Chapter 1: Payable

Up until now, we've covered quite a few **function modifiers**. It can be difficult to try to remember everything, so let's run through a quick review:

1. We have visibility modifiers that control when and where the function can be called from: private means it's only callable from other functions inside the contract; internal is like private but can also be called by contracts that inherit from this one; external can only be called outside the contract; and finally public can be called anywhere, both internally and externally.
2. We also have state modifiers, which tell us how the function interacts with the BlockChain: view tells us that by running the function, no data will be saved/changed. pure tells us that not only does the function not save any data to the blockchain, but it also doesn't read any data from the blockchain. Both of these don't cost any gas to call if they're called externally from outside the contract (but they do cost gas if called internally by another function).
3. Then we have custom modifiers, which we learned about in Lesson 3: onlyOwner and aboveLevel, for example. For these we can define custom logic to determine how they affect a function.

These modifiers can all be stacked together on a function definition as follows:

function test() external view onlyOwner anotherModifier { */\* ... \*/* }

In this chapter, we're going to introduce one more function modifier: payable.

The payable Modifier

payable functions are part of what makes Solidity and Ethereum so cool — they are a special type of function that can receive Ether.

Let that sink in for a minute. When you call an API function on a normal web server, you can't send US dollars along with your function call — nor can you send Bitcoin.

But in Ethereum, because both the money (*Ether*), the data (*transaction payload*), and the contract code itself all live on Ethereum, it's possible for you to call a function **and** pay money to the contract at the same time.

This allows for some really interesting logic, like requiring a certain payment to the contract in order to execute a function.

Let's look at an example

contract OnlineStore {

function buySomething() external payable {

*// Check to make sure 0.001 ether was sent to the function call:*

require(msg.value == 0.001 ether);

*// If so, some logic to transfer the digital item to the caller of the function:*

transferThing(msg.sender);

}

}

Here, msg.value is a way to see how much Ether was sent to the contract, and ether is a built-in unit.

What happens here is that someone would call the function from web3.js (from the DApp's JavaScript front-end) as follows:

*// Assuming `OnlineStore` points to your contract on Ethereum:*

OnlineStore.buySomething({from: web3.eth.defaultAccount, value: web3.utils.toWei(0.001)})

Notice the value field, where the javascript function call specifies how much ether to send (0.001). If you think of the transaction like an envelope, and the parameters you send to the function call are the contents of the letter you put inside, then adding a value is like putting cash inside the envelope — the letter and the money get delivered together to the recipient.

*Note: If a function is not marked payable and you try to send Ether to it as above, the function will reject your transaction.*

# Chapter 2: Withdraws

In the previous chapter, we learned how to send Ether to a contract. So what happens after you send it?

After you send Ether to a contract, it gets stored in the contract's Ethereum account, and it will be trapped there — unless you add a function to withdraw the Ether from the contract.

You can write a function to withdraw Ether from the contract as follows:

contract GetPaid is Ownable {

function withdraw() external onlyOwner {

owner.transfer(this.balance);

}

}

Note that we're using owner and onlyOwner from the Ownable contract, assuming that was imported.

You can transfer Ether to an address using the transfer function, and this.balance will return the total balance stored on the contract. So if 100 users had paid 1 Ether to our contract, this.balance would equal 100 Ether.

You can use transfer to send funds to any Ethereum address. For example, you could have a function that transfers Ether back to the msg.sender if they overpaid for an item:

uint itemFee = 0.001 ether;

msg.sender.transfer(msg.value - itemFee);

Or in a contract with a buyer and a seller, you could save the seller's address in storage, then when someone purchases his item, transfer him the fee paid by the buyer: seller.transfer(msg.value).

These are some examples of what makes Ethereum programming really cool — you can have decentralized marketplaces like this that aren't controlled by anyone.