Lesson 7

Rust Errors etc.

Rust distinguishes between recoverable and un recoverable errors, we will handle these errors differently

1. Recoverable errors
   With these we will probably want to notify the user, but there is a clear structured way to proceed. We handle these using the type `Result<T, E>`
   `enum Result<T, E> { Ok(T), Err(E), }`
   Many standard operations will return a type of `Result` to allow for an error
   A good pattern to handle this is to use matching

```
      let f = File::open("foo.txt"); // returns a Result
      let f = match f {
              Ok(file) => file,
              Err(error) => // some error. handling
              }
```

You can propagate the error back up the stack if that is appropriate

```
  let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
};
```

The ? Operator

A shortcut for propagating errors is to use the ? operator

```
let mut f = File::open("hello.txt")?;
```

The `?` placed after a `Result` value works in almost the same way as the `match` expressions.
If the value of the `Result` is an `Ok`, the value inside the `Ok` will get returned from this expression, and the program will continue.
If the value is an `Err`, the `Err` will be returned from the whole function as if we had used the `return` keyword so the error value gets propagated to the calling code.

2. Un recoverable errors
   In this case, we have probably come across a bug in the code and there is no safe way to proceed. We handle these with the `panic!` macro
   For a discussion of when to use `panic!` see the docs
   When the `panic!` macro executes, your program will print an error message, unwind and clean up the stack, and then quit.
   We can specify the error message to be produced as follows

```
panic!("Bug found ....");
```

Traits examples in Solana

A trait tells the Rust compiler about the functionality of a generic type and defines shared behaviour. Traits are often called interfaces in other languages, although with some differences.

A `trait` is a collection of methods defined for an unknown type: `Self`. Trait methods can access other trait methods.

```
  trait Details {
    fn get_owner(&self) -> &Pubkey;
    fn get_admin(&self) -> &Pubkey;
    fn set_owner(&self) -> &Pubkey;
    fn set_admin(&self) -> &Pubkey;
    fn get_amount(&self) -> u64;
    fn set_amount(&self);
    fn print_details(&self) {
            println!("Owner is {:?}", self.get_owner())
            println!("Admin is {:?}", self.set_admin())
            println!("Amount is {}", self.get_amount())
    }
}
```

If certain methods are frequently reused between structs, it makes sense to define a trait.

Implementations

Implementations are defined with the `impl` keyword and contain functions that belong to an instance of a type, statically, or to an instance that is being implemented.

For a struct `AccountA` representing the layout of data on chain:

```

pub struct AccountA {
        pub admin: Pubkey,
        pub owner: Pubkey,
        pub amount: u64,
}
```

To implement previously defined abstract functions of the trait `Details`:

```
  impl Details for AccountA {
        fn get_owner(&self) -> &Pubkey {
                        &self.owner
        }
        fn get_admin(&self) -> &Pubkey {
                        &self.admin
        }
        ...
}
```

In Solana this is often used for instructions relating to serialisation of the data stored under a given account:

```
  impl AccountA {
        fn unpack(input: &[u8]) -> Result<Self, ProgramError> {
                ...
        }

        fn pack(&self, dst: &mut [u8]) {
                ...
        }
}
```

Then in the processor bare bytes can be converted to a usable struct as simply as

```
  let mut account_temp = AccountA::unpack(ADDRESS)?;
```

Solana Accounts

See Account Model
and Cookbook

Solana separates code from data, all programs are stateless so any data they need must be passed in from the outside.
All accounts are owned by programs.
Some accounts are owned by System program, and some can be owned by your own program.

Accounts are both used by and owned by programs, and a single program can own many different accounts.

Account Fields

- key
- isSigner
- isWritable
- lamports
- data
- owner
- executable
- rent_epoch

Owner versus holder

The *owner* is not the person who own the private key of the account ,they are called the *holder*.
The holder is able to transfer the balance from the account.

The owner in has the right to amend the data of any account.
In Solana, system program is set to be the owner of each account by default.
So for example if you create an account in Solana in order to store some SOL you would be the holder but the System program would be the owner.

Difference to Ethereum

On Ethereum, only smart contracts have storage and naturally they have full control over that storage.
On Solana, *any* account can store state but the storage for smart contracts is only used to store the immutable byte code.
On Solana, the state of a smart contract is actually completely stored in other accounts.

- Accounts can store arbitrary kinds of data as well as SOL.
- Accounts also have metadata which describes who is allowed to access its data and how long the account can live for.
- Anyone can read or credit an account, but only the account owner can debit it or modify its data.

- Accounts are created by simply generating a new keypair and registering its public key with the System Program.
- Each account is identified by its unique address, the same as in a wallet.

There are 3 kinds of accounts on Solana:

- Data accounts that store data
- Program accounts that store executable programs
- Native accounts that indicate native programs on Solana such as System, Stake, and Vote

Within data accounts, there are 2 types:

- System owned accounts
- PDA (Program Derived Address) accounts

Every account in Sealevel has a specified owner.
Since accounts can be created by simply receiving lamports, each account must be assigned a default owner when created.
The default owner in Sealevel is called the "System Program".
The System Program is primarily responsible for account creation and lamport transfers.

Program Derived Address (PDA)

A Program Derived Address is an account that's designed to be controlled by a specific program.
With PDAs, programs can programatically sign for certain addresses without needing a private key.
At the same time, PDAs ensure that no external user could also generate a valid signature for the same address.

It may be helpful to consider that PDAs are not technically `created`, but rather `found`.

Accounts part 2

On Solana blockchain everything is an account and they are pages in the shared memory.

Account type	Executable	Writable
Program	✅	❌
Data keypair owned	❌	✅
Data program owned (PDA)	❌	✅

Accounts maintain:

- arbitrary data that persists beyond the lifetime of a program
- balance in SOL proportional to the storage size
- metadata relating to permissions

Accounts have an "owner" field which is the Public Key of the program that governs the state transitions for the account.
Programs are accounts which store executable byte code and have no state. They rely on the data vector in the Accounts assigned to them for state transitions. All programs are owned by BPF Loader (BPFLoaderUpgradeab1e11111111111111111111111).

A PDA has no private key.

1. Programs can only change the data of accounts they own.
2. Programs can only debit accounts they own.
3. Any program can credit any account.
4. Any program can read any account.

By default, all accounts start as owned by the System Program.

1. System Program is the only program that can assign account ownership.
2. System Program is the only program that can allocate zero-initialized data.
3. Assignment of account ownership can only occur once in the lifetime of an account.

A user-defined program is loaded by the loader program. The loader program is able to mark the data in the accounts as executable. The user performs the following transactions to load a custom program:

1. Create a new public key.
2. Transfer coin to the key.
3. Tell System Program to allocate memory.
4. Tell System Program to assign the account to the Loader.
5. Upload the bytecode into the memory in pieces.
6. Tell Loader program to mark the memory as executable.

> At this point, the loader verifies the bytecode, and the account to which the bytecode is loaded into can be used as an executable program. New Accounts can be marked as owned by the user-defined program.
>
> The key insight here is that programs are code, and within our key-value store, there exists some subset of keys that the program and only that program has write access.

Programming Model

See Docs
An app interacts with a Solana cluster by sending it transactions with one or more instructions.
The Solana runtime passes those instructions to programs deployed by app developers beforehand.
An instruction might, for example, tell a program to transfer lamports from one account to another or create an interactive contract that governs how lamports are transferred. Instructions are executed sequentially and atomically for each transaction.
If any instruction is invalid, all account changes in the transaction are discarded.

Sealevel — Parallel Processing of smart contracts

From Introduction

On Solana, each instruction tells the VM which accounts it wants to read and write ahead of time. This is the root of the optimisations to the VM.

1. Sort millions of pending transactions.
2. Schedule all the non-overlapping transactions in parallel.

(Aside see Ethereum new transaction type and access list)

Signature checking

The parallel nature of Sealevel means that signatures can be checked quickly using GPUs.

Transaction structure

Transactions specify a collection of instructions
Each instruction contains the program, program instruction, and a list of accounts the transaction wants to read and write.

- Signatures — this is a list of ed25519 curve signatures of the message hash, where the "message" is made up of metadata and "instructions".
- Metadata, the message has a header, which includes 3 fields describing how many accounts will sign the payload, how many won't, and how many are read-only.
- Instructions, this contains three main pieces of information:
    - a) set of accounts being used and whether each one is a signer and/or writable,
    - b) a program ID which references the location of the code you will be calling
    - c) a buffer with some data in it, which functions as calldata.

Msg Macro

See docs
msg! can be used to output text to the console in Solana

Breaking program into modules

Modules give code structure by introducing a hierarchy similar to the file tree. Each module has a different purpose, and functionality can be restricted.
At the root module multiple modules are compiled into a unit called a crate.
Crate is synonymous with a 'library' or 'package' in other languages.

Modules are defined using the `mod` keyword and often contained in `lib.rs`.
A Common pattern seen throughout program implementations is:

```
  // lib.rs
pub mod entrypoint;
pub mod error;
pub mod instruction;
pub mod processor;
pub mod state;
```

Where:

- `lib.rs`: registering modules
- `entrypoint.rs`: entrypoint to the program
- `instruction.rs`: (de)serialisation of instruction data
- `processor.rs`: program logic
- `state.rs`: (de)serialisation of accounts' state
- `error.rs`: program specific errors

Development Tools

Solana playground

See playground

Local Validator

See Docs
You need to have installed the solana-cli, see lessons 3 and 4
Run
`solana-test-validator` to start it.

Anchor Framework

We will look at this next week, once we have the covered the basics of programs on Solana.