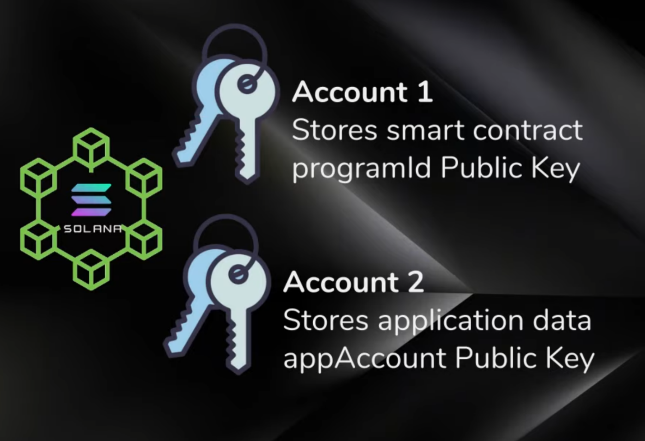
Solana uses proof of history for consensus and proof of stake

At high level, base element in Solana is an Account. Data living in blockchain is stored in the Account.

Programs (smart contracts) are an account with an executable flag set and change the data in other accounts.

mint account holds the supply, has an authority to 'mint' tokens. spl-token creae-token actually creates a mint account. a token account, is created passing a mint, so the token account is of that type of mint, and can hold a balance, and hold the actual coins minted, by the mint. the mint is like the money press and you hold the dollars in your dollars token account.



Smart contracts are stateless and are read-only.

Second account can interact with contract and store application data

Block (master container)

* Transactions (metadata, accounts)
  + Array of signatures, perhaps needed for permissions to do various tasks, crediting lamports in account and sending to another account. Writing into and modifying data of account
  + Message
    - Header (metadata similar to http request) [account of signatures, account of read-only addresses that require signatures, account of read-only addresses that don’t require signatures]
    - Array of account addresses
    - Prev blockhash
    - Array of instructions, commands launched from specific programs (program id, list of accounts the program needs access to which need to be in array above, set of params called instruction data,

**RUST**

Variables by default immutable

Primitives (fixed size): knows how much memory to allocate and uses stack

Unsized Types: allocate memory on the heap and pointer that points to it stored on heap (ptr, len, capacity, etc)

Rules of Ownership

1. Each variable has value called its owner
2. Can only be one owner at a time
   1. If you try to copy variable, it will move ownership, making old variable invalid
   2. If you do want to create a copy run clone() function
3. When value goes out of scope, value will be dropped

Borrowing

For **read-only access**, can have as many borrows as you like, using the symbol (&), while a mut variable x is being borrowed it cannot be changed, if we want to change the value x after borrowing, we can create a scope for the borrowing segment and mutate it afterwards

For **borrowing mutably**, only single mutable borrow can occur at a time, using the symbol(&mut)

Ex1: clear string function does mutable borrow but ownership remains outside function.

fn clear\_string(s: &mut String) { s.clear(); }

Ex2: a = (takes ownership), b = (read-only borrow), c = (mutable borrow)

fn example\_fn(a: String, b: &str, c: &mut str) {}

Borrowing slice of a vector/list,

a = “hello”

b = &a[0..2] /// “he”

c = &a[1..5] /// “ello”

can even grab non-overlapping slices of a vector using **split\_at\_mut** fn

**Function bodies contain statements and expressions**

Statements don’t return values

Expressions evaluate to a value. Ex. 5+6, functions (with return type), calling macro, the block we use to create new scopes, {}, is an expression

let y = {

let x = 3;

x + 1

};

If you add a semicolon to the end of an expression, you turn it into a statement, which will not return a value

Function return values

The return value of the function is synonymous with value of final expression in the block of the function body

**Loop, while, for**

fn main() {

let mut count = 0;

'counting\_up: loop {

println!("count = {}", count);

let mut remaining = 10;

loop {

println!("remaining = {}", remaining);

if remaining == 9 {

break;

}

if count == 2 {

break 'counting\_up;

}

remaining -= 1;

}

count += 1;

}

println!("End count = {}", count);

}

Here we can specify which loop to break from with loop labels, ex. ‘counting\_up here

**Structs**

**Enum**

 Option, which is another enum defined by the standard library.

**Rust does NOT have null feature**

Rust does not have nulls but does have an enum that encodes concept of value being present or abset. This enum is Option<T>

Defined by the standard library as follows:

enum Option<T> {

None,

Some(T),

}

You can use None, Some(T) directly without the Option:: prefix

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.

Some(T) and T are two different types, know the difference!

**Matching**

match dice\_roll {

3 => add\_fancy\_hat(),

7 => remove\_fancy\_hat(),

other => move\_player(other),

}

Catch-all pattern and assigned variable value to other

When we don’t want to use the value in the catch-all pattern: can use \_

match dice\_roll {

3 => add\_fancy\_hat(),

7 => remove\_fancy\_hat(),

\_ => reroll(),

Solana Programs

Use statements, indicating libraries needed

No access to rand std::fs, std::net, std::os, std::future, std::net, std::process, std::sync, std::task, std::thread, std::time

**Process\_instruction**

program entrypoint

Params: program\_id &Pubkey, accounts &[AccountInfo], \_instruction\_data: &[u8]

msg!(“Hello World Logging!”) #command for logging Solana

account.owner does not mean owner like user with private keys. Means controller of programmatic controller of account.

BorshDeserialize, BorshSerialize

Takes binary format, either deserializes it or takes datatype and serializes it into binary format

Actual type, the data object is represented inside accounts data as object type

Only types are struct, enum, tuple

entrypoint!(…

possible for smart contracts to call into other smart contracts, better for dividing up work,

or modularizing

**Anchor**

#[program]

Program macro, defines set of instruction handlers, similar to a webserver defining the routes