CSE446 Lab: Solidity Part - 02

Objectives:

- To understand the solidity programming language and smart contracts.

Submission:

- Students should finish the checkpoints during lab class.

Description:

In this lab, you will learn some additional topics of solidity programming language that were not covered in the last lab class. You will write your own smart contract for the ethereum using it. You will learn how to facilitate writing smart contracts using concepts like reference-typed data, error handling, a detailed discussion about different function keywords, inheritance, and solving different problems using Remix IDE.

Solidity is a statically typed object-oriented programming language which facilitates creating smart contracts for various blockchain platforms and is considered as the de facto industry standard for developing smart contracts. It is designed to run on an Ethereum Virtual Machine(EVM) and is highly influenced by C++, Python, and Javascript. However, if you know Java or Javascript you will find similarities in syntax.

Though solidity has similarities in syntax with different languages, it is too computation sensitive and multiple features make it different from others such as uint or int with 8 to 256 bit, mapping, events, string, error comparison, function modifiers, handling techniques, etc. In this lab, we will explore further what we covered in the last lab class.

Prerequisites:

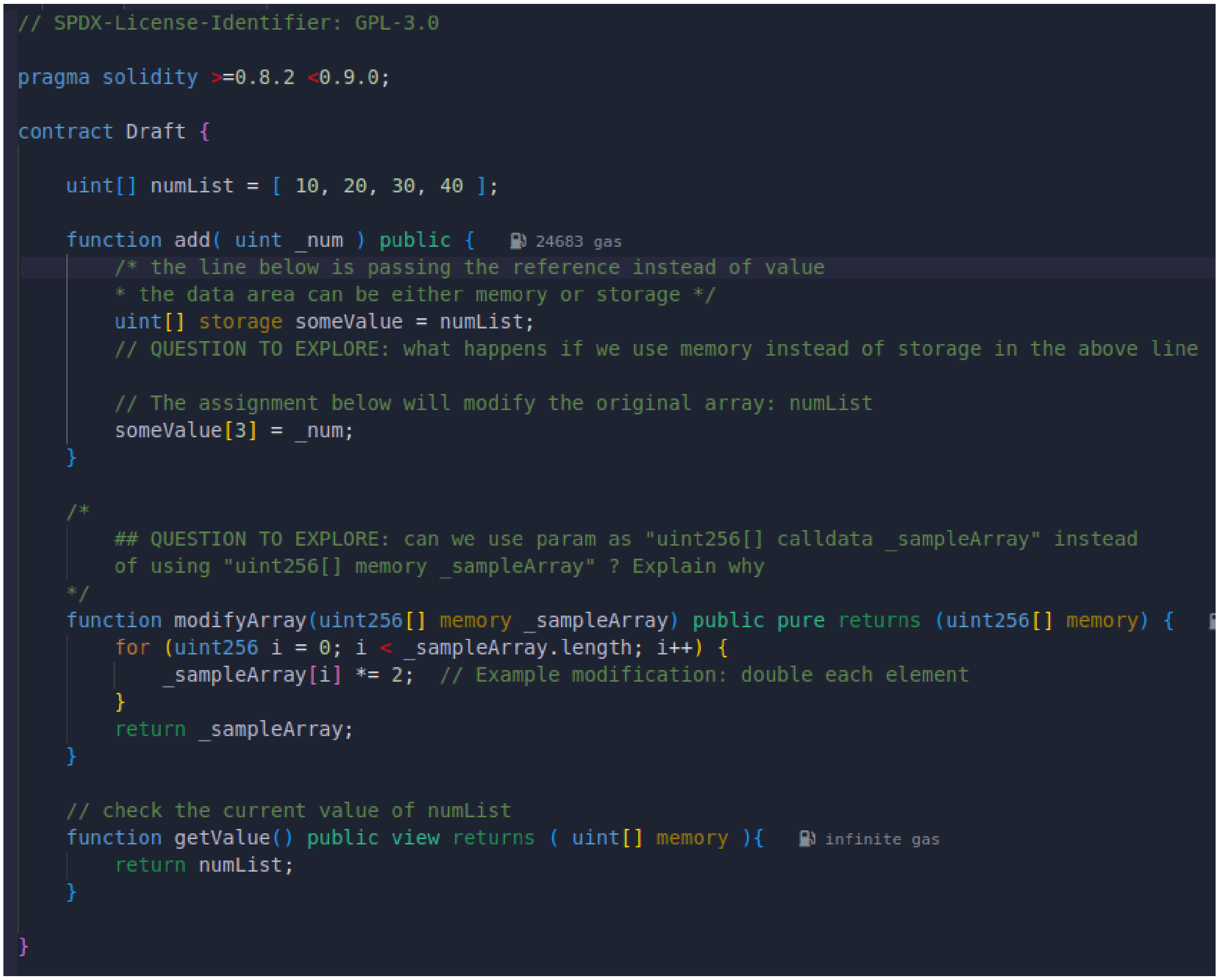
- Good to know, the basic syntax of solidity(Lab part 1)
- Remix IDE
- Web Browser. Recommended - Google Chrome or Firefox

01 Reference Type & Data Area

Like other programming languages, solidity has different types of reference-typed data such as structs, arrays, and mappings. These data types often need to be handled carefully as they **pass the references instead of the value**(pass by reference). Reference typed data requires a specified location where the type is stored is called **data area**(We already saw it in lab part 1). These three data areas, such as ***memory***, ***storage,*** *and* ***calldata*** need to be mentioned explicitly with the type declaration. The storage consumes the highest gas, and calldata consumes the lowest among these areas. We already learned about memory and storage. However, one of the core differences between the use-case of memory and calldata is used for non-persistent and non-modifiable function arguments.

Question to explore at home: We need to explicitly mention the data area for reference typed data. But if we use these reference-typed data as state variables, we don't need to mention them explicitly. Why?
Please go through this documentation to know more about the data area.

Example of reference type and data area:



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

contract Draft {

    uint[] numList = [ 10, 20, 30, 40 ];

    function add( uint _num ) public {    24683 gas
        /* the line below is passing the reference instead of value
        * the data area can be either memory or storage */
        uint[] storage someValue = numList;
        // QUESTION TO EXPLORE: what happens if we use memory instead of storage in the above line

        // The assignment below will modify the original array: numList
        someValue[3] = _num;
    }

    /*
        ## QUESTION TO EXPLORE: can we use param as "uint256[] calldata _sampleArray" instead
        of using "uint256[] memory _sampleArray" ? Explain why
    */
    function modifyArray(uint256[] memory _sampleArray) public pure returns (uint256[] memory) {
        for (uint256 i = 0; i < _sampleArray.length; i++) {
            _sampleArray[i] *= 2;  // Example modification: double each element
        }
        return _sampleArray;
    }

    // check the current value of numList
    function getValue() public view returns ( uint[] memory ){    infinite gas
        return numList;
    }

}
```

Figure - 1

The example above, within the **add** function, we updated **someValue,** however because of the pass-by reference type, **numList** has been modified. The **modifyArray** function receives a reference type _sampleArray of ***memory*** data area. Because its lifespan is limited to function calls, and we also modified it within the function.

02 Error Handling

In solidity, the mechanism is different from other programming languages. There are multiple mechanisms to handle solidity code; however, we will focus only on require, revert, and assert.

A. Require

Require is basically used to validate conditions in smart contracts. For validating inputs, external contract response, using require() is suitable. The syntax of using require is:

i. require(boolean)

ii. require(boolean, String)

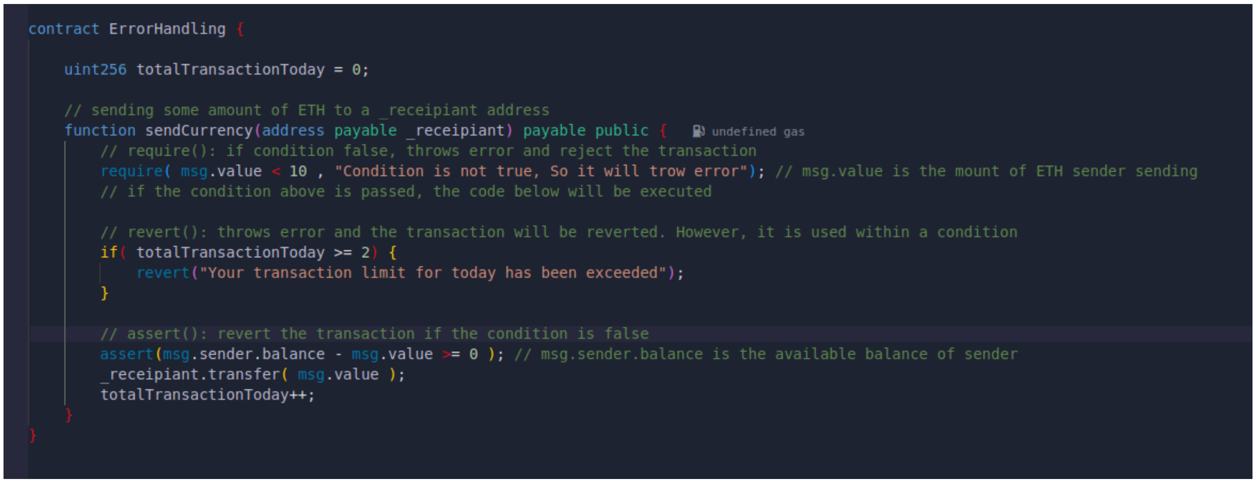
iii. require(boolean, Custom Error)

B. Revert

Conceptually, both revert() and require() do similar things, which is handling errors by reverting the changes. However, in cases where the condition is complex, inside nested code, using revert() over require() is preferred. Syntax: **revert("String").**

C. Assert

In solidity, assert() is a good method to find internal and logical errors in contracts. It is useful to validate states of function after changes, guarding conditions that should never occur.



```
contract ErrorHandling {

    uint256 totalTransactionToday = 0;

    // sending some amount of ETH to a _receipiant address
    function sendCurrency(address payable _receipiant) payable public {   undefined gas
        // require(): if condition false, throws error and reject the transaction
        require( msg.value < 10 , "Condition is not true, So it will trow error"); // msg.value is the mount of ETH sender sending
        // if the condition above is passed, the code below will be executed

        // revert(): throws error and the transaction will be reverted. However, it is used within a condition
        if( totalTransactionToday >= 2) {
            revert("Your transaction limit for today has been exceeded");
        }

        // assert(): revert the transaction if the condition is false
        assert(msg.sender.balance - msg.value >= 0 ); // msg.sender.balance is the available balance of sender
        _receipiant.transfer( msg.value );
        totalTransactionToday++;
    }
}
```

Figure 5: require, revert, assert

Checkpoint 01:

Create a contract called "IssueTracker". This contract holds different issues experienced by users. You have to create a **mapping** called "issueList" which will hold the lists of issues called "*Issue*". Add the visibility of "*issueList*" as **public**, so that it can be accessed from outside the contract directly. The "*issue*" object will hold, *issueId, description, and status.* The "*issueId*" must be **uint** type, and "*description*", should be

string and you can select any types for “*status*”**.** You need to consider 4 status: “*ACTIVE”, “IN_PROGRESS”, “COMPLETE”, “CLOSED”*.

1. ***addIssue()****:* should take an **issue object** in its parameters and add the issue into issueList.
2. Since the ***issueList*** is public, you should get data directly from Remix IDE using a particular issueId;

Checkpoint 02:
Here you will extend checkpoint 01 with an additional function called ***updateIssueStatus()***, which should take the issueId and a status to update the status of that particular issue. You need to maintain a sequence while updating, that is if currently the status is “active”, it can only be updated to “in_progress”, if status: “in_progress”, can be updated to “complete” and so on. You have to consider the NOTEs provided below, and for this if you need any necessary modifications in your code, you can do it.

NOTE:

- In solidity, you cannot directly compare two strings directly by simply trying syntax like: string1 == string2. This will not work in solidity. Therefore, try to understand from the last lab how we can compare the status.
- You need to validate status conditions using any of the error handling techniques mentioned in section 02. Need to use at least 2 types of error methods.
- Hint: study **enum**.

03 Members of Different Types

In solidity, members are various elements associated with different data types. This is similar to the built-in methods, attributes, and properties of objects that we see in different programming languages, e.g. array.append(), array.pop() used in python. However, we are mentioning a few of the members of different types in solidity which may be required for developing the lab project.

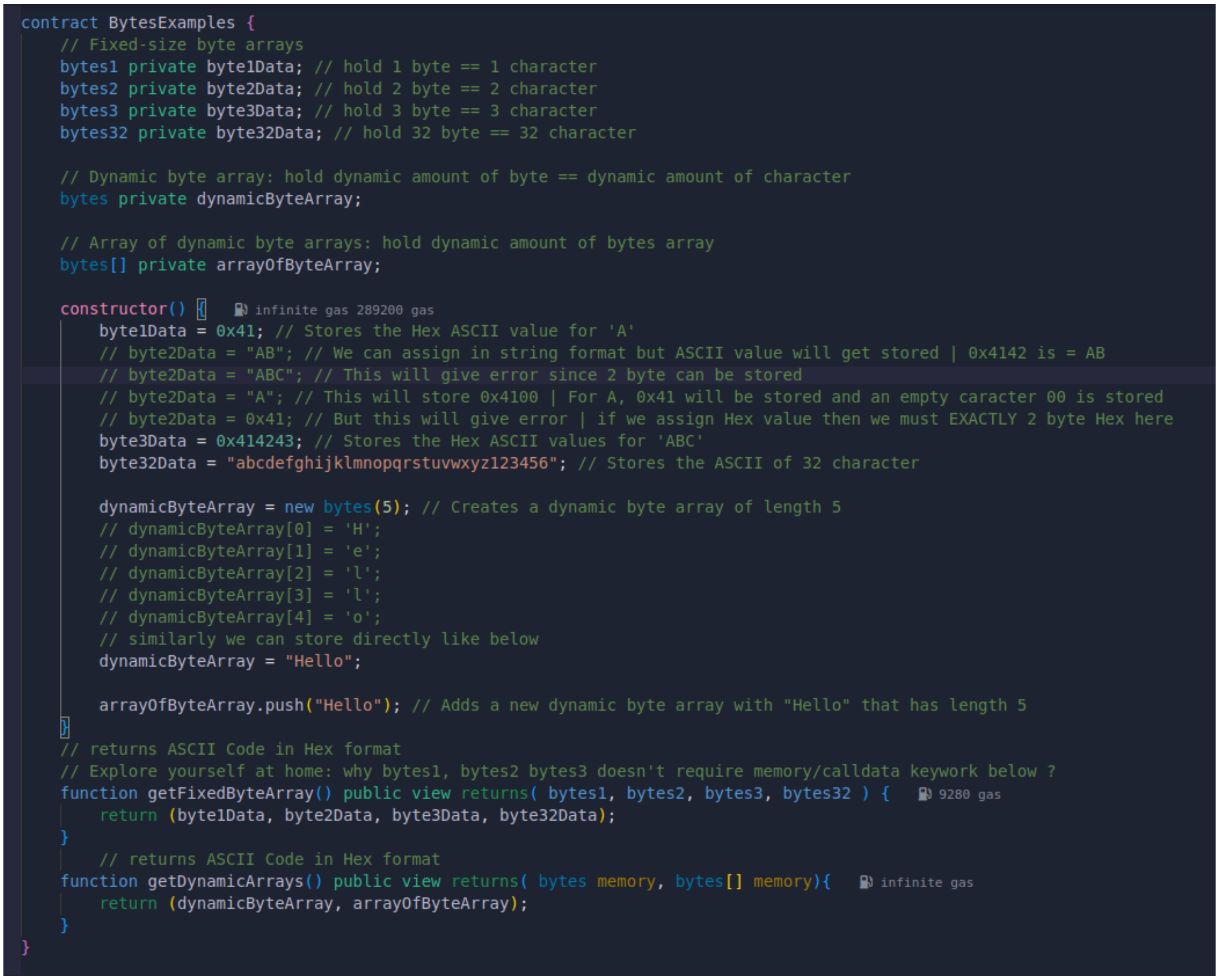
- **Array Members**
  - *push()*: adds an element at the end of the array. E.g myArr.push(5)
  - *pop()*: removes an element from the end of the array. E.g myArr.pop()
  - *length*: returns the length of array. E.g myArr.length
  - *delete*: deletes value of an index or the whole array. E.g delete myArr, delete myArr[1]. If deleted one element, the element will hold its default initialization value such as for string: **“ ”**, for uint: **0**, for boolean: **false**.If the whole array is deleted, the value will be an empty array.
- **String**
  - *string.concat(str1, str2)***:** concates str1, with str2.
- **address**
  - *some_address.balance:* returns the balance of an address
  - *some_address.transfer: sends some currency to an address*
- **msg (This is a global variable)**

- *msg.data:* gives complete calldata of current function call
- msg.value: amount of currency(wei) sent with the message
- msg.sender: address of the entity sending message
- msg.gas: returns the remaining gas for current transaction

04 Bytes & Strings

Bytes

In solidity, bytes is another heavily used reference type. It is a special type of array that holds a sequence of bytes. To store fixed-size arrays of bytes, bytes1, bytes2, bytes3,..., bytes32 are used, such as bytes1 exactly holds 1 byte, bytes2 exactly holds 2 bytes, and so on up to bytes32. On the other hand, if we only use "**bytes"** excluding the numbers, it becomes a dynamic byte array that is resizable. Additionally, **bytes[],** is an array of dynamic bytes array.



```
contract BytesExamples {
    // Fixed-size byte arrays
    bytes1 private byte1Data; // hold 1 byte == 1 character
    bytes2 private byte2Data; // hold 2 byte == 2 character
    bytes3 private byte3Data; // hold 3 byte == 3 character
    bytes32 private byte32Data; // hold 32 byte == 32 character

    // Dynamic byte array: hold dynamic amount of byte == dynamic amount of character
    bytes private dynamicByteArray;

    // Array of dynamic byte arrays: hold dynamic amount of bytes array
    bytes[] private arrayOfByteArray;

    constructor() {    infinite gas 289200 gas
        byte1Data = 0x41; // Stores the Hex ASCII value for 'A'
        // byte2Data = "AB"; // We can assign in string format but ASCII value will get stored | 0x4142 is = AB
        // byte2Data = "ABC"; // This will give error since 2 byte can be stored
        // byte2Data = "A"; // This will store 0x4100 | For A, 0x41 will be stored and an empty caracter 00 is stored
        // byte2Data = 0x41; // But this will give error | if we assign Hex value then we must EXACTLY 2 byte Hex here
        byte3Data = 0x414243; // Stores the Hex ASCII values for 'ABC'
        byte32Data = "abcdefghijklmnopqrstuvwxyz123456"; // Stores the ASCII of 32 character

        dynamicByteArray = new bytes(5); // Creates a dynamic byte array of length 5
        // dynamicByteArray[0] = 'H';
        // dynamicByteArray[1] = 'e';
        // dynamicByteArray[2] = 'l';
        // dynamicByteArray[3] = 'l';
        // dynamicByteArray[4] = 'o';
        // similarly we can store directly like below
        dynamicByteArray = "Hello";

        arrayOfByteArray.push("Hello"); // Adds a new dynamic byte array with "Hello" that has length 5
    }
    // returns ASCII Code in Hex format
    // Explore yourself at home: why bytes1, bytes2 bytes3 doesn't require memory/calldata keywork below ?
    function getFixedByteArray() public view returns( bytes1, bytes2, bytes3, bytes32 ) {    9280 gas
        return (byte1Data, byte2Data, byte3Data, byte32Data);
    }
        // returns ASCII Code in Hex format
    function getDynamicArrays() public view returns( bytes memory, bytes[] memory){    infinite gas
        return (dynamicByteArray, arrayOfByteArray);
    }
}
```

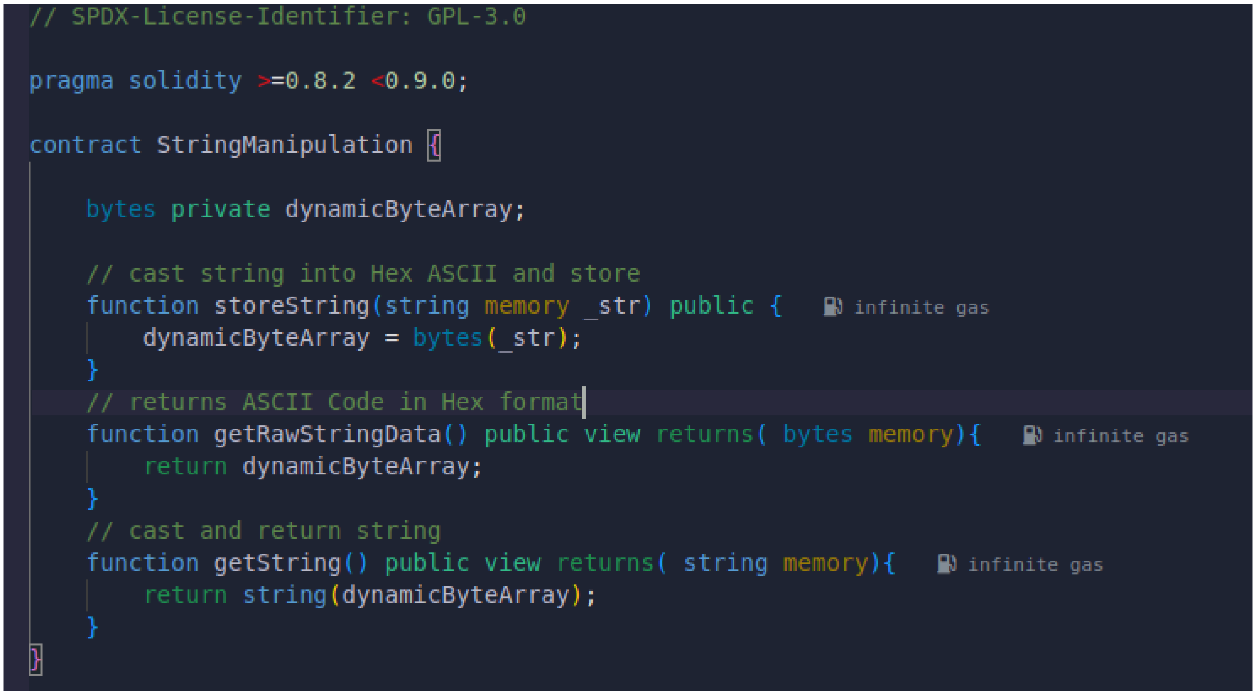
Figure 6: variants of bytes

Question to explore at home: What happens if we write dynamicByteArray = “HelloA” and if we write dynamicByteArray[5] = “A”, in the above code?

String

Working with strings is a bit challenging in Solidity as it does not have any members or built-in methods to manipulate, except for concat. String is unique in solidity compared to other well-known programming languages. In solidity, string is a special typed array and it is equal to bytes. However, it does not have the access capabilities that an array/bytes has. We can manipulate strings by converting them into bytes and casting them again into a string.

Direct modification or accessing the characters feature is not available in solidity. This limitation is because of the design goal of solidity, EVM constraints, and gas efficiency. Though there are libraries/packages available that provide various methods to manipulate strings. However, this is not within the scope of this lab.



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

contract StringManipulation {

    bytes private dynamicByteArray;

    // cast string into Hex ASCII and store
    function storeString(string memory _str) public {
        dynamicByteArray = bytes(_str);
    }
    // returns ASCII Code in Hex format
    function getRawStringData() public view returns( bytes memory){
        return dynamicByteArray;
    }
    // cast and return string
    function getString() public view returns( string memory){
        return string(dynamicByteArray);
    }
}
```

Figure 7: String & Byte Conversion

Checkpoint 03: Create a contract that will have 2 functions. Name it **storeString** and **getSlicedString**. The storeString will receive a string from the user. The getSlicedString will take the start and end index of the string and return the substring. For example, if you have a string str = “Hello” and you pass getSlicedString(1, 4), this should return a substring “ello”.

<u>TestCases</u>:

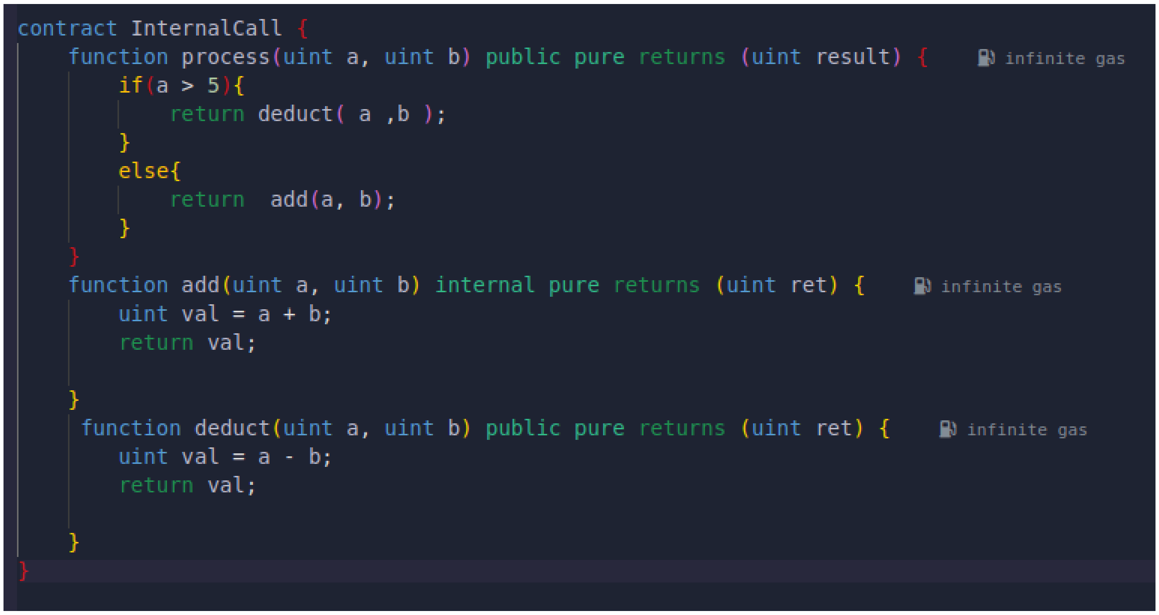
0, 0 should return = > “H”
0, 1 should return => “He”
0, 2 should return => “Hel”

Question to explore at home: How can you compare two strings to determine whether they are equal or not? Explore at least two approaches to do it.

05 Function Calls - Internal & External

A. Internal Function Call

To call a function within the same contract instance, in Solidity, a special keyword called ***internal*** is used. These types of functions are called Internal functions, and the call itself is known as internal function calls. Within the contract, a function can also be called from another one without using the ***internal*** keyword. But, if we use an internal keyword, the function will not be visible from outside the contract instance directly.



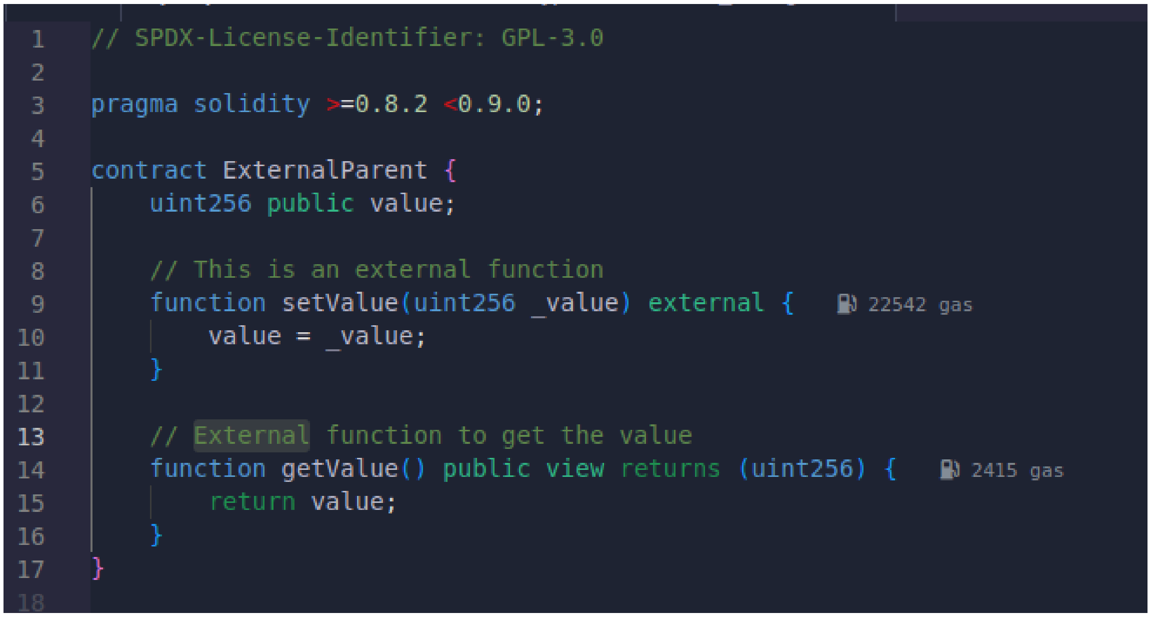
```
contract InternalCall {
    function process(uint a, uint b) public pure returns (uint result) {
        if(a > 5){
            return deduct( a ,b );
        }
        else{
            return  add(a, b);
        }
    }
    function add(uint a, uint b) internal pure returns (uint ret) {
        uint val = a + b;
        return val;

    }
     function deduct(uint a, uint b) public pure returns (uint ret) {
        uint val = a - b;
        return val;

    }
}
```

Figure - 2

Figure 2 shows two function calls, “add” and “deduct,” invoked from the function “process”. Though the two functions can be called from another function within the same contract, the difference is that “add” is an **internal** function and it is only used from another function call within the same contract or a derived contract. On the contrary, the “deduct” can also be called externally.

B. External Function Call

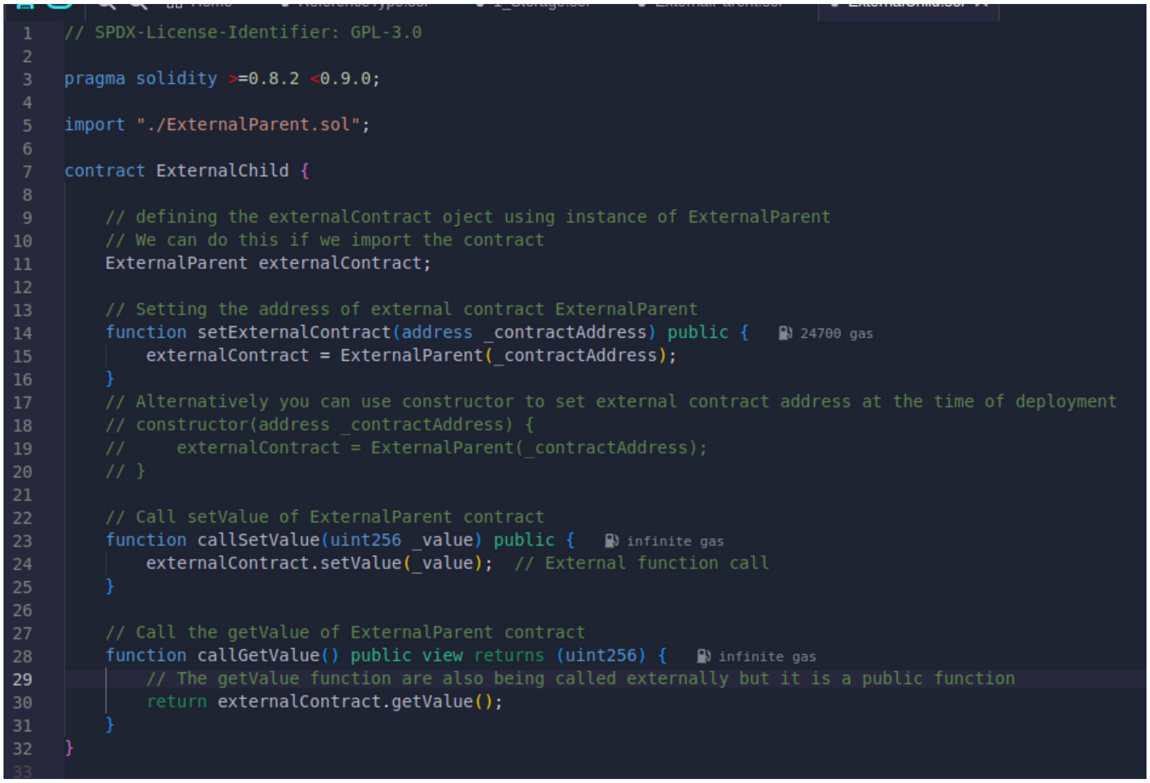
In solidity, we can call a function from one contract to another contract function by instantiating the reference. This is similar to the public functions; however, external functions can only be called externally and are more gas-efficient if used correctly. To define an external function, the keyword **external** is used. In Figures 3 and 4 provided below, we have two smart contracts called **ExternalParent** and **ExternalChild**. In solidity, we can use one contract from another by importing it. An example can be found in Figure 4. The **ExternalChild** function **imports ExternalParent** contracts and creates an instance of it. **However, to use an external contract, such as Figure 4 imported *ExternalParent,* where we first need to deploy the *ExternalParent* and then we need to pass the address to the ExternalChild so that it can call the external contract functions. We can set this address by using a constructor or calling a function that takes the address.** Next, the **ExternalChild** contract assigns the address of the deployed **ExternalParent** contract to the variable **externalContract**. This can be done by either using the function or the constructor(line 15 or 19 in Figure 4).



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

contract ExternalParent {
    uint256 public value;

    // This is an external function
    function setValue(uint256 _value) external {
        value = _value;
    }

    // External function to get the value
    function getValue() public view returns (uint256) {
        return value;
    }
}
```

Figure 3: Contract with external function

Finally, **callSetValue** and **callGetValue** can call the functions of the **ExternalParent** contract externally using simple dot notation. Notice, **setValue** is an **external** function, and **getValue** is public, and both can be called externally.



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

import "./ExternalParent.sol";

contract ExternalChild {

    // defining the externalContract oject using instance of ExternalParent
    // We can do this if we import the contract
    ExternalParent externalContract;

    // Setting the address of external contract ExternalParent
    function setExternalContract(address _contractAddress) public {
        externalContract = ExternalParent(_contractAddress);
    }
    // Alternatively you can use constructor to set external contract address at the time of deployment
    // constructor(address _contractAddress) {
    //     externalContract = ExternalParent(_contractAddress);
    // }

    // Call setValue of ExternalParent contract
    function callSetValue(uint256 _value) public {
        externalContract.setValue(_value);  // External function call
    }

    // Call the getValue of ExternalParent contract
    function callGetValue() public view returns (uint256) {
        // The getValue function are also being called externally but it is a public function
        return externalContract.getValue();
    }
}
```

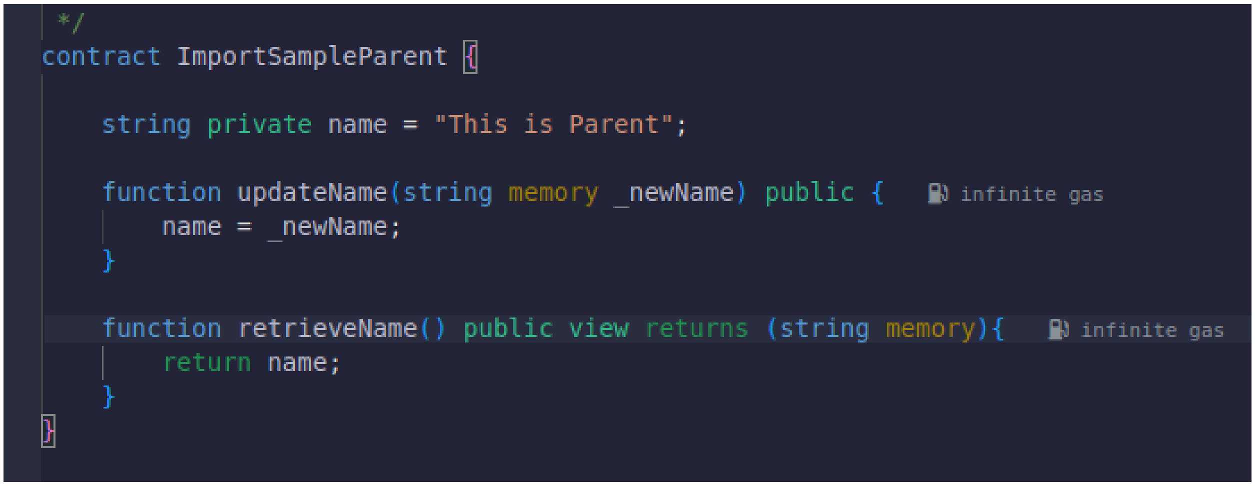
Figure 4: Calling external and public functions externally

06 Working with multiple contracts

In solidity, we can either import or inherit a contract to use its functions from another contract. However, in this lab, we will not work with inheritance, which involves more concepts of object-oriented programming.

import

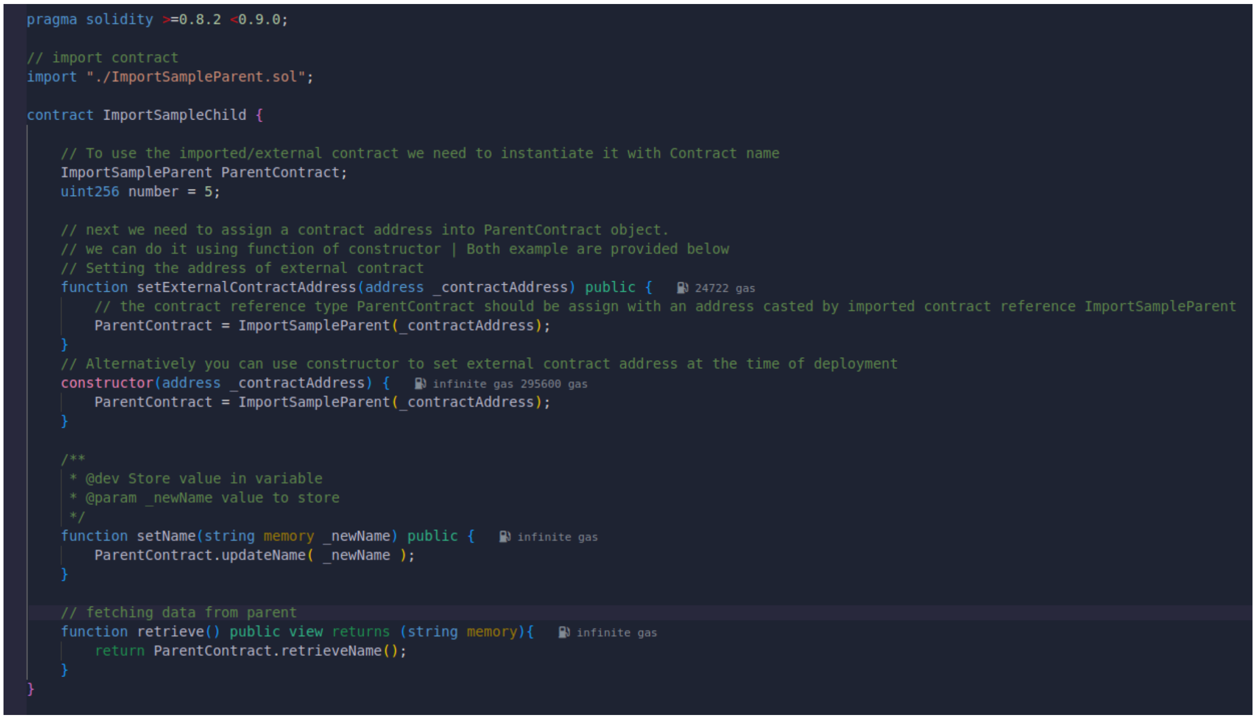
When using **import** to use external contracts, no private/internal data can be directly accessible from the imported contract. The functions need to be public or external. Figure 8 shows an example of a parent contract titled “ImportSampleParent”.



```
  */
contract ImportSampleParent {

    string private name = "This is Parent";

    function updateName(string memory _newName) public {    infinite gas
        name = _newName;
    }

    function retrieveName() public view returns (string memory){    infinite gas
        return name;
    }
}
```

Figure 8: Sample Contract that will get imported

This contract is externally imported in the contract “ImportSampleChild”, which is presented in Figure 9 below.



```
pragma solidity >=0.8.2 <0.9.0;

// import contract
import "./ImportSampleParent.sol";

contract ImportSampleChild {

    // To use the imported/external contract we need to instantiate it with Contract name
    ImportSampleParent ParentContract;
    uint256 number = 5;

    // next we need to assign a contract address into ParentContract object.
    // we can do it using function of constructor | Both example are provided below
    // Setting the address of external contract
    function setExternalContractAddress(address _contractAddress) public {    24722 gas
        // the contract reference type ParentContract should be assign with an address casted by imported contract reference ImportSampleParent
        ParentContract = ImportSampleParent(_contractAddress);
    }
    // Alternatively you can use constructor to set external contract address at the time of deployment
    constructor(address _contractAddress) {    infinite gas 295600 gas
        ParentContract = ImportSampleParent(_contractAddress);
    }

    /**
     * @dev Store value in variable
     * @param _newName value to store
     */
    function setName(string memory _newName) public {    infinite gas
        ParentContract.updateName( _newName );
    }

    // fetching data from parent
    function retrieve() public view returns (string memory){    infinite gas
        return ParentContract.retrieveName();
    }
}
```

Figure 9: A contract importing and using another one

Home-Task: In this checkpoint, you will modify the **IssueTracker** contract that we already solved.

1. However, this time you need to update the visibility of ***issueList*** mapping as **private.** Therefore, this variable **can not be accessed directly** outside the contract.
2. Additionally, make the visibility of ***addIssue,*** and ***updateIssueStatus*** as **external**.

3. Create a new contract called ***Driver*** and build 3 functions within it, through which you can fetch value from ***addIssue*,** and ***updateIssueStatus*** function. Remember that you need to set the contract address of **IssueTracker,** to interact with it.
4. Create a new function ***getIssue*** inside this driver method and try how you can fetch the issue by ID from the **IssueTracker** contract. If you need to create more functions, you can do it.

Some key things about Solidity

- We cannot return data from a function if it modifies the state variable and is being called from EOA accounts.
- In solidity, there are only a few built-in methods/members available.
- In solidity, string manipulation is challenging as it does not have built-in methods/members except concat. We cannot even directly compare two strings. Though some packages/libraries can solve it in one line, we will work only on raw solidity in this course.
- Use the **view-**in function if you are reading data.
- Use **pure** in function if you are neither reading nor modifying data from the contract state.
- While interacting with function parameters and return data type for reference typed data, we are required to mention data areas such as memory, calldata, and storage. Proper use of these data areas can save a lot of gas.
- Few error mechanisms and validation are handled in a bit different way in solidity.

References:

- Official Documentation: https://docs.soliditylang.org/en/v0.8.26/
- Smart Contract Best Practices: https://github.com/ConsenSys/smart-contract-best-practices
- Solidity Cheatsheet: https://docs.soliditylang.org/en/latest/cheatsheet.html#members-of-bytes-and-string