6_Enums_en

May 2, 2023

1 Enums

Enums allow you to define a type by enumerating its possible values.

Currently any IP address can be either a version four or a version six address, but not both at the same time. That property of IP addresses makes the enum data structure appropriate, because enum values can only be one of the variants. Both version four and version six addresses are still fundamentally IP addresses, so they should be treated as the same type when the code is handling situations that apply to any kind of IP address.

[ ]:
```
enum IpAddrKind {
    V4, // this is a variant
    V6, // this is another variant
};
```

2 Enum values

`IpAddrKind` is now a custom data type that we can use elsewhere in our code. We can create instances of each of the two variants of `IpAddrKind` as shown bellow. Both values `IpAddrKind::V4` and `IpAddrKind::V6` are of the same type: `IpAddrKind`.

[ ]:
```
enum IpAddrKind {
    V4, // this is a variant
    V6, // this is another variant
}
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;
```

[ ]:
```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind, // <-- using the IpAddrKind type
    address: String,
}
```

```
let home = IpAddr {
    kind: IpAddrKind::V4,  // <-- instantiating the V4 variant
    address: String::from("127.0.0.1"),
}

let loopback = IpAddr {
    kind: IpAddrKind::V6,  // <-- instantiating the V6 variant
    address: String::from("::1"),
};
```

We can represent the same concept in a more concise way using just an enum, rather than an `enum` inside a `struct`, by putting data directly into each enum variant. This new definition of the `IpAddr` enum says that both `V4` and `V6` variants will have associated `String` values:

[ ]:
```
enum IpAddr {
    V4(String), // <-- the variant hold String data for the address
    V6(String), // <-- same
}

let home = IpAddr::V4(String::from("127.0.0.1")); // <-- instantiating the␣
 ↪variant

let loopback = IpAddr::V6(String::from("::1"));
```

There's another advantage to using an enum rather than a struct: each variant can have different types and amounts of associated data.

Wanting to store IP addresses and encode which kind they are is so common that the standard library has a definition we can use! Let's look at how the standard library defines IpAddr: it has the exact enum and variants that we've defined and used, but it embeds the address data inside the variants in the form of two different structs, which are defined differently for each variant:

[ ]:
```
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
```

You can put any kind of data inside an enum variant: strings, numeric types, or structs, for example. You can even include another enum!

2.1 The *Option* Enum and Its Advantages Over Null Values

Option is an enum defined by the standard library. The Option type is used in many places because it encodes the very common scenario in which a value could be something or it could be nothing. Expressing this concept in terms of the type system means the compiler can check whether you've handled all the cases you should be handling; this functionality can prevent bugs that are extremely common in other programming languages.

Rust doesn't have the null feature that many other languages have. Null is a value that means there is no value there. In languages with null, variables can always be in one of two states: null or not-null.The problem with null values is that if you try to use a null value as a not-null value, you'll get an error of some kind. Rust does not have nulls, but it does have an enum that can encode the concept of a value being present or absent. This enum is defined by the standard library as follows:

[ ]:
```
enum Option<T> {
    Some(T), // some value
    None,  // not a valid value
};
```

This enum is so useful that it's even included in the prelude; you don't need to bring it into scope explicitly. In addition, so are its variants: you can use **`Some`** and **`None`** directly without the Option:: prefix.

[ ]:
```
let some_number = Some(5);
let some_string = Some("a string");

let absent_number: Option<i32> = None;
```

When we have a `Some` value, we know that a value is present and the value is held within the `Some`. When we have a `None` value, in some sense, it means the same thing as null: we don't have a valid value. Because `Option< T >` and `T` (where `T` can be any type) are different types, the compiler won't let us use an `Option< T >` value as if it were definitely a valid value. For example, this code won't compile because it's trying to add an `i8` to an `Option< i8 >`:

[ ]:
```
let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

You have to convert an `Option< T >` to a `T` before you can perform operations with it. Generally, this helps catch one of the most common issues with null: assuming that something isn't null when it actually is.

2.2 The *match* Control Flow Operator

To get the `T` value out of a `Some` variant when you have a value of type `Option< T >`, you can use the **match** expression. The match expression is a control flow that runs different code depending on which variant of the enum it has, and that code can use the data inside the matching value. It allows you to compare a value against a series of patterns and then execute code based on which pattern matches.

In the following example, the function can take an unknown United States coin and, in a similar way as the counting machine, determine which coin it is and return its value in cents:

[ ]:
```
fn main() {
    enum Coin {
        Penny,
        Nickel,
        Dime,
        Quarter,
    }

    fn value_in_cents(coin: Coin) -> u8 {
        match coin {
            Coin::Penny => 1,  // this is a match "arm"
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }
}
main();
```

If you want to run multiple lines of code in a match arm, you can use curly brackets.

[ ]:
```
fn main() {
    enum Coin {
        Penny,
        Nickel,
        Dime,
        Quarter,
    }

    fn value_in_cents(coin: Coin) -> u8 {
        match coin {
            Coin::Penny => {
                println!("Lucky penny!");
                1  // it is the last value of the block that is returned
            },
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }
    value_in_cents(Coin::Penny);
}
main();
```

Pattern matching with value bind:

[ ]:
```rust
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
}

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

let mut count = 0;
let coin = Coin::Quarter(UsState::Alaska);

match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    //            ^^^^^ <-- pattern matching with value bind (state)
    _ => count += 1,
}
```

2.3 Matching with Option< T >

Let's say we want to write a function that takes an `Option<i32>` and, if there's a value inside, adds 1 to that value. If there isn't a value inside, the function should return the `None` value and not attempt to perform any operations.

[ ]:
```rust
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None, // no value
        Some(i) => Some(i + 1), // add 1 to the value
    }
}

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

Consider this version of our plus_one function that has a bug and won't compile:

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

We didn't handle the `None` case, so this code will cause a bug. Rust knows that we didn't cover

every possible case and even knows which pattern we forgot! Matches in Rust are exhaustive: we must exhaust every last possibility in order for the code to be valid. Especially in the case of `Option<T>`, when Rust prevents us from forgetting to explicitly handle the `None` case, it protects us from assuming that we have a value when we might have null.

2.4 The _ Placeholder

Rust also has a pattern we can use when we don't want to list all possible values. For example, a u8 can have valid values of 0 through 255. If we only care about the values 1, 3, 5, and 7, we don't want to have to list out 0, 2, 4, 6, 8, 9 all the way up to 255. Fortunately, we don't have to: we can use the special pattern _ instead:

[ ]:
```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

2.5 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.

Consider the following program that matches on an `Option<u8>` value but only wants to execute code if the value is 3:

[ ]:
```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

Instead, we could write this in a shorter way using if let. The following code behaves the same:

[ ]:
```
if let Some(3) = some_u8_value {
    println!("three");
}
```

3 Exercises

Exercise 1: Re-write the following code using *if let*

[ ]:
```
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
```

```
}

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

let mut count = 0;
let coin = Coin::Quarter(UsState::Alaska);

// TO DO Re-write the code below using if let
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
```

Solution:

[ ]:
```
#[derive(Debug)]
enum UsState {
    Alabama,
    Alaska,
}

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

let mut count = 0;
let coin = Coin::Quarter(UsState::Alaska);

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

Exercise 2: Create an enum to classify a web event, which may be: * a page load * a page unload * a key pressed * a paste (a text was pasted) * a click (with x and y coordinates)

Implement a function that takes a web event as parameter and depending on the event type, displays: * info that a page was loaded * info that a page was unloaded * the key that was pressed * the pasted text * the coordinates of the clicked position

[ ]:
```
// TODO (1) Define an enum called WebEvent holding the different types of events

// TODO (2) Implement a function that takes a web event as parameter and␣
 ↪displays a custom message for each event type

fn main() {
    // TODO (3) Test your implementation for different event types
}
main();
```

Solution:

[ ]:
```
enum WebEvent {
    PageLoad,
    PageUnload,
    KeyPress(char),
    Paste(String),
    Click { x: i64, y: i64 },
}

fn inspect(event: WebEvent) {
    match event {
        WebEvent::PageLoad => println!("page loaded"),
        WebEvent::PageUnload => println!("page unloaded"),
        WebEvent::KeyPress(c) => println!("pressed '{}'.", c),
        WebEvent::Paste(s) => println!("pasted \"{}\".", s),
        WebEvent::Click { x, y } => {
            println!("clicked at x={}, y={}.", x, y);
        },
    }
}

fn main() {
    let pressed = WebEvent::KeyPress('x');
    let pasted  = WebEvent::Paste("my text".to_string());
    let click   = WebEvent::Click { x: 20, y: 80 };
    let load    = WebEvent::PageLoad;
    let unload  = WebEvent::PageUnload;

    inspect(pressed);
    inspect(pasted);
    inspect(click);
    inspect(load);
    inspect(unload);
}
main();
```