SMART CONTRACTS: SOLIDITY SYNTAX

Minxing Chen

LECTURE OVERVIEW

1. SOLIDITY OVERVIEW
2. DATA
3. FUNCTIONS, LOOPS, AND MORE
4. ADVANCED SOLIDITY
5. DEMO

BLOCKCHAIN AT BERKELEY

1 SOLIDITY OVERVIEW

THE BANK CONTRACT

LEARNING FAST

WHAT DOES A BANK NEED TO DO?

1. Allow Deposits
2. Allow Withdrawals
3. Balance Checks

Learn X in Y minutes, a whirlwind tour of your favorite language

```
contract SimpleBank {
    mapping (address => uint) private balances;
    address public owner;
    event LogDepositMade(address accountAddress, uint amount);

    function SimpleBank() {
        owner = msg.sender;
    }

    function deposit() public returns (uint) {
        balances[msg.sender] += msg.value;
        LogDepositMade(msg.sender, msg.value);
        return balances[msg.sender];
    }

    function withdraw(uint withdrawAmount) public returns (uint remainingBal) {
        if(balances[msg.sender] >= withdrawAmount) {
            balances[msg.sender] -= withdrawAmount;
            if (!msg.sender.send(withdrawAmount)) {
                balances[msg.sender] += withdrawAmount;
            }
        }
        return balances[msg.sender];
    }

    function balance() constant returns (uint) {
        return balances[msg.sender];
    }

    function () {
        throw;
    }
}
```

AUTHOR: COLLIN CHIN

THE BANK CONTRACT

LEARNING FAST

- **contract** has similarities to **class** in other languages (class variables, inheritance, etc.)
  - Declare state variables outside function, persist through life of contract
- **mapping** is a dictionary that maps addresses to balances
  - always be careful about overflow attacks with numbers
  - private means that other contracts can't directly query balances
  - but data is still viewable to other parties on blockchain
- **public** makes externally readable (not writeable) by users or contracts

```
1  contract SimpleBank {
2
3
4
5      mapping (address => uint) private balances;
6
7
8
9
10     address public owner;
11
12
```

AUTHOR: COLLIN CHIN

THE BANK CONTRACT

LEARNING FAST

- **event** - publicize actions to external listeners
- **Constructor** - can receive one or many variables here; only one allowed
- **msg** provides details about the message that's sent to the contract
  - **msg.sender** is contract caller (address of contract creator)

```
4     event LogDepositMade(address accountAddress, uint amount);
5
6     function SimpleBank() {
7         owner = msg.sender;
8     }
```

AUTHOR: COLLIN CHIN

THE BANK CONTRACT

LEARNING FAST

- **deposit()**
  - Takes no parameters, but we are still sending Ether!
  - public makes externally readable (not writeable) by users or contracts
  - Returns user's balance as an unsigned integer (uint)
- balances[msg.sender], no this or self required with state variable
- LogDepositMade event fired

```
14    function deposit() public returns (uint) {
15
16
17        balances[msg.sender] += msg.value;
18
19
20        LogDepositMade(msg.sender, msg.value);
21
22
23        return balances[msg.sender];
24    }
```

AUTHOR: COLLIN CHIN

THE BANK CONTRACT

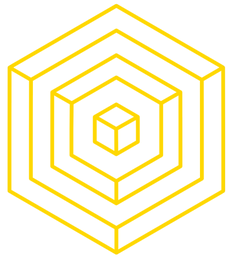
LEARNING FAST

```
function withdraw(uint withdrawAmount) public returns (uint remainingBal) {

    if(balances[msg.sender] >= withdrawAmount) {

        balances[msg.sender] -= withdrawAmount;

        if (!msg.sender.send(withdrawAmount)) {

            balances[msg.sender] += withdrawAmount;
        }
    }
    return balances[msg.sender];
}
```

- **Withdraw()**
  - **withdrawAmount** parameter
  - Returns user's balance
- Note the way we deduct the balance right away, before sending
  - We do this because of the risk of a recursive call that allows the caller to request an amount greater than their balance
- Increment back only on fail, as may be sending to contract that has overridden 'send' on the receipt end

AUTHOR: COLLIN CHIN

2 DATA



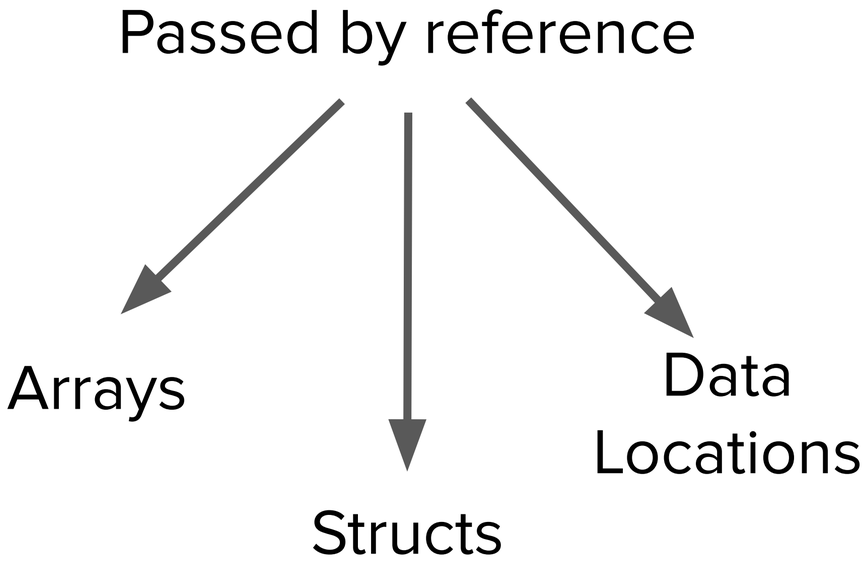
2.1 DATA TYPES

CATEGORIES OF TYPES
DATA TYPES

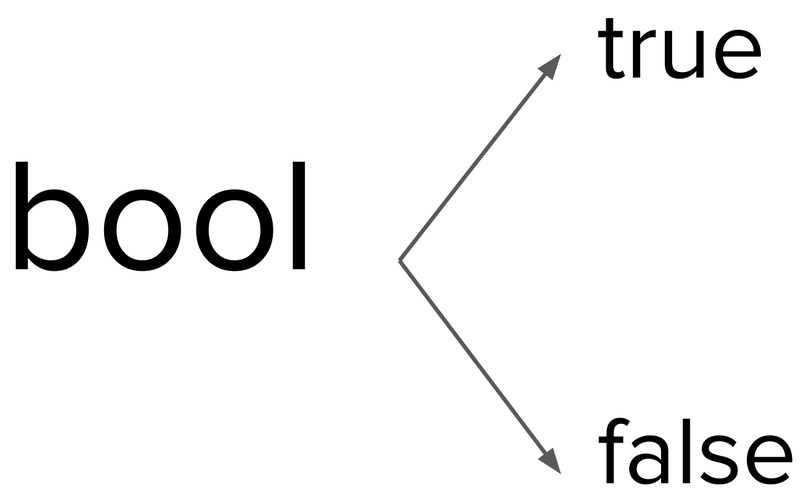
VALUE TYPES

Passed by value

REFERENCE TYPES



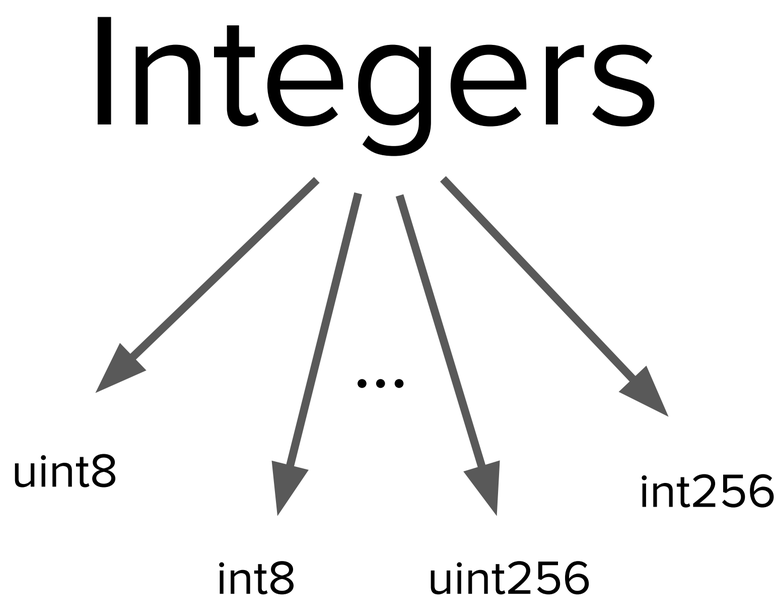
BOOLEANS
VALUE TYPES



Operators:

- ! (logical negation)
- && (logical conjunction, “and”)
- || (logical disjunction, “or”)
- == (equality)
- != (inequality)

INTEGERS
VALUE TYPES



- int/uint ranges from int8/uint8 to int256/uint256.(In steps of 8)
- Int/uint are alias of int256/uint256

Operators:

- Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
- Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
- Shift operators: << (left shift), >> (right shift)
- Arithmetic operators: +, -, unary -, *, /, % (modulo), ** (exponentiation)

INTEGERS
VALUE TYPE

Bits Operation

~int256(0) == int256(-1)

Shifts

x << y == x * 2**y

x >> y == x / 2 ** y

INTEGERS
VALUE TYPE

Addition / Subtraction

uint256(0) - uint256(1) ==
2**256 - 1

Multiplication / Division

int256(-5) / int256(2) ==
int256(-2)

ADDRESS
VALUE TYPE

Address & Address Payable

- Address: Holds a 20 byte value (size of an Ethereum address).

- Address Payable: Same as address, but with the additional members transfer and send.

Operators:

- <=, <, ==, !=, >= and >

ADDRESS PAYABLE
VALUE TYPE

Transfer & Send

```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 &&
myAddress.balance >= 10)
x.transfer(10);
```

The transfer function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The transfer function reverts on failure.

Send is the low-level counterpart of transfer. If the execution fails, the current contract will not stop with an exception, but send will return false.

CONTRACT TYPES

VALUE TYPE

```
pragma solidity ^0.5.0;

contract D {
    uint public x;
    constructor(uint a) public payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's
constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount)
public payable {
        // Send ether along with the creation
        D newD = (new D).value(amount)(arg);
        newD.x();
    }
}
```

- Contracts do not support any operators.
- The members of contract types are the external functions of the contract including public state variables.
- For a contract C you can use type(C) to access type information about the contract.

STRING LITERALS
VALUE TYPES

bytes32 a = "stringliteral"

string b = "stringliteral"

a == b

Hex “DEADBEEF”

"**\n**\"\'\\abc\
def"

- \<newline> (escapes an actual newline)
- \\ (backslash)
- \' (single quote)
- \" (double quote)
- \b (backspace)
- \f (form feed)
- \n (newline)
- \r (carriage return)
- \t (tab)
- \v (vertical tab)
- \xNN (hex escape))
- \uNNNN (unicode escape))

ENUM
VALUE TYPES

```
pragma solidity >=0.4.16 <0.6.0;

contract test {
    enum ActionChoices { GoLeft, GoRight,
GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice =
ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of
"getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity. The integer type used is
just
    // large enough to hold all enum values, i.e. if you have more
than 256 values,
    // `uint16` will be used and so on.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }

    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }
}
```

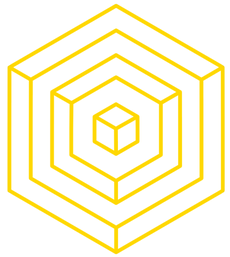
PRECEDENCE
VALUE TYPES

Precedence	Description	Operator
1	Postfix increment and decrement	`++` , `--`
	New expression	`new <typename>`
	Array subscripting	`<array>[<index>]`
	Member access	`<object>.<member>`
	Function-like call	`<func>(<args...>)`
	Parentheses	`(<statement>)`
2	Prefix increment and decrement	`++` , `--`
	Unary plus and minus	`+` , `-`
	Unary operations	`delete`
	Logical NOT	`!`
	Bitwise NOT	`~`
3	Exponentiation	`**`
4	Multiplication, division and modulo	`*` , `/` , `%`
5	Addition and subtraction	`+` , `-`
6	Bitwise shift operators	`<<` , `>>`
7	Bitwise AND	`&`
8	Bitwise XOR	`^`
9	Bitwise OR	`|`
10	Inequality operators	`<` , `>` , `<=` , `>=`
11	Equality operators	`==` , `!=`
12	Logical AND	`&&`
13	Logical OR	`||`
14	Ternary operator	`<conditional> ? <if-true> : <if-false>`
15	Assignment operators	`=` , `|=` , `^=` , `&=` , `<<=` , `>>=` , `+=` , `-=` , `*=` , `/=`
16	Comma operator	`,`

2.2 DATA STRUCTURES

DATA STRUCTURES
ARRAYS

```
// Arrays
bytes32[5] nicknames; // static array
bytes32[] names; // dynamic array
uint newLength = names.push("John"); // adding returns new length of the array
// Length
names.length; // get length
names.length = 1; // lengths can be set (for dynamic arrays in storage only)

// multidimensional array
uint x[][5]; // arr with 5 dynamic array elements (opp order of most languages)
```

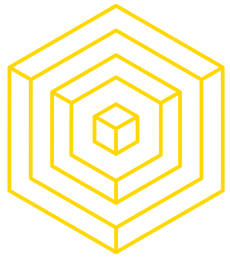



DATA STRUCTURES
STRUCTS

```
struct Bank {
    address owner;
    uint balance;
}
Bank b = Bank({
    owner: msg.sender,
    balance: 5
});
// or
Bank c = Bank(msg.sender, 5);

c.balance = 5; // set to new value
delete b;
// sets to initial value, set all variables in struct to 0, except mappings
```

DATA STRUCTURES
MAPPINGS

```
// Dictionaries (any type to any other type)
mapping (string => uint) public balances;
balances["charles"] = 1;
// balances["ada"] result is 0, all non-set key values return zeroes
// 'public' allows following from another contract
contractName.balances("charles"); // returns 1
// 'public' created a getter (but not setter) like the following:
function balances(string _account) returns (uint balance) {
    return balances[_account];
}
```



DATA STRUCTURES
MAPPINGS

```
// Nested mappings
mapping (address => mapping (address => uint)) public custodians;

// To delete
delete balances["John"];
delete balances; // sets all elements to 0
```

3 FUNCTIONS

Function basics

access classifier

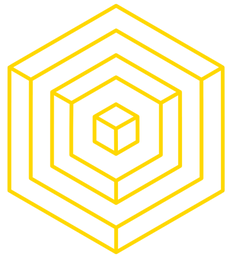
name **argType1 arg1, ...** **returnType (optional return var name)**

```
function withdraw(uint withdrawAmount) public returns (uint remainingBal) {
    require(withdrawAmount <= balances[msg.sender]);

    // Note the way we deduct the balance right away, before sending
    // Every .transfer/.send from this contract can call an external function
    // This may allow the caller to request an amount greater
    // than their balance using a recursive call
    // Aim to commit state before calling external functions, including .transfer/.send
    balances[msg.sender] -= withdrawAmount;

    // this automatically throws on a failure, which means the updated balance is reverted
    msg.sender.transfer(withdrawAmount);

    return balances[msg.sender];
}
```

- public - all can access
- external - Cannot be accessed internally, only externally
- internal - only this contract and contracts deriving from it can access
- private - can be accessed only from this contract

IMPORTANT:
Access classifiers determine who can USE your functions, but everyone can see them, since they will be deployed on a public blockchain.

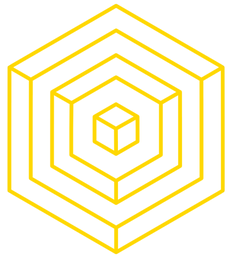
AUTHOR: NISHAAD NAVKAL



Payable functions

Only functions marked as payable can receive Ether
Non-payable functions with ether values will be routed to default function

```
function deposit() public payable returns (uint) {
    // Use 'require' to test user inputs, 'assert' for internal invariants
    // Here we are making sure that there isn't an overflow issue
    require((balances[msg.sender] + msg.value) >= balances[msg.sender]);

    balances[msg.sender] += msg.value;
    // no "this." or "self." required with state variable
    // all values set to data type's initial value by default

    LogDepositMade(msg.sender, msg.value); // fire event

    return balances[msg.sender];
}
```



Fallback function

- Invoked when a function is called which does not match any other contract functions
- Only one per contract
- No arguments, returns nothing
- Typically payable - enables contract to receive ether sent directly to it

Can be thought of as default behavior when the contract does not recognize the command

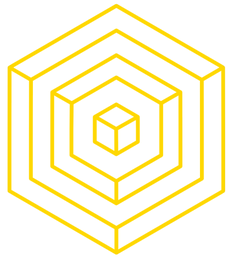
```
function() external payable {
    require(msg.value >= prize || msg.sender == owner);
    king.transfer(msg.value);
    king = msg.sender;
    prize = msg.value;
}
```

Constructors

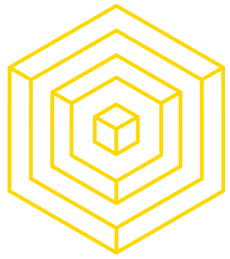
- Create an instance of the contract with the given arguments
- Only one allowed, cannot be overloaded
- 2 implementations:
  - function [contractName] (arg1, arg2 ...)
  - constructor (arg1, arg2 …)

```
contract SimpleBank { // CapWords
    // Declare state variables outside function, persist through life of contract

    // dictionary that maps addresses to balances
    // always be careful about overflow attacks with numbers
    mapping (address => uint) private balances;

    // "private" means that other contracts can't directly query balances
    // but data is still viewable to other parties on blockchain

    address public owner;
    // 'public' makes externally readable (not writeable) by users or contracts

    // Events - publicize actions to external listeners
    event LogDepositMade(address accountAddress, uint amount);

    // Constructor, can receive one or many variables here; only one allowed
    function SimpleBank() public {
        // msg provides details about the message that's sent to the contract
        // msg.sender is contract caller (address of contract creator)
        owner = msg.sender;
    }
```

Break



4 ADVANCED SOLIDITY

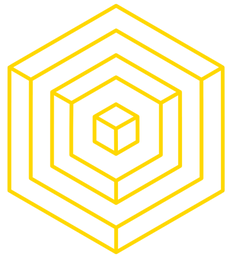


4.1 GLOBAL VARIABLES

GLOBAL VARIABLES
CURRENCY UNITS

- In Solidity, currency units are tracked as **uints**

- A number can take a postfix of **wei**, **finney**, **szabo** or **ether** to convert between denominations of Ether.



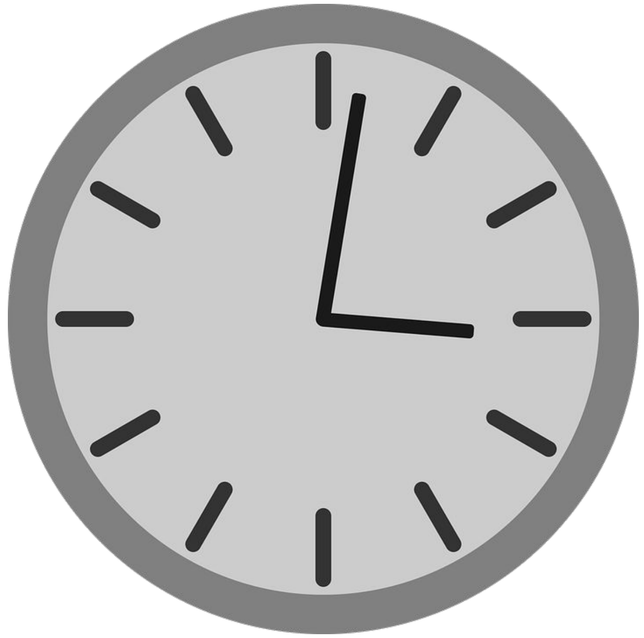
GLOBAL VARIABLES
CURRENCY UNITS

```
// Currency units
// Currency is defined using wei, smallest unit of Ether
uint minAmount = 1 wei;
uint a = 1 ether; // 1 ether == 10**18 wei
uint b = 1 finney; // 1 ether == 1000 finney

// Currency units without a postfix are assumed to be wei
require(a == 10**15 && b == 10**18); // true
```

GLOBAL VARIABLES
TIME

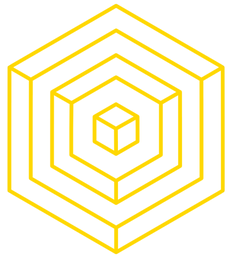
- Solidity tracks time as a **Unix TimeStamp**, stored as a **uint**
  - Find current timestamp: https://www.unixtimestamp.com/index.php

- Similar to currency units, a number can take a postfix of **seconds, minutes, hours, etc**. to convert between time denominations



GLOBAL VARIABLES
TIME

```
// Time units
1 == 1 seconds
1 minutes == 60 seconds
1 hours == 60 minutes
1 days == 24 hours
1 weeks == 7 days
1 years == 365 days

now; // returns current Unix TimeStamp
// Note that this can be manipulated by miners, so use carefully
```



GLOBAL VARIABLES
THIS

```
// ** this **
this; // address of contract
// often used at end of contract life to transfer remaining balance to party
this.balance;
this.someFunction(); // calls func externally via call, not via internal jump
```

GLOBAL VARIABLES

MSG, TX

```
// ** msg - Current message received by the contract ** **
msg.sender; // address of sender
msg.value; // amount of ether provided to this contract in wei, function should be marked "payable"
msg.data; // bytes, complete call data
msg.gas; // remaining gas
```

```
// ** tx - This transaction **
tx.origin; // address of sender of the transaction
tx.gasprice; // gas price of the transaction
```


GLOBAL VARIABLES

MSG.SENDER VS TX.ORIGIN

Externally owned account at address A

Contract at address B

Contract at address C

AUTHOR: TIM HUANG

GLOBAL VARIABLES

MSG.SENDER VS TX.ORIGIN

Externally owned account at address A

Contract at address B

Contract at address C

Msg.sender: address A

Tx.origin: address A

AUTHOR: TIM HUANG

GLOBAL VARIABLES

MSG.SENDER VS TX.ORIGIN

Externally owned account at address A

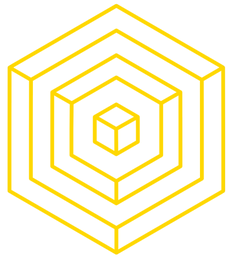
Contract at address B

Contract at address C

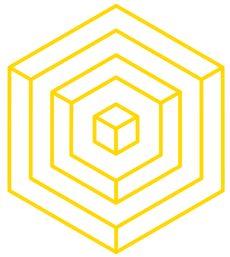
Msg.sender: address A
Tx.origin: address A

Msg.sender: address B
Tx.origin: address A

AUTHOR: TIM HUANG



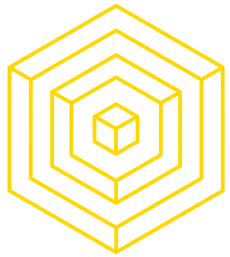
4.2 EXTERNAL CONTRACTS



EXTERNAL CONTRACTS

EXTERNAL CONTRACTS

```
contract InfoFeed {
    function info() returns (uint) {
        return 42;
    }
}
```

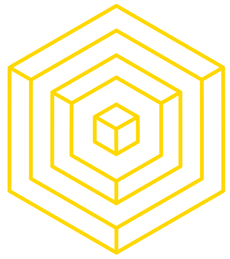


EXTERNAL CONTRACTS

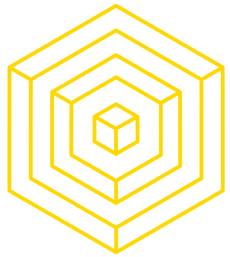
EXTERNAL CONTRACTS

```
contract InfoFeed {
    function info() returns (uint) {
        return 42;
    }
}
```

```
// Import the file if contracts are in separate files
import "./InfoFeed.sol";

contract Consumer {
    InfoFeed feed; // points to contract on blockchain

    // Set feed to new instance of contract
    function createNewFeed() {
        // new instance created, constructor call
        feed = new InfoFeed();
    }

    // Set feed to existing contract instance
    function setFeed(address addr) {
        feed = InfoFeed(addr);
    }

    function callFeed() {
        feed.info();
    }
}
```

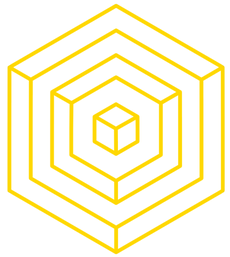
4.3 MODIFIERS



MODIFIERS

MODIFIERS

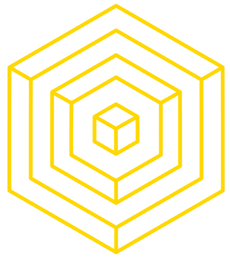
```
// Modifiers validate inputs to functions such as minimal balance or user auth;

// Transfers balances from one address to another
function transferBalance(address from, address to) public onlyOwner {
    balances[to] += balances[from];
    balances[from] = 0;
}
```



MODIFIERS

MODIFIERS

```
// '_' (underscore) often included as last line in body, and indicates
// the function being called should be placed there
modifier onlyOwner() {
    require(msg.sender == owner);
    _;
}

modifier onlyAfter(uint _time) {
    require (now >= _time);
    _;
}
```



Attendance

https://tinyurl.com/sp20-dev-decal-3

QUIZ

https://forms.gle/Ai75MhjsYpeZUykK8