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.

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 {
    address payable _owner = address(uint160(owner()));
    _owner.transfer(address(this).balance);
  }
}
```

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

It is important to note that you cannot transfer Ether to an address unless that address is of type `address payable`. But the `_owner` variable is of type `uint160`, meaning that we must explicitly cast it to `address payable`.

Once you cast the address from `uint160` to `address payable`, you can transfer Ether to that address using the `transfer` function, and `address(this).balance` will return the total balance stored on the contract. So if 100 users had paid 1 Ether to our contract, `address(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.

Random Numbers

So how do we generate random numbers in Solidity?

The real answer here is, you can't. Well, at least you can't do it safely.

Let's look at why.

Random number generation via `keccak256`

The best source of randomness we have in Solidity is the `keccak256` hash function.

We could do something like the following to generate a random number:

```
// Generate a random number between 1 and 100:
uint randNonce = 0;
uint random = uint(keccak256(abi.encodePacked(now, msg.sender, randNonce))) % 100;
randNonce++;
uint random2 = uint(keccak256(abi.encodePacked(now, msg.sender, randNonce))) % 100;
```

What this would do is take the timestamp of `now`, the `msg.sender`, and an incrementing `nonce` (a number that is only ever used once, so we don't run the same hash function with the same input parameters twice).

It would then "pack" the inputs and use `keccak` to convert them to a random hash. Next, it would convert that hash to a `uint`, and then use `% 100` to take only the last 2 digits. This will give us a totally random number between 0 and 99.

This method is vulnerable to attack by a dishonest node

In Ethereum, when you call a function on a contract, you broadcast it to a node or nodes on the network as a ***transaction***. The nodes on the network then collect a bunch of transactions, try to be the first to solve a computationally-intensive mathematical problem as a "Proof of Work", and then publish that group of transactions along with their Proof of Work (PoW) as a ***block*** to the rest of the network.

Once a node has solved the PoW, the other nodes stop trying to solve the PoW, verify that the other node's list of transactions are valid, and then accept the block and move on to trying to solve the next block.

This makes our random number function exploitable.

Let's say we had a coin flip contract — heads you double your money, tails you lose everything. Let's say it used the above random function to determine heads or tails. (`random >= 50` is heads, `random < 50` is tails).

If I were running a node, I could publish a transaction **only to my own node** and not share it. I could then run the coin flip function to see if I won — and if I lost, choose not to include that transaction in the next block I'm solving. I could keep doing this indefinitely until I finally won the coin flip and solved the next block, and profit.

So how do we generate random numbers safely in Ethereum?

Because the entire contents of the blockchain are visible to all participants, this is a hard problem, and its solution is beyond the scope of this tutorial. You can read this StackOverflow thread for some ideas. One idea would be to use an ***oracle*** to access a random number function from outside of the Ethereum blockchain.

Of course, since tens of thousands of Ethereum nodes on the network are competing to solve the next block, my odds of solving the next block are extremely low. It would take me a lot of time or computing resources to exploit this profitably — but if the reward were high enough (like if I could bet $100,000,000 on the coin flip function), it would be worth it for me to attack.

So while this random number generation is NOT secure on Ethereum, in practice unless our random function has a lot of money on the line, the users of your game likely won't have enough resources to attack it.

Because we're just building a simple game for demo purposes in this tutorial and there's no real money on the line, we're going to accept the tradeoffs of using a random number generator that is simple to implement, knowing that it isn't totally secure.

In a future lesson, we may cover using ***oracles*** (a secure way to pull data in

from outside of Ethereum) to generate secure random numbers from outside the blockchain.