Cryptozombies

Chapter 1: Making the Zombie Factory

Contents

Zombie DNA 2
Zombie Factory 2
Implementation 2
Pragma 2
The Contract 2
The Zombie Struct 3
The Data 3
Functions 3
Events 4
Conclusion 4

Zombie DNA

Each Zombie has its own DNA, a 16-digit integer affecting its appearance. This DNA is composed of 8 pairs of digits, each affecting a certain aspect of the zombie's appearance. In the first lesson though, we only have 7 types of heads available, so to keep things consistent and not allow "out of bounds " DNAs we need to normalize each pair, by computing the modulo to the number of aspects available.

Two zombies with the same DNA will look identical. But two zombies with different DNAs are not guaranteed to look different (since if we have 7 different heads, we compute `head pair` % 7 so 07 and 14 would represent the same head). Each Zombie also has a name that 'defines' the DNA, so two zombies with the same name will look the same.

Now that we defined what a zombie DNA is, we can move to the real purpose of chapter 1. Making a Zombie Factory.

Zombie Factory

Our Zombie Factory must serve three purposes:

- Create a Zombie based on a name and a random DNA based on that name
- Save each Zombie created in the Factory's "not-so-in-memory-in-memory database"
- Emit an event called NewZombie when a new Zombie

A Zombie Factory also has 2 main attributes:

- The number of digits in a Zombie DNA
- The DNA modulus (in case the hashed value is larger than the number of digits allowed)

Implementation

Pragma

Guards the version of the Solidity Compiler version. This is to prevent issues with future compiler versions potentially introducing changes that would break your code. In this tutorial, we will use any compiler in the range >= 0.5.0 < 0.6.0. So, the code would be:

```
pragma solidity >=0.5.0 <0.6.0;
```

The Contract

> *A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.*
>
> (Solidity Documentation)

So, basically a contract is much like an instance of a class, it has data and it has functions. But the twist is that this contract is saved somewhere on the Ethereum blockchain, at a specific address. So, the data is persisted on the blockchain.

In this lesson we are building a ZombieFactory contract.

The Zombie Struct

To keep things nice and tidy, we must define a `struct` for our Zombies. Each Zombie will have a name and a DNA.

```
struct Zombie {
        string name;
        uint dna;
    }
```

The Data

Solidty has a bunch of types, ranging from numeric types (*uint8, uint16, uint32, ...*), `string` and many more. You can find the types documentation here: https://docs.soliditylang.org/en/v0.5.3/types.html

In this lesson, we will need a `dnaDigits and a dnaModulus`

```
uint dnaDigits = 16;
uint dnaModulus = 10 ** dnaDigits;
```

Also, we will need a way to store our Zombies, so we will declare an array of Zombie(s):

```
Zombie[] public zombies;
```

Functions

In Solidty, defining functions is pretty straight-forward. Parameters are passed by either value or reference. It's convention (but not required) to start function parameter variable names with an underscore (_) in order to differentiate them from global variables.

Passing by value is done using the `memory` keyword.

We need to define three functions:

Create a Zombie using a name (passed by value) and a DNA, which will be private

```
function _createZombie(string memory _name, uint _dna) private {
        uint id = zombies.push(Zombie(_name, _dna)) - 1;
        emit NewZombie(id, _name, _dna);
```

```
}
```

Generate a random DNA based on a name (passed by value), also private

```
function _generateRandomDna(string memory _str) private view returns
(uint) {
        uint rand = uint(keccak256(abi.encodePacked(_str)));
        return rand % dnaModulus;
}
```

Create a random Zombie based on a name (still passed by value), public

```
function createRandomZombie(string memory _name) public {
        uint randDna = _generateRandomDna(_name);
        _createZombie(_name, randDna);
}
```

Events

Events are a way for your contract to communicate that something happened on the blockchain to your app front-end, which can be 'listening' for certain events and take action when they happen.

We define a new event for when we create a new Zombie:

```
event NewZombie(uint zombieId, string name, uint dna);
```

And you can notice that we emit the event inside the _createZombie method after adding a new Zombie to our database.

We also need to listen for it inside our front-end app:

```
// Listen for the `NewZombie` event, and update the UI
var event = ZombieFactory.NewZombie(function(error, result) {
  if (error) return
  generateZombie(result.zombieId, result.name, result.dna)
})
```

Conclusion

This lesson we created a ZombieFactory contract, able to create zombies based on a name and store them on the Ethereum blockchain.