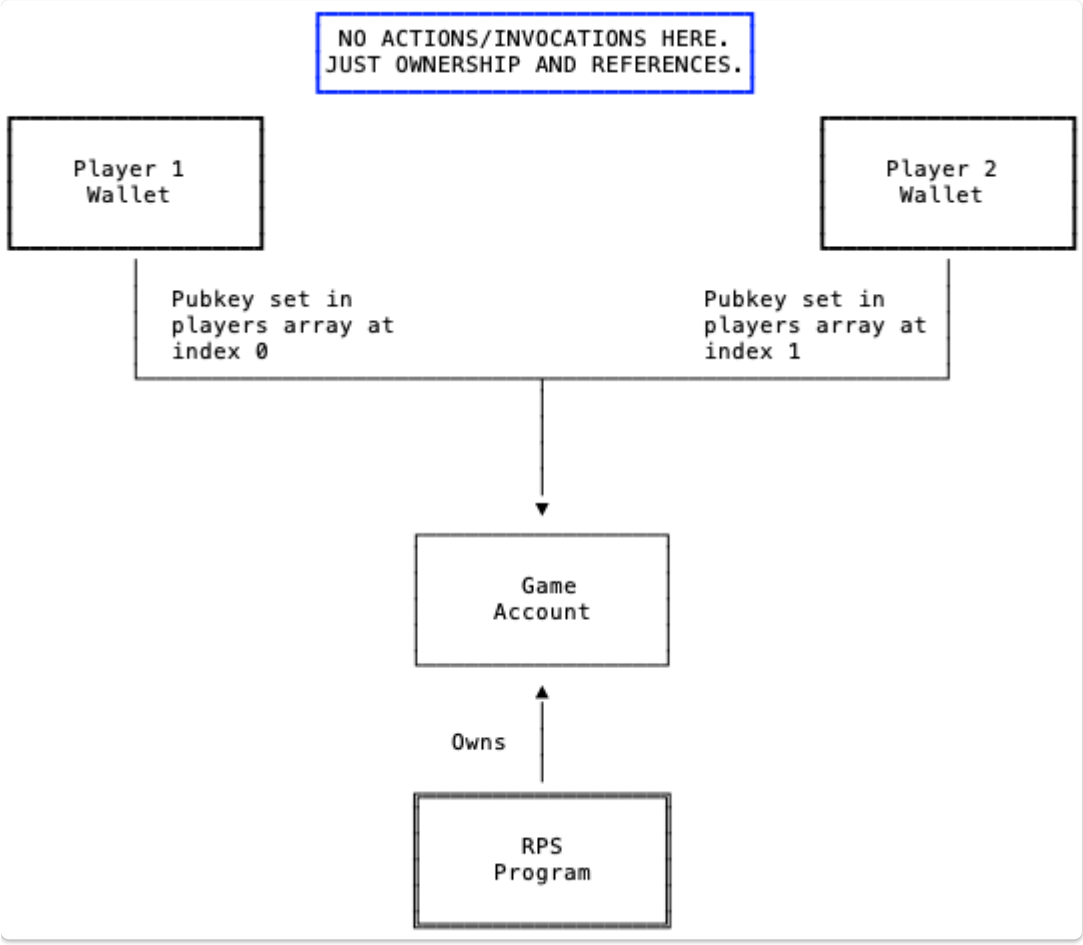
Lesson 16

Anchor examples

Rock Paper Scissors example

Program that implements rock-paper-scissors game.

Account diagram:



Flow:

1. Create game account: `new_game`

```
  #[account]
pub struct Game {
    players: [Pubkey; 2],
    hashed_hand: [[u8; 32]; 2],
    hash_submitted: [bool; 2],
    hand: [Hand; 2],
    hand_submitted: [bool; 2],
    winner: String,
}
```

This account will store the state necessary for a single game such as:

- players public keys (array)
- hashes of hands (array)
- hands with salt (array)
- hands enum (array)
- flag for providing hash (array)
- winner

2. Player *X* submits hash containing their upcoming hand
3. Player *Y* submits hash containing their upcoming hand
4. Player *X* submits string containing their hand and the salt
5. Player *Y* submits string containing their hand and the salt

There is no required order for who submits hash first, but hands/salt strings can't be submitted until both hashes have been provided.

Versioned Transactions

See Proposal

Because of restrictions on the size of messages passed around the network, typically we cannot include more than 35 accounts or equivalent in a single transaction.

A Proposed solution is to

1. Allow address look up tables
2. Add a new transaction format to handle these.

After addresses are stored on-chain in an address lookup table account, they may be succinctly referenced in a transaction using a 1-byte u8 index rather than a full 32-byte address. This will require a new transaction format to make use of these succinct references as well as runtime handling for looking up and loading addresses from the on-chain lookup tables.

Address lookup tables must be rent-exempt when initialized and after each time new addresses are appended. Lookup tables can either be extended from an on-chain buffered list of addresses or directly by appending addresses through instruction data.

Once an address lookup table is no longer needed, it can be deactivated and closed to have its rent balance reclaimed.

Versioned Transactions

In order to support address table lookups, the structure of serialized transactions must be modified. The new transaction format should not affect transaction processing in the Solana program runtime, invoked programs will be unaware of which transaction format was used.

The message header encodes `num_required_signatures` as a `u8`. Since the upper bit of the `u8` will never be set for a valid transaction, we can enable it to denote whether a transaction should be decoded with the versioned format or not.

Limitations

- Max of 256 unique accounts may be loaded by a transaction because `u8` is used by compiled instructions to index into transaction message `account_keys`.
- Address lookup tables can hold up to 256 entries because lookup table indexes are also `u8`.
- Transaction signers may not be loaded through an address lookup table, the full address of each signer must be serialized in the transaction. This ensures that the performance of transaction signature checks is not affected.
- Hardware wallets will not be able to display details about accounts referenced through address lookup tables due to inability to verify on-chain data.
- Only single level address lookup tables can be used. Recursive lookups will not be supported.

Token-2022 Program

The Token-2022 program is a program that includes the functionality of the Token Program, but adding new functionality with new instructions.

Data layout in accounts

The start of the data layout is the same for the Token-2022 and the Token Program, new fields are required for Token-2022, so these are introduced as extensions to the layout.

Extensions

See docs

Mint Account

- Transfer fees
  With Token-2022, it's possible to configure a transfer fee on a mint so that fees are assessed at the protocol level. On every transfer, some amount is withheld on the recipient account, untouchable by the recipient. These tokens can be withheld by a separate authority on the mint.
- Closing mint
  In Token-2022, it is possible to close mints by initializing the `MintCloseAuthority` extension before initializing the mint.
- Interest-bearing tokens
  With the Token-2022 extension model, however, we have the possibility to change how the UI amount of tokens are represented. Using the `InterestBearingMint` extension and the `amount_to_ui_amount` instruction, you can set an interest rate on your token and fetch its amount with interest at any time.
- Non-transferable tokens
  To accompany immutably owned token accounts, the `NonTransferable` mint extension allows for "soul-bound" tokens that cannot be moved to any other entity. For example, this extension is perfect for achievements that can only belong to one person or account.

Token Account

- Memo required on incoming transfers
  By enabling required memo transfers on your token account, the program enforces that all incoming transfers must have an accompanying memo instruction right before the transfer instruction.
- Immutable ownership
  Token-2022 includes the `ImmutableOwner` extension, which makes it impossible to reassign ownership of an account. The Associated Token Account program always uses this extension when creating accounts.
- Default account state
  For example a mint creator may use the `DefaultAccountState` extension, which can force

all new token accounts to be frozen. This way, users must eventually interact with some service to unfreeze their account and use tokens.

- Permanent Delegate
  With Token-2022, it's possible to specify a permanent account delegate for a mint. This authority has unlimited delegate privileges over any account for that mint, meaning that it can burn or transfer any amount of tokens.

Other Languages

Seahorse

Docs

Seahorse lets you write Solana programs in Python. It is a community-led project built on Anchor.

Install with

```
cargo install seahorse-lang
```

It is also available in the Solana playground.

Tokio

Tokio is an asynchronous runtime for the Rust programming language.

It provides

- A multi-threaded runtime for executing asynchronous code.
- An asynchronous version of the standard library.
- A large ecosystem of libraries.

Adding Tokio

You can add Tokio as a dependency to your project in the cargo.toml file
`tokio = { version = "1", features = ["full"] }`

Hello World example

```language-rust
#[tokio::main]
async fn main() {
    // Calling `say_world()` does not execute the body of `say_world()`.
    let op = say_world();

    // This println! comes first
    println!("hello");

    // Calling `.await` on `op` starts executing `say_world`.
    op.await;
}
```

Tokio is a dependency in Solana, used for the network

Course Review

Lesson 1

Decentralisation / Blockchain theory / Cryptography

Lesson 2

Cryptography / Solana Theory

Lesson 3

Solana Command Line / Rust

Lesson 4

Solana Community / Rust

Lesson 5

Rust - Ownership and Borrowing

Lesson 6

Solana Concepts / Intro to Development

Lesson 7

Solana Accounts / Rust Errors

Lesson 8

Intro to DeFi / Token Program

Lesson 9

Solana programs / Upgrading / PDAs

Lesson 10

PDAs continued

Lesson 11

Web3 introduction

Lesson 12

Cross Program Invocation /Anchor Introduction

Lesson 13

Anchor / Solana Program Library

Lesson 14

NFTs / DeFi

Lesson 15

Security / Confidential tokens

Lesson 16

Review / Token-2022