MODULE-2 BLOCKCHAIN AND SMART CONTRACT BASICS

Class-8

Raja Rizwan Saleem
Lead Blockchain Trainer

Address

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. Declaration and Initialization

```
address myAddress =
0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;


address payable myAddress2 =
0x4B0897B0513fdC7C541B6d9D7E929C4e5364D2dB;
```

Address functions

1. Address type provide following five 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 The send function provides 2,300 gas as a fixed limit (by default), which cannot be superseded.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 be holded 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

```solidity
// 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 {
        recipient = _recipient;
    }
    // Function to send Ether to the recipient
    function sendEther() public payable {
        require(msg.value > 0, "You must send some Ether");
        recipient.transfer(msg.value); // Transfer Ether to the recipient address
    }
    // Fallback function to accept Ether
    receive() external payable {}
```

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.

Mappings

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

Mappings

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

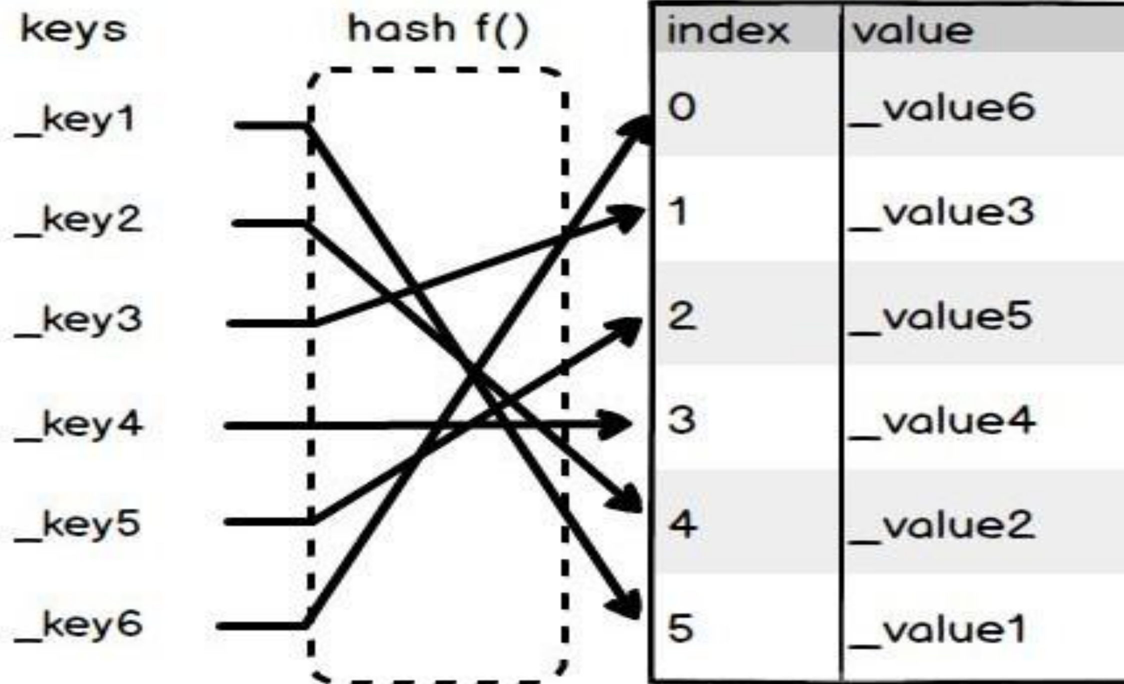
Mappings act as hash tables which consist of key types and corresponding value type pairs. They are defined like any other variable type in Solidity:

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

And

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

Mapping Declaration



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;   //enum in value
mapping (string => Student) m6; //struct in value
```

Thanks

End of Module-2 (Class-8)