

## BLOCKCHAIN EXP 4

**Aim** - Hands on Solidity Programming Assignments for creating Smart Contracts.

### Theory -

#### 1. Primitive Data Types, Variables, Functions – pure, view

In Solidity, primitive data types form the foundation of smart contract development. Commonly used types include:

- **uint / int**: unsigned and signed integers of different sizes (e.g., uint256, int128).
- **bool**: represents logical values (true or false).
- **address**: holds a 20-byte Ethereum account address, often used for storing user accounts or contract addresses.
- **bytes / string**: store binary data or textual data.

Variables in Solidity can be **state variables** (stored on the blockchain permanently), **local variables** (temporary, created during function execution), or **global variables** (special predefined variables such as msg.sender, msg.value, and block.timestamp).

Functions allow execution of contract logic. Special types of functions include:

- **pure**: cannot read or modify blockchain state; they work only with inputs and internal computations.
- **view**: can read state variables but cannot alter them. This classification helps optimize gas usage and enforces function integrity.

#### 2. Inputs and Outputs to Functions

Functions in Solidity can accept input arguments and return one or more output values. Inputs enable users or other contracts to pass data into the contract, while outputs make it possible to return results after computation. For example, a function can accept an amount in Ether and return whether the transfer was successful. Solidity also allows named return variables, which improve readability and debugging.

#### 3. Visibility, Modifiers and Constructors

- **Function Visibility** defines who can access a function:
  - **public**: available both inside and outside the contract.
  - **private**: only accessible within the same contract.
  - **internal**: accessible within the contract and its child contracts.
  - **external**: can be called only by external accounts or other contracts.

- **Modifiers** are reusable code blocks that change the behavior of functions. They are often used for access control, such as restricting sensitive functions to the contract owner (`onlyOwner`).
- **Constructors** are special functions executed only once during contract deployment. They initialize important values, such as setting the deploying account as the owner of the contract.

### 3. Control Flow: if-else, loops

Control flow in Solidity is similar to traditional programming languages:

- **if-else** allows conditional decision-making in contract logic, e.g., checking if a balance is sufficient before transferring funds.
- **Loops** (for, while, do-while) enable repeated execution of code. For example, iterating through an array of users. However, loops must be used carefully, as excessive iterations increase gas consumption, potentially making the contract expensive to execute.

### 5. Data Structures: Arrays, Mappings, Structs, Enums

- **Arrays**: Can be fixed or dynamic and are used to store ordered lists of elements. Example: an array of addresses for registered users.
- **Mappings**: Key-value pairs that allow quick lookups. Example: `mapping(address => uint)` for storing balances. Unlike arrays, mappings do not support iteration.
- **Structs**: Allow grouping of related properties into a single data type, such as creating a struct `Player { string name; uint score; }`.
- **Enums**: Used to define a set of predefined constants, making code more readable. Example: `enum Status { Pending, Active, Closed }`.

### 6. Data Locations

Solidity uses three primary data locations for storing variables:

- **storage**: Data stored permanently on the blockchain. Examples: state variables.
- **memory**: Temporary data storage that exists only while a function is executing. Used for local variables and function inputs.
- **calldata**: A non-modifiable and non-persistent location used for external function parameters. It is gas-efficient compared to memory. Understanding data locations is essential, as they directly impact gas costs and performance.

### 7. Transactions: Ether and Wei, Gas and Gas Price, Sending Transactions

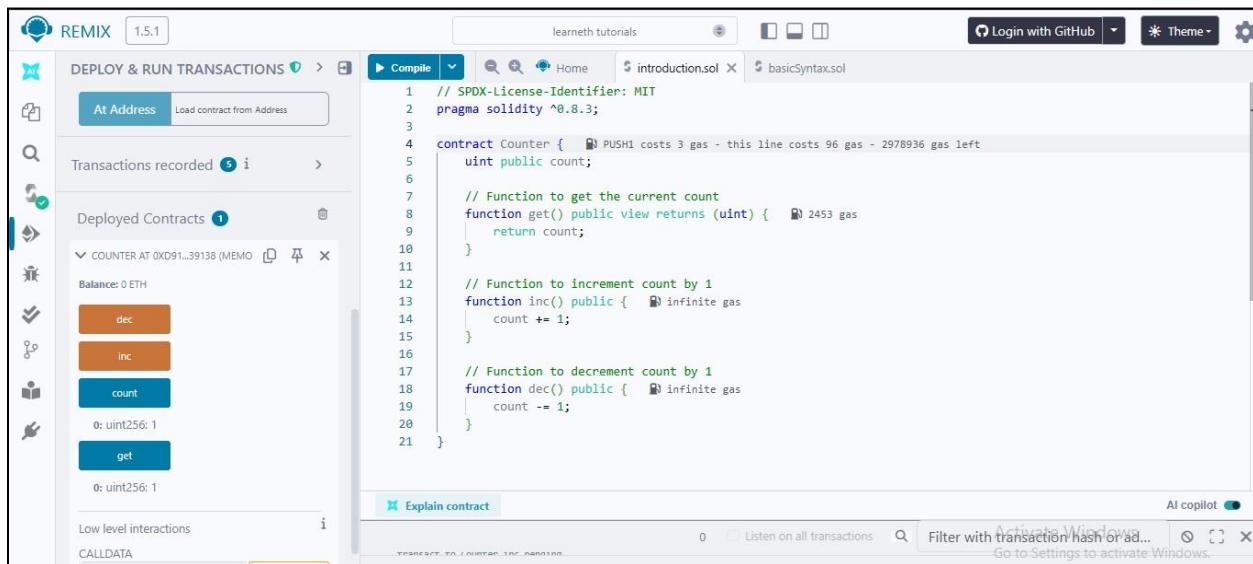
- **Ether and Wei**: Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether =  $10^{18}$  Wei). This ensures high precision in financial transactions.

- **Gas and Gas Price:** Every transaction consumes gas, which represents computational effort. The gas price determines how much Ether is paid per unit of gas. A higher gas price incentivizes miners to prioritize the transaction.
- **Sending Transactions:** Transactions are used for transferring Ether or interacting with contracts. Functions like transfer() and send() are commonly used, while call() provides more flexibility. Each transaction requires gas, making efficiency in contract design very important.

## Code & Output -

### Tutorial 1

#### 1. Get counter value



The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with various icons for deployment, running transactions, and interacting with contracts. In the center, the code editor displays the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Counter {
5     // PUSH1 costs 3 gas - this line costs 96 gas - 2978936 gas left
6     uint public count;
7
8     // Function to get the current count
9     function get() public view returns (uint) {
10         return count;
11     }
12
13     // Function to increment count by 1
14     function inc() public {
15         count += 1;
16     }
17
18     // Function to decrement count by 1
19     function dec() public {
20         count -= 1;
21     }
}

```

On the right side of the interface, there are tabs for 'introduction.sol' and 'basicSyntax.sol'. Below the tabs, there are buttons for 'Compile', 'Run', and 'Deploy'. Further down, there's a section for 'Transactions recorded' showing a single entry for 'COUNTER AT 0xD91...39138 (MEMO)' with a balance of '0 ETH'. There are also buttons for 'dec', 'inc', 'count', and 'get'. At the bottom, there are sections for 'Low level interactions' and 'CALLDATA'.

## 2. Increment counter value

[vm]	from:	0x5B3...eddC4	to:	Counter.inc()	0xd91...39138	value:	0	wei	data:	0x371...303c0	logs:	0	hash:	0xfce...8cdc3	Debug
status	1	Transaction mined and execution succeed													
transaction hash	0xfcce1001233a0cc437ac7e87c2db11f10da2fe0976a8ef4e98ee5c6c5228cdc3														
block hash	0xc8733a651aad2a98d67f80a95784141e7e550cd40388c50feefcd1145229228														
block number	6														
from	0x5B38Da6a701c568545dCfcB03Fc8875f56beddC4														
to	Counter.inc()	0xd9145CCE52D386f254917e481eB44e9943F39138													
transaction cost	26417	gas													

### 3. Decrement counter value

[vm]	from: 0x5B3...eddC4	to: Counter.dec()	0xd91...39138	value: 0	wei data: 0xb3b...cfa82	logs: 0	hash: 0x21d...52d6d	Debug
status	1	Transaction mined and execution succeed						
transaction hash	0x21daa184e0a457ef2f35508a0994a8fc0c2eac05b53ab95e6d96e1c2c4452d6d							
block hash	0xca425c3b6aa253d68e49726515232861814af84154eb74afe6c4683415457db8							
block number	7							
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4							
to	Counter.dec()	0xd9145CCE52D386f254917e481eB44e9943F39138						
transaction cost	26461	gas						

#### 4. Get count value

## Tutorial 2

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with tabs for 'DEPLOY & RUN TRANSACTIONS' and 'Deployed Contracts'. Under 'Deployed Contracts', several contracts are listed: 'COUNTER AT 0xD91...', 'HELLOWORLD AT 0xB27...', 'HELLOWORLD AT 0xCD6...', 'HELLOWORLD AT 0xAE0...', and 'MYCONTRACT AT 0x5E1...'. The 'MYCONTRACT' entry is expanded, showing its source code and deployment details. The source code for 'MyContract' is as follows:

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
pragma solidity ^0.8.3;

contract MyContract {
    string public name = "Alice";
}
```

Below the code, there's an 'Explain contract' section with tabs for 'Execution cost', 'Input', 'Output', 'Decoded input', 'Decoded output', and 'Logs'. The 'Execution cost' tab shows a gas cost of 3298. The 'Input' tab shows the value '0x00f...ddde03'. The 'Output' tab shows a large hex string representing the returned value. The 'Decoded input' and 'Decoded output' tabs both show an array containing a single object with key '0' and value 'string: Alice'. The 'Logs' tab is empty.

## Tutorial 3

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays deployed contracts: MYCONTRACT at 0x5E1...4EFF5 (MEMORY), PRIMITIVES AT 0X8C...24C73 (MEMORY), and VARIABLES AT 0XEC2...CF142 (MEMORY). The VARIABLES contract has a balance of 0 ETH. It contains the following code:

```

int public i = -123; // int is same as int256
address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;
// Default values
// Unassigned variables have a default value
bool public defaultBool; // false
uint public defaultUint; // 0
int public defaultInt; // 0
address public defaultAddr; // 0x0000000000000000000000000000000000000000
address public newAddr = 0x0000000000000000000000000000000000000000;
int public neg = -4;
uint8 public newU;
}

```

The right panel shows the contract's storage layout with variables: doSomething (uint256: 38), blockNumber (uint256: 123), num (uint256: 38), and text (string: Hello). The transaction history shows calls to Variables.text() and Variables.num().

## Tutorial 4

The screenshot shows the REMIX IDE interface. On the left, the sidebar displays deployed contracts: HELLOWORLD at 0XC06...99DF9 (MEM), HELLOWORLD at 0XA0E...96BBB (MEM), and MYCONTRACT at 0X5E1...4EFF5 (MEMORY). The MYCONTRACT contract has a balance of 0 ETH. It contains the following code:

```

contract Variables {
    // State variables are stored on the blockchain.
    string public text = "Hello";
    uint public num = 123;
    uint public blockNumber;

    function doSomething() public {
        // Local variables are not saved to the blockchain.
        uint i = 456;

        // Here are some global variables
        uint timestamp = block.timestamp; // Current block timestamp
        address sender = msg.sender; // address of the caller
        blockNumber = block.number;
    }
}

```

The right panel shows the contract's storage layout with variables: doSomething (uint256: 37), blockNumber (uint256: 123), num (uint256: 37), and text (string: Hello). The transaction history shows calls to Variables.num() and Variables.text().

## Tutorial 5

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract SimpleStorage {
5     // State variable to store a number
6
7     ParameterList
8
9     remix-project-org/remix-workshops/5.1 Functions - Reading and Writing to a State Variable/readAt
10
11     Estimated execution cost: 2453 gas
12
13     function get() public view returns (uint) { 2453 gas
14         return num;
15     }
16 }

```

**Deploy & Run Transactions**

- HELLOWORLD AT 0xae0...96888 (MEMORY)
- MYCONTRACT AT 0x5e1...4eff5 (MEMORY)
- PRIMITIVES AT 0x38c...24c73 (MEMORY)
- VARIABLES AT 0xec2...cf142 (MEMORY)
- SIMPLESTORAGE AT 0x838...2a4dc (MEMORY)

Balance: 0 ETH

ParameterList

remix-project-org/remix-workshops/5.1 Functions - Reading and Writing to a State Variable/readAt

Estimated execution cost: 2453 gas

function get() public view returns (uint) { 2453 gas  
return num;

**Explain contract**

0 Listen on all transactions Filter with transaction hash or address Debug

call to SimpleStorage.num

call [call] from: 0x58380a6a701c568545dCfcB03Fc8875f56beddC4 to: SimpleStorage.num() data: 0xe7...0b1dc Debug

Activate Windows Go to Settings to activate Windows.

## Tutorial 6

```

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 }

```

**Deploy & Run Transactions**

- PRIMITIVES AT 0x38c...24c73 (MEMORY)
- VARIABLES AT 0xec2...cf142 (MEMORY)
- SIMPLESTORAGE AT 0x838...2a4dc (MEMORY)
- VIEWANDPURE AT 0x3c7...41288 (MEMORY)

Balance: 0 ETH

ParameterList

remix-project-org/remix-workshops/5.1 Functions - Reading and Writing to a State Variable/readAt

Estimated execution cost: 2453 gas

function get() public view returns (uint) { 2453 gas  
return num;

**Explain contract**

0 Listen on all transactions Filter with transaction hash or address Debug

call to ViewAndPure.x

call [call] from: 0x58380a6a701c568545dCfcB03Fc8875f56beddC4 to: ViewAndPure.x() data: 0x0c5...5699c Debug

Activate Windows Go to Settings to activate Windows.

## Tutorial 7

```

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.

```

**Deploy & Run Transactions**

- VARIABLES AT 0xEC2...CF142 (MEMORY)
- SIMPLESTORAGE AT 0X838...2A4DC (MEMORY)
- VIEWANDPURE AT 0X3C7...41288 (MEMORY)
- FUNCTIONMODIFIER AT 0XA0...5D3DF (MEMORY)

Balance: 0 ETH

changeOwner address \_newOwner

decrement uint256 i

locked

o: bool: false

owner

0: address: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

x

0: uint256: 10

Low level interactions

CALldata

CALL data: 0x8da...5cb5b  
call to FunctionModifier.x()

[call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: FunctionModifier.x()  
data: 0xc5...5699c

AI copilot

0 Listen on all transactions Filter with transaction hash or address

Debug

Activate Windows Go to Settings to activate Windows.

## Tutorial 8

LEARNETH

5.4 Functions - Inputs and Outputs

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` function to the new local variables `a`, `b`, and `c` (line 60).

**Input and Output restrictions**

There are a few restrictions and best practices for the input and output parameters of contract functions.

"[Mappings] cannot be used as parameters or return parameters of contract functions that are publicly visible." From the [Solidity documentation](#).

Arrays can be used as parameters, as shown in the function `arrayInput` (line 71). Arrays can also be used as return parameters as shown in the function `arrayOutput` (line 76).

You have to be cautious with arrays of arbitrary size because of their gas consumption. While a function using very large arrays as inputs might fail when the gas costs are too high, a function using a smaller array might still be able to execute.

Watch a video tutorial on [Function Outputs](#).

**★ Assignment**

Create a new function called `returnTwo` that returns the values `2` and `true` without using a return statement.

Check Answer Show answer

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 // Pranav Titamble d2ba/60
4 contract Function {
5     // Functions can return multiple values.
6     function returnMany() {
7         public
8             pure
9             returns (
10                 uint,
11                 bool,
12                 uint
13             )
14     {
15         return (1, true, 2);
16     }
17
18     // Return values can be named.
19     function named() {
20         public
21             pure
22             returns (
23                 uint x,
24                 bool b,
25                 uint y
26             )
27     {
28         return (1, true, 2);
29     }
30
31     // Return values can be assigned to their name.
32     // In this case the return statement can be omitted.

```

Explain contract

0 Listen on all transactions Filter

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x

## Tutorial 9

**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**

- Can be called from other contracts or transactions
- State variables can not be `external`.

In this example, we have two contracts, the `base` contract (line 4) and the `child` contract (line 55) which inherits the functions and state variables from the `base` contract.

When you uncomment the `testPrivateFunc` (lines 58-60) you get an error because the child contract doesn't have access to the private function `privateFunc` from the `base` contract.

If you compile and deploy the two contracts, you will not be able to call the functions `privateFunc` and `internalFunc` directly. You will only be able to call them via `testPrivateFunc` and `testInternalFunc`.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Base {
7     // Private function can only be called
8     // - inside this contract
9     // Contracts that inherit this contract cannot call this function.
10    function privateFunc() private pure returns (string memory) { return "infinite gas"; }
11    }
12
13    function testPrivateFunc() public pure returns (string memory) { return privateFunc(); }
14
15    function testInternalFunc() public pure virtual returns (string memory) { return internalFunc(); }
16
17    function internalFunc() internal pure returns (string memory) { return "internal function called"; }
18
19    // Internal function can be called
20    // - inside this contract
21    // - inside contracts that inherit this contract
22    function internalFunc() internal pure returns (string memory) { return "infinite gas"; }
23
24    function testInternalFunc() public pure virtual returns (string memory) { return internalFunc(); }
25
26    function testInternalFunc() public pure virtual returns (string memory) { return internalFunc(); }
27
28    // Public functions can be called
29    // - inside this contract
30    // - inside contracts that inherit this contract
31    // - by other contracts and accounts
32

```

## Tutorial 10

**7.1 Control Flow - If/Else**

Solidity supports different control flow statements that determine which parts of the contract will be executed. The conditional `if/else` statement enables contracts to make decisions depending on whether boolean conditions are either `true` or `false`.

Solidity differentiates between three different `If/Else` statements: `if`, `else`, and `else if`.

**if**

The `if` statement is the most basic statement that allows the contract to perform an action based on a boolean expression.

In this contract's `foo` function (line 5) the `if` statement (line 6) checks if `x` is smaller than `10`. If the statement is true, the function returns `0`.

**else**

The `else` statement enables our contract to perform an action if conditions are not met.

In this contract, the `foo` function uses the `else` statement (line 10) to return `2` 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 `wint` as an argument.
- The function returns `true` if the argument is even, and `false` if the argument is odd.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract IfElse {
7     function foo(uint x) public pure returns (uint) { if (x < 10) { return 0; } else if (x < 20) { return 1; } else { return 2; } }
8
9
10    function ternary(uint _x) public pure returns (uint) { // if (_x < 10) { // return 1; // } // return 2; }
11
12    // shorthand way to write if / else statement
13    return _x < 10 ? 1 : 2;
14
15
16
17
18
19
20
21
22
23
24
25
26

```

## Tutorial 11

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' sidebar displays the 'Tutorials list' and the current section '7.2 Control Flow - Loops'. The main content area contains text about loops and a detailed explanation of the provided Solidity code. The code defines a contract named 'Loop' with two functions: a public infinite-gas function that loops from 0 to 10, skipping index 3, and a private while loop that also loops from 0 to 10, exiting at index 5. The right side of the interface shows the 'Compile' tab, a code editor with syntax highlighting, and a transaction history panel.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/6
5
6 contract Loop {
7     function loop() public {
8         // for loop
9         for (uint i = 0; i < 10; i++) {
10             if (i == 3) {
11                 // Skip to next iteration with continue
12                 continue;
13             }
14             if (i == 5) {
15                 // Exit loop with break
16                 break;
17             }
18         }
19     }
20
21     uint();
22     while (i < 10) {
23         i++;
24     }
25 }
26

```

## Tutorial 12

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' sidebar displays the 'Tutorials list' and the current section '8.1 Data Structures - Arrays'. The main content area contains text about arrays and a detailed explanation of the provided Solidity code. The code defines a contract named 'Arry' with various array-related functions: get, push, and pop. The right side of the interface shows the 'Compiled' tab, a code editor with syntax highlighting, and a transaction history panel.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 // Pranav Titambe D20A/6
4
5 contract Arry {
6     // Several ways to initialize an array
7     uint[] public arr;
8     uint[3] public arr2 = [1, 2, 3];
9     // Fixed sized array, all elements initialize to 0
10    uint[10] public myFixedSizeArr;
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[] memory) {
20        return arr;
21    }
22
23    function push(uint i) public {
24        // Append to arry
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    }

```

## Tutorial 13

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' tutorial sidebar for '8.2 Data Structures - Mappings' is visible. The main content area displays the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Mapping {
7     // Mapping from address to uint
8     mapping(address => uint) public myMap;
9
10    function get(address _addr) public view returns (uint) {
11        // Mapping always returns a value.
12        // If the value was never set, it will return the default value.
13        return myMap[_addr];
14    }
15
16    function set(address _addr, uint _i) public {
17        // Update the value at this address
18        myMap[_addr] = _i;
19    }
20
21    function remove(address _addr) public {
22        // Reset the value to the default value.
23        delete myMap[_addr];
24    }
25
26    contract NestedMapping {
27        // Nested mapping (mapping from address to another mapping)
28        mapping(address => mapping(uint => bool)) public nested;
29
30        function get(address _addr1, uint _i) public view returns (bool) {
31            // You can get values from a nested mapping
32        }
33    }
34}

```

The code defines a `Mapping` contract with a mapping from `address` to `uint`. It includes functions for getting, setting, and removing values. It also defines a nested contract `NestedMapping` with a mapping from `address` to another mapping.

## Tutorial 14

The screenshot shows the REMIX IDE interface. On the left, the 'LEARNETH' tutorial sidebar for '8.3 Data Structures - Structs' is visible. The main content area displays the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Todos {
7     struct Todo {
8         string text;
9         bool completed;
10    }
11
12    // An array of 'Todo' structs
13    Todo[] public todos;
14
15    function create(string memory _text) public {
16        // 3 ways to initialize a struct
17        // - calling it like a function
18        todos.push(Todo(_text, false));
19
20        // key value mapping
21        todos.push(Todo({_text, completed: false}));
22
23        // initialize an empty struct and then update it
24        Todo memory todo;
25        todo.text = _text;
26        // todo.completed initialized to false
27
28        todos.push(todo);
29    }
30
31    // Solidity automatically created a getter for 'todos' so
32    // you don't actually need this function.
33}

```

The code defines a `Todos` contract with a struct `Todo` containing `string text` and `bool completed`. It includes a `create` function that adds `Todo` objects to an array. The code demonstrates three ways to initialize a struct: using a constructor-like function call, using a key-value mapping, and initializing an empty struct and then updating its members.

## Tutorial 15

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Enum {
7     // Enum representing shipping status
8     enum Status {
9         Pending,
10        Shipped,
11        Accepted,
12        Rejected,
13        Canceled
14    }
15
16    // Default value is the first element listed in
17    // definition of the type, in this case "Pending"
18    Status public status;
19
20    // Returns uint
21    // Pending - 0
22    // Shipped - 1
23    // Accepted - 2
24    // Rejected - 3
25    // Canceled - 4
26    function get() public view returns (Status) {
27        return status;
28    }
29
30    // Update status by passing uint into input
31    function set(Status _status) public {
32        status = _status;
33    }

```

**Explain contract**

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

## Tutorial 16

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract DataLocations {
7     uint[] public arr;
8     mapping(uint => address) map;
9     struct MyStruct {
10        uint foo;
11    }
12    mapping(uint => MyStruct) myStructs;
13
14    function f() public {
15        // call _f with state variables
16        _f(arr, map, myStructs[1]);
17
18        // get a struct from a mapping
19        MyStruct storage myStruct = myStructs[1];
20        // create a struct in memory
21        MyStruct storage myMemStruct = MyStruct(0);
22    }
23
24    function _f() {
25        uint[] storage _arr,
26        mapping(uint => address) storage _map,
27        MyStruct storage _myStruct
28    } internal {
29        // do something with storage variables
30    }
31
32    // You can return memory variables

```

**Explain contract**

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

## Tutorial 17

**10.1 Transactions - Ether and Wei**

Ether (ETH) is a cryptocurrency. Ether is also used to pay fees for using the Ethereum network, like making transactions in the form of sending Ether to an address or interacting with an Ethereum application.

**Ether Units**

To specify a unit of Ether, we can add the suffixes `wei`, `gwei`, or `ether` to a literal 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 1,000,000,000,000,000,000 ( $10^{18}$ ) `wei` (line 11).

Watch a video tutorial on Ether and Wei.

**★ Assignment**

- Create a `public uint` called `oneWei` and set it to 1 `gwei`.
- Create a `public bool` called `isOneWei` 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**

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract EtherUnits {
7     uint public oneWei = 1 wei;
8     // 1 wei is equal to 1
9     bool public isOneWei = 1 wei == 1;
10
11    uint public oneEther = 1 ether;
12    // 1 ether is equal to 10^18 wei
13    bool public isOneEther = 1 ether == 1e18;
14 }

```

**Explain contract**

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

## Tutorial 18

**10.2 Transactions - Gas and Gas Price**

As we have seen in the previous section, executing code via transactions on the Ethereum Network costs transaction fees in the form of Ether. The amount of fees that have to be paid to execute a transaction depends on the amount of gas that the execution of the transaction costs.

**Gas**

Gas is the unit that measures the amount of computational effort that is required to execute a specific operation on the Ethereum network.

**Gas price**

The gas that fuels Ethereum is sometimes compared to the gas that fuels a car. The amount of gas your car consumes is mostly the same, but the price you pay for gas depends on the market.

Similarly, the amount of gas that a transaction requires is always the same for the same computational work that is associated with it. However the price that the sender of the transaction is willing to pay for the gas is up to them. Transactions with higher `gas prices` are going through faster; transactions with very low `gas prices` might not go through at all.

When sending a transaction, the sender has to pay the `gas fee` (`gas_price * gas`) upon execution of the transaction. If gas is left over after the execution is completed, the sender gets refunded.

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**

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract Gas {
7     uint public i = 0;
8
9     // Using up all of the gas that you send causes your transaction
10    // State changes are undone.
11    // Gas spent are not refunded.
12    function forever() public {    infinite gas
13        // Here we run a loop until all of the gas are spent
14        // and the transaction fails
15        while (true) {
16            i += 1;
17        }
18    }
19 }

```

**Explain contract**

[vm] from: 0x5B3...eddC4 to: Array.(constructor) value: 0 wei data: 0x608...f0033

## Tutorial 19

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 // Pranav Titambe d20a/60
5
6 contract ReceiveEther {
7     /*
8      * Which function is called?
9      */
10    | send Ether
11    | msg.data is empty?
12    |   / \
13    |   yes no
14    |
15    receive() exists? fallback()
16    |   /
17    |   yes no
18    |   /
19    |   \
20    |   receive() fallback()
21
22    // Function to receive Ether. msg.data must be empty
23    receive() external payable {} // undefined gas
24
25    // Fallback function is called when msg.data is not empty
26    fallback() external payable {} // undefined gas
27
28    function getBalance() public view returns (uint) { // 312 gas
29        return address(this).balance;
30    }
31
32 }

```

The screenshot shows the REMIX IDE interface. On the left, there's a tutorial window titled "10.3 Transactions - Sending Ether" from the "LEARNETH" course. It discusses three methods for sending Ether: `transfer()`, `send()`, and `call()`. It notes that `transfer()` is deprecated. The right side of the screen shows the Solidity code for a `ReceiveEther` contract. A tooltip is displayed over the `receive()` function, asking "Which function is called?" with options "send Ether" and "msg.data is empty?". Below the code, there's an "Explain contract" section and a transaction log entry.

**Conclusion:** Through this experiment, the fundamentals of Solidity programming were explored by completing practical assignments in the Remix IDE. Concepts such as data types, variables, functions, visibility, modifiers, constructors, control flow, data structures, and transactions were implemented and understood. The hands-on practice helped in designing, compiling, and deploying smart contracts on the Remix VM, thereby strengthening the understanding of blockchain concepts. This experiment provided a strong foundation for developing and managing smart contracts efficiently.