RUST
Trait Objects & Box Data Type
Trait & Vector
Trait & HashMap

Trait Object Nedir?

Rust'ın güçlü bir özelliğidir.
Dinamik ve esnek kod yazmamızı sağlar
Genellikle aynı özelliği/davranışı implement eden çoklu veri tiplerinde kullanılır

Trait nesneleri kullanılırken, kullanılan veri tipinin önemi yoktur, önemli olan aynı trait'i implement etmiş olmalarıdır.

Rust bunun için dyn (Dynamic Dispatch - dinamik gönderim) diye adlandırılan bir yöntem kullanır.

3 - Kullanmak istediğiniz bir veri için ; verinin türünün önemli olmadığı ama bu veri tipinin bir trait'i implenment etmesinin daha önemli olduğu durumlarda Box kullanabilirsiniz.

Kısaca veriler için aynı Trait implementasyonu sağlamak için kullanılır.

(TRAIT KONUSUNDAN SONRA AÇIKLANACAK)

BOX VERİ YAPISI KONUSUNDAN

Trait'ten Nesne Oluşturma

```
trait Animal {
    fn noise(&self) -> &'static str;
}
```

```
struct Sheep {}
```

```
impl Animal for Sheep {
    fn noise(&self) -> &'static str {
        "baaaaah!"
    }
}
```

```
let trait_nesnesi: Box<dyn Animal> =
Box::new(Sheep {});
```

Veri tipi önemli değil, önemli olan aynı trait'i impl etmesi:

```
struct Cow {}
```

```
impl Animal for Cow {
    fn noise(&self) -> &'static str {
        "moooooo!"
    }
}
```

```
fn random_animal(random_number: f64) -> Box<dyn Animal> {
    if random_number < 0.5 {
        Box::new(Sheep {})
    } else {
        Box::new(Cow {})
    }
}
```

Neden Box Kullandık :

Çünkü;

Rust compiler, geri dönen değerin ne kadar yer kapladığını bilmesi gerekir.

Bu örnekte iki farklı veri tipinden herhangi biri geri döndürülüyor. Dolayısıyla geri dönen nesnenin boyutu her bir nesne için farklı olacaktır.

Box ile heap memory de tutulan verinin referansı (adres bilgisi) tutulduğu ve veri boyutu sabit olacağı için Box tanımlaması kullanılmaktadır.

dyn Nedir?

dyn -> Dynamic Dispatch (dinamik gönderim) : Verilerinin türü belirtilmeden, trait özelliği kullanılarak gönderilmesidir.

Bu sayede tip dönüşümü yapmadan nesneleri fonksiyonlara gönderebilir ve de dönüş tipi olarak ta kullanabiliriz.

Static Dispatch ?

fn do_something<T: Animal> (x: T) { } -> Daha önceden kullandığımız gönderim türüdür.

Rust Animal traitini implement eden her tür için fonksiyonu yeniden oluşturur.

```
fn main() {
    let random_number = 0.234;
    let animal = random_animal(random_number);
    println!(
        "You've randomly chosen an animal, and it says {}",
        animal.noise()
    );
}
```

Fonksiyona trait nesnesi göndermek:

```
fn get_noise(animal: Box<dyn Animal>) {
    println!("{}",animal.noise());
}
```

```
    let trait_nesnesi1: Box<dyn Animal> = Box::new(Sheep {});
    let trait_nesnesi2: Box<dyn Animal> = Box::new(Cow {});

    get_noise(trait_nesnesi1);
    get_noise(trait_nesnesi2);
```

Trait İle Farklı Veri Tiplerini Birlikte Kullanmak: Vector - HashMap

```
    let mut animals_vector: Vec<Box<dyn Animal>> = Vec::new();

    let mut animals_hashmap: HashMap<String, Box<dyn Animal>> =
HashMap::new();
```

https://doc.rust-lang.org/rust-by-example/trait/dyn.html

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

Başka bir örnek

```rust
fn main() {
    println!("Hello, world!");

    let s_t: Box<dyn Selam> = Box::new(Turkce);
    let s_i: Box<dyn Selam> = Box::new(Ingilizce);

    merhabalar(s_t);
    merhabalar(s_i);
}

trait Selam {
    fn selamlama(&self);
}

struct Turkce;
impl Selam for Turkce {
    fn selamlama(&self) {
        println!("Herkese merhaba!")
    }
}

struct Ingilizce;
impl Selam for Ingilizce {
    fn selamlama(&self) {
        println!("Hello everybody!")
    }
}

fn merhabalar(selam: Box<dyn Selam>) {
    selam.selamlama();
}
```

Celal AKSU
Bilişim Teknolojileri Öğretmeni

celalaksu@gmail.com

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

https://twitter.com/ksacll

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