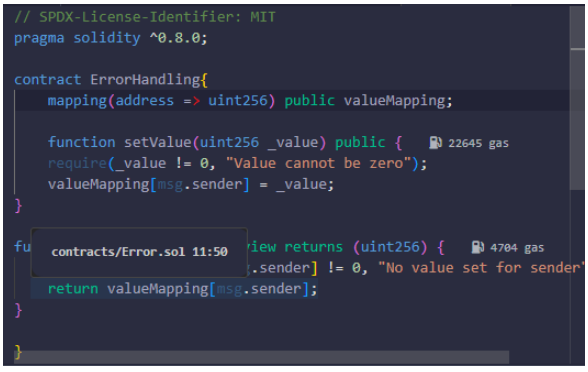
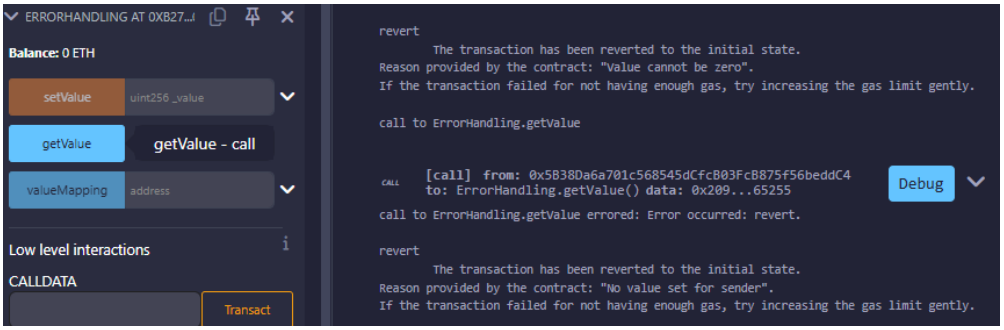
Dungo, Adrian Jeferson T. NW – 301 11/25/2025

1. Mapping

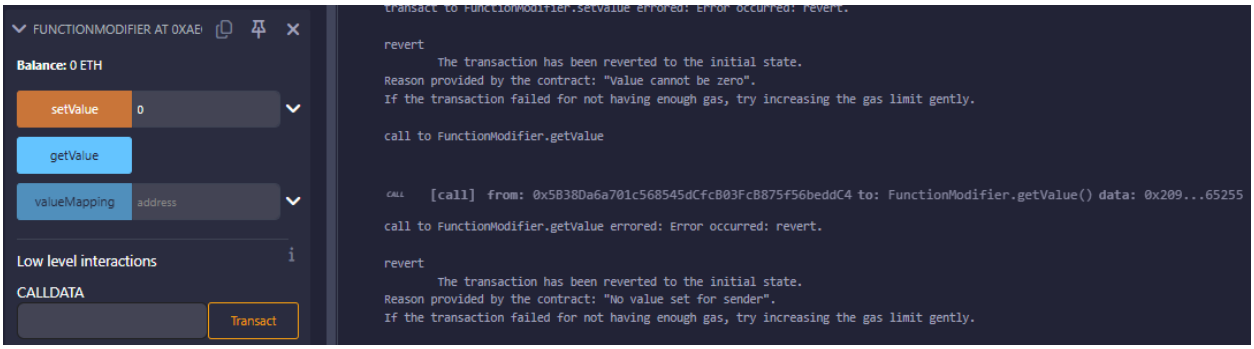
2. Error Handling



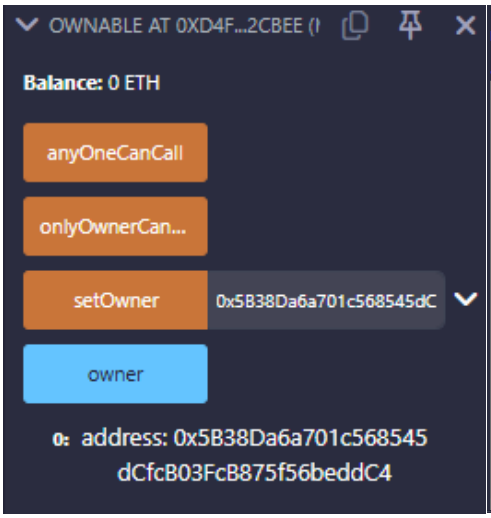


3. Function Modifier

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

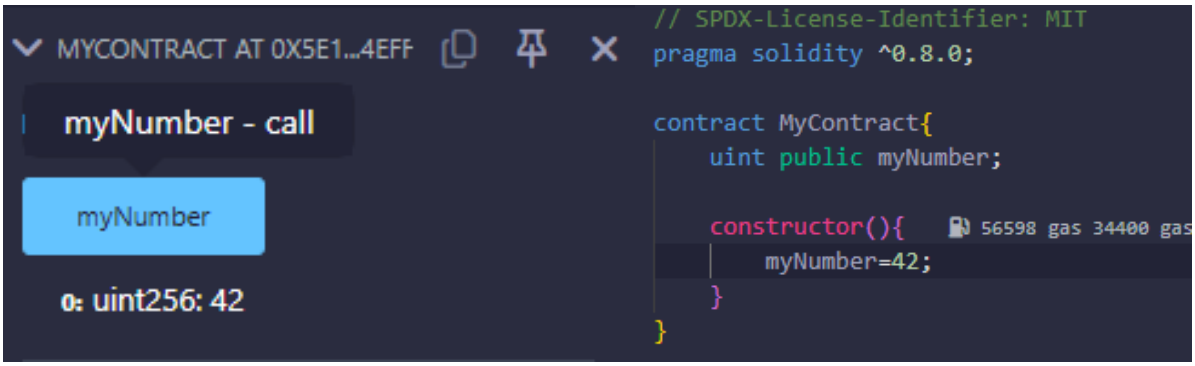


4. Ownable




```
// 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
    }
```

5. Constructor



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.