Anchor Introduction - 01

Constraints

Constraints just give you another lever to pull to add security checks on the accounts being passed in to your program.

You can add constraints to an account with the following format:

```
#[account(<constraints>)]
pub account: AccountType
```

For example, taking a look at the same example from above, we can see the `user` account is utilizing the `mut` constraint to indicate that user should be a mutable account. `my_account` is making use of 3 different constraints that generally go hand in hand.

The `init` constraint tells anchor to create an account at this address, `payer` indicates who is paying for the transaction fees and rent required to create an account, and the space constraint tells Anchor how much `space` to allocate on the account's data field.

```
#[derive(Accounts)]
pub struct Initialize<'info> {
    #[account(init, payer = user, space = 8 + 8)]
    pub my_account: Account<'info, MyAccount>,
    #[account(mut)]
    pub user: Signer<'info>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct Update<'info> {
    #[account(mut)]
    pub my_account: Account<'info, MyAccount>,
}

#[account]
pub struct MyAccount {
    pub data: u64,
}
```

The `#[account(signer)]` checks that the given account signer the transaction. We just learned about the `Signer` account type that does this same thing, why do you think there is also a constraint that verifies an account signed a transaction?

Well, one feature of that `Signer` type is that there are also no other checks on the underlying account, so it's recommended that you do not access the account's data.

The `#[account(signer)]` constraint allows you to verify the account signed the transaction, while also getting the benefits of using the `Account` type if you wanted access to it's underlying data as well.

The `#[account(mut)]` checks the given account is mutable and makes anchor persist any stat changes.

The `#[account(owner = <expr>)]` checks the account owner matches expr.

The `#[account(has_one = <target_account>)]` checks the `target_account` field on the account matches the key of the `target_account` field in the Accounts struct.

```
#[account(mut, has_one = authority)]
pub data: Account<'info, MyData>,
pub authority: Signer<'info>
```

In this example `has_one` checks that `data.authority = authority.key()`.

The `#[account(constraint = <expr>)]` is a constraint that checks where the given expression evaluates to true. Use this when no other constraint fits your use case.

In the previous lesson, we just created an instruction that logged out a message. This was a very simple program so we did not need to pass in any accounts and we also did not need to access accounts in the logic of the instruction - but what if you did want to access the accounts that were passed in? Well, that's where the `Context` object comes in to play.

Each endpoint, or publicly callable instruction, on an Anchor program takes a `Context` type as its first argument. Through this context argument it can access the accounts (`ctx.accounts`), the program id (`ctx.program_d`) of the executing program, and the remaining accounts (`ctx.remainging_accounts`).

Take a look at this example anchor program below. There is an accounts struct called `SetData` that defines the single account this instruction expects as input. Now look at the `set_data` function defined inside the `hello_anchor` program module. The first input parameter to the function is a variable called `ctx` that is of the type `Context<SetData>`. Through this object, you have access to all the accounts defined in the `Accounts` struct wrapped in the `Context`.

While the first argument will always be a `Context` object, any other input parameters can come after the context object.

```
use anchor_lang::prelude::*;
declare_id!("Fg6PaFpoGXkYsidMpWTK6W2BeZ7FEfcYkg476zPFsLnS");

#[program]
mod hello_anchor {
    use super::*;
    pub fn set_data(ctx: Context<SetData>, data: u64) -> Result<()> {
        ctx.accounts.my_account.data = data;
        Ok(())
    }
}
```

```rust
#[account]
#[derive(Default)]
pub struct MyAccount {
    data: u64
}

#[derive(Accounts)]
pub struct SetData<'info> {
    #[account(mut)]
    pub my_account: Account<'info, MyAccount>
}
```

The `Context` provides non-argument inputs to the program.

```rust
pub struct Context<'a, 'b, 'c, 'info, T> {
    pub program_id: &'a Pubkey,
    pub accounts: &'b mut T,
    pub remaining_accounts: &'c [AccountInfo<'info>],
    pub bumps: BTreeMap<String, u8>,
}
```

Because every instruction on a smart contract is unique and will contain different business logic, that generally means each instruction will require/expect different accounts with different constraints to adhere to. For this reason, it is generally best practice to define a unique `Accounts` struct for each individual instruction on the smart contract.