chasechappy Move videos up ... 10 days ago History

..	
readMe.md	10 days ago
scratchpad.cdc	10 days ago

readMe.md

A Note Before You Start

We cannot wait for you to join this community of builders. Before we kick things off, **please watch our Welcome Video**, which will give you an overview of what to expect in the Fast Floward bootcamp and two pieces of advice to help guide you through the next three weeks.

Fast Floward | Week 1 | Day 1

Hello there! My name is Morgan and together with the team at Decentology we'll guide you towards becoming a decentralized application developer during the 3 weeks of FastFloward.

When exploring something new, it's important to limit oneself in terms of scope. There's only so much time and an infinite amount of material, so we'll focus on things that lead towards shipping decentralized applications or DApps the fastest.

To build any App, one of the first decisions is to choose a platform you'll build on. Same with DApps. We've made this decision for you, and our platform of choice is Flow and Cadence. I'll explain why we think this is a good decision to get started.

Videos

- Introduction and Flow Programming Environment
- Cadence Syntax and Basic Types
- Cadence Functions and Composite Types
- Playing around with Cadence + Day 1 Quests

Flow

Flow is a blockchain that's efficient, fast, reliable, and it supports smart contracts. It is designed by developers for developers with tons of tools and resources made available. You can go from zero to executing your first contract in a few minutes, without having to spend loads of time correctly setting up your environment.

Flow has an innovative architecture, and you can learn more about that by going to onflow.org. For the purposes of our Bootcamp, we only need to understand how to communicate with the Flow blockchain, and as such, we won't dig deeper into how it works.

Environment

Let's get started by setting up our environment. Flow has a command-line utility that enables us to interact with the blockchain. I'll show you how to do it both on Windows and Linux/macOS.

Linux/macOS

Following the documentation it's a simple one-liner.

```
sh -ci "$(curl -fsSL https://storage.googleapis.com/flow-cli/install.sh)"
```

But make sure to include `flow` in your `$PATH`. After doing all that, reload your shell settings.

Windows

Again, following the documentation, make sure you have **PowerShell** on your version of Windows. Search for *"PowerShell"* and once it's open, run this command.

```
iex "& { $(irm 'https://storage.googleapis.com/flow-cli/install.ps1') }"
```

Testing

After you've successfully installed `flow-cli` you should be able to run the version command.

```
flow version
```

As of this recording, it should display `v0.26.0`.

```
Version: v0.26.0
Commit: 5cac45ba37572dfe4279d9ad26019950ef53b3c8
```

To take it a step further, let's execute our first **Cadence** command.

```
flow cadence
```

We're greeted by a command-line prompt.

```
Welcome to Cadence v0.18.0!
Type '.help' for assistance.

1>
```

Let's say hi to the World!

```
log("Hello, World!")
```

The response should say:

```
"Hello, World!"
()
```

I'm going to be using **VS Code** as my code editor, the team at Flow have created an extension for VS Code that enables syntax highlighting, type checking, and more. To install it locally, as per their documentation, simply run this command.

```
flow cadence install-vscode-extension
```

Now that we're all set up, we can dig deeper into **Cadence**, the **Flow** smart contract programming language. Going forward, I'll be working in a Linux environment, but everything should work the same in both macOS and Windows.

Cadence

Cadence is a resource-oriented programming language that you will use to write smart contracts for the **Flow** blockchain. A smart contract is simply a program that executes on the blockchain.

We can start executing Cadence code using the Cadence language server (a REPL shell), which we used earlier to print `"Hello, World!"` . Same command can be used to execute whole files, but on that later.

```
flow cadence [filename]
```

Let's begin by learning about Cadence syntax. Please use the documention as your reference.

Syntax

```
// One-line comment
/* Large /* nested */ comment */
```

Same as with most other programming languages, when naming things you may start with upper or lowercase letters `A-Z, a-z` or an underscore `_` . Afterwards you can include numbers `0-9` too.

```
test1234 // cool
1234test // not cool
(-_-) // not cool
```

Semicolons `;` are optional, except for when you put two or more declarations on the same line.

You declare variables with `var` and constants with `let`. When declaring a variable you have to initialize it.

```
var counter = 10
counter = 11
let name = "Morgan"
var bad
```

Everything in Cadence has a type, inferred or explicit.

```
var isGood: Bool = false
isGood = true // duh!
isGood = 42
```

Types

Cadence has a number of useful types. Let's take a look at some of them.

Integers

```
123
0b1111 // props if you know what decimal number this is!
0o17 // definitely not a decimal 17
0xff // how about this one?
1_000_000_000 // one BILLION
```

All of these integers are inferred as `Int` s, these can represent arbitrarily large signed integers. If you want to be more specific, you can use `Int8`, `Int16`, etc. All `Int` and `UInt` types check for overflow and underflow.

```
var tiny: Int8 = 126
tiny = tiny + 1
tiny = tiny + 1
```

Cadence will not allow your integers to be assigned values outside of their range. This protects us as developers from costly overflow mistakes.

Integers have a couple of methods.

```
let million = 1_000_000
million.toString() // "1000000"
million.toBigEndianBytes() // [15, 66, 64]
```

Fixed-Point Numbers

Cadence has `Fix64` and `UFix64` to represent fractional values, which are essentially integers with a scaling factor, in this case with the scaling factor of `8`.

```
let fractional: Fix64 = 10.5
```

Addresses

With Cadence, you'll constantly be interacting with accounts, and you can reference them using the `Address` type.

```
let myAddress: Address = 0x96462d76b0a776b1
```

Strings

Immutable collections of Unicode characters.

```
let name = "Morgan"
```

String methods and fields.

```
name.length // 6
name.utf8 // [77, 111, 114, 103, 97, 110]
name.concat(" Wilde") // "Morgan Wilde"
name.slice(from: 0, upTo: 1) // "M"
```

Optionals

Optionals are used when something can be set to `nil` or have no value assigned.

```
var inbox: String? = nil
inbox = "FastFloward says hi!"
inbox
```

```
inbox = nil
inbox
```

Arrays

Cadence arrays are mutable and can have fixed or variable size. Array elements must share the same type `T` or be of a subtype of `T`.

```
let days = ["Monday", "Tuesday"]
days
days[0]
days[2]
```

Array fields and functions.

```
days.length // 2
days.concat(["Wednesday"]) // ["Monday", "Tuesday", "Wednesday"]
days.contains("Friday") // false
days.append("Wednesday")
days // ["Monday", "Tuesday", "Wednesday"]
days.appendAll(["Thursday", "Friday"])
days.remove(at: 0) // "Monday"
days // ["Tuesday", "Wednesday", "Thursday", "Friday"]
days.insert(at: 0, "Monday")
days // ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]
days.removeFirst() // "Monday"
days.removeLast() // "Friday"
```

Dictionaries

Dictionaries are mutable, unordered collections of key-value pairs. Keys must be hashable and equatable, most of the built-in types conform to these requirements.

```
{} // empty dictionary
let capitals = {"Japan": "Tokyo", "France": "Paris"}
capitals["Japan"] // "Tokyo" of type String?
capitals["England"] = "London"
capitals
```

Dictionary fields and functions.

```
capitals.keys // ["Japan", "France", "England"]
capitals.values // ["Tokyo", "Paris", "London"]
capitals.containsKey("USA") // false
capitals.remove(key: "France") // "London"
```

Functions

Cadence functions are a lot like functions in other languages, especially Swift. They are value types, meaning you can assign them to variables, and pass them as arguments to other functions. Function arguments can have labels, which provide for clarity at the call-site for what each value represents.

Up to this point we were using the **REPL** functionality of `flow cadence`. To explore functions, we'll start executing our code by sending files to the interpreter.

```
flow cadence test.cdc
```

The only difference with `.cdc` files is that you have to declare an entry-point where execution should start. You do that by declaring a function called `main()`.

```
pub fun main() {
  log("Hi!")
}
```

The keyword `pub` that preceeds `fun` is an access modifier and it defines *public* access to the value. We'll discuss it later, for now, just use `pub` when declaring anything outside of the `main()` function.

For example.

```
pub fun sayHi(to name: String) {
  log("Hi, ".concat(name))
}
pub fun main() {
  sayHi(to: "FastFloward")
}
```

Composite Types

We now have the basics to start assembling more complex structures. In Cadence, you have two kinds of composite types.

1. Structures - value types (copied)
2. Resources - linear types (moved, not copied, can only exist once)

Declaration

You declare structures and resources almost the same, each can have fields, functions, and an initializer. Every field must be initialized in the `init()` initializer.

```
pub struct Rectangle {
  pub let width: Int
  pub let height: Int

  init(width: Int, height: Int) {
    self.width = width
    self.height = height
  }
}

pub resource Wallet {
  pub var dollars: UInt

  init(dollars: UInt) {
    self.dollars = dollars
  }
}
```

Instantiation

Structures are initialized like regular types.

```
let square = Rectangle(width: 10, height: 10)
```

Resources are different, instead of `=` we use `<-` to signify that we're moving the resource from one place to another. We also can't simply allow garbage collection to implicitly dispose of our resource, like we do with structures.

```
let myWallet <- create Wallet(dollars: 10)
destroy myWallet
```

We have to use `create` and `destroy` to explicitly mark what happens to our resources. A resource must be assigned to a variable/field that lives outside of a given scope or it must be destroyed.

Playground

With everything that we now know, we can start playing around with Cadence. Let's create some basic structures and functionality that will allow us to eventually build an app where we can draw Non-Fungible Tokens into existance!

Our goal is to be able to draw on a 5x5 grid by turning pixels on and off. For example, if we wanted to draw the letter **X**, we could do it this way (`.` represents *off* pixels, `*` represents *on* pixels).

```
*...*
.*.*.
..*..
.*.*.
*...*
```

We can create a Cadence structure to store our pixel canvases. It's a lot easier to store a single and serialize a string instead of a two-dimensional array, so that's what we'll use for our pixels.

```
pub struct Canvas {

  pub let width: UInt8
  pub let height: UInt8
  pub let pixels: String

  init(width: UInt8, height: UInt8, pixels: String) {
    self.width = width
    self.height = height
    // The following pixels
    // 123
    // 456
    // 789
    // should be serialized as
    // 123456789
    self.pixels = pixels
  }
}
```

However, we still want to declare our drawings in a legible way, so we'll use string arrays, but now, instead of `.` we'll use ` ` to represent *off* pixels.

```
let pixelsX = [
  "*   *",
  " * * ",
  "  *  ",
  " * * ",
  "*   *"
]
```

Unfortunately, we can't pass our array into the `Canvas` initializer, for that we need a new function.

```
pub fun serializeStringArray(_ lines: [String]): String {
  var buffer = ""
  for line in lines {
    buffer = buffer.concat(line)
  }

  return buffer
}
```

This will convert an array of strings into a string that our Canvas structure will accept.

```
pub fun main() {
  let pixelsX = [
    "*   *",
    " * * ",
    "  *  ",
    " * * ",
    "*   *"
  ]
  let letterX = Canvas(
    width: 5,
    height: 5,
    pixels: serializeStringArray(pixelsX)
  )
}
```

We want to provide some tangible ownership to our pixel artists, so let's provide functionality to print a `Canvas` as a `Picture` resource. To do this, we can simply wrap the `Canvas` structure in a resource.

```
pub resource Picture {
  pub let canvas: Canvas

  init(canvas: Canvas) {
    self.canvas = canvas
  }
}
```

Now, we can use it along with `Canvas`.

```
  pub fun main() {
    let pixelsX = [
      "*   *",
      " * * ",
      "  *  ",
      " * * ",
      "*   *"
    ]
    let canvasX = Canvas(
      width: 5,
      height: 5,
      pixels: serializeStringArray(pixelsX)
    )
    let letterX <- create Picture(canvas: canvasX)
    log(letterX.canvas)
    destroy letterX
  }
```

By now, you should start getting ideas for all the ways you could improve on this and add more functionality. Take some time to explore the following quests to see if you can find a solution.

Quests

For day one, we have two quests: `W1Q1` and `W1Q2`. If you need assistance while solving these, feel free to ask questions on Discord in the **burning-questions** channel.

- `W1Q1` – Frame it!

Write a function that displays a canvas in a frame.

```
  pub fun display(canvas: Canvas)
```

```
  "+-----+"
  "|*   *|"
  "| * * |"
  "|  *  |"
```

```
"| * * |"
"|*   *|"
"+-----+"
```

- `W1Q2` – Uniques

Create a resource that prints `Picture`'s but only once for each unique 5x5 `Canvas`.

```
pub resource Printer {
  pub fun print(canvas: Canvas): @Picture?
}
```