한 번에 끝내는
블록체인 개발 A to Z

Chapter 1

Rust Introduction

Chapter 1 Rust Introduction

Pattern Matching

Match

Rust has an extremely powerful control flow construct called match that allows you to compare a value against a series of patterns and then execute code based on which pattern matches.

We don’t typically use curly brackets if the match arm code is short. If you want to run multiple lines of code in a match arm, you must use curly brackets.

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Patterns that Bind to Values

```
#[derive(Debug)] // so
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

Match with Option

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

Exhaustive

We didn't handle the None case, so this code will cause a bug. Luckily, it's a bug Rust knows how to catch. If we try to compile this code, we'll get this error:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        Some(i) => Some(i + 1),
    }
}
```

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/main.rs:3:15
    |
3   |         match x {
    |               ^ pattern `None` not covered
    |
    = help: ensure that all possible cases are being handled,
    = note: the matched value is of type `Option<i32>`
```

_ Placeholder

Rust also has a pattern we can use when we don't want to use the value in the catch-all pattern: _, which is a special pattern that matches any value and does not bind to that value. This tells Rust we aren't going to use the value, so Rust won't warn us about an unused variable.

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

Concise Control Flow with if let

The if let syntax lets you combine if and let into a less verbose way to handle values that match one pattern while ignoring the rest.

```
let config_max = Some(3u8);
match config_max {
    Some(max) => println!("The maximum is configured to be {}", max),
    _ => (),
}
```

```
let config_max = Some(3u8);
if let Some(max) = config_max {
    println!("The maximum is configured to be {}", max);
}
```

if let and else

We can include an else with an if let. The block of code that goes with the else is the same as the block of code that would go with the _ case in the match expression that is equivalent to the if let and else.

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

```
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```