# Enums and Pattern Matching

In this chapter, we'll look at *enumerations*, also referred to as *enums*. Enums allow you to define a type by enumerating its possible *variants*. First we'll define and use an enum to show how an enum can encode meaning along with data. Next, we'll explore a particularly useful enum, called `Option`, which expresses that a value can be either something or nothing. Then we'll look at how pattern matching in the `match` expression makes it easy to run different code for different values of an enum. Finally, we'll cover how the `if let` construct is another convenient and concise idiom available to handle enums in your code.

# 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.

Let's look at a situation we might want to express in code and see why enums are useful and more appropriate than structs in this case. 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.

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 an enum value can only be one of its 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.

We can express this concept in code by defining an `IpAddrKind` enumeration and listing the possible kinds an IP address can be, `V4` and `V6`. These are the variants of the enum:

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

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

## Enum Values

We can create instances of each of the two variants of `IpAddrKind` like this:

```
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 variant:

```
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 in Listing 6-1.

```
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"),
};
```

**Listing 6-1: Storing the data and `IpAddrKind` variant of an IP address using a `struct`**

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 variant. This new definition of the `IpAddr` enum says that both `V4` and `V6` variants will have associated `String` values:

IMP: enum and struct combined

```
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. Here, 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. That is, `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. Enums handle this case with ease:

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

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

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

We've shown several different ways to define data structures to store version four and version six IP addresses. However, as it turns out, 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),
}
```

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! Also, standard library types are often not much more complicated than what you might come up with.

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. We'll talk more about bringing types into scope in Chapter 7.

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),
}
```

**Listing 6-2: A `Message` enum whose variants each store different amounts and types of values**

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.

Defining an enum with variants such as the ones in Listing 6-2 is similar to defining different kinds of struct definitions, except the enum doesn't use the `struct` keyword and all the variants are grouped together under the `Message` type. The following structs could hold the same data that the preceding enum variants hold:

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

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:

Methods for enum just like structs

```
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(String::from("hello"))`, and that is what `self` will be in the body of the `call` method when `m.call()` runs.

Let's look at another enum in the standard library that is very common and useful: `Option`.

## The `Option` Enum and Its Advantages Over Null Values

This section explores a case study of `Option`, which is another enum defined by the standard library. 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.

Programming language design is often thought of in terms of which features you include, but the features you exclude are important too. 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.

In his 2009 presentation "Null References: The Billion Dollar Mistake," Tony Hoare, the inventor of null, has this to say:

---

> I call it my billion-dollar mistake. At that time, I was designing the first comprehensive type system for references in an object-oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

---

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.

The problem isn't really with the concept but with the particular implementation. As such, 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 as follows:

Option enum

```
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. Here are some examples of using `Option` values to hold number types and string types:

```
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. Here, we tell Rust that we mean for `absent_number` to be of type `Option<i32>`.

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;
```


If we run this code, we get an error message like this one:

```
$ 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>`:
            <i8 as Add>
            <i8 as Add<&i8>>
            <&'a i8 as Add<i8>>
            <&i8 as Add<&i8>>

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

Intense! In effect, this error message means that Rust doesn't understand how to add an `i8` and an `Option<i8>`, because they're different types. When we have a value of a type like `i8` in Rust, the compiler will ensure that we always have a valid value. We can proceed confidently without having to check for null before using that value. Only when we have an `Option<i8>` (or whatever type of value we're working with) do we have to worry about possibly not having a value, and the compiler will make sure we handle that case before using the value.

In other words, 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. IMP

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.

So how do you get the `T` value out of a `Some` variant when you have a value of type `Option<T>` so that you can use that value? The `Option<T>` enum has a large number of methods that are useful in a variety of situations; you can check them out in its documentation. Becoming familiar with the methods on `Option<T>` will be extremely useful in your journey with Rust.

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.

Think of a `match` expression as being like a coin-sorting machine: coins slide down a track with variously sized holes along it, and each coin falls through the first hole it encounters that it fits into. In the same way, values go through each pattern in a `match`, and at the first pattern the value "fits," the value falls into the associated code block to be used during execution.

Speaking of coins, let's use them as an example using `match`! We can write a function that takes an unknown US coin and, in a similar way as the counting machine, determines which coin it is and returns its value in cents, as shown in Listing 6-3.

```
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,
    }
}
```

**Listing 6-3: An enum and a `match` expression that has the variants of the enum as its patterns**

Let's break down the `match` in the `value_in_cents` function. First we list the `match` keyword followed by an expression, which in this case is the value `coin`. 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. The type of `coin` in this example is the `Coin` enum that we defined on the first line.

Next are the `match` arms. An arm has two parts: a pattern and some code. The first arm here has a pattern that is the value `Coin::Penny` and then the `=>` operator that separates the pattern and the code to run. The code in this case is just the value `1`. Each arm is separated from the next with a comma.

When the `match` expression executes, it compares the resultant value against the pattern of each arm, in order. If a pattern matches the value, the code associated with that pattern is executed. If that pattern doesn't match the value, execution continues to the next arm, much as in a coin-sorting machine. We can have as many arms as we need: in Listing 6-3, our `match` has four arms.

The code associated with each arm is an expression, and the resultant value of the expression in the matching arm is the value that gets returned for the entire `match` expression.

We don't typically use curly brackets if the match arm code is short, as it is in Listing 6-3 where each arm just returns a value. 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. For example, the following code prints "Lucky penny!" every time the method is called with a `Coin::Penny`, but still returns the last value of the block, `1`:

```
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

Another useful feature of match arms is that they can bind to the parts of the values that match the pattern. This is how we can extract values out of enum variants.

As an example, let's change one of our enum variants to hold data inside it. From 1999 through 2008, the United States minted quarters with different designs for each of the 50 states on one side. No other coins got state designs, so only quarters have this extra value. We can add this information to our `enum` by changing the `Quarter` variant to include a `UsState` value stored inside it, which we've done in Listing 6-4.

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

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

**Listing 6-4: A `Coin` enum in which the `Quarter` variant also holds a `UsState` value**

Let's imagine that a friend is trying to collect all 50 state quarters. While we sort our loose change by coin type, we'll also call out the name of the state associated with each quarter so that if it's one our friend doesn't have, they can add it to their collection.

In the match expression for this code, we add a variable called `state` to the pattern that matches values of the variant `Coin::Quarter`. When a `Coin::Quarter` matches, the `state` variable will bind to the value of that quarter's state. Then we can use `state` in the code for that arm, like so:

```
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
        }
    }
}
```

If we were to call `value_in_cents(Coin::Quarter(UsState::Alaska))`, `coin` would be `Coin::Quarter(UsState::Alaska)`. When we compare that value with each of the match arms, none of them match until we reach `Coin::Quarter(state)`. At that point, the binding for `state` will be the value `UsState::Alaska`. We can then use that binding in the `println!` expression, thus getting the inner state value out of the `Coin` enum variant for `Quarter`.

## Matching with `Option<T>`

In the previous section, we wanted to get the inner `T` value out of the `Some` case when using `Option<T>`; we can also handle `Option<T>` using `match`, as we did with the `Coin` enum! Instead of comparing coins, we'll compare the variants of `Option<T>`, but the way the `match` expression works remains the same.

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.

This function is very easy to write, thanks to `match`, and will look like Listing 6-5.

```
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);
```

**Listing 6-5: A function that uses a `match` expression on an `Option<i32>`**

Let's examine the first execution of `plus_one` in more detail. When we call `plus_one(five)`, the variable `x` in the body of `plus_one` will have the value `Some(5)`. We then compare that against each match arm:

```
        None => None,
```

The `Some(5)` value doesn't match the pattern `None`, so we continue to the next arm:

```
        Some(i) => Some(i + 1),
```

Does `Some(5)` match `Some(i)`? It does! We have the same variant. The `i` binds to the value contained in `Some`, so `i` takes the value `5`. The code in the match arm is then executed, so we add 1 to the value of `i` and create a new `Some` value with our total `6` inside.

Now let's consider the second call of `plus_one` in Listing 6-5, where `x` is `None`. We enter the `match` and compare to the first arm:

```
        None => None,
```

It matches! There's no value to add to, so the program stops and returns the `None` value on the right side of `=>`. Because the first arm matched, no other arms are compared.

Combining `match` and enums is useful in many situations. You'll see this pattern a lot in Rust code: `match` against an enum, bind a variable to the data inside, and then execute code based on it. It's a bit tricky at first, but once you get used to it, you'll wish you had it in all languages. It's consistently a user favorite.

## Matches Are Exhaustive IMP

There's one other aspect of `match` we need to discuss: the arms' patterns must cover all possibilities. Consider this version of our `plus_one` function, which has a bug and won't compile:

```
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. Luckily, it's a bug Rust knows how to catch. If we try to compile this code, we'll get this error:

```
$ 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
  |
note: `Option<i32>` defined here
 -->
/rustc/9b00956e56009bab2aa15d7bff10916599e3d6d6/library/core/src/option.rs:572:1
 :::
/rustc/9b00956e56009bab2aa15d7bff10916599e3d6d6/library/core/src/option.rs:576:5
  |
  = note: not covered
  = note: the matched value is of type `Option<i32>`
help: ensure that all possible cases are being handled by adding a match arm with
a wildcard pattern or an explicit pattern as shown
  |
4 ~             Some(i) => Some(i + 1),
5 ~             None => todo!(),
  |

For more information about this error, try `rustc --explain E0004`.
error: could not compile `enums` (bin "enums") due to 1 previous error
```

IMP

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, thus making the billion-dollar mistake discussed earlier impossible.

## Catch-all Patterns and the _ Placeholder

Using enums, we can also take special actions for a few particular values, but for all other values take one default action. Imagine we're implementing a game where, if you roll a 3 on a dice roll, your player doesn't move, but instead gets a new fancy hat. If you roll a 7, your player loses a fancy hat. For all other values, your player moves that number of spaces on the game board. Here's a `match` that implements that logic, with the result of the dice roll hardcoded rather than a random value, and all other logic represented by functions without bodies because actually implementing them is out of scope for this example:

```
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) {}
```

other & _

For the first two arms, the patterns are the literal values `3` and `7`. For the last arm that covers every other possible value, the pattern is the variable we've chosen to name `other`. The code that runs for the `other` arm uses the variable by passing it to the `move_player` function.

This code compiles, even though we haven't listed all the possible values a `u8` can have, because the last pattern will match all values not specifically listed. This catch-all pattern meets the requirement that `match` must be exhaustive. Note that we have to put the catch-all arm last because the patterns are evaluated in order. If we put the catch-all arm earlier, the other arms would never run, so Rust will warn us if we add arms after a catch-all!

IMP

Rust also has a pattern we can use when we want a catch-all but don't want to *use* the value in the catch-all pattern: `_` 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's change the rules of the game: now, if you roll anything other than a 3 or a 7, you must roll again. We no longer need to use the catch-all value, so we can change our code to use `_` instead of the variable named `other`:

```
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() {}
```

This example also meets the exhaustiveness requirement because we're explicitly ignoring all other values in the last arm; we haven't forgotten anything.

Finally, we'll change the rules of the game one more time so that nothing else happens on your turn if you roll anything other than a 3 or a 7. We can express that by using the unit value (the

empty tuple type we mentioned in "The Tuple Type" section) as the code that goes with the `_` arm:

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

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

Here, we're telling Rust explicitly that we aren't going to use any other value that doesn't match a pattern in an earlier arm, and we don't want to run any code in this case.

There's more about patterns and matching that we'll cover in Chapter 18. For now, we're going to move on to the `if let` syntax, which can be useful in situations where the `match` expression is a bit wordy.

## 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 program in Listing 6-6 that matches on an `Option<u8>` value in the `config_max` variable but only wants to execute code if the value is the `Some` variant.

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

**Listing 6-6: A `match` that only cares about executing code when the value is `Some`**

If the value is `Some`, we print out the value in the `Some` variant by binding the value to the variable `max` in the pattern. We don't want to do anything with the `None` value. To satisfy the `match` expression, we have to add `_ => ()` after processing just one variant, which is annoying boilerplate code to add.

Instead, we could write this in a shorter way using `if let`. The following code behaves the same as the `match` in Listing 6-6:

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

The syntax `if let` takes a pattern and an expression separated by an equal sign. It works the same way as a `match`, where the expression is given to the `match` and the pattern is its first arm. In this case, the pattern is `Some(max)`, and the `max` binds to the value inside the `Some`. We can then use `max` in the body of the `if let` block in the same way we used `max` in the corresponding `match` arm. The code in the `if let` block isn't run if the value doesn't match the pattern.

Using `if let` means less typing, less indentation, and less boilerplate code. However, you lose the exhaustive checking that `match` enforces. Choosing between `match` and `if let` depends on what you're doing in your particular situation and whether gaining conciseness is an appropriate trade-off for losing exhaustive checking.

In other words, you can think of `if let` as syntax sugar for a `match` that runs code when the value matches one pattern and then ignores all other values.

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`. Recall the `Coin` enum definition in Listing 6-4, where the `Quarter` variant also held a `UsState` value. If we wanted to count all non-quarter coins we see while also announcing the state of the quarters, we could do that with a `match` expression, like this:

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

Or we could use an `if let` and `else` expression, like this:

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

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.

## Summary

We've now covered how to use enums to create custom types that can be one of a set of enumerated values. We've shown how the standard library's `Option<T>` type helps you use the type system to prevent errors. When enum values have data inside them, you can use `match` or `if let` to extract and use those values, depending on how many cases you need to handle.

Your Rust programs can now express concepts in your domain using structs and enums. Creating custom types to use in your API ensures type safety: the compiler will make certain your functions only get values of the type each function expects.

In order to provide a well-organized API to your users that is straightforward to use and only exposes exactly what your users will need, let's now turn to Rust's modules.