RUST
Hata Denetimi
panic!
Result<T,E> , ?
unwrap()
expect()

Hata Türleri

Rust'ta hatalar iki ana kategoriye ayrılır;

- Kurtarılabilir hatalar — Result<T,E> enum türü

- Kurtarılamaz hatalar — panic! macrosu

Kurtarılabilir Hata → Örneğin dosya bulunamadı (file not found) hatası. Bu durumda kullanıcıya durum bildirilebilir ve yeniden denemesi istenebilir.

Kurtarılamaz Hata → Programlarda oluşan buglara benzerler. Örneğin bir dizinin olmayan elemanına ulaşmak gibi. Böyle durumlarda programı hemen durdurmak isteyebiliriz.

Çoğu programlama dili hataları "exceptions - istisnalar" adı altında tek bir yapı altında ele alır.
Rust'ta exceptions yoktur.

Kurtarılamaz Hatalar - panic! Makrosu

Bu hatalar iki şekilde oluşur:

- İşletilen bir kod bu duruma sebep olabilir.
- Programcı tarafından panic! makrosu çağrılabilir.

Varsayılan ayarlarında panic! makrosu;

Program Çözülür (Unwinding)

- stack'ı temizler
- hata mesajı yazdırır
- programdan çıkar

Unwinding :

Unwinding → Rust, yığının yukarısına doğru yürür ve karşılaştığı her işlevdeki verileri temizler.

Bu işlem çok fazla iş gerektirir. Bu işlemi iptal ettirebiliriz. Bu programın binary boyutunu da küçültür. Bunu iptal etmek için cargo.toml dosyasındaki uygun [profile] bölümlerine aşağıdaki kod eklenir.

```
[profile.release]
panic = 'abort'
```

Bu durumda hafızayı işletim sisteminin temizlemesi gerekecektir.

panic! Makrosunun Çağrılması:

```rust
fn main() {
    println!("Hello, world!");
    let hata = true;
    if hata == true {
        panic!("Program durur ve hata mesajı olarak bu gelir.");
    }
    println!("Bu satır çalışmaz!");
}
```

```
 Running `target/debug/hata-denetimi`
Hello, world!
thread 'main' panicked at src/main.rs:5:9:
Program durur ve hata mesajı olarak bu gelir.
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

TERMINAL ÇIKTISI

panic! Hatası Hangi Durumlarda Oluşur ?

```
fn main() {
    let v = vec![1, 2, 3];

    v[99];
}
```

```
 Running `target/debug/hata-denetimi`
thread 'main' panicked at src/main.rs:4:6:
index out of bounds: the len is 3 but the index is 99
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Detaylı Hata İncelemesi:

RUST_BACKTRACE=1 cargo run -> Hata daha detaylı incelenir.

RUST_BACKTRACE=full cargo run → Hata ile ilgili bütün kaynaklar gösterilir.

Hata Denetimi İçin Kullanılan Yapılar

std::error → Hata modülü

std::error::Error → Hata traiti

Standart kütüphanesindeki her modülün kendine ait hata türleri vardır.

std::io::Error → Struct türünde
std::io::ErrorKİnd → Enum türünde

https://doc.rust-lang.org/stable/std/index.html?search=error

Kurtarılabilir Hatalar - Result<T,E> Enum Tipi

```
enum Result<T, E> {
      Ok(T),
      Err(E),
}
```

Ok(T) -> Hatanın oluşmadığı durumu ifade eder. T veriyi tutan değişkendir.

Err(E) -> Hatanın oluştuğunu ifade eder. E hata mesajını tutan değişkendir.

Yapılan işlemlerde geri döndürülen veri tip olarak kullanılır ve Match ile sonuçlar işlenir.
İşlem başarılı ise Ok(T) variantı, başarısız ile Err(E) variantı geri döndürülür.

```rust
use std::fs::File;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Hata Türüne Göre İşlem Yapmak

```
. . .
use std::io::ErrorKind;

fn main() {
        . . .
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => fc,
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error);
            }
        },
    };
}
```

Match ve Result<T,E> Yerine Closures Kullanılabilir

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

fn main() {
    let greeting_file = File::open("hello.txt").unwrap_or_else(|error| {
        if error.kind() == ErrorKind::NotFound {
            File::create("hello.txt").unwrap_or_else(|error| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

unwrap & expect → Hata Denetiminin Kısayolu

```rust
use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

```rust
use std::fs::File;

fn main() {
    let greeting_file =
        File::open("hello.txt").expect("hello.txt should be included in this
project");
}
```

Propagating Errors (Hataları Yaymak)

Bir fonksiyon başarısız olabilecek bir işlem çağırdığında, hatanın fonksiyon içinde ele almak yerine, hatayı fonksiyonu çağrıldığı kod bölümüne geri döndürebiliriz. Bu propagating errors olarak isimlendirilmektedir.

Bunun için fonksiyonun aşağıdaki gibi tanımlanması gerekir. Bu durumda dönen değer ya beklenen değer olacaktır ya da hata mesajı olacaktır.

```
fn fonksiyon_adı() -> Result<dönecek_değerin_tip, hata_tipi> { . . .
```

```rust
use std::fs::File;
use std::io::{self, Read};
fn main() {
    let y = read_username_from_file();
    println!("Fonksiyondan dönen -> {:?}", y);
    println!("Program çalışmaya devam etti.!");
}

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

Dosya olmadığı için hata mesajı gözükecektir.

io::Error, File::open() ve read_to_string() metodlarının dönüş tipine ait hata türüdür.

"?" → Propagating Errors Kısa Kullanımı

```rust
#![allow(unused)]
use std::fs::File;
use std::io::{self, Read};
fn main() {
    let y = read_username_from_file();
    println!("Fonksiyondan dönen -> {:?}", y);
    println!("Program çalışmaya devam etti.!");
}
fn read_username_from_file() -> Result<String, io::Error> {
    let mut username_file = File::open("hello.txt")?;
    let mut username = String::new();
    username_file.read_to_string(&mut username)?;
    Ok(username)
}
```

"?" operatörü aşağıdaki gibi de kullanılıbilir.

```rust
    File::open("hello.txt")?.read_to_string(&mut username)?;
```

```rust
fs::read_to_string("hello.txt")
```

Dosyadan veri okuma işlemi en kısa bu şekilde de kullanılabilir.

Fonksiyondan gelen veriyi aşağıdaki gibi işleyebiliriz.

```
fn main() {
. . .
   let kullanici_isim = match y {
        Ok(isim) => {
            if isim == "" {
                "varsayılanad".to_string()
            } else {
                isim
            }
        }
        Err(_) => "varsayılanad".to_string(),
    };
    println!("Kullanıcı adı => {}", kullanici_isim);
```

"?" Operatörü Nerede Kullanılabilir?

Geri dönüş tipi ? değeri ile uyumlu olan fonksiyonlarda kullanılabilir. Yani geri dönüş tipi Result, Option, FromResidual olmalıdır.

Main() fonksiyonunda kullanmak için aşağıdaki gibi bir geri dönüş tipi verilmelidir.

```
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
    let greeting_file = File::open("hello.txt")?;

    Ok(())
}
```

Box<dyn Error> → Bütün hata türlerini kapsar. std::error::Error → ile içe aktarılmıştır.
Box kullanılmasının sebebi, geri dönebilecek hata türlerinin hepsinin boyutlarının aynı olmamasından kaynaklanmaktadır. (trait videosunda anlatılmıştı.)

main() fonksiyonu std::process::Termination traitini uygulayan herhangi bir türü döndürebilir.

Result içinde döndürülen hata türü fonksiyonun yaptığı işleme göre değişebilir. Error hata türü std::error::Error kütüphanesinden gelen ve bütün hata türlerini kapsayan bir yapıdır.

https://doc.rust-lang.org/std/process/trait.Termination.html

"?" Option İle Kullanımı

Kendisine gönderilen &str nin son karakterini verir.

```
fn last_char_of_first_line(text: &str) -> Option<char> {
    text.lines().next()?.chars().last()
}
```

panic! Ve Result Ne Zaman Kullanılmalı

Result → Geri dönen değerler olduğunda (fonksiyonlarda) kullanılabilir. Yada kod içinde olması muhtemel hatalar varsa kullanışlı olacaktır.

panic! → Örneklerde, prototip kodlarda, testlerde kullanılabilir. unwrap ve expect ile hızlı bir kullanım yapabilirsiniz.

Kodun kötü durma düşme olasılığına karşı panic kullanılabilir. (geçersiz - çelişkili - eksik değer olmasını önlemek için, örneğin garanti, sözleşme, değişmez ihlali olmaması için)

Kodun kesin sonuç üretmesini istediğiniz durumlarda kulanmalısınız. Böylece kodun devamında yanlış bir iş yapması veya yanlış bir değer üretmesi engellenmiş olur.

Doğrulama İçin Özel Veri Tiplerinin Oluşturulması

```
pub struct Guess {
    value: i32,
}

impl Guess {
    pub fn new(value: i32) -> Guess {
        if value < 1 || value > 100 {
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess { value }
    }

    pub fn value(&self) -> i32 {
        self.value
    }
}
```

Burada veri 0-100 arasında sınırlandırılmıştır.

Yine de ;

let g1 = Guess { value:150 } çalışacak ve kısıtlamaya uymayacaktır.

Fakat yapı "mod" içinde tanımlanırsa value değeri pub ile tanımlanmadığından özel alan olacak ve erişilmeyecektir. Dolayısıyla pub ile tanımlanan new() metodunun kullanılması gerekecektir.

```
mod guess {
    pub struct Guess {
        value: i32,
    }
    impl Guess {
        pub fn new(value: i32) -> Guess {
            if value < 1 || value > 100 {
                panic!("Guess value must be between 1 and 100, got {}.", value);
            }
            Guess { value }
        }
        pub fn value(&self) -> i32 {
            self.value
        }
    }
}
```

pub → Koda modül (mod) dışından erişim sağlar

Özel Hata Tipi Tanımlama

```
struct OzelHata;
```

Hata mesajı için Display implementasyonu

```
impl fmt::Display for OzelHata {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "invalid first item to double")
    }
}
```

```rust
fn fonksiyon(x: i32) -> Result<i32, OzelHata> {
    if x > 50 {
        Err(OzelHata)
    } else {
        Ok(x)
    }
}

fn main() {

    match fonksiyon(222) {
        Ok(x) => println!("{}", x),
        Err(e) => print!("{}", e),
    }
}
```

Enum Tipi İle Çoklu Hatalar Tanımlayabiliriz.

```
enum OzelHata {
    Hata1,
    Hata2,
}

impl fmt::Display for OzelHata {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            OzelHata::Hata1 => write!(f, "Hata 1"),
            OzelHata::Hata2 => write!(f, "Hata 2"),
        }
    }
}
```

Standart kütüphanesindeki error traitini de özel hatamız için implement edebiliriz.

```
impl std::error::Error for OzelHata {}
```

https://doc.rust-lang.org/book/ch09-00-error-handling.html

https://web.mit.edu/rust-lang_v1.25/arch/amd64_ubuntu1404/share/doc/rust/html/book/first-edition/error-handling.html

Celal AKSU
Bilişim Teknolojileri Öğretmeni

celalaksu@gmail.com

https://www.linkedin.com/in/cllaksu/

https://twitter.com/ksacll

https://www.youtube.com/@eemcs