RUST

Enumarations

enum

Option<T>

match

if let

enum Nedir?

Enumeration, numaralandırma ya da sıralama olarak tercüme edebiliriz

Enum'u iki amaçla kullanabiliriz:

1 - Bir değişkenin alabileceği değerin farklı çeşitlerini (variant) tanımlamak için kullanılır. (Not : "Verinin Variant'ı" ile "Veri Tipi" aynı şey değildir.)

Örneğin ip adresini düşünün. İp adresi iki çeşittir (v4 ve v6). Burada ;

→ hem ip adresi çeşidi vardır

→ hem de ip adresinin kendisi veri olarak vardır.

Değişkenin değeri, yani veri belirlenen veri çeşitlerinden sadece bir tanesini olabilir.

enum Nedir?

2 - Ya da bir değişkenin alabileceği değerler önceden belirlenebilir. Burada oluşturulan değerlere de veri tipinin çeşidi (variant) olarak isimlendirilir.

Boolean veri tipi için olabilecek değerler true ve false tur. Buna benzer kullanacağınız değişkenin alabileceği değerleri belli ise bu değerleri enum türünde tanımlayıp kullanabilirsiniz.

Örneğin cinsiyet için kadın ve erkek diye iki tür değer vardır ve enum tanımlanarak bu değerler önceden belirtilebilir.

Enum Tanımlama :

```
enum IpAddrKind {
        V4,
        V6,
}
```

enum isminin başharfinin büyük olmasına dikkat ediniz

Burada ip address türleri enum olarak tanımlanmış

IpAddrKind adında yeni bir veri tipi oluşturulmuş
İp addres türünü tutan değişkene alanlardan bir tanesi atanacaktır

V4 ve V6 nın değişken değeri yani value olarak kullanılacağını unutmayınız.
IpAddrKind tipinden oluşturulacak değişkene başka bir değer atanamaz

Enum Örneği (instance) Oluşturma :

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;

println!("IP nin türü : {:?}", four);
```

Veri türünü yazdıktan sonra :: notasyonu kullanılarak enum alanı seçilir.

Fonksiyonlarda Enum Kullanımı

```
fn route(ip_kind: IpAddrKind) {

        println!("Adresin türü : {:?}",ip_kind);

}
```

```
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}
```

```
let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};
println!("Adresin türü : {:?}",home.kind); // Debug traidini derive etmeyi unutmayın.


let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

Enumlarda Verinin ve Veri Çeşidinin Birlikte Tanımlanması

Enum yapısı ile bir verinin türünü belirterek, bu türün içerisine veriler de ekleyebiliriz.

```
enum IpAddr {
    V4(String),
    V6(String),
}
```

(String) şeklinde tanımlana yere veri eklenir.
Burada belirtilen veri sayısı farklı olabilir. Örnek : V4(String, u32, ….) gibi
Her bir veri için veri tip belirtilmelidir.

```
let home = IpAddr::V4(String::from("127.0.0.1"));

let loopback = IpAddr::V6(String::from("::1"));
```

Veriye Erişim : match

```
    match home {
        IpAddrKind::V4(veri) => println!("ip adresi {}", veri),
        _ => (),
    }
```

Veriyi değişkene de atayabiliriz.

```
let falan = match home {
        IpAddrKind::V4(veri) => veri,
        _ => "none".to_string(),
    };
```

Veriye Erişim : Birden Fazla Değer Varsa

IpAddrKind::V4(veri1, veri2,…..) şeklinde belirtilir.

IpAddrKind::V4(_, _, veri3) almak istemediğiniz verinin yerine "_" alt tire ekleyebilirsiniz.

Veriye Erişim : if let

Veriye erişmek için "match" ya da "if let" yapılarından birini kullanabiliriz.

```
if let IpAddrKind::V4(veri) = home {

        println!("ip adresi {}", veri)

}
```

Veriyi değişkene de atayabiliriz.

```
let veri = if let IpAddrKind::V4(alınan_veri) = home { alınan_veri } else { 0 }
```

Enum ve Fonksiyon Bağlantısı

- IpAddr::V4() kullanımı fonksiyon çağırma işlemi yapar.
- () içinde yazılan veri türlerini fonksiyon parametreleridir
- () Buraya yazılan string veri IpAddr::V4() fonksiyonu tarafından değişkene geri gönderilmiş olur.

let address = IpAddr::V4("127.0.01") şeklinde tanımlama yaptığımızda "127.0.0.1" verisi addres değişkenine döndürülmüş (return) olur.

Enum ve Struct

```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

Enum çeşidinin (variant) içine herhangi bir veri türü eklenebilir :

- Struct
- Enum
- String, numerics
- .....

Enum ve Struct

```
struct QuitMessage; // unit struct
struct MoveMessage { x: i32, y: i32, }
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

Yukarıdaki gibi oluşturduğumu birden fazla struct veri yapısını;

aşağıdaki gibi tek bir enum içerisinde toplayabiliriz.

```
enum Message {
      Quit,
      Move { x: i32, y: i32 },
      Write(String),
      ChangeColor(i32, i32, i32),
}
```

- Farkı struct tanımlamak farklı veri tip tanımlamaktır
- Struct yapısında fonksiyon tanımlamak daha zor olur
- Bu yüzden structları enum içerisinde birleştirebiliriz.

Enum ve Metodlar

Structlarda olduğu gibi enumlar içinde metodlar oluşturabiliriz.

```
impl Message {
    fn call(&self) {
        match self {
            Self::WriteMessage(veri) => {
                println!("Gelen mesaj : {}", veri);
            }
            Self::QuitMessage => (),
            _ => (),
        }
    }
}
```

```
let m = Message::Write(String::from("hello"));
m.call();
```

Option Enum Tipi

Standard kütüphanede tanımlıdır
Bir değerin var olduğunu ya da boş olduğunu kontrol etmek için kullanılır.

Veri üreten bir yapı, herhangi bir değer oluşturabilir ya da boş bir değer oluşturabilir. Eğer yapı boş bir değer üretirse programda bug-hata oluşabilir. Boş değer üretilmesi ihtimaline karşı hata oluşmasını engellemek için Option enum tipi kullanılır.

Option<T> Enum Tipi

Option enum tipi Rust standard kütüphanesinde aşağıdaki gibi tanımlanmıştır.

```
enum Option<T> {
        None,
        Some(T),
}
```

<T > → Herhangi bir veri tipini temsil eder.
None → Değer boş geldiğinde döndürülen option çeşididir.
Some(T) → Değer var olduğunda döndürülen option çeşididir. T dönen değeri temsil eder.

Kullanımı :

Option:: ön tanımlanmasını kullanmak zorunlu değildir
Some<T> ve None tek başlarına kullanılabilir

```
let some_number = Some(5);
let some_char = Some('e');
let absent_number: Option<i32> = None;
```

Option tipinde olan bir değişkenin bir değerinin var olup olmadığı belli değildir. Dolayısıyla bu değişkenin bir değerinin var olduğunu kesinleştirmek için işlenmesi gerekir.

Option üzerinde işlemler

Option tipinde olan bir değişkenin bir değerinin var olup olmadığı belli değildir. Dolayısıyla bu değişkenin bir değerinin var olduğunu kesinleştirmek için işlenmesi gerekir.

```
let x: i8 = 5;
let y: Option<i8> = Some(5);
let sum = x + y; // HATA VERİR
```

```
let sum :i8;
match y {
        Some(z) => sum = x + z,
        None => sum = 0
    }
```

Option<T> ve Match

```
fn bir_ekle(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
```

```
    let bes = Some(5);
    let alti = bir_ekle(bes);
    println!("{:?}", alti);
    let none = bir_ekle(None);
    println!("{:?}", none);
```

if let, option<T> ve match

```
    let olası_sayı = Some(456);

    match olası_sayı {
        Some(sayı) => println!("Var olan sayı : {}", sayı),
        _ => (),
    }
```

Bu gibi durumları aşağıdaki gibi daha etkin bir şekilde yazabiliriz.

```
    if let Some(sayı) = olası_sayı {
        println!("Var olan sayı : {}", sayı)
    }
```

Değerin olmadığı durumları else bloğu ile de kontrol edebiliriz.

```
    let olmayan_veri: Option<i32> = None;

    match olmayan_veri {
        Some(sayı) => println!("Var olan sayı : {}", sayı),
        _ => println!("Ne yazıkki veri gelmedi ve değer
None'dur"),
    }
```

```
    if let Some(sayı) = olmayan_veri {
        println!("Var olan sayı : {}", sayı)
    } else {
        println!("Ne yazıkki veri gelmedi ve değer None'dur");
    }
```

Enum option metotları ve diğer kullanımları için aşağıdaki dökümantasyonu kullanabiliriz.

https://doc.rust-lang.org/std/option/enum.Option.html

Enum ve Match

Örnekte madeni para çeşitleri enum ile tanımlanmış ve fonksiyonda her madeni paranın değeri geri döndürülmektedir.

```
enum MadenPara {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn madenpara_degeri(madenpara: MadenPara) -> u16 {
    match madenpara {
        MadenPara::Penny => 1,
        MadenPara::Nickel => 5,
        MadenPara::Dime => 10,
        MadenPara::Quarter => 25,
    }
}
```

Enum ve Match (içe aktarma)

Örnekte madeni para çeşitleri enum ile tanımlanmış ve fonksiyonda her madeni paranın değeri geri döndürülmektedir.

```
enum MadenPara {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn madenpara_degeri(madenpara: MadenPara) -> u16 {
    use MadenPara::*;
    match madenpara {
        Penny => 1,
        Nickel => 5,
        Dime => 10,
        Quarter => 25,
    }
}
```

Match → değişkenin bütün durumlarını kontrol etmek ister. Eğer bütün değerler kontrol edilmeyecekse , "_ =>" yapısı kullanılır.

```
fn madenpara_degeri(madenpara: MadenPara) -> u16 {
    match madenpara {
        MadenPara::Penny => 1,
        _ => 0,
    }
}
```

"_ =>" 0 → Fonksiyonda u16 türünde döndürülen değer gerektiği için yazıldı.

" _=> () " → Şeklinde de kullanılır

Enum variantının içine başka bir enumu u değer olarak verebiliriz.

```
#[derive(Debug)]
enum MadenYeri {
    Zonguldak,
    Artvin,
    Edirne,
      …..}

enum MadenPara {
    Penny,
    Nickel,
    Dime,
    Quarter(MadenYeri),
}
```

madenin çıkarıldığı bölgeyide belirtebiliriz.

```rust
fn madenpara_degeri(madenpara: MadenPara) -> u16 {
    match madenpara {
        MadenPara::Penny => 1,
        MadenPara::Nickel => 5,
        MadenPara::Dime => 10,
        MadenPara::Quarter(bolge) => {
            println!("Çıkartıldığı bölge : {:?}", bolge);
            25
        }
    }
}

fn main() {
    madenpara_degeri(MadenPara::Quarter(MadenYeri::Zonguldak));
}
```

Celal AKSU
Bilişim Teknolojileri Öğretmeni

celalaksu@gmail.com

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

https://twitter.com/ksacll

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