**Solidity Functions**

We've learned about how to store values inside of our smart contracts. Now it's time to learn how to read and modify those values! We accomplish this by adding functions to our contract code. Technically we already created functions in the previous lesson on data types! By setting our state variables to public, the compiler was creating a getter function under the hood:

**contract Contract {**

**uint public x = 3;**

**}**

We can query the value in x externally using a generated function called x(). What if we wanted to modify x?

**contract Contract {**

**uint public x = 3;**

**function changeX() external {**

**x = 5;**

**}**

**}**

Cool! That's just a taste of what's to come! Let's learn about functions.

1. **Arguments**

**Solidity Arguments**

The first function we'll talk about is the constructor:

**bool public isOpen;**

**constructor() {**

**isOpen = true;**

**}**

Here we are setting the value of a storage variable on the contract's deployment. The constructor for Solidity contracts is quite similar to the constructor in classes of many object-oriented languages. The constructor function is invoked only once during the contract's deployment and never again. It is generally used for setting up initial contract values. What if we wanted to let the deployer of the contract decide the value of isOpen? We can pass an argument to our constructor! Let's see that in action:

**bool public isOpen;**

**constructor(bool \_isOpen) {**

**isOpen = \_isOpen;**

**}**

Check it out! Now the contract deployer can decide the value of isOpen. Notice how the parameter name (\_isOpen) has an underscore in front of it? This prevents the variable from having the same name as the storage variable. When the names collide it is referred to as variable shadowing. It can happen in Solidity quite often since we can refer to storage variables without using this. Let's explore this further in details.

1. **Increment**

**Contract Functions**

Besides the constructor, contracts can define other functions which can be invoked directly by an externally owned account, or by another contract. Let's take a look at the Solidity function syntax:

**function myFunction() external {**

**// do something!**

**}**

The external visibility specifier allows this function to be accessed from other contracts or from an EOA. External visibility is quite similar to the public visibility for functions. External is better than public if you know that you are only calling the function externally (outside the EVM). Public visibility requires more gas because it can be called externally and internally, which complicates the assembly code.

1. **View Addition**

**Returning Values**

It's time to learn how to return values from Solidity functions! Let's see an example:

**contract Contract {**

**bool \_isRunning = true;**

**function isRunning() external view**

**returns(bool) {**

**// return the state variable**

**return \_isRunning;**

**}**

**}**

The isRunning() function indicates it is returning a boolean data type within the function signature: returns(bool). Once declared, we can use the return keyword to return a boolean value within this function. Adding the view keyword to the isRunning function signature guarantees it will not modify state variables. You can think of view functions as read-only; they can read the state of the contract but they cannot modify it.

1. **Console Log**

Using Console.log When developing smart contracts, Foundry makes it super easy to write Solidity unit tests, among many other features. We are using foundry for code execution in this very code lesson! For that reason, you can make use of the console library, for logging console data when you need to test something! Here's an example: **import "forge-std/console.sol";**

**contract C {**

**function x() external {**

**if(condition) {**

**console.log("condition was**

**met!");**

**}**

**}**

**}**

In this example, if the condition is met, we'll see a message in the test results! This can be very helpful when you're trying to debug your code. You can do this in any of the solidity lessons throughout this course, just remember to import "forge-std/console.sol" at the top of your file!  
**5. Pure Double**

Pure Functions Occasionally there is the necessity for Solidity functions that neither read from nor write to state. These functions can be labeled as pure. Let's say we wanted to add together two uint values:

**function double(uint x, uint y)**

**external pure returns(uint) {**

**return x + y;**

**}**

This function is just performing simple arithmetic without reading/writing state so we can label it pure. It's also worth noting there is an alternative syntax for returning values in Solidity: **function double(uint x, uint y)**

**external pure returns(uint sum) {**

**sum = x + y;**

**}**

Woah, that's new. In the returns keyword we specified the name of the returned parameter sum. Then we assigned the x + y to sum inside our function body. The value of sum is implicitly returned.

**6 .Double Overload**

Overloading Functions In Solidity it is perfectly valid to declare two functions with the same name if they have different parameters: **function add(uint x, uint y) external**

**pure returns(uint) {**

**return x + y;**

**}**

**function add(uint x, uint y, uint z)**

**external pure returns(uint) {**

**return x + y + z;**

**}**

Solidity will run the function whose signature matches the arguments provided. For example, add(2,4) will invoke the first function while add(2,3,4) will invoke the second function. Also, Solidity can return multiple values from functions:

**function addTwo(uint x, uint y)**

**external pure returns(uint, uint) {**

**return (x + 2, y + 2);**

**}**

Notice that the returns keyword specifies two return values. Also notice that we are wrapping the values in a parenthesis in order to return multiple values. This is referred to as a tuple. Tuples are not a formal type in Solidity. They are a list of values that can be used as a temporary structure to return values or do assignment destructuring. The data types of the return values in tuples can be different from each other. We can also use tuples in assignment destructuring. For example, if we call the function addTwo which we just defined above:

**(uint x, uint y) = addTwo(4, 8);**

**console.log(x); // 6**

**console.log(y); // 10**

**Message Calls**

**Accounts**

In Ethereum, accounts are often distinguished into two types: Externally Owned Accounts and Contract Accounts. The differences between these accounts is largely conceptual as the EVM essentially treats them the same! Every account on the EVM has a public address and a balance. Contract accounts will also store their bytecode as well as their internal storage data. When making a call from an EOA to a Contract Account it's important to know things like who is making the call, how much ether they are sending and the function they are intending to invoke with which arguments. The Solidity language handles the wiring up of the transaction data to the function we have defined on the contract. The language also gives us access to the transaction parameters through globals like msg.sender and msg.value. By providing these utilities for working with accounts we can easily define roles, permissions and track token balances in contracts. Let's learn all about working with accounts in Solidity!

1. **Storing Owner**

Solidity Addresses Let's talk about the address data type in Solidity! An address on the EVM is a 160 bits long, or a 40 character, hexadecimal string:

address a = 0xc783df8a850f42e7f7e57013759c285caa701eb6;

This is valid Solidity! We can store a fixed address in our contracts if we need to. We can also find the sender of the current message: **import "forge-std/console.sol";**

**contract Example {**

**constructor() {**

**console.log( msg.sender ); // 0xc783df8a850f42e7f7e57013759c285caa701eb6**

**}**

**}**

Here we are logging the address of the account calling this contract. What is msg? We'll take a closer look at EVM messages in Details.  
**2. Receive Ether**

Receive Function

In the latest versions of Solidity, contracts cannot receive ether by default. In order to receive ether, a contract must specify a payable function. This is another keyword which affects the function's mutability similar to view and pure. Let's see a payable function in action:

**import "forge-std/console.sol";**

**contract Contract {**

**function pay() public payable {**

**console.log( msg.value ); // 100000**

**}**

**}**

Here the msg.value represents the amount of ether, in Wei, sent to the pay function. By simply adding the payable keyword to this function, it gains the ability to accept ether. Once received, the ether is automatically added to the contract's balance—no additional steps required! What if someone tried to send a payment to a nonpayable function? The transaction will fail, sending the ether back to the sender. In the case above we used the method pay as a payable function. This means we have to call this function in order to send the ether to the contract. What if we wanted to send it directly without specifying a method? Turns out, we can do that too: **import "forge-std/console.sol";**

**contract Contract {**

**receive() external payable {**

**console.log(msg.value);**

**// 100000**

**}**

**}**

You'll notice that receive does not use the function keyword. This is because it is a special function (like constructor). It is the function that runs when a contract is sent ether without any calldata, or when the calldata does not match a function signature on the contract. The receive function must be external, payable, it cannot receive arguments and it cannot return anything. Another option to receive ether without specifying a function signature on a contract is to use a payable fallback function.  
**3. Tip Owner**

Transferring Funds

We can make any regular function payable. This allows us to differentiate the purpose of the ether coming into the smart contract. Perhaps a contract has two stored addresses and we want to be able to pay each:

**contract Contract {**

**address public a;**

**address public b;**

**constructor(address \_a, address \_b)**

**{**

**a = \_a;**

**b = \_b;**

**}**

**function payA() public payable {**

**(bool s, ) = a.call{ value:**

**msg.value }("");**

**require(s);**

**}**

**function payB() public payable {**

**(bool s, ) = b.call{ value:**

**msg.value }("");**

**require(s);**

**}**

**}**

We have two pay methods payA and payB which will transfer ether to the respective address. It takes a uint amount of Wei and transfers it from the contract account to the address.  
**4. Charity**

**Contract Account**

Within contracts, the this keyword can explicitly converted to an address:

**import "forge-std/console.sol";**

**contract Contract {**

**constructor() {**

**console.log( address(this) );**

**// 0x7c2c195cd6d34b8f845992d380aadb2730bb9c6f console.log(**

**address(this).balance ); // 0**

**}**

**}**

Using this we can easily find the address and balance of the contract! Let's take a look at another use for the Solidity keyword this in Details.

1. **Self Destruct**

Contracts can destroy themselves by using the SELFDESTRUCT opcode on the EVM! This opcode actually refunds ether in order to incentivize cleaning up the blockchain of unused contracts. Let's see it in action:

**contract Contract {**

**uint \_countdown = 10;**

**constructor() payable { }**

**function tick() public {**

**\_countdown--;**

**if(\_countdown == 0) {**

**// NOTE: we must cast to address payable here**

**// some solidity methods protect**

**// against accidentally sending ether selfdestruct(payable(msg.sender));**

**}**

**}**

**}**

After 10 calls to the tick function the Contract will selfdestruct! So you might be wondering, why did we provide msg.sender as the argument to selfdestruct? The address provided to the selfdestruct function gets all of the ether remaining in the contract! Ether sent to the payable constructor will be refunded to the final caller of the tick function. Before self-destructing your smart contract you may want to consider the repercussions. Let's discuss this in details.