Chapter 1 Day 1

1) The Blockchain is a gigantic database, built on decentralized networks, that people can access publicly and anywhere from the world online.

2) A Smart Contract is a computer program that allows a user to interact with the Blockchain to accomplish a wide variety of actions for a specific purpose: for example, make a money transaction such as paying for a good-service or giving donations to a cause.

3) A Script is also a program that reads the data on the Blockchain and lets the user view what's currently stored in the Blockchain. **It's complete free of charge to run Scripts.**
A Transaction is the interaction between a Smart Contract and the Blockchain that the user initiated, and this modifies the data stored in the Blockchain. **As a result of this modifying, making transactions costs money.**

Day 2

1) The 5 Pillars of Cadence are:
* Safety and Security
* Resource-Oriented Programming
* Clarity
* Approachability
* Developer Experience

2) These 5 Pillars are useful because:
* Users won't make any transactions that uses a technology they don't trust. And developers won't program here either for the exact same reason (no security and safety == no trust == no people involvement).

* No response required for this point.

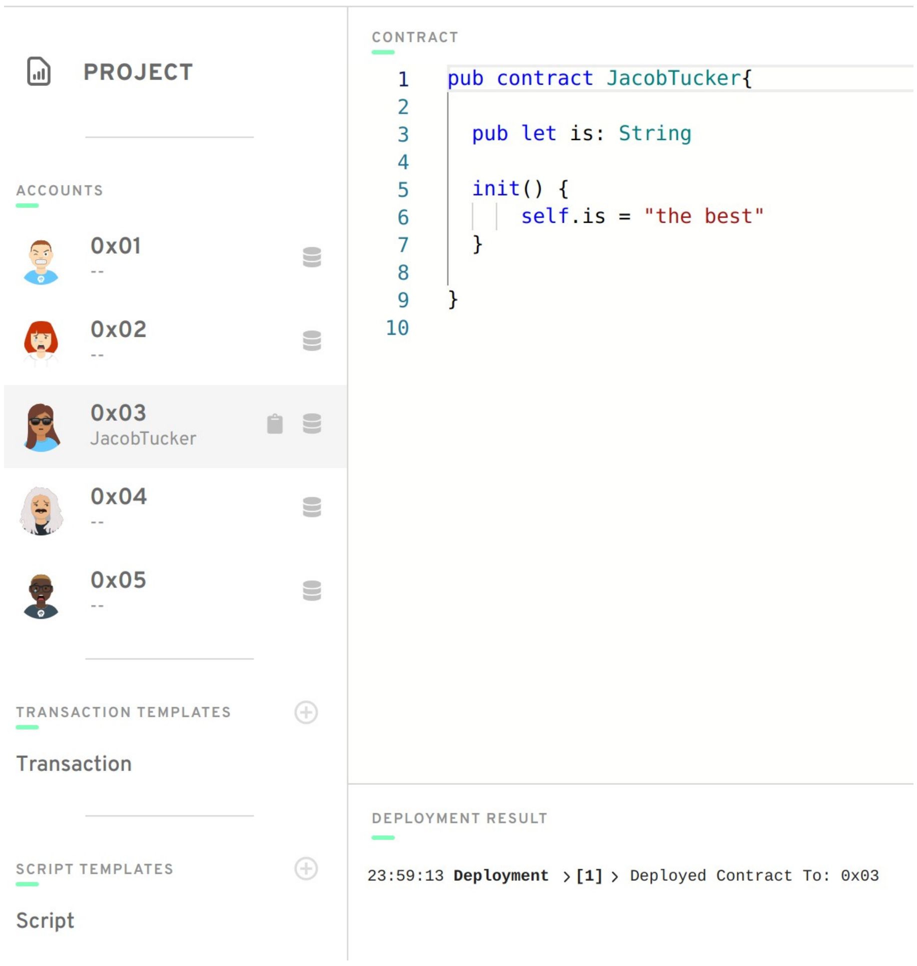
* Clarity goes hand-in-hand with transparency, understanding and trust. If people, including developers and users, clearly see what’s going on when they are interacting with the Flow Blockchain, they will understand how it works, see the benefits of this technology, and definitely use it since now they know it better and, therefore, trust it.

* Approachability – People adopt anything they find familiar with greater ease and way less resistance, since they can relate to is as something they already know (at least to some extent).

* Making the programming ecosystem reliable and easy to follow allows developers to have a way more pleasant experience. As a result of this nicer experience, more developers will stay to further contribute to Flow as well as invite new programmers to join the dev team and ecosystem.

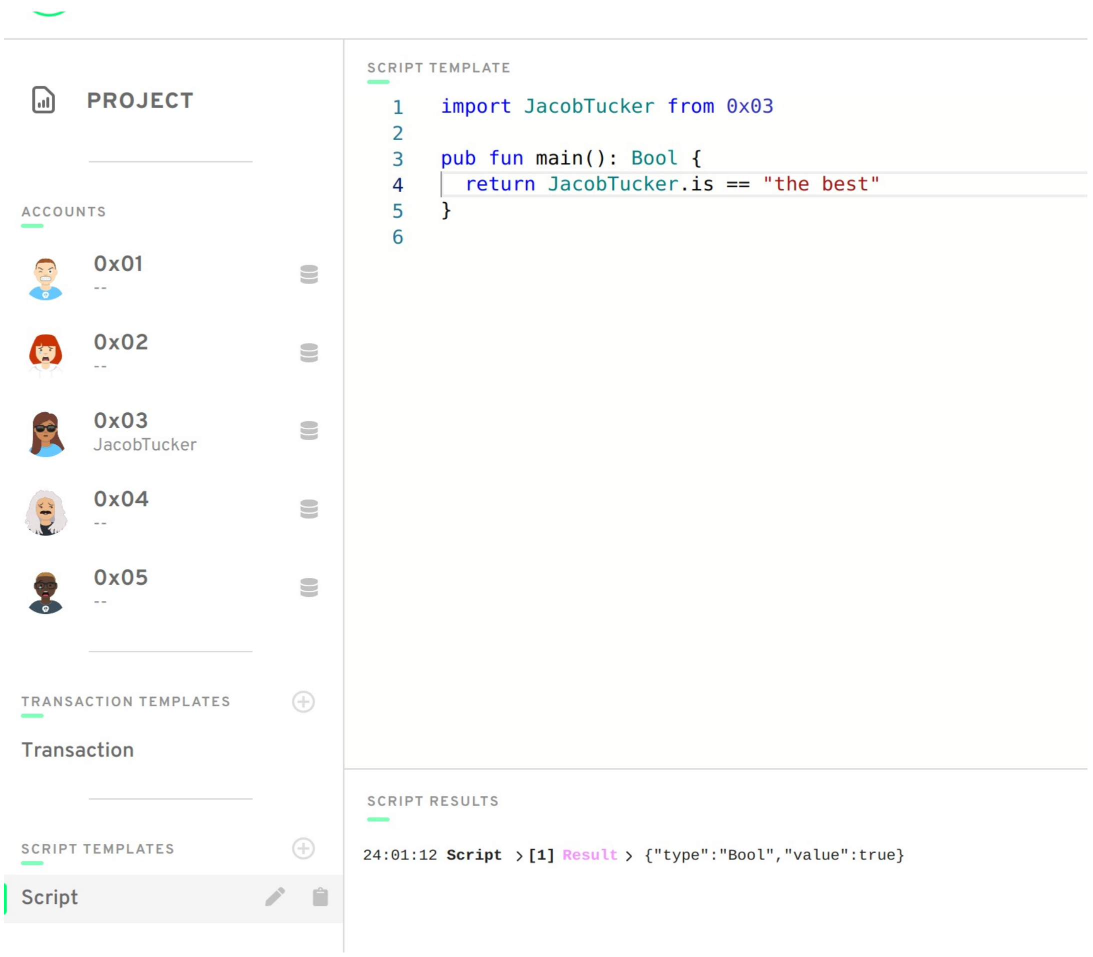
Chapter 2 Day 1

1 and 2)



```
pub contract JacobTucker{

  pub let is: String

  init() {
      self.is = "the best"
  }

}
```

```
23:59:13 Deployment  > [1] > Deployed Contract To: 0x03
```



```
import JacobTucker from 0x03

pub fun main(): Bool {
  return JacobTucker.is == "the best"
}

```

```
24:01:12 Script  > [1] Result > {"type":"Bool","value":true}
```

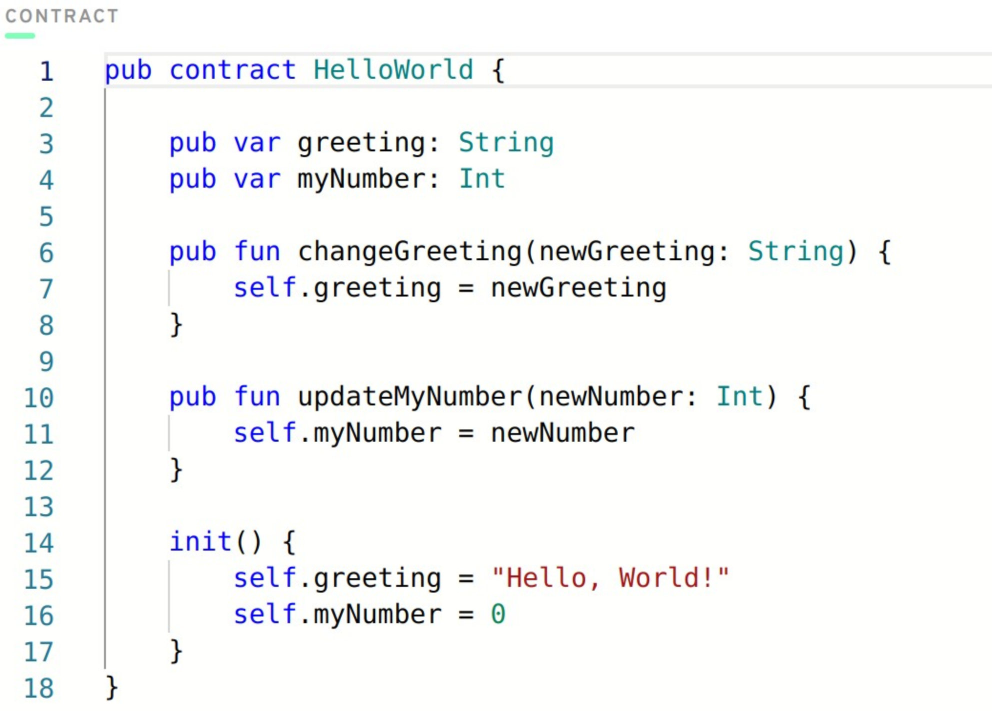
Day 2

1) We cannot call the function changeGreeting inside the script, since it changes the information on the Blockchain (as the name of function points out). Only transactions can make any data changes.

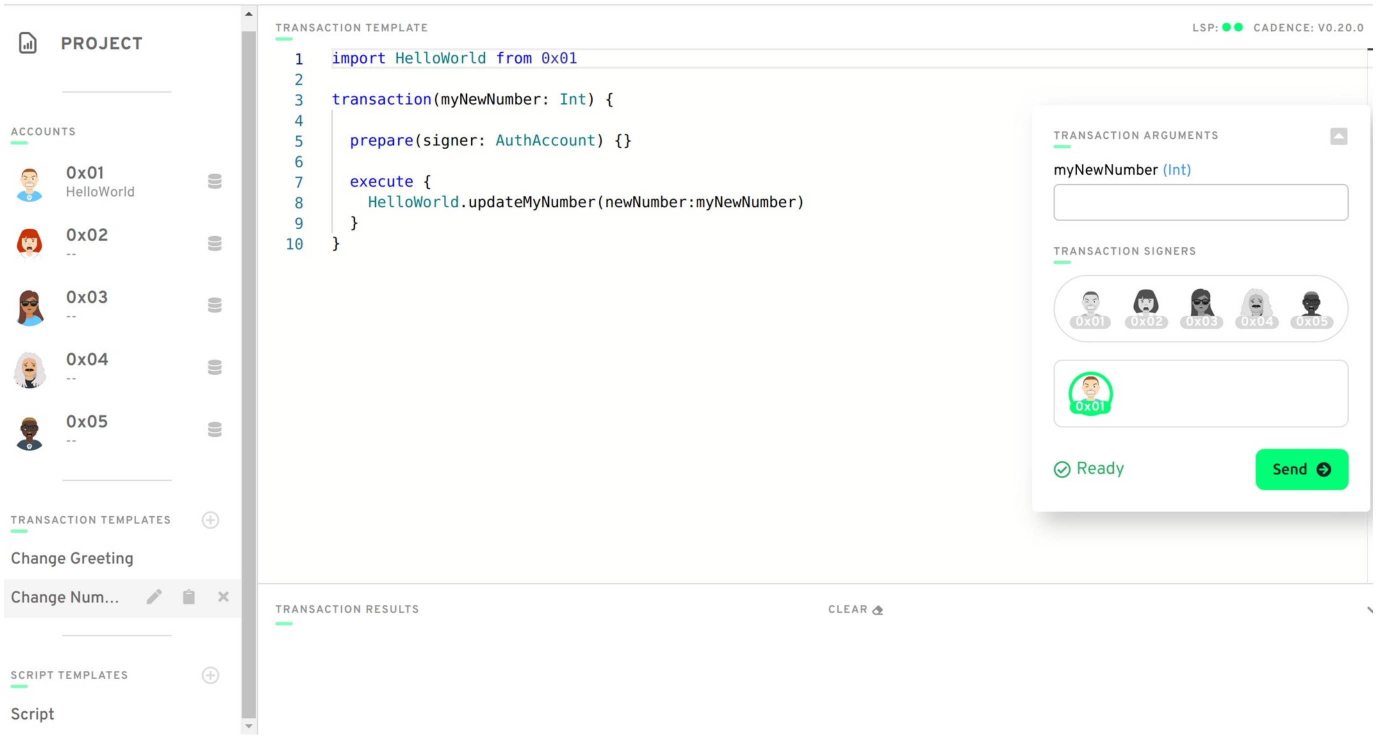
2) AuthAccount refers to the account of the person that started the transaction. We need to know this info in the prepare phase, so we know who is requesting this transaction and be able to send them the info they are requesting (given that the transaction is indeed completed successfully and approved).

3) The prepare phase is where
The execute phase is where functions written inside the smart contracts are actively called once a user starts a transaction (this is the “do phase).

4)

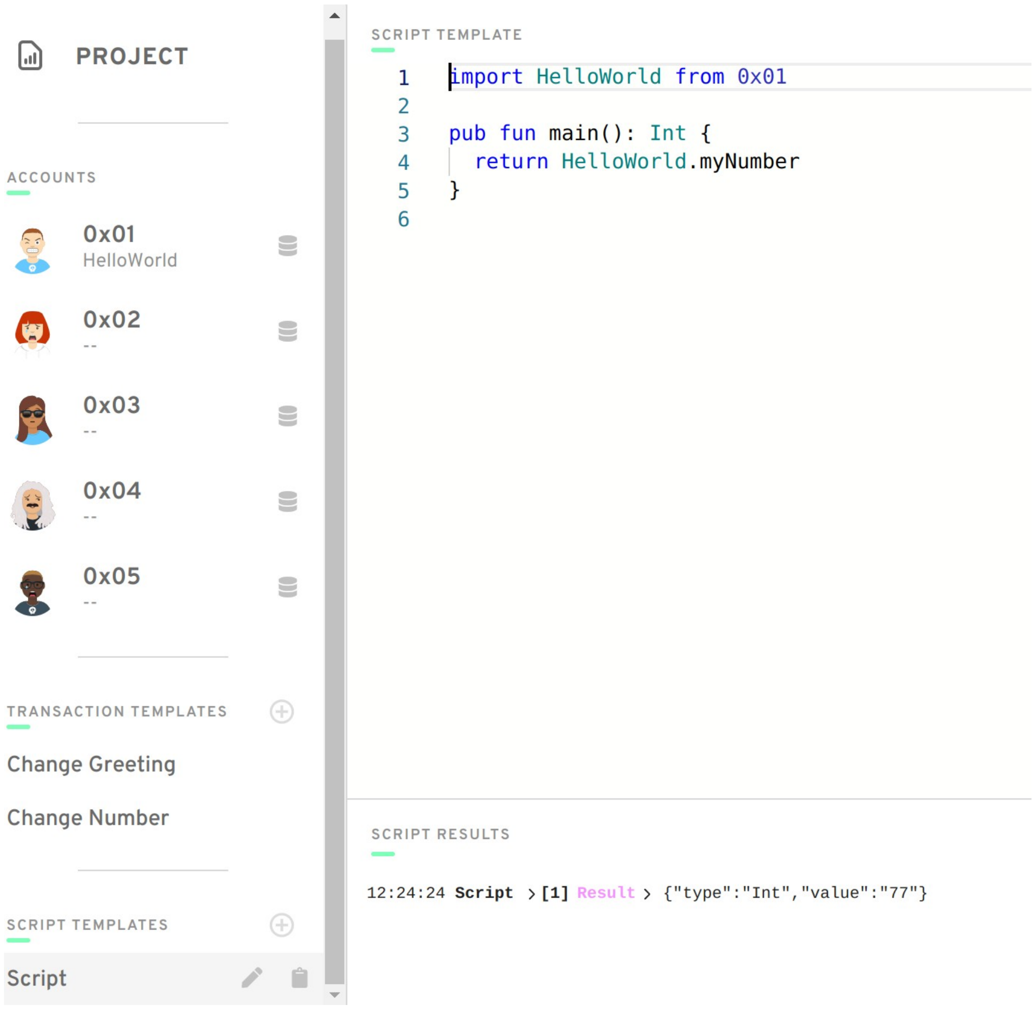


```
pub contract HelloWorld {

    pub var greeting: String
    pub var myNumber: Int

    pub fun changeGreeting(newGreeting: String) {
        self.greeting = newGreeting
    }

    pub fun updateMyNumber(newNumber: Int) {
        self.myNumber = newNumber
    }

    init() {
        self.greeting = "Hello, World!"
        self.myNumber = 0
    }
}
```



```
import HelloWorld from 0x01

transaction(myNewNumber: Int) {

  prepare(signer: AuthAccount) {}

  execute {
    HelloWorld.updateMyNumber(newNumber:myNewNumber)
  }
}
```

myNewNumber (Int)

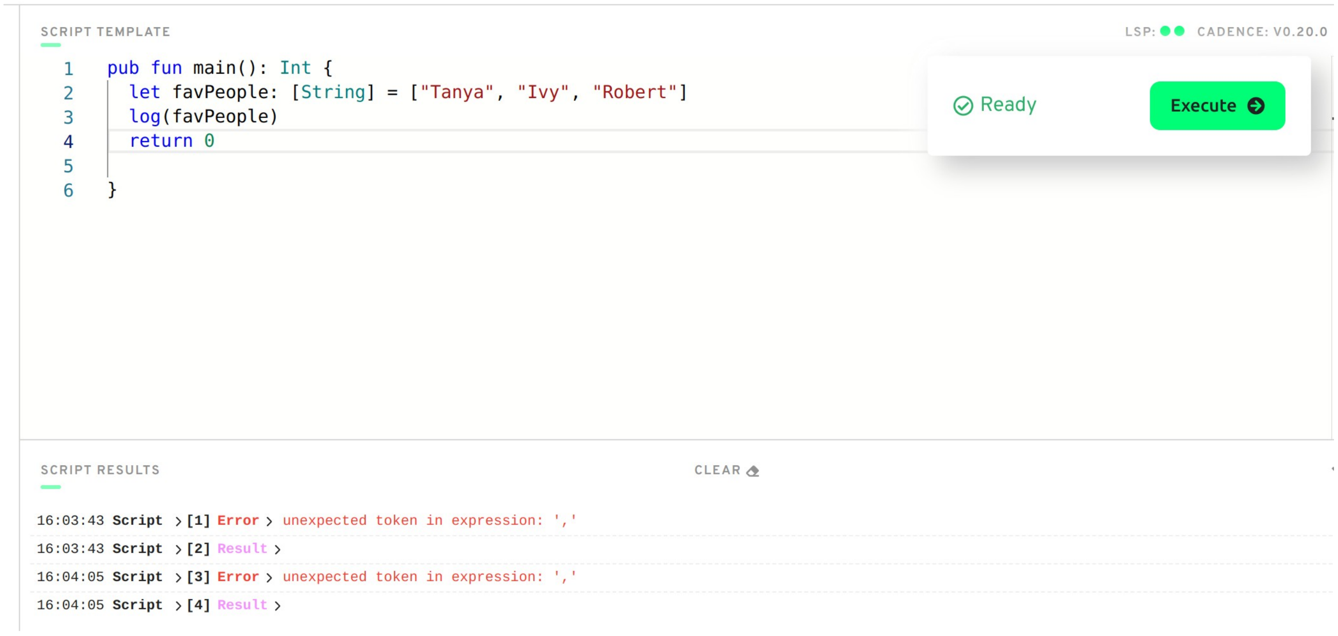


```
import HelloWorld from 0x01

pub fun main(): Int {
  return HelloWorld.myNumber
}

```

```
12:24:24 Script  > [1] Result > {"type":"Int","value":"77"}
```

Day 3

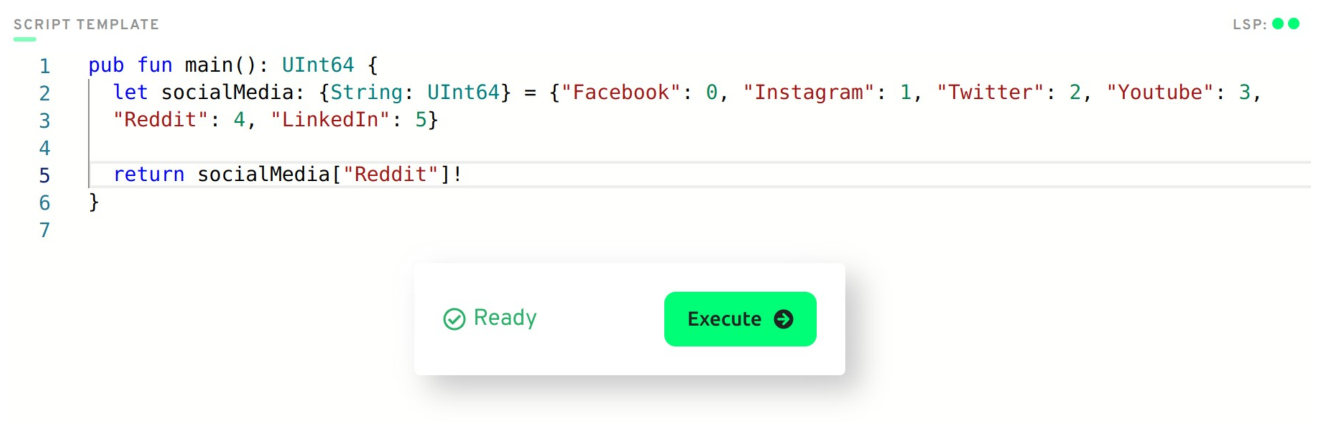
1) **Note:** Checked with Jacob and code is correct. Refreshed multiple times, but I still get the same script results. Playground is just being glitchy.



```
pub fun main(): Int {
  let favPeople: [String] = ["Tanya", "Ivy", "Robert"]
  log(favPeople)
  return 0

}
```

```
16:03:43 Script  > [1] Error > unexpected token in expression: ','
16:03:43 Script  > [2] Result >
16:04:05 Script  > [3] Error > unexpected token in expression: ','
16:04:05 Script  > [4] Result >
```

2)



```
pub fun main(): UInt64 {
  let socialMedia: {String: UInt64} = {"Facebook": 0, "Instagram": 1, "Twitter": 2, "Youtube": 3,
  "Reddit": 4, "LinkedIn": 5}

  return socialMedia["Reddit"]!
}
```

3) The force-unwrap operator "forces" the program to only see and return the actual value of a dictionary, instead of "letting the program choose" (in this case, the program can return either the key or the actual value). But what we want is only the value, hence this operator tells it directly or "forces" it to do so.

Example:

```
pub fun main(): String{
    let summerMonths: {UInt64: String} = {6: "June", 7: "July", 8:
"August"}
    return thing[7]! // we added the force-unwrap operator, and it
returns "July"
}
```

4) * The error message means that we have a type mismatch: meaning, what the function returns is different from what it is expected to return.

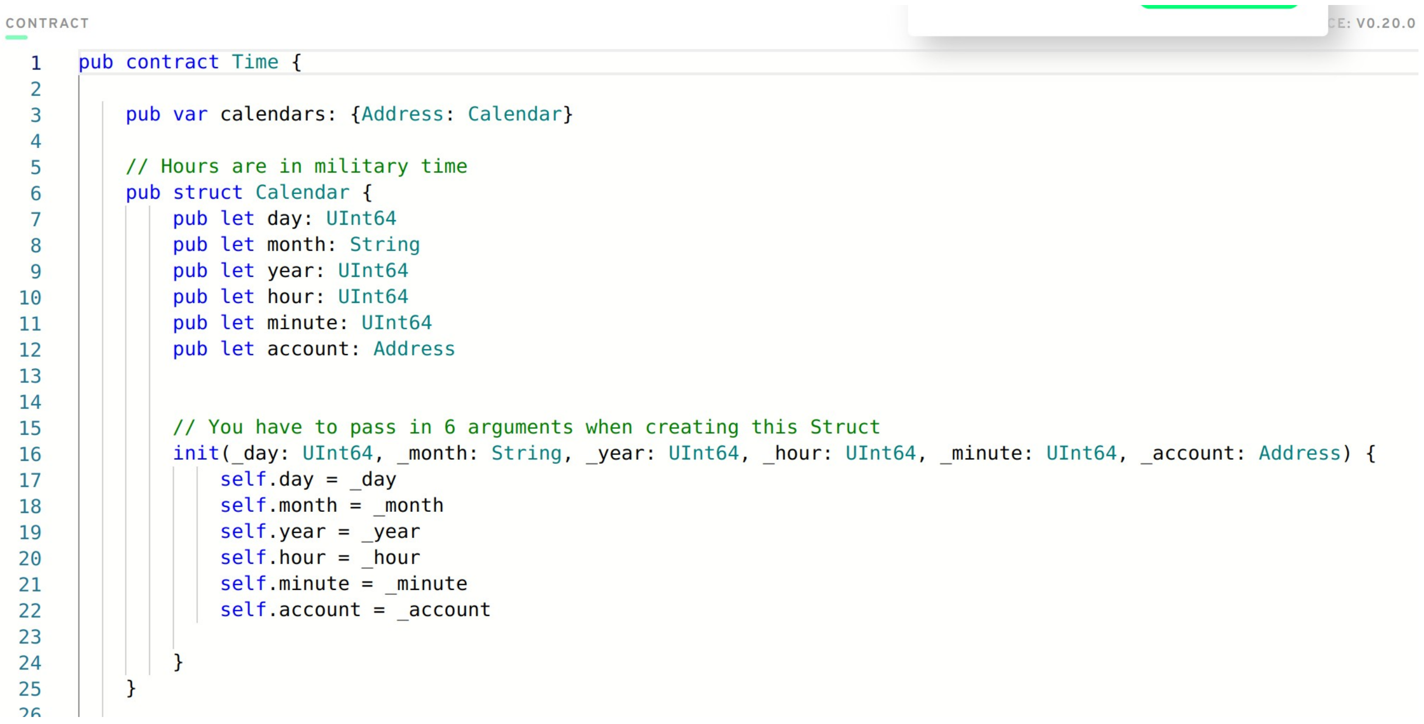
* Since we have a dictionary, we have 2 types (1 from the key and 1 from the value). In this example, the value's and key's types are different (but they can technically be the same). Because we want to return the value's type, we need to add the operator "!" at the end of the return statement. This tells the program to only see and return the value's type, instead of the other option in the dictionary (the key's type).

* We would change the return line in the script to:

return thing[0x03]!

Day 4

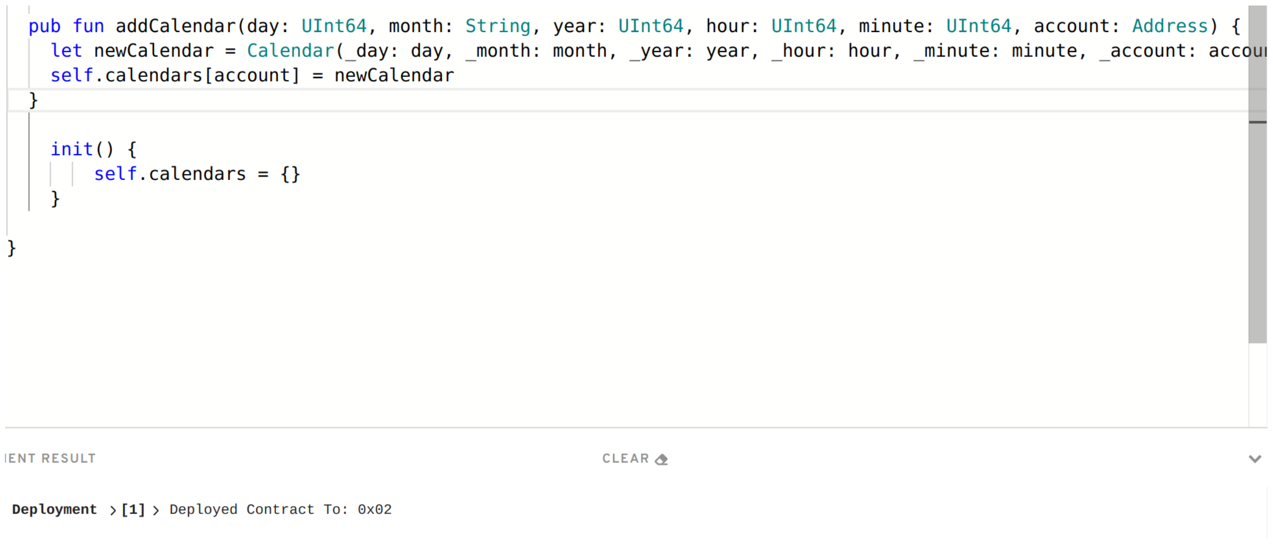
1 - 5)



CONTRACT

CE: V0.20.0

```
pub contract Time {

    pub var calendars: {Address: Calendar}

    // Hours are in military time
    pub struct Calendar {
        pub let day: UInt64
        pub let month: String
        pub let year: UInt64
        pub let hour: UInt64
        pub let minute: UInt64
        pub let account: Address


        // You have to pass in 6 arguments when creating this Struct
        init(_day: UInt64, _month: String, _year: UInt64, _hour: UInt64, _minute: UInt64, _account: Address) {
            self.day = _day
            self.month = _month
            self.year = _year
            self.hour = _hour
            self.minute = _minute
            self.account = _account

        }
    }
```

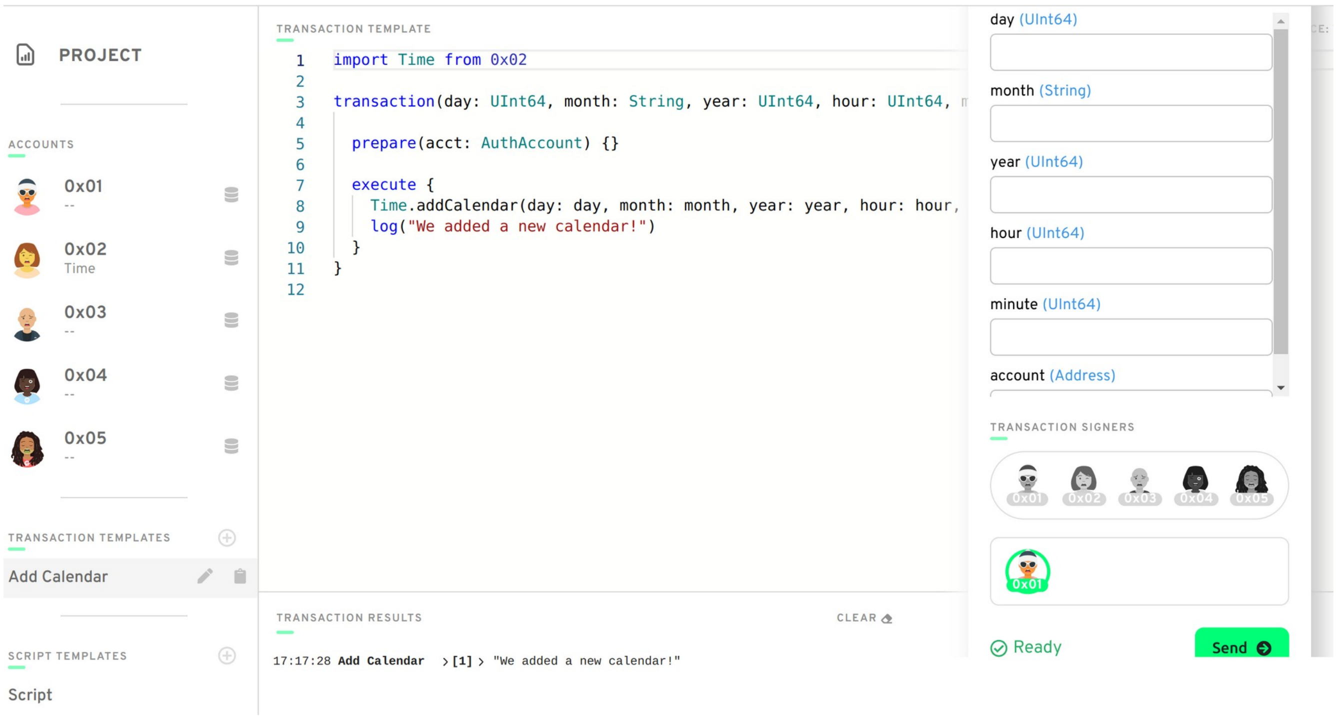


```
    pub fun addCalendar(day: UInt64, month: String, year: UInt64, hour: UInt64, minute: UInt64, account: Address) {
      let newCalendar = Calendar(_day: day, _month: month, _year: year, _hour: hour, _minute: minute, _account: accou
      self.calendars[account] = newCalendar
    }

      init() {
          self.calendars = {}
      }

}
```

ENT RESULT

CLEAR

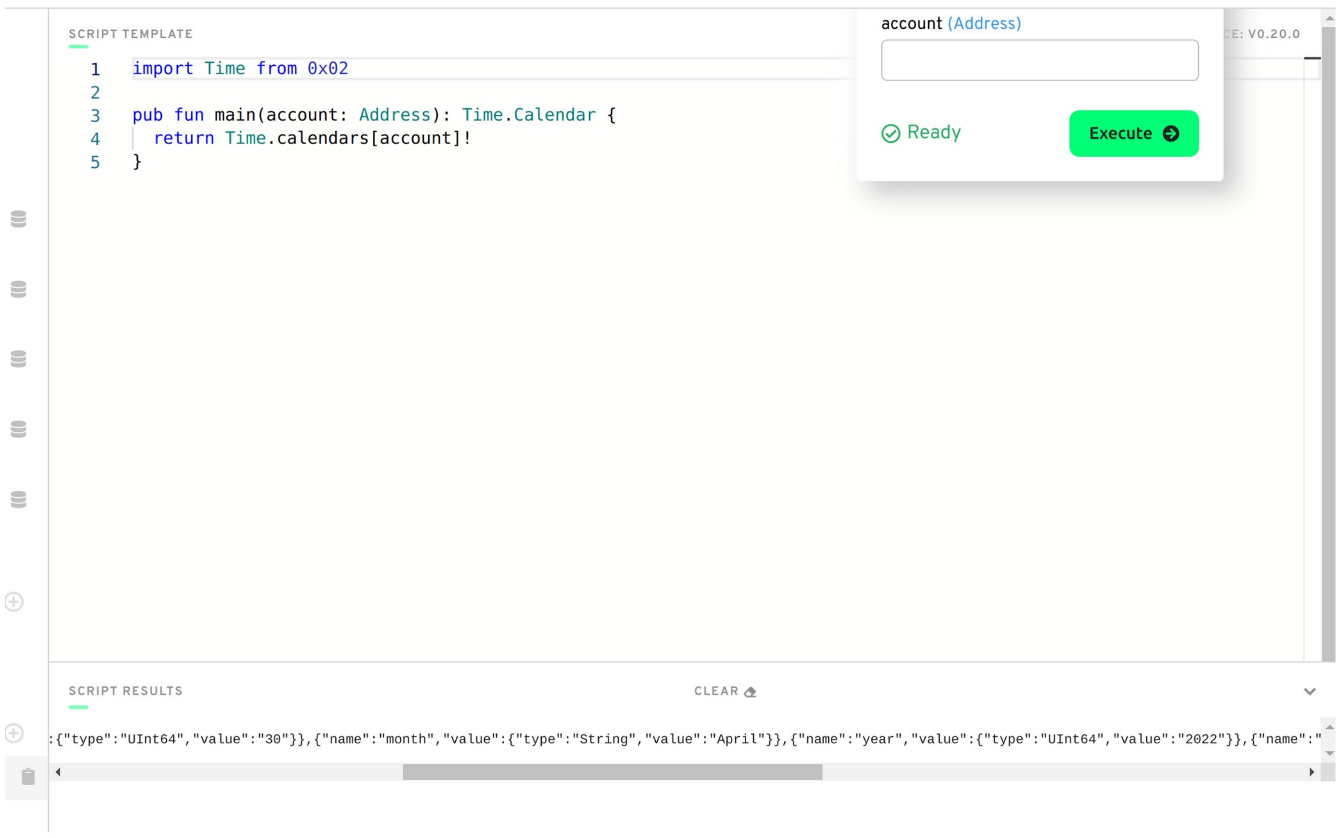
```
Deployment  > [1] > Deployed Contract To: 0x02
```



```
import Time from 0x02

transaction(day: UInt64, month: String, year: UInt64, hour: UInt64,

  prepare(acct: AuthAccount) {}

  execute {
    Time.addCalendar(day: day, month: month, year: year, hour: hour,
    log("We added a new calendar!")
  }
}
```

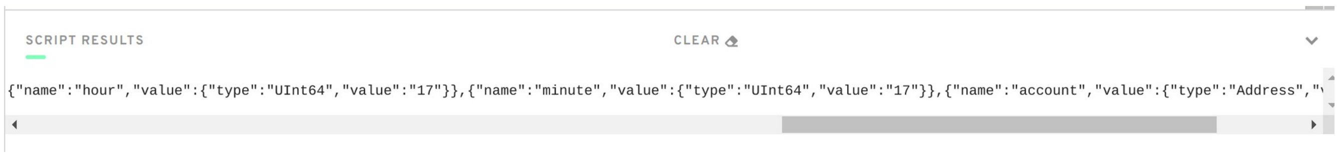
day (UInt64)
month (String)
year (UInt64)
hour (UInt64)
minute (UInt64)
account (Address)

```
17:17:28 Add Calendar  >[1] > "We added a new calendar!"
```

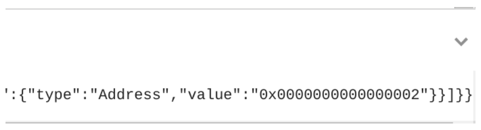


```
import Time from 0x02

pub fun main(account: Address): Time.Calendar {
  return Time.calendars[account]!
}
```

```
:{"type":"UInt64","value":"30"}},{"name":"month","value":{"type":"String","value":"April"}},{"name":"year","value":{"type":"UInt64","value":"2022"}},{"name":"
```



```
{"name":"hour","value":{"type":"UInt64","value":"17"}},{"name":"minute","value":{"type":"UInt64","value":"17"}},{"name":"account","value":{"type":"Address","v
```



```
':{"type":"Address","value":"0x0000000000000002"}}]}}
```

Chapter 3 Day 1

1) * Resources cannot be copied, while Structs can.

* Resources cannot be lost or overwritten, while Structs can.

* We must explicitly do something with a Resource every step of the way, while we don't have to do this for Structs.

2) In the context of NFTs, resources are far a better and safer choice than structs. Becuase NFTs are limited, original works and thus very valuable, we do not want them to get lost, copied, etc., to avoid scams, buyers losing their NFTs somehow, and the people involved losing lots of money.

3) The keyword to make a new resource is ***create.*** And it is used like this:

let myNFT <- create NFT()

4) I would say no, resources cannot be explicitly created inside transactions and scripts. Or it is simply bad practice to do this: for safety and security, all logic is separated into 3 places (smart contracts, scripts and transactions). This is why we have these 3 in the Playground to begin with.

5) It's type Jacob! :D

6)

1st line: it must be **pub fun createJacob: @Jacob** to communicate it is of type Resource

```
        pub fun createJacob(): Jacob { // there is 1 here
```

2nd line: resources cannot be copied (so this line is invalid) and resources need to be explicitly created like so **let myJacob ← create Jacob()**.

```
        let myJacob = Jacob() // there are 2 here
```

3rd line: it must be **return ← myJacob** since we use this operator to handle resources**.**

```
        return myJacob // there is 1 here
    }
```