CMSC 398F
Week #11
Solidity Continuation

•••

Announcements

- Quiz 4 will be released soon.
  - Will be on Solidity
    - Specifically information on cryptozombies.io
- Join the class Piazza!
  - piazza.com/umd/fall2022/cmsc398f

From Last Time

- Four visibility modifiers
  - Private
  - Internal
  - Public
  - External
- Complex Data Types
  - Arrays, Structs, Mappings
- ERC20 - fungible token
  - Requires specific and necessary functions to transfer and receive money
    - transfer(address a, uint256 value), balanceOf(address a), approve(address spender, uint256 value), etc.

msg.sender

- Msg.sender is a global variable that is available to all functions. It refers to the address of the person who called the current function

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

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

Require Keyword

- When the require statement is reached, function will check the condition within the require statement. If it is false, execution will be halted and an error will be thrown.
- There is also revert() and assert(), which will be discussed later.

```
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("Vita
  // If it's true, proceed with the function:
  return "Hi!";
}
```

Inheritance

- Contracts can inherit from other contracts (contract inheritance)
- Only public functions can be inherited.

```
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";
  }
}
```

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

contract newContract is SomeOtherContract {

}
```

Storage vs. Memory

- Storage variables are stored on the blockchain (data is saved between function calls)
- Memory variables are temporary, and are erased between external function calls to your contract
- State variables are storage by default
- Variables declared inside functions are memory variables, and disappear when the function call ends.
- These keywords are important for handling 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.
  }
}
```

Storage

- You can pass structs as arguments to private or internal functions with the storage keyword.
- Sort of like a pointer to something on the Heap in java
- Need to use storage keyword for passing structs

```
function _doStuff(Zombie storage _zombie) internal {
  // do stuff with _zombie
}
```

Interfacing with other contracts

- Suppose we want our contract to talk to another contract on the blockchain that we don't own. To do this we need to define an interface.
- In an interface, we declare functions we want to interact with.
    - No state variables or function bodies
- Our contract now knows what the other contracts functions look like, how to call them, and what response to expect

```
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];
  }
}
```

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

Using an Interface

```
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
  }
}
```

Immutability

- Ethereum dApps are far different than other applications we see on a daily basis
- After you deploy a contract to Ethereum, it becomes immutable
  - It can never be modified or updated again
- If there's a flaw in your contract code, there's no way for you to patch it later
  - You would have to tell your users to start using a different smart contract address that has the fix
- In order to solve this, it becomes common practice to rely on external dependencies
  - I.E. Use functions with external modifier in order to update key information about your dApp.
  - This can allow you to save your DApp in certain situations

Contract Ownership

- Solidity offers features of Ownership to allow certain addresses to call functions
- If we want to call a function that only the owner can access, we would need to have specific modifiers in place to ensure this
- OpenZeppelin is an open-source platform for building secure dApps. The framework provides the required tools to create and automate Web3 applications
    - A lot of people will begin their DApp by importing OpenZeppelin's ownable.sol file
        - Contains everything that is needed to transfer, renounce, and check for ownership

OpenZeppelin - Smart Contract Library

- A library of modular, reusable, secure smart contracts for the Ethereum network, written in Solidity.
- The contracts are completely modular and reusable and contain the most used implementations of ERC standards.
- This means that the companies do not need to develop security features and tools for smart contracts.
- Popular libraries:
  - Payment: Provides payment-related utilities.
  - Token: Provides the most popular ERC token utilities.
  - Utils: Provides miscellaneous smart contract utility functions.
  - SafeMath: Math operations on solidity

OpenZeppelin’s onlyOwner Modifier

```
constructor() internal {
  _owner = msg.sender;
  emit OwnershipTransferred(address(0), _owner);
}
```

```
function isOwner() public view returns(bool) {
  return msg.sender == _owner;
}
```

```
modifier onlyOwner() {
  require(isOwner());
  _;
}
```

```
KittyInterface kittyContract;

function setKittyContractAddress(address _address) external onlyOwner {
  kittyContract = KittyInterface(_address);
}
```

View and Pure modifiers

- **view** keyword simply means "this function is read-only"
- They do not cost much to execute since they are not attempting to add anything to the blockchain

```
function getKittyName() view {
  return addressToKitty[msg.sender];
}
```

- **pure** keyword means that it returns a value using only the parameters of the function without any side effects

```
function doCrazyMath(int num1, int num2, int num3) pure {
  return (num1 + num2 * num3 % num1) * (num2 * num3) + num3;
}
```

Time Units

- Solidity provides some native units for dealing with time.
- The global variable **now** will return the current unix (Epoch) timestamp of the latest block
  - the number of seconds that have passed since January 1st, 1970
  - Note: There isn't any difference between **block.timestamp** and **now**. But, in Solidity v0.7.0, the **now** keyword has been deprecated.
- Solidity also contains the time units in seconds, minutes, hours, days, weeks and years
  - These will convert to a uint of the number of seconds in that length of time
    - 1 hour -> 60
    - 1 day -> 86400 (24 hours x 60 minutes x 60 seconds)

Payable Modifier

- **payable** is a modifier that can be added to a function or variable
  - ensures that the function/variable can send and receive Ether
- It can process transactions with non-zero Ether values and rejects any transactions with a zero Ether value

```
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 func
    transferThing(msg.sender);
  }
}
```

Withdraw Function

- You can write a function to withdraw Ether from the contract in many ways; however, the one that follows is :

```
contract GetPaid is Ownable {
  function withdraw() external onlyOwner {
    address payable _owner = address(uint160(owner()));
    _owner.transfer(address(this).balance);
  }
}
```

Gas Fees when coding

- **Gas** is a unit of execution fee used to compensate stakeholders for the computational resources required to power smart contracts.
- How much **ether** do you need to pay for a transaction?
  - You pay **gas spent * gas price** amount of ether, where
    - **gas** is a unit of computation
    - **gas spent** is the total amount of gas used in a transaction
    - **gas price** is how much ether you are willing to pay per gas
- Transactions with higher gas price have higher priority to be included in a block and any unspent gas will be refunded
- Gas-saving patterns:
  - Short-circuiting
  - Library use
  - Explicit function visibility
  - Proper data types
  - Struct-packing

ERC's and Minting

- ERC-20
  - A fungible token standard that follows EIP-20 standards, such as fiat currencies
- ERC-721
  - Is an NFT token standard that follows EIP-721
- ERC-1155
  - Represents both fungible and non-fungible tokens
  - It is known as a multi-token contract and follows EIP-1155 standards
- ERC-721 and ERC-1155 are both token standards that allows you to mint NFTs.

OpenZeppelin - ERC721 Token

NFT Creation