Solidity Pt. 1

Portland State
Computer Science

Solidity

- Javascript-like programming language for writing programs that run on the Ethereum Virtual Machine
- Domain-specific language that supports abstractions required for operation of smart contracts
  - e.g. contracts, addresses, ownership, payments, hash functions, block information
- Will incrementally learn language using lessons from a guided, on-line Solidity CTF
  - 6 lessons

Lesson 1-2

Basic language features, modifiers, special functions, Web3 events

Mappings, msg object, inheritance, importing code, asserts, exceptions, custom modifiers, storage/memory, calling other contracts

Contract setup

- `pragma` statement to identify compiler version
  - Note that the syntax of Solidity has changed significantly over time
  - Language is a moving target
  - Will learn the version used in the CTFs

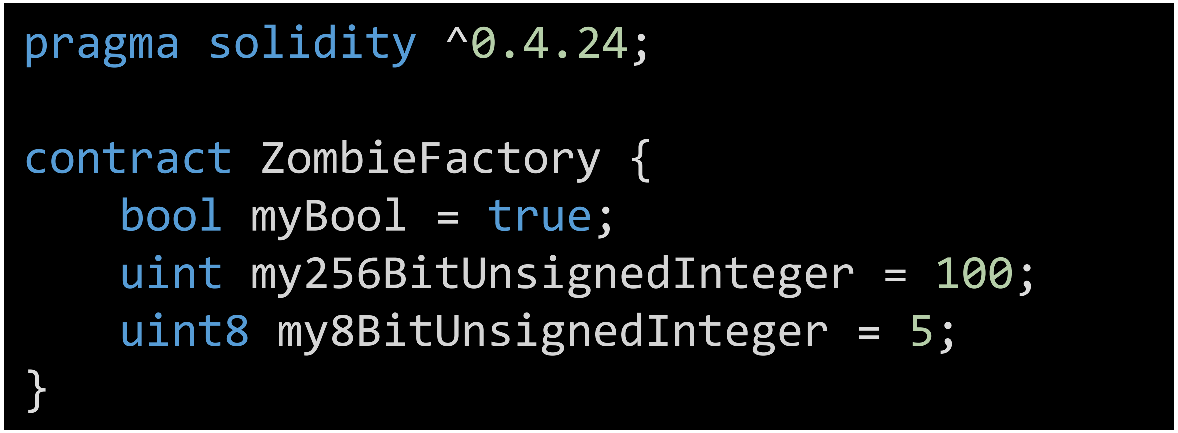
```
pragma solidity ^0.4.24;
```

- `contract` keyword specifies contract code

```
contract HelloWorld {

}
```

Data types

- Boolean (`bool`)
- Signed integers of various widths
  - `int` = 256 bits
  - Can also use smaller versions (to save gas)
    - `int8`, `int16` … etc.
- Unsigned integers of various widths
  - `uint` = 256 bits
  - Can also use smaller versions (to save gas)
    - `uint8`, `uint16` … etc.



```
pragma solidity ^0.4.24;

contract ZombieFactory {
    bool myBool = true;
    uint my256BitUnsignedInteger = 100;
    uint8 my8BitUnsignedInteger = 5;
}
```

- Note: Contract state variables stored on blockchain!

- Aside: Typecasting and coercison between integers
  - Must understand the rules for correctness
  - Implicit cast to higher precision when types mixed

```
uint8 a = 5;
uint b = 6;
// Type of a * b ?
```

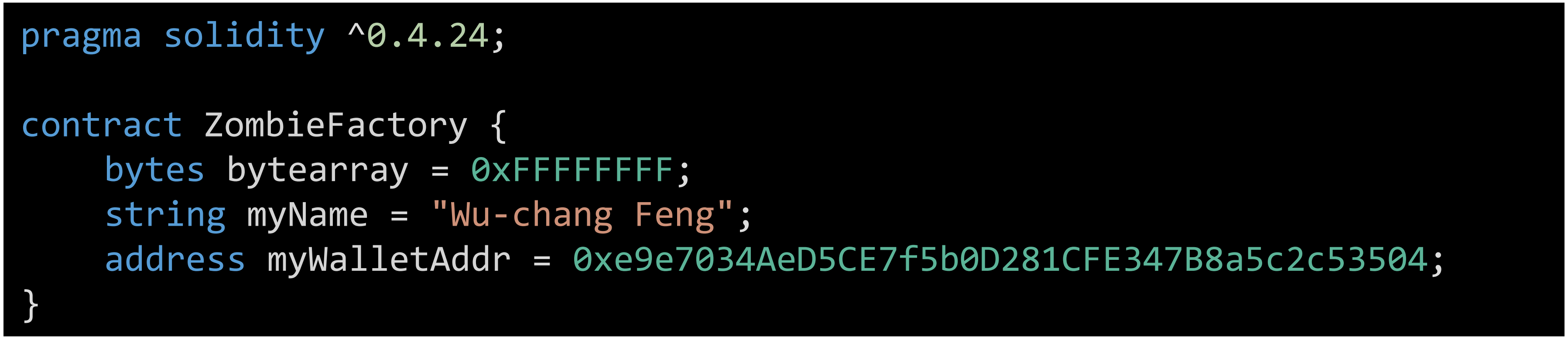
  - Throws an error when types not compatible
    - Product returns a `uint` not a `uint8`

```
// throws an error
uint8 c = a * b;
```

  - Must perform explicit cast to make work

```
uint8 c = a * uint8(b);
```

- bytes
  - Dynamic array of bytes
  - Individual bytes accessed via [] indexing
- string
  - Array of characters
- address
  - 20 byte Ethereum address used to send and receive Ether (in units of Wei)



```
pragma solidity ^0.4.24;

contract ZombieFactory {
    bytes bytearray = 0xFFFFFFFF;
    string myName = "Wu-chang Feng";
    address myWalletAddr = 0xe9e7034AeD5CE7f5b0D281CFE347B8a5c2c53504;
}
```

- Arrays
  - Fixed arrays of typed objects

```
// Fixed array of 2 unsigned integers
uint[2] uintArray;
// Fixed Array of 5 strings:
string[5] stringArray;
```

  - Dynamic arrays of typed objects

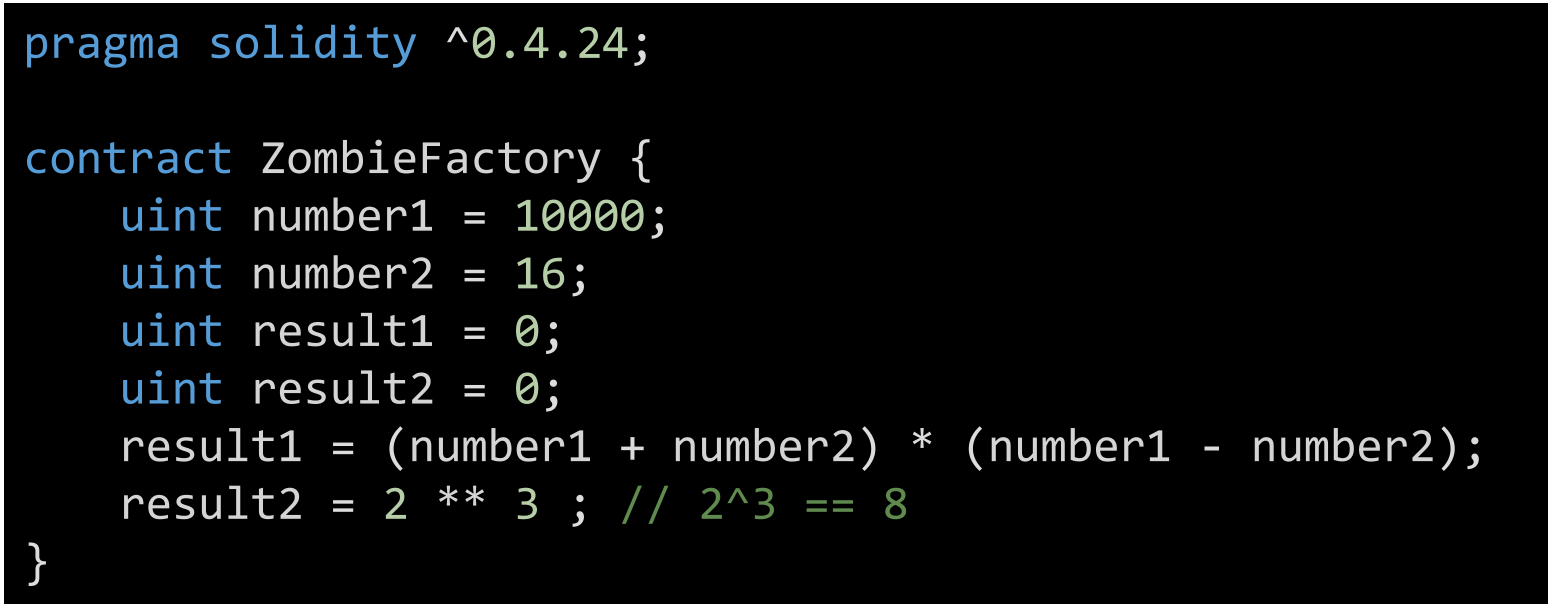
```
// Dynamic array of unsigned integers (can keep growing)
uint[] dynamicArray;
```

    - Add via Array's built-in `push()` method

```
dynamicArray.push(5);
dynamicArray.push(10);
dynamicArray.push(15);
```

Arithmetic operators

+ - * / % ** (exponentiation)



```
pragma solidity ^0.4.24;

contract ZombieFactory {
    uint number1 = 10000;
    uint number2 = 16;
    uint result1 = 0;
    uint result2 = 0;
    result1 = (number1 + number2) * (number1 - number2);
    result2 = 2 ** 3 ; // 2^3 == 8
}
```

Bitwise operators

& | ^ ~ << >>

Logical operators

- Boolean results
  - Negation, AND, OR

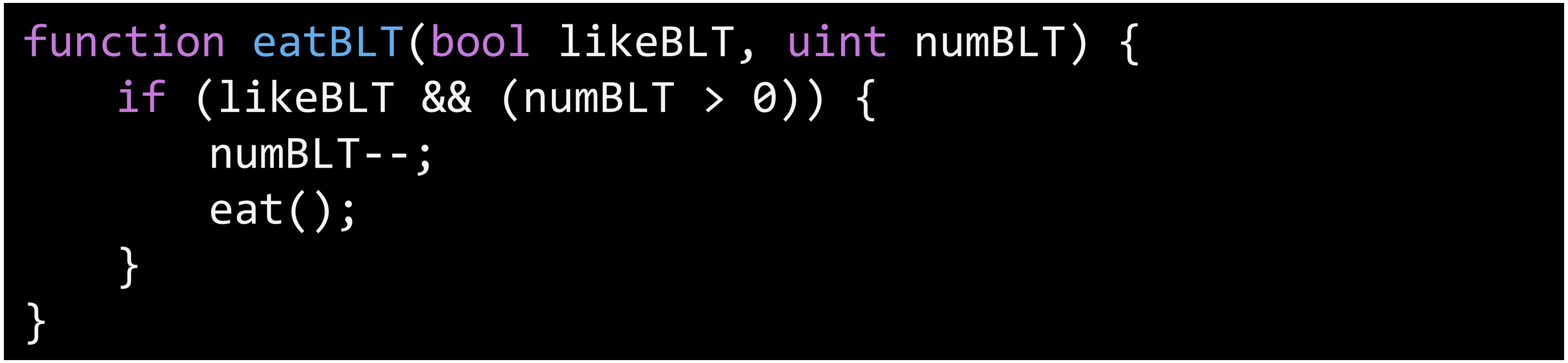
    `!     &&     ||`
  - Equality and inequality

    `==     !=`
  - Magnitude comparisons

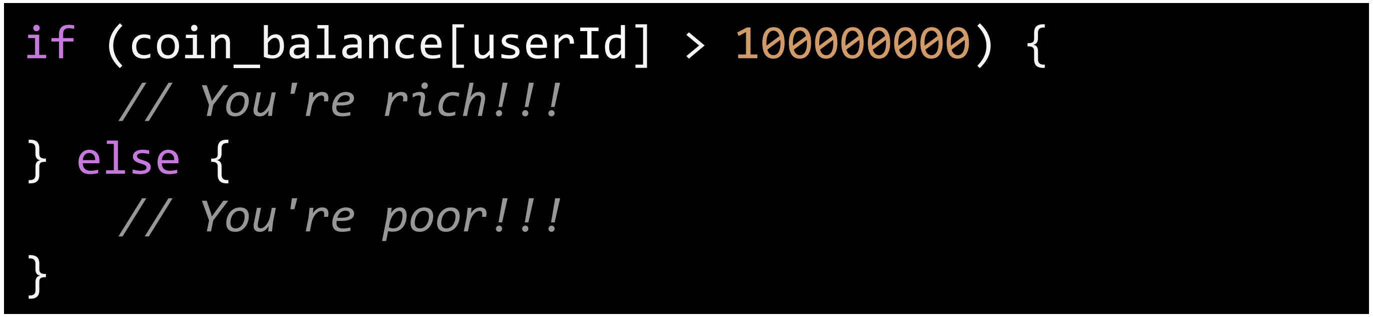
    `<=     >=     <     >`

Conditionals

- Common control flow
  - if, else, while, do, for, break, continue, return



```
function eatBLT(bool likeBLT, uint numBLT) {
    if (likeBLT && (numBLT > 0)) {
        numBLT--;
        eat();
    }
}
```



```
if (coin_balance[userId] > 100000000) {
    // You're rich!!!
} else {
    // You're poor!!!
}
```

- Example `for` loop for creating an array of even numbers

```
uint[] evens = new uint[](5);
uint counter = 0;
for (uint i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        evens[counter] = i;
        counter++;
    }
}
```

Functions, parameters, and return values

- Declared with statically typed parameters & return values
  - Return value specified in function definition via `returns` keyword

```
function sum(uint _input1, uint _input2) returns (uint){
    return(_input1 + _input2);
}
```

Inheritance and polymorphism

- `is` keyword to specify inheritance
- Derive specialized contracts from a more generic one

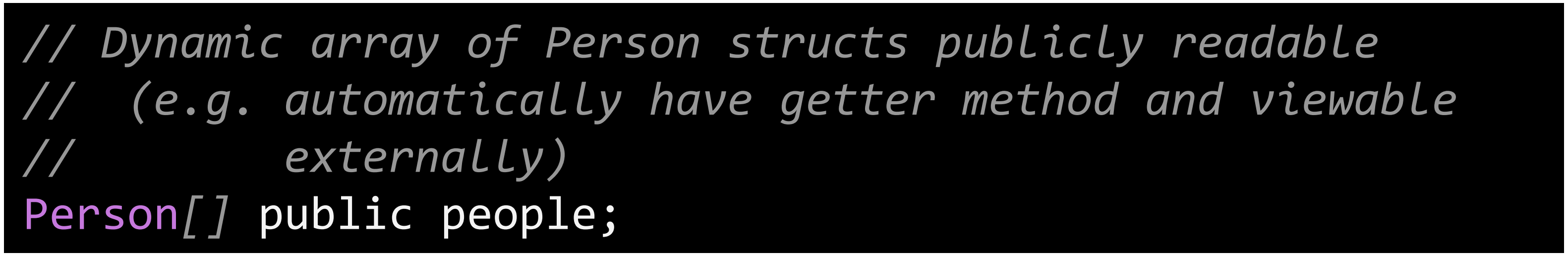
```
contract BasicToken {
    uint totalSupply;
    function balanceOf(address who) returns (uint);
    function transfer(address to, uint value) returns (bool);
}
contract AdvancedToken is BasicToken {
    ...
}
```

- Can inherit from multiple contracts

```
contract SatoshiNakamoto is NickSzabo, HalFinney {

}
```

Visibility modifiers

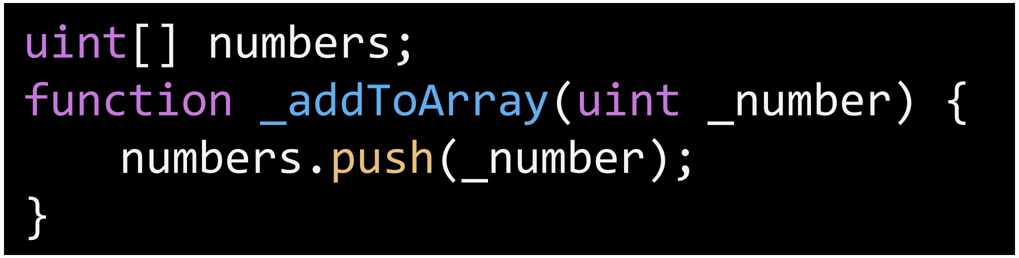
- Modfiers applied to functions and variables to annotate them with where they can be accessed from
  - Software engineering (not a security) mechanism
- `public`
  - Similar to OO languages
  - Functions and variables can be accessed either internally or from any other contract including those derived from it (e.g. from anywhere)



```
// Dynamic array of Person structs publicly readable
//  (e.g. automatically have getter method and viewable
//        externally)
Person[] public people;
```

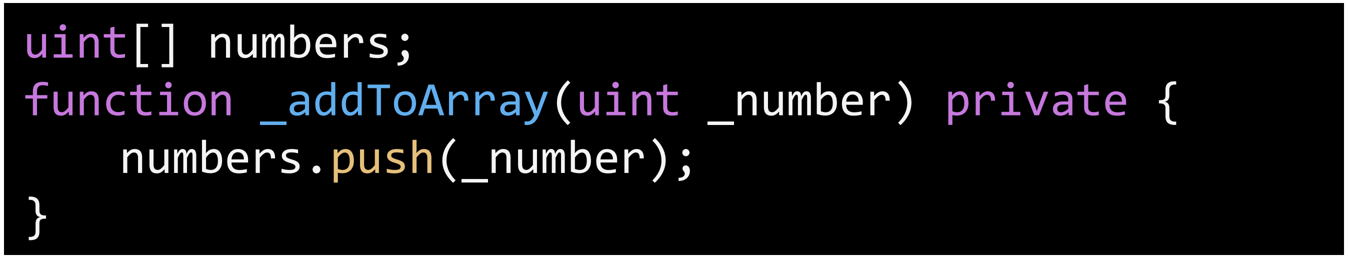
- `private`
  - Function and variable access only to code within contract they are defined in (and not in derived contracts)
  - Note: Do not confuse this with secrecy
    - Data resides on blockchain still!

- If not specified, default `public`
  - Any user or contract can call `_addToArray`



```
uint[] numbers;
function _addToArray(uint _number) {
    numbers.push(_number);
}
```

- Use `private` modifier after parameter declaration to make private
  - Only other functions within our contract can add to array of numbers



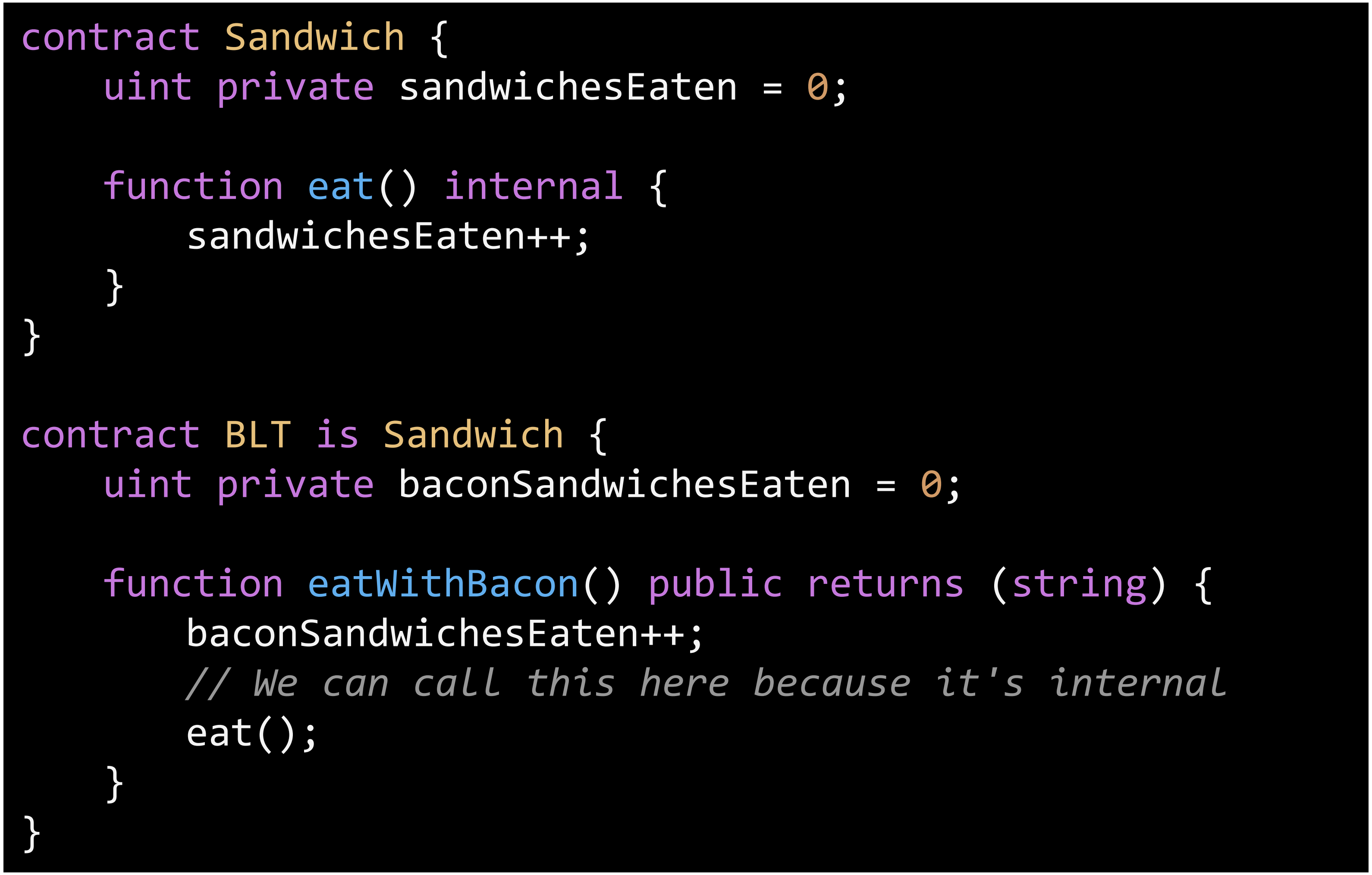
```
uint[] numbers;
function _addToArray(uint _number) private {
    numbers.push(_number);
}
```

  - Array is *still* visible to a full node

Additional visibility modifiers

- `external`
  - Declare as part of the contract interface that can be called
  - Used to construct its application binary interface (ABI)
  - Similar to `public`, but function can *only* be called from outside of the contract by other contracts and via transactions
  - Can not be called internally unless via "`this`" (e.g. `this.f()`)
    - `msg.sender` use contract's address vs address of initial caller
- `internal`
  - Similar to `private`, but allows access both to other code within contract and contracts derived from it via inheritance
  - Akin to `protected` visibility of methods in OO languages

- `eatWithBacon()` callable from anywhere, but `eat()` callable only from derived class
  - No way to eat a sandwich without bacon!



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

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

contract BLT is Sandwich {
    uint private baconSandwichesEaten = 0;

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

Auditing visibility modifiers for security

- Improper setting of `internal/external` and `public/private` are a common source of vulnerabilities
- Ensure all `public` and `external` function calls are intended to be called by anyone!

Modifiers

- Modifiers applied to functions to annotate them with whether they access or modify state
- `view`
  - Does not modify any data in contract

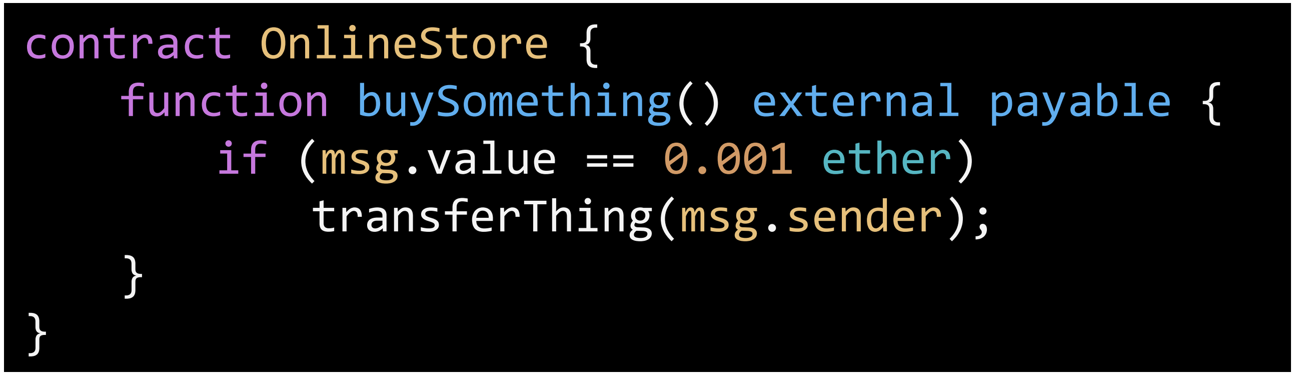
```
string greeting = "What's up dog?";

function sayHello() external view returns (string) {
    return greeting;
}
```

  - Called for free since transaction handled by a single node (light node)
  - Make `external view` functions whenever possible
- `pure`
  - Does not access any data in contract

```
function _multiply(uint a, uint b) private pure returns (uint) {
    return a * b;
}
```

`payable` modifier

- Functions in contracts can accept Ether
  - Unique to Ethereum since money (ether) and contract code/data both stored on blockchain
  - `payable` modifier specifies function that can receive payment
  - Examples
    - Charging caller $ for execution of an API call!
    - Purchase an item in a smart contract



```
contract OnlineStore {
    function buySomething() external payable {
        if (msg.value == 0.001 ether)
            transferThing(msg.sender);
    }
}
```

Constructor function

- Special function executed upon contract creation
  - Example: Initialize number of tokens in an ICO contract

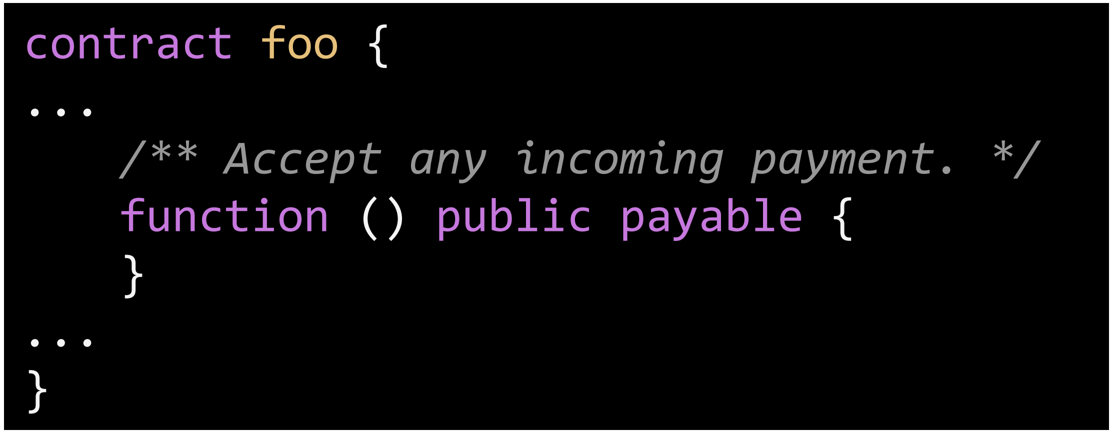
```
contract ICO {
    uint private _totalSupply;
    constructor(uint totalSupply) {
        _totalSupply = totalSupply;
    }
...
}
```

  - Earlier versions specify it as function named after contract

```
contract ICO {
    uint private _totalSupply;
    ICO(uint totalSupply) {
        _totalSupply = totalSupply;
    }
...
}
```

Fallback functions

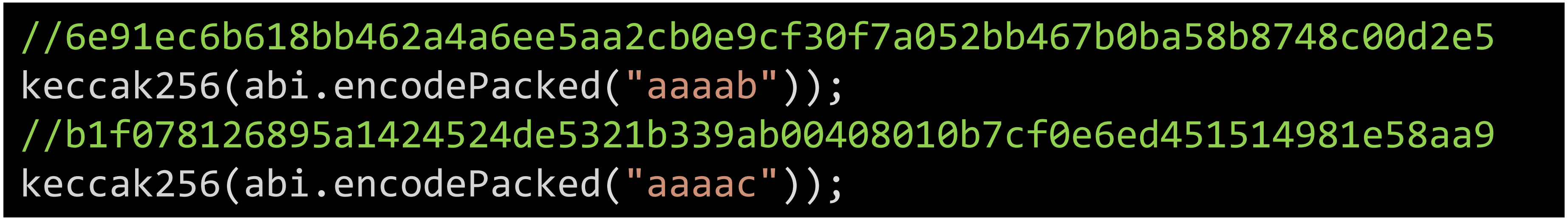
- Contracts can declare precisely one unnamed function in its code that takes no arguments and does not return anything
- Special function that is executed when…
  - Contract is called with a function that does not match any of the functions
  - Contract receives Ether without any data (e.g. an EOA just wants to send money to contract)
    - To actually receive Ether, the fallback function must be marked as "`payable`"
- Part of the EVM design (not Solidity)
  - Often assumed to consume < 2300 gas and to always complete
  - A tenuous assumption when using one smart contract to pay another one



```
contract foo {
...
    /** Accept any incoming payment. */
    function () public payable {
    }
...
}
```

keccak256()

- Native, built-in function for performing a version of SHA3
  - Maps input into a random 256-bit hexadecimal number
  - Slight change in input causes (on average) half of the bits in random number to flip (avalanche effect)



```
//6e91ec6b618bb462a4a6ee5aa2cb0e9cf30f7a052bb467b0ba58b8748c00d2e5
keccak256(abi.encodePacked("aaaab"));
//b1f078126895a1424524de5321b339ab00408010b7cf0e6ed451514981e58aa9
keccak256(abi.encodePacked("aaaac"));
```

  - Note the return is a `bytes32` object not a `uint256`!
    - Bytes are individually indexable in bytes32 while uint256 typically used for single addresses, numbers, and balances

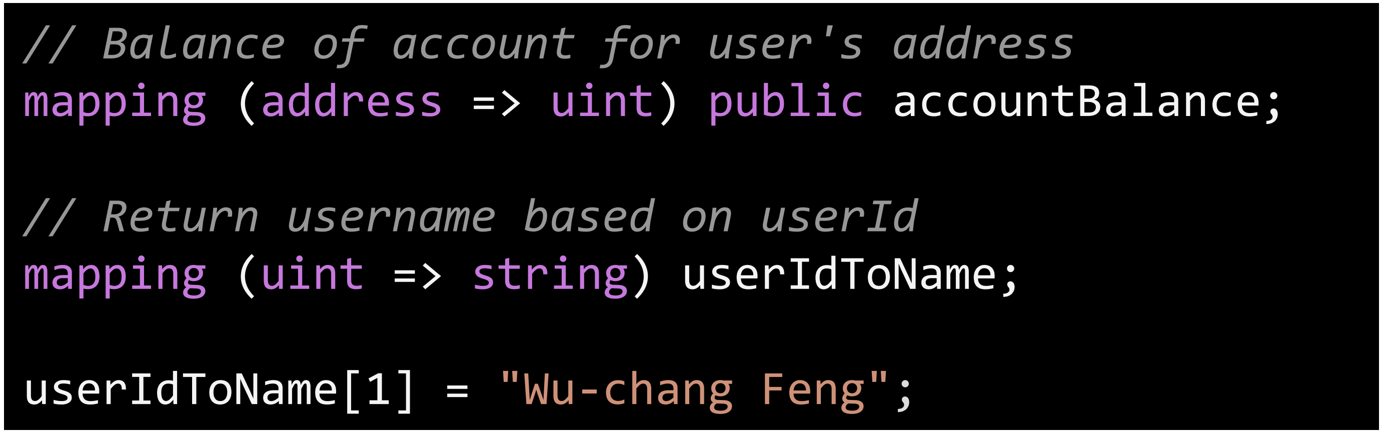
selfdestruct()

- Native, built-in function for destroying a contract and sending its balance to a specific address
  - Will be flagged as a potential vector for denial of service by compiler

```
address beneficiary = 0x38E1a0d... ;

function collect() external {
    // If called after April 14, 2019, send balance
    //    to beneficiary
    if (now > 1555280607)
        selfdestruct(beneficiary);
}
```

Mappings

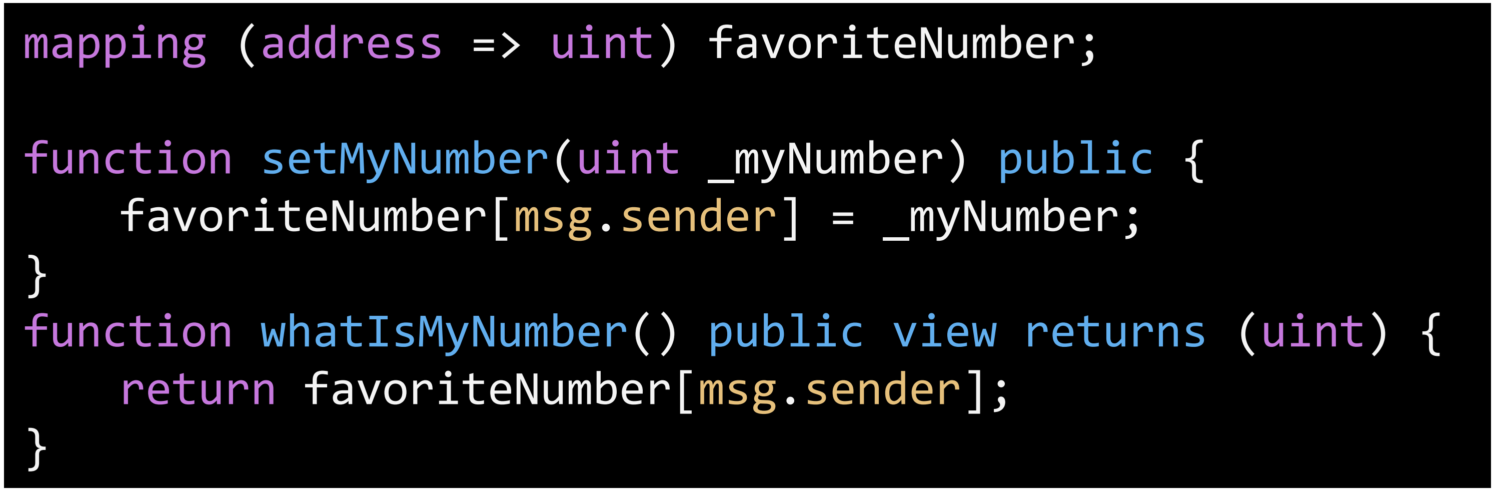
- Data type that implements a dictionary
  - Both keys and entries statically typed
  - Unlike Python dictionaries that can use multiple types for both keys and entries
- Syntax similar to arrays for access



```
// Balance of account for user's address
mapping (address => uint) public accountBalance;

// Return username based on userId
mapping (uint => string) userIdToName;

userIdToName[1] = "Wu-chang Feng";
```

msg

- Special object denoting what caller has sent to contract
  - Various parts of `msg` accessible within contract
  - `msg.sender` : address of caller

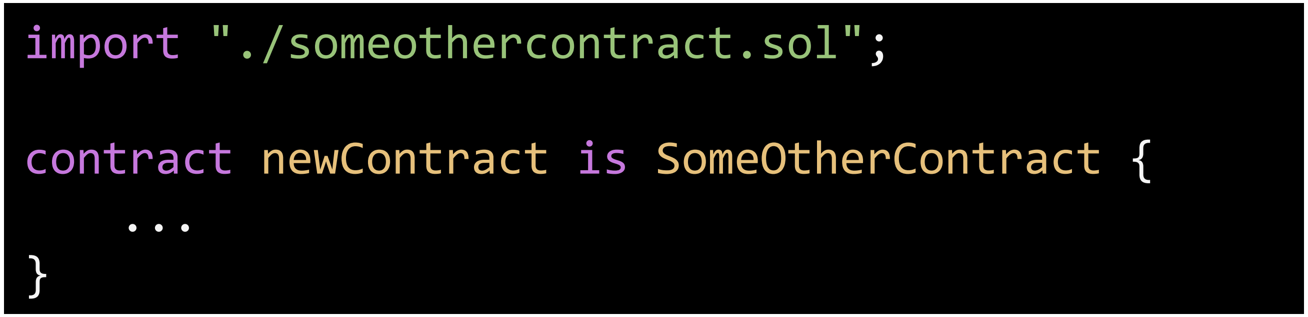


```
mapping (address => uint) favoriteNumber;

function setMyNumber(uint _myNumber) public {
    favoriteNumber[msg.sender] = _myNumber;
}
function whatIsMyNumber() public view returns (uint) {
    return favoriteNumber[msg.sender];
}
```

  - `msg.value` : amount Ether caller has sent in transaction

`import` other code

- Done as source-code
- Typically located as relative path from current directory



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

contract newContract is SomeOtherContract {
    ...
}
```

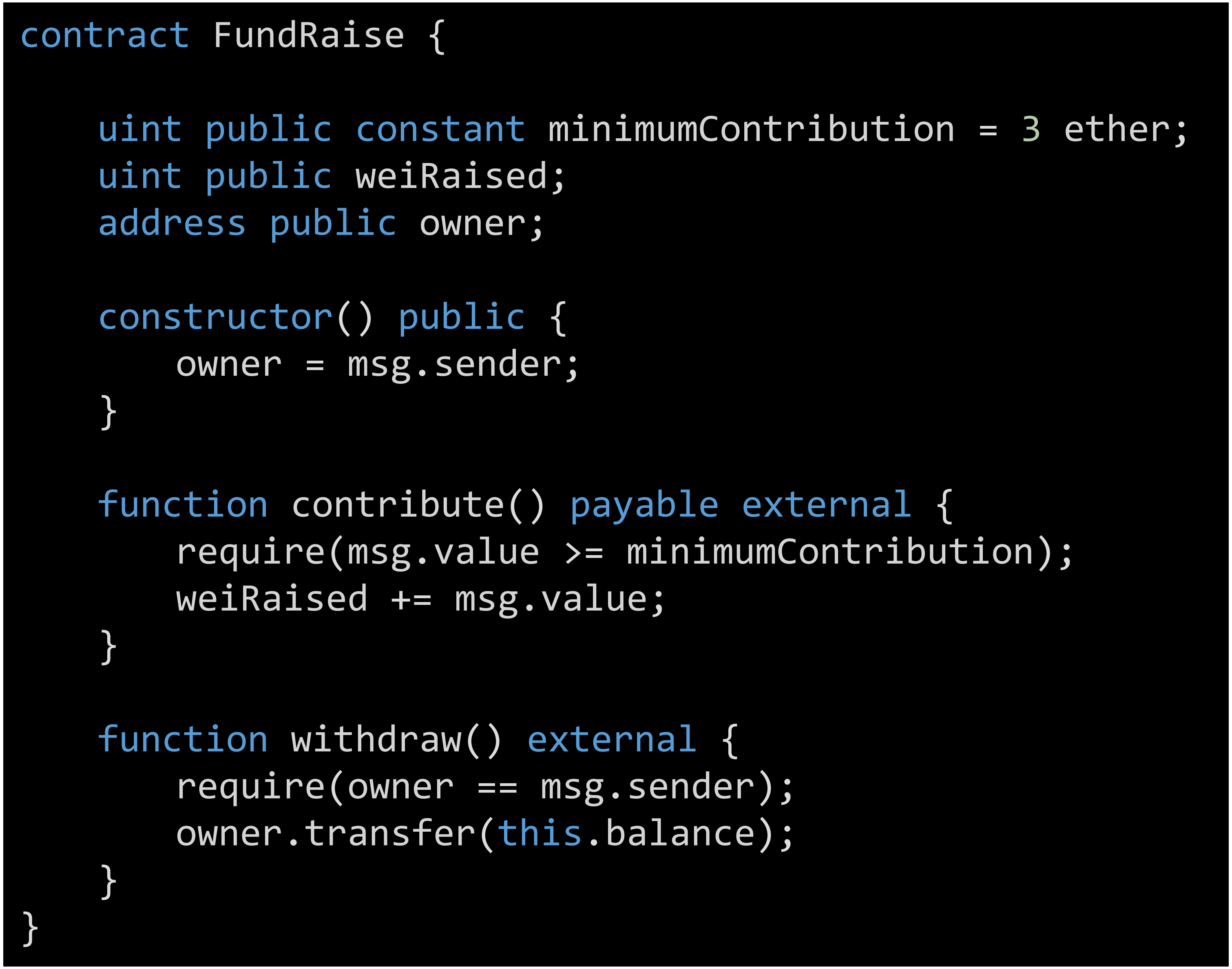
`assert/require` exceptions

- Throw error, stop execution, and revert state if condition not met
  - Exceptions bubble up to caller and cannot be caught
  - `require` used to check externally provided input data
  - `assert` used to check for internal conditions that should not occur

```
function sayHiToVitalik(string _name) public returns (string) {
    // See if _name is "Vitalik" via keccak256 hash
    // Throws an error and exits if not true.
    // No native string comparison in Solidity
    require(keccak256(_name) == keccak256("Vitalik"));
    // If it's true, proceed with the function:
    return "Hi!";
}
```

  - `require` refunds user the rest of their gas when a function fails, `assert` will not
    - Both call `revert()` to undo state and return an error string

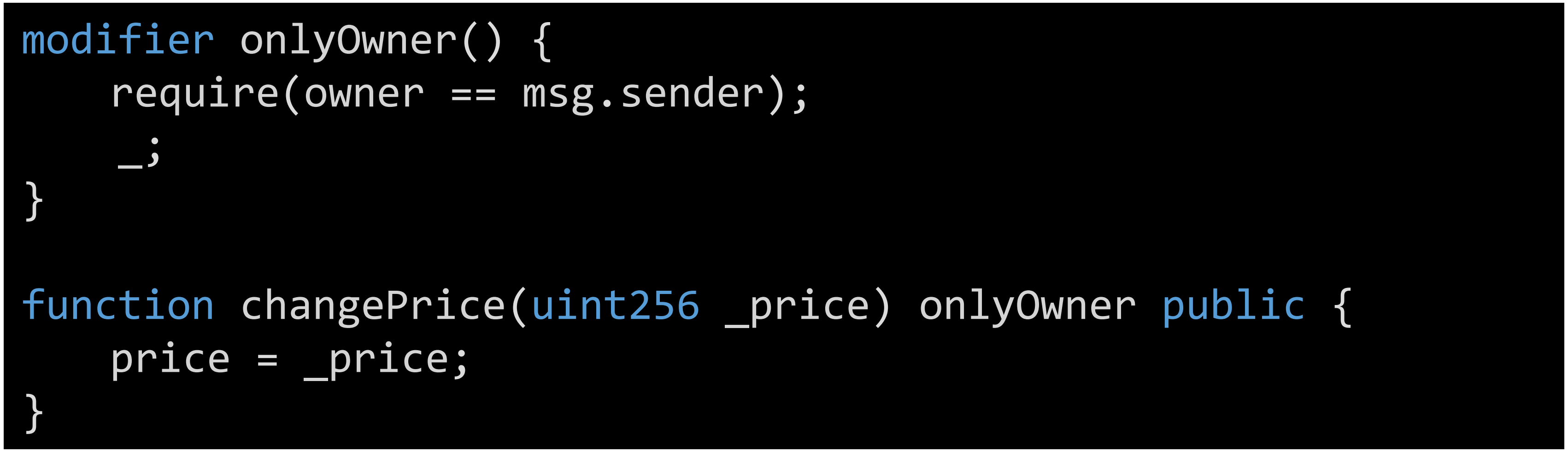
- Ensure `contribute` call has a minimum value
- Ensure `withdraw` is from owner



```
contract FundRaise {

    uint public constant minimumContribution = 3 ether;
    uint public weiRaised;
    address public owner;

    constructor() public {
        owner = msg.sender;
    }

    function contribute() payable external {
        require(msg.value >= minimumContribution);
        weiRaised += msg.value;
    }

    function withdraw() external {
        require(owner == msg.sender);
        owner.transfer(this.balance);
    }
}
```

Custom modifiers with `require`

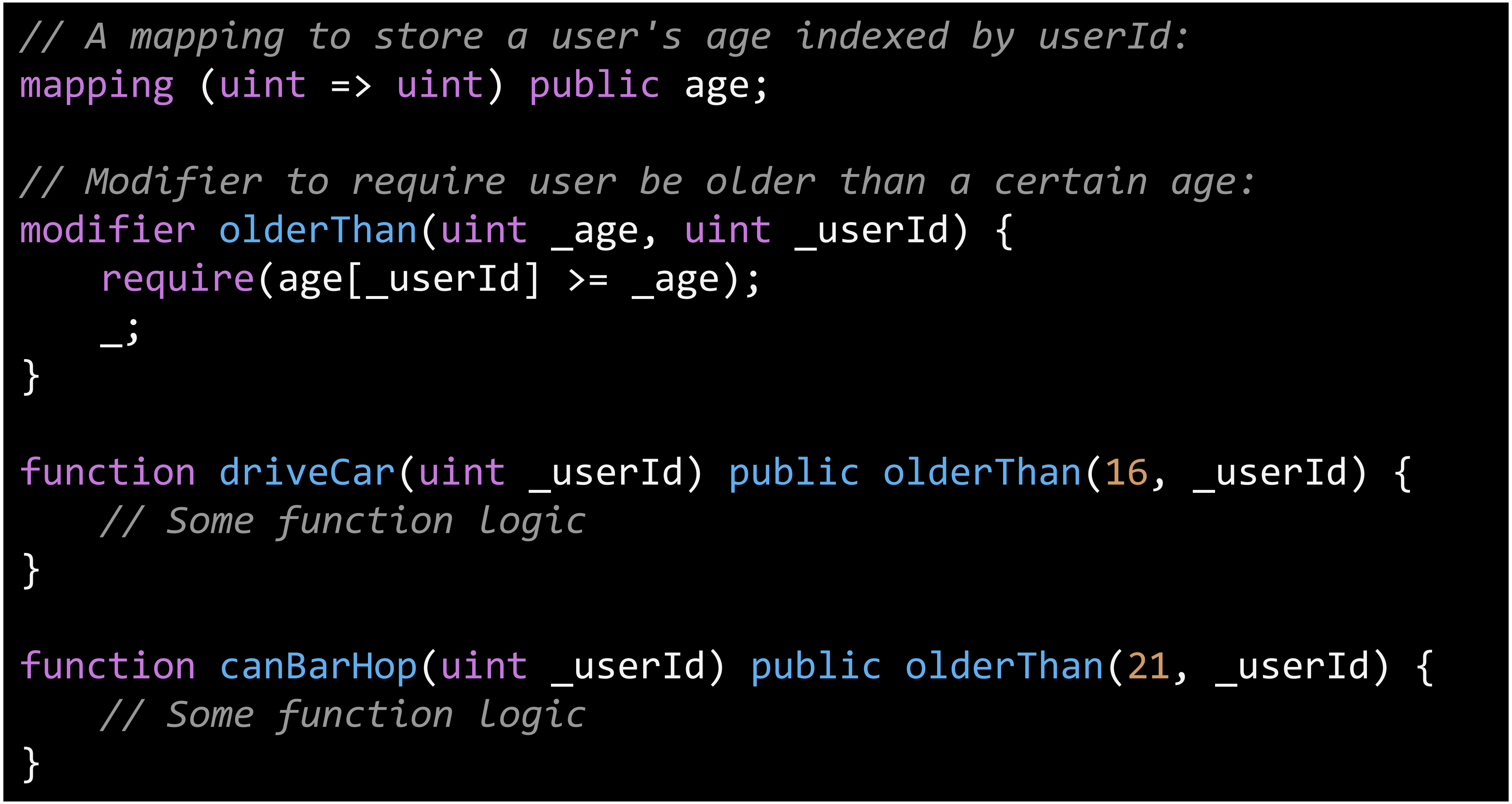
- Often used to amend a function in-line
- Defined using `modifier` keyword
- Modifier must end with `_;` to call original function



```
modifier onlyOwner() {
    require(owner == msg.sender);
    _;
}

function changePrice(uint256 _price) onlyOwner public {
    price = _price;
}
```

- Modifier `onlyOwner` executed when `changePrice` called
- Similar to Python function decorators (430P/530) and detours/trampolines in Windows and x86 (492/592)

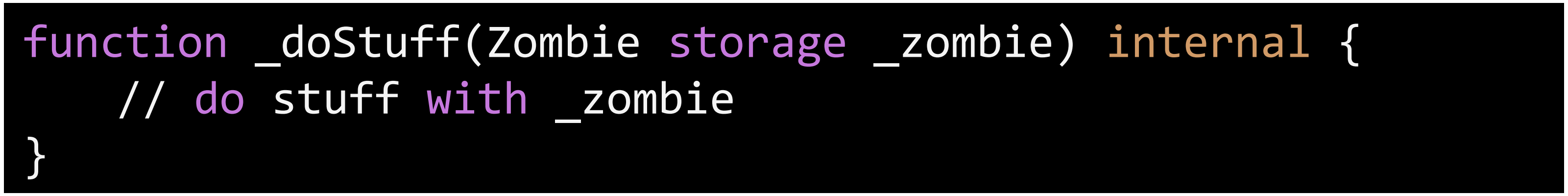
- Modifier can take parameters



```
// A mapping to store a user's age indexed by userId:
mapping (uint => uint) public age;

// Modifier to require user be older than a certain age:
modifier olderThan(uint _age, uint _userId) {
    require(age[_userId] >= _age);
    _;
}

function driveCar(uint _userId) public olderThan(16, _userId) {
    // Some function logic
}

function canBarHop(uint _userId) public olderThan(21, _userId) {
    // Some function logic
}
```

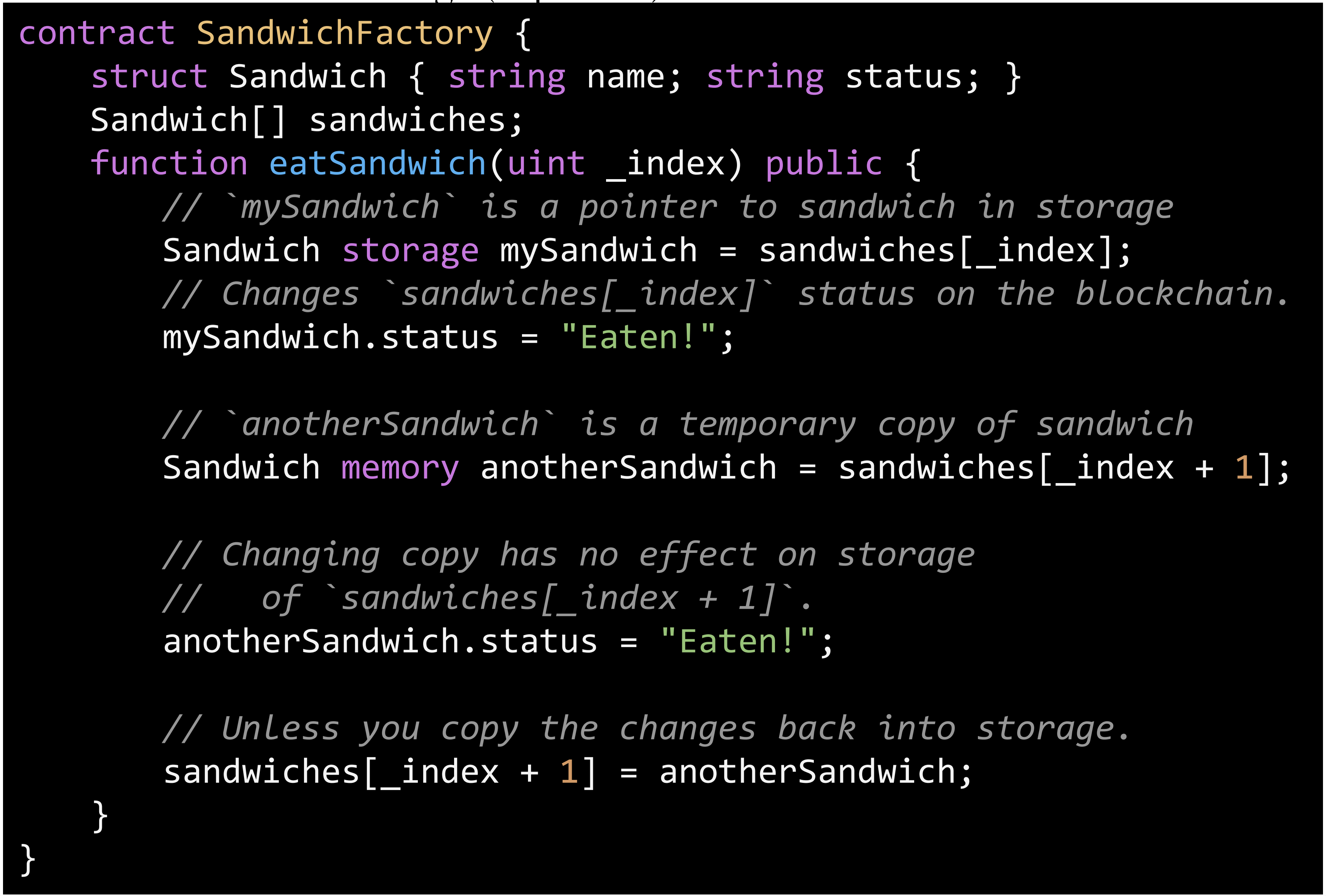
Storage and memory

- Two types of variables
- Storage
  - Persistent storage on blockchain itself (survives between function invocations)
  - Any state variables outside of function call are placed in storage
- Memory
  - Temporary storage used within lifetime of a function execution
  - Any state variables within function calls are placed in temporary memory
  - Disappear when function ends
- Similar to pass by reference (storage) and pass by value (memory)
  - Can specify with keywords `memory` and `storage`



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

- Sandwich on the blockchain accessed and changed (expensive)
- Copy of sandwich in memory (cheap)
  - Written back to storage (expensive)

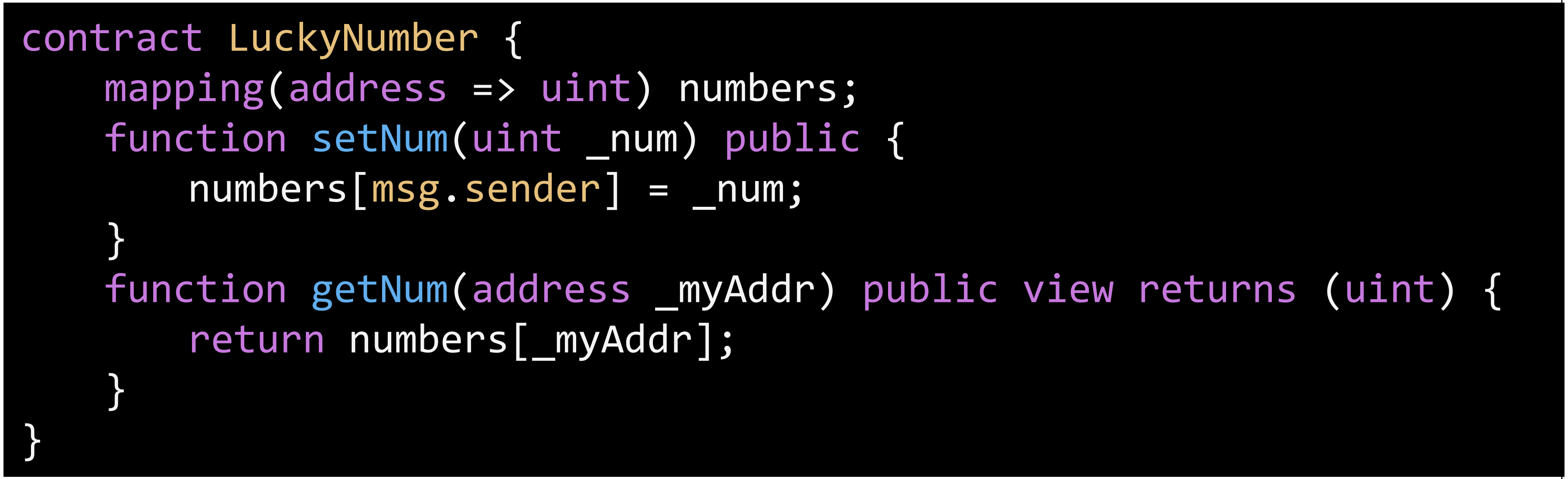


```
contract SandwichFactory {
    struct Sandwich { string name; string status; }
    Sandwich[] sandwiches;
    function eatSandwich(uint _index) public {
        // `mySandwich` is a pointer to sandwich in storage
        Sandwich storage mySandwich = sandwiches[_index];
        // Changes `sandwiches[_index]` status on the blockchain.
        mySandwich.status = "Eaten!";

        // `anotherSandwich` is a temporary copy of sandwich
        Sandwich memory anotherSandwich = sandwiches[_index + 1];

        // Changing copy has no effect on storage
        //   of `sandwiches[_index + 1]`.
        anotherSandwich.status = "Eaten!";

        // Unless you copy the changes back into storage.
        sandwiches[_index + 1] = anotherSandwich;
    }
}
```

- Note: $ storage > $ computation on Ethereum
  - Must optimize to reduce modifications to storage
- Example
  - Keep a list of collectibles a contract has
  - Items can be exchanged at anytime
  - Goal: Return a sorted list of items
    - Strategy #1: Sort in storage (requires significant updates to data on blockchain each time an item is either added or removed)
      - A common vector for bricking a contract
    - Strategy #2: Keep items unsorted, update in-place. Sort items via array in memory
    - Strategy #3: Keep items unsorted, update in-place. Require front-end to sort

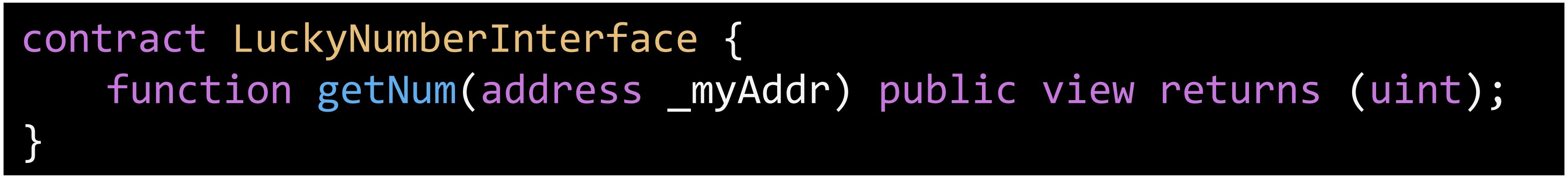
Calling other contracts

- Done via defining contract's calling interface and address
  - Similar to C's ".h" and function linking mechanisms
  - Function call prototype (parameters, return values, and their types) with declaration ending with a semi-colon
- Contract code



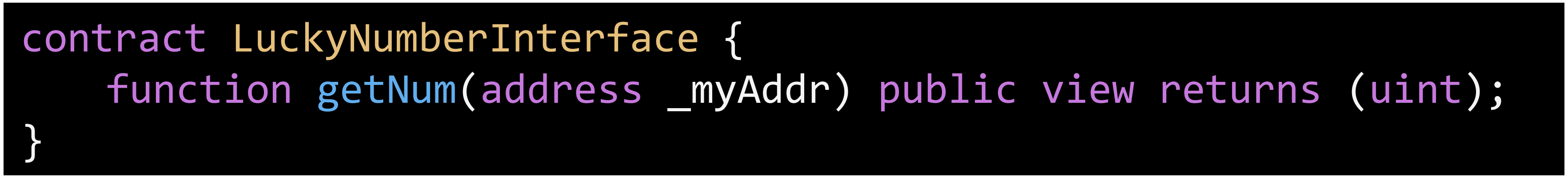
```
contract LuckyNumber {
    mapping(address => uint) numbers;
    function setNum(uint _num) public {
        numbers[msg.sender] = _num;
    }
    function getNum(address _myAddr) public view returns (uint) {
        return numbers[_myAddr];
    }
}
```

- Interface to call contract



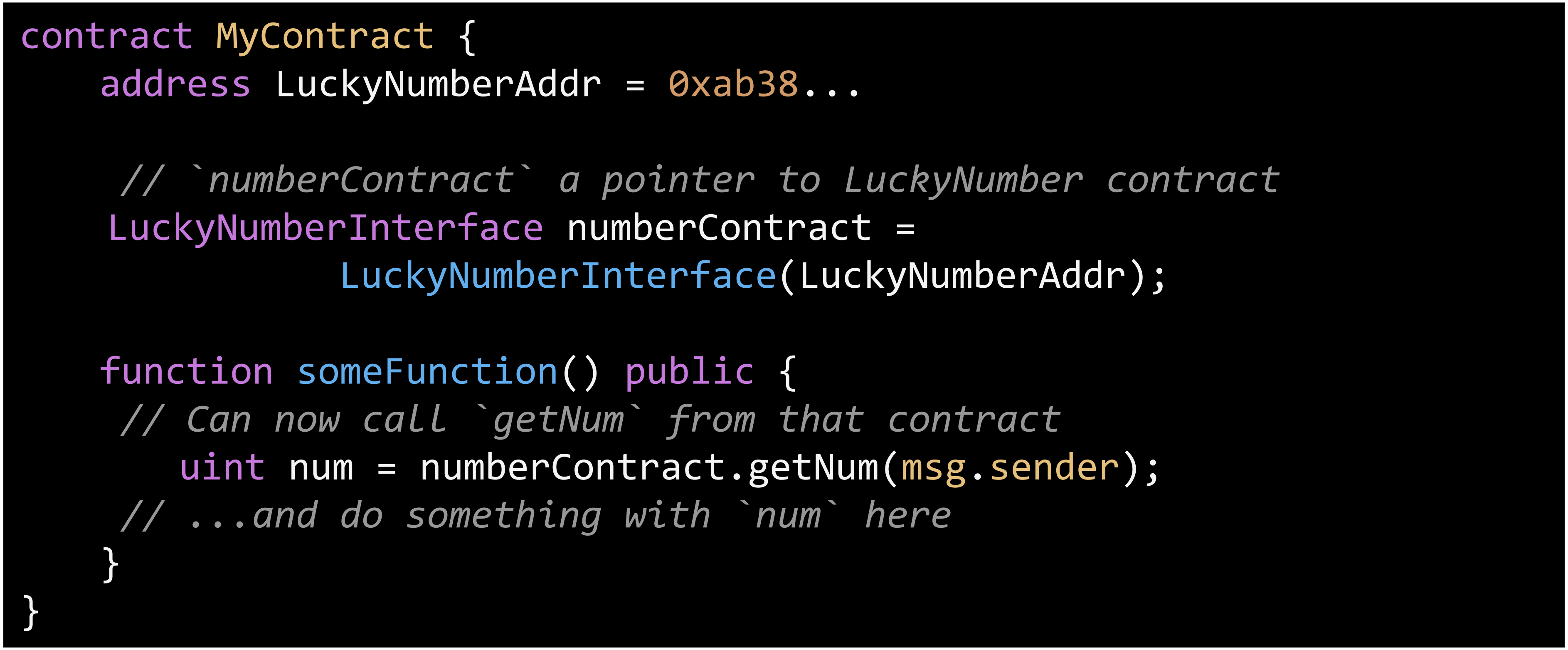
```
contract LuckyNumberInterface {
    function getNum(address _myAddr) public view returns (uint);
}
```

- Interface can now be used to call into `LuckyNumber` contract



```
contract LuckyNumberInterface {
    function getNum(address _myAddr) public view returns (uint);
}
```

- Suppose `LuckyNumber` contract is at `0xab38…`. and we wish to call its `getNum` function from our contract (`MyContract`)



```
contract MyContract {
    address LuckyNumberAddr = 0xab38...

    // `numberContract` a pointer to LuckyNumber contract
    LuckyNumberInterface numberContract =
               LuckyNumberInterface(LuckyNumberAddr);

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

web3.js

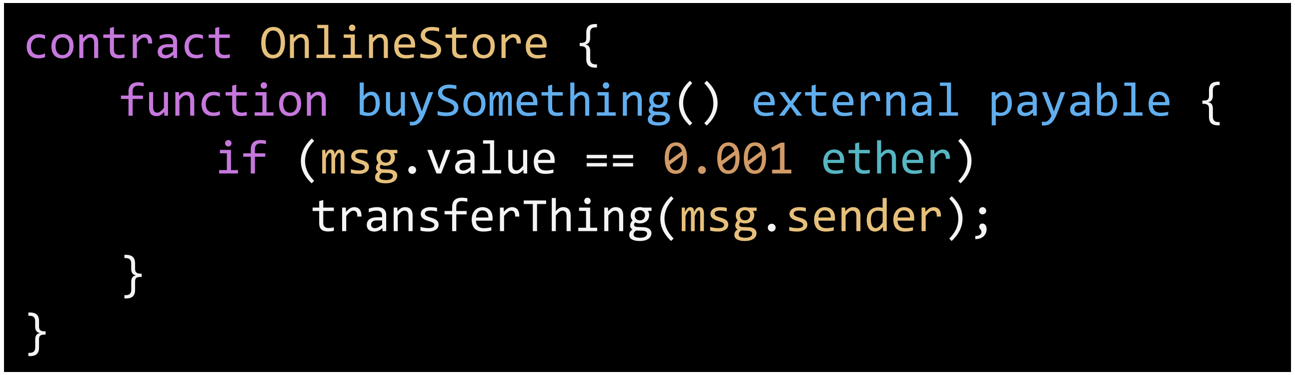
web3.js

- Javascript library to interface Ethereum VM to a front-end web app
  - Provider typically points to a full-node (e.g. Infura), but can be set
  - If `geth` (Ethereum client written in Go) or `Parity` (Ethereum client written in Rust) running locally, then

    ```
    import Web3 from 'web3';
    const web3 = new Web3('http://localhost:8545');
    ```
  - web3.js communicates directly to locally running node
- Also interfaces with a wallet (e.g. Metamask) to provide bridge between user, wallet, browser, and blockchain

web3.js example

- Recall purchasing function in on-line store



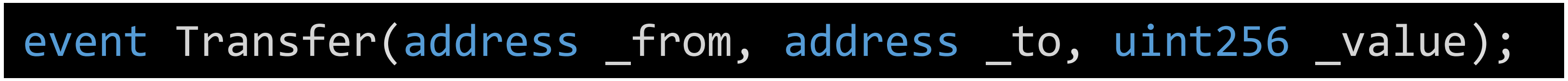
```
contract OnlineStore {
    function buySomething() external payable {
        if (msg.value == 0.001 ether)
            transferThing(msg.sender);
    }
}
```

- JavaScript in web browser to trigger purchase via web3.js
  - `web3.eth.defaultAccount` to connect wallet

```
var abi = /* generated by the compiler */
var OnlineStoreContract = web3.eth.contract(abi)
var contractAddress = 0x1A3... /* contract address on Ethereum */
var OnlineStore = OnlineStoreContract.at(contractAddress)

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

Events

- Used to invoke JavaScript callbacks to send Ethereum events to browser
  - e.g. notify browser (via `web3.js`) that something has happened on the blockchain
- Defined via `event` keyword in Solidity
  - e.g. a transfer that has happened between two accounts on a tokenwill emit…



```
event Transfer(address _from, address _to, uint256 _value);
```

-
  - Javascript via `web3.js` updates browser UI to show transfer
    - Used to generate update UI and generate Javascript popup in CTF

- Example
  - Event notification in smart contract

```
// Declare event
event IntegersAdded(uint x, uint y, uint result);

function add(uint _x, uint _y) public {
    uint result = _x + _y;
    // Notify app that function was called:
    emit IntegersAdded(_x, _y, result);
    return result;
}
```

  - Emit in function execution triggers JavaScript callback in browser (more later)

```
YourContract.IntegersAdded(function(error, result) {
   // Do something with result (e.g. update UI)
}
```