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.

For the scope of this tutorial, we'll want to be able to compile our smart contracts with any compiler version in the range of 0.5.0 (inclusive) to 0.6.0 (exclusive). It looks like this: `pragma solidity >=0.5.0 <0.6.0;`.

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 >=0.5.0 <0.6.0;

contract HelloWorld {

}
```

State Variables & Integers

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.

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
```

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!"`

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 from, but not write to, this array. So this is a useful pattern for storing public data in your contract.

Function Declarations

A function declaration in solidity looks like the following:

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

}
```

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 that we're specifying the function visibility as `public`. We're also providing instructions about where the `_name` variable should be stored- in `memory`. This is required for all reference types such as arrays, structs, mappings, and strings.

What is a reference type you ask?

Well, there are two ways in which you can pass an argument to a Solidity function:

- By value, which means that the Solidity compiler creates a new copy of the parameter's value and passes it to your function. This allows your function to modify the value without worrying that the value of the initial parameter gets changed.
- By reference, which means that your function is called with a… reference to the original variable. Thus, if your function changes the value of the variable it receives, the value of the original variable gets changed.

  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);
```

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]
```

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 desirable, 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 (_).

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 memory) {
  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 memory) {
```

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.

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 returns (uint) {
  uint result = _x + _y;
  // fire an event to let the app know the function was called:
  emit 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
})
```

Web3.js

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
}
```