# Contents

# 1 6 Enums and pattern matching

- enums - enumerations
- allow you to define a type by enumerating it's possible

## 1.1 chapter goals

- show what enums are
- then explore the `option` enum
- then pattern matching w/ `match`
- then, `if / let`

# 2 6.1 defining an enum

- here's an example where enums may be more useful than structs
- since there are two possible types of IP address that we could come across, we can enumerate those possibilities

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

## 2.1 enum values

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

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

- so, subtypes of an enum are accessible using the double colon syntax - that is how they are namespaced
- now we can create functions that take either `type`, because the type of each is `IpAddrKind`

### 2.2 ok, so now combining enums and structs

- wow, this seems quite useful in order to associate the address w/ the type of enum

```
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"),
};
```

### 2.3 however, this can be done just by an enum!

- to do this, we define that each of the enum variants will have a `String` associated with it

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

let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));
```

- this way, we don't need to define the additional struct to hold the address data!

### 2.4 enum variant types can have different types and amounts of associated data

- ok so watch this flexibilitym, that would not be possible with a struct

- IPv4 addresses always have four sets of three numbers in [0,255]
- if we wanted to store those four numbers themselves and still store the V6 address as a string, we could do the following
-

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

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

### 2.5 working with IP addresses is so common that the `std` library contains a definition for them

IpAddr in std::net - Rust

- so the way it's implemented there is that there ARE structs for Ipv4Addr and Ipv6Addr, with are then included in the `enum IpAddr`
- one thing is that since we haven't brought `stdlib` into scope, we can define our own versions no problem
-

### 2.6 message enum demonstrating storage of different

- quit: has no data associated with it
- move: has named fields like a struct
- Write: has one String value
- ChangeColor: three i32's

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

```
    ChangeColor(i32,i32,i32),
}

// the equivalent structs
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

### 2.7 we can define methods on `enums` using `impl`

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

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

- in the example above, we are showing that `m` is the `Write` variant of the `Message` enum

### 2.8 The `Option` Enum and its Advantages Over Null Values

- `Option` is another enum define in the standard library
- this enum is so common that it, AND its variants, are "included in the prelude" -> you don;t have to explicitly import them
  - Aka you can call `Some`, `None` without the `Option` prefix
- The `Option` enum covers the situation where a value may be something, or it may be nothing
- the advantage to covering this very common situation in this way as opposed to using Null is that the compiler is able to check all cases before compilation

- aka, avoids tons of errors that are possible due to the way the Null type is usually implemented

### 2.8.1 the concept of null is good

Null values handle the situation in which a value is either absent or invalid The implementation problem is that when a null value is used as a non-nul value, errors will result

### 2.8.2 The `Option` Enum

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

### 2.8.3 the <T> syntax and the type differences it causes

- so, the variants are of the type `Option<T>`
- the `T` is a "generic type parameter" -> more in ch. 10
- "for now, all you need to know is that `<T>` means the `Some` variant of the `Option` enum can hold one piece of data at any time, and that each concrete type that gets used in palce of `T` makes the overall `Option<T>` type a different type"

```
let some_number = Some(5); // type is Option<i32>
let some_string = Some("a string"); // type is Option<&str>

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

- we **annotate the type** for the example of `absent_number` above, because rust can't infer from the `None` value that we want the type of `absent_number` to be `Option<i32>`
- effectively what will get returned from the examples below are values of the type `Option«whatever type»`, instead of just `<Type>`

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

let sum = x + y;
```

- this is where the error will be: can't add an `i8` to an `Option<i8>`
- "in short, because they are different types, the compiler won't let us use an `Option<T>` *as if it were definitely a valid value*"

#### 2.8.4 why have this variation in type?

- we have this difference because it allows us to compare whether a value is safe, and will always exist, or whether there may or may not be a value there

#### 2.8.5 TODO rewrite this section

- eliminating the rrisk 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>`

#### 2.8.6 so, how do you get the `T` value out of a `Some` variant?

- `Option<T>` has a very large number of available methods
- they will be extremely useful
- highly recommended to check out the documentation for `Option<T>` methods

#### 2.8.7 To use `Option<T>` values (in general), you must have code to handle each variant

- you want code for if you have a `Some(T)` value that is able to use the inner `T`
- you want code for if you have a `None` value, "and that code doesn't have a `T` value available.
- ooh here we go, the `Match` expression!

## 3 6.2 the `match` control flow construct

- match allows comparison of values against a series of patterns, and then to execute code based on the match.

- main benefit here is that the compiler is able to confirm that all possible cases have been handled

### 3.1 example

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


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

- new syntax: the pattern and the code in the match arm are separated by `=>`
- the number (5,10,25) is the code to be run
- the first match shows how we execute multiple lines of code within a `match` arm: we enclose it withinc urly brackets

### 3.2 patterns that bind to values

- "Another useful feature of match arms is that they can bind to the parts of the values that match the pattern"
- since it binds to the parts of the values, they can extract values from `enum` variants
- the pretext for the example below is that US states had specific back side variant pictures, so we've added a value called `UsState` to be contained within the `Quarter` variant of an enum

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


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
        }
    }
}
```

- so in the updated `value_in_cents` fn, now the passed-in value of `Coin::Quarter(UsState::Alaska)`, `coin` would proceed until it reaches `Coin::Quarter(state)`
- then, `state` is `UsState(Alaska)`, which is accessible to the `println!` macro

### 3.3 Matching with `Option<T>`

- "instead of comparing coins, we'll compare the variants of `Option<T>` but the way the match expression works remains the same"

  - goal of the above is to get the inner `T` value out of a `Some` case <- I don't remember doing this in the previous section?

- ah right, `Some` and `None` are variants of the `Option` enum that are automatically imported during the prelude because of how common they are ,_ and `Some` / `None` are both accessible without prefixing.

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

    }
}
let five = Some(5); // an enum of type ~Option<i32>~
let six = plus_one(five);
let none = plus_one(None);
```

- here, the variable within `plus_one` is Some(5). When that matches the Some(i) arm, it will then bind 5 to i and return the value 6
- in the case with `None`, it matches `None` and returns `None`

### 3.3.1 matches are sequential, and they will throw errors at compile time if you've put a catch-all arm before a more specific arm

- in the previous code block, had we not included a pattern to deal with `None` (aka, when a value is not present), the code would not have compiled
- this is an example of Rust catching all the possible cases at compile-time
- "..Rust prevents us from forgetting to explicitly handle the `None` case, it protects us from assuming we have a value when we might have null..", thus preventing the billion-dollar mistake

### 3.3.2 Catch-All Patterns And The _ Placeholder

- if you roll a three, specific action
- if you roll a seven, specific action

- otherwise, you move a certain number of spaces in some imaginary game
    - for that last arm, have we just *named* it other or is other a special word?
- and we are USING the value passed as other in this case
- when we dont wan't to *use* the value that may be present in other, we can use the catch-all pattern _~
- _ will match any value and then **not bind to that value**
- this way, rust won't warn us about an unused variable
- say if we want the player to have to reroll when they don't roll a 3 or a 7, we can use the _ operator instead of `other` because we don't care about the value and want it thrown away, OR if we want it to just do nothing aka pass the player's turn, we can do that using the unit value `()`

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other), // using the value
    _ => reroll(), // calling a function that does not use the value because _ throws i
    _ => (), // doing nothing whatever by passing the unit value and not using the valu
}


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

### 3.4 next up: concise control flow with `if let`

## 4 6.3 Concise Control Flow With `If Let`

The `if let` syntax - more concise way to match one pattern to control the rest

## 4.1 if vs match

### 4.1.1 match

- the block below shows code where a `match` only executes code when a value is `Some`
- (as a side note, the value of Some here is 3 and it's type is u8)

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

- the `_ => ()` line satisfies the match expression, but is basically boiler-plate

### 4.1.2 match - interesting syntax

here's how we can do it more concisely with `if let`

- the code in this `if let` expression will only run if the value is

```
let config_match = Some(3u8);
if let Some(max) = config_max { // this seems syntacticall odd / almost reversed to me
    println!("the max is configured to be {}", max);
}
```

- so in the if, it's basically saying **if** the type of the value it is given is Some(<x>), then execute the code in the brackets
- oh, jimbus. Thanks Matt for the explanation - we're literally just combining the if and the let. we use let all the time for assignment anyways
- within that expression, the value of `max` will be whatever was within the `Some(<x>)` type it was originally given
- Ah, because the `if let` takes:
  1. First, a pattern
  2. then, the expression to use

- this way, the expression is given to `match`
- Advantage to `if let` is shorter, less typing, easier to take
- disadvantage is that you lose the "exhaustive type checking that `match` enforces"
-

#### 4.1.3 you can also include `else` with `if let`

- here, the `else` code is equivalent to the `_ => <code>` in the full match expression here's how we'd write something to count all non-quarter coins, using the `Coin` enum we made earlier

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

- and here's how we'd do it with `if let else`

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

#### 4.1.4 summmary

"If you have a situation in which your program has logic that is too verbose to express using a `match`, remember that `if let` is in your rust toolbox as well" We've shown:

- creating custom `enum` types
- how standard library's `Option<T>` can be used to prevent errors with potential values
- using `match`, and `if let` to extract values
-

"Your Rust programs can now express concepts in your domain using structs and enums." Creating custom types in the API ensures type safety <- lets the compiler help ye out

#### 4.1.5 Next up: organizing through modules