Mappings and Addresses

We'll need 2 new data types: `mapping` and `address`. ## Addresses

The Ethereum blockchain is made up of ***accounts***, which you can think of like bank accounts. An account has a balance of ***Ether*** (the currency used on the Ethereum blockchain), and you can send and receive Ether payments to other accounts, just like your bank account can wire transfer money to other bank accounts.

Each account has an `address`, which you can think of like a bank account number. It's a unique identifier that points to that account, and it looks like this:

`0x0cE446255506E92DF41614C46F1d6df9Cc969183`

We'll get into the nitty gritty of addresses in a later lesson, but for now you only need to understand that **an address is owned by a specific user** (or a smart contract).

So we can use it as a unique ID for ownership of our zombies. When a user creates new zombies by interacting with our app, we'll set ownership of those zombies to the Ethereum address that called the function.

Mappings

In Lesson 1 we looked at ***structs*** and ***arrays***. ***Mappings*** are another way of storing organized data in Solidity.

Defining a `mapping` looks like this:

```
// For a financial app, storing a uint that holds the user's account balance:
mapping (address => uint) public accountBalance;
// Or could be used to store / lookup usernames based on userId
mapping (uint => string) userIdToName;
```

A mapping is essentially a key-value store for storing and looking up data. In the first example, the key is an `address` and the value is a `uint`, and in the second example the key is a `uint` and the value a `string`.

Msg.sender

msg.sender

In Solidity, there are certain global variables that are available to all functions. One of these is `msg.sender`, which refers to the `address` of the person (or smart contract) who called the current function.

> Note: In Solidity, function execution always needs to start with an external caller. A contract will just sit on the blockchain doing

> nothing until someone calls one of its functions. So there will always be a `msg.sender`.

Here's an example of using `msg.sender` and updating a `mapping`:

```
mapping (address => uint) favoriteNumber;

function setMyNumber(uint _myNumber) public {
  // Update our `favoriteNumber` mapping to store `_myNumber` under `msg.sender`
  favoriteNumber[msg.sender] = _myNumber;
  // ^ The syntax for storing data in a mapping is just like with arrays
}

function whatIsMyNumber() public view returns (uint) {
  // Retrieve the value stored in the sender's address
  // Will be `0` if the sender hasn't called `setMyNumber` yet
  return favoriteNumber[msg.sender];
}
```

In this trivial example, anyone could call `setMyNumber` and store a `uint` in our contract, which would be tied to their address. Then when they called `whatIsMyNumber`, they would be returned the `uint` that they stored.

Using `msg.sender` gives you the security of the Ethereum blockchain — the only way someone can modify someone else's data would be to steal the private key associated with their Ethereum address.

Require

Let's make it so each player can only call this function once. That way new players will call it when they first start the game in order to create the initial zombie in their army.

How can we make it so this function can only be called once per player?

For that we use `require`. `require` makes it so that the function will throw an error and stop executing if some condition is not true:

```
function sayHiToVitalik(string memory _name) public returns (string memory) {
  // Compares if _name equals "Vitalik". Throws an error and exits if not true.
  // (Side note: Solidity doesn't have native string comparison, so we
  // compare their keccak256 hashes to see if the strings are equal)
  require(keccak256(abi.encodePacked(_name)) == keccak256(abi.encodePacked("Vitalik")));
  // If it's true, proceed with the function:
  return "Hi!";
}
```

If you call this function with `sayHiToVitalik("Vitalik")`, it will return "Hi!". If you call it with any other input, it will throw an error and not execute.

Thus `require` is quite useful for verifying certain conditions that must be true before running a function.

Inheritance

Rather than making one extremely long contract, sometimes it makes sense to split your code logic across multiple contracts to organize the code.

One feature of Solidity that makes this more manageable is contract ***inheritance***:

```
contract Doge {
  function catchphrase() public returns (string memory) {
    return "So Wow CryptoDoge";
  }
}

contract BabyDoge is Doge {
  function anotherCatchphrase() public returns (string memory) {
    return "Such Moon BabyDoge";
  }
}
```

`BabyDoge` ***inherits*** from `Doge`. That means if you compile and deploy `BabyDoge`, it will have access to both `catchphrase()` and `anotherCatchphrase()` (and any other public functions we may define on `Doge`).

This can be used for logical inheritance (such as with a subclass, a `Cat` is an `Animal`). But it can also be used simply for organizing your code by grouping similar logic together into different contracts.

Import

Whoa! You'll notice we just cleaned up the code to the right, and you now have tabs at the top of your editor. Go ahead, click between the tabs to try it out.

Our code was getting pretty long, so we split it up into multiple files to make it more manageable. This is normally how you will handle long codebases in your Solidity projects.

When you have multiple files and you want to import one file into another, Solidity uses the `import` keyword:

```
import "./someothercontract.sol";

contract newContract is SomeOtherContract {

}
```

So if we had a file named `someothercontract.sol` in the same directory as this contract (that's what the `./` means), it would get imported by the compiler.

Storage vs Memory (Data location)

In Solidity, there are two locations you can store variables — in `storage` and in `memory`.

Storage refers to variables stored permanently on the blockchain. ***Memory*** variables are temporary, and are erased between external function calls to your contract. Think of it like your computer's hard disk vs RAM.

Most of the time you don't need to use these keywords because Solidity handles them by default. State variables (variables declared outside of functions) are by default `storage` and written permanently to the blockchain, while variables declared inside functions are `memory` and will disappear when the function call ends.

However, there are times when you do need to use these keywords, namely when dealing with ***structs*** and ***arrays*** within functions:

```
contract SandwichFactory {
  struct Sandwich {
    string name;
    string status;
  }

  Sandwich[] sandwiches;

  function eatSandwich(uint _index) public {
    // Sandwich mySandwich = sandwiches[_index];

    // ^ Seems pretty straightforward, but solidity will give you a warning
    // telling you that you should explicitly declare `storage` or `memory` here.

    // So instead, you should declare with the `storage` keyword, like:
    Sandwich storage mySandwich = sandwiches[_index];
    // ...in which case `mySandwich` is a pointer to `sandwiches[_index]`
    // in storage, and...
    mySandwich.status = "Eaten!";
    // ...this will permanently change `sandwiches[_index]` on the blockchain.

    // If you just want a copy, you can use `memory`:
    Sandwich memory anotherSandwich = sandwiches[_index + 1];
    // ...in which case `anotherSandwich` will simply be a copy of the
    // data in memory, and...
    anotherSandwich.status = "Eaten!";
    // ...will just modify the temporary variable and have no effect
    // on `sandwiches[_index + 1]`. But you can do this:
    sandwiches[_index + 1] = anotherSandwich;
    // ...if you want to copy the changes back into blockchain storage.
  }
}
```

Don't worry if you don't fully understand when to use which one yet — throughout this tutorial we'll tell you when to use `storage` and when to use `memory`, and the Solidity compiler will also give you warnings to let you know when you should be using one of these keywords.

For now, it's enough to understand that there are cases where you'll need to explicitly declare `storage` or `memory`!

More on Function Visibility

Internal and External

In addition to `public` and `private`, Solidity has two more types of visibility for functions: `internal` and `external`.

`internal` is the same as `private`, except that it's also accessible to contracts that inherit from this contract.

`external` is similar to `public`, except that these functions can ONLY be called outside the contract — they can't be called by other functions inside that contract. We'll talk about why you might want to use `external` vs `public` later.

For declaring `internal` or `external` functions, the syntax is the same as `private` and `public`:

```
contract Sandwich {
  uint private sandwichesEaten = 0;

  function eat() internal {
    sandwichesEaten++;
  }
}

contract BLT is Sandwich {
  uint private baconSandwichesEaten = 0;

  function eatWithBacon() public returns (string memory) {
    baconSandwichesEaten++;
    // We can call this here because it's internal
    eat();
  }
}
```

Interacting with other contracts

For our contract to talk to another contract on the blockchain that we don't own, first we need to define an ***interface***.

Let's look at a simple example. Say there was a contract on the blockchain that looked like this:

```
contract LuckyNumber {
  mapping(address => uint) numbers;

  function setNum(uint _num) public {
    numbers[msg.sender] = _num;
  }

  function getNum(address _myAddress) public view returns (uint) {
    return numbers[_myAddress];
  }
}
```

This would be a simple contract where anyone could store their lucky number, and it will be associated with their Ethereum address. Then anyone else could look up that person's lucky number using their address.

Now let's say we had an external contract that wanted to read the data in this contract using the `getNum` function.

First we'd have to define an ***interface*** of the `LuckyNumber` contract:

```
contract NumberInterface {
  function getNum(address _myAddress) public view returns (uint);
}
```

Notice that this looks like defining a contract, with a few differences. For one, we're only declaring the functions we want to interact with — in this case `getNum` — and we don't mention any of the other functions or state variables.

Secondly, we're not defining the function bodies. Instead of curly braces (`{` and `}`), we're simply ending the function declaration with a semi-colon (`;`).

So it kind of looks like a contract skeleton. This is how the compiler knows it's an interface.

By including this interface in our dapp's code our contract knows what the other contract's functions look like, how to call them, and what sort of response to expect

Using an Interface

Continuing our previous example with `NumberInterface`, once we've defined the interface as:

```
contract NumberInterface {
  function getNum(address _myAddress) public view returns (uint);
}
```

We can use it in a contract as follows:

```
contract MyContract {
  address NumberInterfaceAddress = 0xab38...
  // ^ The address of the FavoriteNumber contract on Ethereum
  NumberInterface numberContract = NumberInterface(NumberInterfaceAddress);
  // Now `numberContract` is pointing to the other contract

  function someFunction() public {
    // Now we can call `getNum` from that contract:
    uint num = numberContract.getNum(msg.sender);
    // ...and do something with `num` here
  }
}
```

In this way, your contract can interact with any other contract on the Ethereum blockchain, as long they expose those functions as `public` or `external`.

Handling Multiple Return Values

```
function multipleReturns() internal returns(uint a, uint b, uint c) {
  return (1, 2, 3);
}

function processMultipleReturns() external {
  uint a;
  uint b;
  uint c;
  // This is how you do multiple assignment:
  (a, b, c) = multipleReturns();
}

// Or if we only cared about one of the values:
function getLastReturnValue() external {
  uint c;
  // We can just leave the other fields blank:
  (,,c) = multipleReturns();
}
```

If statements

If statements in Solidity look just like javascript:

```
function eatBLT(string memory sandwich) public {
  // Remember with strings, we have to compare their keccak256 hashes
  // to check equality
  if (keccak256(abi.encodePacked(sandwich)) == keccak256(abi.encodePacked("BLT"))) {
    eat();
  }
}
```