📝

Solidity Notes by Rushi Padhiyar

`pragma solidity >=0.5.0 <0.6.0;`

here >=0.5.0 <0.6.0 indicates the version of solidity.

Creating an empty “Hello World” smart contract will be like this:

`pragma solidity >=0.5.0 <0.6.0;`

`contract HelloWorld {`

`//start here`

`}`

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 t
o 100.
```

uint data type is an unsigned integer, meaning **its value must be non-negative**

int data type is an for signed integer

Example:

`pragma solidity >=0.5.0 <0.6.0;`

`contract ZombieFactory {`

`uint dnaDigits = 16;`

`}`

// here we declared an unsigned Integer named “dnaDigits” with the value 16.

Math in Solidity:

- Addition: `x + y`
- Subtraction: `x - y`
- Multiplication: `x * y`
- Division: `x / y`
- Modulus / remainder: `x % y`
- Power : `x ** y`

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

Example 2:

```
// Create a unsigned variable which shows the value 10^dnaDigits

pragma solidity >=0.5.0 <0.6.0;
contract ZombieFactory {
uint dnaDigits = 16;
uint dnaModulus = 10 ** dnaDigits;
```

To handle more of the complex data in Solidity:

Struct:

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

Example:

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

💡 Struct is like a class. It helps you to combine multiple variable’s type in one block.

Example 2:

```
// Create a Struct named Zombie and make the 'name' and 'dna' variable in it

pragma solidity >=0.5.0 <0.6.0;
contract ZombieFactory {

uint dnaDigits = 16;

uint dnaModulus = 10 ** dnaDigits;

struct Zombie {

      string name;

      uint dna;

}

}
```

Arrays:

There are two types of array: Fixed and Dynamic

Fixed Array:

We are limiting our Array size by some value

Dynamic Array:

We are not limiting our Array size and rather giving it an infinite amount of input.

Example:

```
// 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;
```

💡 Note: 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.

Example:

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

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.

Example 2:

```
// Create a public array called Zombie within the Struct Zombies
pragma solidity >=0.5.0 <0.6.0;

contract ZombieFactory {

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

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;


}
```

Function Declarations:

A function declaration in solidity looks like the following:

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

}
```

- This is a function named `Norway` 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.

Example 2:

```
//Create a public function named createZombie. It should take two parameters:
// _name (a string), and _dna (a uint).
// Don't forget to pass the first argument by value by using the memory keyword
contract ZombieFactory {

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

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function createZombie(string memory _name, uint _dna) public{

    }

}
```

Understanding the concept of Array and Structs:

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 jenish = Person(172, "Jenish");

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

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

```
people.push(Person(20, "Jenish"));
```

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]
```

Example 2:

```
// 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
```

```
pragma solidity >=0.5.0 <0.6.0;

contract ZombieFactory {

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

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function createZombie(string memory _name, uint _dna) public {
        zombies.push(Zombie(_name, _dna));
    }

}
```

Public/Private 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 (`_`).

Example 2:

```
//Modify createZombie so it's a private function in previous problem

function _createZombie(string memory _name, uint _dna) private {
```

`Return` : To return a value from a function

```
// Example of return function
string welcome = "Wassup dog";

function comeHome() public returns (string memory) {
  return welcome;
}
```

Now to know that our function is in only read only mode then we have to use `view` keyword after declaring the function in private.

```
function comeHome() public view returns (string memory) {
```

Another keyword `pure` is also used which means you're not even accessing any data in the app.

Confusion can be created in using both the function but the compiler of Solidity will give us an error if we used another one in place of other.

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

Example 2:

```
//Create a private function called _generateRandomDna.
//It will take one parameter named _str (a string), and return a uint.
//This function will view some of our contract's variables but not modify them,
//so mark it as view.

pragma solidity >=0.5.0 <0.6.0;

contract ZombieFactory {

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

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function _createZombie(string memory _name, uint _dna) private {
        zombies.push(Zombie(_name, _dna));
    }

    function _generateRandomDna(string memory _str) private view returns (uint){

    }

}
```

Keccak256

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 into a random 256-bit hexadecimal number. A slight change in the input 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.

Also important, `keccak256` expects a single parameter of type `bytes`. This means that we have to "pack" any parameters before calling `keccak256`:

Example:

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

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

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.

Example 2:

1. The first line of code should take the `keccak256` hash of `abi.encodePacked(_str)` to generate a pseudo-random hexadecimal, 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`.

```
pragma solidity  >=0.5.0 <0.6.0;

contract ZombieFactory {

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

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    function _createZombie(string memory _name, uint _dna) private {
        zombies.push(Zombie(_name, _dna));
    }

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

}
```