

1. Mapping

The screenshot shows a Solidity IDE with a contract named `MappingExample`. The contract code is as follows:

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

contract MappingExample {
    mapping(address => uint256) public valueMapping;

    function setValue(uint256 _value) public {
        valueMapping[msg.sender] = _value;
    }

    function getValue() public view returns (uint256) {
        return valueMapping[msg.sender];
    }
}
```

On the left, a web interface for the contract is shown. It displays the balance as 0 ETH. The `setValue` function is called with the value 156. The `getValue` function is called, returning 0. The `valueMapping` is shown as a mapping from address to uint256, with the value 156 set for the sender's address.

2. Error Handling

The screenshot shows a Solidity IDE with a contract named `ErrorHandler`. The contract code is as follows:

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

contract ErrorHandler {
    mapping(address => uint256) public valueMapping;

    function setValue(uint256 _value) public {
        require(_value != 0, "Value cannot be zero");
        valueMapping[msg.sender] = _value;
    }

    function getValue() public view returns (uint256) {
        return valueMapping[msg.sender];
    }
}
```

The screenshot shows a Solidity IDE with a contract named `ErrorHandler`. The contract code is as follows:

```
revert

The transaction has been reverted to the initial state.
Reason provided by the contract: "Value cannot be zero".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

call to ErrorHandler.getValue

CALL [call] from: 0x5838Da6a701c568545dCfc803Fc8875f56beddC4
to: ErrorHandler.getValue() data: 0x209...65255
call to ErrorHandler.getValue errored: Error occurred: revert.

revert

The transaction has been reverted to the initial state.
Reason provided by the contract: "No value set for sender".
If the transaction failed for not having enough gas, try increasing the gas limit gently.
```

On the left, a web interface for the contract is shown. It displays the balance as 0 ETH. The `setValue` function is called with the value `uint256 _value`. The `getValue` function is called, returning `getValue - call`. The `valueMapping` is shown as a mapping from address to uint256, with the value `address` set for the sender's address.

3. Function Modifier

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

contract FunctionModifier {
    mapping(address => uint256) public valueMapping;

    modifier nonZeroValue(uint256 _value) {
        require(_value != 0, "Value cannot be zero");
        _;
    }

    modifier valueHasBeenSet() {
        require(valueMapping[msg.sender] != 0, "No value set for sender");
        _;
    }

    function setValue(uint256 _value) public nonZeroValue(_value) { 22650 gas
        valueMapping[msg.sender] = _value;
    }

    function getValue() public view valueHasBeenSet returns (uint256) { 4704 gas
        return valueMapping[msg.sender];
    }
}

Transaction to FunctionModifier.setValue errored: Error occurred: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "Value cannot be zero".
If the transaction failed for not having enough gas, try increasing the gas limit gently.

call to FunctionModifier.getValue

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: FunctionModifier.getValue() data: 0x209...65255
call to FunctionModifier.getValue errored: Error occurred: revert.

revert
The transaction has been reverted to the initial state.
Reason provided by the contract: "No value set for sender".
If the transaction failed for not having enough gas, try increasing the gas limit gently.
```

FUNCTION MODIFIER AT 0XA6E

Balance: 0 ETH

setValue 0

getValue

valueMapping address

Low level interactions

CALLDATA

Transact

4. Ownable

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

contract Ownable {
    address public owner;

    constructor() { 246725 gas 222200 gas
        owner = msg.sender;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "not owner");
        _;
    }

    function setOwner(address newOwner) external onlyOwner {
        require(newOwner != address(0), "invalid address");
        owner = newOwner;
    }

    function onlyOwnerCanCallThisFunc() external onlyOwner {
        // code
    }

    function anyOneCanCall() external { 165 gas
        // code
    }
}

OWNABLE AT 0XD4F...2CBEE
```

Balance: 0 ETH

anyOneCanCall

onlyOwnerCan...

setOwner 0x5B38Da6a701c568545dC

owner

0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

5. Constructor

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

contract MyContract{
    uint public myNumber;

    constructor(){
        myNumber=42;
    }
}
```

myNumber - call

myNumber

0: uint256: 42

Reflection Questions:

- **Function Modifiers & Ownable**

When should you use a modifier like `onlyOwner` instead of inline checks, and what risks arise if ownership isn't managed properly?

You should use a modifier like `onlyOwner` instead of inline checks when you need to check access-control in multiple functions. If ownership isn't managed properly, the wrong owner or address could gain access to the funds and data of other addresses. Improper management of ownership could also cause anyone to call sensitive functions that are not supposed to be accessed.

- **Error Handling**

How do you choose between `require`, `revert`, and `assert`, and why might custom errors be better than error strings?

In error handling, `require` is used for validating user inputs, external calls, and normal preconditions. `Revert` is used for early exits, usually with custom errors for simpler logic. `Assert` is used for checking internal invariants that should never fail. Custom errors might be better than error strings because they require lesser gas fees.

- **Constants & Variables**

When should a value be constant, immutable, or mutable, and how does that choice affect gas cost and flexibility?

A value should be constant when it is known before compiling and it will never change in the future. A value should be immutable when it is set once in the constructor and never changes afterward. A value should be mutable when it must be changeable after deployment. A constant value has the cheapest gas cost but it

has lesser flexibility. An immutable is almost as cheap but has more flexibility. Lastly, a mutable value is the most expensive but it provides flexibility.