RUST
Traits - Ortak Davranış Tanımlama
trait

trait Nedir?

- Trait → ortak davranış anlamına gelir
- Standart kütüphanede bulunan ve bazı veri tiplerinin sahip olduğu bir davranışı (behavior) farklı veri tiplerine aktarmamızı sağlar. (PartialEq ...)
- Farklı veri tiplerinin kullanabileceği davranışlar-özellikler oluşturmak için kullanılır.
- Paylaşılacak olan davranış, trait ile soyut bir şekilde tanımlanır.
- Sonrasında bu davranışlar veri tipleri için implement edilir.

trait Tanımlama

Bir veri tipi için kullanılacak olan davranış aslında metodlardan oluşur.
Trait tanımlanmasında, metodların imzaları yer alır.

```
pub struct NewsArticle {
        pub headline: String,
        pub location: String,
        pub author: String,
        pub content: String,
}
```

```
pub struct Tweet {
        pub username: String,
        pub content: String,
        pub reply: bool,
        pub retweet: bool,
}
```

Bu iki tür içinde geçerli olacak ve bunlarda tutulan verinin özetini verecek bir özellik tanımlayalım.

Bunun için önce trait tanımalaması yapmamız gerekir;

```
pub trait Summary {

}
```

Sonrasında bu türlere uygulamak istediğimiz özellikler metod imzası olarak tanımlanır

```
pub trait Summary {
    fn summarize(&self) -> String;
}
```

METOD İMZASI : Sadece fonksiyonun tanımlanmasıdır. Fonksiyon için kodlar yazılmaz.

Sonraki adımda da trait türler için implement edilir

```
impl Summary for NewsArticle {

}

impl Summary for Tweet {

}
```

Ve son olarak ta trait içinde oluşturulan metod, türlerin implementasyonunda kendilerine özgü kodlarla oluşturulur.

```
impl Summary for NewsArticle {

      fn summarize(&self) -> String {
            format!("{}, by {} ({})", self.headline,
self.author, self.location)
      }
}

impl Summary for Tweet {

      fn summarize(&self) -> String {
            format!("{}: {}", self.username, self.content)
      }
}
```

Ve artık summarize() özelliğini iki tür içinde kullanabiliriz.

```
fn main() {
      let tweet = Tweet {
            username: String::from("horse_ebooks"),
            content: String::from( "of course, as you
probably already know, people", ),
            reply: false,
            retweet: false, };
      println!("1 new tweet: {}", tweet.summarize()); }
```

Varsayılan Trait Tanımlanması

Trait tanımlaması yapılırken; metodun imzasının yazılması yanında, metodun kodları da eklenebilir.
Bu durumda bu özellik implement edilen bütün türlerde geçerli olur

```
pub trait Summary {
      fn summarize_default(&self) -> String {
            String::from("(Read more...)")
      }
}
```

Sadece varsayılan özellği implement etmek için boş bir yapı kullanılır

```
impl Summary for NewsArticle {
}
```

```
let article = NewsArticle {
        headline: String::from("Penguins win the Stanley Cup Championship!"),
        location: String::from("Pittsburgh, PA, USA"),
        author: String::from("Iceburgh"),
        content: String::from( "The Pittsburgh Penguins once again are the best
\ hockey team in the NHL.", ),
};

println!("New article available! {}", article.summarize_default());
```

Varsayılan tanımlamalar aynı trait içindeki diğer metodları da çağırabilir.

```
pub trait Summary {

        fn summarize_author(&self) -> String;

        fn summarize_default(&self) -> String {
                format!("(Read more from {}...)", self.summarize_author())
        }
}
```

Bu durumda türler için sadece imzası yazılan metodu doldurmamız yeterlidir.

```
impl Summary for Tweet {
        fn summarize_author(&self) -> String {
                format!("@{}", self.username)
        }
}
```

```
let tweet = Tweet {
      username: String::from("horse_ebooks"),
      content: String::from( "of course, as you probably
already know, people", ),
      reply: false,
      retweet: false,
};

println!("1 new tweet: {}", tweet.summarize_default());
```

Traitlerin Fonksiyon Parametresi Olarak Kullanılması

```
pub fn notify(item: &impl Summary) {

      println!("Breaking news! {}", item.summarize());
}
```

Main() fonksiyonun da da aşağıdaki gibi çağrılır.

notify (&tweet)

Burada fonksiyona gönderilen türün; trait özelliğinin uygulanmış olan türlerden olduğuna dikkat etmeliyiz.

Yani tweet değişkeninin sahip olduğu veri türü, Summary tarit'ini implement etmiş olması gerekir.

Traitlerin Geri Dönüş Tipi Olarak Kullanılması

```
fn returns_summarizable() -> impl Summary {
        Tweet {
                username: String::from("horse_ebooks"),
                content: String::from( "of course, as you probably already know,
people", ),
                reply: false,
                retweet: false,
        }
}
```

Burada dönüş türü olarak tek bir tür vermemiz gerekir. İf kalıbıyla iki farklı türü döndürmek istesek hata oluşur.

```
fn returns_summarizable(switch: bool) -> impl Summary {
    if switch {
        NewsArticle {
            headline: String::from(
                "Penguins win the Stanley Cup Championship!",
            ),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        }
    } else {
        Tweet {
            username: String::from("horse_ebooks"),
            content: String::from(
                "of course, as you probably already know, people",
            ),
            reply: false,
            retweet: false,
        }
    }
}
```

Hata oluşur

ÇÖZÜM

```
fn returns_summarizable(switch: bool) -> Box<dyn Summary> {
    if switch {
        Box::new(NewsArticle {
            headline: String::from("Penguins win the Stanley Cup Championship!"),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        })
    } else {
        Box::new(Tweet {
            username: String::from("horse_ebooks"),
            content: String::from("of course, as you probably already know, people"),
            reply: false,
            retweet: false,
        })
    }
}
```

Trait Bound Sözdizimi (fonksiyonun farklı bir söz dizimi ile tanımlanmasıdır)

Daha uzun bir tanımlamadır. Bir önceki tanımlama basit durumlar için kullanılır.

```
pub fn notify<T: Summary>(item: &T) {

    println!("Breaking news! {}", item.summarize());

}
```

Fonksiyona birden fazla parametre göndermemiz gerekebilir

```
pub fn notify(item1: &impl Summary, item2: &impl Summary) {
```

```
pub fn notify<T: Summary>(item1: &T, item2: &T) {
```

Buradaki parametrelerin aynı türden olmasına dikkat etmelisiniz.

Fonksiyonlara Birden Fazla Trait Gönderebiliriz (+ operatörü kullanılır)

```
pub fn notify(item: &(impl Summary + Display)) {
```

```
pub fn notify<T: Summary + Display>(item: &T) {
```

Display → Değişken değerlerini yazarken (scalar types için) kullanılan { } yapısının tanımlanmasıdır.

Fonksiyonlarda Trait Kullanılımını where ile Daha Net Hale Getirilmesi

Birden fazla generic tip ve birden fazla trait kullanıldığında tanımlamayı daha okunur yapmak için where değimi kullanılır.

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {
```

Tanımlamasını aşağıdaki gibi daha okunur hale getirebiliriz.

```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
      T: Display + Clone,
      U: Clone + Debug,
{
```

Özellikleri Sınırlandırmak İçin Yöntemlerde Koşul Kullanma

```
use std::fmt::Display;

struct Pair<T> {
        x: T,
        y: T,
}
```

```
impl<T> Pair<T> {
        fn new(x: T, y: T) -> Self {
                Self { x, y }
        }
}
```

Burada new() metodu ile structın yeni bir örneği döndürülmektedir.

Display → Print içinde { } parantezler arasında değişken değeri yazmak için rust içinde tanımlanmış olan trait'tir.
PartialOrd → <T> gibi generic tiplerde >, >= gibi operatörlerin kullanımını sağlayan trait'tir.

Aşağıdaki implement işleminde ise x, y değerleri karşılaştırılarak çıktı verilmektedir.

```
impl<T: Display + PartialOrd> Pair<T> {

      fn cmp_display(&self) {
            if self.x >= self.y {
                  println!("The largest member is x = {}", self.x);
            } else {
                  println!("The largest member is y = {}", self.y);
            }
      }
}
```

Blanket Implementation - Kapsamlı Implementasyon

Başka bir trait'i implement eden herhangi bir tip için de bir trait'i koşullu olarak implement edebiliriz.

Örneğin Rust standard kütüphanesi ToString traitini, Display traitini implement eden bütün türler için implement eder.

```
impl<T: Display> ToString for T {
        // --snip--
}
```

Biz bu implementasyonu (integerlar için) → `let s = 3.to_string();` Şeklinde kullanmaktayız.

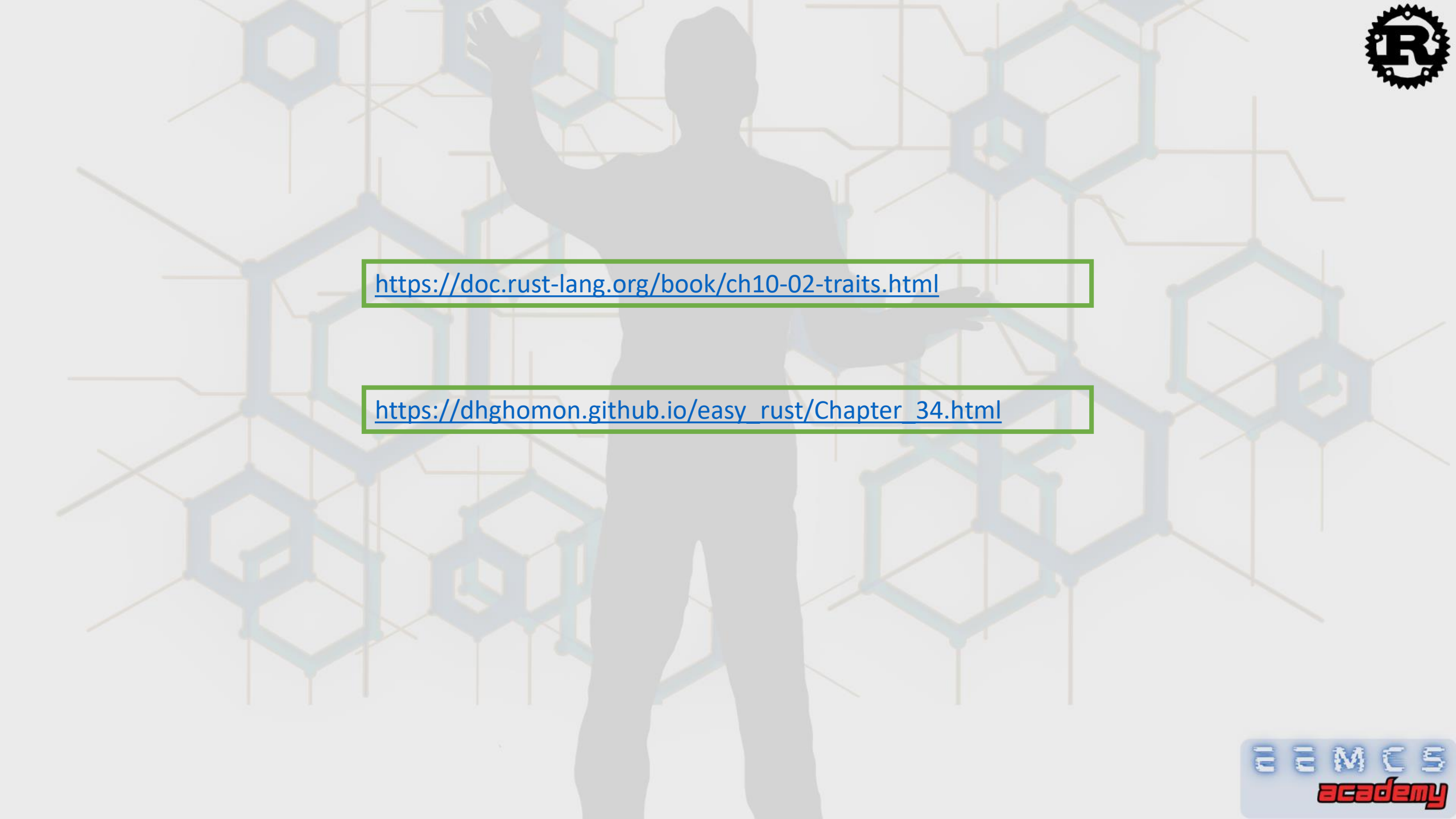
Struct İçin Display Trait Implementasyonu

```
use std::fmt;

struct Point {
        x: i32,
        y: i32,
}

impl fmt::Display for Point {
        fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
                write!(f, "({}, {})", self.x, self.y)
        }
 }

fn main() { let p1 = Point { x: 1, y: 2 }; println!("{}", p1); → Çıktı (1, 2) olur
}
```



https://doc.rust-lang.org/book/ch10-02-traits.html

https://dhghomon.github.io/easy_rust/Chapter_34.html

Celal AKSU
Bilişim Teknolojileri Öğretmeni

celalaksu@gmail.com

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

https://twitter.com/ksacll

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