한 번에 끝내는 블록체인 개발 A to Z

Chapter 1

Solana Introduction

Chapter 3 Rust Introduction

Programming Model

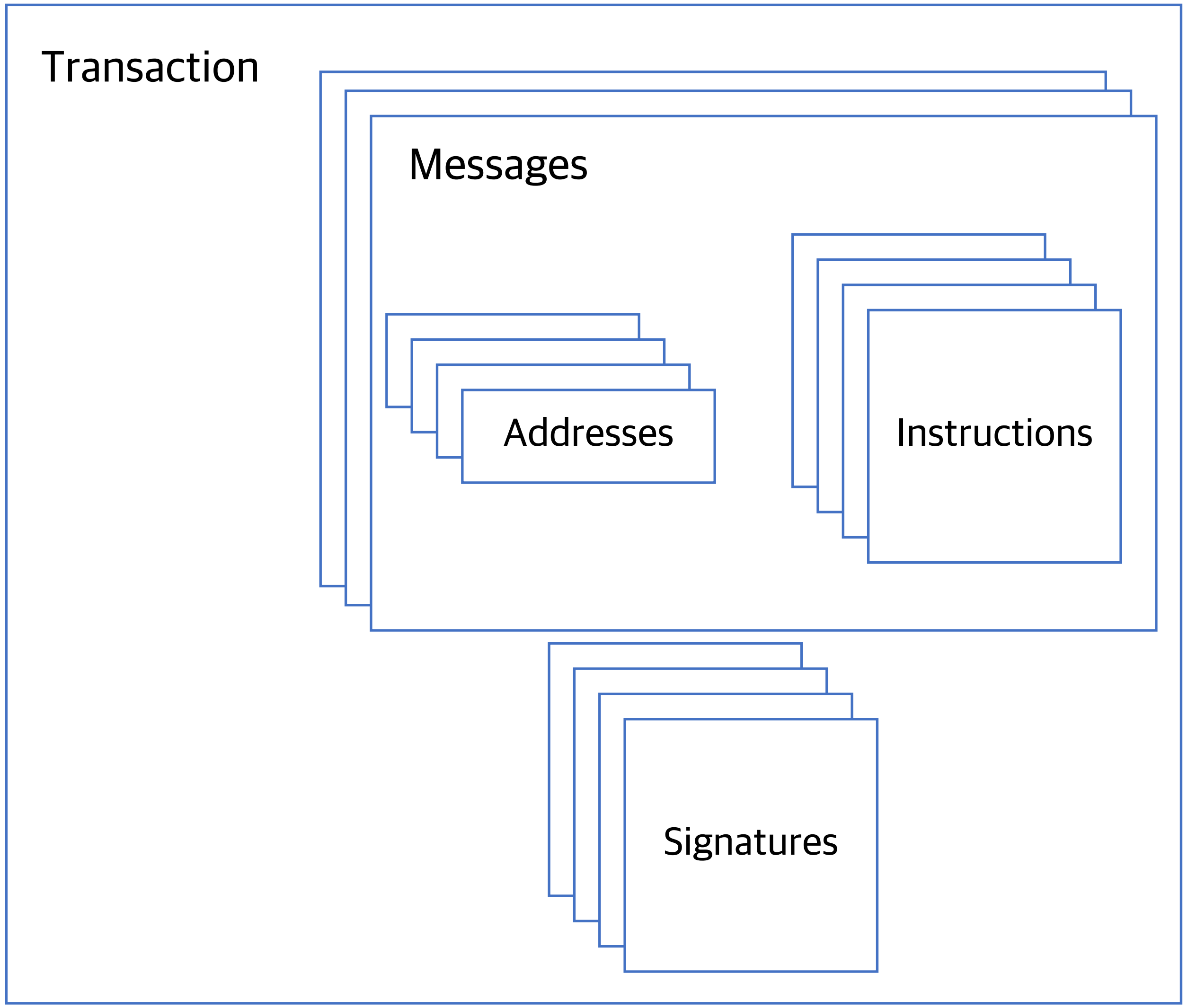
Transactions

Overview

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.

Transactions

Program execution begins with a transaction being submitted to the cluster. The Solana runtime will execute a program to process each of the instructions contained in the transaction, in order, and atomically.



Instructions

Each instruction specifies a single program, a subset of the transaction's accounts that should be passed to the program, and a data byte array that is passed to the program.

The program interprets the data array and operates on the accounts specified by the instructions. The program can return successfully, or with an error code. An error return causes the entire transaction to fail immediately.

SystemInstruction::CreateAccount

```
pub fn create_account(
    from_pubkey: &Pubkey,
    to_pubkey: &Pubkey,
    lamports: u64,
    space: u64,
    owner: &Pubkey,
) -> Instruction {
    let account_metas = vec![
        AccountMeta::new(*from_pubkey, true),
        AccountMeta::new(*to_pubkey, true),
    ];
    Instruction::new_with_bincode(
        system_program::id(),
        &SystemInstruction::CreateAccount {
            lamports,
            space,
            owner: *owner,
        },
        account_metas,
    )
}
```

Program ID

The instruction's program id specifies which program will process this instruction. The program's account's owner specifies which loader should be used to load and execute the program, and the data contains information about how the runtime should execute the program.

In the case of on-chain BPF programs, the owner is the BPF Loader and the account data holds the BPF bytecode. Program accounts are permanently marked as executable by the loader once they are successfully deployed. The runtime will reject transactions that specify programs that are not executable.

SystemInstruction::CreateAccount

```
pub fn create_account(
    from_pubkey: &Pubkey,
    to_pubkey: &Pubkey,
    lamports: u64,
    space: u64,
    owner: &Pubkey,
) -> Instruction {
    let account_metas = vec![
        AccountMeta::new(*from_pubkey, true),
        AccountMeta::new(*to_pubkey, true),
    ];
    Instruction::new_with_bincode(
        system_program::id(),
        &SystemInstruction::CreateAccount {
            lamports,
            space,
            owner: *owner,
        },
        account_metas,
    )
}
```

Accounts

The accounts referenced by an instruction represent on-chain state and serve as both the inputs and outputs of a program.

Accounts are similar to files in operating systems such as Linux in that they may hold arbitrary data that persists beyond the lifetime of a program.

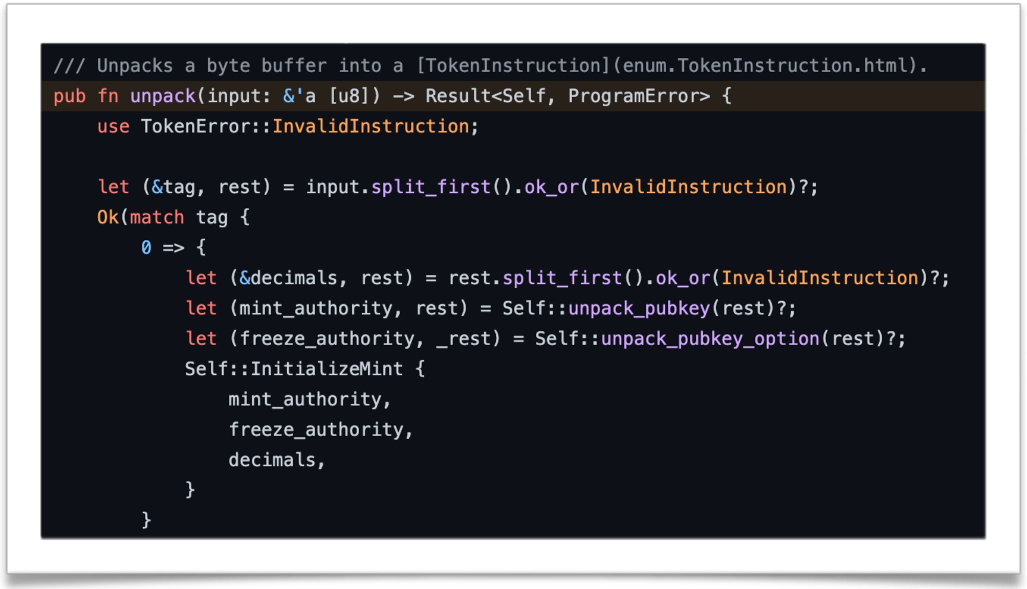
SystemInstruction::CreateAccount

```
pub fn create_account(
    from_pubkey: &Pubkey,
    to_pubkey: &Pubkey,
    lamports: u64,
    space: u64,
    owner: &Pubkey,
) -> Instruction {
    let account_metas = vec![
        AccountMeta::new(*from_pubkey, true),
        AccountMeta::new(*to_pubkey, true),
    ];
    Instruction::new_with_bincode(
        system_program::id(),
        &SystemInstruction::CreateAccount {
            lamports,
            space,
            owner: *owner,
        },
        account_metas,
    )
}
```

Instruction Data

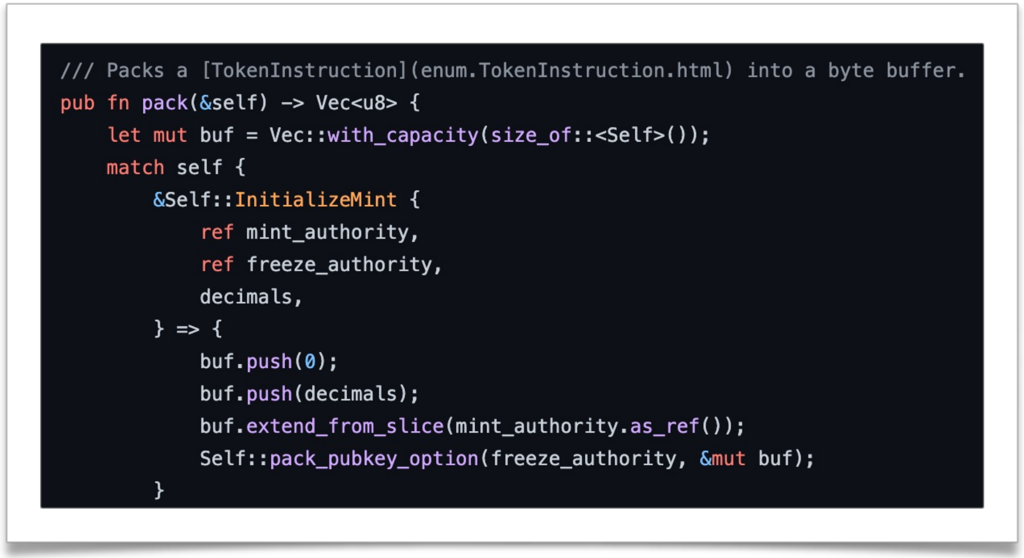
Each instruction carries a general purpose byte array that is passed to the program along with the accounts. The contents of the instruction data is program specific and typically used to convey what operations the program should perform, and any additional information those operations may need above and beyond what the accounts contain.

Programs are free to specify how information is encoded into the instruction data byte array.



```rust
/// Unpacks a byte buffer into a [TokenInstruction](enum.TokenInstruction.html).
pub fn unpack(input: &'a [u8]) -> Result<Self, ProgramError> {
    use TokenError::InvalidInstruction;

    let (&tag, rest) = input.split_first().ok_or(InvalidInstruction)?;
    Ok(match tag {
        0 => {
            let (&decimals, rest) = rest.split_first().ok_or(InvalidInstruction)?;
            let (mint_authority, rest) = Self::unpack_pubkey(rest)?;
            let (freeze_authority, _rest) = Self::unpack_pubkey_option(rest)?;
            Self::InitializeMint {
                mint_authority,
                freeze_authority,
                decimals,
            }
        }
```



```rust
/// Packs a [TokenInstruction](enum.TokenInstruction.html) into a byte buffer.
pub fn pack(&self) -> Vec<u8> {
    let mut buf = Vec::with_capacity(size_of::<Self>());
    match self {
        &Self::InitializeMint {
            ref mint_authority,
            ref freeze_authority,
            decimals,
        } => {
            buf.push(0);
            buf.push(decimals);
            buf.extend_from_slice(mint_authority.as_ref());
            Self::pack_pubkey_option(freeze_authority, &mut buf);
        }
```

Multiple Instructions

A transaction can contain instructions in any order. This means a malicious user could craft transactions that may pose instructions in an order that the program has not been protected against. Programs should be hardened to properly and safely handle any possible instruction sequence.

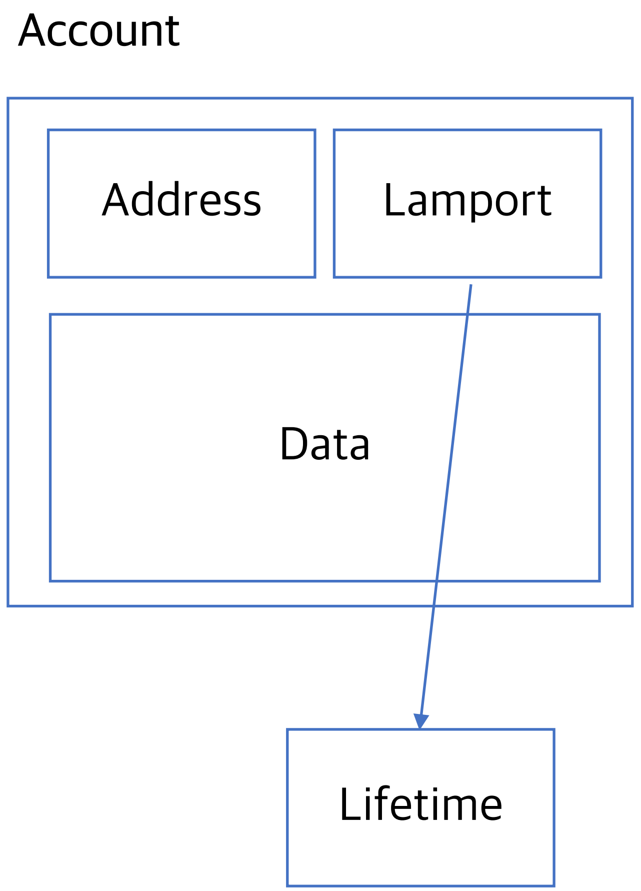
One not so obvious example is account **deinitialization**. Some programs may attempt to deinitialize an account by setting its **lamports** to zero, with the assumption that the **runtime** will delete the account. This assumption may be valid between transactions, but it is not between instructions or cross-program invocations. To harden against this, the program should also explicitly zero out the account's data.

Accounts

Storing State between Transactions

If the program needs to store state between transactions, it does so using accounts. **Accounts** are similar to files in operating systems such as Linux in that they may hold arbitrary data that persists beyond the lifetime of a program. Also like a file, an account includes metadata that tells the runtime who is allowed to access the data and how.

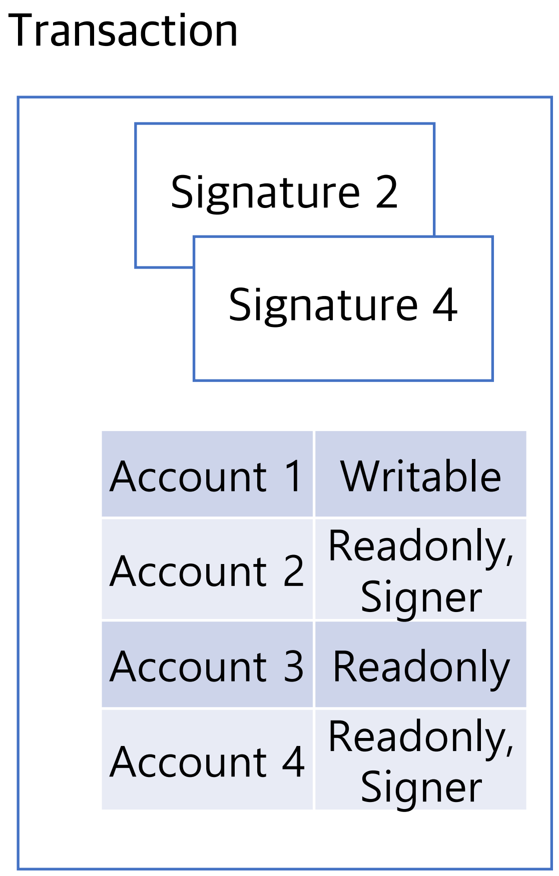
Any account that drops to zero **lamports** is purged. Accounts can also be marked rent-exempt if they contain a sufficient number of lamports.



Signers

Transactions include one or more digital signatures each corresponding to an account address referenced by the transaction.

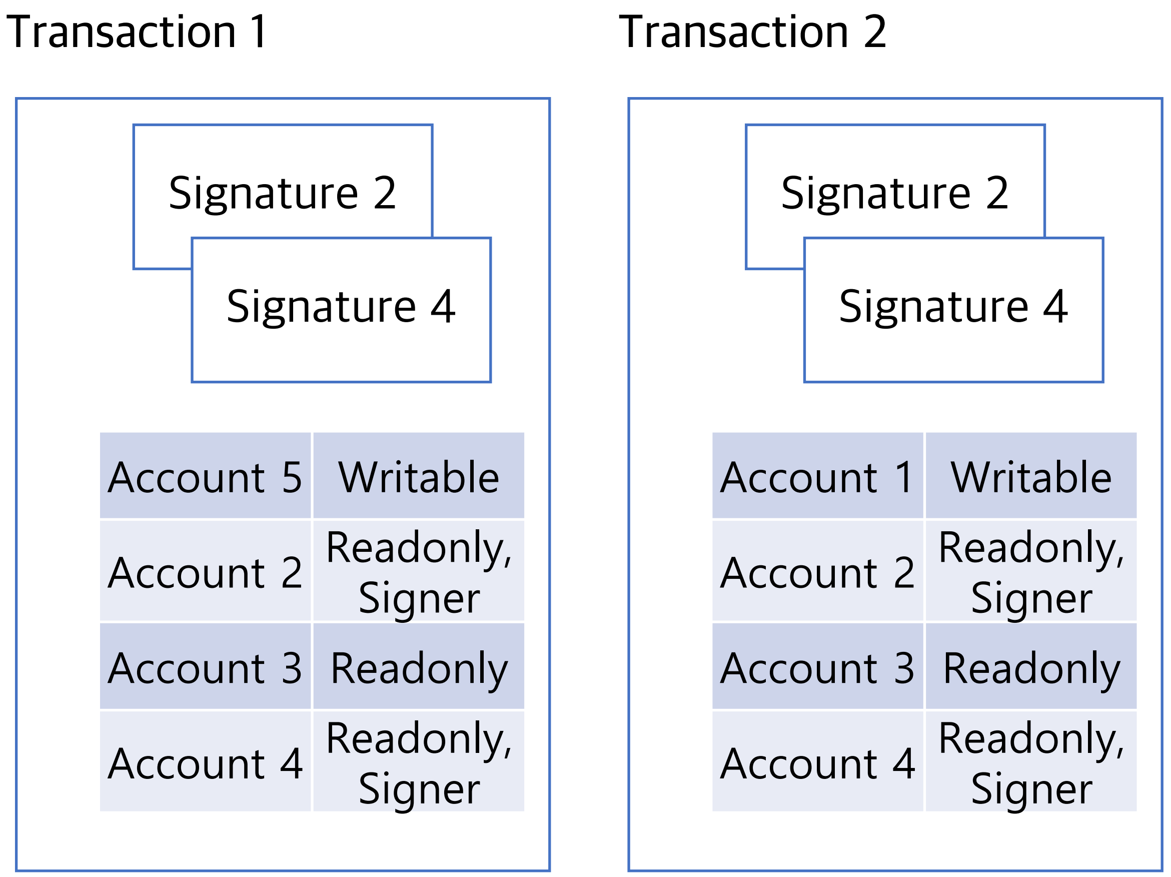
Whether an account is a signer or not is communicated to the program as part of the account's metadata. Programs can then use that information to make authority decisions.



Account 1	Writable
Account 2	Readonly, Signer
Account 3	Readonly
Account 4	Readonly, Signer

Read-only

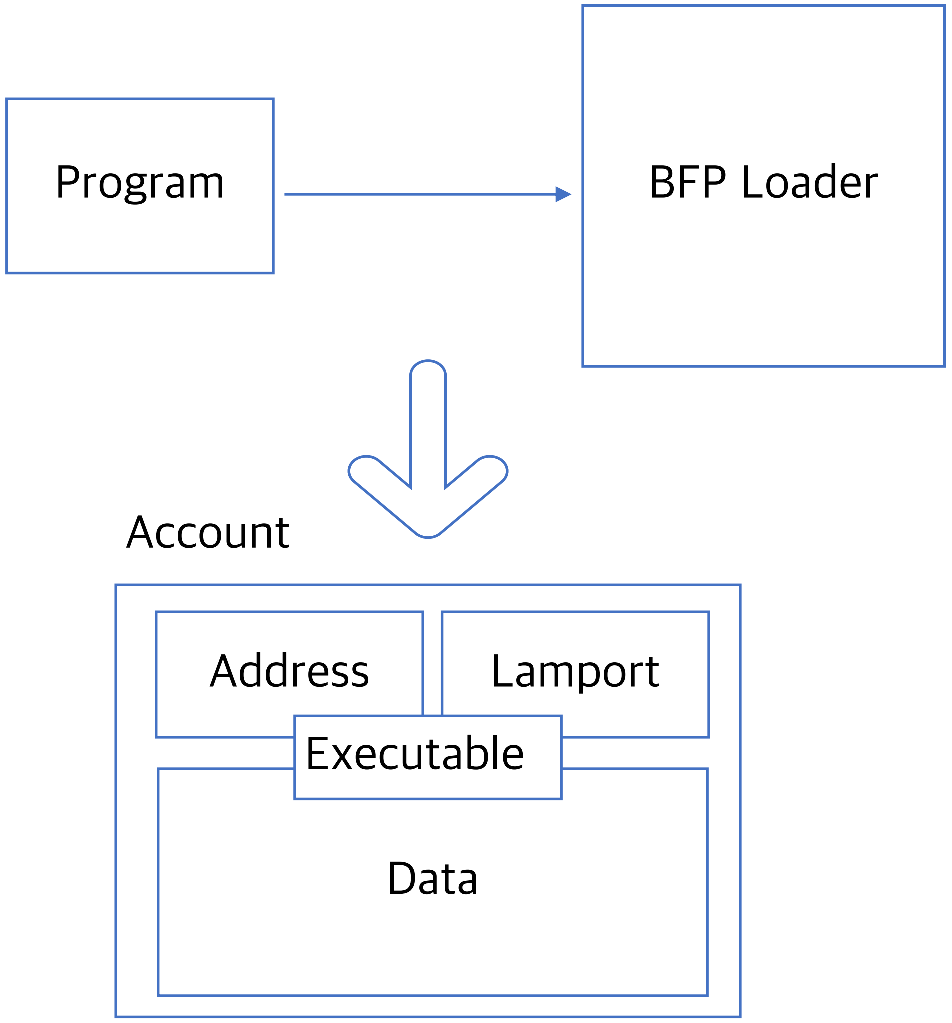
Transactions can indicate that some of the accounts it references be treated as **read-only accounts** in order to enable **parallel account processing between transactions**. The runtime permits read-only accounts to be read concurrently by multiple programs. If a program attempts to modify a read-only account, the transaction is rejected by the runtime.



Executable

If an account is marked "**executable**" in its metadata, then it is considered a **program** which can be executed by including the account's public key in an instruction's **program id**. Accounts are marked as executable during a successful program deployment process by the loader that owns the account.

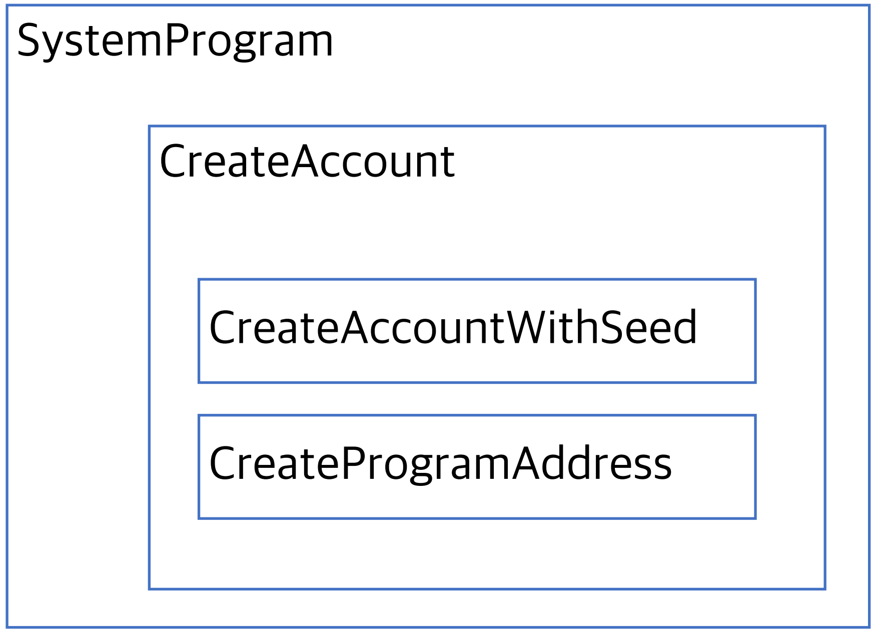
If a program is marked as final (non-upgradeable), the runtime enforces that the account's data (the program) is **immutable**. Through the upgradeable loader, it is possible to upload a totally new program to an existing program address.



Creating

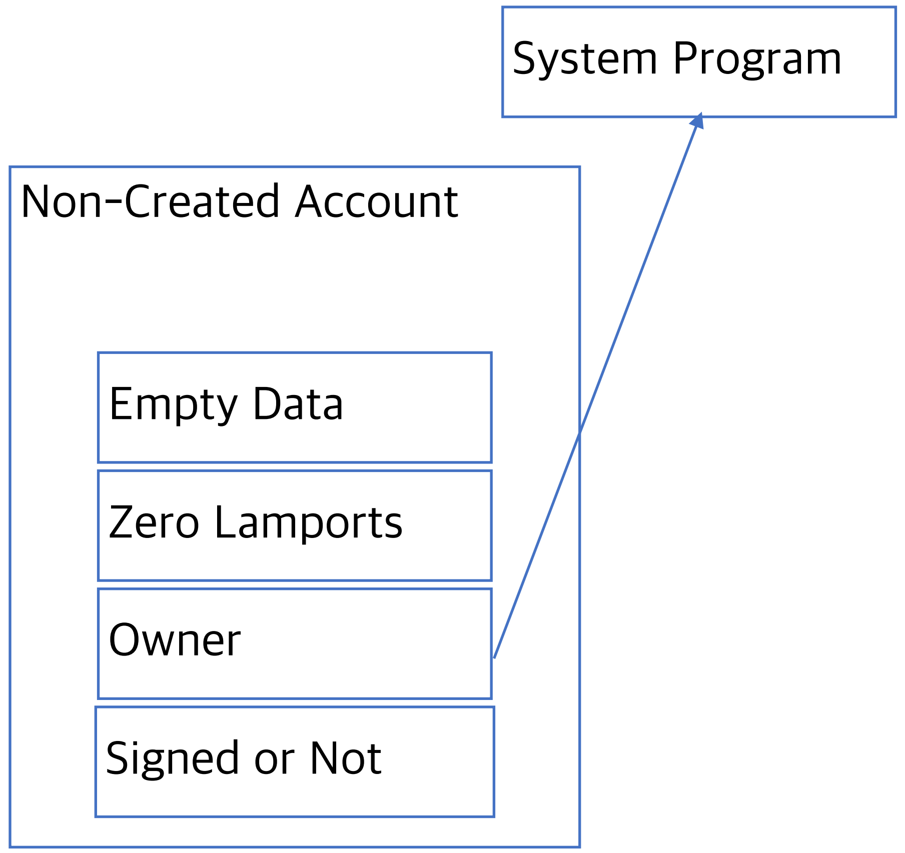
To create an account, a client generates a **keypair** and registers its public key using the **SystemProgram::CreateAccount** instruction with a fixed storage size in bytes preallocated. The current maximum size of an account's data is 10 megabytes.

An account address can be any arbitrary 256 bit value.



Creating (2)

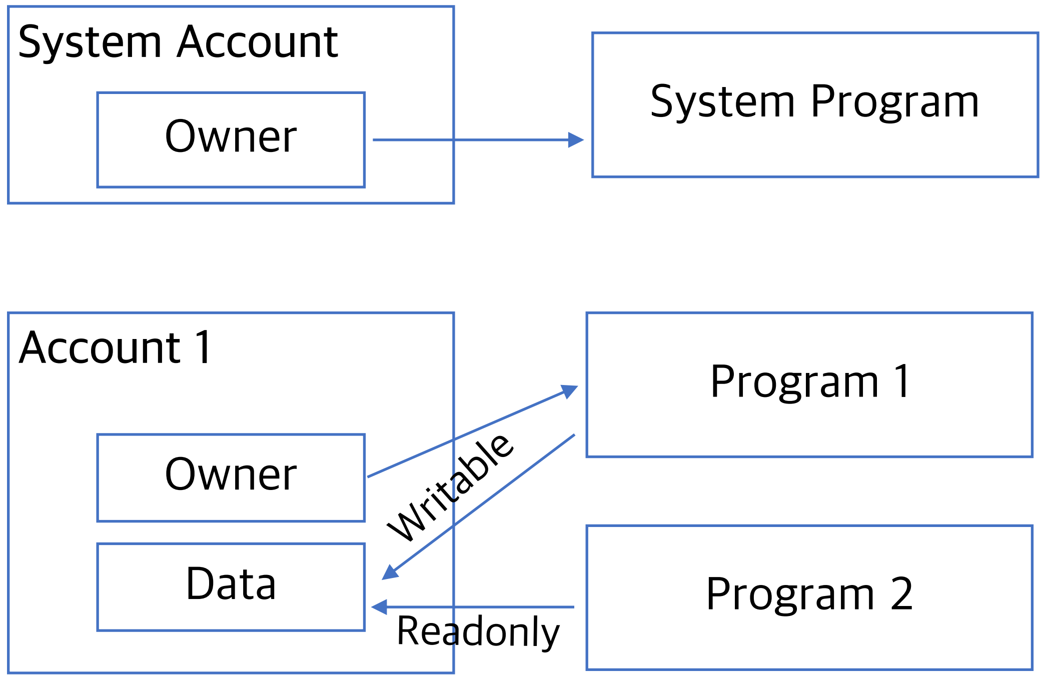
Accounts that have never been created via the system program can also be passed to programs. When an instruction references an account that hasn't been previously created, the program will be passed an account with **no data** and **zero lamports** that is owned by the system program.



Creating (3)

A created account is initialized to be owned by a built-in program called the System program and is called a system account aptly. An account includes "**owner**" metadata. The owner is a program id. The runtime grants the program write access to the account if its id matches the owner.

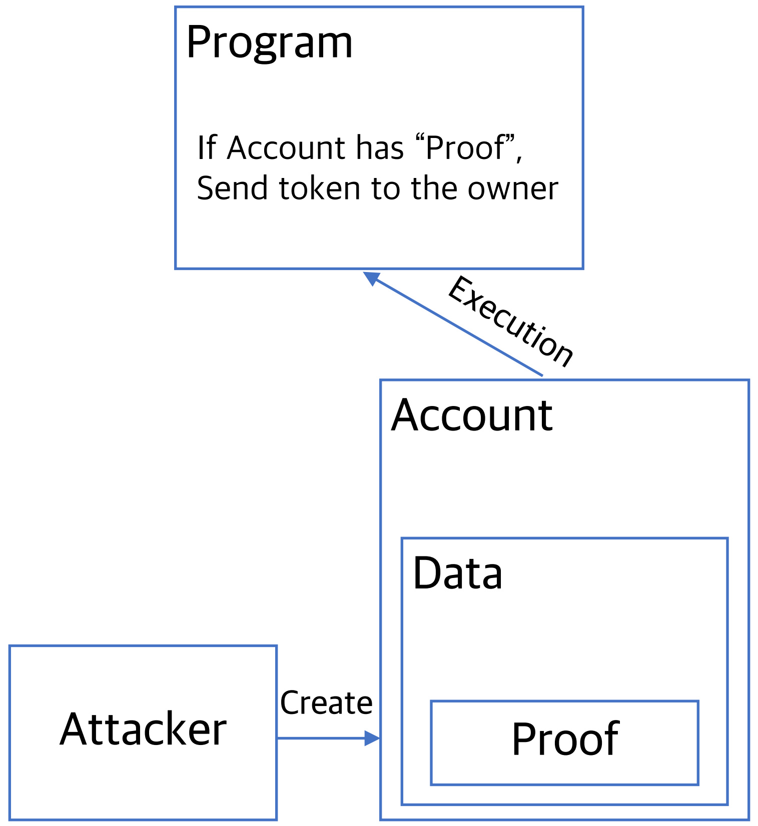
For the case of the System program, the runtime allows clients to transfer lamports and importantly assign account ownership, meaning changing the owner to a different program id. If an account is **not owned** by a program, the program is **only permitted to read** its data and credit the account.



Account Validity

For security purposes, it is recommended that programs check the validity of any account it reads, but does not modify.

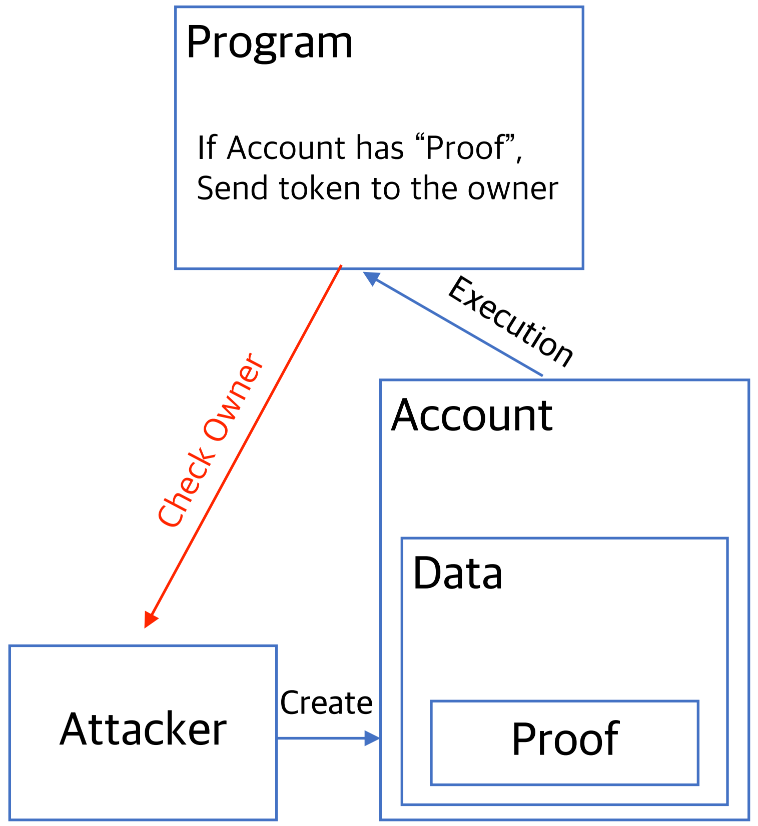
This is because a malicious user could create accounts with arbitrary data and then pass these accounts to the program in place of valid accounts. The arbitrary data could be crafted in a way that leads to unexpected or harmful program behavior.



Account Validity (2)

To check an account's validity, the program should either check the account's address against a known value, or check that the account is indeed owned correctly (usually owned by the program itself).

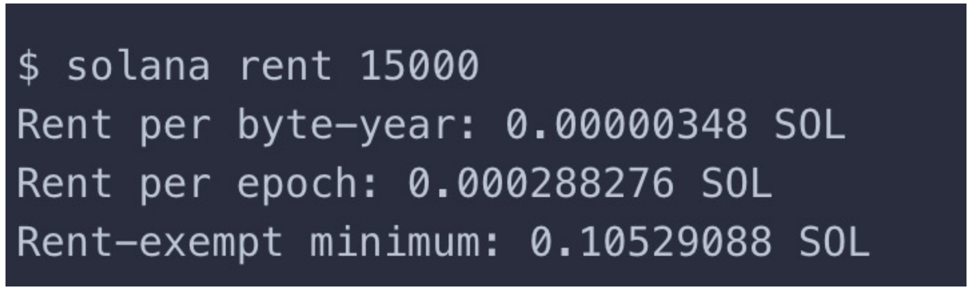
One example is when programs use a sysvar account. Unless the program checks the account's address or owner, it's impossible to be sure whether it's a real and valid sysvar account merely by successful deserialization of the account's data.



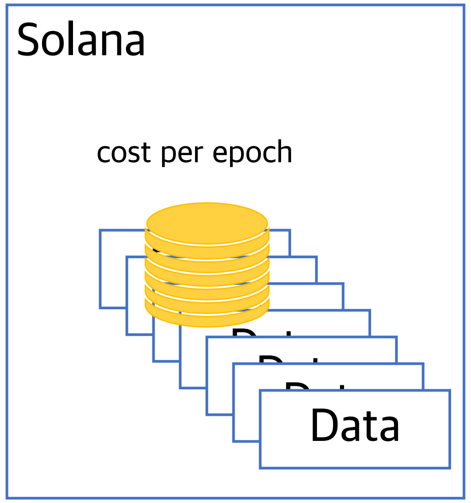
Rent

Keeping accounts alive on Solana incurs a storage cost called rent because the blockchain cluster must actively maintain the data to process any future transactions. This is different from Bitcoin and Ethereum, where storing accounts doesn't incur any costs.

Currently, all new accounts are required to be **rent-exempt**.



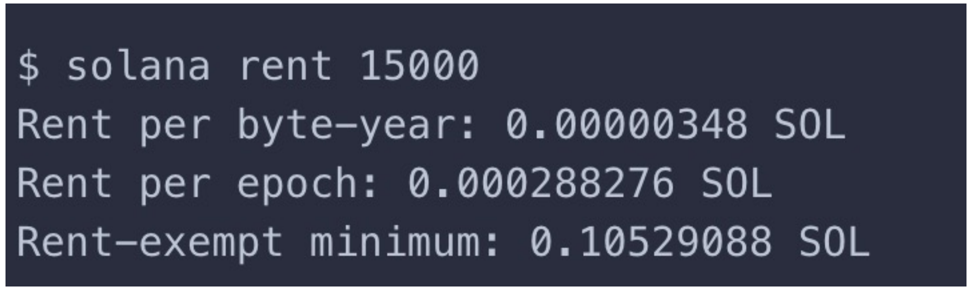
```
$ solana rent 15000
Rent per byte-year: 0.00000348 SOL
Rent per epoch: 0.000288276 SOL
Rent-exempt minimum: 0.10529088 SOL
```



Rent Exemption

An account is considered **rent-exempt** if it holds at least **2 years** worth of rent. This is checked every time an account's balance is reduced, and transactions that would reduce the balance to below the minimum amount will **fail**.

Note: Use the getMinimumBalanceForRentExemption RPC endpoint to calculate the minimum balance for a particular account size. The following calculation is illustrative only.



```
$ solana rent 15000
Rent per byte-year: 0.00000348 SOL
Rent per epoch: 0.000288276 SOL
Rent-exempt minimum: 0.10529088 SOL
```