🦀 به راه Rust با یونس : جلسه پنجم (فصل پنجم کتاب اصلی Rust)

> **مدرس:** یونس قربانی **موضوع:** ساختارهای داده (Using Structs to Structure Related Data) **مرجع:** The Rust Programming Language (Latest Edition)

🎯 چک‌لیست تدریس در استریم (Presenter Checklist)

- [x] ~~۱. تعریف Struct و نحوه مقداردهی (Instantiating)~~
- [x] ~~۲. استفاده از Field Init Shorthand (وقتی نام متغیر و فیلد یکی است)~~
- [x] ~~۳. مفهوم Struct Update Syntax (کپی کردن فیلدها با `..`)~~
- [x] ~~۴. انواع Structهای خاص: **Tuple Structs** و **Unit-like Structs**~~
- [x] ~~۵. مالکیت داده در ساختار (Ownership) و اشاره‌ای به لایف‌تایم‌ها~~
- [x] ~~۶. استفاده از `#[derive(Debug)]` برای چاپ و دیباگ Struct~~
- [ ] ۷. پیاده‌سازی متدها با بلوک `impl` و پارامتر `&self`
- [ ] ۸. توابع مرتبط (Associated Functions) که متد نیستند (مثل `String::from`)
- [ ] ۹. مدیریت چندین بلوک `impl` برای یک Struct

💻 تکه‌های کد کلیدی جلسه (Code Snippets)

۱. تعریف، ساخت و آپدیت Struct

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

fn main() {
    // 1. Basic instantiation
    let mut user1 = User {
        active: true,
        username: String::from("younes"),
        email: String::from("younes@example.com"),
        sign_in_count: 1,
    };

    // 2. Modifying a field (requires the instance to be mutable)
    user1.email = String::from("newemail@example.com");

    // 3. Struct Update Syntax (creating user2 from user1 fields)
    let user2 = User {
        email: String::from("another@example.com"),
        ..user1 // Note: username is moved here because String doesn't implemen
Copy!
```

```
    };
}
```

۲. تاپل استراکت‌ها و یونیت استراکت‌ها

```
// Tuple structs: useful for giving a name to a whole tuple
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

// Unit-like structs: useful for traits when no data storage is needed
struct AlwaysEqual;

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

    let subject = AlwaysEqual;
}
```

۳. متدها و توابع مرتبط (Rectangle Example)

```
#[derive(Debug)] // Allows printing with {:?}
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    // Method: takes &self to read the struct data
    fn area(&self) -> u32 {
        self.width * self.height
    }

    // Method with extra parameters
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }

    // Associated Function: doesn't take self (often used as a constructor)
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };
    println!("The area is {} pixels.", rect1.area());
}
```

🧠 تقلب‌نامه و نکات کلیدی برای دانشجویان (Key Concepts)

۱. چرا Struct؟

استراکت‌ها به ما اجازه میدن داده‌های مرتبط رو بسته‌بندی کنیم و بهشون اسم بدیم. برخلاف Tuple، اینجا ترتیب فیلدها مهم نیست و با اسم بهشون دسترسی داریم، که کد رو خیلی خواناتر میکنه.

۲. مالکیت (Ownership) در استراکت

وقتی یک فیلد از نوع `String` رو در استراکت داریم، استراکت مالک اون داده هست. اگه از **Struct Update Syntax** (`..user1`) استفاده کنید، فیلدهایی که `Copy` نیستند (مثل String) به استراکت جدید **Move** میشن و استراکت قدیمی دیگه قابل استفاده نیست (مگر اینکه اون فیلد خاص رو دوباره مقداردهی کنید).

۳. متدها (`&self`) در برابر توابع مرتبط

- **متدها:** داخل بلوک `impl` تعریف میشن و پارامتر اولشون همیشه `self` (یا `&self` یا `&mut self`) هست. با سینتکس نقطه (`rect.area()`) صدا زده میشن.
- **توابع مرتبط (Associated Functions):** پارامتر `self` ندارند. معمولاً برای ساخت اینسنتس جدید (مثل `new` یا `square`) استفاده میشن و با دو نقطه دوتایی (`Rectangle::square(10)`) صدا زده میشن.

۴. دیباگ کردن با `#[derive(Debug)]`

چون استراکت‌ها ساختار پیچیده‌ای دارن، مستقیماً با `{}` چاپ نمیشن. باید بالای استراکت از `#[derive(Debug)]` استفاده کنید و موقع چاپ از `{:?}` (یا `{:#?}` برای چاپ خوش‌فرم و چندخطی) استفاده کنید.