Learning outcome 2: Apply Solidity Basics

Prepare by tr Nicholas JT

Preparation of environment

Description of key terms

1. Solidity

- **Description**: Solidity is a high-level programming language designed specifically for writing smart contracts that run on the Ethereum blockchain. Created to be statically typed, Solidity has syntax and structure influenced by C++, Python, and JavaScript. Solidity allows developers to encode business logic, manage data on the blockchain, and interact with other contracts.
- **Purpose**: Solidity makes it easy to write contracts that control digital assets and are enforceable by code rather than by an external authority.

2. Syntax

- **Description** : Solidity syntax defines how contracts, functions, and statements are written. Solidity uses braces {} for blocks of code, like JavaScript, and requires a semicolon(;)at the end of each statement.
- Example:

```
 // Simple variable declaration in Solidity
uint x = 10;    // Defines an unsigned integer variable
string public greeting = "Hello, Ethereum!"; // A public string       variable
```

Key Concepts

- **Pragma Directive**: At the start of each Solidity file, the pragma specifies the Solidity version, ensuring the contract compiles correctly.

```
pragma solidity ^0.8.0;
```

3. Data Types

- Description: Solidity includes several basic data types for handling information. Knowing these types helps optimize memory use and manage the contract’s behavior effectively.
- Types:
  - **uint (Unsigned Integer):** Non-negative integers. Example: uint8 (0 to 255), uint256 (0 to very large number).
  - **int (Signed Integer):** Includes negative and positive values.
  - **bool (Boolean):** Stores true or false.
  - **address**: Stores Ethereum wallet or contract addresses. Addresses are crucial in Solidity because they are used to store contract and user locations on Ethereum.
  - **string**: For UTF-8 encoded text.
  - **bytes**: Fixed-length (like bytes32) or variable-length arrays of raw data.

Examples

```
bool isActive = true;
address contractOwner = 0x1234567890123456789012345678901234567890;
```

4. Variables

- **Description**: Variables store data within a contract. Solidity distinguishes between different types of variables based on where they are declared and stored:
  - **State Variables**: Declared outside of functions and permanently stored on the blockchain.
  - **Local Variables**: Declared within functions and exist temporarily during the function’s execution.

Example

```
// State variable, stored permanently on the blockchain
uint public count = 0;

function increment() public {
    // Local variable, temporary during function execution
    uint temp = count + 1;
    count = temp;
}
```

5. Identifiers

- **Description**: Identifiers are names used to define variables, functions, contracts, and parameters. Solidity identifiers must be descriptive, unique, and cannot use reserved keywords (like function, uint, etc.).
- **Naming Rules:**
  - ✔ Cannot begin with a number.
  - ✔ Should use camelCase or underscore separation.

    Example:

```
uint public totalSupply;   // `totalSupply` is an identifier for a state variable
```

6. Arrays

Description: Arrays in Solidity store collections of elements of the same data type. Arrays can be either:

- **Fixed-size**: Length is specified at declaration.
- **Dynamic**: Length can grow or shrink during execution.

Example:

```
uint[] public numbers;       // Dynamic array of unsigned integers
uint[5] public fixedNumbers; // Fixed array with a length of 5

// Adding an element to the dynamic array
function addNumber(uint num) public {
    numbers.push(num);
}
```

7. Struct

Description: Structs are custom data types that group different types of data into a single entity, useful for organizing related data in complex smart contracts.

Example:

Example

```
// Define a struct named Student
struct Student {
    string name;
    uint age;
}

// Create an instance of Student
Student public student = Student("Alice", 20);

// Creating a list of students
Student[] public students;

// Adding a student to the list
function addStudent(string memory _name, uint _age) public {
    students.push(Student(_name, _age));
}
```

8. Functions

- **Description**: Functions are blocks of code in Solidity that perform specific actions.
- Functions can:
  - Accept parameters.
  - Return values.
  - Change state variables (if they are not marked as view or pure).

Example:

```
// Function that adds two numbers and returns the result
function add(uint a, uint b) public pure returns (uint) {
    return a + b;
}
```

Example

```
// Function that adds two numbers and returns the result
function add(uint a, uint b) public pure returns (uint) {
    return a + b;
}
```

9. Control Structures

- **Description**: Solidity supports standard control structures such as:
  - **if-else**: Conditional logic.
  - **for loop**: Repeats code a specific number of times.
  - **while loop**: Repeats code while a condition is true.

If else example

```
function findEven(uint num) public pure returns (bool) {
    if (num % 2 == 0) {
        return true;
    } else {
        return false;
    }
}
```

While loop example

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

contract WhileLoopExample {
    uint public count;

    // Function to increment count until it reaches a specified limit
    function incrementUntil(uint limit) public {
        // Using a while loop to increment count
        while (count < limit) {
            count++;
        }
    }
}
```

Explanation

1. **count Variable:** A state variable count is used to store the current count value.
2. **incrementUntil Function**: This function accepts an integer limit and increments count until it reaches the specified limit.
   - While Loop: The while loop continues to increment count as long as the condition count < limit is true. Once count reaches the limit, the loop stops.

Usage and Gas Considerations

To use this contract:

- Call incrementUntil(limit) with a desired limit value. The function will run the loop and increment count until it meets the specified limit.

Note: Be cautious with limit values, as higher limits lead to more iterations and higher gas costs. For large iterations, consider alternatives like updating the state in multiple smaller transactions to avoid hitting the gas limit.

For loop example

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

contract ForLoopExample {
    uint public sum;

    // Function to calculate the sum of first 'n' natural numbers
    function calculateSum(uint n) public {
        sum = 0;  // Reset sum to 0 before calculation

        // For loop to add numbers from 1 to n
        for (uint i = 1; i <= n; i++) {
            sum += i;
        }
    }
}
```

Explanation

1. **sum Variable:** A state variable sum is used to store the result of the summation.
2. **calculateSum Function**: This function accepts a parameter n, which is the upper limit of the summation.
   - For Loop: The for loop iterates from 1 to n, adding each number to sum. Once the loop completes, sum will contain the sum of all numbers from 1 to n.

- **Usage and Gas Considerations**
  - ✔ Call **calculateSum(n)** with a value for n (e.g., 10) to calculate the sum of numbers from 1 to n.
  - ✔ **Gas Costs**: Each iteration in a for loop consumes gas, so large values for n can result in high gas costs. Use small values for n to keep costs manageable and avoid exceeding block gas limits.
  - ✔ This example demonstrates a basic use of for loops in Solidity while being mindful of potential gas limitations.

10. State Variables

- **Description**: State variables are variables that store contract data and are permanently recorded on the blockchain.
- Example:

```
contract Counter {
    uint public count = 0; // State variable

    function increment() public {
        count += 1;
    }
}
```

11. Modifiers (Conditions)

- **Description**: Modifiers are used to add constraints or conditions to functions. For example, a modifier can ensure that only the contract owner can perform a particular action.

Example

```
address public owner;

// Modifier to check if caller is the owner
modifier onlyOwner() {
    require(msg.sender == owner, "Not authorized");
    _; // The underscore represents the rest of the function code
}
// Function that can only be called by the owner
function setOwner(address newOwner) public onlyOwner {
    owner = newOwner;
}
```

12. Smart Contract

- **Description**: A smart contract is a self-executing piece of code with the rules directly written into it. Once deployed, smart contracts operate autonomously without the need for intermediaries.

Example

```
contract SimpleStorage {
    uint data;

    function setData(uint _data) public {
        data = _data;
    }

    function getData() public view returns (uint) {
        return data;
    }
}
```

13. Visibility and Access Control

- **Description**: Visibility defines where a function or variable can be accessed. In Solidity, visibility options include:
  - **public**: Accessible by anyone.
  - **private**: Accessible only within the contract.
  - **internal**: Accessible within the contract and derived contracts.
  - **external**: Accessible only from outside the contract.

Example

```
uint private secretData = 42; // Only accessible inside this contract

function getSecretData() public view returns (uint) {
    return secretData;
}
```

14. Ethereum

- **Description**: Ethereum is a decentralized platform that allows for the deployment and execution of smart contracts. It uses the cryptocurrency **Ether (ETH)** for transactions and incentivizing participants in the network.
- **Purpose**: Ethereum powers DApps (Decentralized Applications) that are trustless, secure, and operate without a central authority.

15. Ethereum Virtual Machine (EVM)

- **Description**: The EVM is a global computer that executes all smart contracts on Ethereum. It allows Solidity code to be compiled into bytecode and executed on every Ethereum node.

- **Example**: Every Solidity smart contract is deployed to the EVM, where it is run and maintained by the nodes of the Ethereum network.

Set up solidity environment

To set up a Solidity development environment, follow these steps to install essential tools and components, including code editors, the Node.js package manager, the Solidity compiler, and Ethereum development frameworks like Truffle and Hardhat.

Step 1: Install a Code Editor

- ✔ For Solidity development, two popular code editors are **Remix** and **Visual Studio Code (VS Code)**.
- ✔ **Option A: Remix IDE (Web-Based Editor)**
- ✔ **Description**: Remix is an online IDE specifically designed for Solidity and Ethereum development. It provides built-in tools for writing, compiling, testing, and deploying smart contracts.
- ✔ **Access**: Visit Remix in your web browser.
- ✔ **Features**: Remix offers syntax highlighting, integrated debugging, contract deployment, and plugin support.

Option B: Visual Studio Code (VS Code)

- **Description**: VS Code is a versatile, widely-used code editor that supports multiple languages, including Solidity.
- **Download**: Go to the Visual Studio Code website and download the appropriate version for your operating system.

Set Up Solidity in VS Code:

1. Open VS Code and go to the **Extensions** tab (left sidebar).
2. Search for the **Solidity** extension and install it. This extension provides syntax highlighting, auto-completion, and Solidity linting tools.

Step 2: Install Node.js and npm

- Node.js and npm are required for running development tools like Truffle and Hardhat. **Node.js** is a JavaScript runtime, and **npm** (Node Package Manager) manages JavaScript libraries and tools for your project.
  1. **Download and Install Node.js**: Visit the Node.js website and download the latest **LTS (Long-Term Support)** version for your operating system.
  2. **Verify Installation**: After installation, open a terminal or command prompt and type the following commands to check that Node.js and npm were installed correctly:
     - node -v
     - npm -v

Step 3: Install the Solidity Compiler (solc)

The Solidity compiler (solc) is essential for compiling Solidity code into EVM (Ethereum Virtual Machine) bytecode.

- You can install the Solidity compiler globally using npm:
  - ✔ npm install -g solc
- Verify installation: check if solc is installed correctly by running:
  - ✔ Solc –version

Alternatively, you can let tools like **Truffle** and **Hardhat** handle the Solidity compilation for you, as they come with integrated Solidity compilation support.

Step 4: Install Ethereum Development Tools (Truffle and Hardhat)

- Ethereum development frameworks like **Truffle** and **Hardhat** provide essential tools for compiling, testing, and deploying smart contracts.
- **Option A: Installing Hardhat:**
  1. **Create a Project Folder**:
     ```
     mkdir MySolidityProject
     cd MySolidityProject
     ```
  2. **Initialize Hardhat**:

     Install Hardhat as a local dependency:
     ```
     npm install --save-dev hardhat
     ```

- Initialize your Hardhat project:

  ```
  npx hardhat
  ```

  Follow the prompts to create a sample project. Hardhat will set up a basic project structure with example contracts, tests, and configuration files.

Project Structure:

- A typical Hardhat project structure will look like this :

```
MySolidityProject/
├── contracts/        // Solidity smart contracts
├── scripts/          // Deployment scripts
├── test/             // Test files
└── hardhat.config.js    // Hardhat configuration file
```

Option B: Installing Truffle

1. **Install Truffle Globally**:

```
npm install -g truffle
```

2. **Initialize Truffle Project**:

   Create a new folder and initialize Truffle:

```
mkdir MySolidityProject
cd MySolidityProject
truffle init
```

- This will create a basic Truffle project structure:

```
MySolidityProject/
├── contracts/          // Solidity smart contracts
├── migrations/         // Deployment scripts
├── test/               // Test files
└── truffle-config.js   // Truffle configuration file
```

- Compiling Contracts in Truffle:
  - Use the truffle compile command to compile your contracts:

```
truffle compile
```