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

> **مدرس:** یونس قربانی
> **موضوع:** درک مفهوم مالکیت (Understanding Ownership, References, Borrowing & Slices)
> **مرجع:** The Rust Programming Language (Latest Edition)

🎯 چک‌لیست تدریس در استریم (Presenter Checklist)

- [x] ~~۱. فلسفه Ownership و تفاوت مدیریت حافظه در Rust~~
- [x] ~~۲. مقایسه حافظه استک (Stack) و هیپ (Heap) و هزینه‌های دسترسی~~
- [x] ~~۳. قوانین سه‌گانه و مقدس مالکیت (Ownership Rules)~~
- [x] ~~۴. طول عمر متغیرها (Variable Scope) و متد آزادسازی `drop`~~
- [x] ~~۵. نوع داده `String` پویا و تفاوت رفتار متغیرها: `Move` در برابر `Copy` (تفاوت Traitهای `Copy` و `Clone`)~~
- [ ] ۶. تعامل توابع با مالکیت (Move شدن پارامترها و بازپس‌گیری مالکیت در Return)
- [ ] ۷. مراجع و قرض‌گرفتن (References & Borrowing) با عملگر `&`
- [ ] ۸. مراجع تغییرپذیر (`&mut`) و قوانین آهنین جلوگیری از Data Race
- [ ] ۹. جلوگیری کامپایلر از ارجاع آویزان (Dangling References)
- [ ] ۱۰. مفهوم برش‌ها (`&str` :The Slice Type و Array Slices) برای ارجاع امن به زیرمجموعه‌ای از داده‌ها

💻 تکه‌های کد کلیدی جلسه (Code Snippets)

۱. مکانیزم انتقال مالکیت (Move) و کپی عمیق (Clone)

```rust
fn main() {
    // 1. Simple stack data (automatically copied because it implements Copy)
    let x = 5;
    let y = x; // The value was copied; x is still valid!
    println!("x: {x}, y: {y}");

    // 2. Heap-allocated data (ownership is moved)
    let s1 = String::from("hello");
    let s2 = s1; // Ownership moved to s2; s1 is no longer valid!

    // println!("{s1}"); // ❌ Compiler error: borrow of moved value: `s1`
    println!("{s2}"); // ✅ This is valid

    // 3. Deep copy of heap-allocated data
    let s3 = s2.clone();
    println!("s2: {s2}, s3: {s3}"); // Both are valid because the heap data was cloned
}
```

۲. قرض‌گرفتن (Borrowing) و مراجع تغییرپذیر (`&mut`)

```rust
fn main() {
    let mut s = String::from("hello");

    // 1. Immutable borrowing (immutable reference)
    let len = calculate_length(&s);
    println!("The length of '{s}' is {len}.");

    // 2. Mutable borrowing (mutable reference)
    change(&mut s);
    println!("After the change: {s}");

    // 3. Borrowing rules that prevent data races
    let mut data = String::from("Rust");
    let r1 = &data; // No problem
    let r2 = &data; // No problem
    println!("{r1} and {r2}");
    // The borrows held by r1 and r2 end after their last use, thanks to NLL (Non-Lexical Lifetimes).

    let r3 = &mut data; // Now we can create a mutable reference without an error!
    r3.push_str(" Lang");
}

fn calculate_length(s: &String) -> usize {
    s.len()
} // s goes out of scope, but the heap data isn't dropped because s doesn't own it.

fn change(some_string: &mut String) {
    some_string.push_str(", world!");
}
```

۳. برش‌ها (Slices) و پیدا کردن اولین کلمه

```rust
fn main() {
    let my_string = String::from("hello world");

    // Slice a String to produce a &str
    let word = first_word(&my_string);
    println!("The first word is: {word}");

    // Works directly with string literals (&str), too
    let my_literal = "hello world";
    let word = first_word(my_literal);
    println!("The first word is: {word}");

    // Slicing an array
    let a = [1, 2, 3, 4, 5];
    let slice: &[i32] = &a[1..3]; // Includes the elements at indices 1 and 2
    assert_eq!(slice, &[2, 3]);
}

// Taking a &str is more flexible than taking a &String; it accepts both.
fn first_word(s: &str) -> &str {
```

```rust
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

🧠 تقلب‌نامه و نکات کلیدی برای دانشجویان (Key Concepts)

۱. قوانین سه‌گانه مالکیت (The Ownership Rules)

1. در زبان Rust، هر مقدار در حافظه دارای **یک مالک (Owner)** است.
2. در هر لحظه از زمان، **فقط و فقط یک مالک** می‌تواند وجود داشته باشد.
3. هنگامی که مالک از اسکوپ (محدوده `{ }`) خارج شود، مقدار موجود در حافظه فوراً و خودکار با فراخوانی تابع `drop` **آزاد (Free)** میشود.

---

۲. استک (Stack) در برابر هیپ (Heap)

- **استک (Stack):** فوق‌العاده سریع، ساختار LIFO (آخرین ورود، اولین خروج)، اندازه داده‌ها باید در زمان کامپایل **دقیقاً مشخص و ثابت** باشد (مانند انواع عددی، بولین و آرایه‌های با طول معین).
- **هیپ (Heap):** انعطاف‌پذیر، برای داده‌هایی که در زمان اجرا سایز نامشخص دارند یا تغییر سایز می‌دهند (مثل `String` یا وکتورها). سیستم‌عامل فضای خالی را پیدا کرده و یک اشاره‌گر (Pointer) روی استک قرار می‌دهد.

---

۳. تفاوت Move با Shallow Copy

- در زبان‌هایی مثل پایتون یا جاوااسکریپت، وقتی یک آبجکت را به متغیر دیگر انتساب می‌دهید، پوینتر کپی می‌شود اما هر دو به یک حافظه نگاه می‌کنند (Shallow Copy).
- در Rust، برای جلوگیری از خطای وحشتناک **Double Free** (آزاد کردن دوباره یک حافظه در پایان اسکوپ)، به محض اختصاص `s1` به `s2`، متغیر قبلی بی‌اعتبار (Invalidate) می‌شود که به این عمل **Move** می‌گویند.
- اگر دیتایی Trait مربوط به `Copy` را داشته باشد (مثل `i32`)، مقدارش خودکار کپی می‌شود و Move رخ نمی‌دهد.

---

۴. قوانین آهنین قرض گرفتن (The Rules of References)

کامپایلر Rust (توسط Borrow Checker) در زمان کامپایل این قانون را چک می‌کند:

> در هر اسکوپ، شما می‌توانید **هر تعداد رفرنس فقط‌خواندنی** (`&T`) داشته باشید، **یا** فقط و فقط **یک رفرنس تغییرپذیر** (`&mut T`). داشتن همزمان این دو حالت غیرمجاز است!

- **چرا این قانون وجود دارد؟** برای جلوگیری از وقوع **Data Race** در زمان اجرا (زمانی که چند اشاره‌گر همزمان قصد تغییر یا خواندن داده‌ای را دارند بدون اینکه همگام‌سازی شده باشند).
- **قانون دوم:** رفرنس‌ها همیشه باید معتبر باشند؛ هیچ رفرنسی نباید به مقداری اشاره کند که قبلاً در حافظه نابود شده است (**No Dangling References**).

---

۵. نوع اسلایس (Slice Type)

- اسلایس‌ها نمایشی (View) بدون مالکیت به بخشی متوالی از یک کالکشن هستند.
- **برش رشته (`&str`):** ساختاری دونقطه‌ای روی استک شامل یک اشاره‌گر به شروع داده و یک طول (`length`).
- **نکته تجربی در کدنویسی:** برای امضای توابع، به جای اینکه ورودی را `&String` تعریف کنید، آن را `&str` بگذارید؛ چون به واسطه قابلیت `Deref Coercion` ، تابع شما هم `&String` و هم رشته‌های لیترال (`&str`) را می‌پذیرد.