RUST
Kullanıcı Tanımlı Veri Tipi
Ve
Metotlar
struct & impl

struct Nedir?

- Rust'ta tanımlı olan veri tiplerini kullanarak kendinize ait bir yeni veri tipi oluşturmanızı sağlar.
- Bu sayede bağlantılı olan değişkenleri bir araya getirerek gruplandırmış olursunuz.
- tuple'a benzeyen, içerisinde farklı veri türlerini barındıran yeni bir veri tipi oluşturmuş olursunuz.
- tuple'dan farkı, verilere bir değişken adı verebilirsiniz ve verilerin sıralaması önemli değildir.
- Struct, oluşturacağınız veri tipinin bir **şablon**udur.
- Struct içindeki değişkenler **fields/alanlar** olarak isimlendirilir.
- Değişken tanımlama kurallarını ve struct kurallarını takip ederek yeni veri tipinden değişkenler tanımlar ve veri atamalarını yaparsınız.
- Struct'tan tanımlanan değişkenlere **struct's instance (struct'ın örneği)** olarak isimlendirilir.
- Yeni veri tipinde oluşturulan değişkenler üzerinde işlem yapan özel fonksiyonlar oluşturabilirsiniz. Bu fonksiyonlar artık **method** olarak isimlendirilirler.

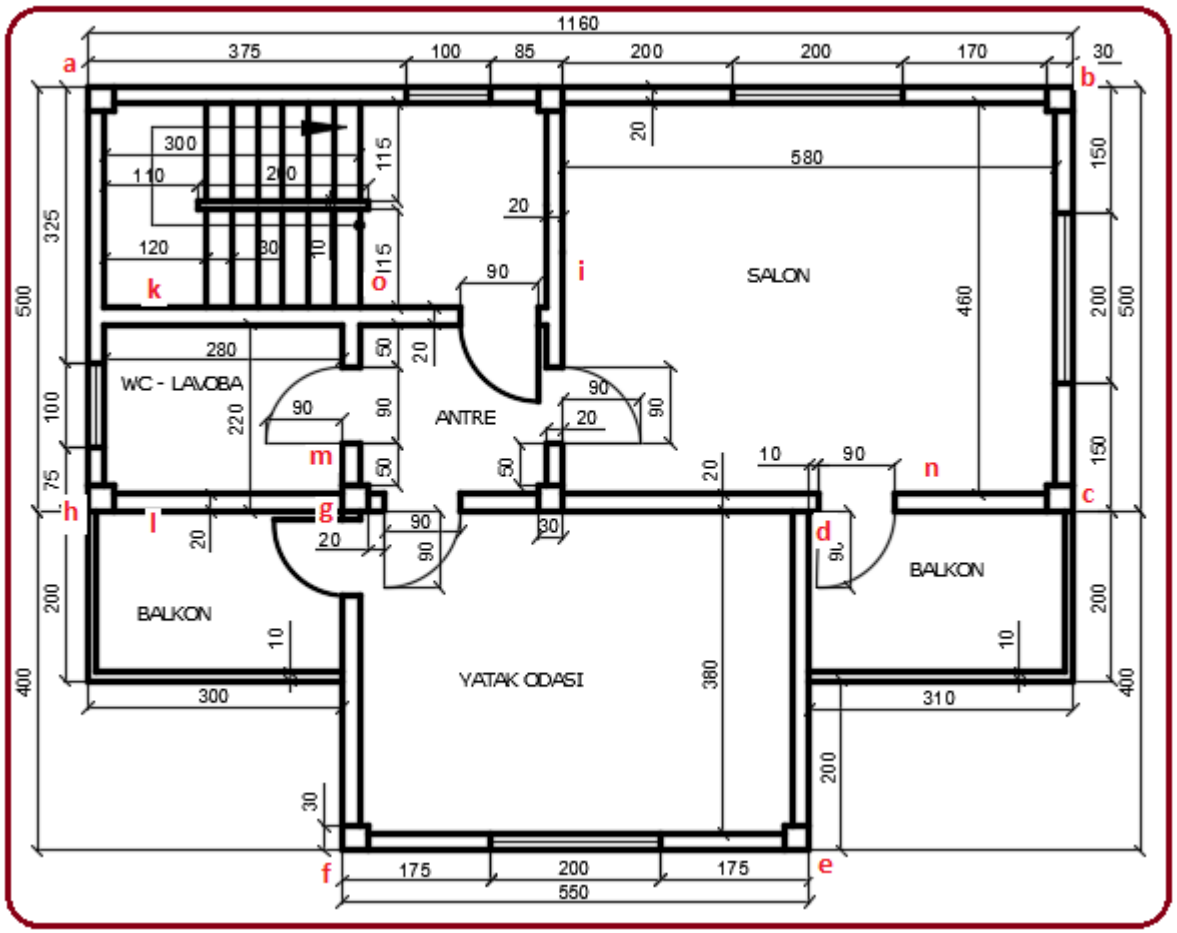
struct

İnstance-örnek

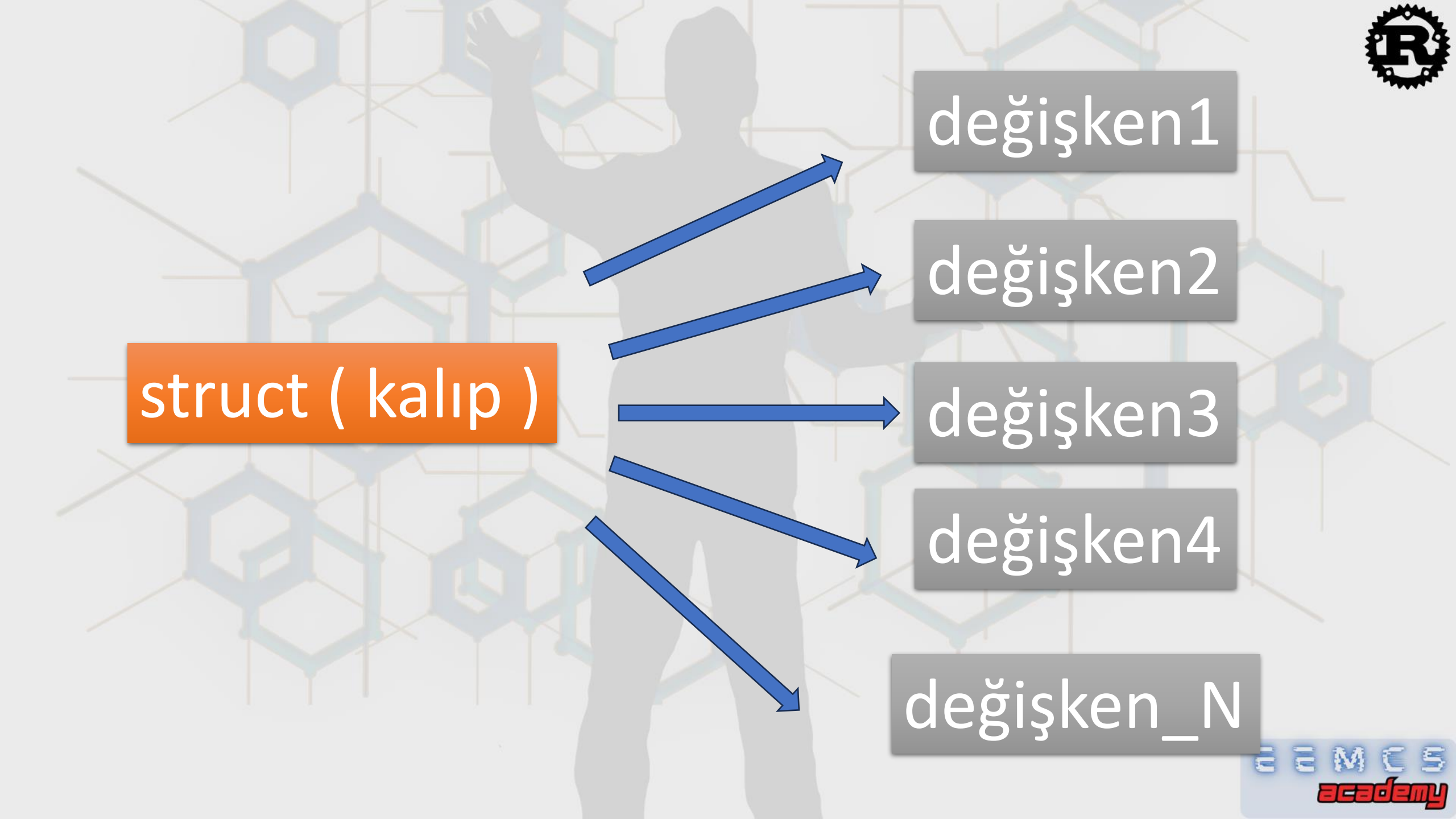
struct

İnstance - örnek

struct



İnstance - örnek (object - nesne)


struct (kalıp)
değişken1
değişken2
değişken3
değişken4
değişken_N

struct Tanımlama

```
struct Structadı {

        field1 : veritipi ,
        field2 : veritipi ,
        .
        .
        fieldN : veritipi ,

}
```

- Struct adının baş harfi büyük diğer harfler küçük yazılır
- field/alan değişken isimleridir
- Oluşturulan struct, yeni veri tipinin şablonudur.
- Her alan adından sonra ,-virgül olmalıdır.
- main() dışında tanımlanabilirler

```
struct User {
        active: bool,
        username: String,
        email: String,
        sign_in_count: u64,
}
```

Struct Tipinden Değişken Tanımlama - Struct'ın örneğini (instance) oluşturmak

```
let ornek_adi = Struct_Adı {
      field1 : value1 ,
      field2 : value2 ,
      .
      fieldN : valueN,
};
```

Alanların hangi sırada yazıldığı önemli değildir

```
let user1 = User {
      active: true,
      username: String::from("someusername123"),
      email: String::from("someone@example.com"),
      sign_in_count: 1,
};
```

Struct Örneğinin Alanlarına Ulaşmak (.)

ornek_adı.field_adı

Mutable Instance Tanımlama

Struct'nın tamamı mutable olmalıdır.
Struct içindeki alanlar ayrı ayrı mutable olarak tanımlanamaz

```
let mut ornek_adı = Struct_adı { ……..……..
```

Fonksiyon Kullanarak Instance Oluşturmak

```
fn build_user(email: String, username: String) -> User {
      User {
            active: true,
            username: username,
            email: email,
            sign_in_count: 1,
      }
}
```

User sonunda noktalı virgül olmadığına dikkat ediniz. Return kuralı

Fonksiyon Kullanarak Instance Oluşturmak (field init shorhand syntax)

```
fn build_user(email: String, username: String) -> User {
      User {
            active: true,
            username ,
            email ,
            sign_in_count: 1,
      }
}
```

Var olan instance'ı kullanarak yeni instance oluşturmak

```
let user2 = User {
      active: user1.active,
      username: user1.username,
      email: String::from("another@example.com"),
      sign_in_count: user1.sign_in_count,
};
```

Ownership ve heap memory kurallarına dikkat etmeliyiz.
String veride move olayı gerçekleşmektedir
user1'den yeni örnek oluşturduğumuzda kullanılamaz duruma gelir.

Var olan instance'ı kullanarak yeni instance oluşturmak - Değişmeyen değerlerin aktarılması ..ornek_adı

```
let user2 = User {
      email: String::from("another@example.com"),
      ..user1
};
```

Başka bir structtan veri alınırken "=" atama operatörü kullanılmış gibi davranır.
Sadece stack memory deki veriler alınırsa önceki struct kullanılır fakat heap memorydeki veri alınırsa kullanılamaz (copy ve move - ownership)

İsimsiz Tuple Structlar

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Yapılar aynı olsa da farklı değişkenlerdir

Verilere erişmek için .index kullanılır

Alansız Unit Struct

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Trait implementasyonunda kullanılır.

Struct içinde Referans (&) Tiplerinin Kullanımı

Kullanılabilir
Lifetime özelliği uygulanmalıdır
(HENÜZ ANLATILMADI)

```
struct User {
        active: bool,
        username: &str,
        email: &str,
        sign_in_count: u64,
}
```

```
let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
};

// HATA VERİR
```

Örnek :

```
struct Rectangle {
    width: u32,
    height: u32,
}
```

```
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
```

```
fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Debug trait ve Struct - `#[derive(Debug)]`

- println! İle struct yapısını bütün olarak yazdıramayız
- Struct yapısını bütün olarak görmek için kullanılır.
- Hata ayıklamamızı kolaylaştırır.

Basit veriler yazdırılırken Display trait kullanılır ve bu standard kütüphanede bulunur fakat Struct yapısı için tanımlanmamıştır.

Struct'tan önce `#[derive(Debug)]` şeklinde özellik olarak eklenir. Çıktı için println! içinde verinin yazdırılacağı güzel parantezler `{:?}` şeklinde ya da `{:#?}` kullanılır ve değişken olarakta sadece struct örneğinin adı yazılır.

```
#[derive(Debug)]
struct Rectangle {
        width: u32,
        height: u32,
}

fn main() {
        let rect1 = Rectangle {
                                width: 30,
                                height: 50,
                                };

        println!("rect1 is {:?}", rect1);
}
```

dbg! macrosu

Debug kullanarak çıktı almak için başka bir yoldur.

Bir ifadenin sahipliğini alır (println! Makrosu referans almaktadır.)

Sahipliğini aldığı ifadenin dosya ve satır numarasını yazdırır.

Değerin sahipliğini geri döndürür.

Not : dbg! → standard error console stream (stderr) çıktısı verir. (hata yönetimi için kullanılır)

println! → standard output console stream (stdout) çıktısı verir.

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

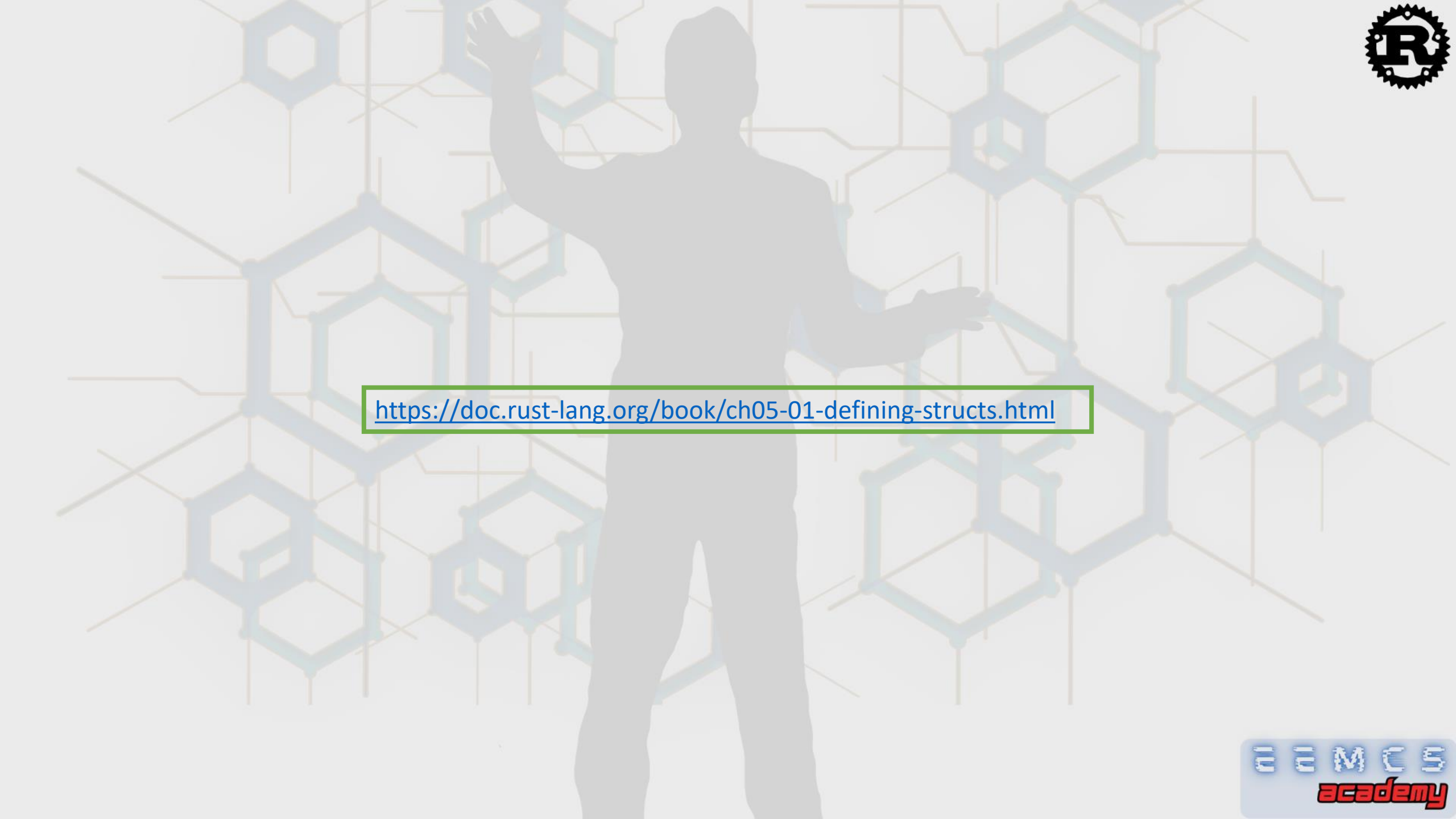
```
[src/main.rs:10] 30 * scale = 60
[src/main.rs:14] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

Default trait ve Struct - #[derive(Default)]

Değer atamadan struct örneği oluşturmamızı sağlar.

```
let mut struct_örnek_adı = Struct_adı::default();

#[derive(Default)]
struct Struct_adı {
        .
        .
        .
}
```



https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Struct için Method Tanımlama

- Metod yapısı fonksiyonlarla aynıdır.
- Burada tanımlanan fonksiyon struct'a ait olduğu için metod olarak isimlendirilir.
- Sadece ait olduğu struct tipi için kullanılır.
- Farklı olarak **&self** parametresini almaktadır
- **impl (implementation)** değimi ile oluşturulurlar.

Struct için Method Tanımlama

```
impl Struct_adı {

        fn fonksiyon_adı(&self, parametre : &parametre_tipi, ....) -> return_tipi {

        }
}
```

Kullanımı:

```
ornek_adı.method(parametreler )
```

&self → Struct'ın kendisini ifade eder.

Yani parametre olarak **parametre : &Struct_adı** verilmesi ile aynıdır.

self : & self yapısının kısaltılmış halidir.

& referans alınarak yani ödünç alınarak kullanıldığına dikkat ediniz.

&mut self → Eğer veride değişiklik yapılacak ise mutable tanımlanmalıdır.

Parametreler → Fronksiyonlarla aynıdır.

```rust
struct Rectangle {
    width: u32,
    height: u32,
}
```

```rust
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

```rust
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
```

Associated Functions - self parametresiz fonksiyonlar

- Bu fonksiyonlar struct'ın örneğine ihtiyaç duymaz
- Bu yüzden self parametresi yoktur
- String::from fonksiyonu buna örnektir ve bu şekilde kullanılır.
- Method değillerdir.
- Genellikle constructor olarak kullanılırlar. Constructor, bir structın örneğinin nasıl oluşturulacağını tanımlayan fonksiyondur.
- Constructor bazen new olarak isimlendirilir fakat new özel bir isim değildir, dil içinde varsayılan olarak tanımlanmamıştır.

```rust
impl Rectangle {

        fn square(size: u32) -> Self {

                Self {
                        width: size,
                        height: size,
                }

}
```

```
Kullanımı:

let sq = Rectangle::square(3);
```

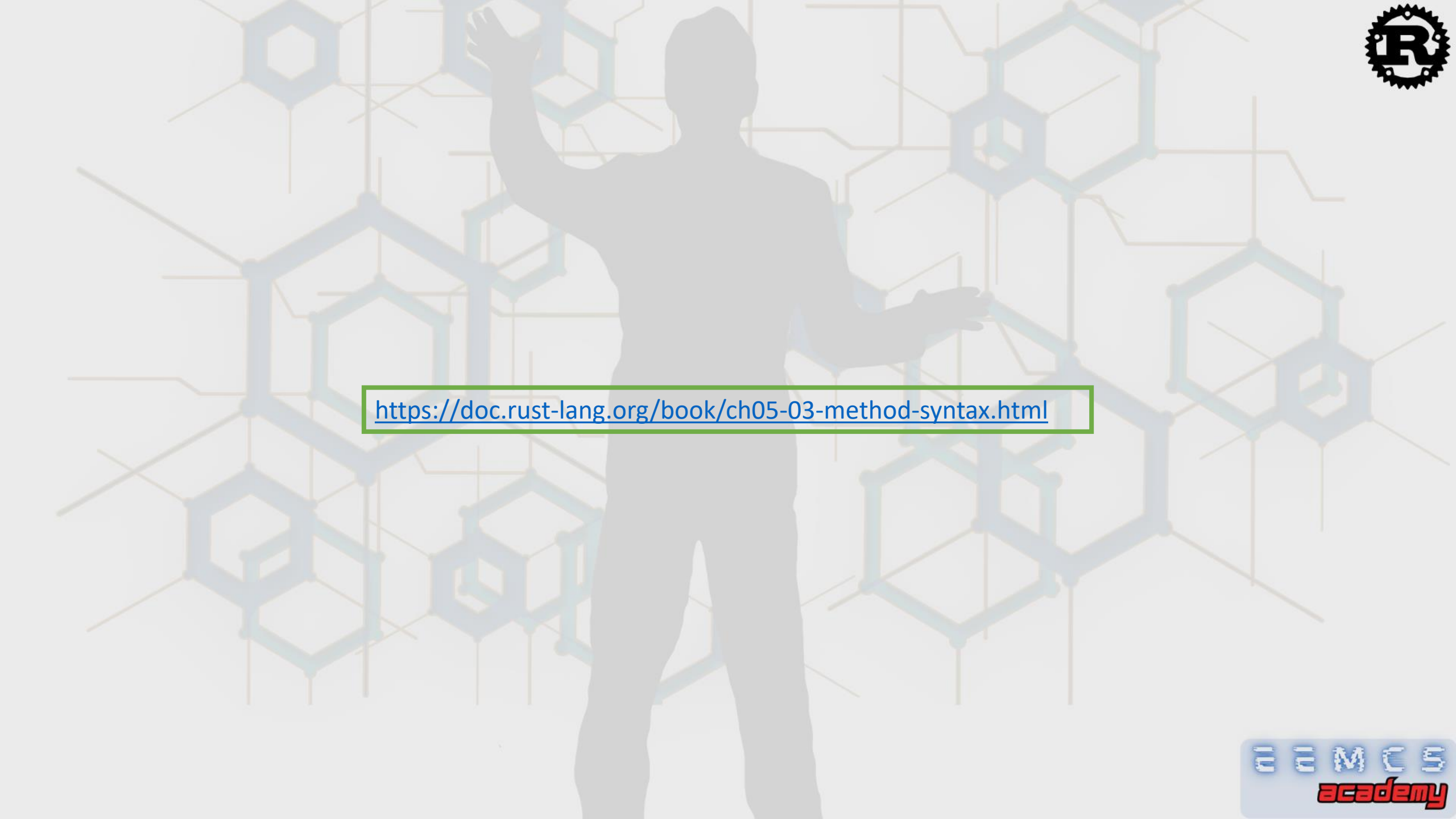
Çoklu impl Blokları

```
impl Rectangle {
            fn area(&self) -> u32 {
                self.width * self.height
            }
}

impl Rectangle {
            fn can_hold(&self, other: &Rectangle) -> bool {
                self.width > other.width && self.height >
other.height
            }
}
```

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}",
rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}",
rect1.can_hold(&rect3));
}
```



https://doc.rust-lang.org/book/ch05-03-method-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