Used to write smart contracts for the blockchain.

Note: Use contract name same as file name

Syntax:

First line contain the version of the solidity required to run the file:

* pragma solidity ^0.5.0

Initialize smart contract as:

* contract <contract name> { …. }

declaring variable:

<type> <access specifier> <name>

Solidity is statically typed language. Giving variable type is required.

Default access specifier is private. It only allows read/update the value from inside the smart contract.

To access variable outside the contract use public. If we want to return the variable value declare it public.

State variable: essentially writes information in the blockchain. Declared as class variable inside the contract{}.

Create constructor:

* constructor(val) public{

name = “Himanshu”

value = val

}

Create a data structure:

Struct Product{

Uint id; //struct property variable

..

}

Data types:

Uint: unsigned integer

String: string

Address: to store the address

Bool: Boolean

uint and int are aliases for uint256 and int256

It is possible to query the balance of an address using the property balance and to send Ether (in units of wei) to a payable address using the transfer function:

address **payable** x = address(0x123);

address myAddress = address(this);

**if** (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);

Mapping work like associative arrays, or hash tables. It is a key value pair used to get a value when a key is passed to it

Create a mapping: mappint(type => type) <mapping name>;

Mappint(unit => Product) public products; //get the product when an id (unit) is passed

Note: there is no function to get the number of pairs in a mapping.

Function types are notated as follows:

**function** (<parameter types>) {**internal**|**external**} [**pure**|**view**|**payable**] [**returns** (<**return** types>)]

Create a function to create new product which will

* Create a new product with a struct
* Add the struct to the mapping, and store it on the blockchain
* Trigger an event that lets someone know a product was creatd

function **createProduct**(**string** memory \_name, **uint** \_price) **public** {

// Require a valid name

require(bytes(\_name).length > 0);

// Require a valid price

require(\_price > 0);

// Increment product count

productCount ++;

// Create the product

products[productCount] = Product(productCount, \_name, \_price, msg.sender, false);

// Trigger an event

emit **ProductCreated**(productCount, \_name, \_price, msg.sender, false);

}

Create an event:

**event** **ProductCreated**(

**uint** id,

**string** name,

**uint** price,

address owner,

**bool** purchased

);

External subscribers can listen for this event to verify that a product was created on the blockchain.

Emit **ProductCreated()** is used to log out the value which can be used by other function/tests.

there are three types of memories:

1. Storage
2. Memory
3. Stack

“memory” keyword is used to create a copy of data in memory and not just used that is available in blockchain.

**Passing meta data to the function:**

Last arg in function call additional to function parameter is used to pass meta data to the function. Like

result = **await** marketplace.createProduct('iPhone X', web3.utils.toWei('1', 'Ether'), { **from**: seller })

Meta data contain a dictionary of key value pair.

From: used to tell who is calling the function

Value: used to pass the ether to the function

To access the meta data in the function use:

Msg.sender : get from meta data

Msg.value: get value

**Transfer ether to an account:**

Address(seller).transfer(msg.value)

**Make a function to transfer ether:** use “payable” to function

**function** **purchaseProduct**(uint \_id) **public** **payable** {

// Fetch the product

Product memory \_product = products[\_id];

// Fetch the owner

address payable \_seller = \_product.owner;

// Make sure the product has a valid id

require(\_product.id > 0 && \_product.id <= productCount);

// Require that there is enough Ether in the transaction

require(msg.value >= \_product.price);

// Require that the product has not been purchased already

require(!\_product.purchased);

// Require that the buyer is not the seller

require(\_seller != msg.sender);

// Transfer ownership to the buyer

\_product.owner = msg.sender;

// Mark as purchased

\_product.purchased = true;

// Update the product

products[\_id] = \_product;

// Pay the seller by sending them Ether

address(\_seller).transfer(msg.value);

// Trigger an event

emit ProductPurchased(productCount, \_product.name, \_product.price, msg.sender, true);

}

Add “payable” to the address to allow to payment to the address.

Like: address payable seller = product.owner

Operators:

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