Pre-requisites : *Install rust and git globally onto your machine.*
Use VS Code, IntelliJ or Sublime Text for the Following walkthrough.

Download
bit.ly/start-git
bit.ly/start-rust
bit.ly/start-vscode

Terminal - Run :

```
1. Cargo - - version
2. Git  - - version
```

Step 1 : Please Clone the ALEO HQ Repository through the viscose terminal. :
>> `git clone https://github.com/AleoHQ/workshop`

1. After rust up is finished, copy and paste the command to set the PATH variable to start using rust immediately and then run cargo version to check to see if it works in VS code.

VS Code

Step 2 : *Install the Leo plugin in VSCode.*
Step 3 : Go to the workshop folder >> `cd workshop-master`
Step 4 : install the workshop software >> `./install.sh`
- **this step will take a few minutes to complete.** `(~5 minutes)`

Step 5 : As that is downloading..
- Go to your local workstation and then make a new directory and create a token using the commands found in the LEO CLI. Leo is a CLI tool

1. `>> cd ..`
2. `>> mkdir temp`
3. `>> ls`
4. `>> Leo new token`
5. `>> Cd token`
6. `>> Ls`
7. `>> Code .`
8. Go to source folder and write `>> Leo run`
9. This will synthesize a predicate and compiles the program. The execution takes about 3 seconds.

10. Remove the existing code and write the following code.

```
program token.aleo {
      record token {
            owner: address,
            gates: u64,
            amount: u64
}
      transition mint(receiver: address, amount: u64) -> token {
            return token {
                  owner: receiver,
                  gates: 0u64,
                  amount
            };
      }
} // end of program
```

11. Go back to your Terminal, and build the program.
- `>> Leo build` (*this command compiles the program*)
- Take a look at the build folder and check out the **main.aleo** instructions file.
- Notice that main.aleo is a 'cast instruction'; this register is creating an instance of the **main.leo** token file.
- Notice that we also have **mint.prover** and **mint.verifier** file, these are the Proving and Verifying keys for the ZK-proof.
- Go to **program.json** file, and copy your Aleo address, this was generated upon compilation. *This is your account address*.

12. Now we want to use our token contract and mint tokens into our address.
- `>> Leo run mint` (*`pass in your address you copied`*) `100u64`
- This will produce a new record for you. You can see that you have a record with `owner`, `gates` and `amount` plus a (hidden) `_nonce`. This is a piece of the record that is *deterministically random*.

13. Let's write more code, go back to your main.leo token program.

```
transition transfer(sender: token, receiver: address, amount: u64) ->
(token, token)  {
      //this operation is safe. If under/overflow the program will fail.
      let remaining: u64 = sender.amount - amount

      //payments for the receiver
      let payment: token = token {
            owner: receiver,
            gates: 0u64,
            amount
      };

      //send the balance back to the receiver
      let remainder: token = token {
            owner: sender.owner,
            gates: sender.gates,
            amount: remaining
      };
      return(payment, remainder):

} // end of transition

} // end of program
```

NOTE : The high level overview of this program is that it's behaving like an Erc20 contract. I wouldn't want to send free money that is not the token itself to the person (**receiver**). In this case, "We are keeping the gates to myself"; this is effectively keeping any balance in the record for myself. We are issuing a new currency for the *receiver*. We wouldn't want to give someone free Ether! We want to give the tokens we want to give them.

14. Let's build this program by running it. This will automatically compile the program.

```
>> Leo run
```

- Make another address we want to transfer too

```
  >> aleo account new
```

- Go back to our token program terminal and transfer to the new account.

```
>> Leo run transfer "{paste in the output record we minted for ourselves in the previous "Leo run step"}" Paste the new account address "aleo account new" 10u64
```

— It will transfer money to the other address. The first is the payment, and the second is the remainder.

Step 6: Congratulations! You built and ran your first Leo program! Now we can try running the other workshop demonstrations by changing into the other workshop directories (`>> cd ..`) like **tic-tac-toe**, **battleship**, **vote.leo** …
We may run the scripts to play with the programs to watch how they function.

```
>> ./run.sh
```

Summary of Tic-Tac-Toe Workshop

– The programs will compile and run. You will see the output of the program. The inputs Wille execute themselves through transitions. You are allowed up to 15 transitions off chain before the proof of the program is committed on chain. Tic-tac-toe is happening purely off chain. You have 15 transitions in 1 transaction. You can play games peer-to-peer and later broadcast the outcome of that program with others on-chain. Essentially, you are forcing the other player to play based off of your board state, because you are giving them the execution in a ZKP; they cannot cheat off that logic. What If you wanted to withhold from the chain this particular broadcast? Well, you could create a challenge that forces the players to execute logic that will or won't commit logic on chain after an (x) amount of transitions. You can say things like "if this program isn't executed within 100 blocks, then the money is returned to both players executing the program. In poker, we could deal cards and play 1 round (1 transaction) that checks for "bet" for example, and then simulate transactions 1 round at a time proceeding thereafter. This is a different kind of computing construct and something that may be familiar if you are coming from Crypto. If you're used to programming with functional programming paradigms of logic, then this process will make much more sense to you.

Summary of Bank workshop

– `mapping :` "on chain storage", I am holding some unique Identifier, which is a field element (some number) and also some amount of balance with respect of that field element.
– `finalize:` " every transition has an optional scope", I can call the finalize to increment the balance and wait to execute new logic before the next transition.
– We encourage feedback on the **block height** wait times and changing the syntax of how to properly use the opcodes in Arcs for issuing Off/On-chain executions (Help Aleo/Leo figure out the right syntax to use) .

Summary of the Vote workshop

– Illustrates the use of `structs`, with the use of fields to keep the struct "hash-based".
– the `record` is the vote
— The `mappings` are the state and tally of the voting record.
– The creation of tickets and the agreement and disagreement logic for voting constructs are hardcoded into the program.

Summary of the Auction workshop

– There's a bid with a bidder. It's a managed auction, so the auctioneer will denote who is the winner of the auction at the end of the program.
— You could have a parent program that does an inter-programatic call that we could import and use which then could check `bid.is_winner` is **TRUE** and if so, then give the winner that asset.

Summary of the Battleship workshop

— Player 1 places their pieces and encrypts it under their address. Player 2 does the same. Player 1 is passed back the board with encrypted pieces and then committing to firing at the Player 2 and checks to see if the message does or does not **hit** the Player 2 accordingly. There are 4 pieces of different sizes. The full game happens within 14 rounds, which is just under 15 transitions. Meaning, the entire game happens in a single transaction.