MODULE-2 BLOCKCHAIN AND SMART CONTRACT BASICS

Class-9

Raja Rizwan Saleem
Lead Blockchain Trainer

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

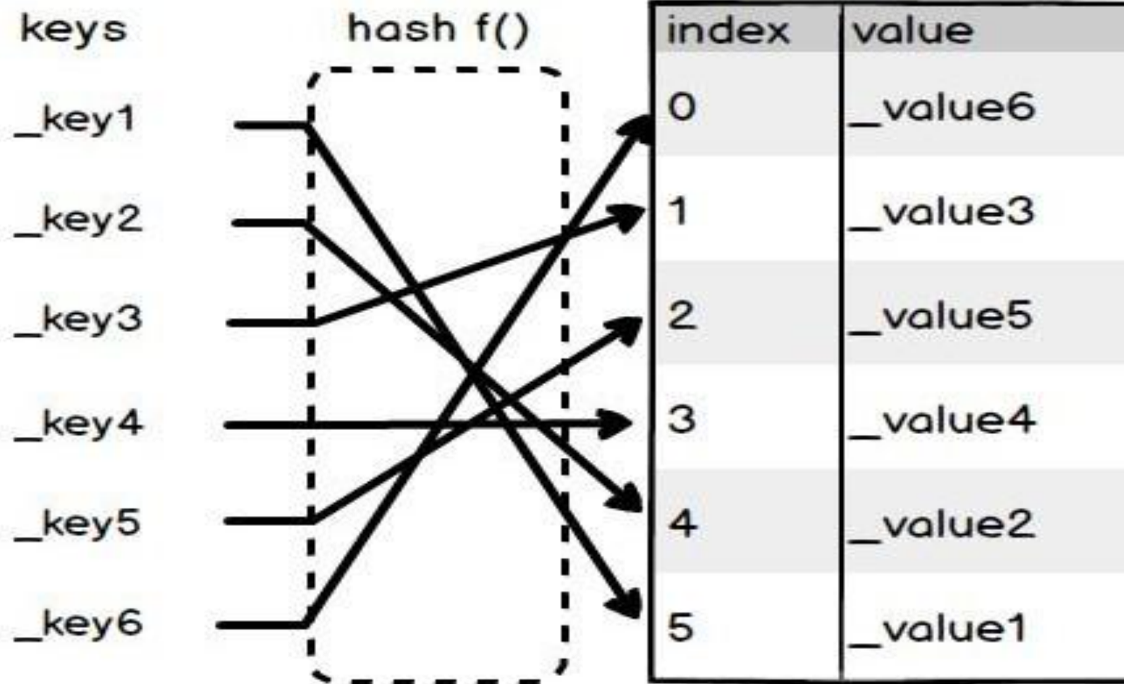
Mappings act as hash tables which consist of key types and corresponding value type pairs. They are defined like any other variable type in Solidity:

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

And

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

Mapping Declaration



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
```

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"];  // 3
}
```

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 nested mapping, 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() public returns (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 MappingExample {

mapping (int => mapping (address => string)) names;
function upateValue(int a, address b, string memory c) public {
    names[a][b] = c;
    // key1 & 2 = value
    // int, add = string
}
function getValue(int a, address b) public view returns (string memory) {
   return names [a][b];
}
}
```

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 {
        users[msg.sender] = User(_name, _balance);
    }
    function getUser(address _address) public view returns (string memory, uint) {
        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,
bool) {
        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 {
        balances[msg.sender] = _amount;
    }
    function removeBalance() public {
        delete balances[msg.sender]; // Resets the value to 0
    }
    function getBalance(address _address) public view returns (uint) {
        return balances[_address];
    }
}
```

Mapping with Struct and Enum Combined

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;
contract MappingWithStructAndEnum {
    enum Role { User, Admin, SuperAdmin }

    struct User {
        string name;
        uint balance;
        Role role;
    }
    mapping(address => User) public users;
    function setUser(string memory _name, uint _balance, Role _role) public {
        users[msg.sender] = User(_name, _balance, _role);
    }
    function getUser(address _address) public view returns (string memory, uint, Role) {
        User memory user = users[_address];
        return (user.name, user.balance, user.role);
    }}
```

Thanks

End of Module-2 (Class-9)