

1. Mapping

```
// 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];
    }
}
```

2. Error Handling

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

contract ErrorHandling{
    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) {
        require(msg.sender != 0, "No value set for sender");
        return valueMapping[msg.sender];
    }
}
```

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 ErrorHandling.getValue

call [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beDDc4 to: ErrorHandling.getValue() data: 0x209...65255
call to ErrorHandling.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.

3. Function Modifier

The screenshot shows the Truffle UI interface. On the left, the Solidity code for a `FunctionModifier` contract is displayed. It includes a mapping of addresses to uint256 values, a `nonZeroValue` modifier, a `valueHasBeenCalled` modifier, a `setValue` function, and a `getValue` function. On the right, the transaction results for each function are shown.

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

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

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

FUNCTIONMODIFIER AT 0xae1...
Balance: 0 ETH
Low level interactions
CALLDATA
Transact

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: 0x5838Da6a701c568545dFcB03FcB875f56beDDC4 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.

4. Ownable

The screenshot shows the Truffle UI interface. On the left, the Solidity code for an `Ownable` contract is displayed. It includes a constructor that sets the owner to the msg.sender, a `onlyOwner` modifier, a `setOwner` function, and two external functions: `anyOneCanCall` and `onlyOwnerCanCallThisFunc`. On the right, the transaction results for each function are shown.

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

contract Ownable {
    address public owner;

    constructor() {
        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 {
        // code
    }
}
```

OWNABLE AT 0xD4F...2CBEE ()
Balance: 0 ETH
Low level interactions
CALLDATA
Transact

anyOneCanCall

onlyOwnerCan...
setOwner 0x5838Da6a701c568545dC
owner

0: address: 0x5838Da6a701c568545
dFcB03FcB875f56beDDC4

5. Constructor

The screenshot shows a Solidity code editor with the following code:

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

contract MyContract{
    uint public myNumber;

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

A call to the `myNumber` function is shown in the interface, returning the value `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.