**Blockchain Study Notes Day 14:**

**Module 2 - Solidity Basics**  
**Chapter 10 - Errors in Solidity**

**Introduction to Errors**

Errors in Solidity help manage exceptional situations and provide a mechanism to handle invalid operations within smart contracts. By using error-handling techniques, developers can ensure the reliability and security of their contracts.

**1. Types of Errors in Solidity**

**1.1. Assert**

* **Purpose**:
  + Used to test for conditions that should never occur.
  + Typically used to check for internal errors and invariants.
* **Behavior**:
  + Reverts the transaction and consumes all gas.
* **Syntax**:

assert(condition);

* **Example**:

function checkInvariant(uint a, uint b) public pure {

assert(a + b > a);

}

**1.2. Require**

* **Purpose**:
  + Validates inputs and conditions before executing a function.
  + Commonly used for input validation, access control, or preconditions.
* **Behavior**:
  + Reverts the transaction but refunds unused gas.
* **Syntax**:

require(condition, "Error message");

* **Example**:

function deposit(uint amount) public {

require(amount > 0, "Amount must be greater than zero.");

// Deposit logic here

}

**1.3. Revert**

* **Purpose**:
  + Similar to require but more flexible for complex error handling.
  + Allows specifying custom error messages.
* **Syntax**:

revert("Error message");

* **Example**:

function withdraw(uint amount) public {

if (amount > balance) {

revert("Insufficient balance.");

}

// Withdrawal logic here

}

**1.4. Custom Errors (Solidity v0.8.4 and later)**

* **Purpose**:
  + More gas-efficient way to handle errors with custom names.
  + Useful for providing descriptive error types.
* **Syntax**:

error CustomError(string message);

* **Example**:

error InsufficientBalance(uint requested, uint available);

function withdraw(uint amount) public {

if (amount > balance) {

revert InsufficientBalance(amount, balance);

}

}

**2. Example Program Demonstrating Error Handling (Using Munawar)**

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract MunawarErrors {

uint public balance;

// Function to deposit funds

function deposit(uint amount) public {

require(amount > 0, "Deposit amount must be greater than zero.");

balance += amount;

}

// Function to withdraw funds

function withdraw(uint amount) public {

if (amount > balance) {

revert("Insufficient balance.");

}

balance -= amount;

}

// Function to demonstrate assert

function checkBalanceInvariant() public view {

assert(balance >= 0);

}

// Function using custom error

error UnauthorizedAccess(address caller);

function restrictedFunction() public view {

if (msg.sender != address(0x123)) {

revert UnauthorizedAccess(msg.sender);

}

}

}

**3. Best Practices for Error Handling**

* **Use assert for Internal Checks**:
  + Apply assert for conditions that should never fail during contract execution.
* **Use require for Input Validation**:
  + Validate user inputs and access rights with require.
* **Use Custom Errors for Gas Optimization**:
  + Prefer custom errors for better gas efficiency when throwing specific errors.
* **Provide Descriptive Error Messages**:
  + Always include clear messages to help users and developers understand the issue.

**Home Task**

1. **Enhance the Example Program**:
   * Add a function transfer with error handling for invalid transfers.
2. **Write a New Contract**:
   * Implement a contract that tracks inventory, using require and revert for inventory validation.
3. **Research**:
   * Explore real-world smart contracts on Ethereum that utilize error-handling techniques.

**Conclusion**

Error handling is a critical aspect of Solidity development. By understanding and effectively using assert, require, revert, and custom errors, developers can build robust and secure smart contracts that handle exceptional situations gracefully.

Day 14 Notes

***Prepared by Munawar Johar***