RUST
Generic Data Types
(Genel Veri Tipleri)
<T, V, ...>

Generic Data Type Nedir? <K, V, ..>

- Herhangi bir veri tipini ifade eder
- Fonksiyon, struct, enum, vector, metod gibi yapılarda veri tipi kısıtlamasını kaldırmak için kullanılır.
- Yani oluşturulan yapı bütün veri tipleri ile çalışabilir duruma gelir. (istisnalar olabilir)
- Kod tekrarını önler.
- Generic Veri Tipi büyük harflerle belirtilirler.
- Her harf, karşılığında birbirinden farklı veri tipi kullanılacağını ifade eder.
- Bir harfin kullanıldığı her yerde, harfin temsil ettiği veri tipi kullanılır.

Fonksiyonlarda Kullanımı

```rust
fn buyuk_sayiyi_bul(list: &[i32]) ->
&i32 {
    let mut buyuk_olan = &list[0];

    for sayi in list {
        if sayi > buyuk_olan {
            buyuk_olan = sayi;
        }
    }
    buyuk_olan
}
```

```rust
fn buyuk_karakteri_bul(list: &[char]) ->
&char {
    let mut buyuk_olan = &list[0];

    for karakter in list {
        if karakter > buyuk_olan {
            buyuk_olan = karakter;
        }
    }
    buyuk_olan
}
```

Bu iki fonksiyon aynı işi iki farklı tür için yapmaktadır. Bu aynı kodu tekrar yazmak demektir. İki fonksiyon yerine tek bir fonksiyon yazabiliriz.

```rust
fn main() {
    let sayı_listesi = vec![34, 50, 25, 100, 65];
    let buyuk_sayi = buyuk_sayiyi_bul(&sayı_listesi);
    println!("The largest number is {}", buyuk_sayi);

    let karakter_listesi = vec!['y', 'm', 'a', 'q'];
    let buyuk_karakter = buyuk_karakteri_bul(&karakter_listesi);
    println!("The largest char is {}", buyuk_karakter);
}
```

Generic data types kullanılarak iki tür için yapılan işi gerçekleştirern tek bir fonksiyon yazabiliriz.

```rust
fn buyuk_elemanı_bul<T>(list: &[T]) -> &T {
    let mut buyuk_olan = &list[0];

    for eleman in list {
        if eleman > buyuk_olan {
            buyuk_olan = eleman;
        }
    }
    buyuk_olan
}
```

Not : ">" operatörü generic tiplerle kullanmak için fonksiyon adından sonra

<T:PartialOrd>

trait'ini eklememiz gerekir.

Buradaki T harfi, fonksiyonun farklı veri tipleri ile çalıştığını ifade etmektedir.
Veri tipi belirtmemiz gereken yerlerde kullandığımıza dikkat edelim.
Fonksiyon adından sonra kullandığımıza dikkat edelim. Aslında burada kulanılacak generic type sayısı belirtilir.
Eğer iki veya daha fazla veri tipi kullanılacaksa <T,K,.. > şeklinde fonksiyon adından sonra yazılır.

```rust
let sayı_listesi = vec![34, 50, 25, 100, 65];

let buyuk_sayi = buyuk_elemanı_bul(&sayı_listesi);

println!("The largest number is {}", buyuk_sayi);

let karakter_listesi = vec!['y', 'm', 'a', 'q'];

let buyuk_karakter = buyuk_elemanı_bul(&karakter_listesi);

println!("The largest char is {}", buyuk_karakter);
```

Böylece tek fonksiyon her iki veri tipi için de kullanılmış olur.

Struct ve Generic Data Types

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

fn main() {
	let integer = Point { x: 5, y: 10 };
	let float = Point { x: 1.0, y: 4.0 };

	let wont_work = Point { x: 5, y: 4.0 }; // Burada farklı iki veri tipi gönderilmiş. T tek bir veri tipini temsil ettiği için hata verir.
}
```

Struct ve Generic Data Types

```
struct Point<T, U> {
      x: T,
      y: U,
}

fn main() {
      let both_integer = Point { x: 5, y: 10 };
      let both_float = Point { x: 1.0, y: 4.0 };
      let integer_and_float = Point { x: 5, y: 4.0 };
}
```

Enum ve Generic Data Types

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

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

Method ve Generic Data Types

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

impl<T> Point<T> {
        fn get_x(&self) -> &T {
                        &self.x
        }
}

fn main() {
        let p = Point { x: 5, y: 10 };
        println!("p.x = {}", p.get_x());
}
```

Method ve Generic Data Types

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

// Tek bir veri tipi içinde metod tanımlayabiliriz.

impl Point<f32> {
        fn distance_from_origin(&self) -> f32 { (
                self.x.powi(2) + self.y.powi(2)).sqrt()
        }
}
```

Method ve Generic Data Types

Farklı veri tiplerinden oluşan struct yapılarını da metotlarda işleyebiliriz.

```
struct Point<X1, Y1> {
        x: X1,
        y: Y1, }

impl<X1, Y1> Point<X1, Y1> {
        fn karıstır<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
                Point { x: self.x, y: other.y, }
        }
}
```

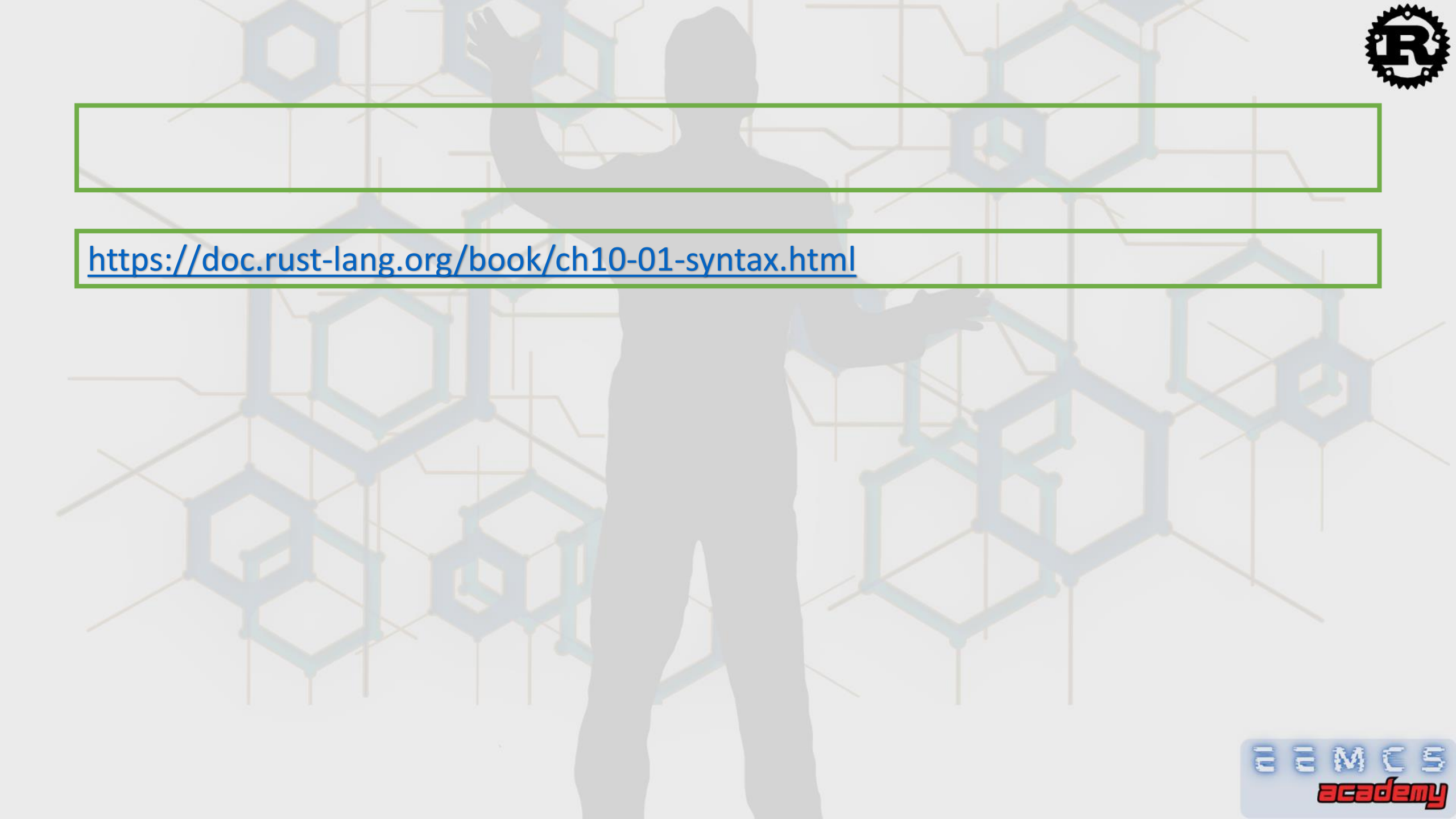
Burada metod, farklı veri tipine sahip iki struct'ı ele almaktadır.
Birinci struct self parametresi ile alınmakta ve metodu çağıran structtır.
İkinci struct ise metoda parametre olarak gönderilmektedir.
İki structın kullandığı veri tipleri farklıdır.

Metod iki struct tan birer değer alıp yeni bir struct oluşturmaktadır.

```
fn main() {

    let p1 = Point { x: 5, y: 10.4 };

    let p2 = Point { x: false, y: 'c' };

    let p3 = p1.mixup(p2);

// p1 ve p2 nin kullandığı veri tipleri farklıdır.

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);

}
```

impl struct'a ait olduğu için X1 ve Y1 impl ile birlikte belirtilmiştir.

X2 ve Y2 parametreleri sadece metod ile ilgili olduğu için orada belirtilmişlerdir.



https://doc.rust-lang.org/book/ch10-01-syntax.html

Celal AKSU
Bilişim Teknolojileri Öğretmeni

celalaksu@gmail.com

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

https://twitter.com/ksacll

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