

Name: Palak Chanchlani

Class: D20A

Roll No: 12

Experiment 4

Aim: Hands-on Solidity Programming Assignments for creating Smart Contracts

Theory

1. Introduction to Solidity and Smart Contracts

Solidity is a contract-oriented programming language designed for Ethereum. It is high-level, statically typed, and strongly influenced by languages like JavaScript, Python, and C++. Smart contracts written in Solidity allow developers to encode business logic directly on the blockchain, ensuring transparency, immutability, and automation.

Primitive Data Types:

- **uint / int:** Represent unsigned and signed integers of different bit sizes. For example, uint256 is widely used in token contracts to store balances, while int128 may be used in mathematical models.
- **bool:** Stores logical values (true or false). Often used in conditions such as verifying ownership or validating transactions.
- **address:** Holds Ethereum account or contract addresses (20 bytes). This type is essential for identifying participants and enabling contract-to-contract communication.
- **string / bytes:** Used for textual and binary data. Strings are suitable for names or metadata, while bytes are efficient for raw data like cryptographic hashes.

Variable Categories:

- **State Variables:** Permanently stored on-chain. Example: total supply of a token.
- **Local Variables:** Exist only during function execution and vanish afterward.
- **Global Variables:** Predefined values such as msg.sender, msg.value, and block.timestamp that provide transaction context.

Function Types:

- **pure:** Functions that neither read nor modify blockchain state. Example: a function that calculates square of a number.
- **view:** Functions that can read state variables but cannot alter them. Example: fetching a stored balance.

2. Inputs and Outputs in Functions

Functions in Solidity can accept parameters and return values, making contracts interactive.

- **Inputs:** Allow external users or contracts to pass data. Example: a deposit function accepting Ether.
- **Outputs:** Provide results back to the caller. Example: a withdrawal function returning a boolean.
- **Named Returns:** Improve readability and debugging by explicitly naming return variables.

3. Visibility, Modifiers, and Constructors

Visibility:

- **public:** Accessible internally and externally.
- **private:** Restricted to the same contract.
- **internal:** Available within the contract and derived contracts.
- **external:** Callable only from outside.

Modifiers: Reusable logic that alters function behaviour.

Example:

```
modifier onlyOwner {  
    require(msg.sender == owner, "Not authorized");  
}
```

Constructor: Special function executed once during deployment. Typically used to initialize critical variables, such as setting the deployer as the contract owner.

4. Control Flow

Solidity supports conditional and iterative structures:

- **if-else:** Enables decision-making, such as checking sufficient balance before transfer.
- **Loops (for, while, do-while):** Allow repetitive tasks like iterating through arrays. Must be used cautiously to avoid excessive gas consumption.

5. Data Structures

- **Arrays:** Ordered collections of elements. Example: storing participant addresses.
- **Mappings:** Key-value pairs for fast lookups. Example: mapping(address => uint) for balances.
- **Structs:** Group related attributes.

Example:

```
struct Player {  
    string name;  
    uint score;  
}
```

- **Enums:** Define a set of constant values for clarity. Example:
`enum Status { Pending, Active, Closed }`

6. Data Locations

Solidity defines three primary data locations:

- **storage:** Permanent, on-chain data (state variables).
- **memory:** Temporary data during function execution.
- **calldata:** Non-modifiable, efficient storage for external function inputs.

Efficient use of data locations reduces gas costs and improves performance.

7. Transactions: Ether, Wei, Gas, and Transfers

Ether and Wei: Ether is Ethereum's native currency. 1 Ether = (10^{18}) Wei. This precision is vital for financial transactions.

Gas and Gas Price:

- Gas represents computational effort.
- Gas price determines transaction speed and cost. Higher gas prices incentivize miners to process transactions faster.

Sending Transactions:

- **transfer():** Safe, limited gas forwarding.
- **send():** Returns a boolean indicating success or failure.
- **call():** Flexible and widely used in modern Solidity development.

Output -

1.introduction.sol

The screenshot shows the LEARNETH IDE interface. On the left, the '1. Introduction' tutorial is displayed, explaining the purpose of the counter contract and the roles of various keywords like `uint`, `public`, `contract`, `view`, `inc`, and `dec`. On the right, the 'introduction.sol' code editor shows the Solidity code for the `MyContract`, which includes a license, version pragma, a state variable `name`, and a constructor that sets `name` to 'Alice'.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract MyContract {
5     string public name;
6
7     constructor() {
8         name = "Alice";
9     }
10 }
11
```

2. basicSyntax.sol

The screenshot shows the LEARNETH IDE interface for the '2. Basic Syntax' tutorial. The tutorial on the left explains how to create a simple smart contract that holds the string 'Hello World!'. It details the use of `pragma` for the Solidity version, `contract` for defining the contract, and `string` for the state variable `greet`. The code editor on the right shows the corresponding Solidity code for `basicSyntax.sol`.

```
1 pragma solidity ^0.8.3;
2
3 contract MyContract {
4     string public name = "Alice";
5 }
6
```

3. primitiveDataTypes.sol

LEARNETH

Tutorials list

Syllabus

3. Primitive Data Types

3 / 19

3. Primitive Data Types

In this section, we will show you Solidity's primitive data types, how to declare them, and their characteristics.

bool

You can declare data a boolean type by using the keyword 'bool'. Booleans can either have the value `true` or `false`.

uint

We use the keywords `uint` and `uint8` to `uint256` to declare an *unsigned integer type* (they don't have a sign, unlike -12, for example). Uints are integers that are positive or zero and range from 8 bits to 256 bits. The type `uint` is the same as `uint256`.

int

We use the keywords `int` and `int8` to `int256` to declare an integer type. Integers can be positive, negative, or zero and range from 8 bits to 256 bits. The type `int` is the same as `int256`.

address

Variables of the type `address` hold a 20-byte value, which is the size of an Ethereum address. There is also a special kind of Ethereum address, `address`.

Compile

introduction.sol

basicSyntax.sol 1

primitiveDataTypes.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.31;
3
4 contract Primitives {
5     address public addr = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;
6
7     // New public address (different from addr)
8     address public newAddr = 0x4B0897b0513fDc7C54186d9D7E929C4e5364D2d8;
9
10    // Public negative number
11    int public neg = -10;
12
13    // Smallest uint type with smallest value
14    uint8 public newU = 0;
15 }
16
```

4. variables.sol

LEARNETH

Tutorials list

Syllabus

4. Variables

4 / 19

Global variables, also called *special variables*, exist in the global namespace. They don't need to be declared but can be accessed from within your contract. Global Variables are used to retrieve information about the blockchain, particular addresses, contracts, and transactions.

In this example, we use `block.timestamp` (line 14) to get a Unix timestamp of when the current block was generated and `msg.sender` (line 15) to get the caller of the contract function's address.

A list of all Global Variables is available in the [Solidity documentation](#).

Watch video tutorials on [State Variables](#), [Local Variables](#), and [Global Variables](#).

★ Assignment

1. Create a new public state variable called `blockNumber`.
2. Inside the function `doSomething()`, assign the value of the current block number to the state variable `blockNumber`.

Tip: Look into the global variables section of the Solidity documentation to find out how to read the current block number.

Check Answer

Show answer

Next

Well done! No errors.

Compiled

introduction.sol

basicSyntax.sol 1

primitiveDataTypes.sol

variables.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.31;
3
4 contract Variables {
5
6     uint public blockNumber;
7
8     function doSomething() public { 22255 gas
9         blockNumber = block.number;
10     }
11 }
12
```

5.1 readAndWrite.sol

LEARNETH

Tutorials list

Syllabus

5.1 Functions - Reading and Writing to a State Variable

5 / 19

5.1 Functions - Reading and Writing to a State Variable

This section will give a short introduction to functions and teach you how to use them to read from and write to a state variable.

As in other languages, we use functions in Solidity to create modular, reusable code. However, Solidity functions have some particularities.

Solidity functions can be split into two types:

1. Functions that modify the state of the blockchain, like writing to a state variable. In this contract, the `set` function (line 9) changes the state variable `num`.
2. Functions that don't modify the state of the blockchain. These functions are marked `view` or `pure`. For example, in this contract, the `get` function (line 14) marked `view` that only returns `num` does not change the state.

To define a function, use the `function` keyword followed by a unique name.

If the function takes inputs like our `set` function (line 9), you must specify the parameter types and names. A common convention is to use an underscore as a prefix for the parameter name to distinguish them from state variables.

You can then set the visibility of a function and declare them `view` or `pure` as we

Compile

variables.sol

.prettierrc.json

readAndWrite.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.31;
3
4 contract SimpleStorage {
5
6     bool public b = true;
7
8     function get_b() public view returns (bool) { 2472 gas
9         return b;
10     }
11 }
12
```

5.2 viewAndPure.sol

LEARNETH

Tutorials list

Syllabus

5.2 Functions - View and Pure

6 / 19

5.2 Functions - View and Pure

This section will look into the types of functions that don't modify the state of the blockchain: *view* and *pure* functions.

View Functions

View functions promise to not modify the state.

"The following statements are considered modifying the state:

1. Writing to state variables.
2. Emitting events.
3. Creating other contracts.
4. Using `selfdestruct`.
5. Sending Ether via calls.
6. Calling any function not marked `view` or `pure`.
7. Using low-level calls.
8. Using inline assembly that contains certain opcodes."

From the [Solidity documentation](#).

You can declare a view function using the keyword `view`. In this contract, `addToX` (line 8) is a view function. This function takes the parameter `y` and returns the

Compile

variables.sol

.prettierrc.json

readAndWrite.sol

viewAndPure.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract ViewAndPure {
5     uint public x = 1;
6
7     // Promise not to modify the state.
8     function addToX(uint y) public view returns (uint) { infinite gas
9         return x + y;
10     }
11
12     // Promise not to modify or read from the state.
13     function add(uint i, uint j) public pure returns (uint) { infinite gas
14         return i + j;
15     }
16
17     function addToX2(uint y) public { infinite gas
18         x = x + y;
19     }
20 }
```

5.3 modifiersAndConstructor.sol

LEARNETH

Tutorials list

Syllabus

5.3 Functions - Modifiers and Constructors

7 / 19

A constructor function is executed upon the creation of a contract. You can use it to run contract initialization code. The constructor can have parameters and is especially useful when you don't know certain initialization values before the deployment of the contract.

You declare a constructor using the `constructor` keyword. The constructor in this contract (line 11) sets the initial value of the owner variable upon the creation of the contract.

Watch a video tutorial on Function Modifiers.

★ Assignment

- Create a new function, `increaseX` in the contract. The function should take an input parameter of type `uint` and increase the value of the variable `x` by the value of the input parameter.
- Make sure that `x` can only be increased.
- The body of the function `increaseX` should be empty.

Tip: Use modifiers.

Check Answer

Show answer

Next

Well done! No errors.

Compiled

pure.sol

viewAndPure_answer.sol

modifiersAndConstructors.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract FunctionModifier {
5     // We will use these variables to demonstrate how to use
6     // modifiers.
7     address public owner;
8     uint public x = 10;
9     bool public locked;
10
11     constructor() {
12         // Set the transaction sender as the owner of the contract.
13         owner = msg.sender;
14     }
15
16     // Modifier to check that the caller is the owner of
17     // the contract.
18     modifier onlyOwner() {
19         require(msg.sender == owner, "Not owner");
20         // Underscore is a special character only used inside
21         // a function modifier and it tells Solidity to
22         // execute the rest of the code.
23         _;
24     }
25
26     // Modifiers can take inputs. This modifier checks that the
27     // address passed in is not the zero address.
28     modifier validAddress(address _addr) {
29         require(_addr != address(0), "Not valid address");
30         _;
31     }
32 }
```

5.4 inputsAndOutputs.sol

LEARNETH

Tutorials list

Syllabus

5.4 Functions - Inputs and Outputs

8 / 19

5.4 Functions - Inputs and Outputs

In this section, we will learn more about the inputs and outputs of functions.

Multiple named Outputs

Functions can return multiple values that can be named and assigned to their name.

The `returnMany` function (line 6) shows how to return multiple values. You will often return multiple values. It could be a function that collects outputs of various functions and returns them in a single function call for example.

The `named` function (line 19) shows how to name return values. Naming return values helps with the readability of your contracts. Named return values make it easier to keep track of the values and the order in which they are returned. You can also assign values to a name.

The `assigned` function (line 33) shows how to assign values to a name. When you assign values to a name you can omit (leave out) the return statement and return them individually.

Deconstructing Assignments

You can use deconstructing assignments to unpack values into distinct variables.

The `destructingAssignments` function (line 49) assigns the values of the `returnMany`

Compiled

undPure.sol

viewAndPure_answer.sol

modifiersAndConstructors.sol

inputsAndOutputs.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Function {
5     // Functions can return multiple values.
6     function returnMany() infinite gas
7     {
8         public
9         pure
10         returns (
11             uint,
12             bool,
13             uint
14         )
15     }
16
17     {
18         return (1, true, 2);
19     }
20
21     // Return values can be named.
22     function named() infinite gas
23     {
24         public
25         pure
26         returns (
27             uint x,
28             bool b,
29             uint y
30         )
31     }
32
33     {
34         return (1, true, 2);
35     }
36
37     // Return values can be assigned to their name.
38     // In this case the return statement can be omitted.
```

6. visibility.sol

LEARNETH

Tutorials list

Syllabus

6. Visibility

9 / 19

6. Visibility

The `visibility` specifier is used to control who has access to functions and state variables.

There are four types of visibilities: `external`, `public`, `internal`, and `private`.

They regulate if functions and state variables can be called from inside the contract, from contracts that derive from the contract (child contracts), or from other contracts and transactions.

private

- Can be called from inside the contract

internal

- Can be called from inside the contract
- Can be called from a child contract

public

- Can be called from inside the contract
- Can be called from a child contract
- Can be called from other contracts or transactions

external

Compiled

viewAndPure_answer.sol

modifiersAndConstructors.sol

inputsAndOutputs.sol

visibility.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Base {
5     // Private function can only be called
6     // - inside this contract
7     // Contracts that inherit this contract cannot call this function.
8     function privateFunc() private pure returns (string memory) {
9         return "private function called";
10    }
11
12    function testPrivateFunc() public pure returns (string memory) {
13        return privateFunc();
14    }
15
16    // Internal function can be called
17    // - inside this contract
18    // - inside contracts that inherit this contract
19    function internalFunc() internal pure returns (string memory) {
20        return "internal function called";
21    }
22
23    function testInternalFunc() public pure virtual returns (string memory) {
24        return internalFunc();
25    }
26
27    // Public functions can be called
28    // - inside this contract
29    // - inside contracts that inherit this contract
30    // - by other contracts and accounts
31    function publicFunc() public pure returns (string memory) {
32        return "public function called";
33    }
34}
```

7.1 ControlFlow.sol

LEARNETH

Tutorials list

Syllabus

7.1 Control Flow - If/Else

10 / 19

7.1 Control Flow - If/Else

In this contract, the `foo` function uses the `if/else` statement (line 10) to return `0` if none of the other conditions are met.

else if

With the `else if` statement we can combine several conditions.

If the first condition (line 6) of the `foo` function is not met, but the condition of the `else if` statement (line 8) becomes true, the function returns `1`.

[Watch a video tutorial on the If/Else statement.](#)

★ **Assignment**

Create a new function called `evenCheck` in the `IfElse` contract:

- That takes in a `uint` as an argument.
- The function returns `true` if the argument is even, and `false` if the argument is odd.
- Use a ternary operator to return the result of the `evenCheck` function.

Tip: The modulo (%) operator produces the remainder of an integer division.

Check Answer

Show answer

Next

Well done! No errors.

Compiled

er.sol

modifiersAndConstructors.sol

inputsAndOutputs.sol

visibility.sol

IfElse.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract IfElse {
5     function foo(uint x) public pure returns (uint) {
6         if (x < 10) {
7             return 0;
8         } else if (x < 20) {
9             return 1;
10        } else {
11            return 2;
12        }
13    }
14
15    function ternary(uint _x) public pure returns (uint) {
16        // if (_x < 10) {
17        //     return 1;
18        // }
19        // return 2;
20
21        // shorthand way to write if / else statement
22        return _x < 10 ? 1 : 2;
23    }
24
25    function evenCheck(uint y) public pure returns (bool) {
26        return y % 2 == 0 ? true : false;
27    }
28 }
```

7.2 loops.sol

LEARNETH

Tutorials list

Syllabus

7.2 Control Flow - Loops

11 / 19

7.2 Control Flow - Loops

Solidity supports iterative control flow statements that allow contracts to execute code repeatedly.

Solidity differentiates between three types of loops: `for`, `while`, and `do while` loops.

for

Generally, `for` loops (line 7) are great if you know how many times you want to execute a certain block of code. In solidity, you should specify this amount to avoid transactions running out of gas and failing if the amount of iterations is too high.

while

If you don't know how many times you want to execute the code but want to break the loop based on a condition, you can use a `while` loop (line 20). Loops are seldom used in Solidity since transactions might run out of gas and fail if there is no limit to the number of iterations that can occur.

do while

The `do while` loop is a special kind of while loop where you can ensure the code is executed at least once, before checking on the condition.

Compiled

differsAndConstructors.sol

inputsAndOutputs.sol

visibility.sol

ifElse.sol

loops

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Loop {
5     uint public count;
6     function loop() public {
7         // for loop
8         for (uint i = 0; i < 10; i++) {
9             if (i == 5) {
10                 // Skip to next iteration with continue
11                 continue;
12             }
13             if (i == 5) {
14                 // Exit loop with break
15                 break;
16             }
17             count++;
18         }
19
20         // while loop
21         uint j;
22         while (j < 10) {
23             j++;
24         }
25     }
26 }
27
```

Explain contract

8.1 arrays.sol

LEARNETH

Tutorials list

Syllabus

8.1 Data Structures - Arrays

12 / 19

8.1 Data Structures - Arrays

In the next sections, we will look into the data structures that we can use to organize and store our data in Solidity.

Arrays, mappings and structs are all *reference types*. Unlike *value types* (e.g. *booleans* or *integers*) reference types don't store their value directly. Instead, they store the location where the value is being stored. Multiple reference type variables could reference the same location, and a change in one variable would affect the others, therefore they need to be handled carefully.

In Solidity, an array stores an ordered list of values of the same type that are indexed numerically.

There are two types of arrays, compile-time *fixed-size* and *dynamic arrays*. For fixed-size arrays, we need to declare the size of the array before it is compiled. The size of dynamic arrays can be changed after the contract has been compiled.

Declaring arrays

We declare a fixed-size array by providing its type, array size (as an integer in square brackets), visibility, and name (line 9).

We declare a dynamic array in the same manner. However, we don't provide an array size and leave the brackets empty (line 6).

Compiled

ctors.sol

inputsAndOutputs.sol

visibility.sol

ifElse.sol

loops.sol

arrays.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Array {
5     // Several ways to initialize an array
6     uint[] public arr;
7     uint[] public arr2 = [1, 2, 3];
8     // Fixed sized array, all elements initialize to 0
9     uint[10] public myFixedSizeArr;
10    uint[3] public arr3 = [0, 1, 2];
11
12    function get(uint i) public view returns (uint) {
13        return arr[i];
14    }
15
16    // Solidity can return the entire array.
17    // But this function should be avoided for
18    // arrays that can grow indefinitely in length.
19    function getArr() public view returns (uint[3] memory) {
20        return arr3;
21    }
22
23    function push(uint i) public {
24        // Append to array
25        // This will increase the array length by 1.
26        arr.push(i);
27    }
28
29    function pop() public {
30        // Remove last element from array
31        // This will decrease the array length by 1
32        arr.pop();
33    }
34 }
```

8.2 mapping.sol

LEARNETH

Tutorials list

Syllabus

8.2 Data Structures - Mappings

13 / 19

8.2 Data Structures - Mappings

In Solidity, *mappings* are a collection of key types and corresponding value type pairs.

The biggest difference between a mapping and an array is that you can't iterate over mappings. If we don't know a key we won't be able to access its value. If we need to know all of our data or iterate over it, we should use an array.

If we want to retrieve a value based on a known key we can use a mapping (e.g. addresses are often used as keys). Looking up values with a mapping is easier and cheaper than iterating over arrays. If arrays become too large, the gas cost of iterating over it could become too high and cause the transaction to fail.

We could also store the keys of a mapping in an array that we can iterate over.

Creating mappings

Mappings are declared with the syntax `mapping(KeyType => ValueType) VariableName`. The key type can be any built-in value type or any contract, but not a reference type. The value type can be of any type.

In this contract, we are creating the public mapping `balances` (line 6) that associates the key type `address` with the value type `uint`.

Accessing values

Compiled

visibility.sol ifElse.sol loops.sol arrays.sol mappings.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Mapping {
5     // Mapping from address to uint
6     mapping(address => uint) public balances;
7
8     function get(address _addr) public view returns (uint) {
9         // Mapping always returns a value.
10        // If the value was never set, it will return the default value.
11        return balances[_addr];
12    }
13
14    function set(address _addr) public {
15        // Update the value at this address
16        balances[_addr] = _addr.balance;
17    }
18
19    function remove(address _addr) public {
20        // Reset the value to the default value.
21        delete balances[_addr];
22    }
23 }
24
25 contract NestedMapping {
26     // Nested mapping (mapping from address to another mapping)
27     mapping(address => mapping(uint => bool)) public nested;
28
29     function get(address _addr1, uint _i) public view returns (bool) {
30         // You can get values from a nested mapping
31         // even when it is not initialized
32         return nested[_addr1][_i];
33     }
34 }
```

8.3 structs.sol

LEARNETH

Tutorials list

Syllabus

8.3 Data Structures - Structs

14 / 19

8.3 Data Structures - Structs

In Solidity, we can define custom data types in the form of *structs*. Structs are a collection of variables that can consist of different data types.

Defining structs

We define a struct using the `struct` keyword and a name (line 5). Inside curly braces, we can define our struct's members, which consist of the variable names and their data types.

Initializing structs

There are different ways to initialize a struct.

Positional parameters: We can provide the name of the struct and the values of its members as parameters in parentheses (line 16).

Key-value mapping: We provide the name of the struct and the keys and values as a mapping inside curly braces (line 19).

Initialize and update a struct: We initialize an empty struct first and then update its member by assigning it a new value (line 23).

Accessing structs

To access a member of a struct we can use the dot operator (line 33).

Compiled

visibility.sol ifElse.sol loops.sol arrays.sol mappings.sol structs.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Todos {
5     struct Todo {
6         string text;
7         bool completed;
8     }
9
10    // An array of 'Todo' structs
11    Todo[] public todos;
12
13    function create(string memory _text) public {
14        // 3 ways to initialize a struct
15        // - calling it like a function
16        todos.push(Todo(_text, false));
17
18        // key value mapping
19        todos.push(Todo({text: _text, completed: false}));
20
21        // initialize an empty struct and then update it
22        Todo memory todo;
23        todo.text = _text;
24        // todo.completed initialized to false
25
26        todos.push(todo);
27    }
28
29    // Solidity automatically created a getter for 'todos' so
30    // you don't actually need this function.
31    function get(uint _index) public view returns (string memory text, bool completed) {
32        Todo storage todo = todos[_index];
33    }
34 }
```

8.4 enums.sol

LEARNETH

Tutorials list

Syllabus

8.4 Data Structures - Enums

15 / 19

8.4 Data Structures - Enums

In Solidity *enums* are custom data types consisting of a limited set of constant values. We use enums when our variables should only get assigned a value from a predefined set of values.

In this contract, the state variable `status` can get assigned a value from the limited set of provided values of the enum `Status` representing the various states of a shipping status.

Defining enums

We define an enum with the `enum` keyword, followed by the name of the custom type we want to create (line 6). Inside the curly braces, we define all available members of the enum.

Initializing an enum variable

We can initialize a new variable of an enum type by providing the name of the enum, the visibility, and the name of the variable (line 16). Upon its initialization, the variable will be assigned the value of the first member of the enum, in this case, `Pending` (line 7).

Even though enum members are named when you define them, they are stored as unsigned integers, not strings. They are numbered in the order that they were defined, the first member starting at 0. The initial value of `status`, in this case, is 0.

Compiled

enums.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Enum {
5     // Enum representing shipping status
6     enum Status {
7         Pending,
8         Shipped,
9         Accepted,
10        Rejected,
11        Canceled
12    }
13
14    enum Size {
15        S,
16        M,
17        L
18    }
19
20    // Default value is the first element listed in
21    // definition of the type, in this case "Pending"
22    Status public status;
23    Size public sizes;
24
25    function get() public view returns (Status) { 2605 gas
26        return status;
27    }
28
29    function getSize() public view returns (Size) { 2633 gas
30        return sizes;
31    }
32}
```

9. dataLocations.sol

LEARNETH

Tutorials list

Syllabus

9. Data Locations

16 / 19

9. Data Locations

The values of variables in Solidity can be stored in different data locations: *memory*, *storage*, and *calldata*.

As we have discussed before, variables of the value type store an independent copy of a value, while variables of the reference type (array, struct, mapping) only store the location (reference) of the value.

If we use a reference type in a function, we have to specify in which data location their values are stored. The price for the execution of the function is influenced by the data location; creating copies from reference types costs gas.

Storage

Values stored in *storage* are stored permanently on the blockchain and, therefore, are expensive to use.

In this contract, the state variables `arr`, `map`, and `myStructs` (lines 5, 6, and 10) are stored in storage. State variables are always stored in storage.

Memory

Values stored in *memory* are only stored temporarily and are not on the blockchain. They only exist during the execution of an external function and are discarded afterward. They are cheaper to use than values stored in *storage*.

Compiled

dataLocations.sol 2

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract DataLocations {
5     uint[] public arr;
6     mapping(uint => address) map;
7     struct MyStruct {
8         uint foo;
9     }
10    mapping(uint => MyStruct) public myStructs;
11
12    function f() public returns (MyStruct memory, MyStruct memory, MyStruct memory) { infinite gas
13        // call _f with state variables
14        _f(arr, map, myStructs[1]);
15        // get a struct from a mapping
16        MyStruct storage myStruct = myStructs[1];
17        myStruct.foo = 4;
18        // create a struct in memory
19        MyStruct memory myMemStruct = MyStruct(0);
20        MyStruct memory myMemStruct2 = myMemStruct;
21        myMemStruct2.foo = 1;
22
23        MyStruct memory myMemStruct3 = myStruct;
24        myMemStruct3.foo = 3;
25        return (myStruct, myMemStruct2, myMemStruct3);
26    }
27
28    function _f(
29        uint[] storage _arr,
30        mapping(uint => address) storage _map,
31        MyStruct storage _myStruct
32    ) internal {
```

10.1 etherAndWei.sol

LEARNETH

Tutorials list

Syllabus

10.1 Transactions - Ether and Wei

17 / 19

number.

`wei`

Wei is the smallest subunit of *Ether*, named after the cryptographer *Wei Dai*. *Ether* numbers without a suffix are treated as `wei` (line 7).

`gwei`

One `gwei` (giga-wei) is equal to 1,000,000,000 (10^9) `wei`.

`ether`

One `ether` is equal to 1,000,000,000,000,000 (10^{18}) `wei` (line 11).

Watch a video tutorial on Ether and Wei.

★ Assignment

1. Create a `public uint` called `oneGwei` and set it to 1 `gwei`.
2. Create a `public bool` called `isOneGwei` and set it to the result of a comparison operation between 1 `gwei` and 10^9 .

Tip: Look at how this is written for `gwei` and `ether` in the contract.

Check Answer

Show answer

Next

Well done! No errors.

Compiled

oops.sol arrays.sol mappings.sol structs.sol enums.sol etherAndWei.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract EtherUnits {
5     uint public oneWei = 1 wei;
6     // 1 wei is equal to 1
7     bool public isOneWei = 1 wei == 1;
8
9     uint public oneEther = 1 ether;
10    // 1 ether is equal to 10^18 wei
11    bool public isOneEther = 1 ether == 1e18;
12
13    uint public oneGwei = 1 gwei;
14    // 1 ether is equal to 10^9 wei
15    bool public isOneGwei = 1 gwei == 1e9;
16 }
```

10.2 gasAndGasPrice.sol

LEARNETH

Tutorials list

Syllabus

10.2 Transactions - Gas and Gas Price

18 / 19

Gas prices are denoted in `gwei`.

Gas limit

When sending a transaction, the sender specifies the maximum amount of gas that they are willing to pay for. If they set the limit too low, their transaction can run out of gas before being completed, reverting any changes being made. In this case, the gas was consumed and can't be refunded.

Learn more about gas on ethereum.org.

Watch a video tutorial on Gas and Gas Price.

★ Assignment

Create a new `public` state variable in the `Gas` contract called `cost` of the type `uint`. Store the value of the gas cost for deploying the contract in the new variable, including the cost for the value you are storing.

Tip: You can check in the Remix terminal the details of a transaction, including the gas cost. You can also use the Remix plugin *Gas Profiler* to check for the gas cost of transactions.

Check Answer

Show answer

Next

Well done! No errors.

Compile

ps.sol arrays.sol mappings.sol structs.sol enums.sol gasAndGasPrice.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Gas {
5     uint public i = 0;
6     uint public cost = 170367;
7
8     // Using up all of the gas that you send causes your transaction to fail.
9     // State changes are undone.
10    // Gas spent are not refunded.
11    function forever() public ( infinite gas
12        // Here we run a loop until all of the gas are spent
13        // and the transaction falls
14        while (true) {
15            i += 1;
16        }
17    }
18 }
```

10.3 sendingEther.sol

The screenshot displays the LEARNETH IDE interface. On the left, a sidebar shows the 'Tutorials list' with '10.3 Transactions - Sending Ether' selected. Below the title, there is explanatory text about parameter types for `sendViaTransfer` and `sendViaSend`, a video tutorial link, and an 'Assignment' section with four steps: 1. Create a contract called `Charity`. 2. Add a public state variable called `owner` of type `address`. 3. Create a donate function that is public and payable without parameters or function code. 4. Create a withdraw function that is public and sends the total balance to the `owner` address. At the bottom of the sidebar are buttons for 'Check Answer', 'Show answer', and 'Next', along with a green status bar saying 'Well done! No errors.'

The main editor on the right shows the Solidity code for `sendingEther.sol`. The code is as follows:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract ReceiveEther {
5     /*
6      * Which function is called, fallback() or receive()?
7      *
8      * send Ether
9      *
10     * msg.data is empty?
11     * / \
12     * yes no
13     * / \
14     * receive() exists? fallback()
15     * / \
16     * yes no
17     * / \
18     * receive() fallback()
19     */
20
21     // Function to receive Ether. msg.data must be empty
22     receive() external payable {} // undefined gas
23
24     // Fallback function is called when msg.data is not empty
25     fallback() external payable {} // undefined gas
26
27     function getBalance() public view returns (uint) { // 312 gas
28         return address(this).balance;
29     }
30 }
31
32 contract SendEther {
```

Conclusion

This experiment provides practical exposure to Solidity programming by covering essential blockchain concepts such as data types, variables, functions, control structures, data storage, and transaction handling. By mastering these fundamentals, developers can design secure, efficient, and scalable smart contracts that form the backbone of decentralized applications.