Lab: Introduction to Solidity and Remix IDE

Objectives:

- To understand the solidity programming language and interact with remix IDE.

Submission:

- Students should finish the checkpoints during lab class.

Description:

In this lab, you will learn the basics of solidity programming language and write your own smart contract for the ethereum using it. You will learn how to facilitate writing smart contracts and interact with it 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 highly influenced by C++, Python and Javascript. However, if you know Java or Javascript you will find similarities in syntax.

To develop, test and debug smart contracts using solidity language, Remix IDE is considered to be the best IDE among the blockchain developer community. It is considered as the industry standard for its rich set of plugins, debug features having intuitive Graphical User Interface.

Prerequisites:

- Familiarity with programming with at least one programming language
- blockchain
- Web Browser. Recommended - Google chrome or Firefox

Section 1:

In this section we will discuss some of the concepts that are necessary to write smart contracts using solidity.

- **Ethereum Virtual Machine (EVM):** EVM is the runtime environment of smart contracts in Ethereum. It is totally sandboxed and completely isolated meaning the code running inside the EVM has no access to the network, filesystem or other processes. Even smart contracts have limited access to other smart contracts
- **Accounts:** In ethereum, Every account has a persistent key-value store mapping 256-bit words to 256-bit words called **storage.** Moreover, every account has a balance in Ether (in “wei”, to be exact 1 ether is 10^{18} wei) which can be modified by sending transactions with ether. Ethereum has two kinds of accounts that share the same address space are **external accounts** and **contract accounts.**

- External accounts are controlled by public-private key pairs and the account is determined from the public key
- Contract accounts are controlled by the code stored together with the account details. The address of a contract is determined at the time the contract is created.
- Regardless of whether or not the account stores code, the two types are treated equally by EVM.

- **Transactions:** A transaction is a message that is sent from one account to another account. The message can include binary data(payload) and ether. In ethereum, there are three types of transaction :
  1. If the target account is an external account, the value in the transaction is transferred to the target account
  2. If the target account contains code, the code is executed and the payload is provided as input data. These can act as parameters while calling a specific function within the contract
  3. If the target account is not set meaning the transaction does not have a recipient, the transaction creates a **new contract.** The output data of this execution is permanently stored as the code of the contract.

- **Gas:** A transaction which executes code is charged with a certain amount of balance called **gas**. While the EVM executes the transaction, the gas gradually depletes according to specific rules. While doing transaction, a value as **gas price** is set by the creator of the transaction, who has to pay **gas_price * gas** upfront from the sending account. If any gas is left, it is refunded to the creator's account. However, if the gas ends before the transaction is completed then an **out-of-gas** exception is triggered and all the modifications made to the state gets reverted.
- **Areas to store data:** EVM has three areas where it stores data: storage, memory and the stack
  1. **Storage:** in ethereum, each account has its own data area called **storage**. It is persistent meaning once data is stored, it is saved in the blockchain. Storage holds data in key-value form that maps 256-bit words to 256-bit words and comparatively it is costly to read and even cost more to modify. A contract neither reads nor writes to any storage apart from its own.
  2. **Memory:** It is volatile in the sense that a contract obtains a freshly cleared instance in each message call to a contract. The larger the memory grows the more it costs and at the time of expansion, the cost in gas must be paid.
  3. **Stack**: EVM is not a register machine but a stack of machines, so all computations are performed on a data area called stack. It has a maximum size of 1024 elements and contains words of 256 bits. It is mainly used by the code to carry out its computation. All other operations take the topmost two or more elements depending on the operations from the stack and push the result onto the stack. In stack, it is possible to move elements to storage or memory in order to get deeper access to the stack but not possible to access arbitrary deeper elements in the stack without first removing the top of the stack.

- **EVM Instruction Set**: In Evm, instruction sets are kept minimal to avoid incorrect or inconsistent implementations. The usual arithmetic, bit, logical and comparison operations and conditional and unconditional jump are present.
- **Message Calls**: contracts can call other contracts or send ether to non-contract accounts by the means of message calls. Message calls are similar to transactions, in that they have a source, a target, data payload, ether, gas and return data. In fact, every transaction consists of a top-level message call which in turn creates further message calls.
- **Delegatecall & Library:** There exists a special variant of a message call, named **delegatecall.** It is identical to a message call apart from the fact that the code at the target address is executed in the context of the calling contract. This means that a contract can dynamically load code from a different address at runtime
    - Storage, current address and balance still refer to the calling contract, only the code is taken from the called address

  This makes it possible to implement the “library” feature in Solidity. Reusable library code that can be applied to a contract’s storage, e.g. in order to implement a complex data structure

- **Logs:** It is possible to store data in a specially indexed data structure called **logs**. It is used to implement events. A contract cannot access log data after it has been created, but they can be efficiently accessed from outside the blockchain.
- **Events:** Events let you conveniently access the EVM logging facilities. They may be used for “calling” JavaScript callbacks inside the DApp's user interface which allow to listen for the events. A contract can emit events after a certain condition is met or a certain code is executed. A DApp will catch that event and take some actions.
- **Create:** Contracts can even create other contracts using a special opcode. The only difference between these **create calls** and normal message calls is that the payload data is executed and the result stored as code and the caller/ creator receives the address of the new contract on the stack.
- **Self-destruct:** To remove code from the blockchain, a contract at that address performs a **selfdestruct** operation. It removes the storage and code after the remaining ether at that particular address is sent to a designated target. It should be noted that, “selfdestruct” is not the same as deleting data from a hard disk. Even if a contract is removed by the operation it is still part of the history of blockchain and retained by most ethereum nodes. Removing the contract is potentially dangerous as if someone sends ether to remove contracts, the ether is lost forever.
- **Deactivate:** To deactivate contracts, one should instead **disable** them by changing some internal state which causes all functions to revert. This makes it impossible to use the contract, as it returns Ether immediately.

Scoping and Declaration in Solidity:

In solidity, variables only get scoped up to their declaration at the end of the semantic block. On the other hand, declared variables are initialized with their default value at the beginning of the function.

- The default value of a **bool** is **false**
- For **uint** or **int** types, it is **0**
- In the case of statically-sized arrays, as well as **bytes1** to **bytes32**, all individual elements will be initialized to the default value that corresponds to the type of the element
- Lastly, for dynamic size arrays, **strings** and **bytes**, the default value will be an empty string or array

Section 2:

In this section, you will learn the basics of solidity programming language and smart contracts.

Solidity Data Types:

Solidity has a rich collection of built-in and user-defined data types. We will discuss some common types here:

- **Boolean:** a value of either **true** or **false**. We also can represent boolean types using operators such as:
  - ! (logical negation)
  - && (logical conjunction, “and”)
  - || (logical disjunction, “or”)
  - == (equality)
  - != (inequality)

- **Integer:** There are int and uint integers of various sizes. uint and int are aliases for uint256 and int256, respectively.
  - uint8 to uint256 in steps of 8. For example: uint8, uint16, uint24 etc. In solidity, **uint** refers to the unsigned integer values. It holds **only positive values**. Here, the range of value that you can store using uint8 is: 0 to 255, the range of value that you can store using uint16 is: 0 to 65,535.

```
uint8 num1 = 127;
uint256 num2 = 6532195;
```

  - int8 to int256 in steps of 8. For Example: int8, int16, int24 etc. In solidity, **int** refers to the integers. It holds **both negative and positive values**. Here, the range of values that you can store using int8 is: -128 to 127, the range of value that you can store using int16 is: -32,768 to 32,767.

```
int8 number1 = -127;
int256 number2 =  6532195;
```

- **String Literals and Types:** String literals are written with either double or single-quotes. For example: “Jhon Doe” or ‘Jhon Doe’. String literals also support escape characters for example: \n (newline), \b (backspace).

```
string location = "66, mohakhali, BRAC University, Dhaka";
```

- **Address:** Address is a unique type in solidity. Address type holds the ethereum account address which is a value of 20-byte.

```
address owner = 0xc0ffee254729296a45a3885639AC7E10F9d54979;
```

Address types hold some members of their own. For example: using the property **balance**, we can query the balance of an address. For example: the condition below executes some code if the balance of the owner account is greater or equal to 10.

```
if (owner.balance >= 10){
        // sample code
    }
```

- **Function Type:** From the contract’s perspective, functions can be classified as internal and external. From the access modifiers perspective, functions are:
  - **Internal:** can only be called inside the current contract
  - **External:** consists of an address and a function signature and can be passed via or returned from external function calls
  - **public:** part of the contract interface and can be called either internally or via message calls
  - **private:** only visible inside the contract they are defined.
  - **View:** will read the state variable but will not modify it.
  - **pure:** can not read state variables and also cannot change it.
  - **payable:** allows functions to send or receive ethers.

A general syntax of function is given below:

```
function name (<parameter types>) {internal|external|public|private} [pure|view|payable] [returns (<return types>)]
```

Sample 1:

A sample function called ***retrieve*** which reads(view) the state variable ***number*** and return a uint256 value ***newNumber*** are shown below:

```
function retrieve() public view returns (uint256){
        uint256 newNumber = number + 100;
        return newNumber;
    }
```

Sample 2: The sample function doesn't read or modify the state variable. It takes two parameters called **firstName** and **lastName** and returns the **fullName**.

```
function mergeName( string memory firstName, string memory lastName ) public pure returns (string memory){
    string memory fullName = string.concat(firstName, lastName);
    return fullName;
}
```

Sample 3:

The sample function modifies the state variable ***number*** and returns nothing.

```
function store(uint256 num) public {
        number = num;
    }
```

Checkpoint 1: Create a function that takes your studentID of integer value and returns your name. If you enter the ID of any other student, it should return "This is not your ID". If the code works fine, convert the studentID into string and try to compare it again. Let your teacher know, whether this new approach is working or not.

- **Enum:** Enums are one way to create a user-defined type in solidity. It restricts variables to have a value other than its predefined one. This is really helpful because it increases the readability of code and reduces the number of bugs in the program. For example: from the code in fig-1 we see a code which will not allow users to select any device other than laptop, computer and mobile. If you notice the setter functions, for example **setMobile()**, we can select mobile by using **DeviceList.Mobile** or **DeviceList(0).** Both give the same result. Basically enums stores user defined data that holds values like array index in uint8. For example, the first member of the enum of given code in fig-1 holds value: Mobile = 0, the second one holds Laptop= 1 and the third one, Desktop = 2.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract EnumSample {
   enum DeviceList{ Mobile, Laptop, Desktop }
   DeviceList myDevice;

   function setMobile() public {
       myDevice = DeviceList.Mobile;
       // below line of code works same as the line of code above
       // myDevice = DeviceList(0)
   }

   function setLaptop() public {
       myDevice = DeviceList(1);
       // below line of code works same as the line of code above
       //myDevice = DeviceList.Laptop
   }

   function setDesktop() public {
       myDevice = DeviceList(2);
       // below line of code works same as the line of code above
       //myDevice = DeviceList.Desktop
   }

   // get state variable
   function getMyDevice() public view returns (DeviceList) {
       return myDevice;
   }
}
```

Fig-1: enum

- **Array:** In solidity, arrays can be of both fixed and dynamic size. An array of fixed size k and element type T is written as T[k], and an array of dynamic size is T[]. A sample code of arrays is given in fig-2. **push**, **pop**, **length** are some heavily used members of array. However, fixed size arrays only have access to length among them.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

pragma solidity >=0.7.0 <0.9.0;

contract SampleVariables {
 string[3] fixedArray = ["Laptop", "Mobile" , "Desktop"];
 string[] dynamicArray;
 // if you want to declare dynamic array with some initial length you can use:
 //string[] dynamicArray = new string[](3);
 // above line creates dynamic array with initial lengh 3

 function add(string memory val1, string memory val2) public {
    dynamicArray.push(val1);
    dynamicArray.push(val2);
    dynamicArray.pop();
    // fixedArray.push(val1);  //==> This gives error
    // fixedArray.pop();  //==> This gives error
 }
 function getFixedArray() public view returns (string[3] memory) {
   return fixedArray;
 }
 function getDynamicArray() public view returns (string[] memory) {
   return dynamicArray;
 }
}
```

Fig-2: fixed and dynamic length arrays

- **Mapping:** Mapping is one of the heavily used types in solidity for large data. Mapping stores data in (keyType => valueType form). **keyType** can be any elementary type such as any built-in value types. However, user defined or complex types like contract types, enums, other mapping, structs and any array type apart from bytes and string are not allowed as **keyType**. On the other hand, **valueType** can be of any type including a new mapping.
  Sample 1: A sample mapping that holds addresses and their respective balance

```
mapping(address => uint) public balances;
```

  Sample 2: A sample smart contract that contains a mapping of vote count of individual candidates.

```
pragma solidity >=0.7.0 <0.9.0;

contract SampleVariables {
  mapping( string => uint256 ) public voteCount;

  function vote( string memory voterName ) public{
      voteCount[ voterName ] = voteCount[ voterName ] + 1;
  }
   function getVote( string memory voterName ) public view returns(uint256){
      return voteCount[ voterName ];
  }
}
```

Checkpoint 2: Write a smart contract that stores student id and the name of the student. Also create a separate function that takes studentID and returns student name. [You must use mapping]

- **Struct:** In solidity, we can define new types of our own from the other types in the form of structs. For example:

```
contract SampleVariables {
  struct Student {
      string name;
      uint256 id;
      bool isCurrentStatus;
  }
  Student[] studentList;

  function addStudent(string memory _name, uint256 _id, bool currentStatus ) public {
       Student memory newStudent = Student(_name, _id, currentStatus);
       // you can also write it as mentioned below:
       // Student memory newStudent = Student({ name: _name, id: _id, isCurrentStatus: currentStatus });
       // also can be written like below:
       // Student memory newStudent;
       // newStudent.name = _name;
       // newStudent.id = _id;
       // newStudent.isCurrentStatus = currentStatus;
       studentList.push( newStudent );
  }

   function getStudents() public view returns(Student[] memory){
       return studentList;

   }
}
```

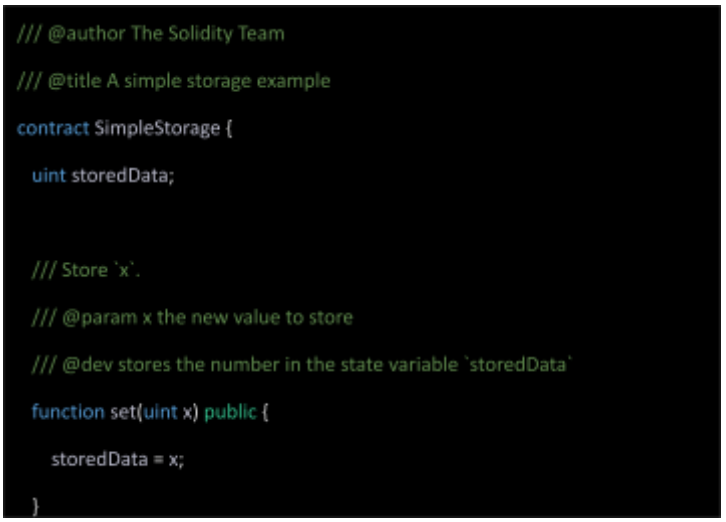
Fig-3: struct structure

Checkpoint 3: Write a smart contract that stores gpa, studentID, semesterName, courseTaken by students of each semester. Additionally, create a separate function so that we can see all the semester data of students. [Consider the value of your **gpa: string**, **semesterName: string**, **studentID:uint**, **courseTaken: uint.**]

- **Control structure:** The majority of the JavaScript control structures are provided in Solidity as well, with the exception of **goto** and **switch**. All with the normal semantics that are known from JavaScript or C Parentheses, for conditionals, must *not* be omitted, however, curly braces are possible to omit around bodies that are single-statement. Solidity provides these control structures:
  - **If**
  - **while**
  - **else**
  - **for**
  - **do-while**
  - **continue**
  - **return**
  - **break**

Solidity Contract:

- **License Identifier:** A smart contract starts with its license identifier in a comment which describes its copyright. If you do not want to specify a license, you can use UNLICENSED as the license identifier. For example: below is the syntax of mapping which holds the balance of addresses.
- **pragma:** We must mention the compiler version you are going to use to compile the contract under the "pragma" keyword. In the given sample contract in Fig-5, you will see the pragma directive telling us that the mentioned smart contract is written for solidity compiler version 0.7.0 or newer versions that does not include or exceed version 0.9.0. We also can use caret(^) to specify the versions. For example, ^0.7.0 tells us that the smart contract cannot be compiled using earlier than version 0.7.0 and also it will not work on a compiler starting from version 0.8.0
- **Comments:** Like in most other programming languages, comments can be used for providing hints throughout the code to make it easier to read. Solidity also features the ***natspec*** comment. A sample of ***natspec*** comments is provided in fig-4.



```
/// @author The Solidity Team
/// @title A simple storage example
contract SimpleStorage {
  uint storedData;

  /// Store `x`.
  /// @param x the new value to store
  /// @dev stores the number in the state variable `storedData`
  function set(uint x) public {
    storedData = x;
  }
```

Fig-4: sample natspec comments

- **Contract Structure:** Solidity contracts are much like **classes** in object-oriented programming languages. It can **inherit** elements from other contracts. Every contract contains declarations of elements such as functions, functions modifiers, state variables, struct, enum types, events etc. as needed for its purpose. A sample solidity program of smart contract is given below:

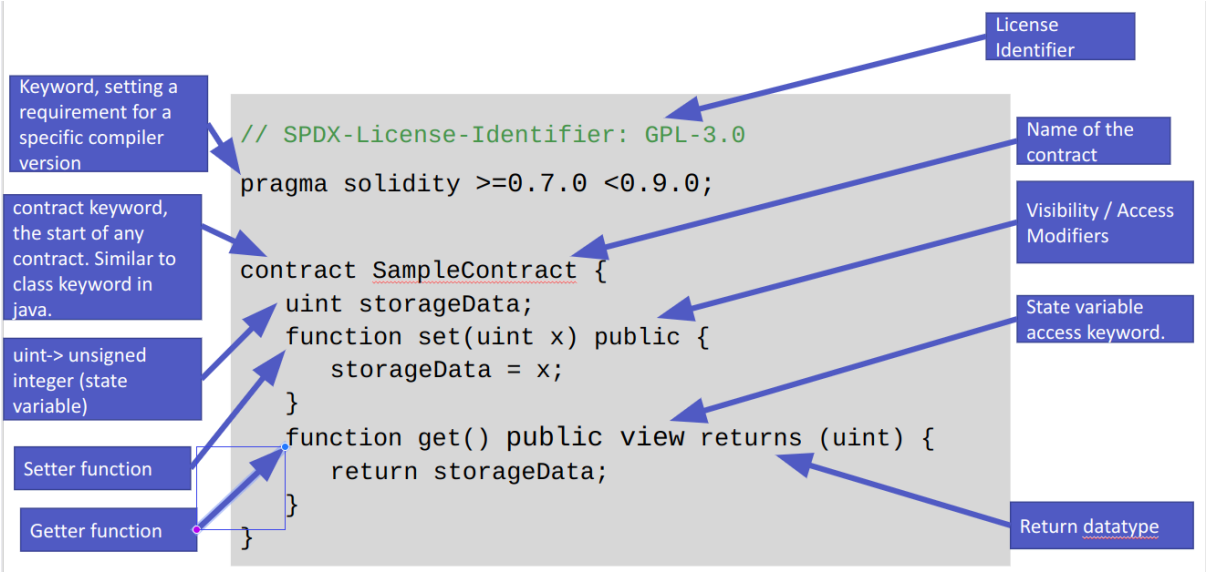


Fig-5: a simple smart contract

- **Function:** In solidity, function visibility varies to other contracts and can be called both internally and externally. There are four types of function visibility which are: public, private, internal and external. The visibility specifier is given after the type **for state variables** or in function, between parameter list and return parameter list.

```
function get() public view returns (uint) {
    return storageData;
}
```

For example, if we notice the above get() function, we will see a keyword called **public.** This declares the function is part of the contract interface and can be called either internally or via message calls.

- **Function Modifier:** Functions modifiers in solidity are used in a declarative way to amend function semantics. One common use of modifiers is to restrict or validate access. A simple example of modifier is given below:

```
pragma solidity >=0.4.22 <0.6.0;

contract Purchase {
        address public seller;

        modifier onlySeller() { // Modifier
          require( msg.sender == seller, "Only seller can call this." ); _;
        }

        function abort() public view onlySeller {
          // Modifier usage // ...
        }
}
```

- **State and Local variable:** State variables are values that are permanently stored in the contract's **storage**. State variables cost gas for storing and manipulating data in the state. For example, in the fig-7, ***uint statePrice*** is a state variable.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

contract SampleVariables {
  uint statePrice = 100;

  // adding price worth of statePrice added with vat to state variable
  function reducePrice(uint vat) public {
    uint localPrice = 50;

    localPrice += vat;
    statePrice = localPrice;
  }

  // function that return a local variable data
  function checkLocalName() public pure returns (string memory) {
    string memory localText  = "CSE446-Blockchain & Cryptocurrencies";

    return localText;
  }

  // get state variable
  function get() public view returns (uint) {
    return statePrice;
  }
}
```

Fig-7: state and local variables

On the other hand, in fig-7, ***localPrice*** is a local variable that initiates when the function is called and gets destroyed when the call is completed. Local variables get stored in the stack and do not cost any gas. However, there are certain types of data that by default gets stored in storage. **String** is one of them. From fig-7, we see a string variable called ***localText.*** Therefore, to use string type data inside a function like local variable we must use a memory keyword which declares that the data is neither stored in storage nor in the stack. It gets stored in memory and costs less gas than storage.

- **Events: Solidity**, events allow **convenient** use of **EVM logging** facilities, that can be used for "calling" JavaScript **callbacks** inside the user interface of the dapp, allowing us to **listen** to the events. As a member of contracts, events are inheritable meaning

a contract inheriting another contract can call the event of the parent contract. An event needs to be dispersed using emit. A sample contract of events is shown in Fig-8.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract SampleEvent {
        event Deposit( address indexed _from, uint indexed _id, uint _value );

        function deposit(uint _id) public payable {
            // Events are emitted using `emit`, followed by
            // the name of the event and the arguments
            // (if any) in parentheses. Any such invocation
            // (even deeply nested) can be detected from
            // the JavaScript API by filtering for `Deposit`.
            emit Deposit(msg.sender, _id, msg.value);
        }
}
```

Fig-8: event and emit

Checkpoint – 4: Write a solidity contract that will take the last two digits of your id . If your full id is divisible by those last two digits, return your full name and if it is not divisible, return a message saying not found. **[NOTE: you must use at least one local and a state variable]**

Checkpoint– 5: Find the second lowest value in a fixed length array. Assuming values of array will be from range 0-50. Also duplicate values can exist. Your program should provide correct result for the test case provided below:

- [1, 3, 2, 5, 3, 50, 8, 20, 1]
- [2, 2, 5, 6, 3, 100, 50, 8, 20]
- [0, 2, 0, 2]
- [1, 1, 1, 2]

Additional Resources

You can find many more tutorials on the Internet. Two samples of additional Solidity tutorials can be found from these locations. Please go through one of them to harness your solidity skills.

- https://www.w3schools.io/blockchain/solidity-tutorials/
- https://www.tutorialspoint.com/solidity/index.htm