LAB 01 August, 2023

1 Enums

1.1 Introduction

- In Solidity, enums (which is short for enumeration) serves as a means to create **user-defined data types** that enable developers to model choices and track states within their smart contracts.
- Enums are particularly useful for enhancing contract maintenance and readability by providing names for integral constants.
- By restricting a variable to a predefined set of values, enums contribute to **reducing the occurrence of bugs** in code.

1.2 Syntax

To declare an enum in Solidity, the following syntax is used:

```
enum <enum _name> {
element 1,
element 2,
..., element n
}
```

The enum name represents the name of the enum, while the elements within the curly braces denote the possible values that the enum can take. **These values are referred to as enums or enum elements**.

Each enum element is associated with an **integer value starting from zero**, and a default value can also be assigned to the enum.

1.3 Major Advantage

One of the advantages of enums in Solidity is their ability to **enhance code clarity** and **maintainability**. By providing clear and meaningful names for predefined values, enums make the intention of the code more explicit and easier to understand.

By utilizing enums, developers can enhance code reliability, readability, and maintainability. Their ability to model choices and track state within contracts contributes to reducing bugs and improving the overall quality of the codebase.

1.4 Example

An example use case can be illustrated by considering an application in which you need to track the progress of a student's paper. By utilizing enums, it becomes possible to restrict the paper status to only **Checking** (checking is in progress), **Pass**, **Fail** and **Scholarship** (student has passed with exceptional marks).

This avoids any ambiguity about any other state that the paper could be in, eliminating the possibility of errors or inconsistencies in the code.

```solidity
//SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;

contract Enum{

    enum Exam{
        Checking,
        Pass,
        Fail,
        Scholarship
    }
}
```

1.5 Making Enum Variable

The enum that you make is a user-defined data type. So like all other data types you can also make a variable for your enum data type as well.

You can make a state variable for your enum by writing the **enum name**, followed by its **visibility**, and then the **variable name**.

An enum variable for the example in section 1.4 is given below:

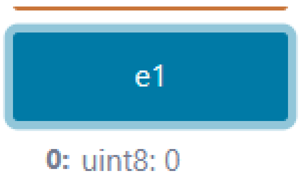
```
Exam public e1;
```

1.6 Default Value

The default value of an enum data type will be the element you have defined in the **0th index of the enum** that you have made.

The **value** that is returned by the **enum variable** is of **uint type**, and it is the **index number of** the **element** currently stored in the enum.

So for a **default enum variable** value you will get 0, and it will represent the 0th index element.



0: uint8: 0

Exercise

a) Observe and Run the example in section 1.4 and 1.5.

b) Make a contract named PizzaDeli. In it make an Enum named PizzaOrder, its **elements** should include **Pending** (your pizza order is pending), **Preparing** (your pizza is being prepared), **Delivering** (the rider has picked up your order and is now delivering it to your address) and **Received** (you have received the order).

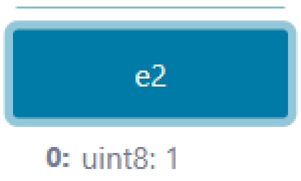
1.7 Setting A Value to The Enum Variable

There are multiple methods to set a value to the enum variable:

1. You can simply write the Enum name and the element separated by a dot.

```
Exam public e2 = Exam.Pass;
```

   In this case, the value returned by e2 will be 1, as Pass is in 1st index place.



```
0: uint8: 1
```

2. You can write the enum name and then the index number of the element that you want to assign enclosed in circular brackets.

```
Exam public e3 = Exam(3);
```

Exercise

What do you think will be the uint value returned by e3 in this case?

Exercise

In your contract PizzaDeli, make three variables named P1, P2 and P3 respectively for your PizzaOrder enum where:

a) P1 will be assigned no value.

b) P2 will be assigned Preparing status using method 1 in section 1.7.

c) P3 will be assigned Received status using method 2 in section 1.7.

1.8 Setting and Getting Enums

- When setting a value to an enum variable, **you will write the Enum name in place of the datatype name in the input parameters followed by a variable name**. And the value that you will input when the function is called will be the index number of the element that you want to store in your enum variable.

```
Exam public e4;
    function set(Exam _e) public{

        e4 = _e;
    }
```

- Similarly, when getting enums, the return type in the returns parameter will be the enum name. The value that will be returned will be the index number of the element stored in the enum variable.

```
function get() public view returns(Exam){

        return e4;
    }
```

1.9 Deleting an Enum Variable

You can also delete enum variables in a function by using delete keyword followed by the enum variable name. This will set the enum variable to its default value.

```
function udelete() public{
        delete e4;
    }
```

Exercise

In your contract PizzaDeli:

a) Make a setter function for P1 variable.

b) Using what you have learned in the previous Lab about Functions, make a single getter function that returns values of all three variables P1, P2 and P3.

1.10 Modifying Enum Based On Conditions

Suppose you want to make a function Pass, that modifies the enum variable state to Pass, but only if the enum variable was previously in Checking state. You can implement such a function using a simple if condition.

```
Exam public e1;
    function Pass() public{
        if(e1 == Exam.Checking){
            e1 = Exam.Pass;
            //e1 = Exam(2); can also be used
        }
    }
```

The above function has an if conditional that checks if the e1 variable is in the Checking state, if the condition is True, then the state of e1 will be updated to Pass. However, if the condition is False then the if block will be skipped and there will be no change in the state of e1.

Exercise

a) Observe and Run the example given in section 1.4 and 1.5.

b) Using the example given in section 1.10 as a guide, make a function named Fail that updates the state of ‘e1’ to Fail, only if ‘e1’ is in the Checking state.

c) Make a function named Scholarship, that updates the state of ‘e1’ to Scholarship only if the previous state of e1 is ‘Pass’.

d) Make a function named idelete that resets the state of e1 to its default value. (See section 1.9)

2 Structs

2.1 Introduction:

- In Solidity, structs allow you to create more complex data types that consist of **multiple properties or fields.**
- By defining your own struct, you can group together related data and create a custom data type.
- Structs can be declared outside of a contract and imported into other contracts for reusability.
- They are commonly used to represent records or entities with a set of distinct attributes.

2.2 Syntax

To define a struct in Solidity, the following syntax is used:

```
struct <structure_name> {

  <data type> variable_1;

  <data type> variable_2;

// …                  }
```

The <structure_name> represents the name of the struct, while the variables within the curly braces denote the individual properties or fields of the struct.

Each variable is associated with a specific data type, which can be a primitive type (e.g., uint256, string) or a reference type (e.g., another struct, array, or mapping).

2.3 Practical Example

Once a struct is defined, you can create variables of that struct type to store and manipulate data. For example:

```
struct Student {

  uint256 id;

  uint256 score;    }

Student student; // Declaration of a student variable of type
Student
```

2.4 Methods to Create Struct

To assign values to a struct, there are several methods available:

Method 1: Creating a storage struct reference within a function:

```
function initStudent1() public {

Student storage _student = student; // Assign a copy of the
student struct

  _student.id = 11;

  _student.score = 100;            }

}
```

Method 2: Directly referring to the struct of a state variable:

```
function initStudent2() public {

  student.id = 1;

  student.score = 80;

}
```

Method 3: Using a struct constructor:

```
function initStudent3() public {

  student = Student(3, 90);

}
```

Method 4: Assigning values using key-value pairs:

```
function initStudent4() public {

  student = Student({id: 4, score: 60});

}
```

Each method provides a way to assign values to the individual fields of the struct

EXERCISE

1) Deploy, observe and explain the following smart contract.
2) 3 ways have been used here to initialize struct here. Highlight them.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;

  struct Voter {

        string Name;

        address addr;

        bool hasVoted;

    }

contract VotingSystem {

     // An array of 'Voter' structs

    Voter[] public voters;


    function create1(string memory _Name, address _addr)
public {

        voters.push(Voter(_Name, _addr, false));

    }
function create2(string memory _Name, address _addr) public {

        voters.push(Voter({Name: _Name, addr:_addr, hasVoted:
false}));

    }

    function create3(string memory _Name, address _addr)
public {

        Voter memory voter;

        voter.Name = _Name;

        voter.addr = _addr;

        // Voter.hasVoted initialized to false

        voters.push(voter);

    }

    function get(uint _index) public view returns (string
memory Name, address addr, bool hasVoted) {

        Voter storage voter = voters[_index];

        return (voter.Name, voter.addr,voter.hasVoted);

    }


    // update hasVoted

    function togglehasVoted(uint _index) public {

        Voter storage voter = voters[_index];

        voter.hasVoted = !voter.hasVoted;

    }

}
```

EXERCISE

Create a contract that defines a **BankAccount** struct, containing ‘accountID’ (uint256), ‘balance’ (uint256), and ‘owner’ (address) properties.

Create four different functions that assign values to struct in four different forms as discussed in **Section 2.4.** Also create a getter function to retrieve your stored values.

3 Structs vs Enums

Structs and enums are both powerful features in programming languages like Solidity, but they serve different purposes.

1) **Purpose:** Structs allow you to define custom data types with multiple properties or fields, making them useful for grouping related data and representing complex entities. Enums, on the other hand, restrict a variable to a predefined set of values, providing a way to represent and enforce a finite set of options or choices.

2) **Usage**: Structs are commonly used for modeling records or entities, such as representing a student with properties like name, age, and grade. Enums, on the other hand, are often used to define a set of named values, like representing the different states of an order (e.g., "pending," "processing," "completed").

3) **Composition:** Structs can contain multiple variables of different data types, allowing for the creation of more complex data structures. Enums, on the other hand, are typically composed of a single variable restricted to a limited set of named values.

In summary, while structs enable the creation of custom data types with multiple properties, enums provide a means to restrict variables to a predefined set of named values.

They serve different purposes and are used in different contexts within programming languages like Solidity.

[Note: Hands on exercise relating to Struct vs Enum will be covered in next lab]