Enum and Pattern Matching

Defining an Enum

Enum or enumeration gives you a way of defining a set of possible values for one value. "This value can be these possible set of values".

To define an enum to be a set of possible values here is an example:

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

Here, the enum `IpAddrKind` is a custom data type that can have two possible values, V4 and V6. After defining this enum you can use it elsewhere in your code.

To define an enum with one of its variations you would use the `::` syntax under the enum name like such:

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

Function taking enum

You can also then use enum as a function parameter:

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


// invoking it
route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

Advantage of enum over struct

Using enum you can actually give an associated data with each possible variations. You can give say `V4` three `u32` and for `V6` a String value like such:

```
enum IpAddr {
    V4(u32, u32, u32),
    V6(String),
}
```

Doing it this way, you do not need to make extra struct to associate data with each of the enum variations.

The associated data for each enum variant can be anything: strings, numeric types, or even structs!

To actually give the value when creating it you would do something like so:

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

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

Enum examples

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

In this case there are four variants of the Message enum

1. Quit: Has no associated data
2. Move: has named fields just like a struct
3. Write: Has a single String
4. ChangeColor: Has three `i32` values

Methods on Enum

Rememberthat the `impl` block work on structs and also enums. So you can define methods for each enum type that you have defined:

```
impl Message {
    fn call(&self) {
        // method body
    }
}

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

You can call it on the enum variant that you have defined just like how you can call it on an instance of a struct.

The match control flow construct

To actually retrieve the associated value out from the enum variant and do conditional with it you would have to use the `match` construct.

You would be able to use the `match` construct to compare a value against a series of patterns then execute code based on which pattern it is matched.

`match` not only work with enum types but literal values, variable names, wildcards, and other things as well!

Testing match with enum

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

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

The function `value_in_cents` will take in an enum of Coin type and then return it's corresponding variant value in `u8`.

To use the `match` expression, you would have to use the match keyword follow by the value that you want to pattern match, then you would list out all the possible combination of pattern that you are matching for this particular value.

The match arms have two parts, the pattern and some code, if the code is short and one line long, then you don't need another set of brackets, however, if you have multiple lines of code to execute if the pattern matches then you would need the brackets. The pattern and code is separated by the `=>` operator.

When `match` expression executes, it compares the value against the pattern of each arm in order, if it matches then that code is executed.

Pattern with associated values

To retrieve the value for a corresponding enum variant you can follow the same syntax like so:

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

In this case, we have a enum variant Quarter with an associated value of `bool` to indicate whether it is rare or not. If we are going to write a function to retrieve that `bool` value from the Quarter variant, how would we do that?

```
fn retrieve_rare(coin: Coin) -> bool {
match coin {
    Coin::Quarter(rare) => rare,
        other => false,
}
}
```

Now if we are calling the function like so `retrieve_rare(Coin::Quarter(true))` this will yield `true` as it's return value. How does it work? Well, we are constructing a Quarter variant of `Coin` enum with `true` as it's associated data to indicate that it is indeed rare. Then when that enum type is passed into the function it will be doing a pattern match, it matches the first pattern because it is a Quarter, the associated data is binded to the variable `rare`, and then we are just simply return `rare` as it is because we just want to get that value out from the enum.

> Matches must be exhaustive, it must cover all possibilities. If is missing some possibility then the code will not compile!

Catch-all pattern

If you are only interested in say two out of ten possible patterns and want to handle the rest of the pattern one way, you don't have to code out all of the pattern matches, and instead use a catch-all pattern.

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

fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

As you can see, the `other` arm at the end will be a catch-all pattern that handles all other patterns that is not 3 and 7. The value of all the none matched pattern are stored into `other`.

If we do not need the value, and just need to catch-all then you can use the pattern `_` to do the catch-all. Otherwise, if you don't use the value in catch-all Rust will warn you about unused variable.

`if let` control flow

Combining if and let let you handle values that match one pattern while ignoring the rest. For example:

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

This code will print out the value inside the enum `Some` and for any other variant it will do nothing (i.e. for the `None` variant it will do nothing).

However, writing `_` everytime and writing these verbose pattern matching for just doing something small for one pattern is just too repetitive, so `if let` let you condense this into much shorter syntax:

```
let config_max = Some(3);
if let Some(max) = config_max {
println!("The max is {}", max);
}
```

This is equivalent to the `match` expression done previously but now you don't have to write the `_` catch-all pattern and just focus on the one case you actually care. It works the same way as a `match` expression, it will bind the value inside `Some` to `max` variable if `config_max` is a `Some` enum variant.

You can also include an `else` with `if let` syntax. Which is the same as the things inside `_` the catch-all pattern:

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

This code will increment `count` if the Coin enum isn't an Quarter variant.

> Generally, use `if let` if you are only expecting to handle one of the enum variant and ignoring the rest.

Revision #1
Created 2023-01-29 18:10:49 UTC by Tamarine
Updated 2023-01-29 21:37:08 UTC by Tamarine