Structs in Rust

Lecture 9

Slides: Neil Kaushikkar

Goals For Today

- Review borrowing rules
- Defining & instantiating **structs**
- Constructor patterns for **structs**
- Defining functions to operate on **structs**
- Automatically deriving functionality on custom types

References

- An ampersand (&) represents a *reference*
- Allows you to refer to some value *without taking ownership* of it
- We call the action of creating a reference *borrowing*
- At any given time, you can have either:
  - **one** mutable reference using **&mut** or…
  - An **infinite** number of immutable references using &
- A mutable reference must be a reference to a mutable variable
  - You cannot make a mutable reference to an immutable variable

Reference:

- https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

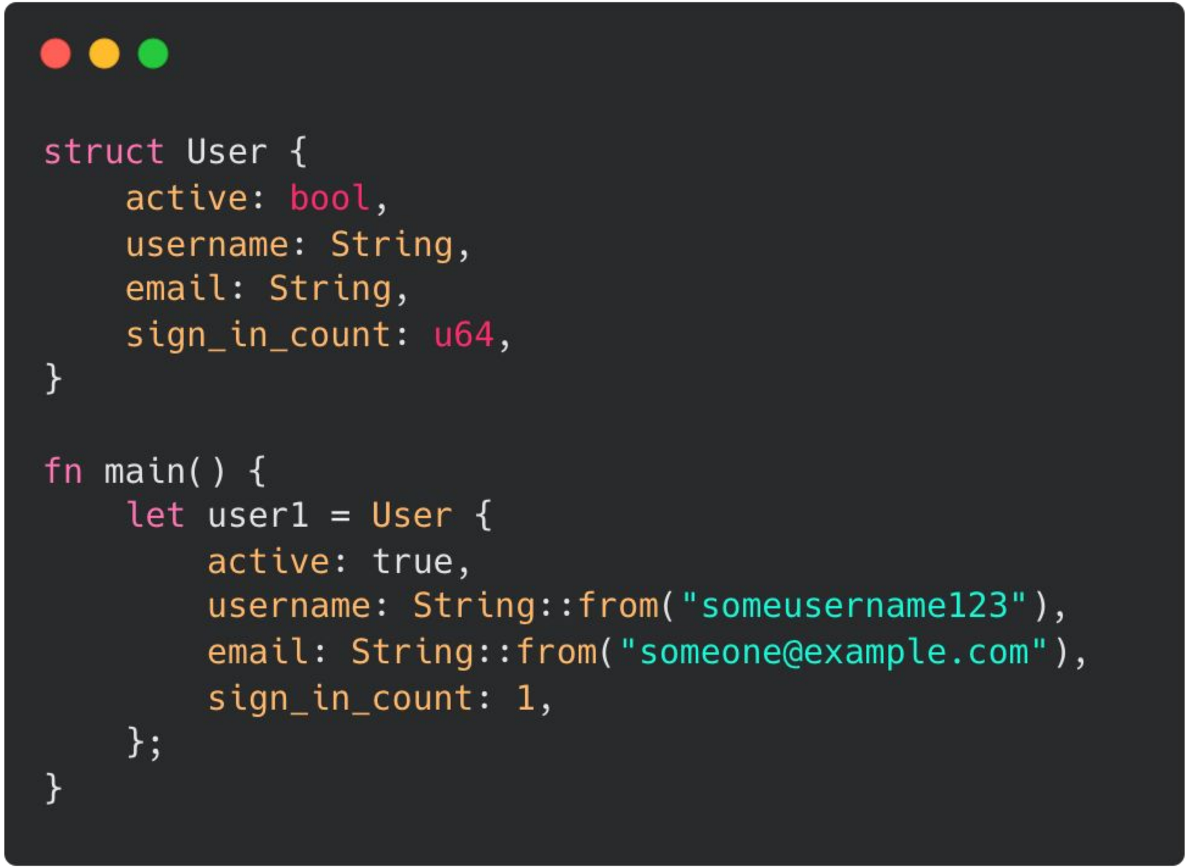
What Are Structs?

- **structs** are custom made data types that hold multiple values
  - Similar to tuples, **structs** had hold data of multiple types without issue
  - Unlike tuples, you must assign a variable to the data you are storing
- **structs** are functionally similar to classes in OOP languages
- Different instances of **structs** of the same type do not share variables
  - Just as in all other OOP languages you've used
- You can declare functions in the context of an instance of a **struct** using **traits** or as methods with an **impl** block

Reference:

- https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Defining & Instantiating Structs

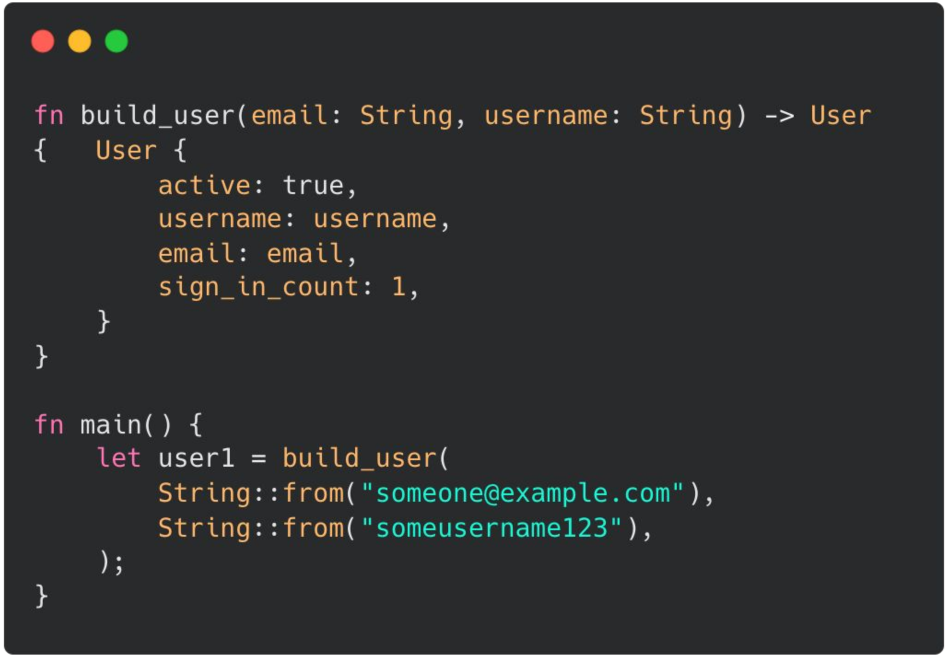


```rust
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

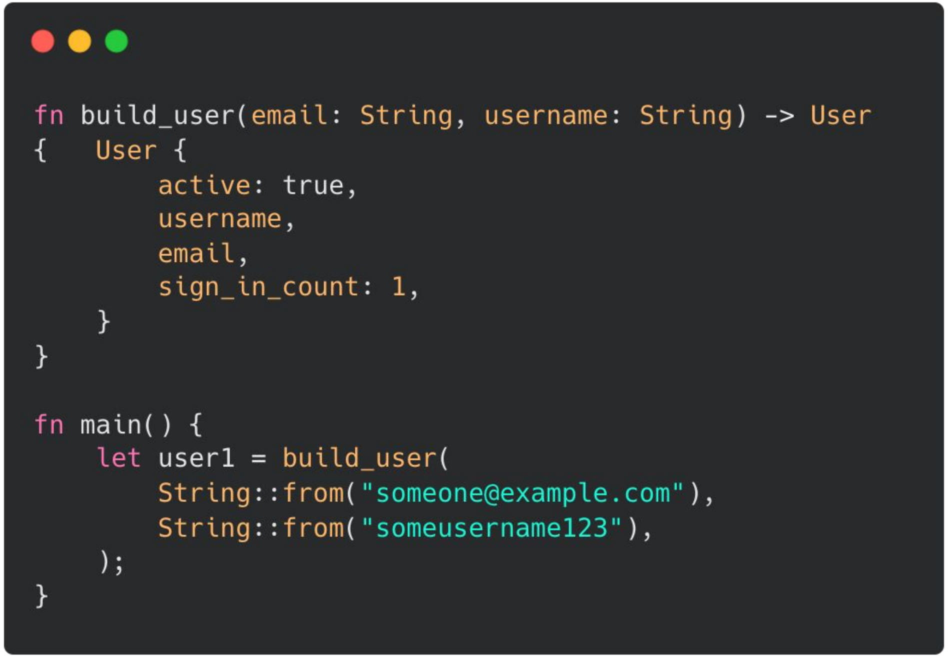
Reference:

- https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Defining & Instantiating Structs (cont.)



```
fn build_user(email: String, username: String) -> User
{   User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}

fn main() {
    let user1 = build_user(
        String::from("someone@example.com"),
        String::from("someusername123"),
    );
}
```



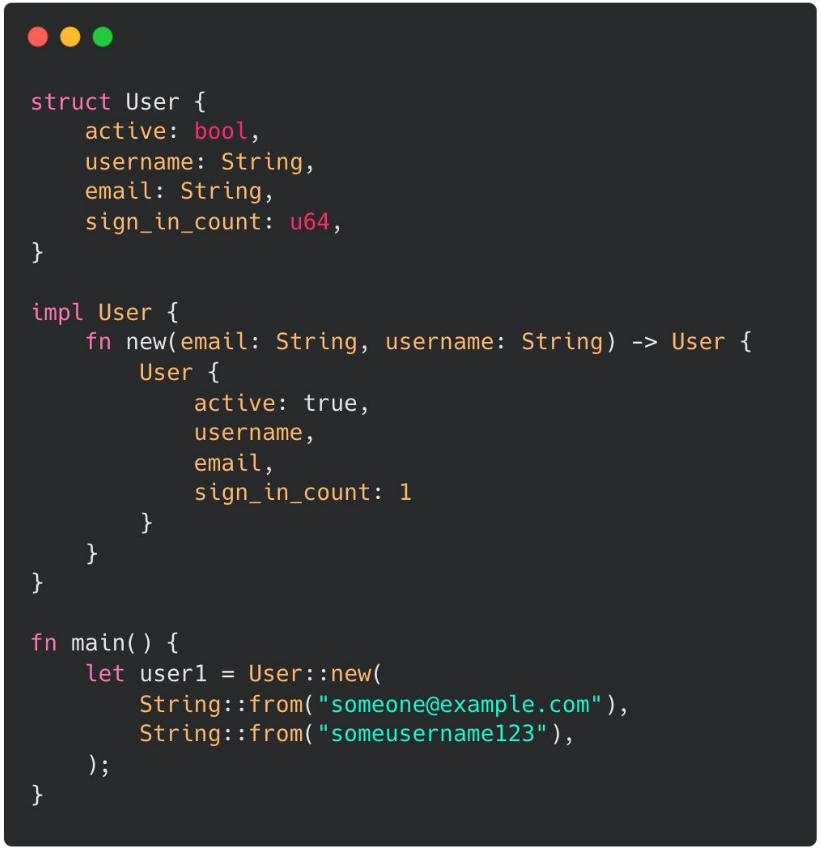
```
fn build_user(email: String, username: String) -> User
{   User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}

fn main() {
    let user1 = build_user(
        String::from("someone@example.com"),
        String::from("someusername123"),
    );
}
```

Field initialization: the fields inside the **struct** are the same name as the variables they are being initialized to

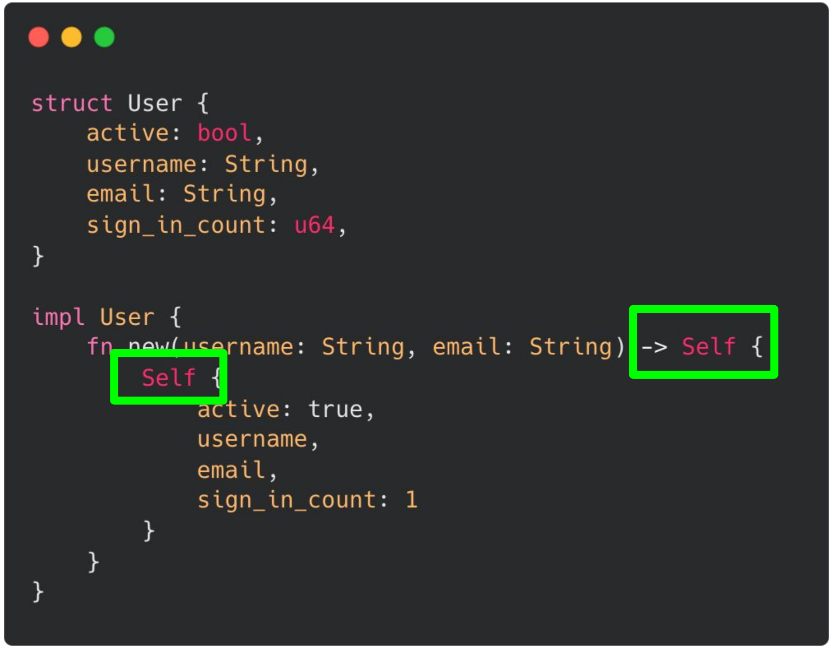
Struct Constructor Pattern

- We want to define a function in the context of the **struct** we defined
  - How do we go about making a struct owned function?
- The **impl** keyword
  - Indicates the implementation of a function for a struct
  - Can have multiple **impl** blocks over a particular type
    - This is important when we cover **trait**s (think interfaces in Java)!
- We can define a constructor that is in the namespace of the **struct** and also returns an instance of the **struct** when called
  - Use the scope resolution operator (the **::** symbol)
  - Syntax: **TypeName::constructor_name()**
    - ex: **User::new()** or **String::new()**

Struct Constructor Pattern



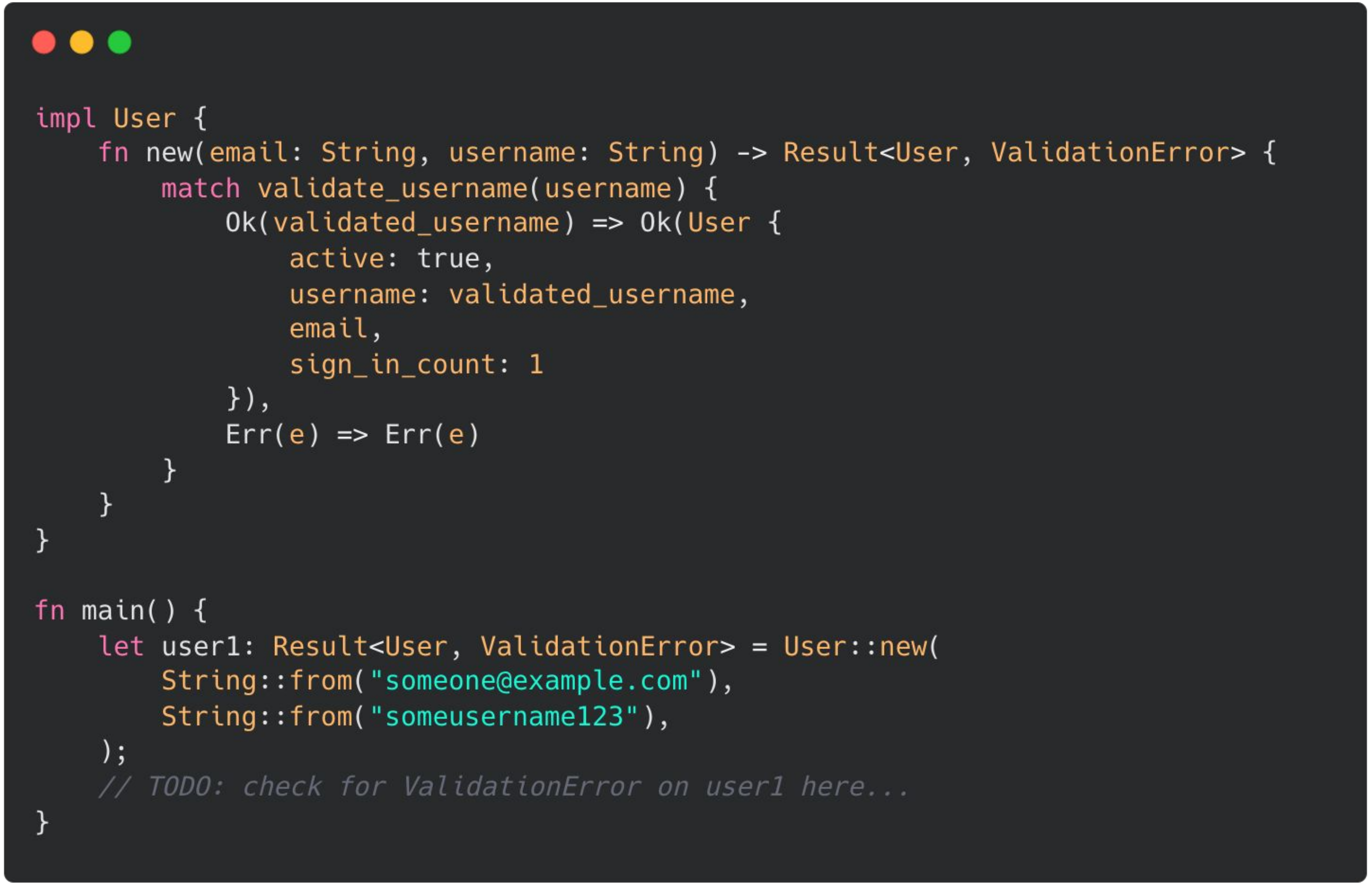
```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

impl User {
    fn new(email: String, username: String) -> User {
        User {
            active: true,
            username,
            email,
            sign_in_count: 1
        }
    }
}

fn main() {
    let user1 = User::new(
        String::from("someone@example.com"),
        String::from("someusername123"),
    );
}
```



```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

impl User {
    fn new(username: String, email: String) -> Self {
        Self {
            active: true,
            username,
            email,
            sign_in_count: 1
        }
    }
}
```

IMPORTANT: Self (with a capital S) refers to the type name

Constructors Can Fail! Remember Result?

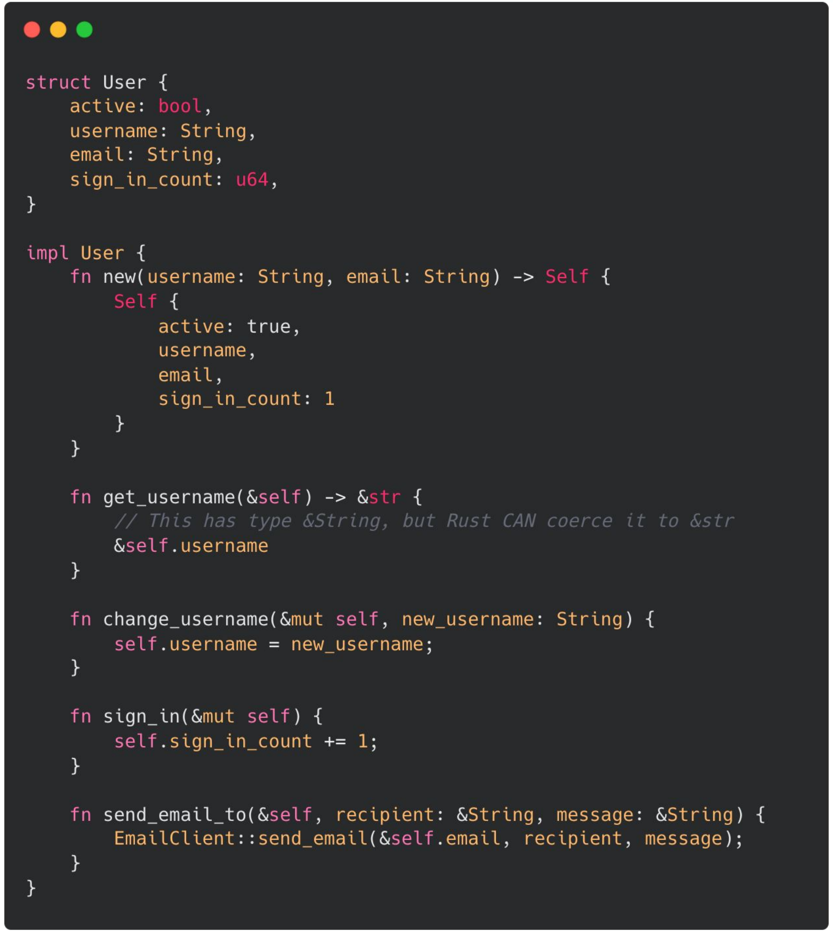


```
impl User {
    fn new(email: String, username: String) -> Result<User, ValidationError> {
        match validate_username(username) {
            Ok(validated_username) => Ok(User {
                active: true,
                username: validated_username,
                email,
                sign_in_count: 1
            }),
            Err(e) => Err(e)
        }
    }
}

fn main() {
    let user1: Result<User, ValidationError> = User::new(
        String::from("someone@example.com"),
        String::from("someusername123"),
    );
    // TODO: check for ValidationError on user1 here...
}
```

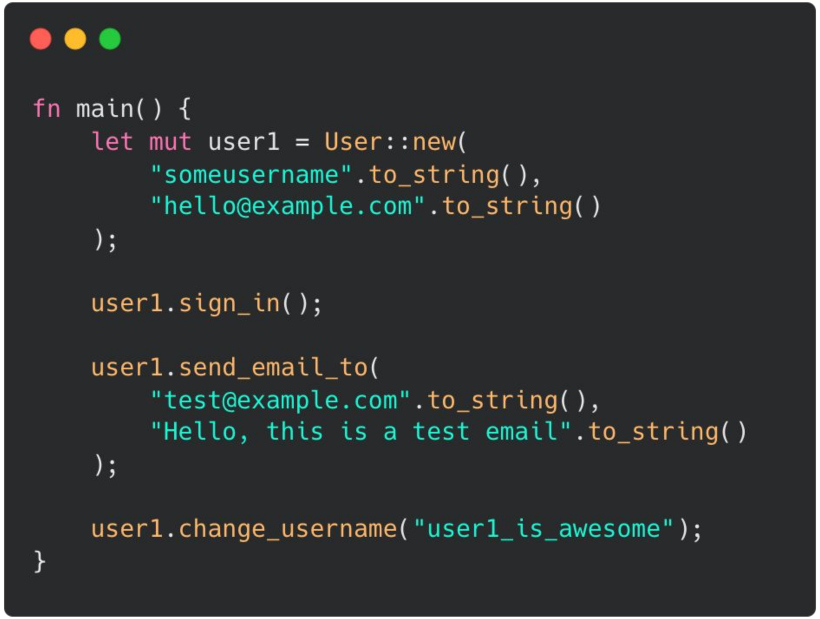
Defining Functions for Instances of Structs

- Recall, **struct** instances retain ownership over the data within
- If we want to call a function on the **struct** data, we need to borrow that data!
- Use the **self** keyword to refer to the current instance of a struct that the method is being called upon (similar to syntax for class methods in Python)
  - We want to ***BORROW*** the data from a **struct** instance in the function, so either immutably borrow with **&self** OR mutably borrow with **&mut self**
  - The 1st parameter of the function should be some borrow to **self**
  - There are use cases for not borrowing self, but those are very rare
  - Use dot notation to access the fields in the **struct**

Defining Functions for Instances of Structs



```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}

impl User {
    fn new(username: String, email: String) -> Self {
        Self {
            active: true,
            username,
            email,
            sign_in_count: 1
        }
    }

    fn get_username(&self) -> &str {
        // This has type &String, but Rust CAN coerce it to &str
        &self.username
    }

    fn change_username(&mut self, new_username: String) {
        self.username = new_username;
    }

    fn sign_in(&mut self) {
        self.sign_in_count += 1;
    }

    fn send_email_to(&self, recipient: &String, message: &String) {
        EmailClient::send_email(&self.email, recipient, message);
    }
}
```



```
fn main() {
    let mut user1 = User::new(
        "someusername".to_string(),
        "hello@example.com".to_string()
    );

    user1.sign_in();

    user1.send_email_to(
        "test@example.com".to_string(),
        "Hello, this is a test email".to_string()
    );

    user1.change_username("user1_is_awesome");
}
```

If you want to call a method that <u>mutably borrows</u> self on an instance of your struct, you must declare the instance to be mutable!!!

Self vs self

- **Self** – the name of the type in the context of an **impl** block
- **self** – the current instance of the type in the context of a function
- There is a subtle but VERY important distinction

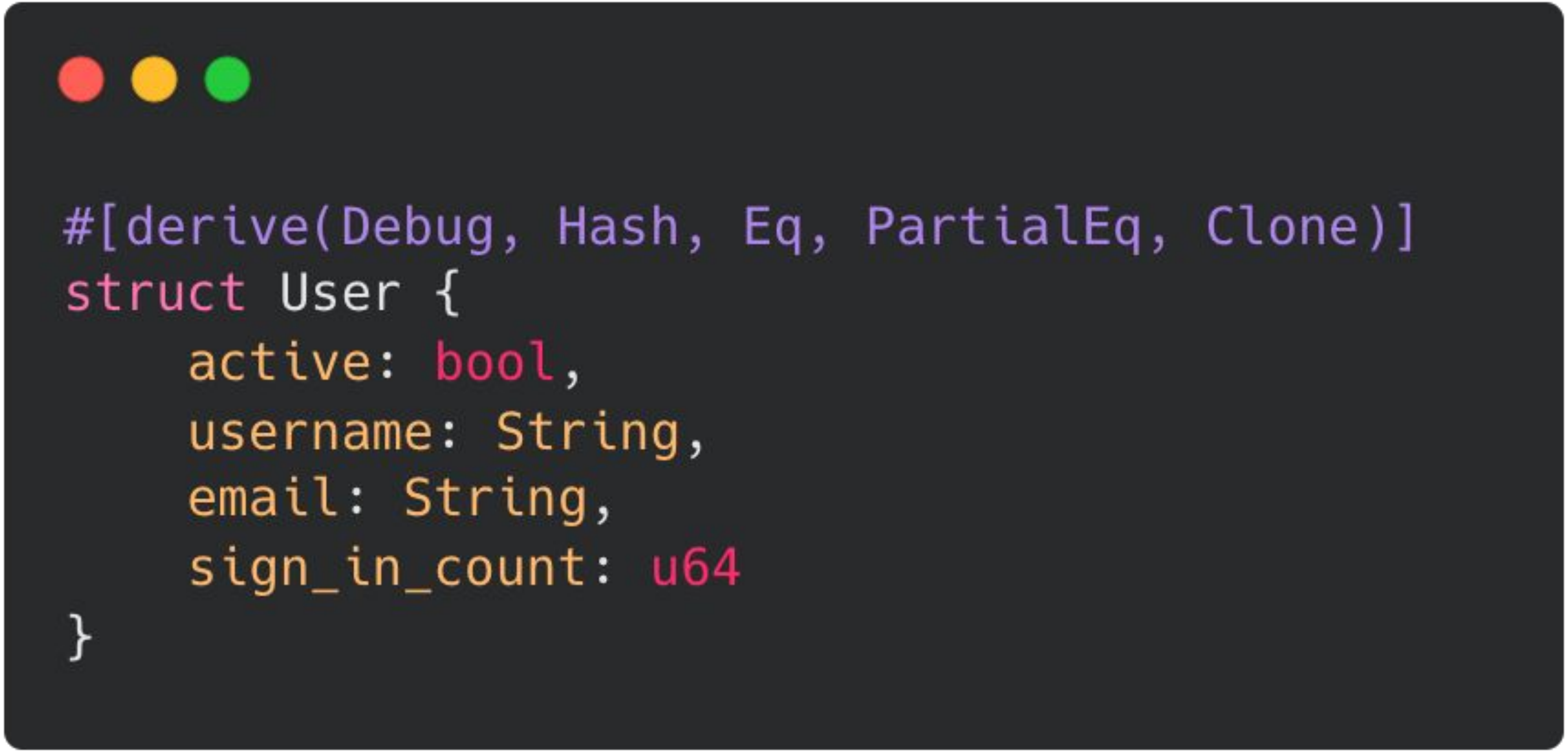
Deriving Functionality

- Automatically define certain behaviors on our **struct**s with built-in **traits**
  - Comparison traits: **Eq**, **PartialEq**, **Ord**, **PartialOrd**
  - **Clone**, to create a deep copy of your **struct** instance using **.clone()**
  - **Hash**, to compute a hash from your type (useful when you want to use your struct as a key to a **HashMap** or **HashSet**)
  - **Default**, to create an empty instance of a data type
  - **Debug**, to format a value using the **{:?}** formatter i.e. **println("{:?}", ...);**

Reference:

- https://doc.rust-lang.org/rust-by-example/trait/derive.html

Deriving Functionality



```
#[derive(Debug, Hash, Eq, PartialEq, Clone)]
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64
}
```

Reference:

- https://doc.rust-lang.org/rust-by-example/trait/derive.html

That’s All Folks!