Chapter 1: Lesson Overview

In Lesson 1, you're going to build a "Zombie Factory" to build an army of zombies.

- Our factory will maintain a database of all zombies in our army
- Our factory will have a function for creating new zombies
- Each zombie will have a random and unique appearance

In later lessons, we'll add more functionality, like giving zombies the ability to attack humans or other zombies! But before we get there, we have to add the basic functionality of creating new zombies.

How Zombie DNA Works

The zombie's appearance will be based on its "Zombie DNA". Zombie DNA is simple — it's a 16-digit integer, like:

```
8356281049284737
```

Just like real DNA, different parts of this number will map to different traits. The first 2 digits map to the zombie's head type, the second 2 digits to the zombie's eyes, etc.

Note: For this tutorial, we've kept things simple, and our zombies can have only 7 different types of heads (even though 2 digits allow 100 possible options). Later on we could add more head types if we wanted to increase the number of zombie variations.

For example, the first 2 digits of our example DNA above are `83`. To map that to the zombie's head type, we do `83 % 7 + 1` = 7. So this Zombie would have the 7th zombie head type.

In the panel to the right, go ahead and move the `head gene` slider to the 7th head (the Santa hat) to see what trait the `83` would correspond to.

Put it to the test

1. Play with the sliders on the right side of the page. Experiment to see how the different numerical values correspond to different aspects of the zombie's appearance.

Ok, enough playing around. When you're ready to continue, hit "Next Chapter" below, and let's dive into learning Solidity!

Chapter 2: Contracts

Starting with the absolute basics:

Solidity's code is encapsulated in **contracts**. A `contract` is the fundamental building block of Ethereum applications — all variables and functions belong to a contract, and this will be the starting point of all your projects.

An empty contract named `HelloWorld` would look like this:

```
contract HelloWorld {

}
```

Version Pragma

All solidity source code should start with a "version pragma" — a declaration of the version of the Solidity compiler this code should use. This is to prevent issues with future compiler versions potentially introducing changes that would break your code.

It looks like this: `pragma solidity ^0.4.19;` (for the latest solidity version at the time of this writing, 0.4.19).

Putting it together, here is a bare-bones starting contract — the first thing you'll write every time you start a new project:

```
pragma solidity ^0.4.19;

contract HelloWorld {

}
```

Put it to the test

To start creating our Zombie army, let's create a base contract called `ZombieFactory`.

1. In the box to the right, make it so our contract uses solidity version `0.4.19`.
2. Create an empty contract called `ZombieFactory`.

When you're finished, click "check answer" below. If you get stuck, you can click "hint".

Chapter 3: State Variables & Integers

Great job! Now that we've got a shell for our contract, let's learn about how Solidity deals with variables.

State variables are permanently stored in contract storage. This means they're written to the Ethereum blockchain. Think of them like writing to a DB.

Example:

```
contract Example {
  // This will be stored permanently in the blockchain
  uint myUnsignedInteger = 100;
}
```

In this example contract, we created a `uint` called `myUnsignedInteger` and set it equal to 100.

Unsigned Integers: `uint`

The `uint` data type is an unsigned integer, meaning **its value must be non-negative**. There's also an `int` data type for signed integers.

Note: In Solidity, `uint` is actually an alias for `uint256`, a 256-bit unsigned integer. You can declare uints with less bits — `uint8`, `uint16`, `uint32`, etc.. But in general you want to simply use `uint` except in specific cases, which we'll talk about in later lessons.

Put it to the test

Our Zombie DNA is going to be determined by a 16-digit number.

Declare a `uint` named `dnaDigits`, and set it equal to `16`.

Chapter 4: Math Operations

Math in Solidity is pretty straightforward. The following operations are the same as in most programming languages:

- Addition: `x + y`
- Subtraction: `x - y`,
- Multiplication: `x * y`
- Division: `x / y`
- Modulus / remainder: `x % y` *(for example, `13 % 5` is `3`, because if you divide 5 into 13, 3 is the remainder)*

Solidity also supports an **exponential operator** (i.e. "x to the power of y", x^y):

```
uint x = 5 ** 2; // equal to 5^2 = 25
```

Put it to the test

To make sure our Zombie's DNA is only 16 characters, let's make another `uint`equal to 10^16. That way we can later use the modulus operator `%` to shorten an integer to 16 digits.

1. Create a `uint` named `dnaModulus`, and set it equal to **10 to the power of `dnaDigits`**.

Chapter 5: Structs

Sometimes you need a more complex data type. For this, Solidity provides **structs**:

```
struct Person {
  uint age;
  string name;
}
```

Structs allow you to create more complicated data types that have multiple properties.

Note that we just introduced a new type, `string`. Strings are used for arbitrary-length UTF-8 data. Ex. `string greeting = "Hello world!"`

Put it to the test

In our app, we're going to want to create some zombies! And zombies will have multiple properties, so this is a perfect use case for a struct.

1. Create a `struct` named `Zombie`.
2. Our `Zombie` struct will have 2 properties: `name` (a `string`), and `dna` (a `uint`).

Chapter 6: Arrays

When you want a collection of something, you can use an **array**. There are two types of arrays in Solidity: **fixed** arrays and **dynamic** arrays:

```
// Array with a fixed length of 2 elements:
uint[2] fixedArray;
// another fixed Array, can contain 5 strings:
string[5] stringArray;
// a dynamic Array - has no fixed size, can keep growing:
uint[] dynamicArray;
```

You can also create an array of **structs**. Using the previous chapter's `Person`struct:

```
Person[] people; // dynamic Array, we can keep adding to it
```

Remember that state variables are stored permanently in the blockchain? So creating a dynamic array of structs like this can be useful for storing structured data in your contract, kind of like a database.

Public Arrays

You can declare an array as `public`, and Solidity will automatically create a **getter** method for it. The syntax looks like:

```
Person[] public people;
```

Other contracts would then be able to read (but not write) to this array. So this is a useful pattern for storing public data in your contract.

Put it to the test

We're going to want to store an army of zombies in our app. And we're going to want to show off all our zombies to other apps, so we'll want it to be public.

1. Create a public array of `Zombie` **structs**, and name it `zombies`.

Chapter 7: Function Declarations

A function declaration in solidity looks like the following:

```
function eatHamburgers(string _name, uint _amount) {

}
```

This is a function named `eatHamburgers` that takes 2 parameters: a `string`and a `uint`. For now the body of the function is empty.

Note: It's convention (but not required) to start function parameter variable names with an underscore (_) in order to differentiate them from global variables. We'll use that convention throughout our tutorial.

You would call this function like so:

```
eatHamburgers("vitalik", 100);
```

Put it to the test

In our app, we're going to need to be able to create some zombies. Let's create a function for that.

1. Create a function named `createZombie`. It should take two parameters: **_name** (a `string`), and **_dna** (a `uint`).

Leave the body empty for now — we'll fill it in later.

Chapter 8: Working With Structs and Arrays

Creating New Structs

Remember our `Person` struct in the previous example?

```
struct Person {
  uint age;
  string name;
}

Person[] public people;
```

Now we're going to learn how to create new `Person`s and add them to our `people` array.

```
// create a New Person:
Person satoshi = Person(172, "Satoshi");

// Add that person to the Array:
people.push(satoshi);
```

We can also combine these together and do them in one line of code to keep things clean:

```
people.push(Person(16, "Vitalik"));
```

Note that `array.push()` adds something to the **end** of the array, so the elements are in the order we added them. See the following example:

```
uint[] numbers;
numbers.push(5);
numbers.push(10);
numbers.push(15);
// numbers is now equal to [5, 10, 15]
```

Put it to the test

Let's make our createZombie function do something!

1. Fill in the function body so it creates a new `Zombie`, and adds it to the `zombies` array. The `name` and `dna` for the new Zombie should come from the function arguments.
2. Let's do it in one line of code to keep things clean.

Chapter 9: Private / Public Functions

In Solidity, functions are `public` by default. This means anyone (or any other contract) can call your contract's function and execute its code.

Obviously this isn't always desireable, and can make your contract vulnerable to attacks. Thus it's good practice to mark your functions as `private` by default, and then only make `public` the functions you want to expose to the world.

Let's look at how to declare a private function:

```
uint[] numbers;

function _addToArray(uint _number) private {
  numbers.push(_number);
}
```

This means only other functions within our contract will be able to call this function and add to the `numbers` array.

As you can see, we use the keyword `private` after the function name. And as with function parameters, it's convention to start private function names with an underscore (`_`).

Put it to the test

Our contract's `createZombie` function is currently public by default — this means anyone could call it and create a new Zombie in our contract! Let's make it private.

1. Modify `createZombie` so it's a private function. Don't forget the naming convention!

Chapter 10: More on Functions

In this chapter, we're going to learn about Function **return values**, and function modifiers.

Return Values

To return a value from a function, the declaration looks like this:

```
string greeting = "What's up dog";

function sayHello() public returns (string) {
  return greeting;
}
```

In Solidity, the function declaration contains the type of the return value (in this case `string`).

Function modifiers

The above function doesn't actually change state in Solidity — e.g. it doesn't change any values or write anything.

So in this case we could declare it as a **view** function, meaning it's only viewing the data but not modifying it:

```
function sayHello() public view returns (string) {
```

Solidity also contains **pure** functions, which means you're not even accessing any data in the app. Consider the following:

```
function _multiply(uint a, uint b) private pure returns (uint) {
  return a * b;
}
```

This function doesn't even read from the state of the app — its return value depends only on its function parameters. So in this case we would declare the function as **pure**.

Note: It may be hard to remember when to mark functions as pure/view. Luckily the Solidity compiler is good about issuing warnings to let you know when you should use one of these modifiers.

Put it to the test

We're going to want a helper function that generates a random DNA number from a string.

1. Create a `private` function called `_generateRandomDna`. It will take one parameter named `_str` (a `string`), and return a `uint`.
2. This function will view some of our contract's variables but not modify them, so mark it as `view`.
3. The function body should be empty at this point — we'll fill it in later.

Chapter 11: Keccak256 and Typecasting

We want our `_generateRandomDna` function to return a (semi) random `uint`. How can we accomplish this?

Ethereum has the hash function `keccak256` built in, which is a version of SHA3. A hash function basically maps an input string into a random 256-bit hexidecimal number. A slight change in the string will cause a large change in the hash.

It's useful for many purposes in Ethereum, but for right now we're just going to use it for pseudo-random number generation.

Example:

```
//6e91ec6b618bb462a4a6ee5aa2cb0e9cf30f7a052bb467b0ba58b8748c00d2e5
keccak256("aaaab");
//b1f078126895a1424524de5321b339ab00408010b7cf0e6ed451514981e58aa9
keccak256("aaaac");
```

As you can see, the returned values are totally different despite only a 1 character change in the input.

*Note: **Secure** random-number generation in blockchain is a very difficult problem. Our method here is insecure, but since security isn't top priority for our Zombie DNA, it will be good enough for our purposes.*

Typecasting

Sometimes you need to convert between data types. Take the following example:

```
uint8 a = 5;
uint b = 6;
// throws an error because a * b returns a uint, not uint8:
uint8 c = a * b;
// we have to typecast b as a uint8 to make it work:
uint8 c = a * uint8(b);
```

In the above, `a * b` returns a `uint`, but we were trying to store it as a `uint8`, which could cause potential problems. By casting it as a `uint8`, it works and the compiler won't throw an error.

Put it to the test

Let's fill in the body of our `_generateRandomDna` function! Here's what it should do:

1. The first line of code should take the `keccak256` hash of `_str` to generate a pseudo-random hexidecimal, typecast it as a `uint`, and finally store the result in a `uint` called `rand`.
2. We want our DNA to only be 16 digits long (remember our `dnaModulus`?). So the second line of code should `return` the above value modulus (`%`) `dnaModulus`.

Chapter 12: Putting It Together

We're close to being done with our random Zombie generator! Let's create a public function that ties everything together.

We're going to create a public function that takes an input, the zombie's name, and uses the name to create a zombie with random DNA.

Put it to the test

1. Create a `public` function named `createRandomZombie`. It will take one parameter named `_name` (a `string`). *(Note: Declare this function `public` just as you declared previous functions `private`)*
2. The first line of the function should run the `_generateRandomDna` function on `_name`, and store it in a `uint` named `randDna`.
3. The second line should run the `_createZombie` function and pass it `_name` and `randDna`.
4. The solution should be 4 lines of code (including the closing `}` of the function).

Chapter 13: Events

Our contract is almost finished! Now let's add an **event**.

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.

Example:

```
// declare the event
event IntegersAdded(uint x, uint y, uint result);

function add(uint _x, uint _y) public {
  uint result = _x + _y;
  // fire an event to let the app know the function was called:
  IntegersAdded(_x, _y, result);
  return result;
}
```

Your app front-end could then listen for the event. A javascript implementation would look something like:

```
YourContract.IntegersAdded(function(error, result) {
  // do something with result
}
```

Put it to the test

We want an event to let our front-end know every time a new zombie was created, so the app can display it.

1. Declare an `event` called `NewZombie`. It should pass `zombieId` (a `uint`), `name` (a `string`), and `dna` (a `uint`).
2. Modify the `_createZombie` function to fire the `NewZombie` event after adding the new Zombie to our `zombies` array.

3. You're going to need the zombie's `id`. `array.push()` returns a `uint`of the new length of the array - and since the first item in an array has index 0, `array.push() - 1` will be the index of the zombie we just added. Store the result of `zombies.push() - 1` in a `uint` called `id`, so you can use this in the `NewZombie` event in the next line.

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    // declare our event here

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

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function _createZombie(string _name, uint _dna) private {
        zombies.push(Zombie(_name, _dna));
        // and fire it here
    }

    function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }

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

}
```

Chapter 14: Web3.js

Our Solidity contract is complete! Now we need to write a javascript frontend that interacts with the contract.

Ethereum has a Javascript library called **Web3.js**.

In a later lesson, we'll go over in depth how to deploy a contract and set up Web3.js. But for now let's just look at some sample code for how Web3.js would interact with our deployed contract.

Don't worry if this doesn't all make sense yet.

```
// Here's how we would access our contract:
var abi = /* abi generated by the compiler */
var ZombieFactoryContract = web3.eth.contract(abi)
var contractAddress = /* our contract address on Ethereum after deploying */
var ZombieFactory = ZombieFactoryContract.at(contractAddress)
// `ZombieFactory` has access to our contract's public functions and events

// some sort of event listener to take the text input:
$("#ourButton").click(function(e) {
  var name = $("#nameInput").val()
  // Call our contract's `createRandomZombie` function:
  ZombieFactory.createRandomZombie(name)
})

// 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)
})

// take the Zombie dna, and update our image
function generateZombie(id, name, dna) {
  let dnaStr = String(dna)
  // pad DNA with leading zeroes if it's less than 16 characters
  while (dnaStr.length < 16)
    dnaStr = "0" + dnaStr

  let zombieDetails = {
    // first 2 digits make up the head. We have 7 possible heads, so % 7
    // to get a number 0 - 6, then add 1 to make it 1 - 7. Then we have 7
    // image files named "head1.png" through "head7.png" we load based on
    // this number:
    headChoice: dnaStr.substring(0, 2) % 7 + 1,
    // 2nd 2 digits make up the eyes, 11 variations:
    eyeChoice: dnaStr.substring(2, 4) % 11 + 1,
    // 6 variations of shirts:
    shirtChoice: dnaStr.substring(4, 6) % 6 + 1,
    // last 6 digits control color. Updated using CSS filter: hue-rotate
    // which has 360 degrees:
    skinColorChoice: parseInt(dnaStr.substring(6, 8) / 100 * 360),
    eyeColorChoice: parseInt(dnaStr.substring(8, 10) / 100 * 360),
    clothesColorChoice: parseInt(dnaStr.substring(10, 12) / 100 * 360),
```

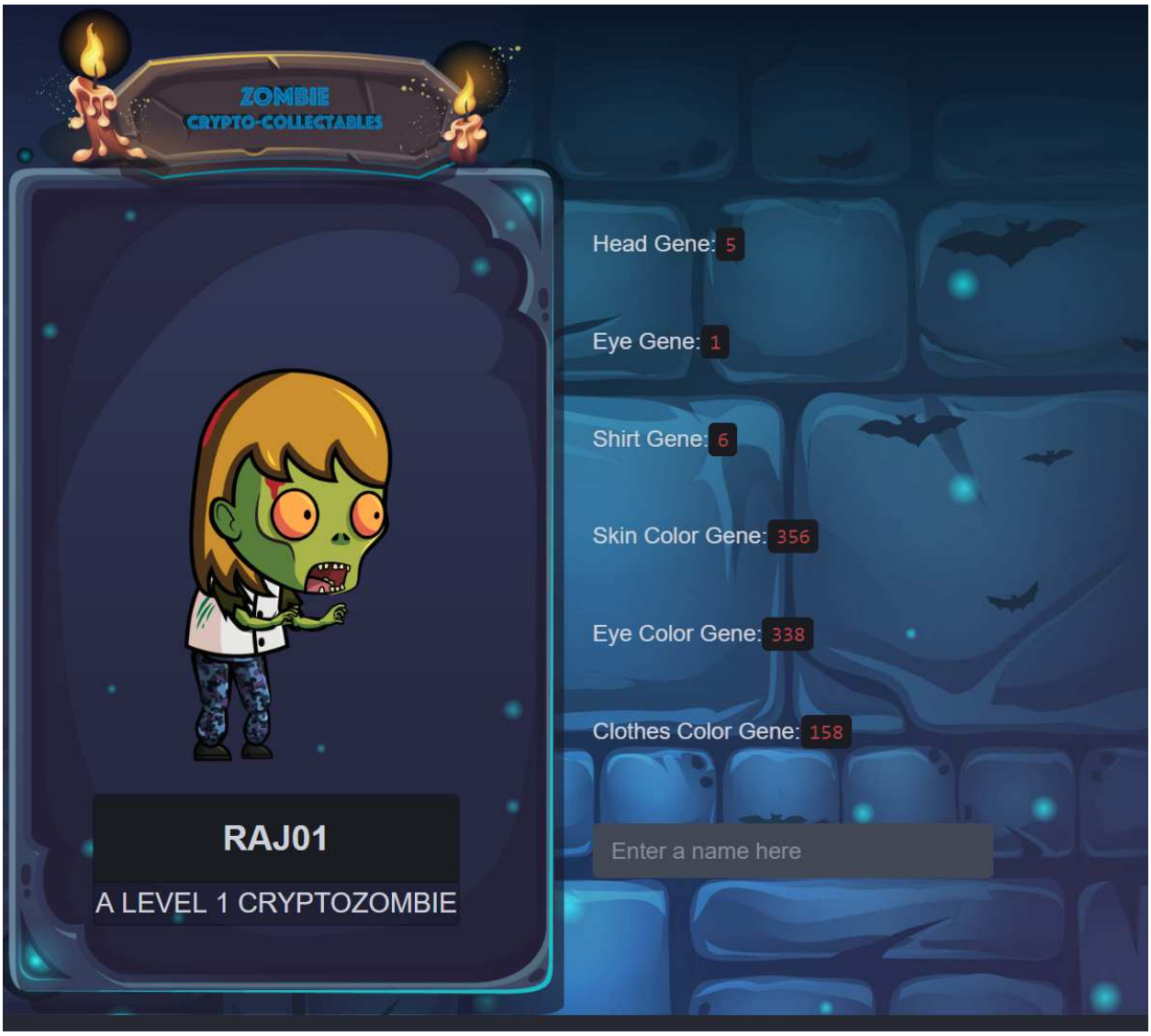
```
    zombieName: name,
    zombieDescription: "A Level 1 CryptoZombie",
  }
  return zombieDetails
}
```

What our javascript then does is take the values generated in `zombieDetails`above, and use some browser-based javascript magic (we're using Vue.js) to swap out the images and apply CSS filters. You'll get all the code for this in a later lesson.

Give it a try!

Go ahead — type in your name to the box on the right, and see what kind of zombie you get!

Once you have a zombie you're happy with, go ahead and click "Next Chapter" below to save your zombie and complete lesson 1!


ZOMBIE
CRYPTO-COLLECTABLES
RAJ01
A LEVEL 1 CRYPTOZOMBIE
Head Gene: 5
Eye Gene: 1
Shirt Gene: 6
Skin Color Gene: 356
Eye Color Gene: 338
Clothes Color Gene: 158
Enter a name here