
Chainlink
SOLANA
Blockchain Developer Bootcamp
Day 1

Harry Papacharissiou

Developer Advocate, Chainlink Labs

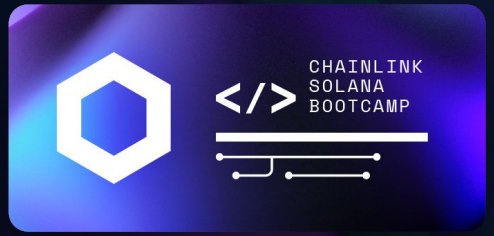
harry@chainlinklabs.com

@pappas9999

harry-papacharissiou

Chainlink

Chainlink



Solana Developer Bootcamp

Day 1

Introduction to Solana, Solana Architecture and Programming Model, Building Solana Smart Contracts

Solana Blockchain Developer Bootcamp Outline

1

Day 1: Introduction

Learn about Solana, it's architecture and programming model. Build some example smart contracts such as a simple GM program, and a token contract.

2

Day 2: Anchor and Chainlink

Learn how to use the Anchor framework, as well as how to use Chainlink Price Feeds to get pricing data in your Solana smart contracts

Chainlink

Housekeeping Rules

- 3 hour session, broken up into sections, with breaks
- Presentation, followed by questions, demo and exercises
- Questions can be asked at the end of each section
- If you need help during a practical exercise, ask in the Q&A, or join one of the pasted google meet breakout sessions
- Technical questions can be asked in the chat or in the #**bootcamp** channel in the Chainlink Discord after the session

Day 1 Exercises

- Exercise requirements:
  - NodeJS
  - Rust
  - Solana Tool Suite
  - Visual Studio Code

Agenda: Day 1

1. **Smart Contracts**
2. Introduction to Solana
3. Solana Architecture and Programming Model
4. Rust Fundamentals
5. Building Solana Smart Contracts
6. Exercise: GM contract
7. SPL Token Program and Program Derived Addresses
8. Exercise: Token Contract
9. Summary

Chewing Glass

Smart Contracts

What is a Contract and What is its Purpose?

Contract:

A binding agreement between two or more persons or parties.

Merriam-Webster

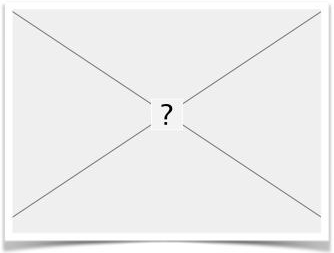
Contract:

A legal document that states and explains a formal agreement between two different people or groups.

Cambridge Dictionary

Chainlink

The Next Stage: Technologically Enforced Contracts



Telegraph Agreements (Electronically Signed) 1869

Telex Machines (Telecom Based) 1930s

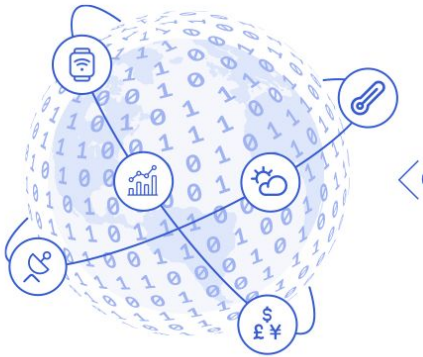
Digital Agreements (Internet Based) 1980s to Present

Smart Contracts (Blockchain Based) 2009 to Present

Chainlink

Digital Agreements

Performance Data



0010111011

Contract Terms

Contract Operator
Conflicts of Interest
(Pricing Monopolies)

Participants

1100100101

0110010010

Eight charged with defrauding McDonald's for prizes

By BY NAFTALI BENDAVID AND MELISSA HARRIS
TRIBUNE STAFF REPORTERS | AUG 22, 2001 AT 7:23 AM

More than $13 million in winnings from McDonald's "Monopoly," "Who Wants to Be a Millionaire" and other promotional games wound up in the pockets of a multistate crime ring, federal prosecutors said Tuesday.

Bad Faith Insurance

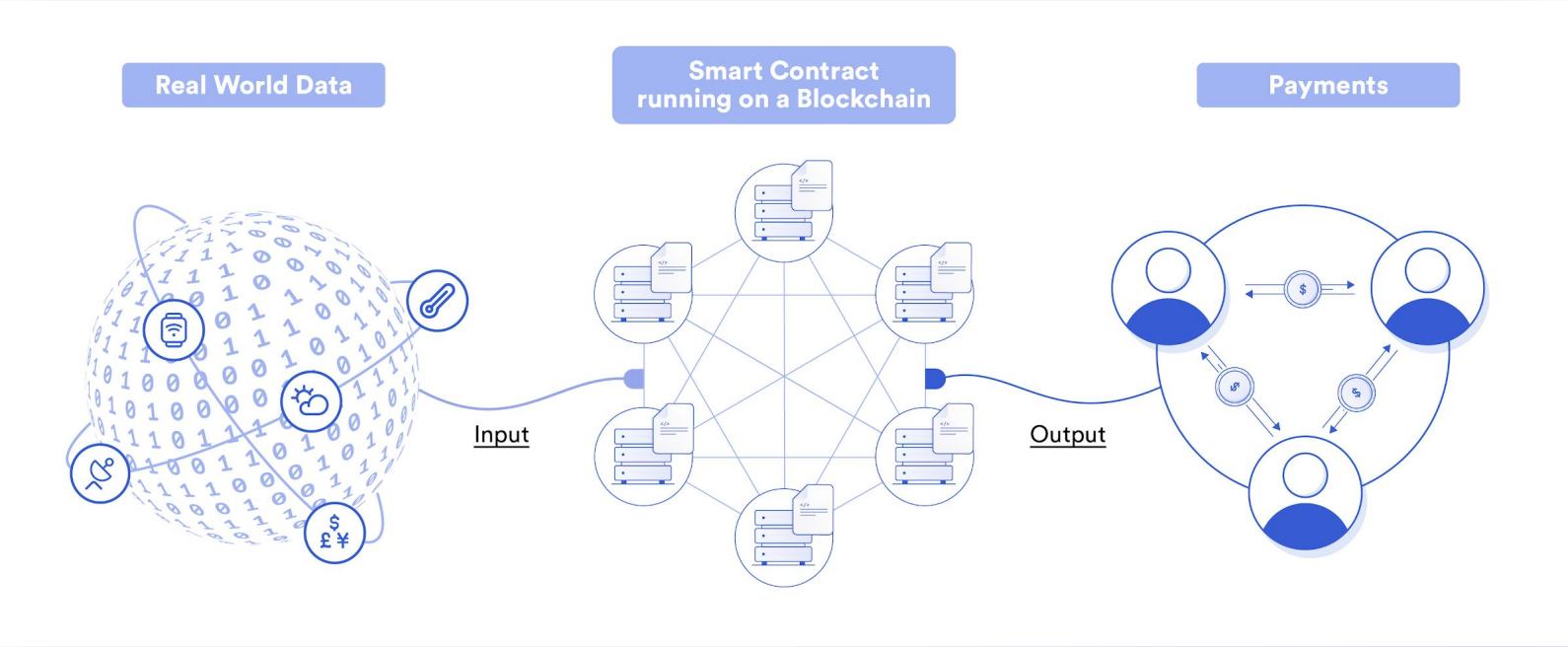
Chainlink

Smart Contracts / Programs

A Smart Contract is **a self-executing contract** with the terms of the agreement being **directly written into computer code**

Programmatically implement **a series of if-then rules** without the need for third-party human interaction

Solana Programs Run on the Blockchain



Chainlink

Cryptographic Truth is Strictly Better than “Just Trust Us”

“Just Trust Us” Paper Promises

Insurance

Banks

Ponzi Schemes

Fiat Currency

NFTs

DeFi

Multi-signature

Cryptoexchange

DEX

Hybrid Smart Contracts

Cryptographic Truth

Chainlink

Smart Contracts are Superior Digital Agreements

Security

Tamper-proof digital agreements that can't be influenced by a single party

Chainlink

Smart Contracts are Superior Digital Agreements

Security

Tamper-proof digital agreements that can't be influenced by a single party

Guaranteed Execution

Execution and enforcement of the agreement is performed by an always-on decentralized network

Chainlink

Smart Contracts are Superior Digital Agreements

Security

Tamper-proof digital agreements that can't be influenced by a single party

Guaranteed Execution

Execution and enforcement of the agreement is performed by an always-on decentralized network

Transparency

Transparency of the agreement and it's enforcement is unavoidably built-in

Chainlink

Smart Contracts are Superior Digital Agreements

Security

Tamper-proof digital agreements that can't be influenced by a single party

Guaranteed Execution

Execution and enforcement of the agreement is performed by an always-on decentralized network

Transparency

Transparency of the agreement and it's enforcement is unavoidably built-in

Trust Minimization

Reduced counterparty risk due to neither party having control on the agreements execution or enforcement

Chainlink

Smart Contracts are Superior Digital Agreements

Security

Tamper-proof digital agreements that can't be influenced by a single party

Guaranteed Execution

Execution and enforcement of the agreement is performed by an always-on decentralized network

Transparency

Transparency of the agreement and it's enforcement is unavoidably built-in

Trust Minimization

Reduced counterparty risk due to neither party having control of the agreements execution or enforcement

Efficiency

All these traits provide an opportunity to migrate existing manual processes to be automated

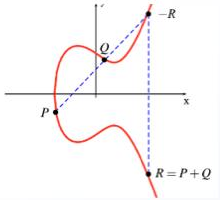
Chainlink

Cryptographic Truth is Strictly Better than “Just Trust Us”

“Just Trust Us” Paper Promises

- **Control is completely given away**
- Counterparty risk is high and opaque
- Transparency is purposefully removed

Cryptographic Truth Guarantees



$$y^2 = x^3+7$$

- **Control is in the user’s hands**
- Counterparty risk is low and transparent
- Transparency is unavoidably built-in

https://hackernoon.com/what-is-the-math-behind-elliptic-curve-cryptography-f61b25253da3

Chainlink

Introduction to Solana

High-Level Overview

- Open source, permissionless blockchain
- Unit of Account: SOL and Lamports. 1 Lamport = 0.000000001 SOL
- Approximately 1500 nodes capable of 50k TPS
- Implements a number of real enhancements to achieve this high level of throughput

High-Level Overview

- Comprised of multiple clusters
  - **Devnet**: for development
  - **Testnet**: similar to mainnet in terms of infrastructure, ledger and programs.
  - **Mainnet beta:** Mainnet
  - **Localnet/Test Validator**: Local full-featured cluster
- Uses a BFT PoS consensus mechanism and ‘Proof of History’ to maintain state
- Doesn’t have a mempool. Uses ‘Gulf-Stream’ to forward transactions to future ‘leaders’
- Clients or off-chain dApps can communicate with on-chain programs via the JSON RPC API or any SDK built on top of the API

Sending Transactions to a Cluster

- Clients send transactions to any validator
  - Ends up being sent to the designated leader
- Leader bundles transactions, timestamps, creates an entry, pushes it onto the clusters data plane
- Transactions are validated by validator nodes, appending to the ledger

Chainlink

Consensus Mechanism and Proof of History

- Blocks are broken up into smaller sets of transactions called entries
    - Entry time is 800ms
- The node that generates the block is called the Leader and the other nodes which perform the verification are called Validators.
- PoH - used to determine the order of incoming ‘blocks’
    - Allows to prove ordering of events or txs

      Allows for parallel processing by splitting up txs that are ordered
    - Puzzle Analogy

Chainlink

Consensus Mechanism and Proof of History

- Tower BFT: Uses PoH as a ‘clock’ before consensus to reduce communication overhead and latency
    - Implementation of practical BFT

Chainlink

Turbine

- Block propagation protocol that enables the fast propagation of large amounts of data to a large number of peers
- Splits the block up into smaller packets up to 64KB in size
- Establishes a random path per data packet through the network

Gulf Stream

- Mempool-less transaction forwarding protocol
- Each Validator knows the order of upcoming Leaders.
- Clients and Validators forward transactions to upcoming Leaders before they act as a Leader in the network.
- This allows Validators to start processing transactions ahead of time.
- Results in fewer transactions cached in Validators' memory and faster confirmations.

Sealevel

- Solana smart contracts run-time
- Enables the parallel execution of smart contracts
    - Sorts and classifies transactions by those reading the same data from accounts, or those which can be processed without overlapping
    - Once transactions are sorted, uses the validators hardware (GPU) to execute them across multiple cores

Solana Architecture and Programming Model

Programs/Accounts Model

- Program is an on-chain smart contract
  - Stateless, no global variables
- Program ID is the address/account the program is stored in
- Data for programs is stored in Accounts
  - Account is actually a buffer.
  - Can be thought of like a file in an O/S
  - Includes metadata that tells the runtime who is allowed to access the data and how
  - Held in validator memory and pay ‘rent’ to stay there
- ERC-20 Example

Accounts

```
pub struct AccountInfo {
    pub key: Pubkey,
    pub is_signer: bool,
    pub is_writable: bool,
    pub lamports: mut u64>>,
    pub data: mut [u8]>>,
    pub owner: Pubkey,
    pub executable: bool,
    pub rent_epoch: Epoch,
}
```

Instructions and Transactions

- Basic operational unit on Solana is an instruction
  - Eg 1 call to a program
- Instruction is split into three different parts:
  - Program ID
  - Accounts
  - Instruction Data
- Transaction is a list of instructions
- Clients submit transactions and fetch data via the RPC API

Instruction 1

```
Program ID: abcdefg12345
Accounts: <Acct1, Acct2>
Data: [1,2,3,4,5,6]
```

Instruction 2

```
Program ID: abcdefg12345
Accounts: <Acct3, Acct4>
Data: [9,8,7,6,5,4]
```

Transaction 1

```
[Instruction 1, Instruction 2]
```

Chainlink

Instructions: Accounts

- Each Account element in the array contains 3 fields:
  - Public Key
  - Is_signer: boolean
  - Is_writeable: boolean

Account 1

```
Public Key: abcdefg12345
is_signer: false
is_writeable: false
```

Account 2

```
Public Key: zyx987652
is_signer: true
is_writeable: true
```

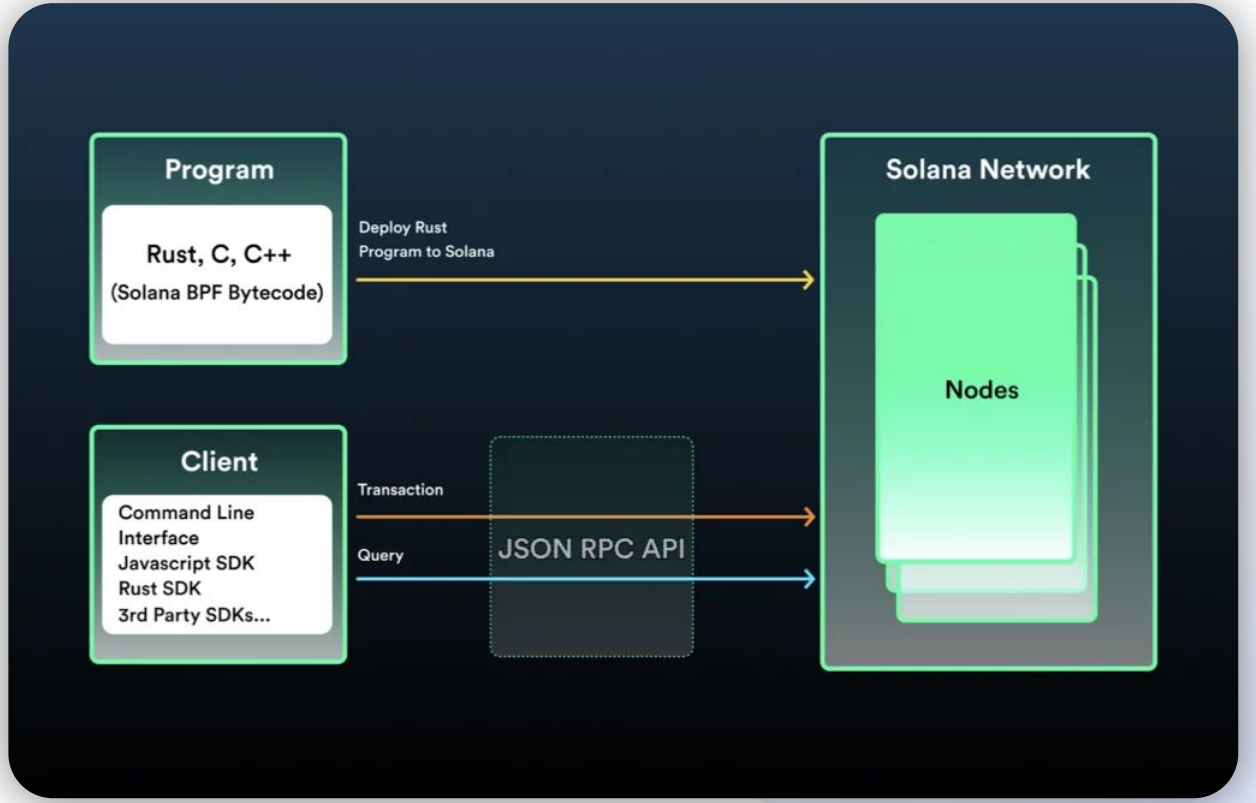
Instructions: Data

- Byte array of a certain size
- Used for sending input data to on-chain programs

```
03 05 00 00 00 00 00 00 00
```

Chainlink

Solana Instruction Flow



Chainlink

Economics and Rent

- Validators get paid transaction fees + inflationary rewards
- Stakers are rewarded for helping to validate the ledger by delegating their stake to validator nodes
- Inflation currently set to approximately 8%
- Each transaction submitted to the ledger imposes costs
  - Tx fees only cover processing the transaction, not storing it long term
- Storage Rent covers the cost of storing data in Accounts over time.
- 2 methods of storage rent:
  - Set and Forget
  - Pay per Byte

Writing and Compiling Programs

- Solana programs are compiled to Berkley Packet Filter (BPF).
- When called, a program must be passed to something called a BPF loader
- Programs can be written in any language that compiles to BPF bytecode.
- Rust, C++ and C programming languages are currently supported

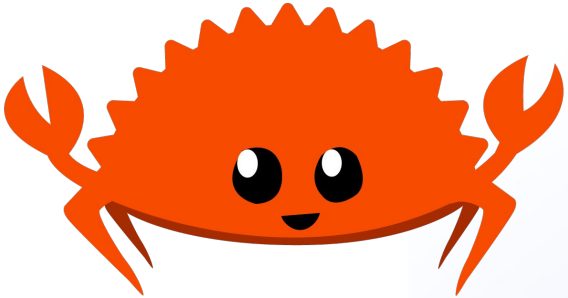
Developing on Solana

- Developing on Solana is more difficult compared to EVM chains.
  - Many things usually abstracted away need to be done by the developer
  - Added complexities around storage and retrieval of data
  - Separation of state and logic adds complexity
  - Libraries and frameworks help to alleviate some of this

Rust Fundamentals

Rust Fundamentals

- Fast, reliable and memory efficient
- Modern replacement for C++ and C
- High level of control on memory allocation and management



Rust Fundamentals: Data Types

- Split into two types:
  - Scalar
  - Compound

Chainlink

Rust Fundamentals: Scalar Data Types

- Represents a single value
- Four types:
  - Integer
  - Floating-point number
  - Boolean
  - Character

Chainlink

Rust Fundamentals: Integers

```
fn main() {
    let result = 10;     // i32 by default
    let age:u32 = 20;
    println!("Result value is {}",result);
    println!("Age is {}",age);
}
```

```
result value is 10
Age is 20
```

Chainlink

Rust Fundamentals: Floating-point Numbers

```
fn main() {
    let x = 2.0; // f64
    let y: f32 = 3.0; // f32
}
```

Chainlink

Rust Fundamentals: Boolean Data Type

```
fn main() {
    let isfun:bool = true;
    println!("Is Solana awesome ? {}",isfun);
}
```

```
Is Solana awesome ? true
```

Chainlink

Rust Fundamentals: Character Data Type

```
fn main() {
    let special_character = '@'; //default
    let alphabet:char = 'A';
    let emoji:char = '💪';

    println!("special character is {}",special_character);
    println!("alphabet is {}",alphabet);
    println!("emoji is {}",emoji);
}
```

```
special character is @
alphabet is A
emoji is 💪
```

Rust Fundamentals: Compound Data Types

- Can group multiple values into one type.
- Types:
  - Arrays
  - Vectors
  - Tuples

Chainlink

Rust Fundamentals: Arrays

- An array declaration allocates sequential memory blocks.
- Arrays are static (ie they can’t be resized)
- Each memory block represents an array element.
- Array elements are identified by a unique integer called the subscript/ index of the element.
- Populating the array elements is known as array initialization.
- Array element values can be updated or modified but cannot be deleted.

Chainlink

Rust Fundamentals: Arrays

```
fn main(){
   let arr:[i32;4] = [10,20,30,40];
   println!("array is {:?}",arr);
   println!("array size is
:{}",arr.len());
}
```

```
array is [10, 20, 30, 40]
array size is :4
```

Chainlink

Rust Fundamentals: Vectors

- An resizable array of specific element types
- Used when wanting to store an unknown number of elements
- Has a ‘capacity’ that defines how much data can be stored without reallocation
- Can specify capacity at creation

Chainlink

Rust Fundamentals: Vectors

```
fn main(){
   let mut vec = Vec::with_capacity(5)[10,20,30,40];
   vec.push(50)
   println!("Vector is {:?}",vec);
   println!("Vector length is :{}",vec.len());
   println!("Vector Capacity is :{}",vec.capacity());
}
```

```
Vector is [10, 20, 30, 40, 50]
Vector length is :5
Vector Capacity is :5
```

Rust Fundamentals: Tuples

- A compound data type that can store more than one value at a time
- Data can be of different types
- Fixed length
- Similar to EVM version of a struct

Chainlink

Rust Fundamentals: Tuples

```
fn main() {
   let tuple:(i32,f64,u8) = (-325,4.9,22);
   println!("integer is :{:?}",tuple.0);
   println!("float is :{:?}",tuple.1);
   println!("unsigned integer is
:{:?}",tuple.2);
}
```

```
integer is :-325
float is :4.9
unsigned integer is :2
```

Chainlink

Rust Fundamentals: Cargo and Crates

- Rust package manager
- Works with ‘crates’
- Allows Rust packages to declare their various dependencies and ensure builds are repeatable

Rust Fundamentals: Cargo

- To achieve this, Cargo does 4 things:
  - Introduces 2 metadata files with various bits of package information
    - Cargo.toml
    - Cargo.lock
  - Fetches and builds your packages dependencies
  - Invokes the compiler (rustc) or another build tool with the correct parameters to build your package
  - Introduces conventions to make working with rust easier

Chainlink

Rust Fundamentals: Cargo

- Cargo.toml
  - Describing your dependencies in a broad sense
  - Written by the developer
  - A manifest file where we can specify a bunch of different metadata about our package.
    - Eg, can say your package depends on another package
- Cargo.lock
  - contains exact information about your dependencies. It is maintained by Cargo and should not be manually edited

Chainlink

Rust Fundamentals: Cargo.toml

```
[package]
name = "hello_world"
version = "0.1.0"

[dependencies]
regex = { git =
"https://github.com/rust-lang/regex.git" }
```

Chainlink

Rust Fundamentals: Cargo.lock

```
[[package]]
name = "hello_world"
version = "0.1.0"
dependencies = [
 "regex 1.5.0
(git+https://github.com/rust-lang/regex.git#9f9f693768c584
971a4d53bc3c586c33ed3a6831)",
]

[[package]]
name = "regex"
version = "1.5.0"
source =
"git+https://github.com/rust-lang/regex.git#9f9f693768c584
971a4d53bc3c586c33ed3a6831"
```

Chainlink

Rust Fundamentals: Libraries

- Specific type of Rust Crate
- A crate can be compiled into a binary or into a library. By default, rustc will produce a binary from a crate.
- Prefixed with ‘lib’, and named after their crate by default
- Can browse library crates at https://crates.io/

Chainlink

Rust Fundamentals: Creating a Library

```
# library.rs file
pub fn public_function() {
    println!("called the `public_function()`");
}

fn private_function() {
    println!("called the `private_function()`");
}
```

```
rustc -create-type=lib library.rs
```

Rust Fundamentals: Using a Library

```
fn main() {
    library::public_function();

    library::private_function();
}
```

```
# Where library.rlib is the path to the compiled library, assumed that
it's in the same directory here:

$ rustc executable.rs --extern library=library.rlib && ./executable
called the `public_function()`
called the `private_function()`
```

Chainlink

Rust Fundamentals: Modules

- Allow for the organization of code into groups for readability and re-use
- Flexibility for controlling the privacy of items (public, private within the module)
- Can also hold definitions of other items inside, including other modules
- Allow for the grouping of related definitions together

Chainlink

Rust Fundamentals: Modules

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

Rust Fundamentals: Use

- Not an import statement!
- Creates one or more local name bindings for some other path
- Usually used to shorten the path required to refer to a module item
- Usually declared at the top of a module

Chainlink

Rust Fundamentals: Use

```
// -- Initial code without the `use` keyword --
mod phrases {
  pub mod greetings {
    pub fn hello() {
      println!("Hello, world!");
    }
  }
}

fn main() {
  phrases::greetings::hello();  // Using full path
}
```

Chainlink

Rust Fundamentals: Use

```
// -- Usage of the `use` keyword --
// 01. Create an alias for module
use phrases::greetings;
fn main() {
  greetings::hello();
}

// 02. Create an alias for module elements
use phrases::greetings::hello;
fn main() {
  hello();
}

// 03. Customize names with the `as` keyword
use phrases::greetings::hello as greet;
fn main() {
  greet();
}
```

Rust Fundamentals: Mut

- Mutation in Rust is modifying memory contents
- Variable bindings are immutable by default, but this can be overridden using the mut modifier
- The mut modifier applies to the binding itself, not the memory it binds to

Chainlink

Rust Fundamentals: Mut

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    // Ok
    mutable_binding += 1;

    // Error!
    _immutable_binding += 1;

}
```

Chainlink

Rust Fundamentals: Referencing

- A non-owning pointer type that references another value in memory
- Created using the borrow-operator **&**

```
let x = 10;
let r = &x;
```

Chainlink

Rust Fundamentals: Dereferencing

- References can be dereferenced using the * operator
- This enables the value to be accessed in memory

```
let x = 10;
let r = &x;

if *r == 10 {
  println!("Same!");
}
```

```
Same
```

Building Solana Smart Contracts

Chainlink

Compiling Solana Programs

- Solana programs are compiled to Berkley Packet Filter (BPF).
- When called, a program must be passed to something called a BPF loader
- Programs can be written in any language that compiles to BPF bytecode.
- Rust, C++ and C programming languages are currently supported

Compiling Solana Programs: WASM

- WebAssembly (WASM) is a portable low-level language that Rust can compile to.
- Solana provides a great set of tools for developers to write smart contracts in C/C++ and Rust that execute contracts on GPUs.
- Solana isn't directly using WASM, but is compatible with code targeted for other WASM chains

Solana Project Layout

```
/examples/
/inc/
/src/
 -  lib.rs
 -  entrypoint.rs/main.rs
 -  error.rs
 -  instruction.rs
 -  processor.rs
 -  state.rs
 -  main.rs
/tests/
/Cargo.toml
/Cargo.lock
```

Chainlink

Native Programs

- Native ‘built-in’ programs deployed to the cluster that are required to run the network
  - Can think of them as the operating system of a computer
- Part of the validator implementation
- Can be upgraded
- Each one has a pre-assigned program ID
  - Eg System Program is 11111111111111111111111111111111

Native Programs

- System Program
- Config Program
- Stake Program
- Vote Program
- BPF Loader
- Ed25519 Program
- Secp256k1 Program

Chainlink

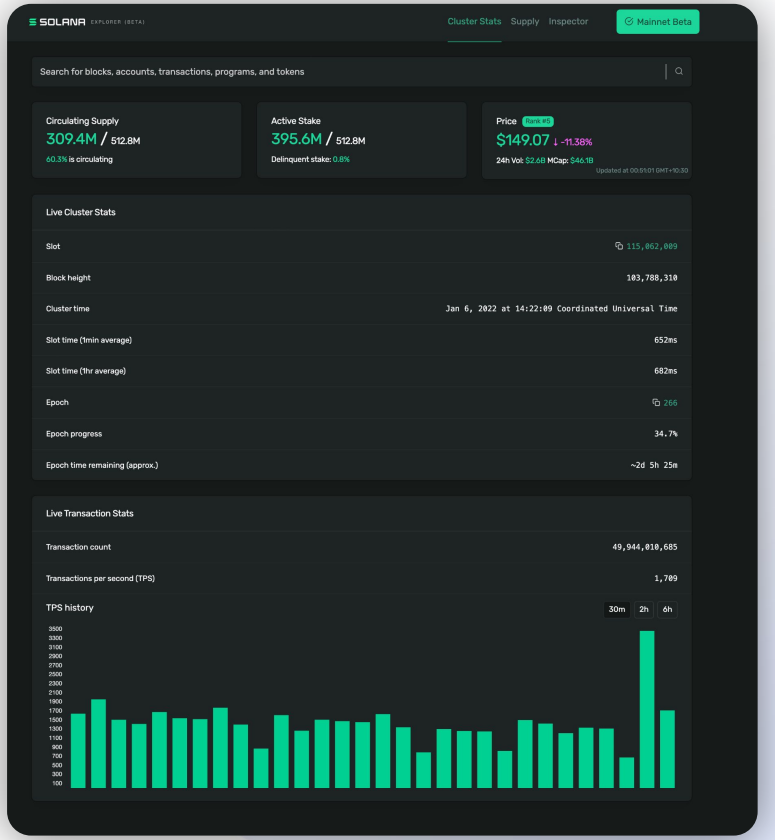
Solana Development Tools: CLI

The Solana CLI is a tool that allows you to interact with the Solana blockchain via the command line.

```
pappas99@Harrys-MacBook-Pro anchor % solana help
solana-cli 1.7.10 (src:03b93051; feat:660526986)
Blockchain, Rebuilt for Scale

USAGE:
    solana [FLAGS] [OPTIONS] <SUBCOMMAND>

FLAGS:
    -h, --help                          Prints help information
        --no-address-labels             Do not use address labels in the output
        --skip-seed-phrase-validation   Skip validation of seed phrases. Use this if your phrase does not use the BIP39
                                        official English word list
    -V, --version                       Prints version information
    -v, --verbose                       Show additional information

OPTIONS:
        --commitment <COMMITMENT_LEVEL>    Return information at the selected commitment level [possible values:
                                           processed, confirmed, finalized]
    -C, --config <FILEPATH>                Configuration file to use [default:
                                           /Users/pappas99/.config/solana/cli/config.yml]
    -u, --url <URL_OR_MONIKER>             URL for Solana's JSON RPC or moniker (or their first letter): [mainnet-beta,
                                           testnet, devnet, localhost]
    -k, --keypair <KEYPAIR>                Filepath or URL to a keypair
        --output <FORMAT>                  Return information in specified output format [possible values: json, json-
                                           compact]
        --ws <URL>                         WebSocket URL for the solana cluster

SUBCOMMANDS:
```

Solana Development Tools: Solana Explorer

https://explorer.solana.com/



Chainlink

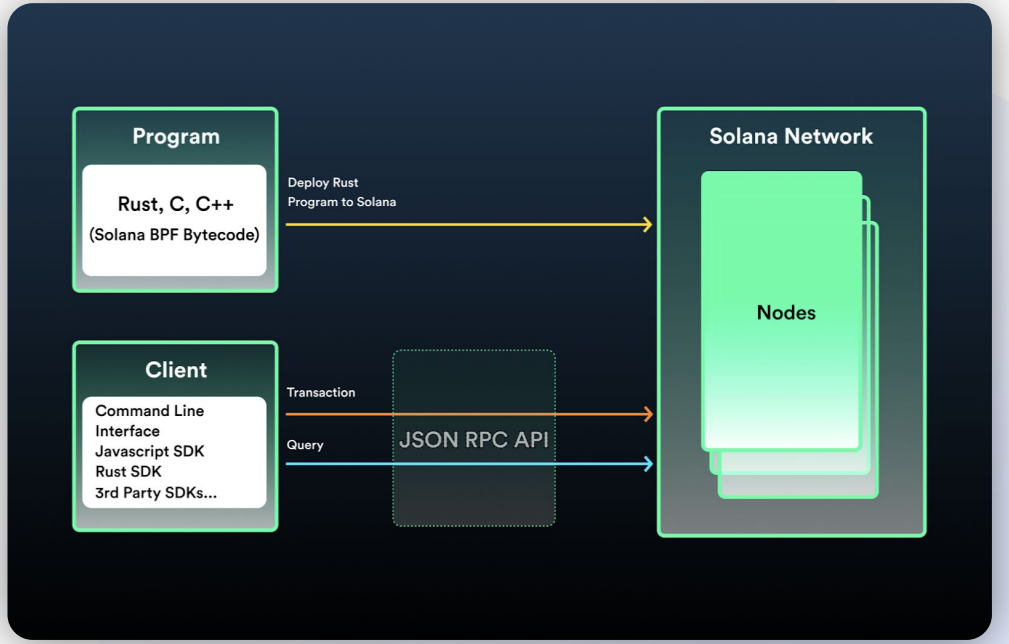
Solana Development Tools: Test Validator

- Local full-featured single node cluster
  - No RPC rate-limits
  - No airdrop limits
  - Other configurable options
- Comes with the Solana CLI

```
pappas99@ip-192-168-0-2 chainlink-solana-demo % solana-test-validator
Ledger location: test-ledger
Log: test-ledger/validator.log
⠒ Initializing...
Identity: 7PVMV8mFXyQHPHxA17t7amjXjXr42N84XbwNPjGvTX4B
Genesis Hash: E23ucUzruy4NcUWQRRjAy514me8o9mEQnXqTnh1fc8Zt
Version: 1.9.4
Shred Version: 34140
Gossip Address: 127.0.0.1:1024
TPU Address: 127.0.0.1:1027
JSON RPC URL: http://127.0.0.1:8899
⠒ 00:00:10 | Processed Slot: 19 | Confirmed Slot: 19 | Finalized Slot: 0 | Full Snapshot Slot: - | Incremental Snapsh
```

Interacting with Solana Programs

- JSON RPC API to interact with Solana programs
- Can use web3js or other libraries to interact with the API



Chainlink

Serialization and Deserialization in Solana

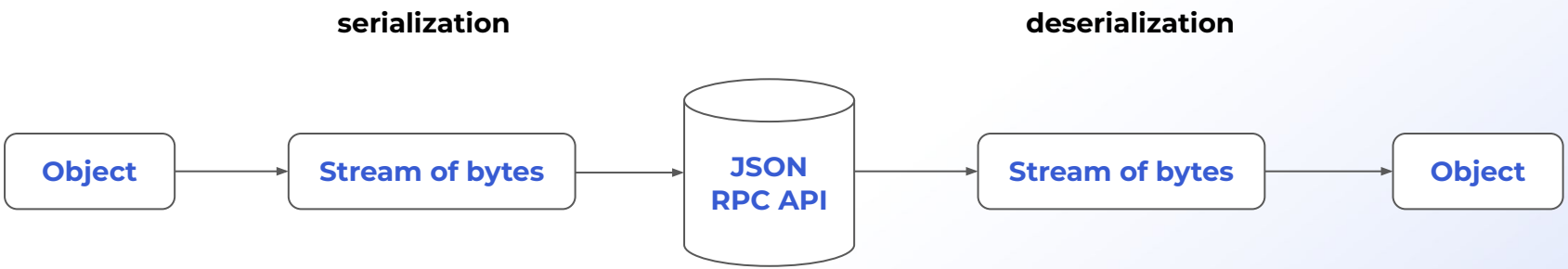
- Serialization: converting an object in memory into a stream of bytes
- Deserialization: converting a stream of bytes into a readable object in memory
- Sending data to/from a program via the JSON RPC requires serialization and deserialization

Instruction 1

```
Program ID: abcdefg12345
Accounts: <Acct1, Acct2>
Data: [1,2,3,4,5,6]
```

Chainlink

Serialization and Deserialization in Solana



Chainlink

Serialization and Deserialization in Solana

- Serializing instruction data sent from a client to Solana
- Deserializing instruction data received from the client in the on-chain program
- Deserializing AccountInfo data received from the client in the program
- Serializing AccountInfo data to be stored in accounts via the program

Serialization and Deserialization in Solana

- Borsh is the most common format used for Serializing and Deserializing in Solana
  - Integers are little endian
  - Sizes of dynamic containers are written before values as u32
  - Structs are serialization in the order of fields in the struct

Serialization and Deserialization: Borsh Example

```
//client code
import * as borsh from "@project-serum/borsh";

const PARAMS = borsh.struct([
  borsh.u8("uint 8"),
  borsh.publicKey("key property"),
  borsh.str("string"),
]);
```

Serialization and Deserialization: Client Serializing

```
//client code
import { PublicKey } from "@solana/web3.js";

const payload = {
  u8: 7
  publickey: new PublicKey("..."),
  str: "eat_glass",
};

//start with a big enough buffer
Const buffer = Buffer.alloc(1000);

//Serialize the data into the buffer
PARAMS.encode(payload, buffer);

//Remove the unused bytes from the buffer
Const serializedData = buffer.slice(0, PARAMS.getSpan(buffer));
```

Serialization and Deserialization: On-chain Code

```
//on-chain program
use borsh::{BorshSerialize, BorshDeserialize};
use solana_program::pubkey::Pubkey;

#[derive(BorshSerialize, BorshDeserialize, Debug, Clone)]
pub struct Params {
    pub u8_int: u8,
    pub public_key: Pubkey,
    pub string_param: String
}
```

Serialization and Deserialization: On-chain Code

```
//on-chain program
use solana_program::borsh::try_from_slice_unchecked;

//data is of type &[u8]

//this will fail
let d = Params::try_from_slice(data)?;

//this will pass
let d = try_from_slice_unchecked::Params>(data).unwrap();
```

Serialization and Deserialization: Serializing in Rust

```
//on-chain program
use solana_program::borsh::try_from_slice_unchecked;

//data is of type &[u8]
d.serialize(&mut &mut data.borrow_mut()[..])?
```

Serialization and Deserialization: Deserializing

```
//client program
const paramData = PARAM.decode(buffer);
```

Chainlink

Chainlink

Exercise 1: GM Smart Contract

- Program that accepts a name parameter, says GM to it and stores the name in an account
- Client that reads the account contents
- 30 minutes

Word Cloud

https://www.menti.com/7pi29tbn1z

Solana Program Library (SPL)

- Library of on-chain programs targeting the Sealevel parallel runtime
- Can think of them as optimized programs to be used for a variety of specific use-cases (like working with tokens), but unrelated to the underlying protocol

Solana Program Library (SPL)

- Token Program
- Token Swap Program
- Token Lending Program
- Associated Token Account Program
- Memo Program
- Name Service
- Shared Memory Program
- Stake Pool
- Feature Proposal Program

Chainlink

Solana Program Derived Address (PDA)

- Account whose owner is a program and thus is not controlled by a private key like other accounts
- Deterministically derived from a collection of seeds and a program ID using a 256-bit hash function
- Allows programs to sign for certain addresses without needing a private key

Chainlink

Solana PDA Use Cases

- Allow clients to programmatically derive account addresses for a particular program, negating the need to store public keys or keypairs off-chain

```
await anchor.web3.PublicKey.findProgramAddress(
  [Buffer.from("some_seed_string")],
  program.programId
);
```

Exercise 2: Token Contract

- Simplified version of the SPL Token Program
  - Create Token
  - Create Token Account
  - Mint Tokens
  - Transfer Tokens
- 60 minutes

Solana Blockchain Developer Bootcamp Day 1 Summary

- Intro to Solana
- Solana Architecture and Programming Model
- Smart Contracts Overview
- Rust Fundamentals
- Building Solana Smart contracts
- Exercises

Chainlink

Chainlink

SOLANA

Congratulations for completing Day 1 of the Solana Blockchain Developer Bootcamp