Intro to Rust Lang

Structs and Enums

Benjamin Owad, David Rudo, and Connor Tsui

Review: Ownership

- Manual memory management is tricky
    - Prone to memory leaks or double frees
- Garbage collection is slow and unpredictable
- What if the compiler generated allocations and frees for us?
    - Rust does this for us through the *ownership* system

Review: Ownership Rules

- Each value in Rust has an *owner*
- A value can only have one owner at a time
- When the owner goes out of scope, the value will be *dropped*

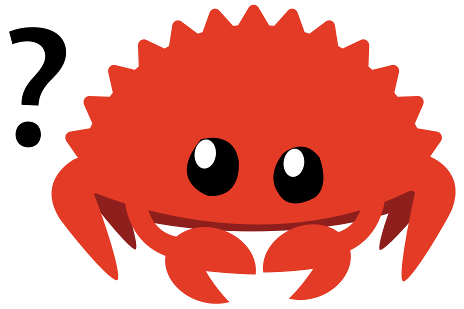
Review: Borrowing Rules

- At any given time, you can have either:
  - One mutable reference (exclusive reference)
  - Any number of immutable references (shared references)
- Memory access through references is guaranteed to be valid

Review Question 1

Why doesn't this compile?

```
fn main() {
    let s = String::from("yo");
    taker(s);
    println!("I *totally* still own {}", s);
}

fn taker(some_string: String) {
    println!("{} is mine now!", some_string);
}
```



Review Question 1

```
error[E0382]: borrow of moved value: `s`
 --> src/main.rs:4:42
  |
2 |     let s = String::from("yo");
  |         - move occurs because `s` has type `String`,
  |           which does not implement the `Copy` trait
3 |     taker(s);
  |           - value moved here
4 |     println!("I *totally* still own {}", s);
  |                                          ^ value borrowed here after move
  |
```

- Looks like `taker` does not still own `s`, after all

Review Question 1

```
note: consider changing this parameter type in function `taker` to borrow
      instead if owning the value isn't necessary
 --> src/main.rs:7:23
  |
7 | fn taker(some_string: String) {
  |    -----              ^^^^^^ this parameter takes ownership of the value
  |    |
  |    in this function
```

- Suggestion from the compiler: Rewrite `taker` to *borrow* `some_string`
- Is it necessary for `taker` to own the value?

Review Question 1 (References Solution)

```rust
fn main() {
    let s = String::from("yo");
    taker(&s);                          // <-- Change to pass a reference to a String
    println!("I *totally* still own {}", s);
}

fn taker(some_string: &String) { // <-- Change to expect a reference to a String
    println!("{} is mine now!", some_string);
}
```

- Let `taker` borrow the value instead of moving it and transferring ownership

Review Question 1 (Alternative Solution)

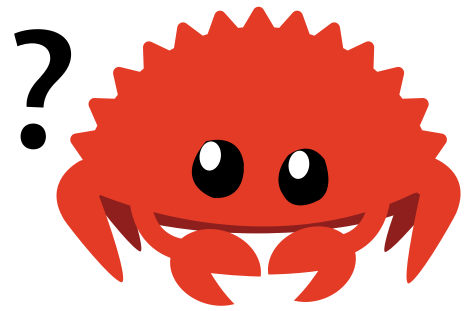
```
help: consider cloning the value if the performance cost is acceptable
  |
3 |     taker(s.clone());
  |            ++++++++
```

- Making a clone (deep copy) allows this compile
  - If `s` represents a large `String`, cloning will be expensive

Review Question 2

Why doesn't this compile?

```rust
fn main() {
    let favorite_computers = Vec::new();
    add_to_list(favorite_computers,
        String::from("Framework Laptop"));
}

fn add_to_list(fav_items: Vec<String>, item: String) {
    fav_items.push(item);
}
```



Review Question 2

```
error[E0596]: cannot borrow `fav_items` as mutable, as it is not declared as mutable
 --> src/main.rs:8:5
  |
8 |     fav_items.push(item);
  |     ^^^^^^^^^ cannot borrow as mutable
  |
help: consider changing this to be mutable
  |
7 | fn add_to_list(mut fav_items: Vec<String>, item: String) {
  |                +++
```

- Missing one `mut` annotation

Review Question 2 (Solution)

```rust
fn cool_guy() {
    let favorite_computers = Vec::new();
    add_to_list(favorite_computers, String::from("Framework Laptop"));
}

//             +++ Add `mut` here
fn add_to_list(mut fav_items: Vec<String>, item: String) {
    fav_items.push(item);
}
```

Review Question 2b

```
fn cool_guy() {
    let favorite_computers = Vec::new();
    add_to_list(favorite_computers, String::from("Framework Laptop"));
    println!("{:?}", favorite_computers); // <-- I want to print this!
}

fn add_to_list(mut fav_items: Vec<String>, item: String) {
    fav_items.push(item);
}
```

- What if we want to print the list?
- `favorite_computers` was moved in the `add_to_list` call
- Same issue as review question 1

Review Question 2b (Solution)

Let's try a mutable reference instead of moving the entire value.

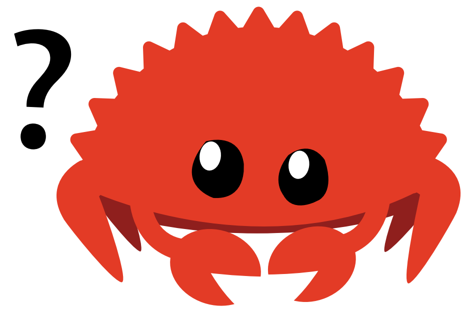
```rust
fn cool_guy() {
    let favorite_computers = Vec::new();
    add_to_list(&mut favorite_computers, String::from("Framework Laptop"));
    println!("{:?}", favorite_computers);
}

fn add_to_list(fav_items: &mut Vec<String>, item: String) {
    fav_items.push(item);
}
```

- This now works as intended!

Review Question 3

Why doesn't this compile?

```
fn x_shouldnt_exist() {
    let mut v = vec![1, 2, 3, 4];
    let x = &v[3];
    v.pop(); // Removes last element in `v`
    println!("{}", x); // What is `x`?
}
```



Review Question 3

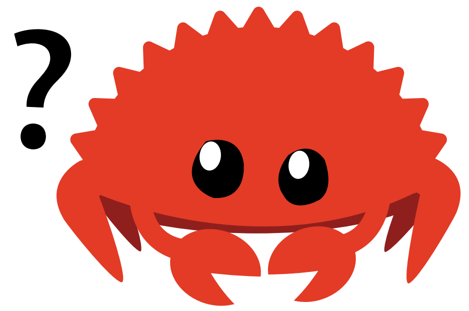
```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:5
  |
3 |     let x = &v[3];
  |              - immutable borrow occurs here
4 |     v.pop(); // Removes last element in `vec`
  |     ^^^^^^^ mutable borrow occurs here
5 |     println!("{}", x); // What is `x`?
  |                    - immutable borrow later used here
```

- We cannot mutably borrow a value with an existing immutable borrow
- In this case, if it were allowed, `x` would point to invalid memory!

Review Question 3b

What about this scenario?

```rust
fn please_dont_move() {
    let mut v = vec![1, 2, 3, 4];
    let x = &v[2];
    v.push(5); // Add an element to the end of `v`
    println!("{}", x); // What is `x`?
}
```



- This time we aren't removing the last element, instead we're adding more elements!
  - Surely nothing happens to `x` this time, right?

Review Question 3b

We still get the same error!

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:5
  |
3 |     let x = &v[2];
  |              - immutable borrow occurs here
4 |     v.push(5);
  |     ^^^^^^^^^ mutable borrow occurs here
5 |     println!("{}", x); // What is `x`?
  |                    - immutable borrow later used here
```

- In this case, what if pushing `5` onto `v` resizes the entire vector?
  - Resizing means allocating new memory, copying over data, then deallocating the old memory
- `x` would no longer point to valid memory!

Structs

Structs

A *struct* is a custom data type that lets you package together and name related values that make up a meaningful group.

```rust
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}
```

- To define a struct, we enter the keyword `struct` and name the entire group
- Within the curly braces, we define *fields*
- Each field is named and has an associated type

Instantiating Structs

We can create an *instance* of a struct using the name of the struct and `key: value` pairs inside curly brackets.

```
fn init_connor() -> Student {
    Student {
        andrew_id: String::from("cjtsui"),
        stress_level: u64::MAX,
        grade: 80,
        attendance: vec![true, false, false, false, false, false, false],
    }
}
```

- You don't have to specify fields in any order
- You *must* define every field of the struct to create an instance

Accessing Fields

We can access fields of a struct using dot notation.

```rust
fn init_connor() -> Student {
    let mut connor = Student {
        andrew_id: String::from("cjtsui"),
        stress_level: u64::MAX,
        grade: 80,
        attendance: vec![true, false, false, false, false, false, false],
    };

    connor.grade = 60; // shh
    println!("{} has grade {}", connor.andrew_id, connor.grade);

    connor
}
```

- Note that the entire instance must be `mut` to modify *any* field

Field Init Shorthand

We can use *field init shorthand* to remove repetitive wording.

```rust
fn init_student(andrew_id: String, grade: u8) -> Student {
    Student {
        andrew_id,
        grade,
        attendance: Vec::new(),
        stress_level: u64::MAX, // 😔
    }
}
```

- We can shorten `andrew_id: andrew_id` to simply `andrew_id`

Struct Update Syntax

There is a shorthand to use values from an existing struct to create a new one.

```rust
fn relax_student(prev_student: Student) -> Student {
    Student {
        stress_level: 0,
        grade: 100,
        ..prev_student
    }
}
```

- Note that this moves the data of the old struct
    - `prev_student` is moved, so we can't use it again (*unless*...)

Tuple Structs

We can created named tuples called "tuple structs".

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let red = Color(255, 0, 0);
    let origin = Point(0, 0, 0);
}
```

- The same as structs, except without named fields
- The same as tuples, except with an associated name

Unit Structs

We can declare "unit structs" as such:

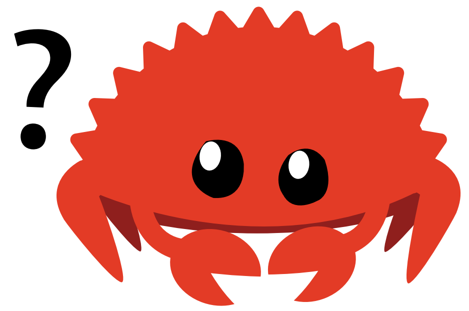
```rust
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

- Structs that have no fields
- Most commonly used as compile-time markers since they are zero-sized types

References in Structs

Can we store references inside structs?

```rust
struct Student {
    andrew_id: &str, // <- &str instead of String
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}
```



Lifetimes Sneak Peek

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:2:16
  |
2 |     andrew_id: &str, // <- &str instead of String
  |                ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct Student<'a> {
2 ~     andrew_id: &'a str, // <- &str instead of String
```

- We can store references in structs, but we need lifetime specifiers
    - We will talk about these in Week 8!

Struct Example

```rust
fn draw_rectangle(x: u32, y: u32, width: u32, height: u32) {}
```

```rust
fn draw_rectangle(rect_tuple: (u32, u32, u32, u32)) {}
```

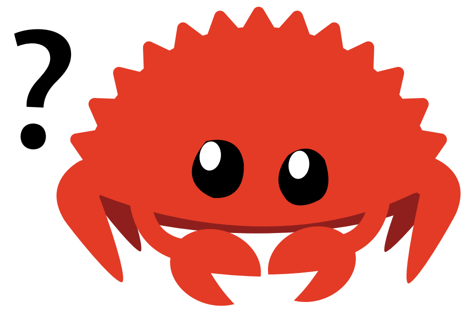
```rust
struct Rectangle {
    x: u32,
    y: u32,
    width: u32,
    height: u32,
}

fn draw_rectangle(rect: Rectangle) {}
```

- Which do you prefer?

Printing Structs

What if we wanted to print these structs for debugging?

```rust
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}

fn main() {
    let connor = init_connor();

    println!("{:?}", connor);
}
```



Printing Structs

We get an error if we try to print something that is not printable:

```rust
fn main() {
    let connor = init_connor();

    println!("{:?}", connor);
}
```

```
error[E0277]: `Student` doesn't implement `Debug`
  --> src/main.rs:11:22
   |
11 |     println!("{:?}", connor);
   |                      ^^^^^^ `Student` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Student`
```

Traits Sneak Peek

What's this all about?

```
error[E0277]: `Student` doesn't implement `Debug`
<-- snip -->
   = help: the trait `Debug` is not implemented for `Student`
```

- More on traits in Week 5!
    - They define shared functionality and behavior between types

Derived Traits

As is often the case, the compiler provides a helpful suggestion.

```
help: consider annotating `Student` with `#[derive(Debug)]`
   |
2  + #[derive(Debug)]
3  | struct Student {
   |
```

- For now, let's just follow the advice blindly

Derived Traits

As a quick overview, derived traits allow us to quickly add functionality to our types.

```rust
#[derive(Debug)]
struct Student {
    andrew_id: String,
    attendance: Vec<bool>,
    grade: u8,
    stress_level: u64,
}
```

- We can *derive* a trait using the `derive` macro
- This will allow us to print this struct!

Derived Traits

We can try again now:

```rust
#[derive(Debug)]
struct Student {
    // <-- snip -->
}

fn main() {
    let connor = init_connor();

    println!("{:?}", connor);
}
```

```
Student { andrew_id: "cjtsui", attendance: [true, false], grade: 80, stress_level: 18446744073709551615 }
```

- We are given a relatively nice output for free!

Methods

Struct Methods

Suppose we wanted to write a function that was only dependent on the data inside a single instance of a struct.

```rust
struct Rectangle {
    x: u32,
    y: u32,
    width: u32,
    height: u32,
}
```

- What if we wanted to get the area of this rectangle?

Struct Methods

Methods are like functions, but their first parameter is always `self`, and they are always defined within the context of a struct.

```rust
struct Rectangle {
    x: u32,
    y: u32,
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

Method Syntax

Let's dive a bit deeper into this:

```rust
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

- We start with an `impl` block for `Rectangle`
- We use `&self` instead of `rectangle: &Rectangle`
    - `&self` is actually syntactic sugar for `self: &Self`
    - In `impl` blocks, `Self` is shorthand for the type being implemented
    - So `&Self` is the same as `&Rectangle`

Calling Methods

We can call a method using dot notation.

```rust
fn main() {
    let rect = Rectangle { x: 0, y: 0, width: 42, height: 98 };

    println!("Area: {}", rect.area());
}
```

- Note that we don't need to pass anything in for `self`

Consuming Methods

What would happen if we didn't borrow with `&self` and instead use `self`?

```rust
impl Rectangle {
    fn area(self) -> u32 {
        self.width * self.height
    }
}
```

```rust
fn main() {
    let rect = Rectangle { width: 42, height: 98 };
    println!("Area: {}", rect.area());
    // println!("Width: {}", rect.width); <-- Cannot do this
}
```

- We take in `self` and "consume" it by taking ownership

`&mut self`

We can take a mutable reference to our struct as well.

```rust
impl Rectangle {
    fn change_width(&mut self, new_width: u32) {
        self.width = new_width;
    }
}

fn main() {
    let mut rect = Rectangle { x: 0, y: 0, width: 42, height: 98 };
    rect.change_width(100);

    println!("{:?}", rect);
}
```

- Follows the same rules for mutable references as before

Associated Functions

We can define an *associated function* in `impl` blocks that do not take `self`.

```rust
impl Rectangle {
    fn create_square(x: u32, y: u32, side_length: u32) -> Self {
        Self { x, y, width: side_length, height: side_length }
    }
}

fn main() {
    let square = Rectangle::create_square(0, 0, 213);
}
```

- A reminder that `Self` is shorthand for `Rectangle` here
- We cannot use dot notation for these functions
    - Instead we use the struct name and the `::` operator

Aside: What About `->`?

```
p1.distance(&p2);
(&p1).distance(&p2); // This is the same!
```

- In C and C++, you use `.` for direct access and `->` for access through a pointer
- Rust instead has ***automatic referencing and dereferencing***
- When you call `object.something()`, Rust will automatically add in the `&`, `&mut`, or `*` so that `object` matches the signature of the method
    - Makes ownership and borrowing more ergonomic

Enums

Enums

- Defines a type with multiple possible *variants*
- Manifestation of the algebraic data type known as the "sum type"
    - Structs are "product types"
- Sum types hold a value that takes on one of several distinct variants.
    - Think of sum types as a value that can be of type A ***or*** B

Enum Definition

IP addresses have two major standards, IPv4 and IPv6.

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

- IP addresses can be *either* IPv4 *or* IPv6
- We can express this concept in code with an enum consisting of V4 and V6 variants
- In general, we enumerate variants of a sum type as fields in an enum

Enum Variants

We can make a value of type `IpAddrKind` as such:

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

- The `::` operator represents a *namespace*
  - `V4` is in the namespace of `IpAddrKind`
- Useful syntax, because we can see both values are of the same type: `IpAddrKind`

Enum Variants

We can define a function that takes an `IpAddrKind`:

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

And call it with any of the variants:

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

Enums vs Structs

At the moment, `IpAddrKind` only encodes the *kind* of address, and the address itself has to be stored elsewhere.

We could do this using structs:

```rust
enum IpAddrKind {
    V4, // IPv4 addresses look like `8.8.8.8`
    V6, // IPv6 addresses look like `2001:4860:4860:0:0:0:0:8888`
}

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

- When we have an `IpAddr` struct, we could check the `kind` field to determine how to interpret the `address` field

Enums Can Hold Data

Instead of using structs to hold data, we can have the enums themselves hold data.

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

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

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

- Much cleaner than before!

Enum Associated Data

Each enum can also hold different types and different amounts of data.

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

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

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

- Cleaner than carrying around a `String` that we need to parse

Aside: `std::net::IpAddr`

The Rust Standard Library actually has its own implementation of `IpAddr`.

```rust
struct Ipv4Addr {
    // --snip--
}

struct Ipv6Addr {
    // --snip--
}

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

Enum Example

Let's take a look at another example of an enum that models data with variants.

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

- `Quit` has no associated data
- `Move` has named fields like a struct
- `Write` includes a single `String`
- `ChangeColor` includes 3 `i32` values

Enums vs Structs

How would this look if we just used structs?

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

- Each of these structs has a separate type
    - We couldn't easily define a function to take in all of these types

Enum Methods

We can define `impl` blocks for enums as well as structs.

```rust
struct Message {
    Write(string),
    // <-- snip -->
}

impl Message {
    fn call(&self) {
        // <-- snip -->
    }
}

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

- `self` holds the value of the enum
    - Same borrowing semantics as with structs

The Option Type

`NULL`

`NULL` is a pointer that does not point to a valid object or value.

> I call it my billion-dollar mistake…
> 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.
> — Tony Hoare, "inventer of `NULL`", 2009

- The issue is not the concept of `NULL`, rather its *implementation*
- We still want a way to express that a value could be *something* **or** *nothing*

The Option Type

The standard library defines an enum `Option<T>`:

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

- We can return either `None` or `Some`, where `Some` contains a value
- The `<T>` is a generic type parameter which means it can hold any type
    - We'll talk about this next week!

The Option Type

Here are some examples of `Option<T>`:

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

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

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

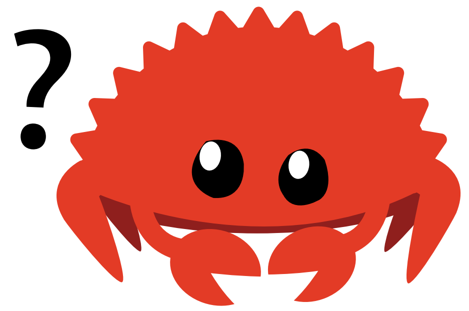
- Rust will infer that `some_number` has type `Option<i32>` and `some_char` has type `Option<char>`
- We still have to annotate `absent_number` with `Option<i32>`

`Option<T>` vs `NULL`

So why is `Option<T>` better than `NULL`? Consider this:

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

let sum = x + y;
```

- What might be wrong with this?



`Option<T>` vs `NULL`

If we try to compile this, we get an compile-time error.

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

let sum = x + y;
```

```
error[E0277]: cannot add `Option<i8>` to `i8`
 --> src/main.rs:6:17
  |
6 |     let sum = x + y;
  |               ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`
```

- Instead of runtime error, we catch the error immediately at compile time!

Working With `Option<T>`

We still need a way to extract the number out of the `Some(5)`.

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

if y.is_none() {
    // do something
} else if y.is_some() {
    // How do we even extract the `5` out???
    // Something like `y.get() + x`???
}
```

- This syntax is also kind of ugly…

Pattern Matching

`match`

Rust has a powerful control flow construct called `match`.

- You can compare a value against a series of patterns
- You can execute code based on which pattern matches
- Patterns can be made up of literal values, variable names, wildcards, etc.

Pattern Matching

Here's an example of a coin sorting function that returns the value of the coin.

```rust
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,
    }
}
```

Pattern Matching

Let's break this down:

```
match coin {
    Coin::Penny => 1,
    Coin::Nickel => 5,
    Coin::Dime => 10,
    Coin::Quarter => 25,
}
```

- First we write `match`, followed by an expression (in this case `coin`)
- Similar to `if` branch, but the expression does not need to be a `bool`
- Each arm has a pattern, followed by `=>`, followed by another expression
  - The patterns here are the `Coin` enum variants
  - The expressions here are just the values of each coin

Pattern Matching

Here's another similar example.

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

- The `match` arms can be any valid expression, including code blocks!

Binding Patterns: Quarters

Patterns can bind to specific parts of the values that match the pattern.

```rust
#[derive(Debug)] // Allows us to print `UsState`
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState), // Quarters have states on them
}
```

Binding Patterns: Quarters

```rust
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
        }
    }
}
```

- We bind the variable `state` to the `UsState` that the `Quarter` variant holds!

Binding Patterns: `Option<T>`

Let's revisit the example from before.

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

let sum = match y {
    Some(num) => Some(x + num),
    //    ^^^ `num` binds to 5
    None => None,
};

println!("{:?}", sum); // Prints "Some(10)"
```

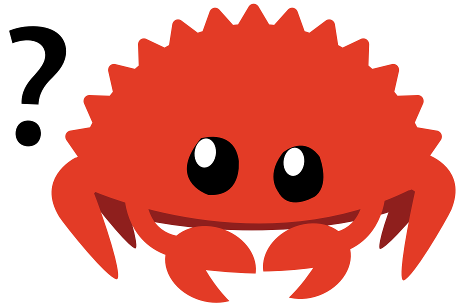
- Clean, and we can access the value in the `Some` variant easily

Matches Are Exhaustive

The `match` patterns must cover all possible values that the matched expression may take.

What happens when we miss a case?

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

let sum = match y {
    Some(num) => x + num,
};
```



Matches Are Exhaustive

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

let sum = match y {
    Some(num) => x + num,
};
```

```
error[E0004]: non-exhaustive patterns: `None` not covered
   --> src/main.rs:6:21
    |
6   |     let sum = match y {
    |                     ^ pattern `None` not covered
```

- Forces us to explicitly handle the `None` case
- Protecting us from the billion-dollar mistake!

Catch-all Pattern

Sometimes we don't need to do something special for every case, and can instead have a fallback case.

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

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

- `other` matches anything not covered by previous patterns

`_` Pattern

If we don't need the matched value, we can use `_` instead.

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

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

- `_` matches anything, but it doesn't bind the value

Concise Control Flow with `if let`

Sometimes we just want to match against 1 pattern while ignoring the rest.

`if let` provides a more concise way to do this:

```rust
if let Coin::Penny = coin {
    println!("Lucky penny!");
}
```

- Works with `else if let <pattern> = <expr>` and `else` as well

`if let` Example

Here's another example of the same program written 2 different ways:

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

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

Pattern Matching

Pattern Matching is an incredibly powerful tool.

- Gives you more utilities for managing a program's control flow
- Allows you to you quickly and cleanly case on structures, typically enums
- Very useful for compilers and parsers
- Rust has many more patterns than we have time to cover!
    - Read Chapter 18 of the Rust Book to find out more!
        - *Will take less than 20 minutes*

Homework 3

- This is the first homework where you will need to actually synthesize code!
- You have been tasked with implementing two types of Pokemon:
  - `Charmander` struct
  - `Eevee` struct that can evolve into `EvolvedEevee`
    - `EvolvedEevee` is an enum representing different evolutions
- We *highly* recommend reading Chapter 18 of the Rust book if you have time!

Next Lecture: Standard Collections and Generics

- Thanks for coming!