Rust - Day 7

Enums and Pattern Matching

- Enums allow you to define a type by enumerating its possible variants

Defining an Enum

- Where structs give you a way of grouping together related fields and data, like a `Rectangle` with its `width` and `height`, enums give you a way of saying a value is one of a possible set of values.
- For example, we may want to say that `Rectangle` is one of a set of possible shapes that also includes `Circle` and `Triangle`. To do this, Rust allows us to encode these possibilities as an enum.
- Say we need to work with IP addresses. Currently, two major standards are used for IP addresses: version four and version six. Because these are the only possibilities for an IP address that our program will come across, we can *enumerate* all possible variants, which is where enumeration gets its name.
- **An enum value can only be one of its variants**

```
enum IpAddrKind {
    V4,
    V6,
}
```

- `IpAddrKind` is now a custom data type that we can use elsewhere in our code

Enum Values

```
let four = IpAddrKind::V4;
let six = IpAddrKind::V6
```

- Note that the variants of the enum are namespaced under its identifier, and we use a double colon to separate the two.
- This is useful because now both values `IpAddrKind::V4` and `IpAddrKind::V6` are of the same type: `IpAddrKind`. We can then, for instance, define a function that takes any `IpAddrKind`:

```
fn route(ip_kind: IpAddrKind) {}
```

- And we can call this function with either vairant

```
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
```

- Using enums has even more advantages. Thinking more about our IP address type, at the moment we don’t have a way to store the actual IP address *data*; we only know what *kind* it is. Given that you just learned about structs in Chapter 5, you might be tempted to tackle this problem with structs as shown below.

```
enum IpAddrKind {
    V4,
    V6,
}

struct IpAddr {
    kind: IpAddrKind,
    address: String,
}
```

```
let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
}

let loopback = ipAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
}
```

- Here, we've defined a struct `IpAddr` that has two fields: a `kind` field that is of type `IpAddrKind` (the enum we defined previously) and an `address` field of type `String`.
- We have two instances of this struct. The first is `home`, and it has the value `IpAddrKind::V4` as its `kind` with associated address data of `127.0.0.1`. The second instance is `loopback`. It has the other variant of `IpAddrKind` as its `kind` value, `V6`, and has address `::1` associated with it. We've used a struct to bundle the `kind` and `address` values together, so now the variant is associated with the value.
- However, representing the same concept using just an enum is more concise: rather than an enum inside a struct, we can put data directly into each enum vairant. This new definition of the `IpAddr` enum says that both `V4` and `V6` variants will have associated `String` values:

```
enum IpAddr {
        V4(String),
        V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));

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

- We attach data to each variant of the enum directly, so there is no need for an extra struct.
- **It's also easier to see another detail of how enums work: the name of each enum variant that we define also becomes a function that constructs an instance of the enum.**
- **`IpAddr::V4()` is a function call that takes a `String` argument and returns an instance of the `IpAddr` type. We automatically get this constructor function defined as a result of defining the enum.**
- **There's another advantage to using an enum rather than a struct: each variant can have different types and amounts of associated data. Version four IP addresses will always have four numeric components that will have values between 0 and 255. If we wanted to store `V4` addresses as four `u8` values but still express `V6` addresses as one `String` value, we wouldn't be able to with a struct.**

```
enum IpAddr {
        V4(u8, u8, u8, u8),
        V6(String),
}

let home = IpAddr::V4(127, 0, 0 , 1);

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

- 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),
}
```

- **This code illustrates that you can put any kind of data inside an enum variant: strings, numeric types, or structs, for example. You can even include another enum!**
- Note that even though the standard library contains a definition for `IpAddr`, we can still create and use our own definition without conflict because we haven't brought the standard library's definition into our scope.
- Let's look at another example of an enum in Listing 6-2: this one has a wide variety of types embedded in its variants.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

- This enum has four variants with different types:
  - `Quit` has no data associated with it at all.
  - `Move` has named fields, like a struct does.
  - `Write` includes a single `String`.
  - `ChangeColor` includes three `i32` values.
- The following structs could hold the same data that the preceding enum variants hold:

```
struct QuitMessage;
struct MoveMessage {
        x: i32,
        y: i32,
}
struct WriteMessage(String);
struct ChangeColorMessage(i32, i32, i32);
```

- **But if we used the different structs, each of which has its own type, we couldn't as easily define a function to take any of these kinds of messages as we could with the `Message` enum defined in Listing 6-2, which is a single type.**
- **There is one more similarity between enums and structs: just as we're able to define methods on structs using `impl`, we're also able to define methods on enums. Here's a method named `call` that we could define on our `Message` enum:**

```
impl Message {
        fn call(&self) {
                // method body would be defined here
        }

        let m = Message::Write(String::from("hello"));
        m.call();
}
```

- The body of the method would use `self` to get the value that we called the method on. In this example, we've created a variable `m` that has the value `Message::Write  from("hello")`, and that is what `self` will be in the body of the `call` method when `m.call()` runs.

Option Enum and its advantages over Null values

- The `Option` type encodes the very common scenario in which a value could be something or it could be nothing.

- For example, if you request the first item in a non-empty list, you would get a value. If you request the first item in an empty list, you would get 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. Because this null or not-null property is pervasive, it's extremely easy to make this kind of error.
- **However, the concept that null is trying to express is still a useful one: a null is a value that is currently invalid or absent for some reason.**
- **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 `Option<T>`**, and it is defined by the standard library

```
enum Option<T> {
    None,
    Some(T),
}
```

> **The `Option<T>` enum is so useful that it's even included in the prelude; you don't need to bring it into scope explicitly. Its variants are also included in the prelude: you can use `Some` and `None` directly without the `Option::` prefix. The `Option<T>` enum is still just a regular enum, and `Some(T)` and `None` are still variants of type `Option<T>`.**

- **The `<T>` syntax is a feature of Rust we haven't talked about yet. It's a generic type parameter, and we'll cover generics in more detail in Chapter 10. For now, all you need to know is that `<T>` means that the `Some` variant of the `Option` enum can hold one piece of data of any type, and that each concrete type that gets used in place of `T` makes the overall `Option<T>` type a different type.**

```
let some_number = Some(5);
let some_char = Some('e');

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

- **The type of `some_number` is `Option<i32>`. The type of `some_char` is `Option<char>`, which is a different type. Rust can infer these types because we've specified a value inside the `Some` variant.**
- **For `absent_number`, Rust requires us to annotate the overall `Option` type: the compiler can't infer the type that the corresponding `Some` variant will hold by looking only at a `None` value.**
- **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. So why is having `Option<T>` any better than having null?**
- In short, 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;
```

```
$ cargo run
   Compiling enums v0.1.0 (file:///projects/enums)
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:5:17
  |
5 |     let sum = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
```

```
 = help: the trait `Add<Option<i8>>` is not implemented for `i8`
 = help: the following other types implement trait `Add<Rhs>`:
           `&'a i8` implements `Add<i8>`
           `&i8` implements `Add<&i8>`
           `i8` implements `Add<&i8>`
           `i8` implements `Add`

For more information about this error, try `rustc --explain E0277`.
error: could not compile `enums` (bin "enums") due to 1 previous error
```

- You have to convert an `Option<T>` to a `T` before you can perform `T` 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.
- **Eliminating the risk of incorrectly assuming a not-null value helps you to be more confident in your code. In order to have a value that can possibly be null, you must explicitly opt in by making the type of that value `Option<T>`. Then, when you use that value, you are required to explicitly handle the case when the value is null. Everywhere that a value has a type that isn't an `Option<T>`, you *can* safely assume that the value isn't null. This was a deliberate design decision for Rust to limit null's pervasiveness and increase the safety of Rust code.**
- In general, in order to use an `Option<T>` value, you want to have code that will handle each variant. You want some code that will run only when you have a `Some(T)` value, and this code is allowed to use the inner `T`. You want some other code to run only if you have a `None` value, and that code doesn't have a `T` value available. The `match` expression is a control flow construct that does just this when used with enums: it will run different code depending on which variant of the enum it has, and that code can use the data inside the matching value.

The `match` Control Flow Construct

- 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.
- Patterns can be made up of literal values, variable names, wildcards, and many other things; Chapter 18 covers all the different kinds of patterns and what they do. The power of `match` comes from the expressiveness of the patterns and the fact that the compiler confirms that all possible cases are handled.

```
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,
    }
}
```

- **This seems very similar to a conditional expression used with `if`, but there's a big difference: with `if`, the condition needs to evaluate to a Boolean value, but here it can be any type.**
- If you want to run multiple lines of code in a match arm, you must use curly brackets, and the comma following the arm is then optional.

```
fn value_in_cents(coin: Coin) -> u8 {
  match coin {
    Coin::Penny => {
      println!("Lucky penny!");
```

```
            1
        }
        Coin::Nickel ⇒ 5,
        Coin::Dime ⇒ 10,
        Coin::Quarter ⇒ 25,
    }
}
```

Catch all

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

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

moving on with some practical stuff now, too much theory is covered already