ETHEREUM 2.0 MASTERY PROGRAM

Instructor: Raja Rizwan Saleem

MODULE-2

BLOCKCHAIN AND SMART CONTRACT BASICS

Raja Rizwan Saleem
Lead Blockchain Trainer

Class-11

Addresses

An address is a 20 bytes data type. It is specifically designed to hold account addresses in Ethereum, which are 160 bits or 20 bytes in size.

Address

1. Address can hold contract account addresses as well as Externally Owned Account addresses.
2. Address is a value type and it creates a new copy while being assigned to another variable.
3. Address has a ***balance*** property that returns balance available in an account (contract or individual) in wei

Address

1. Address type comes in two flavours, which are largely identical
    a. **address**: Holds a 20 byte value (size of an Ethereum address).
    b. **address payable**: Same as address, but with the additional members *transfer* and *send*.
2. *address payable* is an address you can send Ether to, while a plain *address* cannot be sent Ether

Address

✅ 1. Address Types

- `address`: Non-payable Ethereum address (20 bytes).
- `address payable`: Payable address, capable of sending and receiving Ether.

✅ 2. Address Declaration

- Declare an `address` or `address payable`.
- Use `msg.sender` to access the caller's address.

✅ 3. Address Functions

- `.balance`: Check the Ether balance of an address.
- `.transfer()`: Send Ether (2300 gas, reverts on failure).
- `.send()`: Send Ether (2300 gas, returns `false` on failure).
- `.call()`: Send Ether with arbitrary calls (forwards all gas, returns success status).

Address

✅ 4. Address Typecasting

- Convert `address` → `address payable` using `payable()`.
- Convert `address payable` → `address` directly.

✅ 5. Address Comparisons

- Compare two addresses using `==` or `!=`.
- Validate non-zero address using `address(0)`.

✅ 6. Address Usage in Payments

- Use `receive()` or `fallback()` to accept Ether.
- Use `.transfer()` or `.call()` to send Ether.

✅ 7. Special Address Keywords

- `msg.sender`: Caller's address.
- `msg.value`: Ether amount sent.
- `address(this)`: Current contract's address.

Address

1. Declaration and Initialization

```
address myAddress =

0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;

address payable myAddress2 =
0x4B0897B0513fdC7C541B6d9D7E929C4e5364D2dB;
```

Address functions

1. Address type provide following functions:
    a. send
    b. transfer

Address - send function

1. The send method is used to send Ether to a contract or to an individually owned account (EOA).
2. The send function provides 2,300 gas as a fixed limit (by default), which cannot be superseded.
3. The send function returns a boolean true/false as a return value.
4. In case failure, an exception is not returned; instead, **false** is returned from the function.

Address - send function

5. In case of failure, your funds will not be returns and it will beholded by contract of the function which you have called

Address - send function

5. In case of failure, your funds will not be returns and it will beholded by contract of the function which you have called

Address - send function

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MemoryAndStorage {

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

    function abc(address _from, address _to, uint256 _value) public {

        emit Transfer(_from, _to, _value );    }}
```

Address - send function

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract MemoryAndStorage {

    address payable public myAddress2 =
payable(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4);

    function sendFunds() public payable returns (bool) {
        require(msg.value >= 2 ether, "Insufficient funds to send 2 ether");
        bool isSent = myAddress2.send(2 ether);
        return isSent;  }}
```

Address - send function

initialization of `myAddress2`:

- The address should be cast to `payable` when initializing myAddress2.
- Added public visibility to allow `myAddress2` to be accessed externally if needed.

Handling Insufficient Funds:

- Added a require statement to check if the msg.value is at least 2 ether before sending the funds. This prevents the function from failing silently when there are insufficient funds.

General Style and Best Practices:

- The code is formatted for readability.
- payable is explicitly mentioned when initializing `myAddress2.`

Address - send function

```
address payable myAddress2 =
0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB;

function sendFunds() public payable returns(bool)
{
      bool isSent = myAddress2.send(msg.value);
      return isSent;    }
```

Address - send function = Example

```
pragma solidity ^0.5.0;

contract First {

        address payable myAddress = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;

                function sendFunds() public payable returns (bool) {
                    bool isFundsSent = myAddress.send (84 ether);
                    return isFundsSent;
                     }
}
```

Address - transfer function

1. The transfer method is similar to the send method.
2. It is responsible for sending Ether or wei to an address.
3. However, the difference here is that transfer raises an exception in the case of execution failure, instead of returning false , and all changes are reverted.
4. The transfer method is preferred over the send method as it raises an exception in the event of an error, meaning exceptions are bubbled up in the stack and halt execution.

Address - transfer function

5. In case of failure funds will not be deducted and it will stay in account which initiate the transaction

Address - transfer function

```
address payable myAddress2 =
0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB;

function sendFunds() public payable {
    myAddress2.transfer(2 ether);

}
```

Address - transfer function

```
address payable myAddress2 =
0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB;

function transferFunds() public
    payable {
    myAddress2.transfer(msg.value);  }
```

Address - transfer function = Example

```
pragma solidity ^0.5.0;

contract First {

	address payable myAddress =
0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;

			function transferFunds() public payable {
				myAddress.transfer ( 15 ether);

			}   }
```

Address - transfer function = Example

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract AddressPayableExample {

    address payable public recipient;

    // Function to set the recipient address as a payable address
    function setRecipient(address payable _recipient) public {    undefined gas
        recipient = _recipient;
    }

    // Function to send Ether to the recipient
    function sendEther() public payable {    infinite gas
        require(msg.value > 0, "You must send some Ether");
        recipient.transfer(msg.value); // Transfer Ether to the recipient
    }

    // Fallback function to accept Ether
    receive() external payable {}    undefined gas
}
```

Mappings

Mappings are similar to hash tables or dictionaries in other languages.

They help in storing key-value pairs and enable retrieving values based on the supplied key.

Mapping Definition

1. Mappings act as hash tables which consist of key types and corresponding value type pairs
2. Mappings are useful because they can store many _KeyTypes to _ValueTypes
3. Mappings do not have a length, nor do they have a concept of a key or a value being set
4. Mappings can only be used for state variables that act as storage reference types
5. When mappings are initialized every possible key exists in the mappings and are mapped to values whose byte-representations are all zeros

Mapping Definition

1. Mappings are one of the most used complex data types in Solidity.
2. Mappings are declared using the mapping keyword followed by data types for both key and value separated by the => notation.

Mapping Declaration

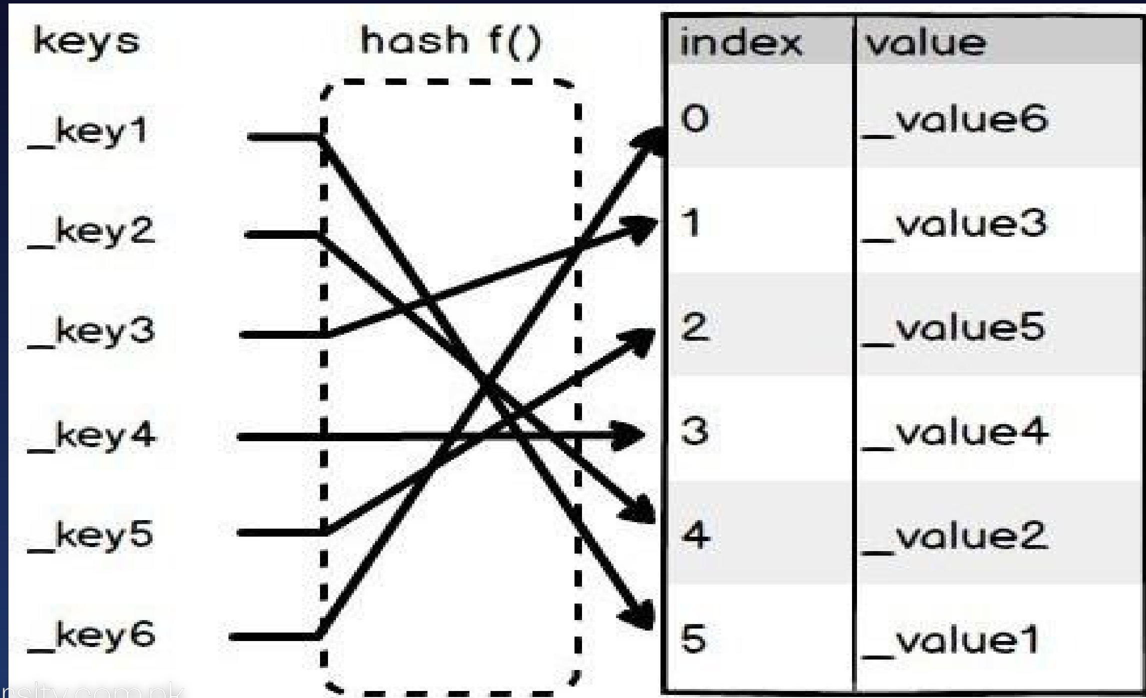
Mappings act as hash tables which consist of key types and corresponding value type pairs. They are defined like any other variable type in Solidity:

```
mapping(_KeyType => _ValueType) public mappingName
```

And

```
mapping(key => value) <access specifier> <name>;
```

Mapping Declaration



index	value
0	_value6
1	_value3
2	_value5
3	_value4
4	_value2
5	_value1

Mapping Declaration

```
mapping (int => string) public m1;
```

1. In above example int data type is used for storing keys and string data type is used for storing values.
2. You can use different data types for key and value.
3. ***enum*** and ***struct*** can be used as values but **not as key**.

Mapping Declaration

Few different combination of key value pair

```
mapping (uint => address) m2;
mapping (address => int) m3;
mapping (string => bool) m4;
mapping (string => gender) m5;
mapping (string => Student) m6;
```

Mapping Usage

1. To access any particular value in mapping, the associated key should be used along with the mapping name

```
mapping (int => string) names;
function update() public returns (string emory){
    names[1] = "Waseem";
    names[2] = "Shafeeq"; return
    names[2];}    //Shafeeq
```

Mapping Usage

```
mapping (string => uint) names;

function update() public returns (uint){
    names["Alishba"] = 1;
    names["Yasir"] = 2;
    names["Basit"] = 3;
    return names["Basit"];
```

Mapping Usage Example

```
pragma solidity ^0.5.0;
contract First {
     mapping (int => string) names;
    function upateValue() public {
        names [1]="Raja";
        names [2]="Rizwan";
        names [3]="Saleem";

    }
    function getValue() public view returns (string memory) {
        return names [2];

    } }
```

Mapping Usage Example

```
pragma solidity ^0.5.0;
contract First {

        mapping (int => string) names;

        function upateValue(int a, string memory b) public { names[a]=b;

        }
        function getValue(int a) public view returns (string memory) { return names
                [a];

        }
}
```

Mapping Rules

- **Key-Value Pairing**: Mappings store data as key-value pairs, where each key is unique and maps directly to a value.
- **Data Access**: Mappings are used to look up values based on their associated keys.
- **Default Values**: If a key does not exist in the mapping, accessing it will return the default value for the type (0 for uint, false for bool, etc.).
- **No Length or Iteration**: Mappings do not store keys or values in any particular order, and you cannot directly obtain a list of keys or iterate through a mapping.
- **No Deletion**: Mappings cannot be deleted entirely. However, individual key-value pairs can be reset by setting the value to the default state.
- **Gas Efficiency**: Mappings are generally more gas-efficient compared to arrays for key-value lookups because they do not involve iteration.

Mapping Rules

- **Nested Mappings**: Mappings can be nested within other mappings, allowing for complex data structures.
- **No Containment Checks**: You cannot directly check if a key exists in a mapping, but you can infer it by checking if the value is the default for that type.
- **Not Enumerable**: Mappings are not enumerable, meaning you cannot retrieve all the keys or values directly from the mapping.
- **Type Restrictions**: Mappings can use any built-in or user-defined type as the key type except for mapping, dynamic array, contract, enum, and struct types. The value can be of any type, including mappings and structs.
- **Storage Only**: Mappings can only be declared at the storage level and cannot be used in memory or as parameters in functions.

Nested Mapping Declaration

1. It is also possible to have nestedmapping, that is mapping consisting of mappings
2. Mapping can be used as value of another mapping so it will be mapped to key mapping

```
mapping (uint => mapping(address => string)) accountDetails;
```

Nested Mapping Example

```
mapping (string => mapping(uint => string)) stuCourses;
function update() publicreturns (string memory) {
stuCourses["PBI001"][1] = "AI";
stuCourses["PBI001"][2] = "Cloud";
stuCourses["PBI024"][1] = "IOT";
stuCourses["PBI024"][2] = "Blockchain";

return stuCourses["PBI024"][1]; }
}
```

Nested Mapping Example

```
pragma solidity ^0.5.0; contract First {

        mapping (string => mapping(int=> string)) stuCourses;

        function addCourse(string memory rollNo, int counter, string memory course) public {

                stuCourses [rollNo][counter]=course;

        }
        function findCourse(string memory rollNo, int counter) public view returns (string memory) {

                return stuCourses [rollNo][counter];

        }
}
```

Nested Mapping Example

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract NestedMapping {

    // Nested mapping: Maps an address to another mapping of address to uint
    mapping(address => mapping(address => uint)) public allowances;

    // Set allowance for a spender
    function setAllowance(address _spender, uint _amount) public {    22909 gas
        allowances[msg.sender][_spender] = _amount;
    }

    // Get allowance for a spender
    function getAllowance(address _owner, address _spender) public view returns (uint) {    infinite gas
        return allowances[_owner][_spender];
    }

    // Reset allowance for a spender
    function removeAllowance(address _spender) public {    5676 gas
        delete allowances[msg.sender][_spender];
    }
}
```

}

Nested Mapping Example

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract BasicMapping {
    mapping(address => uint) public balances;

    function setBalance(uint _amount) public {
        balances[msg.sender] = _amount;
    }

  function getBalance(address _address) public view returns (uint) {
        return balances[_address];
    }}
```

Basic Mapping Example

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract BasicMapping {
    mapping(address => uint) public balances;

    function setBalance(uint _amount) public {
        balances[msg.sender] = _amount;
    }

  function getBalance(address _address) public view returns (uint) {
        return balances[_address];
    }}
```

Mapping Example with Struct

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MappingWithStructs {
    struct User {
        string name;
        uint balance;
    }

    mapping(address => User) public users;

    function setUser(string memory _name, uint _balance) public {    infinite gas
        users[msg.sender] = User(_name, _balance);
    }

    function getUser(address _address) public view returns (string memory, uint) {    infinite gas
        User memory user = users[_address];
        return (user.name, user.balance);
    }
}
```

Nested Mapping

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract NestedMapping {
    mapping(address => mapping(uint => bool)) public accessControl;
    function setAccess(uint _id, bool _hasAccess) public {
        accessControl[msg.sender][_id] = _hasAccess;
    }

    function checkAccess(address _address, uint _id) public view returns (bool) {
        return accessControl[_address][_id];
    }
}
```

Mapping with Enumerations

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract MappingWithEnum {
    enum Status { Inactive, Active, Suspended }

    mapping(address => Status) public userStatus;

    function setUserStatus(Status _status) public {
        userStatus[msg.sender] = _status;
    }

    function getUserStatus(address _address) public view returns (Status) {
        return userStatus[_address];
```

Mapping to Store Multi Types

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract MultiTypeMapping {
    mapping(address => uint) public balances;
    mapping(address => bool) public registered;
    mapping(address => string) public names;
    function registerUser(string memory _name, uint _balance) public {
        names[msg.sender] = _name;
        balances[msg.sender] = _balance;
        registered[msg.sender] = true;          }
    function getUserDetails(address _address) public view returns (string memory, uint,

        return (names[_address], balances[_address], registered[_address]);
}}
```

Mapping with Arrays

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MappingWithArray {
    mapping(address => uint[]) public userScores;

    function addScore(uint _score) public {
        userScores[msg.sender].push(_score);
    }

    function getScores(address _address) public view returns (uint[] memory) {
        return userScores[_address];
    } }
```

Mapping for Ownership Management

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract OwnershipMapping {
	mapping(address => address) public ownerOf;

	function setOwner(address _item) public {
		ownerOf[_item] = msg.sender;
	}

	function getOwner(address _item) public view returns (address) {
		return ownerOf[_item];
	}}
```

Mapping with Removal (Resetting Values)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract RemovableMapping {

    mapping(address => uint) public balances;

    function setBalance(uint _amount) public {    22646 gas
        balances[msg.sender] = _amount;
    }

    function removeBalance() public {    5238 gas
        delete balances[msg.sender]; // Resets the value to 0
    }

    function getBalance(address _address) public view returns (uint) {    2851 gas
        return balances[_address];
    }
}
```

Mapping with Struct and Enum Combined

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract MappingWithStructAndEnum {

    // Enum representing different user roles
    enum Role { User, Admin, SuperAdmin }

    // Struct representing user details
    struct User {
        string name;
        uint balance;
        Role role;
    }

    // Mapping to store user information associated with an address
    mapping(address => User) public users;

    // Function to set user information
    function setUser(string memory _name, uint _balance, Role _role) public {    undefined gas
        users[msg.sender] = User(_name, _balance, _role);
    }

    // Function to retrieve user information
    function getUser(address _address) public view returns (string memory, uint, Role) {    infinite gas
        User memory user = users[_address];
        return (user.name, user.balance, user.role);
    }
}
```

THANK-YOU