl011;

use crate::pl011::Uart;
use core::fmt::Write;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::error;
use safe_mmio::UniqueMmioPointer;
use smccc::Hvc;
```

```
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<pl011::Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let mut uart = Uart::new(unsafe { UniqueMmioPointer::new(PL011_BASE_ADDRESS) });

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    loop {
        if let Some(byte) = uart.read_byte() {
            uart.write_byte(byte);
            match byte {
                b'\r' => uart.write_byte(b'\n'),
                b'q' => break,
                _ => continue,
            }
        }
    }

    writeln!(uart, "\n\nBye!").unwrap();
    system_off::<Hvc>().unwrap();
}
```

- Run the example in QEMU with `make qemu_safemmio` under `src/bare-metal/aps/examples`.

## 53.7 Kayıt Tutma (Logging)

It would be nice to be able to use the logging macros from the `log` crate. We can do this by implementing the `Log` trait.

```
use crate::pl011::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart<'static>>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
```

```
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(
    uart: Uart<'static>,
    max_level: LevelFilter,
) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

- The first unwrap in `log` will succeed because we initialise `LOGGER` before calling `set_logger`. The second will succeed because `Uart::write_str` always returns `Ok`.

### 53.7.1 Using it

We need to initialise the logger before we use it.

```
#![no_main]
#![no_std]

mod asm;
mod exceptions;
mod logger;
mod pl011;

use crate::pl011::Uart;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info};
use safe_mmio::UniqueMmioPointer;
use smccc::Hvc;
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<pl011::Registers> =
```

```
    NonNull::new(0x900_0000 as _).unwrap();

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn main(x0: u64, x1: u64, x2: u64, x3: u64) {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})");

    assert_eq!(x1, 42);

    system_off::<Hvc>().unwrap();
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

- Note that our panic handler can now log details of panics.
- Run the example in QEMU with `make qemu_logger` under `src/bare-metal/aps/examples`.

## 53.8 İstisnalar

AArch64 defines an exception vector table with 16 entries, for 4 types of exceptions (synchronous, IRQ, FIQ, SError) from 4 states (current EL with SP0, current EL with SPx, lower EL using AArch64, lower EL using AArch32). We implement this in assembly to save volatile registers to the stack before calling into Rust code:

```
use log::error;
use smccc::Hvc;
use smccc::psci::system_off;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    error!("irq_current");
    system_off::<Hvc>().unwrap();
}
```

```
// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

- EL is exception level; all our examples this afternoon run in EL1.
- For simplicity we aren't distinguishing between SP0 and SPx for the current EL exceptions, or between AArch32 and AArch64 for the lower EL exceptions.
- For this example we just log the exception and power down, as we don't expect any of them to actually happen.
- We can think of exception handlers and our main execution context more or less like different threads. Send and Sync will control what we can share between them, just like with threads. For example, if we want to share some value between exception handlers

and the rest of the program, and it's Send but not Sync, then we'll need to wrap it in something like a Mutex and put it in a static.

## 53.9 aarch64-rt

The aarch64-rt crate provides the assembly entry point and exception vector that we implemented before. We just need to mark our main function with the entry! macro.

It also provides the initial_pagetable! macro to let us define an initial static pagetable in Rust, rather than in assembly code like we did before.

We can also use the UART driver from the arm-pl011-uart crate rather than writing our own.

```
#![no_main]
#![no_std]

mod exceptions;

use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::fmt::Write;
use core::panic::PanicInfo;
use core::ptr::NonNull;
use smccc::Hvc;
use smccc::psci::system_off;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
```

```
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
});

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let mut uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };

    writeln!(uart, "main({x0:#x}, {x1:#x}, {x2:#x}, {x3:#x})").unwrap();

    system_off::<Hvc>().unwrap();
    panic!("system_off returned");
}

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    system_off::<Hvc>().unwrap();
    loop {}
}
```

- Run the example in QEMU with `make qemu_rt` under `src/bare-metal/aps/examples`.

## 53.10 Other projects

- oreboot
  - ”coreboot without the C”.
  - Supports x86, aarch64 and RISC-V.
  - Relies on LinuxBoot rather than having many drivers itself.
- Rust RaspberryPi OS tutorial
  - Initialisation, UART driver, simple bootloader, JTAG, exception levels, exception handling, page tables.
  - Some dodginess around cache maintenance and initialisation in Rust, not necessarily a good example to copy for production code.
- `cargo-call-stack`
  - Static analysis to determine maximum stack usage.
- The RaspberryPi OS tutorial runs Rust code before the MMU and caches are enabled. This will read and write memory (e.g. the stack). However, this has the problems mentioned at the beginning of this session regarding unaligned access and cache coherency.

# Chapter 54

# Kullanışlı kasalar (crates)

We'll look at a few crates which solve some common problems in bare-metal programming.

## 54.1 `zerocopy`

The zerocopy crate (from Fuchsia) provides traits and macros for safely converting between byte sequences and other types.

```
use zerocopy::{Immutable, IntoBytes};

#[repr(u32)]
#[derive(Debug, Default, Immutable, IntoBytes)]
enum RequestType {
    #[default]
    In = 0,
    Out = 1,
    Flush = 4,
}

#[repr(C)]
#[derive(Debug, Default, Immutable, IntoBytes)]
struct VirtioBlockRequest {
    request_type: RequestType,
    reserved: u32,
    sector: u64,
}

fn main() {
    let request = VirtioBlockRequest {
        request_type: RequestType::Flush,
        sector: 42,
        ..Default::default()
    };

    assert_eq!(
```

```
        request.as_bytes(),
        &[4, 0, 0, 0, 0, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0, 0]
    );
}
```

This is not suitable for MMIO (as it doesn't use volatile reads and writes), but can be useful for working with structures shared with hardware e.g. by DMA, or sent over some external interface.

- `FromBytes` can be implemented for types for which any byte pattern is valid, and so can safely be converted from an untrusted sequence of bytes.
- Attempting to derive `FromBytes` for these types would fail, because `RequestType` doesn't use all possible u32 values as discriminants, so not all byte patterns are valid.
- `zerocopy::byteorder` has types for byte-order aware numeric primitives.
- Run the example with `cargo run` under `src/bare-metal/useful-crates/zerocopy-example/`. (It won't run in the Playground because of the crate dependency.)

## 54.2 aarch64-paging

The aarch64-paging crate lets you create page tables according to the AArch64 Virtual Memory System Architecture.

```
use aarch64_paging::{
    idmap::IdMap,
    paging::{Attributes, MemoryRegion},
};

const ASID: usize = 1;
const ROOT_LEVEL: usize = 1;

// Create a new page table with identity mapping.
let mut idmap = IdMap::new(ASID, ROOT_LEVEL);
// Map a 2 MiB region of memory as read-only.
idmap.map_range(
    &MemoryRegion::new(0x80200000, 0x80400000),
    Attributes::NORMAL | Attributes::NON_GLOBAL | Attributes::READ_ONLY,
).unwrap();
// Set `TTBR0_EL1` to activate the page table.
idmap.activate();
```

- This is used in Android for the Protected VM Firmware.
- There's no easy way to run this example by itself, as it needs to run on real hardware or under QEMU.

## 54.3 buddy_system_allocator

buddy_system_allocator is a crate implementing a basic buddy system allocator. It can be used both to implement GlobalAlloc (using LockedHeap) so you can use the standard `alloc` crate (as we saw before), or for allocating other address space (using FrameAllocator). For example, we might want to allocate MMIO space for PCI BARs:

```
use buddy_system_allocator::FrameAllocator;
use core::alloc::Layout;

fn main() {
    let mut allocator = FrameAllocator::<32>::new();
    allocator.add_frame(0x200_0000, 0x400_0000);

    let layout = Layout::from_size_align(0x100, 0x100).unwrap();
    let bar = allocator
        .alloc_aligned(layout)
        .expect("Failed to allocate 0x100 byte MMIO region");
    println!("Allocated 0x100 byte MMIO region at {:#x}", bar);
}
```

- PCI BARs always have alignment equal to their size.
- Run the example with `cargo run` under `src/bare-metal/useful-crates/allocator-example/`. (It won't run in the Playground because of the crate dependency.)

## 54.4 tinyvec

Sometimes you want something which can be resized like a `Vec`, but without heap allocation. tinyvec provides this: a vector backed by an array or slice, which could be statically allocated or on the stack, which keeps track of how many elements are used and panics if you try to use more than are allocated.

```
use tinyvec::{ArrayVec, array_vec};

fn main() {
    let mut numbers: ArrayVec<[u32; 5]> = array_vec!(42, 66);
    println!("{numbers:?}");
    numbers.push(7);
    println!("{numbers:?}");
    numbers.remove(1);
    println!("{numbers:?}");
}
```

- `tinyvec` requires that the element type implement `Default` for initialisation.
- The Rust Playground includes `tinyvec`, so this example will run fine inline.

## 54.5 spin

`std::sync::Mutex` and the other synchronisation primitives from `std::sync` are not available in `core` or `alloc`. How can we manage synchronisation or interior mutability, such as for sharing state between different CPUs?

The spin crate provides spinlock-based equivalents of many of these primitives.

```
use spin::mutex::SpinMutex;

static COUNTER: SpinMutex<u32> = SpinMutex::new(0);
```

```
fn main() {
    dbg!(COUNTER.lock());
    *COUNTER.lock() += 2;
    dbg!(COUNTER.lock());
}
```

- Be careful to avoid deadlock if you take locks in interrupt handlers.
- `spin` also has a ticket lock mutex implementation; equivalents of `RwLock`, `Barrier` and `Once` from `std::sync`; and `Lazy` for lazy initialisation.
- The `once_cell` crate also has some useful types for late initialisation with a slightly different approach to `spin::once::Once`.
- The Rust Playground includes `spin`, so this example will run fine inline.

# Chapter 55

# Android'de Yalın Sistem (Bare Metal)

To build a bare-metal Rust binary in AOSP, you need to use a `rust_ffi_static` Soong rule to build your Rust code, then a `cc_binary` with a linker script to produce the binary itself, and then a `raw_binary` to convert the ELF to a raw binary ready to be run.

```
rust_ffi_static {
    name: "libvmbase_example",
    defaults: ["vmbase_ffi_defaults"],
    crate_name: "vmbase_example",
    srcs: ["src/main.rs"],
    rustlibs: [
        "libvmbase",
    ],
}

cc_binary {
    name: "vmbase_example",
    defaults: ["vmbase_elf_defaults"],
    srcs: [
        "idmap.S",
    ],
    static_libs: [
        "libvmbase_example",
    ],
    linker_scripts: [
        "image.ld",
        ":vmbase_sections",
    ],
}

raw_binary {
    name: "vmbase_example_bin",
    stem: "vmbase_example.bin",
    src: ":vmbase_example",
```

```
    enabled: false,
    target: {
        android_arm64: {
            enabled: true,
        },
    },
}
```

## 55.1 vmbase

For VMs running under crosvm on aarch64, the vmbase library provides a linker script and useful defaults for the build rules, along with an entry point, UART console logging and more.

```
#![no_main]
#![no_std]

use vmbase::{main, println};

main!(main);

pub fn main(arg0: u64, arg1: u64, arg2: u64, arg3: u64) {
    println!("Hello world");
}
```

- The `main!` macro marks your main function, to be called from the `vmbase` entry point.
- The `vmbase` entry point handles console initialisation, and issues a PSCI_SYSTEM_OFF to shutdown the VM if your main function returns.

# Chapter 56

# Alıştırmalar

We will write a driver for the PL031 real-time clock device.

After looking at the exercises, you can look at the solutions provided.

## 56.1 RTC driver

The QEMU aarch64 virt machine has a PL031 real-time clock at 0x9010000. For this exercise, you should write a driver for it.

1. Use it to print the current time to the serial console. You can use the `chrono` crate for date/time formatting.
2. Use the match register and raw interrupt status to busy-wait until a given time, e.g. 3 seconds in the future. (Call `core::hint::spin_loop` inside the loop.)
3. *Extension if you have time:* Enable and handle the interrupt generated by the RTC match. You can use the driver provided in the `arm-gic` crate to configure the Arm Generic Interrupt Controller.
    - Use the RTC interrupt, which is wired to the GIC as `IntId::spi(2)`.
    - Once the interrupt is enabled, you can put the core to sleep via `arm_gic::wfi()`, which will cause the core to sleep until it receives an interrupt.

Download the exercise template and look in the `rtc` directory for the following files.

*src/main.rs*:

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;

use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::GicV3;
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
```

```
use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut Gicd = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut GicrSgi = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
});

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // SAFETY: `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic =
        unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS, 1, false) };
```

```
    gic.setup(0);

    // TODO: Create instance of RTC driver and print current time.

    // TODO: Wait for 3 seconds.

    system_off::<Hvc>().unwrap();
    panic!("system_off returned");
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

*src/exceptions.rs* (you should only need to change this for the 3rd part of the exercise):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use arm_gic::gicv3::{GicV3, InterruptGroup};
use log::{error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_exception_current(_elr: u64, _spsr: u64) {
    error!("sync_exception_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_current(_elr: u64, _spsr: u64) {
    trace!("irq_current");
    let intid = GicV3::get_and_acknowledge_interrupt(InterruptGroup::Group1)
        .expect("No pending interrupt");
    info!("IRQ {intid:?}");
```

```
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_current(_elr: u64, _spsr: u64) {
    error!("fiq_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_current(_elr: u64, _spsr: u64) {
    error!("serr_current");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn sync_lower(_elr: u64, _spsr: u64) {
    error!("sync_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn irq_lower(_elr: u64, _spsr: u64) {
    error!("irq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn fiq_lower(_elr: u64, _spsr: u64) {
    error!("fiq_lower");
    system_off::<Hvc>().unwrap();
}

// SAFETY: There is no other global function of this name.
#[unsafe(no_mangle)]
extern "C" fn serr_lower(_elr: u64, _spsr: u64) {
    error!("serr_lower");
    system_off::<Hvc>().unwrap();
}
```

*src/logger.rs* (you shouldn't need to change this):

```
// Copyright 2023 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
```

```
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

use arm_pl011_uart::Uart;
use core::fmt::Write;
use log::{LevelFilter, Log, Metadata, Record, SetLoggerError};
use spin::mutex::SpinMutex;

static LOGGER: Logger = Logger { uart: SpinMutex::new(None) };

struct Logger {
    uart: SpinMutex<Option<Uart<'static>>>,
}

impl Log for Logger {
    fn enabled(&self, _metadata: &Metadata) -> bool {
        true
    }

    fn log(&self, record: &Record) {
        writeln!(
            self.uart.lock().as_mut().unwrap(),
            "[{}] {}",
            record.level(),
            record.args()
        )
        .unwrap();
    }

    fn flush(&self) {}
}

/// Initialises UART logger.
pub fn init(
    uart: Uart<'static>,
    max_level: LevelFilter,
) -> Result<(), SetLoggerError> {
    LOGGER.uart.lock().replace(uart);

    log::set_logger(&LOGGER)?;
    log::set_max_level(max_level);
    Ok(())
}
```

*Cargo.toml* (you shouldn't need to change this):

```
[workspace]
```

```toml
[package]
name = "rtc"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
aarch64-paging = { version = "0.10.0", default-features = false }
aarch64-rt = "0.2.2"
arm-gic = "0.6.1"
arm-pl011-uart = "0.3.2"
bitflags = "2.9.3"
chrono = { version = "0.4.41", default-features = false }
log = "0.4.27"
safe-mmio = "0.2.5"
smccc = "0.2.2"
spin = "0.10.0"
zerocopy = "0.8.26"
```

*build.rs* (you shouldn't need to change this):

```
// Copyright 2025 Google LLC
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

fn main() {
    println!("cargo:rustc-link-arg=-Timage.ld");
    println!("cargo:rustc-link-arg=-Tmemory.ld");
    println!("cargo:rerun-if-changed=memory.ld");
}
```

*memory.ld* (you shouldn't need to change this):

```
/*
 * Copyright 2023 Google LLC
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *     https://www.apache.org/licenses/LICENSE-2.0
 *
```

```
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

MEMORY
{
    image : ORIGIN = 0x40080000, LENGTH = 2M
}
```

*Makefile* (you shouldn't need to change this):

```
# Copyright 2023 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

.PHONY: build qemu_minimal qemu qemu_logger

all: rtc.bin

build:
    cargo build

rtc.bin: build
    cargo objcopy -- -O binary $@

qemu: rtc.bin
    qemu-system-aarch64 -machine virt,gic-version=3 -cpu max -serial mon:stdio -display

clean:
    cargo clean
    rm -f *.bin
```

*.cargo/config.toml* (you shouldn't need to change this):

```
[build]
target = "aarch64-unknown-none"
```

Run the code in QEMU with `make qemu`.

## 56.2 Yalın (Bare) Metal Rust Öğleden Sonra

### RTC driver

(back to exercise)

*main.rs*:

```
#![no_main]
#![no_std]

mod exceptions;
mod logger;
mod pl031;

use crate::pl031::Rtc;
use arm_gic::{IntId, Trigger, irq_enable, wfi};
use chrono::{TimeZone, Utc};
use core::hint::spin_loop;
use aarch64_paging::paging::Attributes;
use aarch64_rt::{InitialPagetable, entry, initial_pagetable};
use arm_gic::gicv3::GicV3;
use arm_gic::gicv3::registers::{Gicd, GicrSgi};
use arm_pl011_uart::{PL011Registers, Uart, UniqueMmioPointer};
use core::panic::PanicInfo;
use core::ptr::NonNull;
use log::{LevelFilter, error, info, trace};
use smccc::Hvc;
use smccc::psci::system_off;

/// Base addresses of the GICv3.
const GICD_BASE_ADDRESS: *mut Gicd = 0x800_0000 as _;
const GICR_BASE_ADDRESS: *mut GicrSgi = 0x80A_0000 as _;

/// Base address of the primary PL011 UART.
const PL011_BASE_ADDRESS: NonNull<PL011Registers> =
    NonNull::new(0x900_0000 as _).unwrap();

/// Attributes to use for device memory in the initial identity map.
const DEVICE_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_0)
    .union(Attributes::ACCESSED)
    .union(Attributes::UXN);

/// Attributes to use for normal memory in the initial identity map.
const MEMORY_ATTRIBUTES: Attributes = Attributes::VALID
    .union(Attributes::ATTRIBUTE_INDEX_1)
    .union(Attributes::INNER_SHAREABLE)
    .union(Attributes::ACCESSED)
    .union(Attributes::NON_GLOBAL);

initial_pagetable!({
```

```
    let mut idmap = [0; 512];
    // 1 GiB of device memory.
    idmap[0] = DEVICE_ATTRIBUTES.bits();
    // 1 GiB of normal memory.
    idmap[1] = MEMORY_ATTRIBUTES.bits() | 0x40000000;
    // Another 1 GiB of device memory starting at 256 GiB.
    idmap[256] = DEVICE_ATTRIBUTES.bits() | 0x4000000000;
    InitialPagetable(idmap)
});

/// Base address of the PL031 RTC.
const PL031_BASE_ADDRESS: NonNull<pl031::Registers> =
    NonNull::new(0x901_0000 as _).unwrap();
/// The IRQ used by the PL031 RTC.
const PL031_IRQ: IntId = IntId::spi(2);

entry!(main);
fn main(x0: u64, x1: u64, x2: u64, x3: u64) -> ! {
    // SAFETY: `PL011_BASE_ADDRESS` is the base address of a PL011 device, and
    // nothing else accesses that address range.
    let uart = unsafe { Uart::new(UniqueMmioPointer::new(PL011_BASE_ADDRESS)) };
    logger::init(uart, LevelFilter::Trace).unwrap();

    info!("main({:#x}, {:#x}, {:#x}, {:#x})", x0, x1, x2, x3);

    // SAFETY: `GICD_BASE_ADDRESS` and `GICR_BASE_ADDRESS` are the base
    // addresses of a GICv3 distributor and redistributor respectively, and
    // nothing else accesses those address ranges.
    let mut gic =
        unsafe { GicV3::new(GICD_BASE_ADDRESS, GICR_BASE_ADDRESS, 1, false) };
    gic.setup(0);

    // SAFETY: `PL031_BASE_ADDRESS` is the base address of a PL031 device, and
    // nothing else accesses that address range.
    let mut rtc = unsafe { Rtc::new(UniqueMmioPointer::new(PL031_BASE_ADDRESS)) };
    let timestamp = rtc.read();
    let time = Utc.timestamp_opt(timestamp.into(), 0).unwrap();
    info!("RTC: {time}");

    GicV3::set_priority_mask(0xff);
    gic.set_interrupt_priority(PL031_IRQ, None, 0x80);
    gic.set_trigger(PL031_IRQ, None, Trigger::Level);
    irq_enable();
    gic.enable_interrupt(PL031_IRQ, None, true);

    // Wait for 3 seconds, without interrupts.
    let target = timestamp + 3;
    rtc.set_match(target);
    info!("Waiting for {}", Utc.timestamp_opt(target.into(), 0).unwrap());
    trace!(
        "matched={}, interrupt_pending={}",
```

```
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.matched() {
        spin_loop();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("Finished waiting");

    // Wait another 3 seconds for an interrupt.
    let target = timestamp + 6;
    info!("Waiting for {}", Utc.timestamp_opt(target.into(), 0).unwrap());
    rtc.set_match(target);
    rtc.clear_interrupt();
    rtc.enable_interrupt(true);
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    while !rtc.interrupt_pending() {
        wfi();
    }
    trace!(
        "matched={}, interrupt_pending={}",
        rtc.matched(),
        rtc.interrupt_pending()
    );
    info!("Finished waiting");

    system_off::<Hvc>().unwrap();
    panic!("system_off returned");
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    error!("{info}");
    system_off::<Hvc>().unwrap();
    loop {}
}
```

*pl031.rs*:

```
#[repr(C, align(4))]
pub struct Registers {
    /// Data register
    dr: ReadPure<u32>,
    /// Match register
```

```
    mr: ReadPureWrite<u32>,
    /// Load register
    lr: ReadPureWrite<u32>,
    /// Control register
    cr: ReadPureWrite<u8>,
    _reserved0: [u8; 3],
    /// Interrupt Mask Set or Clear register
    imsc: ReadPureWrite<u8>,
    _reserved1: [u8; 3],
    /// Raw Interrupt Status
    ris: ReadPure<u8>,
    _reserved2: [u8; 3],
    /// Masked Interrupt Status
    mis: ReadPure<u8>,
    _reserved3: [u8; 3],
    /// Interrupt Clear Register
    icr: WriteOnly<u8>,
    _reserved4: [u8; 3],
}

/// Driver for a PL031 real-time clock.
#[derive(Debug)]
pub struct Rtc<'a> {
    registers: UniqueMmioPointer<'a, Registers>,
}

impl<'a> Rtc<'a> {
    /// Constructs a new instance of the RTC driver for a PL031 device with the
    /// given set of registers.
    pub fn new(registers: UniqueMmioPointer<'a, Registers>) -> Self {
        Self { registers }
    }

    /// Reads the current RTC value.
    pub fn read(&self) -> u32 {
        field_shared!(self.registers, dr).read()
    }

    /// Writes a match value. When the RTC value matches this then an interrupt
    /// will be generated (if it is enabled).
    pub fn set_match(&mut self, value: u32) {
        field!(self.registers, mr).write(value);
    }

    /// Returns whether the match register matches the RTC value, whether or not
    /// the interrupt is enabled.
    pub fn matched(&self) -> bool {
        let ris = field_shared!(self.registers, ris).read();
        (ris & 0x01) != 0
    }
```

```rust
    /// Returns whether there is currently an interrupt pending.
    ///
    /// This should be true if and only if `matched` returns true and the
    /// interrupt is masked.
    pub fn interrupt_pending(&self) -> bool {
        let mis = field_shared!(self.registers, mis).read();
        (mis & 0x01) != 0
    }

    /// Sets or clears the interrupt mask.
    ///
    /// When the mask is true the interrupt is enabled; when it is false the
    /// interrupt is disabled.
    pub fn enable_interrupt(&mut self, mask: bool) {
        let imsc = if mask { 0x01 } else { 0x00 };
        field!(self.registers, imsc).write(imsc);
    }

    /// Clears a pending interrupt, if any.
    pub fn clear_interrupt(&mut self) {
        field!(self.registers, icr).write(0x01);
    }
}
```

# Part XIII

# Eşzamanlılık: Sabah

# Chapter 57

# Welcome to Concurrency in Rust

Rust has full support for concurrency using OS threads with mutexes and channels.

The Rust type system plays an important role in making many concurrency bugs compile time bugs. This is often referred to as *fearless concurrency* since you can rely on the compiler to ensure correctness at runtime.

## Zamanlama (Schedule)

Bu oturum 10 dakikalık aralar dahil yaklaşık 3 saat 20 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| İş Parçacıkları (Threads) | 30 dakika |
| Kanallar | 20 dakika |
| Send ve Sync | 15 dakika |
| Paylaşımlı Durum (State) | 30 dakika |
| Alıştırmalar | 1 saat 10 dakika |

- Rust lets us access OS concurrency toolkit: threads, sync. primitives, etc.
- The type system gives us safety for concurrency without any special features.
- The same tools that help with ”concurrent” access in a single thread (e.g., a called function that might mutate an argument or save references to it to read later) save us from multi-threading issues.

# Chapter 58

# İş Parçacıkları (Threads)

Bu bölüm yaklaşık 30 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Temel İş Parçacıkları (Plain Threads) | 15 dakika |
| Kapsamlı İş Parçacıkları (Threads) | 15 dakika |

## 58.1 Temel İş Parçacıkları (Plain Threads)

Rust threads work similarly to threads in other languages:

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 0..10 {
            println!("Count in thread: {i}!");
            thread::sleep(Duration::from_millis(5));
        }
    });

    for i in 0..5 {
        println!("Main thread: {i}");
        thread::sleep(Duration::from_millis(5));
    }
}
```

- Spawning new threads does not automatically delay program termination at the end of `main`.
- Thread panics are independent of each other.
  - Panics can carry a payload, which can be unpacked with `Any::downcast_ref`.

This slide should take about 15 minutes.

- Run the example.
  - 5ms timing is loose enough that main and spawned threads stay mostly in lockstep.
  - Notice that the program ends before the spawned thread reaches 10!
  - This is because `main` ends the program and spawned threads do not make it persist.
    * Compare to `pthreads`/C++ `std::thread`/`boost::thread` if desired.
- How do we wait around for the spawned thread to complete?
- `thread::spawn` returns a `JoinHandle`. Look at the docs.
  - `JoinHandle` has a `.join()` method that blocks.
- Use `let handle = thread::spawn(...)` and later `handle.join()` to wait for the thread to finish and have the program count all the way to 10.
- Now what if we want to return a value?
- Look at docs again:
  - `thread::spawn`'s closure returns `T`
  - `JoinHandle .join()` returns `thread::Result<T>`
- Use the `Result` return value from `handle.join()` to get access to the returned value.
- Ok, what about the other case?
  - Trigger a panic in the thread. Note that this doesn't panic `main`.
  - Access the panic payload. This is a good time to talk about `Any`.
- Now we can return values from threads! What about taking inputs?
  - Capture something by reference in the thread closure.
  - An error message indicates we must move it.
  - Move it in, see we can compute and then return a derived value.
- If we want to borrow?
  - Main kills child threads when it returns, but another function would just return and leave them running.
  - That would be stack use-after-return, which violates memory safety!
  - How do we avoid this? See next slide.

## 58.2 Kapsamlı İş Parçacıkları (Threads)

Normal threads cannot borrow from their environment:

```
use std::thread;

fn foo() {
    let s = String::from("Merhaba");
    thread::spawn(|| {
        dbg!(s.len());
    });
}

fn main() {
```

```
    foo();
}
```

However, you can use a scoped thread for this:

```
use std::thread;

fn foo() {
    let s = String::from("Merhaba");
    thread::scope(|scope| {
        scope.spawn(|| {
            dbg!(s.len());
        });
    });
}

fn main() {
    foo();
}
```

This slide should take about 13 minutes.

- The reason for that is that when the `thread::scope` function completes, all the threads are guaranteed to be joined, so they can return borrowed data.
- Normal Rust borrowing rules apply: you can either borrow mutably by one thread, or immutably by any number of threads.

# Chapter 59

# Kanallar

Bu bölüm yaklaşık 20 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Vericiler ve Alıcılar | 10 dakika |
| Sınırsız Kanallar | 2 dakika |
| Sınırlı Kanallar | 10 dakika |

## 59.1 Vericiler ve Alıcılar

Rust channels have two parts: a `Sender<T>` and a `Receiver<T>`. The two parts are connected via the channel, but you only see the end-points.

```
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    tx.send(10).unwrap();
    tx.send(20).unwrap();

    println!("Received: {:?}", rx.recv());
    println!("Received: {:?}", rx.recv());

    let tx2 = tx.clone();
    tx2.send(30).unwrap();
    println!("Received: {:?}", rx.recv());
}
```

This slide should take about 9 minutes.

- `mpsc` stands for Multi-Producer, Single-Consumer. `Sender` and `SyncSender` implement `Clone` (so you can make multiple producers) but `Receiver` does not.
- `send()` and `recv()` return `Result`. If they return `Err`, it means the counterpart `Sender` or `Receiver` is dropped and the channel is closed.

## 59.2 Sınırsız Kanallar

You get an unbounded and asynchronous channel with `mpsc::channel()`:

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

This slide should take about 2 minutes.

- An unbounded channel will allocate as much space as is necessary to store pending messages. The `send()` method will not block the calling thread.
- A call to `send()` will abort with an error (that is why it returns `Result`) if the channel is closed. A channel is closed when the receiver is dropped.

## 59.3 Sınırlı Kanallar

With bounded (synchronous) channels, `send()` can block the current thread:

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::sync_channel(3);

    thread::spawn(move || {
        let thread_id = thread::current().id();
        for i in 0..10 {
            tx.send(format!("Message {i}")).unwrap();
            println!("{thread_id:?}: sent Message {i}");
        }
        println!("{thread_id:?}: done");
    });
```

```
    thread::sleep(Duration::from_millis(100));

    for msg in rx.iter() {
        println!("Main: got {msg}");
    }
}
```

This slide should take about 8 minutes.

- Calling `send()` will block the current thread until there is space in the channel for the new message. The thread can be blocked indefinitely if there is nobody who reads from the channel.
- Like unbounded channels, a call to `send()` will abort with an error if the channel is closed.
- A bounded channel with a size of zero is called a "rendezvous channel". Every send will block the current thread until another thread calls `recv()`.

# Chapter 60

# Send ve Sync

Bu bölüm yaklaşık 15 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Marker Özellikleri (Traits) | 2 dakika |
| Send | 2 dakika |
| Sync | 2 dakika |
| Örnekler | 10 dakika |

## 60.1 Marker Özellikleri (Traits)

How does Rust know to forbid shared access across threads? The answer is in two traits:

- Send: a type T is Send if it is safe to move a T across a thread boundary.
- Sync: a type T is Sync if it is safe to move a &T across a thread boundary.

Send and Sync are unsafe traits. The compiler will automatically derive them for your types as long as they only contain Send and Sync types. You can also implement them manually when you know it is valid.

This slide should take about 2 minutes.

- One can think of these traits as markers that the type has certain thread-safety properties.
- They can be used in the generic constraints as normal traits.

## 60.2 Send

> A type T is Send if it is safe to move a T value to another thread.

The effect of moving ownership to another thread is that *destructors* will run in that thread. So the question is when you can allocate a value in one thread and deallocate it in another.

This slide should take about 2 minutes.

As an example, a connection to the SQLite library must only be accessed from a single thread.

## 60.3 Sync

> A type `T` is Sync if it is safe to access a `T` value from multiple threads at the same time.

More precisely, the definition is:

> `T` is `Sync` if and only if `&T` is `Send`

This slide should take about 2 minutes.

This statement is essentially a shorthand way of saying that if a type is thread-safe for shared use, it is also thread-safe to pass references of it across threads.

This is because if a type is Sync it means that it can be shared across multiple threads without the risk of data races or other synchronization issues, so it is safe to move it to another thread. A reference to the type is also safe to move to another thread, because the data it references can be accessed from any thread safely.

## 60.4 Örnekler

### `Send + Sync`

Most types you come across are `Send + Sync`:

- `i8`, `f32`, `bool`, `char`, `&str`, ...
- `(T1, T2)`, `[T; N]`, `&[T]`, `struct { x: T }`, ...
- `String`, `Option<T>`, `Vec<T>`, `Box<T>`, ...
- `Arc<T>`: Explicitly thread-safe via atomic reference count.
- `Mutex<T>`: Explicitly thread-safe via internal locking.
- `mpsc::Sender<T>`: As of 1.72.0.
- `AtomicBool`, `AtomicU8`, ...: Uses special atomic instructions.

The generic types are typically `Send + Sync` when the type parameters are `Send + Sync`.

### `Send + !Sync`

These types can be moved to other threads, but they're not thread-safe. Typically because of interior mutability:

- `mpsc::Receiver<T>`
- `Cell<T>`
- `RefCell<T>`

### `!Send + Sync`

These types are safe to access (via shared references) from multiple threads, but they cannot be moved to another thread:

- `MutexGuard<T: Sync>`: Uses OS level primitives which must be deallocated on the thread which created them. However, an already-locked mutex can have its guarded variable read by any thread with which the guard is shared.

### `!Send + !Sync`

These types are not thread-safe and cannot be moved to other threads:

- `Rc<T>`: each `Rc<T>` has a reference to an `RcBox<T>`, which contains a non-atomic reference count.
- `*const T`, `*mut T`: Rust assumes raw pointers may have special concurrency considerations.

# Chapter 61

# Paylaşımlı Durum (State)

Bu bölüm yaklaşık 30 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Arc | 5 dakika |
| Mutex | 15 dakika |
| Örnek | 10 dakika |

## 61.1 `Arc`

`Arc<T>` allows shared, read-only ownership via `Arc::clone`:

```
use std::sync::Arc;
use std::thread;

/// A struct that prints which thread drops it.
#[derive(Debug)]
struct WhereDropped(Vec<i32>);

impl Drop for WhereDropped {
    fn drop(&mut self) {
        println!("Dropped by {:?}", thread::current().id())
    }
}

fn main() {
    let v = Arc::new(WhereDropped(vec![10, 20, 30]));
    let mut handles = Vec::new();
    for i in 0..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            // Sleep for 0-500ms.
            std::thread::sleep(std::time::Duration::from_millis(500 - i * 100));
            let thread_id = thread::current().id();
```

```
            println!("{thread_id:?}: {v:?}");
        }));
    }

    // Now only the spawned threads will hold clones of `v`.
    drop(v);

    // When the last spawned thread finishes, it will drop `v`'s contents.
    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

This slide should take about 5 minutes.

- `Arc` stands for "Atomic Reference Counted", a thread safe version of `Rc` that uses atomic operations.
- `Arc<T>` implements `Clone` whether or not `T` does. It implements `Send` and `Sync` if and only if `T` implements them both.
- `Arc::clone()` has the cost of atomic operations that get executed, but after that the use of the `T` is free.
- Beware of reference cycles, `Arc` does not use a garbage collector to detect them.
  - `std::sync::Weak` can help.

## 61.2 Mutex

`Mutex<T>` ensures mutual exclusion *and* allows mutable access to `T` behind a read-only interface (another form of interior mutability):

```
use std::sync::Mutex;

fn main() {
    let v = Mutex::new(vec![10, 20, 30]);
    println!("v: {:?}", v.lock().unwrap());

    {
        let mut guard = v.lock().unwrap();
        guard.push(40);
    }

    println!("v: {:?}", v.lock().unwrap());
}
```

Notice how we have a `impl<T: Send> Sync for Mutex<T>` blanket implementation.

This slide should take about 14 minutes.

- `Mutex` in Rust looks like a collection with just one element --- the protected data.
  - It is not possible to forget to acquire the mutex before accessing the protected data.
- You can get an `&mut T` from an `&Mutex<T>` by taking the lock. The `MutexGuard` ensures that the `&mut T` doesn't outlive the lock being held.
- `Mutex<T>` implements both `Send` and `Sync` if and only if `T` implements `Send`.
- A read-write lock counterpart: `RwLock`.
- Why does `lock()` return a `Result`?

- If the thread that held the `Mutex` panicked, the `Mutex` becomes "poisoned" to signal that the data it protected might be in an inconsistent state. Calling `lock()` on a poisoned mutex fails with a `PoisonError`. You can call `into_inner()` on the error to recover the data regardless.

## 61.3 Örnek

Let us see `Arc` and `Mutex` in action:

```
use std::thread;
// use std::sync::{Arc, Mutex};

fn main() {
    let v = vec![10, 20, 30];
    let mut handles = Vec::new();
    for i in 0..5 {
        handles.push(thread::spawn(|| {
            v.push(10 * i);
            println!("v: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

This slide should take about 8 minutes.

Possible solution:

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let v = Arc::new(Mutex::new(vec![10, 20, 30]));
    let mut handles = Vec::new();
    for i in 0..5 {
        let v = Arc::clone(&v);
        handles.push(thread::spawn(move || {
            let mut v = v.lock().unwrap();
            v.push(10 * i);
            println!("v: {v:?}");
        }));
    }

    handles.into_iter().for_each(|h| h.join().unwrap());
}
```

Notable parts:

- `v` is wrapped in both `Arc` and `Mutex`, because their concerns are orthogonal.
  - Wrapping a `Mutex` in an `Arc` is a common pattern to share mutable state between threads.

- `v: Arc<_>` needs to be cloned to make a new reference for each new spawned thread. Note `move` was added to the lambda signature.
- Blocks are introduced to narrow the scope of the `LockGuard` as much as possible.

# Chapter 62

# Alıştırmalar

Bu bölüm yaklaşık 1 saat 10 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Filozofların Akşam Yemeği | 20 dakika |
| Çok İş Parçacıklı Link Denetleyicisi | 20 dakika |
| Çözümler | 30 dakika |

## 62.1 Filozofların Akşam Yemeği

The dining philosophers problem is a classic problem in concurrency:

> Five philosophers dine together at the same table. Each philosopher has their own place at the table. There is a chopstick between each plate. The dish served is spaghetti which requires two chopsticks to eat. Each philosopher can only alternately think and eat. Moreover, a philosopher can only eat their spaghetti when they have both a left and right chopstick. Thus two chopsticks will only be available when their two nearest neighbors are thinking, not eating. After an individual philosopher finishes eating, they will put down both chopsticks.

You will need a local Cargo installation for this exercise. Copy the code below to a file called `src/main.rs`, fill out the blanks, and test that `cargo run` does not deadlock:

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
use std::time::Duration;

struct Chopstick;

struct Philosopher {
    name: String,
    // left_chopstick: ...
    // right_chopstick: ...
    // thoughts: ...
```

```
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        // Pick up chopsticks...
        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Hypatia", "Plato", "Aristotle", "Pythagoras"];

fn main() {
    // Create chopsticks

    // Create philosophers

    // Make each of them think and eat 100 times

    // Output their thoughts
}
```

You can use the following `Cargo.toml`:

```
[package]
name = "dining-philosophers"
version = "0.1.0"
edition = "2024"
```

This slide should take about 20 minutes.

- Encourage students to focus first on implementing a solution that ”mostly” works.
- The deadlock in the simplest solution is a general concurrency problem and highlights that Rust does not automatically prevent this sort of bug.

## 62.2 Çok İş Parçacıklı Link Denetleyicisi

Let us use our new knowledge to create a multi-threaded link checker. It should start at a webpage and check that links on the page are valid. It should recursively check other pages on the same domain and keep doing this until all pages have been validated.

For this, you will need an HTTP client such as `reqwest`. You will also need a way to find links, we can use `scraper`. Finally, we'll need some way of handling errors, we will use `thiserror`.

Create a new Cargo project and `reqwest` it as a dependency with:

```
cargo new link-checker
cd link-checker
cargo add --features blocking,rustls-tls reqwest
cargo add scraper
cargo add thiserror
```

> If `cargo add` fails with `error: no such subcommand`, then please edit the `Cargo.toml` file by hand. Add the dependencies listed below.

The `cargo add` calls will update the `Cargo.toml` file to look like this:

```
[package]
name = "link-checker"
version = "0.1.0"
edition = "2024"
publish = false

[dependencies]
reqwest = { version = "0.11.12", features = ["blocking", "rustls-tls"] }
scraper = "0.13.0"
thiserror = "1.0.37"
```

You can now download the start page. Try with a small site such as `https://www.google.org/`.

Your `src/main.rs` file should look something like this:

```
use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}

fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
```

```
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}: {err}");
            }
        }
    }
    Ok(link_urls)
}

fn main() {
    let client = Client::new();
    let start_url = Url::parse("https://www.google.org").unwrap();
    let crawl_command = CrawlCommand{ url: start_url, extract_links: true };
    match visit_page(&client, &crawl_command) {
        Ok(links) => println!("Links: {links:#?}"),
        Err(err) => println!("Could not extract links: {err:#}"),
    }
}
```

Run the code in `src/main.rs` with

```
cargo run
```

## Görevler

- Use threads to check the links in parallel: send the URLs to be checked to a channel and let a few threads check the URLs in parallel.
- Extend this to recursively extract links from all pages on the `www.google.org` domain. Put an upper limit of 100 pages or so so that you don't end up being blocked by the site.

This slide should take about 20 minutes.

- This is a complex exercise and intended to give students an opportunity to work on a larger project than others. A success condition for this exercise is to get stuck on some ”real” issue and work through it with the support of other students or the instructor.

## 62.3 Çözümler

### Filozofların Akşam Yemeği

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;
use std::time::Duration;

struct Chopstick;

struct Philosopher {
    name: String,
    left_chopstick: Arc<Mutex<Chopstick>>,
    right_chopstick: Arc<Mutex<Chopstick>>,
    thoughts: mpsc::SyncSender<String>,
}

impl Philosopher {
    fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .unwrap();
    }

    fn eat(&self) {
        println!("{} is trying to eat", &self.name);
        let _left = self.left_chopstick.lock().unwrap();
        let _right = self.right_chopstick.lock().unwrap();

        println!("{} is eating...", &self.name);
        thread::sleep(Duration::from_millis(10));
    }
}

static PHILOSOPHERS: &[&str] =
    &["Socrates", "Hypatia", "Plato", "Aristotle", "Pythagoras"];

fn main() {
    let (tx, rx) = mpsc::sync_channel(10);

    let chopsticks = PHILOSOPHERS
        .iter()
        .map(|_| Arc::new(Mutex::new(Chopstick)))
        .collect::<Vec<_>>();

    for i in 0..chopsticks.len() {
        let tx = tx.clone();
        let mut left_chopstick = Arc::clone(&chopsticks[i]);
        let mut right_chopstick =
            Arc::clone(&chopsticks[(i + 1) % chopsticks.len()]);
```

```
        // To avoid a deadlock, we have to break the symmetry
        // somewhere. This will swap the chopsticks without deinitializing
        // either of them.
        if i == chopsticks.len() - 1 {
            std::mem::swap(&mut left_chopstick, &mut right_chopstick);
        }

        let philosopher = Philosopher {
            name: PHILOSOPHERS[i].to_string(),
            thoughts: tx,
            left_chopstick,
            right_chopstick,
        };

        thread::spawn(move || {
            for _ in 0..100 {
                philosopher.eat();
                philosopher.think();
            }
        });
    }

    drop(tx);
    for thought in rx {
        println!("{thought}");
    }
}
```

### Link Checker

```
use std::sync::{Arc, Mutex, mpsc};
use std::thread;

use reqwest::Url;
use reqwest::blocking::Client;
use scraper::{Html, Selector};
use thiserror::Error;

#[derive(Error, Debug)]
enum Error {
    #[error("request error: {0}")]
    ReqwestError(#[from] reqwest::Error),
    #[error("bad http response: {0}")]
    BadResponse(String),
}

#[derive(Debug)]
struct CrawlCommand {
    url: Url,
    extract_links: bool,
}
```

```
fn visit_page(client: &Client, command: &CrawlCommand) -> Result<Vec<Url>, Error> {
    println!("Checking {:#}", command.url);
    let response = client.get(command.url.clone()).send()?;
    if !response.status().is_success() {
        return Err(Error::BadResponse(response.status().to_string()));
    }

    let mut link_urls = Vec::new();
    if !command.extract_links {
        return Ok(link_urls);
    }

    let base_url = response.url().to_owned();
    let body_text = response.text()?;
    let document = Html::parse_document(&body_text);

    let selector = Selector::parse("a").unwrap();
    let href_values = document
        .select(&selector)
        .filter_map(|element| element.value().attr("href"));
    for href in href_values {
        match base_url.join(href) {
            Ok(link_url) => {
                link_urls.push(link_url);
            }
            Err(err) => {
                println!("On {base_url:#}: ignored unparsable {href:?}: {err}");
            }
        }
    }
    Ok(link_urls)
}

struct CrawlState {
    domain: String,
    visited_pages: std::collections::HashSet<String>,
}

impl CrawlState {
    fn new(start_url: &Url) -> CrawlState {
        let mut visited_pages = std::collections::HashSet::new();
        visited_pages.insert(start_url.as_str().to_string());
        CrawlState { domain: start_url.domain().unwrap().to_string(), visited_pages }
    }

    /// Determine whether links within the given page should be extracted.
    fn should_extract_links(&self, url: &Url) -> bool {
        let Some(url_domain) = url.domain() else {
            return false;
        };
```

```
        url_domain == self.domain
    }

    /// Mark the given page as visited, returning false if it had already
    /// been visited.
    fn mark_visited(&mut self, url: &Url) -> bool {
        self.visited_pages.insert(url.as_str().to_string())
    }
}

type CrawlResult = Result<Vec<Url>, (Url, Error)>;
fn spawn_crawler_threads(
    command_receiver: mpsc::Receiver<CrawlCommand>,
    result_sender: mpsc::Sender<CrawlResult>,
    thread_count: u32,
) {
    // To multiplex the non-cloneable Receiver, wrap it in Arc<Mutex<_>>.
    let command_receiver = Arc::new(Mutex::new(command_receiver));

    for _ in 0..thread_count {
        let result_sender = result_sender.clone();
        let command_receiver = Arc::clone(&command_receiver);
        thread::spawn(move || {
            let client = Client::new();
            loop {
                let command_result = {
                    let receiver_guard = command_receiver.lock().unwrap();
                    receiver_guard.recv()
                };
                let Ok(crawl_command) = command_result else {
                    // The sender got dropped. No more commands coming in.
                    break;
                };
                let crawl_result = match visit_page(&client, &crawl_command) {
                    Ok(link_urls) => Ok(link_urls),
                    Err(error) => Err((crawl_command.url, error)),
                };
                result_sender.send(crawl_result).unwrap();
            }
        });
    }
}

fn control_crawl(
    start_url: Url,
    command_sender: mpsc::Sender<CrawlCommand>,
    result_receiver: mpsc::Receiver<CrawlResult>,
) -> Vec<Url> {
    let mut crawl_state = CrawlState::new(&start_url);
    let start_command = CrawlCommand { url: start_url, extract_links: true };
    command_sender.send(start_command).unwrap();
```

```
    let mut pending_urls = 1;

    let mut bad_urls = Vec::new();
    while pending_urls > 0 {
        let crawl_result = result_receiver.recv().unwrap();
        pending_urls -= 1;

        match crawl_result {
            Ok(link_urls) => {
                for url in link_urls {
                    if crawl_state.mark_visited(&url) {
                        let extract_links = crawl_state.should_extract_links(&url);
                        let crawl_command = CrawlCommand { url, extract_links };
                        command_sender.send(crawl_command).unwrap();
                        pending_urls += 1;
                    }
                }
            }
            Err((url, error)) => {
                bad_urls.push(url);
                println!("Got crawling error: {:#}", error);
                continue;
            }
        }
    }
    bad_urls
}

fn check_links(start_url: Url) -> Vec<Url> {
    let (result_sender, result_receiver) = mpsc::channel::<CrawlResult>();
    let (command_sender, command_receiver) = mpsc::channel::<CrawlCommand>();
    spawn_crawler_threads(command_receiver, result_sender, 16);
    control_crawl(start_url, command_sender, result_receiver)
}

fn main() {
    let start_url = reqwest::Url::parse("https://www.google.org").unwrap();
    let bad_urls = check_links(start_url);
    println!("Bad URLs: {:#?}", bad_urls);
}
```

# Part XIV

# Eşzamanlılık: Öğleden Sonra

# Chapter 63

# Hoş Geldiniz

”Async” is a concurrency model where multiple tasks are executed concurrently by executing each task until it would block, then switching to another task that is ready to make progress. The model allows running a larger number of tasks on a limited number of threads. This is because the per-task overhead is typically very low and operating systems provide primitives for efficiently identifying I/O that is able to proceed.

Rust's asynchronous operation is based on ”futures”, which represent work that may be completed in the future. Futures are ”polled” until they signal that they are complete.

Futures are polled by an async runtime, and several different runtimes are available.

## Karşılaştırmalar

- Python has a similar model in its `asyncio`. However, its `Future` type is callback-based, and not polled. Async Python programs require a ”loop”, similar to a runtime in Rust.
- JavaScript's `Promise` is similar, but again callback-based. The language runtime implements the event loop, so many of the details of Promise resolution are hidden.

## Zamanlama (Schedule)

Bu oturum 10 dakikalık aralar dahil yaklaşık 3 saat 30 dakika sürmelidir. İçeriği:

| Bölüm | Süre |
|---|---|
| Async Temelleri | 40 dakika |
| Kanallar ve Kontrol Akışı | 20 dakika |
| Tuzaklar | 55 dakika |
| Alıştırmalar | 1 saat 10 dakika |

# Chapter 64

# Async Temelleri

Bu bölüm yaklaşık 40 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| async/await | 10 dakika |
| Future Özellikleri (Traits) | 4 dakika |
| Durum Makinesi | 10 dakika |
| Çalışma Zamanları | 10 dakika |
| Görevler | 10 dakika |

## 64.1 `async`/`await`

At a high level, async Rust code looks very much like ”normal” sequential code:

```
use futures::executor::block_on;

async fn count_to(count: i32) {
    for i in 0..count {
        println!("Count is: {i}!");
    }
}

async fn async_main(count: i32) {
    count_to(count).await;
}

fn main() {
    block_on(async_main(10));
}
```

This slide should take about 6 minutes.

Anahtar noktalar:

- Note that this is a simplified example to show the syntax. There is no long running operation or any real concurrency in it!
- The ”async” keyword is syntactic sugar. The compiler replaces the return type with a future.
- You cannot make `main` async, without additional instructions to the compiler on how to use the returned future.
- You need an executor to run async code. `block_on` blocks the current thread until the provided future has run to completion.
- `.await` asynchronously waits for the completion of another operation. Unlike `block_on`, `.await` doesn't block the current thread.
- `.await` can only be used inside an `async` function (or block; these are introduced later).

## 64.2 Future Özellikleri (Traits)

`Future` is a trait, implemented by objects that represent an operation that may not be complete yet. A future can be polled, and `poll` returns a `Poll`.

```
use std::pin::Pin;
use std::task::Context;

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

An async function returns an `impl Future`. It's also possible (but uncommon) to implement `Future` for your own types. For example, the `JoinHandle` returned from `tokio::spawn` implements `Future` to allow joining to it.

The `.await` keyword, applied to a Future, causes the current async function to pause until that Future is ready, and then evaluates to its output.

This slide should take about 4 minutes.

- The `Future` and `Poll` types are implemented exactly as shown; click the links to show the implementations in the docs.
- `Context` allows a Future to schedule itself to be polled again when an event such as a timeout occurs.
- `Pin` ensures that the Future isn't moved in memory, so that pointers into that future remain valid. This is required to allow references to remain valid after an `.await`. We will address `Pin` in the ”Pitfalls” segment.

## 64.3 Durum Makinesi

Rust transforms an async function or block to a hidden type that implements `Future`, using a state machine to track the function's progress. The details of this transform are complex, but it helps to have a schematic understanding of what is happening. The following function

```
/// Sum two D10 rolls plus a modifier.
async fn two_d10(modifier: u32) -> u32 {
    let first_roll = roll_d10().await;
    let second_roll = roll_d10().await;
    first_roll + second_roll + modifier
}
```

is transformed to something like

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

/// Sum two D10 rolls plus a modifier.
fn two_d10(modifier: u32) -> TwoD10 {
    TwoD10::Init { modifier }
}

enum TwoD10 {
    // Function has not begun yet.
    Init { modifier: u32 },
    // Waiting for first `.await` to complete.
    FirstRoll { modifier: u32, fut: RollD10Future },
    // Waiting for second `.await` to complete.
    SecondRoll { modifier: u32, first_roll: u32, fut: RollD10Future },
}

impl Future for TwoD10 {
    type Output = u32;
    fn poll(mut self: Pin<&mut Self>, ctx: &mut Context) -> Poll<Self::Output> {
        loop {
            match *self {
                TwoD10::Init { modifier } => {
                    // Create future for first dice roll.
                    let fut = roll_d10();
                    *self = TwoD10::FirstRoll { modifier, fut };
                }
                TwoD10::FirstRoll { modifier, ref mut fut } => {
                    // Poll sub-future for first dice roll.
                    if let Poll::Ready(first_roll) = fut.poll(ctx) {
                        // Create future for second roll.
                        let fut = roll_d10();
                        *self = TwoD10::SecondRoll { modifier, first_roll, fut };
                    } else {
                        return Poll::Pending;
                    }
```

```
                    }
                    TwoD10::SecondRoll { modifier, first_roll, ref mut fut } => {
                        // Poll sub-future for second dice roll.
                        if let Poll::Ready(second_roll) = fut.poll(ctx) {
                            return Poll::Ready(first_roll + second_roll + modifier);
                        } else {
                            return Poll::Pending;
                        }
                    }
                }
            }
        }
    }
}
```

This slide should take about 10 minutes.

This example is illustrative, and isn't an accurate representation of the Rust compiler's transformation. The important things to notice here are:

- Calling an async function does nothing but construct and return a future.
- All local variables are stored in the function's future, using an enum to identify where execution is currently suspended.
- An `.await` in the async function is translated into an a new state containing all live variables and the awaited future. The `loop` then handles that updated state, polling the future until it returns `Poll::Ready`.
- Execution continues eagerly until a `Poll::Pending` occurs. In this simple example, every future is ready immediately.
- `main` contains a naïve executor, which just busy-loops until the future is ready. We will discuss real executors shortly.

## Daha Fazlasını Keşfedin

Imagine the `Future` data structure for a deeply nested stack of async functions. Each function's `Future` contains the `Future` structures for the functions it calls. This can result in unexpectedly large compiler-generated `Future` types.

This also means that recursive async functions are challenging. Compare to the common error of building recursive type, such as

```
enum LinkedList<T> {
    Node { value: T, next: LinkedList<T> },
    Nil,
}
```

The fix for a recursive type is to add a layer of indirection, such as with `Box`. Similarly, a recursive async function must box the recursive future:

```
async fn count_to(n: u32) {
    if n > 0 {
        Box::pin(count_to(n - 1)).await;
        println!("{n}");
    }
}
```

## 64.4 Çalışma Zamanları

A *runtime* provides support for performing operations asynchronously (a *reactor*) and is responsible for executing futures (an *executor*). Rust does not have a "built-in" runtime, but several options are available:

- Tokio: performant, with a well-developed ecosystem of functionality like Hyper for HTTP or Tonic for gRPC.
- smol: simple and lightweight

Several larger applications have their own runtimes. For example, Fuchsia already has one.

This slide and its sub-slides should take about 10 minutes.

- Note that of the listed runtimes, only Tokio is supported in the Rust playground. The playground also does not permit any I/O, so most interesting async things can't run in the playground.
- Futures are "inert" in that they do not do anything (not even start an I/O operation) unless there is an executor polling them. This differs from JS Promises, for example, which will run to completion even if they are never used.

### 64.4.1 Tokio Kasası

Tokio provides:

- A multi-threaded runtime for executing asynchronous code.
- An asynchronous version of the standard library.
- A large ecosystem of libraries.

```
use tokio::time;

async fn count_to(count: i32) {
    for i in 0..count {
        println!("Count in task: {i}!");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

#[tokio::main]
async fn main() {
    tokio::spawn(count_to(10));

    for i in 0..5 {
        println!("Main task: {i}");
        time::sleep(time::Duration::from_millis(5)).await;
    }
}
```

- With the `tokio::main` macro we can now make `main` async.
- The `spawn` function creates a new, concurrent "task".
- Note: `spawn` takes a `Future`, you don't call `.await` on `count_to`.

**Further exploration:**

- Why does `count_to` not (usually) get to 10? This is an example of async cancellation. `tokio::spawn` returns a handle which can be awaited to wait until it finishes.
- Try `count_to(10).await` instead of spawning.
- Try awaiting the task returned from `tokio::spawn`.

## 64.5 Görevler

Rust has a task system, which is a form of lightweight threading.

A task has a single top-level future which the executor polls to make progress. That future may have one or more nested futures that its `poll` method polls, corresponding loosely to a call stack. Concurrency within a task is possible by polling multiple child futures, such as racing a timer and an I/O operation.

```
use tokio::io::{self, AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:0").await?;
    println!("listening on port {}", listener.local_addr()?.port());

    loop {
        let (mut socket, addr) = listener.accept().await?;

        println!("connection from {addr:?}");

        tokio::spawn(async move {
            socket.write_all(b"Who are you?\n").await.expect("socket error");

            let mut buf = vec![0; 1024];
            let name_size = socket.read(&mut buf).await.expect("socket error");
            let name = std::str::from_utf8(&buf[..name_size]).unwrap().trim();
            let reply = format!("Thanks for dialing in, {name}!\n");
            socket.write_all(reply.as_bytes()).await.expect("socket error");
        });
    }
}
```

This slide should take about 6 minutes.

Copy this example into your prepared `src/main.rs` and run it from there.

Try connecting to it with a TCP connection tool like nc or telnet.

- Ask students to visualize what the state of the example server would be with a few connected clients. What tasks exist? What are their Futures?
- This is the first time we've seen an `async` block. This is similar to a closure, but does not take any arguments. Its return value is a Future, similar to an `async fn`.
- Refactor the async block into a function, and improve the error handling using `?`.

# Chapter 65

# Kanallar ve Kontrol Akışı

Bu bölüm yaklaşık 20 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Async Kanalları | 10 dakika |
| Join | 4 dakika |
| Select | 5 dakika |

## 65.1 Async Kanalları

Several crates have support for asynchronous channels. For instance `tokio`:

```
use tokio::sync::mpsc;

async fn ping_handler(mut input: mpsc::Receiver<()>) {
    let mut count: usize = 0;

    while let Some(_) = input.recv().await {
        count += 1;
        println!("Received {count} pings so far.");
    }

    println!("ping_handler complete");
}

#[tokio::main]
async fn main() {
    let (sender, receiver) = mpsc::channel(32);
    let ping_handler_task = tokio::spawn(ping_handler(receiver));
    for i in 0..10 {
        sender.send(()).await.expect("Failed to send ping.");
        println!("Sent {} pings so far.", i + 1);
    }
```

```
    drop(sender);
    ping_handler_task.await.expect("Something went wrong in ping handler task.");
}
```

This slide should take about 8 minutes.

- Change the channel size to 3 and see how it affects the execution.
- Overall, the interface is similar to the `sync` channels as seen in the morning class.
- Try removing the `std::mem::drop` call. What happens? Why?
- The Flume crate has channels that implement both `sync` and `async` `send` and `recv`. This can be convenient for complex applications with both IO and heavy CPU processing tasks.
- What makes working with `async` channels preferable is the ability to combine them with other `futures` to combine them and create complex control flow.

## 65.2 Join

A join operation waits until all of a set of futures are ready, and returns a collection of their results. This is similar to `Promise.all` in JavaScript or `asyncio.gather` in Python.

```
use anyhow::Result;
use futures::future;
use reqwest;
use std::collections::HashMap;

async fn size_of_page(url: &str) -> Result<usize> {
    let resp = reqwest::get(url).await?;
    Ok(resp.text().await?.len())
}

#[tokio::main]
async fn main() {
    let urls: [&str; 4] = [
        "https://google.com",
        "https://httpbin.org/ip",
        "https://play.rust-lang.org/",
        "BAD_URL",
    ];
    let futures_iter = urls.into_iter().map(size_of_page);
    let results = future::join_all(futures_iter).await;
    let page_sizes_dict: HashMap<&str, Result<usize>> =
        urls.into_iter().zip(results.into_iter()).collect();
    println!("{page_sizes_dict:?}");
}
```

This slide should take about 4 minutes.

Copy this example into your prepared `src/main.rs` and run it from there.

- For multiple futures of disjoint types, you can use `std::future::join!` but you must know how many futures you will have at compile time. This is currently in the `futures`

crate, soon to be stabilised in `std::future`.

- The risk of `join` is that one of the futures may never resolve, this would cause your program to stall.
- You can also combine `join_all` with `join!` for instance to join all requests to an http service as well as a database query. Try adding a `tokio::time::sleep` to the future, using `futures::join!`. This is not a timeout (that requires `select!`, explained in the next chapter), but demonstrates `join!`.

## 65.3 Select

A select operation waits until any of a set of futures is ready, and responds to that future's result. In JavaScript, this is similar to `Promise.race`. In Python, it compares to `asyncio.wait(task_set, return_when=asyncio.FIRST_COMPLETED)`.

Similar to a match statement, the body of `select!` has a number of arms, each of the form `pattern = future => statement`. When a `future` is ready, its return value is destructured by the `pattern`. The `statement` is then run with the resulting variables. The `statement` result becomes the result of the `select!` macro.

```
use tokio::sync::mpsc;
use tokio::time::{Duration, sleep};

#[tokio::main]
async fn main() {
    let (tx, mut rx) = mpsc::channel(32);
    let listener = tokio::spawn(async move {
        tokio::select! {
            Some(msg) = rx.recv() => println!("got: {msg}"),
            _ = sleep(Duration::from_millis(50)) => println!("timeout"),
        };
    });
    sleep(Duration::from_millis(10)).await;
    tx.send(String::from("Merhaba!")).await.expect("Failed to send greeting");

    listener.await.expect("Listener failed");
}
```

This slide should take about 5 minutes.

- The `listener` async block here is a common form: wait for some async event, or for a timeout. Change the `sleep` to sleep longer to see it fail. Why does the `send` also fail in this situation?
- `select!` is also often used in a loop in ”actor” architectures, where a task reacts to events in a loop. That has some pitfalls, which will be discussed in the next segment.

# Chapter 66

# Tuzaklar

Async / await provides convenient and efficient abstraction for concurrent asynchronous programming. However, the async/await model in Rust also comes with its share of pitfalls and footguns. We illustrate some of them in this chapter.

Bu bölüm yaklaşık 55 dakika sürmelidir. İçerir:

| Slayt | Süre |
|---|---|
| Yürütücünün/Çalıştırıcının (Executor) Engellenmesi | 10 dakika |
| Pin | 20 dakika |
| Async Özellikleri (Traits) | 5 dakika |
| İptal (Cancellation) | 20 dakika |

## 66.1 Blocking the executor

Most async runtimes only allow IO tasks to run concurrently. This means that CPU blocking tasks will block the executor and prevent other tasks from being executed. An easy workaround is to use async equivalent methods where possible.

```
use futures::future::join_all;
use std::time::Instant;

async fn sleep_ms(start: &Instant, id: u64, duration_ms: u64) {
    std::thread::sleep(std::time::Duration::from_millis(duration_ms));
    println!(
        "future {id} slept for {duration_ms}ms, finished after {}ms",
        start.elapsed().as_millis()
    );
}

#[tokio::main(flavor = "current_thread")]
async fn main() {
    let start = Instant::now();
```

```
    let sleep_futures = (1..=10).map(|t| sleep_ms(&start, t, t * 10));
    join_all(sleep_futures).await;
}
```

This slide should take about 10 minutes.

- Run the code and see that the sleeps happen consecutively rather than concurrently.
- The `"current_thread"` flavor puts all tasks on a single thread. This makes the effect more obvious, but the bug is still present in the multi-threaded flavor.
- Switch the `std::thread::sleep` to `tokio::time::sleep` and await its result.
- Another fix would be to `tokio::task::spawn_blocking` which spawns an actual thread and transforms its handle into a future without blocking the executor.
- You should not think of tasks as OS threads. They do not map 1 to 1 and most executors will allow many tasks to run on a single OS thread. This is particularly problematic when interacting with other libraries via FFI, where that library might depend on thread-local storage or map to specific OS threads (e.g., CUDA). Prefer `tokio::task::spawn_blocking` in such situations.
- Use sync mutexes with care. Holding a mutex over an `.await` may cause another task to block, and that task may be running on the same thread.

## 66.2 `Pin`

Recall an async function or block creates a type implementing `Future` and containing all of the local variables. Some of those variables can hold references (pointers) to other local variables. To ensure those remain valid, the future can never be moved to a different memory location.

To prevent moving the future type in memory, it can only be polled through a pinned pointer. `Pin` is a wrapper around a reference that disallows all operations that would move the instance it points to into a different memory location.

```
use tokio::sync::{mpsc, oneshot};
use tokio::task::spawn;
use tokio::time::{Duration, sleep};

// A work item. In this case, just sleep for the given time and respond
// with a message on the `respond_on` channel.
#[derive(Debug)]
struct Work {
    input: u32,
    respond_on: oneshot::Sender<u32>,
}

// A worker which listens for work on a queue and performs it.
async fn worker(mut work_queue: mpsc::Receiver<Work>) {
    let mut iterations = 0;
    loop {
        tokio::select! {
            Some(work) = work_queue.recv() => {
```

```
                sleep(Duration::from_millis(10)).await; // Pretend to work.
                work.respond_on
                    .send(work.input * 1000)
                    .expect("failed to send response");
                iterations += 1;
            }
            // TODO: report number of iterations every 100ms
        }
    }
}

// A requester which requests work and waits for it to complete.
async fn do_work(work_queue: &mpsc::Sender<Work>, input: u32) -> u32 {
    let (tx, rx) = oneshot::channel();
    work_queue
        .send(Work { input, respond_on: tx })
        .await
        .expect("failed to send on work queue");
    rx.await.expect("failed waiting for response")
}

#[tokio::main]
async fn main() {
    let (tx, rx) = mpsc::channel(10);
    spawn(worker(rx));
    for i in 0..100 {
        let resp = do_work(&tx, i).await;
        println!("work result for iteration {i}: {resp}");
    }
}
```

This slide should take about 20 minutes.

- You may recognize this as an example of the actor pattern. Actors typically call `select!` in a loop.
- This serves as a summation of a few of the previous lessons, so take your time with it.
  - Naively add a `_ = sleep(Duration::from_millis(100)) => { println!(..) }` to the `select!`. This will never execute. Why?
  - Instead, add a `timeout_fut` containing that future outside of the loop:

    ```
    let timeout_fut = sleep(Duration::from_millis(100));
    loop {
        select! {
            ..,
            _ = timeout_fut => { println!(..); },
        }
    }
    ```

  - This still doesn't work. Follow the compiler errors, adding `&mut` to the `timeout_fut` in the `select!` to work around the move, then using `Box::pin`:

    ```
    let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
    ```

```
loop {
    select! {
        ..,
        _ = &mut timeout_fut => { println!(..); },
    }
}
```

  - This compiles, but once the timeout expires it is `Poll::Ready` on every iteration (a fused future would help with this). Update to reset `timeout_fut` every time it expires:

```
let mut timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
loop {
    select! {
        _ = &mut timeout_fut => {
            println!(..);
            timeout_fut = Box::pin(sleep(Duration::from_millis(100)));
        },
    }
}
```

- Box allocates on the heap. In some cases, `std::pin::pin!` (only recently stabilized, with older code often using `tokio::pin!`) is also an option, but that is difficult to use for a future that is reassigned.
- Another alternative is to not use `pin` at all but spawn another task that will send to a `oneshot` channel every 100ms.
- Data that contains pointers to itself is called self-referential. Normally, the Rust borrow checker would prevent self-referential data from being moved, as the references cannot outlive the data they point to. However, the code transformation for async blocks and functions is not verified by the borrow checker.
- `Pin` is a wrapper around a reference. An object cannot be moved from its place using a pinned pointer. However, it can still be moved through an unpinned pointer.
- The `poll` method of the `Future` trait uses `Pin<&mut Self>` instead of `&mut Self` to refer to the instance. That's why it can only be called on a pinned pointer.

## 66.3 Async Özellikleri (Traits)

Async methods in traits were stabilized in the 1.75 release. This required support for using return-position `impl Trait` in traits, as the desugaring for `async fn` includes `-> impl Future<Output = ...>`.

However, even with the native support, there are some pitfalls around `async fn`:

- Return-position `impl Trait` captures all in-scope lifetimes (so some patterns of borrowing cannot be expressed).
- Async traits cannot be used with trait objects (`dyn Trait` support).

The async_trait crate provides a workaround for dyn support through a macro, with some caveats:

```rust
use async_trait::async_trait;
use std::time::Instant;
use tokio::time::{Duration, sleep};

#[async_trait]
trait Sleeper {
    async fn sleep(&self);
}

struct FixedSleeper {
    sleep_ms: u64,
}

#[async_trait]
impl Sleeper for FixedSleeper {
    async fn sleep(&self) {
        sleep(Duration::from_millis(self.sleep_ms)).await;
    }
}

async fn run_all_sleepers_multiple_times(
    sleepers: Vec<Box<dyn Sleeper>>,
    n_times: usize,
) {
    for _ in 0..n_times {
        println!("Running all sleepers...");
        for sleeper in &sleepers {
            let start = Instant::now();
            sleeper.sleep().await;
            println!("Slept for {} ms", start.elapsed().as_millis());
        }
    }
}

#[tokio::main]
async fn main() {
    let sleepers: Vec<Box<dyn Sleeper>> = vec![
        Box::new(FixedSleeper { sleep_ms: 50 }),
        Box::new(FixedSleeper { sleep_ms: 100 }),
    ];
    run_all_sleepers_multiple_times(sleepers, 5).await;
}
```

This slide should take about 5 minutes.

- `async_trait` is easy to use, but note that it's using heap allocations to achieve this. This heap allocation has performance overhead.
- The challenges in language support for `async trait` are too deep to describe in-depth in this class. See this blog post by Niko Matsakis if you are interested in digging deeper. See also these keywords:
  - RPIT: short for return-position `impl Trait`.

  - RPITIT: short for return-position `impl Trait` in trait (RPIT in trait).
- Try creating a new sleeper struct that will sleep for a random amount of time and adding it to the `Vec`.

## 66.4 İptal (Cancellation)

Dropping a future implies it can never be polled again. This is called *cancellation* and it can occur at any `await` point. Care is needed to ensure the system works correctly even when futures are cancelled. For example, it shouldn't deadlock or lose data.

```
use std::io;
use std::time::Duration;
use tokio::io::{AsyncReadExt, AsyncWriteExt, DuplexStream};

struct LinesReader {
    stream: DuplexStream,
}

impl LinesReader {
    fn new(stream: DuplexStream) -> Self {
        Self { stream }
    }

    async fn next(&mut self) -> io::Result<Option<String>> {
        let mut bytes = Vec::new();
        let mut buf = [0];
        while self.stream.read(&mut buf[..]).await? != 0 {
            bytes.push(buf[0]);
            if buf[0] == b'\n' {
                break;
            }
        }
        if bytes.is_empty() {
            return Ok(None);
        }
        let s = String::from_utf8(bytes)
            .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "not UTF-8"))?;
        Ok(Some(s))
    }
}

async fn slow_copy(source: String, mut dest: DuplexStream) -> io::Result<()> {
    for b in source.bytes() {
        dest.write_u8(b).await?;
        tokio::time::sleep(Duration::from_millis(10)).await
    }
    Ok(())
}

#[tokio::main]
```

```
async fn main() -> io::Result<()> {
    let (client, server) = tokio::io::duplex(5);
    let handle = tokio::spawn(slow_copy("hi\nthere\n".to_owned(), client));

    let mut lines = LinesReader::new(server);
    let mut interval = tokio::time::interval(Duration::from_millis(60));
    loop {
        tokio::select! {
            _ = interval.tick() => println!("tick!"),
            line = lines.next() => if let Some(l) = line? {
                print!("{}", l)
            } else {
                break
            },
        }
    }
    handle.await.unwrap()?;
    Ok(())
}
```

This slide should take about 18 minutes.

- The compiler doesn't help with cancellation-safety. You need to read API documentation and consider what state your `async fn` holds.
- Unlike `panic` and `?`, cancellation is part of normal control flow (vs error-handling).
- The example loses parts of the string.
  - Whenever the `tick()` branch finishes first, `next()` and its `buf` are dropped.
  - `LinesReader` can be made cancellation-safe by making `buf` part of the struct:

    ```
    struct LinesReader {
        stream: DuplexStream,
        bytes: Vec<u8>,
        buf: [u8; 1],
    }

    impl LinesReader {
        fn new(stream: DuplexStream) -> Self {
            Self { stream, bytes: Vec::new(), buf: [0] }
        }
        async fn next(&mut self) -> io::Result<Option<String>> {
            // prefix buf and bytes with self.
            // ...
            let raw = std::mem::take(&mut self.bytes);
            let s = String::from_utf8(raw)
                .map_err(|_| io::Error::new(io::ErrorKind::InvalidData, "not UTF-8")
            // ...
        }
    }
    ```

- `Interval::tick` is cancellation-safe because it keeps track of whether a tick has been 'delivered'.

- `AsyncReadExt::read` is cancellation-safe because it either returns or doesn't read data.
- `AsyncBufReadExt::read_line` is similar to the example and *isn't* cancellation-safe. See its documentation for details and alternatives.

# Chapter 67

# Alıştırmalar

Bu bölüm yaklaşık 1 saat 10 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Filozofların Akşam Yemeği | 20 dakika |
| Yayımlamalı Sohbet Uygulaması | 30 dakika |
| Çözümler | 20 dakika |

## 67.1 Filozofların Akşam Yemeği --- Async

See dining philosophers for a description of the problem.

As before, you will need a local Cargo installation for this exercise. Copy the code below to a file called `src/main.rs`, fill out the blanks, and test that `cargo run` does not deadlock:

```
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

struct Philosopher {
    name: String,
    // left_chopstick: ...
    // right_chopstick: ...
    // thoughts: ...
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .await
            .unwrap();
```

```
    }

    async fn eat(&self) {
        // Keep trying until we have both chopsticks
        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
    }
}

// tokio scheduler doesn't deadlock with 5 philosophers, so have 2.
static PHILOSOPHERS: &[&str] = &["Socrates", "Hypatia"];

#[tokio::main]
async fn main() {
    // Create chopsticks

    // Create philosophers

    // Make them think and eat

    // Output their thoughts
}
```

Since this time you are using Async Rust, you'll need a `tokio` dependency. You can use the following `Cargo.toml`:

```
[package]
name = "dining-philosophers-async-dine"
version = "0.1.0"
edition = "2024"

[dependencies]
tokio = { version = "1.26.0", features = ["sync", "time", "macros", "rt-multi-thread"] }
```

Also note that this time you have to use the `Mutex` and the `mpsc` module from the `tokio` crate.

This slide should take about 20 minutes.

- Can you make your implementation single-threaded?

## 67.2 Yayımlamalı Sohbet Uygulaması

In this exercise, we want to use our new knowledge to implement a broadcast chat application. We have a chat server that the clients connect to and publish their messages. The client reads user messages from the standard input, and sends them to the server. The chat server broadcasts each message that it receives to all the clients.

For this, we use a broadcast channel on the server, and `tokio_websockets` for the communication between the client and the server.

Create a new Cargo project and add the following dependencies:

*Cargo.toml*:

```toml
[package]
name = "chat-async"
version = "0.1.0"
edition = "2024"

[dependencies]
futures-util = { version = "0.3.31", features = ["sink"] }
http = "1.3.1"
tokio = { version = "1.47.1", features = ["full"] }
tokio-websockets = { version = "0.12.1", features = ["client", "fastrand", "server", "sh
```

## The required APIs

You are going to need the following functions from `tokio` and `tokio_websockets`. Spend a few minutes to familiarize yourself with the API.

- StreamExt::next() implemented by `WebSocketStream`: for asynchronously reading messages from a Websocket Stream.
- SinkExt::send() implemented by `WebSocketStream`: for asynchronously sending messages on a Websocket Stream.
- Lines::next_line(): for asynchronously reading user messages from the standard input.
- Sender::subscribe(): for subscribing to a broadcast channel.

## Two binaries

Normally in a Cargo project, you can have only one binary, and one `src/main.rs` file. In this project, we need two binaries. One for the client, and one for the server. You could potentially make them two separate Cargo projects, but we are going to put them in a single Cargo project with two binaries. For this to work, the client and the server code should go under `src/bin` (see the documentation).

Copy the following server and client code into `src/bin/server.rs` and `src/bin/client.rs`, respectively. Your task is to complete these files as described below.

*src/bin/server.rs*:

```rust
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{Sender, channel};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    // TODO: For a hint, see the description of the task below.
```

```
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("listening on port 2000");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("New connection from {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Wrap the raw TCP stream into a websocket.
            let (_req, ws_stream) = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

*src/bin/client.rs*:

```
use futures_util::SinkExt;
use futures_util::stream::StreamExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();


    // TODO: For a hint, see the description of the task below.

}
```

### Running the binaries

Run the server with:

```
cargo run --bin server
```

and the client with:

```
cargo run --bin client
```

### Görevler

- Implement the `handle_connection` function in `src/bin/server.rs`.
  - Hint: Use `tokio::select!` for concurrently performing two tasks in a continuous loop. One task receives messages from the client and broadcasts them. The other sends messages received by the server to the client.
- Complete the main function in `src/bin/client.rs`.
  - Hint: As before, use `tokio::select!` in a continuous loop for concurrently performing two tasks: (1) reading user messages from standard input and sending them to the server, and (2) receiving messages from the server, and displaying them for the user.
- Optional: Once you are done, change the code to broadcast messages to all clients, but the sender of the message.

## 67.3 Çözümler

### Filozofların Akşam Yemeği --- Async

```
use std::sync::Arc;
use tokio::sync::{Mutex, mpsc};
use tokio::time;

struct Chopstick;

struct Philosopher {
    name: String,
    left_chopstick: Arc<Mutex<Chopstick>>,
    right_chopstick: Arc<Mutex<Chopstick>>,
    thoughts: mpsc::Sender<String>,
}

impl Philosopher {
    async fn think(&self) {
        self.thoughts
            .send(format!("Eureka! {} has a new idea!", &self.name))
            .await
            .unwrap();
    }

    async fn eat(&self) {
        // Keep trying until we have both chopsticks
        // Pick up chopsticks...
        let _left_chopstick = self.left_chopstick.lock().await;
        let _right_chopstick = self.right_chopstick.lock().await;

        println!("{} is eating...", &self.name);
        time::sleep(time::Duration::from_millis(5)).await;
```

```
        // The locks are dropped here
    }
}

// tokio scheduler doesn't deadlock with 5 philosophers, so have 2.
static PHILOSOPHERS: &[&str] = &["Socrates", "Hypatia"];

#[tokio::main]
async fn main() {
    // Create chopsticks
    let mut chopsticks = vec![];
    PHILOSOPHERS
        .iter()
        .for_each(|_| chopsticks.push(Arc::new(Mutex::new(Chopstick))));

    // Create philosophers
    let (philosophers, mut rx) = {
        let mut philosophers = vec![];
        let (tx, rx) = mpsc::channel(10);
        for (i, name) in PHILOSOPHERS.iter().enumerate() {
            let mut left_chopstick = Arc::clone(&chopsticks[i]);
            let mut right_chopstick =
                Arc::clone(&chopsticks[(i + 1) % PHILOSOPHERS.len()]);
            if i == PHILOSOPHERS.len() - 1 {
                std::mem::swap(&mut left_chopstick, &mut right_chopstick);
            }
            philosophers.push(Philosopher {
                name: name.to_string(),
                left_chopstick,
                right_chopstick,
                thoughts: tx.clone(),
            });
        }
        (philosophers, rx)
        // tx is dropped here, so we don't need to explicitly drop it later
    };

    // Make them think and eat
    for phil in philosophers {
        tokio::spawn(async move {
            for _ in 0..100 {
                phil.think().await;
                phil.eat().await;
            }
        });
    }

    // Output their thoughts
    while let Some(thought) = rx.recv().await {
        println!("Here is a thought: {thought}");
    }
```

```
}
```

### Yayımlamalı Sohbet Uygulaması

*src/bin/server.rs*:

```
use futures_util::sink::SinkExt;
use futures_util::stream::StreamExt;
use std::error::Error;
use std::net::SocketAddr;
use tokio::net::{TcpListener, TcpStream};
use tokio::sync::broadcast::{Sender, channel};
use tokio_websockets::{Message, ServerBuilder, WebSocketStream};

async fn handle_connection(
    addr: SocketAddr,
    mut ws_stream: WebSocketStream<TcpStream>,
    bcast_tx: Sender<String>,
) -> Result<(), Box<dyn Error + Send + Sync>> {

    ws_stream
        .send(Message::text("Welcome to chat! Type a message".to_string()))
        .await?;
    let mut bcast_rx = bcast_tx.subscribe();

    // A continuous loop for concurrently performing two tasks: (1) receiving
    // messages from `ws_stream` and broadcasting them, and (2) receiving
    // messages on `bcast_rx` and sending them to the client.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("From client {addr:?} {text:?}");
                            bcast_tx.send(text.into())?;
                        }
                    }
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            msg = bcast_rx.recv() => {
                ws_stream.send(Message::text(msg?)).await?;
            }
        }
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error + Send + Sync>> {
```

```
    let (bcast_tx, _) = channel(16);

    let listener = TcpListener::bind("127.0.0.1:2000").await?;
    println!("listening on port 2000");

    loop {
        let (socket, addr) = listener.accept().await?;
        println!("New connection from {addr:?}");
        let bcast_tx = bcast_tx.clone();
        tokio::spawn(async move {
            // Wrap the raw TCP stream into a websocket.
            let (_req, ws_stream) = ServerBuilder::new().accept(socket).await?;

            handle_connection(addr, ws_stream, bcast_tx).await
        });
    }
}
```

*src/bin/client.rs*:

```
use futures_util::SinkExt;
use futures_util::stream::StreamExt;
use http::Uri;
use tokio::io::{AsyncBufReadExt, BufReader};
use tokio_websockets::{ClientBuilder, Message};

#[tokio::main]
async fn main() -> Result<(), tokio_websockets::Error> {
    let (mut ws_stream, _) =
        ClientBuilder::from_uri(Uri::from_static("ws://127.0.0.1:2000"))
            .connect()
            .await?;

    let stdin = tokio::io::stdin();
    let mut stdin = BufReader::new(stdin).lines();

    // Continuous loop for concurrently sending and receiving messages.
    loop {
        tokio::select! {
            incoming = ws_stream.next() => {
                match incoming {
                    Some(Ok(msg)) => {
                        if let Some(text) = msg.as_text() {
                            println!("From server: {}", text);
                        }
                    },
                    Some(Err(err)) => return Err(err.into()),
                    None => return Ok(()),
                }
            }
            res = stdin.next_line() => {
                match res {
```

```
                    Ok(None) => return Ok(()),
                    Ok(Some(line)) => ws_stream.send(Message::text(line.to_string())).au
                    Err(err) => return Err(err.into()),
                }
            }

        }
    }
}
```

# Part XV

# Idiomatic Rust

# Chapter 68

# Welcome to Idiomatic Rust

Rust Fundamentals introduced Rust syntax and core concepts. We now want to go one step further: how do you use Rust *effectively* in your projects? What does *idiomatic* Rust look like?

This course is opinionated: we will nudge you towards some patterns, and away from others. Nonetheless, we do recognize that some projects may have different needs. We always provide the necessary information to help you make informed decisions within the context and constraints of your own projects.

> ⚠️ This course is under **active development**.
>
> The material may change frequently and there might be errors that have not yet been spotted. Nonetheless, we encourage you to browse through and provide early feedback!

## Zamanlama (Schedule)

Including 10 minute breaks, this session should take about 25 minutes. It contains:

| Bölüm | Süre |
|---|---|
| Leveraging the Type System | 25 minutes |

The course will cover the topics listed below. Each topic may be covered in one or more slides, depending on its complexity and relevance.

### Foundations of API design

- Golden rule: prioritize clarity and readability at the callsite. People will spend much more time reading the call sites than declarations of the functions being called.
- Make your API predictable
  - Follow naming conventions (case conventions, prefer vocabulary precedented in the standard library - e.g., methods should be called ”push” not ”push_back”, ”is_empty” not ”empty” etc.)

  - Know the vocabulary types and traits in the standard library, and use them in your APIs. If something feels like a basic type/algorithm, check in the standard library first.
  - Use well-established API design patterns that we will discuss later in this class (e.g., newtype, owned/view type pairs, error handling)
- Write meaningful and effective doc comments (e.g., don't merely repeat the method name with spaces instead of underscores, don't repeat the same information just to fill out every markdown tag, provide usage examples)

## Leveraging the type system

- Short recap on enums, structs and type aliases
- Newtype pattern and encapsulation: parse, don't validate
- Extension traits: avoid the newtype pattern when you want to provide additional behaviour
- RAII, scope guards and drop bombs: using `Drop` to clean up resources, trigger actions or enforce invariants
- ”Token” types: force users to prove they've performed a specific action
- The typestate pattern: enforce correct state transitions at compile-time
- Using the borrow checker to enforce invariants that have nothing to do with memory ownership
  - OwnedFd/BorrowedFd in the standard library
  - Branded types

## Don't fight the borrow checker

- ”Owned” types and ”view” types: `&str` and `String`, `Path` and `PathBuf`, etc.
- Don't hide ownership requirements: avoid hidden `.clone()`, learn to love `Cow`
- Split types along ownership boundaries
- Structure your ownership hierarchy like a tree
- Strategies to manage circular dependencies: reference counting, using indexes instead of references
- Interior mutability (Cell, RefCell)
- Working with lifetime parameters on user-defined data types

## Polymorphism in Rust

- A quick refresher on traits and generic functions
- Rust has no inheritance: what are the implications?
  - Using enums for polymorphism
  - Using traits for polymorphism
  - Using composition
  - How do I pick the most appropriate pattern?
- Working with generics
  - Generic type parameter in a function or trait object as an argument?
  - Trait bounds don't have to refer to the generic parameter
  - Type parameters in traits: should it be a generic parameter or an associated type?
- Macros: a valuable tool to DRY up code when traits are not enough (or too complex)

## Hata İşleme

- What is the purpose of errors? Recovery vs. reporting.
- Result vs. Option
- Designing good errors:
  - Determine the error scope.
  - Capture additional context as the error flows upwards, crossing scope boundaries.
  - Leverage the `Error` trait to keep track of the full error chain.
  - Leverage `thiserror` to reduce boilerplate when defining error types.
  - `anyhow`
- Distinguish fatal errors from recoverable errors using `Result<Result<T, RecoverableError>, FatalError>`.

# Chapter 69

# Leveraging the Type System

Rust's type system is *expressive*: you can use types and traits to build abstractions that make your code harder to misuse.

In some cases, you can go as far as enforcing correctness at *compile-time*, with no runtime overhead.

Types and traits can model concepts and constraints from your business domain. With careful design, you can improve the clarity and maintainability of the entire codebase.

This slide should take about 5 minutes.

Additional items speaker may mention:

- Rust's type system borrows a lot of ideas from functional programming languages.

  For example, Rust's enums are known as ”algebraic data types” in languages like Haskell and OCaml. You can take inspiration from learning material geared towards functional languages when looking for guidance on how to design with types. ”Domain Modeling Made Functional” is a great resource on the topic, with examples written in F#.

- Despite Rust's functional roots, not all functional design patterns can be easily translated to Rust.

  For example, you must have a solid grasp on a broad selection of advanced topics to design APIs that leverage higher-order functions and higher-kinded types in Rust.

  Evaluate, on a case-by-case basis, whether a more imperative approach may be easier to implement. Consider using in-place mutation, relying on Rust's borrow-checker and type system to control what can be mutated, and where.

- The same caution should be applied to object-oriented design patterns. Rust doesn't support inheritance, and object decomposition should take into account the constraints introduced by the borrow checker.

- Mention that type-level programming can be often used to create ”zero-cost abstractions”, although the label can be misleading: the impact on compile times and code complexity may be significant.

This segment should take about 25 minutes. It contains:

| Slayt | Süre |
|---|---|
| Leveraging the Type System | 5 dakika |
| Tür Durum Deseni | 20 dakika |

## 69.1 Tür Durum Deseni

A *newtype* is a wrapper around an existing type, often a primitive:

```
/// A unique user identifier, implemented as a newtype around `u64`.
pub struct UserId(u64);
```

Unlike type aliases, newtypes aren't interchangeable with the wrapped type:

```
fn double(n: u64) -> u64 {
    n * 2
}

double(UserId(1)); // 🛠️❌
```

The Rust compiler won't let you use methods or operators defined on the underlying type either:

```
assert_ne!(UserId(1), UserId(2)); // 🛠️❌
```

This slide and its sub-slides should take about 20 minutes.

- Students should have encountered the newtype pattern in the ”Fundamentals” course, when they learned about tuple structs.
- Run the example to show students the error message from the compiler.
- Modify the example to use a typealias instead of a newtype, such as `type MessageId = u64`. The modified example should compile, thus highlighting the differences between the two approaches.
- Stress that newtypes, out of the box, have no behaviour attached to them. You need to be intentional about which methods and operators you are willing to forward from the underlying type. In our `UserId` example, it is reasonable to allow comparisons between `UserId`s, but it wouldn't make sense to allow arithmetic operations like addition or subtraction.

### 69.1.1 Semantic Confusion

When a function takes multiple arguments of the same type, call sites are unclear:

```
pub fn login(username: &str, password: &str) -> Result<(), LoginError> {
    // [...]
}

// In another part of the codebase, we swap arguments by mistake.
// Bug (best case), security vulnerability (worst case)
login(password, username);
```

The newtype pattern can prevent this class of errors at compile time:

```
pub struct Username(String);
pub struct Password(String);

pub fn login(username: &Username, password: &Password) -> Result<(), LoginError> {
    // [...]
}

login(password, username); // 🛠️❌
```

- Run both examples to show students the successful compilation for the original example, and the compiler error returned by the modified example.
- Stress the *semantic* angle. The newtype pattern should be leveraged to use distinct types for distinct concepts, thus ruling out this class of errors entirely.
- Nonetheless, note that there are legitimate scenarios where a function may take multiple arguments of the same type. In those scenarios, if correctness is of paramount importance, consider using a struct with named fields as input:

  ```
  pub struct LoginArguments<'a> {
      pub username: &'a str,
      pub password: &'a str,
  }

  // No need to check the definition of the `login` function to spot the issue.
  login(LoginArguments {
      username: password,
      password: username,
  })
  ```

  Users are forced, at the callsite, to assign values to each field, thus increasing the likelihood of spotting bugs.

### 69.1.2 Parse, Don't Validate

The newtype pattern can be leveraged to enforce *invariants*.

```
pub struct Username(String);

impl Username {
    pub fn new(username: String) -> Result<Self, InvalidUsername> {
        if username.is_empty() {
            return Err(InvalidUsername::CannotBeEmpty)
        }
        if username.len() > 32 {
            return Err(InvalidUsername::TooLong { len: username.len() })
        }
        // Other validation checks...
        Ok(Self(username))
    }

    pub fn as_str(&self) -> &str {
        &self.0
```

```
    }
}
```

- The newtype pattern, combined with Rust's module and visibility system, can be used to *guarantee* that instances of a given type satisfy a set of invariants.

  In the example above, the raw `String` stored inside the `Username` struct can't be accessed directly from other modules or crates, since it's not marked as `pub` or `pub(in ...)`. Consumers of the `Username` type are forced to use the `new` method to create instances. In turn, `new` performs validation, thus ensuring that all instances of `Username` satisfy those checks.

- The `as_str` method allows consumers to access the raw string representation (e.g., to store it in a database). However, consumers can't modify the underlying value since `&str`, the returned type, restricts them to read-only access.

- Type-level invariants have second-order benefits.

  The input is validated once, at the boundary, and the rest of the program can rely on the invariants being upheld. We can avoid redundant validation and "defensive programming" checks throughout the program, reducing noise and improving performance.

### 69.1.3 Is It Truly Encapsulated?

You must evaluate *the entire API surface* exposed by a newtype to determine if invariants are indeed bullet-proof. It is crucial to consider all possible interactions, including trait implementations, that may allow users to bypass validation checks.

```rust
pub struct Username(String);

impl Username {
    pub fn new(username: String) -> Result<Self, InvalidUsername> {
        // Validation checks...
        Ok(Self(username))
    }
}

impl std::ops::DerefMut for Username { // !!
    fn deref_mut(&mut self) -> &mut Self::Target {
        &mut self.0
    }
}
```

- `DerefMut` allows users to get a mutable reference to the wrapped value.

  The mutable reference can be used to modify the underlying data in ways that may violate the invariants enforced by `Username::new`!

- When auditing the API surface of a newtype, you can narrow down the review scope to methods and traits that provide mutable access to the underlying data.

- Remind students of privacy boundaries.

  In particular, functions and methods defined in the same module of the newtype can access its underlying data directly. If possible, move the newtype definition to its own

separate module to reduce the scope of the audit.

# Part XVI

# Emniyetsiz

# Chapter 70

# Welcome to Unsafe Rust

> IMPORTANT: THIS MODULE IS IN AN EARLY STAGE OF DEVELOPMENT
>
> Please do not consider this module of Comprehensive Rust to be complete. With that in mind, your feedback, comments, and especially your concerns, are very welcome.
>
> To comment on this module's development, please use the GitHub issue tracker.

The `unsafe` keyword is easy to type, but hard to master. When used appropriately, it forms a useful and indeed essential part of the Rust programming language.

By the end of this deep dive, you'll know how to work with `unsafe` code, review others' changes that include the `unsafe` keyword, and produce your own.

What you'll learn:

- What the terms undefined behavior, soundness, and safety mean
- Why the `unsafe` keyword exists in the Rust language
- How to write your own code using `unsafe` safely
- How to review `unsafe` code

## Links to other sections of the course

The `unsafe` keyword has treatment in:

- *Rust Fundamentals*, the main module of Comprehensive Rust, includes a session on Unsafe Rust in its last day.
- *Rust in Chromium* discusses how to interoperate with C++. Consult that material if you are looking into FFI.
- *Bare Metal Rust* uses unsafe heavily to interact with the underlying host, among other things.

# Chapter 71

# Setting Up

## Local Rust installation

You should have a Rust compiler installed that supports the 2024 edition of the language, which is any version of rustc higher than 1.84.

```
$ rustc --version
rustc 1.87
```

## (Optional) Create a local instance of the course

```
$ git clone --depth=1 https://github.com/google/comprehensive-rust.git
Cloning into 'comprehensive-rust'...
...
$ cd comprehensive-rust
$ cargo install-tools
...
$ cargo serve # then open http://127.0.0.1:3000/ in a browser
```

This slide should take about 2 minutes.

Ask everyone to confirm that everyone is able to execute `rustc` with a version older that 1.87.

For those people who do not, tell them that we'll resolve that in the break.

# Chapter 72

# Motivasyon

We know that writing code without the guarantees that Rust provides ...

> “Use-after-free (UAF), integer overflows, and out of bounds (OOB) reads/writes comprise 90% of vulnerabilities with OOB being the most common.”
>
> —– **Jeff Vander Stoep and Chong Zang**, Google. ”Queue the Hardening Enhancements”

... so why is `unsafe` part of the language?

Bu bölüm yaklaşık 20 dakika sürmelidir. İçeriği:

| Slayt | Süre |
|---|---|
| Motivasyon | 1 minute |
| Birlikte Çalışabilirlik (Interoperability) | 5 dakika |
| Veri Yapılarında Ömürler | 5 dakika |
| Performance | 5 dakika |

This slide should take about 1 minute.

The `unsafe` keyword exists because there is no compiler technology available today that makes it obsolete. Compilers cannot verify everything.

> TODO: Refactor this content into multiple slides as this slide is intended as an introduction to the motivations only, rather than to be an elaborate discussion of the whole problem.

## 72.1 Birlikte Çalışabilirlik (Interoperability)

Language interoperability allows you to:

- Call functions written in other languages from Rust
- Write functions in Rust that are callable from other languages

However, this requires unsafe.

```rust
unsafe extern "C" {
    safe fn random() -> libc::c_long;
}

fn main() {
    let a = random() as i64;
    println!("{a:?}");
}
```

This slide should take about 5 minutes.

The Rust compiler can't enforce any safety guarantees for programs that it hasn't compiled, so it delegates that responsibility to you through the unsafe keyword.

The code example we're seeing shows how to call the random function provided by libc within Rust. libc is available to scripts in the Rust Playground.

This uses Rust's *foreign function interface*.

This isn't the only style of interoperability, however it is the method that's needed if you want to work between Rust and some other language in a zero cost way. Another important strategy is message passing.

Message passing avoids unsafe, but serialization, allocation, data transfer and parsing all take energy and time.

## Answers to questions

- *Where does "random" come from?*
  libc is dynamically linked to Rust programs by default, allowing our code to rely on its symbols, including `random`, being available to our program.
- *What is the "safe" keyword?*
  It allows callers to call the function without needing to wrap that call in `unsafe`. The `safe` function qualifier was introduced in the 2024 edition of Rust and can only be used within `extern` blocks. It was introduced because `unsafe` became a mandatory qualifier for `extern` blocks in that edition.
- *What is the `std::ffi::c_long` type?*
  According to the C standard, an integer that's at least 32 bits wide. On today's systems, It's an `i32` on Windows and an `i64` on Linux.

## Consideration: type safety

Modify the code example to remove the need for type casting later. Discuss the potential UB - long's width is defined by the target.

```rust
unsafe extern "C" {
    safe fn random() -> i64;
}

fn main() {
    let a = random();
    println!("{a:?}");
}
```

Changes from the original:

```
unsafe extern "C" {
-    safe fn random() -> libc::c_long;
+    safe fn random() -> i64;
}

fn main() {
-    let a = random() as i64;
+    let a = random();
     println!("{a:?}");
}
```

It's also possible to completely ignore the intended type and create undefined behavior in multiple ways. The code below produces output most of the time, but generally results in a stack overflow. It may also produce illegal `char` values. Although `char` is represented in 4 bytes (32 bits), not all bit patterns are permitted as a `char`.

Stress that the Rust compiler will trust that the wrapper is telling the truth.

```
unsafe extern "C" {
    safe fn random() -> [char; 2];
}

fn main() {
    let a = random();
    println!("{a:?}");
}
```

Changes from the original:

```
unsafe extern "C" {
-    safe fn random() -> libc::c_long;
+    safe fn random() -> [char; 2];
}

fn main() {
-    let a = random() as i64;
-    println!("{a}");
+    let a = random();
+    println!("{a:?}");
}
```

Attempting to print a `[char; 2]` from randomly generated input will often produce strange output, including:

```
thread 'main' panicked at library/std/src/io/stdio.rs:1165:9:
failed printing to stdout: Bad address (os error 14)
```

```
thread 'main' has overflowed its stack
fatal runtime error: stack overflow, aborting
```

Mention that type safety is generally not a large concern in practice. Tools that produce wrappers automatically, i.e. bindgen, are excellent at reading header files and producing values of the correct type.

## Consideration: Ownership and lifetime management

While libc's `random` function doesn't use pointers, many do. This creates many more possibilities for unsoundness.

- both sides might attempt to free the memory (double free)
- both sides can attempt to write to the data

For example, some C libraries expose functions that write to static buffers that are re-used between calls.

```
use std::ffi::{CStr, c_char};
use std::time::{SystemTime, UNIX_EPOCH};

unsafe extern "C" {
    /// Create a formatted time based on time `t`, including trailing newline.
    /// Read `man 3 ctime` details.
    fn ctime(t: *const libc::time_t) -> *const c_char;
}

unsafe fn format_timestamp<'a>(t: u64) -> &'a str {
    let t = t as libc::time_t;

    unsafe {
        let fmt_ptr = ctime(&t);
        CStr::from_ptr(fmt_ptr).to_str().unwrap()
    }
}

fn main() {
    let now = SystemTime::now().duration_since(UNIX_EPOCH).unwrap();

    let now = now.as_secs();
    let now_fmt = unsafe { format_timestamp(now) };
    print!("now (1): {}", now_fmt);

    let future = now + 60;
    let future_fmt = unsafe { format_timestamp(future) };
    print!("future:  {}", future_fmt);

    print!("now (2): {}", now_fmt);
}
```

*Aside:* Lifetimes in the `format_timestamp()` function

Neither `'a`, nor `'static`, correctly describe the lifetime of the string that's returned. Rust treats it as an immutable reference, but subsequent calls to `ctime` will overwrite the static buffer that the string occupies.

## Consideration: Representation mismatch

Different programming languages have made different design decisions and this can create impedance mismatches between different domains.

Consider string handling. C++ defines `std::string`, which has an incompatible memory layout with Rust's `String` type. `String` also requires text to be encoded as UTF-8, whereas `std::string` does not. In C, text is represented by a null-terminated sequence of bytes (`char*`).

```
fn main() {
    let c_repr = b"Hello, C\0";
    let rust_repr = (b"Hello, Rust", 11);

    let c: &str = unsafe {
        let ptr = c_repr.as_ptr() as *const i8;
        std::ffi::CStr::from_ptr(ptr).to_str().unwrap()
    };
    println!("{c}");

    let rust: &str = unsafe {
        let ptr = rust_repr.0.as_ptr();
        let bytes = std::slice::from_raw_parts(ptr, rust_repr.1);
        std::str::from_utf8_unchecked(bytes)
    };
    println!("{rust}");
}
```

## 72.2 Veri Yapılarında Ömürler

Some families of data structures are impossible to create in safe Rust.

- graphs
- bit twiddling
- self-referential types
- intrusive data structures

This slide should take about 5 minutes.

Graphs: General-purpose graphs cannot be created as they may need to represent cycles. Cycles are impossible for the type system to reason about.

Bit twiddling: Overloading bits with multiple meanings. Examples include using the NaN bits in `f64` for some other purpose or the higher-order bits of pointers on `x86_64` platforms. This is somewhat common when writing language interpreters to keep representations within the word size the target platform.

Self-referential types are too hard for the borrow checker to verify.

Intrusive data structures: store structural metadata (like pointers to other elements) inside the elements themselves, which requires careful handling of aliasing.

## 72.3 Performance

> TODO: Stub for now

It's easy to think of performance as the main reason for unsafe, but high performance code makes up the minority of unsafe blocks.

# Chapter 73

# Fonksiyonlar

Some fundamental concepts and terms.

This segment should take about 25 minutes. It contains:

| Slayt | Süre |
|---|---|
| Rust Nedir? | 10 dakika |
| When is unsafe used? | 2 dakika |
| Default, yapı güncelleme sözdizimi (struct update syntax) | 2 dakika |
| Actions might not be | 2 dakika |
| Less powerful than it seems | 10 dakika |

## 73.1 What is “unsafety”?

Unsafe Rust is a superset of Safe Rust.

Let's create a list of things that are enabled by the `unsafe` keyword.

This slide should take about 6 minutes.

### Definitions from authoritative docs:

From the unsafe keyword's documentation:

> Code or interfaces whose memory safety cannot be verified by the type system.
>
> ...
>
> Here are the abilities Unsafe Rust has in addition to Safe Rust:
>
> - Dereference raw pointers
> - Implement unsafe traits
> - Call unsafe functions
> - Mutate statics (including external ones)
> - Access fields of unions

From the reference

The following language level features cannot be used in the safe subset of Rust:

- Dereferencing a raw pointer.
- Reading or writing a mutable or external static variable.
- Accessing a field of a union, other than to assign to it.
- Calling an unsafe function (including an intrinsic or foreign function).
- Calling a safe function marked with a target_feature from a function that does not have a target_feature attribute enabling the same features (see attributes.codegen.target_feature.safety-restrictions).
- Implementing an unsafe trait.
- Declaring an extern block.
- Applying an unsafe attribute to an item.

## Group exercise

You may have a group of learners who are not familiar with each other yet. This is a way for you to gather some data about their confidence levels and the psychological safety that they're feeling.

#### Part 1: Informal definition

Use this to gauge the confidence level of the group. If they are uncertain, then tailor the next section to be more directed.

Ask the class: **By raising your hand, indicate if you would feel comfortable defining unsafe?**

If anyone's feeling confident, allow them to try to explain.

#### Part 2: Evidence gathering

Ask the class to spend 3-5 minutes.

- Find a use of the unsafe keyword. What contract/invariant/pre-condition is being established or satisfied?
- Write down terms that need to be defined (unsafe, memory safety, soundness, undefined behavior)

#### Part 3: Write a working definition

#### Part 4: Remarks

Mention that we'll be reviewing our definition at the end of the day.

## Note: Avoid detailed discussion about precise semantics of memory safety

It's possible that the group will slide into a discussion about the precise semantics of what memory safety actually is and how define pointer validity. This isn't a productive line of discussion. It can undermine confidence in less experienced learners.

Perhaps refer people who wish to discuss this to the discussion within the official documentation for pointer types (excerpt below) as a place for further research.

> Many functions in this module take raw pointers as arguments and read from or write to them. For this to be safe, these pointers must be *valid* for the given access.
>
> ...
>
> The precise rules for validity are not determined yet.

## 73.2 When is unsafe used?

The unsafe keyword indicates that the programmer is responsible for upholding Rust's safety guarantees.

The keyword has two roles:

- define pre-conditions that must be satisfied
- assert to the compiler (= promise) that those defined pre-conditions are satisfied

### Further references

- The unsafe keyword chapter of the Rust Reference

This slide should take about 2 minutes.

Places where pre-conditions can be defined (Role 1)

- unsafe functions (`unsafe fn foo() { ... }`). Example: `get_unchecked` method on slices, which requires callers to verify that the index is in-bounds.
- unsafe traits (`unsafe trait`). Examples: `Send` and `Sync` marker traits in the standard library.

Places where pre-conditions must be satisfied (Role 2)

- unsafe blocks (`unafe { ... }`)
- implementing unsafe traits (`unsafe impl`)
- access external items (`unsafe extern`)
- adding unsafe attributes o an item. Examples: `export_name`, `link_section` and `no_mangle`. Usage: `#[unsafe(no_mangle)]`

## 73.3 Data structures are safe ...

Data structures are inert. They cannot do any harm by themselves.

Safe Rust code can create raw pointers:

```
fn main() {
    let n: i64 = 12345;
    let safe = &raw const n;
    println!("{safe:p}");
}
```

This slide should take about 2 minutes.

Consider a raw pointer to an integer, i.e., the value `safe` is the raw pointer type `*const i64`. Raw pointers can be out-of-bounds, misaligned, or be null. But the unsafe keyword is not required when creating them.

## 73.4 … but actions on them might not be

```
fn main() {
    let n: i64 = 12345;
    let safe = &n as *const _;
    println!("{safe:p}");
}
```

This slide should take about 2 minutes.

Modify the example to de-reference `safe` without an `unsafe` block.

## 73.5 Less powerful than it seems

The `unsafe` keyword does not allow you to break Rust.

```
use std::mem::transmute;

let orig = b"RUST";
let n: i32 = unsafe { transmute(orig) };

println!("{n}")
```

This slide should take about 10 minutes.

### Suggested outline

- Request that someone explains what `std::mem::transmute` does
- Discuss why it doesn't compile
- Fix the code

### Expected compiler output

```
   Compiling playground v0.0.1 (/playground)
error[E0512]: cannot transmute between types of different sizes, or dependently-sized ty
 --> src/main.rs:5:27
  |
5 |     let n: i32 = unsafe { transmute(orig) };
  |                           ^^^^^^^^^
  |
  = note: source type: `&[u8; 4]` (64 bits)
  = note: target type: `i32` (32 bits)
```

### Suggested change

```
- let n: i32 = unsafe { transmute(orig) };
+ let n: i64 = unsafe { transmute(orig) };
```

### Notes on less familiar Rust

- the `b` prefix on a string literal marks it as byte slice (`&[u8]`) rather than a string slice (`&str`)

# Part XVII

# Son sözler

# Chapter 74

# Teşekkürler!

*Thank you for taking Comprehensive Rust 🦀!* We hope you enjoyed it and that it was useful.

We've had a lot of fun putting the course together. The course is not perfect, so if you spotted any mistakes or have ideas for improvements, please get in contact with us on GitHub. We would love to hear from you.

- Thank you for reading the speaker notes! We hope they have been useful. If you find pages without notes, please send us a PR and link it to issue #1083. We are also very grateful for fixes and improvements to the existing notes.

# Chapter 75

# Sözlük

The following is a glossary which aims to give a short definition of many Rust terms. For translations, this also serves to connect the term back to the English original.

h1#glossary ~ ul { list-style: none; padding-inline-start: 0; }

h1#glossary ~ ul > li { /* Simplify with "text-indent: 2em hanging" when supported: https://caniuse.com/mdn-css_properties_text-indent_hanging */ padding-left: 2em; text-indent: -2em; }

h1#glossary ~ ul > li:first-line { font-weight: bold; }

- allocate:
  Dynamic memory allocation on the heap.
- argument:
  Information that is passed into a function or method.
- associated type:
  A type associated with a specific trait. Useful for defining the relationship between types.
- Bare-metal Rust:
  Low-level Rust development, often deployed to a system without an operating system. See Bare-metal Rust.
- block:
  See Blocks and *scope*.
- borrow:
  See Borrowing.
- borrow checker:
  The part of the Rust compiler which checks that all borrows are valid.
- brace:
  { and }. Also called *curly brace*, they delimit *blocks*.
- build:
  The process of converting source code into executable code or a usable program.
- call:
  To invoke or execute a function or method.
- channel:
  Used to safely pass messages between threads.
- Comprehensive Rust 🦀:
  The courses here are jointly called Comprehensive Rust 🦀.

- concurrency:
  The execution of multiple tasks or processes at the same time.
- Concurrency in Rust:
  See Concurrency in Rust.
- constant:
  A value that does not change during the execution of a program.
- control flow:
  The order in which the individual statements or instructions are executed in a program.
- crash:
  An unexpected and unhandled failure or termination of a program.
- enumeration:
  A data type that holds one of several named constants, possibly with an associated tuple or struct.
- error:
  An unexpected condition or result that deviates from the expected behavior.
- error handling:
  The process of managing and responding to errors that occur during program execution.
- exercise:
  A task or problem designed to practice and test programming skills.
- function:
  A reusable block of code that performs a specific task.
- garbage collector:
  A mechanism that automatically frees up memory occupied by objects that are no longer in use.
- generics:
  A feature that allows writing code with placeholders for types, enabling code reuse with different data types.
- immutable:
  Unable to be changed after creation.
- integration test:
  A type of test that verifies the interactions between different parts or components of a system.
- keyword:
  A reserved word in a programming language that has a specific meaning and cannot be used as an identifier.
- library:
  A collection of precompiled routines or code that can be used by programs.
- macro:
  Rust macros can be recognized by a `!` in the name. Macros are used when normal functions are not enough. A typical example is `format!`, which takes a variable number of arguments, which isn't supported by Rust functions.
- `main` function:
  Rust programs start executing with the `main` function.
- match:
  A control flow construct in Rust that allows for pattern matching on the value of an expression.
- memory leak:
  A situation where a program fails to release memory that is no longer needed, leading to a gradual increase in memory usage.
- method:
  A function associated with an object or a type in Rust.

- module:
  A namespace that contains definitions, such as functions, types, or traits, to organize code in Rust.
- move:
  The transfer of ownership of a value from one variable to another in Rust.
- mutable:
  A property in Rust that allows variables to be modified after they have been declared.
- ownership:
  The concept in Rust that defines which part of the code is responsible for managing the memory associated with a value.
- panic:
  An unrecoverable error condition in Rust that results in the termination of the program.
- parameter:
  A value that is passed into a function or method when it is called.
- pattern:
  A combination of values, literals, or structures that can be matched against an expression in Rust.
- payload:
  The data or information carried by a message, event, or data structure.
- program:
  A set of instructions that a computer can execute to perform a specific task or solve a particular problem.
- programming language:
  A formal system used to communicate instructions to a computer, such as Rust.
- receiver:
  The first parameter in a Rust method that represents the instance on which the method is called.
- reference counting:
  A memory management technique in which the number of references to an object is tracked, and the object is deallocated when the count reaches zero.
- return:
  A keyword in Rust used to indicate the value to be returned from a function.
- Rust:
  A systems programming language that focuses on safety, performance, and concurrency.
- Rust Fundamentals:
  Days 1 to 4 of this course.
- Rust in Android:
  See Rust in Android.
- Rust in Chromium:
  See Rust in Chromium.
- safe:
  Refers to code that adheres to Rust's ownership and borrowing rules, preventing memory-related errors.
- scope:
  The region of a program where a variable is valid and can be used.
- standard library:
  A collection of modules providing essential functionality in Rust.
- static:
  A keyword in Rust used to define static variables or items with a `'static` lifetime.
- string:
  A data type storing textual data. See Strings for more.

- struct:
  A composite data type in Rust that groups together variables of different types under a single name.
- test:
  A Rust module containing functions that test the correctness of other functions.
- thread:
  A separate sequence of execution in a program, allowing concurrent execution.
- thread safety:
  The property of a program that ensures correct behavior in a multithreaded environment.
- trait:
  A collection of methods defined for an unknown type, providing a way to achieve polymorphism in Rust.
- trait bound:
  An abstraction where you can require types to implement some traits of your interest.
- tuple:
  A composite data type that contains variables of different types. Tuple fields have no names, and are accessed by their ordinal numbers.
- type:
  A classification that specifies which operations can be performed on values of a particular kind in Rust.
- type inference:
  The ability of the Rust compiler to deduce the type of a variable or expression.
- undefined behavior:
  Actions or conditions in Rust that have no specified result, often leading to unpredictable program behavior.
- union:
  A data type that can hold values of different types but only one at a time.
- unit test:
  Rust comes with built-in support for running small unit tests and larger integration tests. See Unit Tests.
- unit type:
  Type that holds no data, written as a tuple with no members.
- unsafe:
  The subset of Rust which allows you to trigger *undefined behavior*. See Unsafe Rust.
- variable:
  A memory location storing data. Variables are valid in a *scope*.

# Chapter 76

# Other Rust Resources

The Rust community has created a wealth of high-quality and free resources online.

## Official Documentation

The Rust project hosts many resources. These cover Rust in general:

- The Rust Programming Language: the canonical free book about Rust. Covers the language in detail and includes a few projects for people to build.
- Rust By Example: covers the Rust syntax via a series of examples which showcase different constructs. Sometimes includes small exercises where you are asked to expand on the code in the examples.
- Rust Standard Library: full documentation of the standard library for Rust.
- The Rust Reference: an incomplete book which describes the Rust grammar and memory model.
- Rust API Guidelines: recommendations on how to design APIs.

More specialized guides hosted on the official Rust site:

- The Rustonomicon: covers unsafe Rust, including working with raw pointers and interfacing with other languages (FFI).
- Asynchronous Programming in Rust: covers the new asynchronous programming model which was introduced after the Rust Book was written.
- The Embedded Rust Book: an introduction to using Rust on embedded devices without an operating system.

## Unofficial Learning Material

A small selection of other guides and tutorial for Rust:

- Learn Rust the Dangerous Way: covers Rust from the perspective of low-level C programmers.
- Rust for Embedded C Programmers: covers Rust from the perspective of developers who write firmware in C.
- Rust for professionals: covers the syntax of Rust using side-by-side comparisons with other languages such as C, C++, Java, JavaScript, and Python.

- Rust on Exercism: 100+ exercises to help you learn Rust.
- Ferrous Teaching Material: a series of small presentations covering both basic and advanced part of the Rust language. Other topics such as WebAssembly, and async/await are also covered.
- Advanced testing for Rust applications: a self-paced workshop that goes beyond Rust's built-in testing framework. It covers `googletest`, snapshot testing, mocking as well as how to write your own custom test harness.
- Beginner's Series to Rust and Take your first steps with Rust: two Rust guides aimed at new developers. The first is a set of 35 videos and the second is a set of 11 modules which covers Rust syntax and basic constructs.
- Learn Rust With Entirely Too Many Linked Lists: in-depth exploration of Rust's memory management rules, through implementing a few different types of list structures.

Please see the Little Book of Rust Books for even more Rust books.

# Chapter 77

# Emekler

The material here builds on top of the many great sources of Rust documentation. See the page on other resources for a full list of useful resources.

The material of Comprehensive Rust is licensed under the terms of the Apache 2.0 license, please see `LICENSE` for details.

## Rust by Example

Some examples and exercises have been copied and adapted from Rust by Example. Please see the `third_party/rust-by-example/` directory for details, including the license terms.

## Rust on Exercism

Some exercises have been copied and adapted from Rust on Exercism. Please see the `third_party/rust-on-exercism/` directory for details, including the license terms.

## CXX

The Interoperability with C++ section uses an image from CXX. Please see the `third_party/cxx/` directory for details, including the license terms.