

## EXPERIMENT NO.4

**Name:**Aryan Anil Patankar

**Class:**D20A

**Roll No:**38

**Batch:**B

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

**Theory:-**

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

**Primitive (Value) Data Types** in Solidity include:

- bool → true/false
- int / uint (signed/unsigned integers, e.g., uint256 is most common)
- address → 20-byte Ethereum address
- bytes1 to bytes32 → fixed-size byte arrays
- string → dynamic UTF-8 encoded text (reference type, but often grouped here)

**Variables** are declared with a type and can be:

- **State variables** → stored permanently on the blockchain (expensive)
- **Local variables** → exist only during function execution

**Functions** can be marked as:

- pure → does not read or write state (computes only from inputs; cheapest gas)
- view → reads state but does not modify it (e.g., getters; no gas when called externally)

## Example

```
function getResult() public view returns (uint product, uint sum) {  
    product = num1 * num2; // reads state  
    sum = num1 + num2;  
}
```

```
function pureCalc(uint a, uint b) public pure returns (uint) {  
    return a + b; // no state access  
}
```

## 2. Inputs and Outputs to Functions

Functions in Solidity can take **parameters** (inputs) and return **values** (outputs).

- **Inputs:** Declared in parentheses; can use data locations like memory or calldata for reference types.
- **Outputs:** Declared after returns keyword; can return multiple values.

## Example:

```
function add(uint a, uint b) public pure returns (uint sum) {  
    sum = a + b;  
}
```

// Multiple outputs

```
function getValues() public view returns (uint, string memory) {  
    return (age, name);  
}
```

- Use calldata for external calls (cheaper, read-only).
- Use memory for temporary copies inside functions.

### 3. Visibility, Modifiers and Constructors

**Visibility Specifiers** (for functions and state variables):

- **public** → anyone can call/read (default for state vars creates getter)
- **private** → only inside current contract
- **internal** → current + derived (child) contracts
- **external** → only external calls (cheaper for large data)

**Modifiers** → reusable code blocks that run before/after function body (e.g., access control).

Use `_;` to insert function body.

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

```
function restricted() public onlyOwner { ... }
```

**Constructors** → special function that runs once on deployment (initializes state).

- Syntax: `constructor() { ... }` (older: same name as contract)

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

Solidity supports standard control structures:

### if-else:

```
if (condition) {  
    // true  
} else if (another) {  
    // else-if  
} else {  
    // false  
}
```

### Loops:

- for (uint i = 0; i < 10; i++) { ... }
- while (condition) { ... }
- do { ... } while (condition);

Avoid unbounded loops (gas limit risk). Use break / continue when needed.

## 5. Data Structures: Arrays, Mappings, structs, enums

- **Arrays**
  - Fixed: uint[5] arr;
  - Dynamic: uint[] arr; (use .push(), .pop(), .length)
- **Mappings** → key-value store (like hash table)  
mapping(address => uint) public balances;

Keys can be most types; no iteration possible.

**Structs** → custom composite types

```
struct User {  
    address wallet;  
    uint balance;  
    bool active;  
}  
User public owner;
```

**Enums** → named constants (integers under the hood)

```
enum Status { Pending, Active, Cancelled }  
Status public state = Status.Pending;
```

## 6. Data Locations

Solidity has three main **data locations** for reference types (arrays, structs, mappings, strings):

- **storage** → permanent blockchain storage (persistent, expensive)  
Default for state variables.
- **memory** → temporary, function lifetime only (deleted after execution)  
Cheap; used for local variables & function args/returns.
- **calldata** → read-only, non-modifiable area for function call data  
Cheapest for external function parameters (immutable copy of tx data).

Rule:

- State vars → always storage
- Function args → prefer calldata (external) or memory
- Local reference vars → must specify location

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

- **Ether units** → smallest is **wei** ( $1 \text{ ETH} = 10^{18} \text{ wei}$ ) Other: **gwei** ( $10^9 \text{ wei}$ ), commonly used for gas prices.
- **Gas** → computational effort unit
  - **Gas Limit** → max gas willing to spend (set by sender)
  - **Gas Price** → price per gas unit (in wei/gwei; set by sender)
  - Total fee = gas used  $\times$  gas price
- **msg.value** → amount of wei sent with transaction
- **Sending Ether** → use `.transfer()`, `.send()`, or `.call{value: amount}("")` (recommended low-level call)

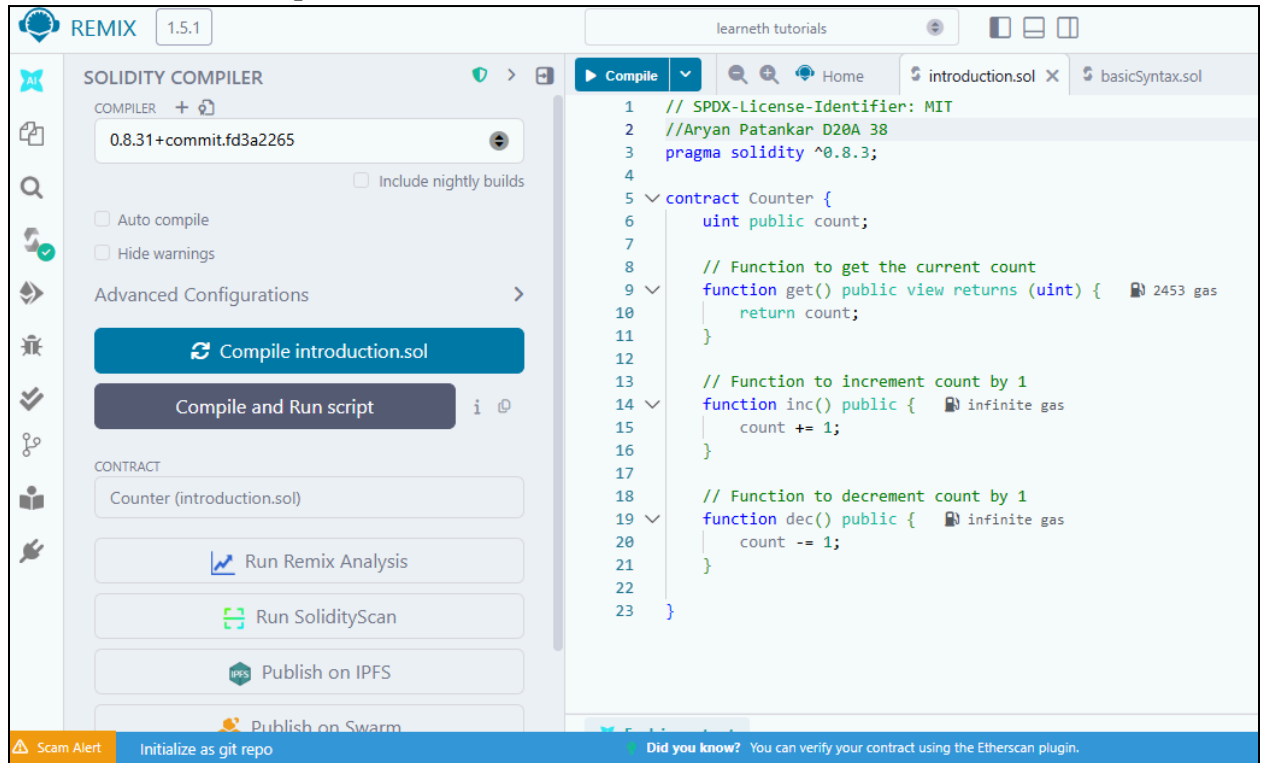
Example:

```
function sendEther(address payable recipient) public payable {  
    recipient.transfer(msg.value);  
}
```

- Use payable for addresses/functions that receive Ether.
- `tx.gasprice` → current gas price (global var)

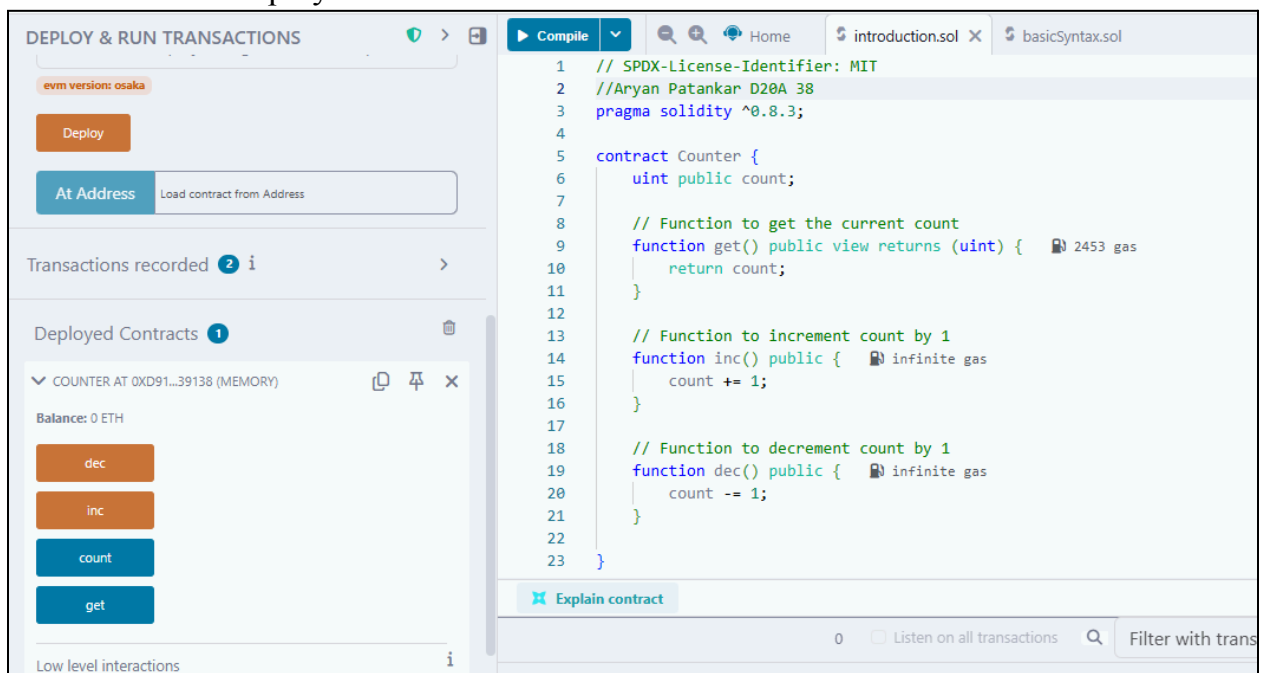
## Implementation:-

- Tutorial no. 1 – Compile the code



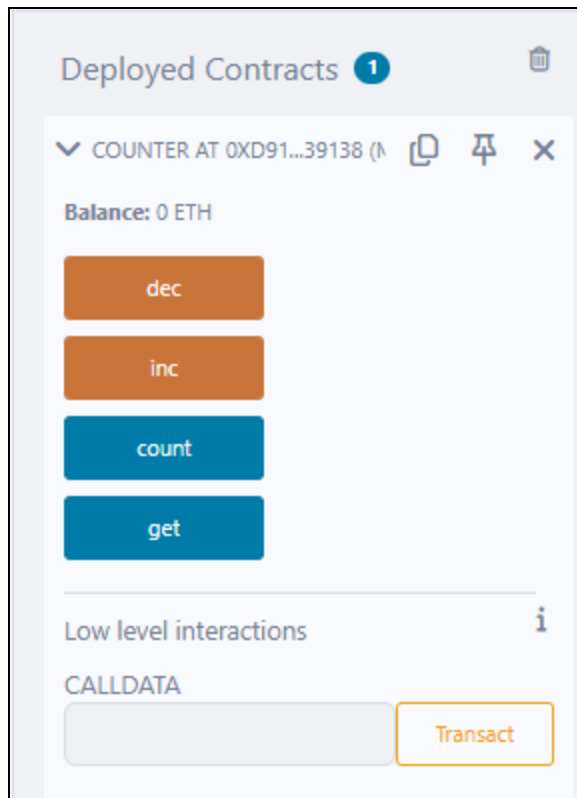
The screenshot shows the Remix IDE interface. On the left, the 'SOLIDITY COMPILER' panel is active, displaying the compiler version '0.8.31+commit.fd3a2265'. Below the version, there are checkboxes for 'Auto compile' and 'Hide warnings', and a section for 'Advanced Configurations'. A large blue button labeled 'Compile introduction.sol' is prominent. Below this, there are buttons for 'Compile and Run script', 'Run Remix Analysis', 'Run SolidityScan', 'Publish on IPFS', and 'Publish on Swarm'. The 'CONTRACT' section shows 'Counter (introduction.sol)'. At the bottom of the compiler panel, there is a 'Scam Alert' and a link to 'Initialize as git repo'. On the right, the code editor shows the Solidity code for the 'Counter' contract. The code includes a license header, a pragma statement for Solidity 0.8.3, and three functions: 'get()' to return the current count, 'inc()' to increment the count by 1, and 'dec()' to decrement the count by 1. The 'Compile' button is highlighted in the top bar.

- Tutorial No.1-Deploy the contract

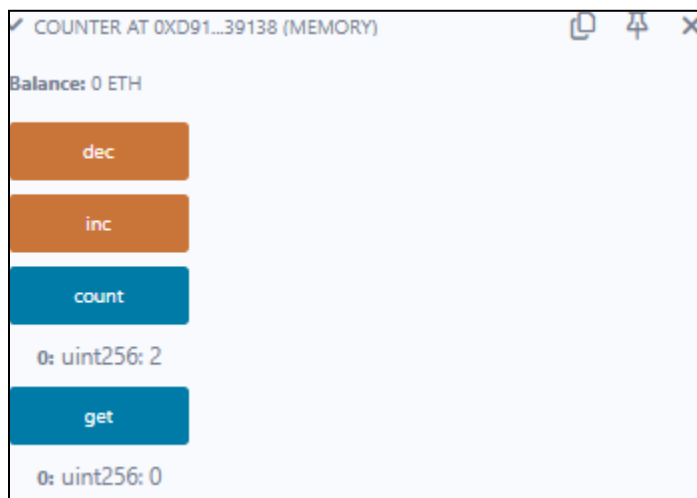


The screenshot shows the 'DEPLOY & RUN TRANSACTIONS' panel in the Remix IDE. The 'evm version: osaka' is selected. The 'Deploy' button is highlighted. Below it, the 'At Address' section shows a text input field with the address '0XD91...39138'. The 'Transactions recorded' section shows 2 transactions. The 'Deployed Contracts' section shows 1 contract, 'COUNTER AT 0XD91...39138 (MEMORY)', with a balance of '0 ETH'. Below the contract name, there are four buttons: 'dec', 'inc', 'count', and 'get'. The code editor on the right shows the same Solidity code as in the previous screenshot. At the bottom of the IDE, there is a search bar for 'Filter with trans' and a checkbox for 'Listen on all transactions'.

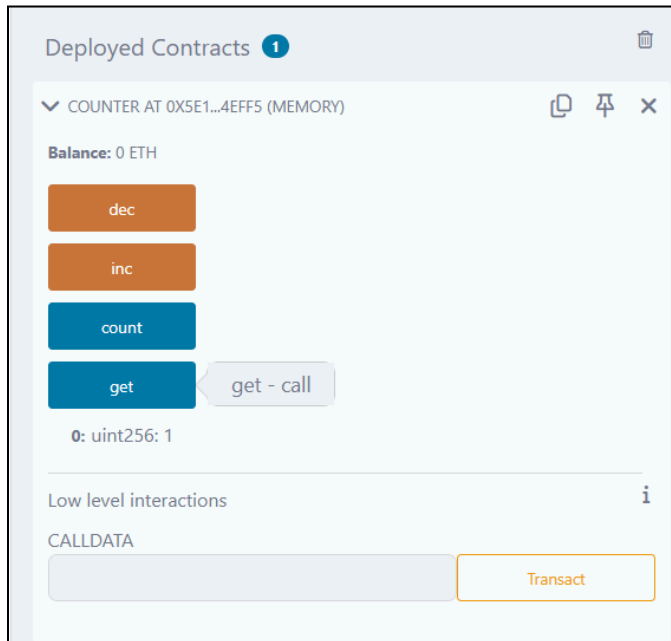




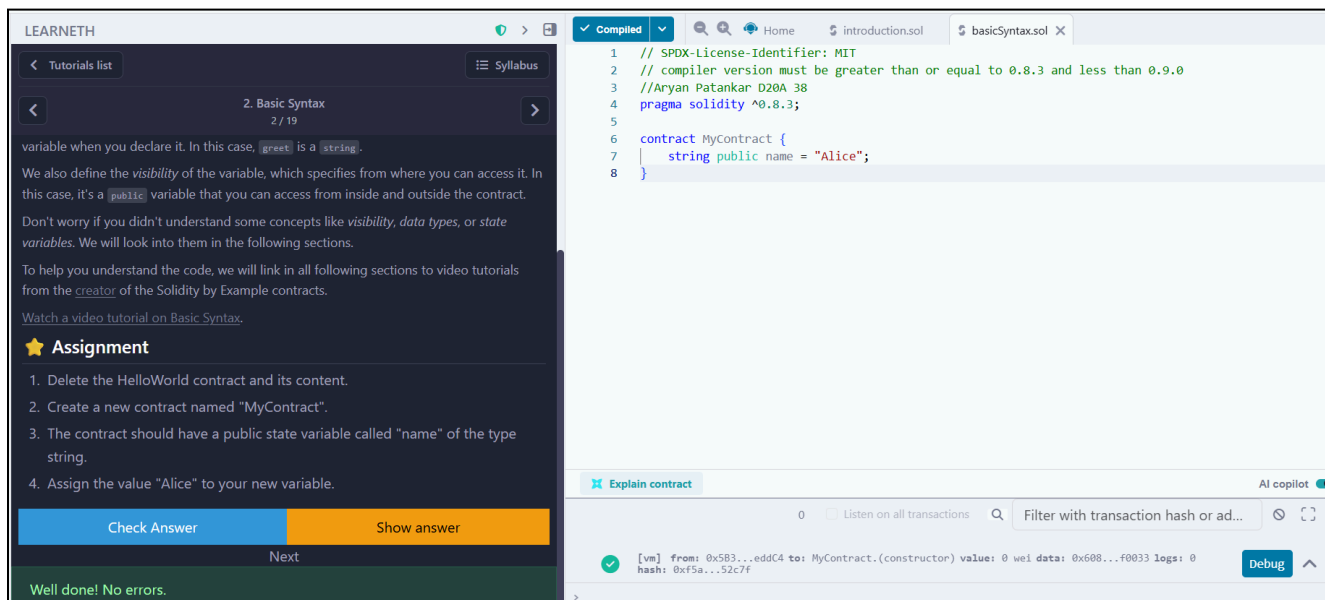
- Tutorial no. 1 – Increment



- Tutorial no. 1 – Decrement



- Tutorial no.2



## • Tutorial no.3

The screenshot shows the LEARNETH interface for Tutorial no.3, '3. Primitive Data Types'. The left sidebar contains a 'Tutorials list' and a 'Syllabus' section. The main content area displays the tutorial title, a brief introduction, and an 'Assignment' section with three tasks. The right pane shows a Solidity code editor with a contract named 'primitiveDataTypes.sol'. The code defines several public variables: `int8`, `int`, `address`, `bool`, `uint`, and `uint8`. Below the code editor, there is an 'Explain contract' button and a transaction filter. The bottom status bar shows a green checkmark and the message 'Well done! No errors.'

**3. Primitive Data Types**  
3 / 19

Later in the course, we will look at data structures like **Mappings**, **Arrays**, **Enums**, and **Structs**.  
Watch a video tutorial on Primitive Data Types.

★ **Assignment**

1. Create a new variable `newAddr` that is a `public address` and give it a value that is not the same as the available variable `addr`.
2. Create a `public` variable called `neg` that is a negative number, decide upon the type.
3. Create a new variable, `newU` that has the smallest `uint` size type and the smallest `uint` value and is `public`.

Tip: Look at the other address in the contract or search the internet for an Ethereum address.

[Check Answer](#) [Show answer](#)

Next

Well done! No errors.

```
23 int8 public i8 = -1;
24 int public i256 = 456;
25 int public i = -123; // int is same as int256
26
27 address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;
28
29 // Default values
30 // Unassigned variables have a default value
31 bool public defaultBool; // false
32 uint public defaultUint; // 0
33 int public defaultInt; // 0
34 address public defaultAddr; // 0x0000000000000000000000000000000000000000
35
36 //Aryan Patankar D20A 38
37 address public newAddr=0x0000000000000000000000000000000000000000;
38 int public neg=-15;
39 uint8 public newU=0;
40 }
```

[Explain contract](#) AI copilot

0 ☐ Listen on all transactions

[vm] from: 0x583...eddC4 to: MyContract.(constructor) value: 0 wei data: 0x608...f0033

[Debug](#)

## • Tutorial no.4

The screenshot shows the LEARNETH interface for Tutorial no.4, '4. Variables'. The left sidebar contains a 'Tutorials list' and a 'Syllabus' section. The main content area displays the tutorial title, a brief introduction, and an 'Assignment' section with two tasks. The right pane shows a Solidity code editor with a contract named 'primitiveDataTypes.sol'. The code defines a contract with state variables `string`, `uint`, and `uint8`, and a function `doSomething()` that uses local and global variables. Below the code editor, there is an 'Explain contract' button and a transaction filter. The bottom status bar shows a green checkmark and the message 'Well done! No errors.'

**4. Variables**  
4 / 19

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.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 // Aryan Patankar D20A 38
4 contract Variables {
5     // State variables are stored on the blockchain.
6     string public text = "Hello";
7     uint public num = 123;
8     uint8 public blockNumber;
9
10    function doSomething() public {
11        // Local variables are not saved to the blockchain.
12        uint i = 456;
13
14        // Here are some global variables
15        uint timestamp = block.timestamp; // Current block timestamp
16        address sender = msg.sender; // address of the caller
17        blockNumber=block.number;
18    }
19 }
```

[Explain contract](#)

0 ☐ Listen on all transactions

[vm] from: 0x583...eddC4 to: MyContract.(constructor) value: 0 wei data: 0x608...f0033

- Tutorial no.5

Tutorials list

Syllabus

5.1 Functions - Reading and Writing to a State

Variable

5 / 19

function if they don't modify the state. Our `get` function also returns values, so we have to specify the return types. In this case, it's a `uint` since the state variable `num` that the function returns is a `uint`.

We will explore the particularities of Solidity functions in more detail in the following sections.

[Watch a video tutorial on Functions.](#)

★ Assignment

1. Create a public state variable called `b` that is of type `bool` and initialize it to `true`.
2. Create a public function called `get_b` that returns the value of `b`.

Check Answer

Show answer

Next

Well done! No errors.

Compiled

Home

Settings

basicSyntax.sol

primitiveDataTypes.sol

var

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
4 contract SimpleStorage {
5     // State variable to store a number
6     uint public num;
7     bool public b=true;
8     // You need to send a transaction to write to a state variable.
9     function set(uint _num) public { 22536 gas
10         num = _num;
11     }
12
13     // You can read from a state variable without sending a transaction.
14     function get() public view returns (uint) { 2475 gas
15         return num;
16     }
17     function get_b() public view returns(bool){ 2539 gas
18         return b;
19     }
20 }
```

Explain contract

0 ☐ Listen on all transactions

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

>

- Tutorial no.6

Tutorials list

Syllabus

5.2 Functions - View and Pure

6 / 19

the state variable `x`.

In Solidity development, you need to optimise your code for saving computation cost (gas cost). Declaring functions view and pure can save gas cost and make the code more readable and easier to maintain. Pure functions don't have any side effects and will always return the same result if you pass the same arguments.

[Watch a video tutorial on View and Pure Functions.](#)

★ Assignment

Create a function called `addToX2` that takes the parameter `y` and updates the state variable `x` with the sum of the parameter and the state variable `x`.

Check Answer

Show answer

Next

Well done! No errors.

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
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 function addToX2(uint y) public{ infinite gas
17 x=x+y;
18 }
19 }

Explain contract

0 ☐ Listen on all transactions

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

>

## • Tutorial no.7

LEARNETH

Tutorials list

Syllabus

5.3 Functions - Modifiers and Constructors

7 / 19

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

1. 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.

2. Make sure that `x` can only be increased.

3. The body of the function `increaseX` should be empty.

Tip: Use modifiers.

Check Answer

Show answer

Next

Well done! No errors.

Compile

rimitiveDataTypes.sol

variables.sol

readAndWrite.sol

viewAndPure.sol

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

bool public locked;

constructor() { 440798 gas 394000 gas

// Set the transaction sender as the owner of the contract.

owner = msg.sender;

}

// Aryan Patankar D20A 38

modifier onlyPositive(uint \_value) {

require(\_value > 0, "Increase value must be greater than zero");

\_;

}

function increaseX(uint \_amount) public onlyPositive(\_amount) { infinite gas

x += \_amount;

}

// Modifier to check that the caller is the owner of

// the contract.

modifier onlyOwner() {

require(msg.sender == owner, "Not owner");

// Underscore is a special character only used inside

}

✖ Explain contract

0 ☐ Listen on all transactions 🔍 Filter with transaction hash

Type the library name to see available commands.

## • Tutorial no.8

LEARNETH

Tutorials list

Syllabus

5.4 Functions - Inputs and Outputs

8 / 19

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

Next

Well done! No errors.

Compile

readAndWrite.sol

viewAndPure.sol

modifiersAndConstructors.sol

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

function arrayInput(uint[] memory \_arr) public {} infinite gas

// Can use array for output

uint[] public arr;

function arrayOutput() public view returns (uint[] memory) { infinite gas

return arr;

}

//Aryan Patankar D20A 38

function returnTwo() public pure returns( 472 gas

int i,

bool b

)

{

i=-2;

b=true;

}

✖ Explain contract

0 ☐ Listen on all transactions 🔍 Filter with transaction hash

Type the library name to see available commands.

## • Tutorial no.9

**LEARNETH**

Tutorials list | Syllabus

6. Visibility  
9 / 19

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`.

Watch a video tutorial on Visibility.

★ **Assignment**

Create a new function in the `Child` contract called `testInternalVar` that returns the values of all state variables from the `Base` contract that are possible to return.

Check Answer | Show answer

Next

Well done! No errors.

```
52 // string external externalVar = "my external variable";
53 }
54
55 contract Child is Base {
56     // Inherited contracts do not have access to private functions
57     // and state variables.
58     // function testPrivateFunc() public pure returns (string memory) {
59     //     return privateFunc();
60     // }
61
62     // Internal function call be called inside child contracts.
63     //Aryan Patankar D20A 38
64     function testInternalFunc() public pure override returns (string memory) {
65         return internalFunc();
66     }
67
68     function testInternalVar() public view returns(string memory x,string memory y){
69         return(internalVar,publicVar);
70     }
71 }
```

Explain contract

0 ☐ Listen on all transactions

Type the library name to see available commands.

## • Tutorial no.10

**LEARNETH**

Tutorials list | Syllabus

7.1 Control Flow - If/Else  
10 / 19

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.

```
10 } else {
11     return 2;
12 }
13
14 }
15
16 function ternary(uint _x) public pure returns (uint) {
17     // if (_x < 10) {
18     //     return 1;
19     // }
20     // return 2;
21
22     // shorthand way to write if / else statement
23     return _x < 10 ? 1 : 2;
24 }
25
26 //Aryan Patankar D20A 38
27 function evenCheck(uint _number) public pure returns (bool) {
28     return (_number % 2 == 0) ? true : false;
29 }
```

Explain contract

0 ☐ Listen on all transactions

Type the library name to see available commands.

- Tutorial no.11

LEARNETH

Tutorials list

Syllabus

7.2 Control Flow - Loops

11 / 19

break

The `break` statement is used to exit a loop. In this contract, the break statement (line 14) will cause the for loop to be terminated after the sixth iteration.

[Watch a video tutorial on Loop statements.](#)

★ Assignment

1. Create a public `uint` state variable called count in the `Loop` contract.
2. At the end of the for loop, increment the count variable by 1.
3. Try to get the count variable to be equal to 9, but make sure you don't edit the `break` statement.

Check Answer

Show answer

Next

Well done! No errors.

Compile

modifiersAndConstructors.sol

inputsAndOutputs.sol

```

15         break;
16     }
17     count++;
18 }
19
20 // while loop
21 //Aryan Patankar D20A 38
22 uint j;
23 while (j < 10) {
24     j++;
25     if (count < 9) {
26         count++;
27     }
28 }
29
30 }
31

```

Explain contract

0 ☐ Listen on all transactions

Filter

Type the library name to see available commands.

- Tutorial no.12

LEARNETH

Tutorials list

Syllabus

8.1 Data Structures - Arrays

12 / 19

the array to the place of the deleted element (line 46), or use a mapping. A mapping might be a better choice if we plan to remove elements in our data structure.

Array length

Using the length member, we can read the number of elements that are stored in an array (line 35).

[Watch a video tutorial on Arrays.](#)

★ Assignment

1. Initialize a public fixed-sized array called `arr3` with the values 0, 1, 2. Make the size as small as possible.
2. Change the `getArr()` function to return the value of `arr3`.

Check Answer

Show answer

Next

Well done! No errors.

Compile

Constructors.sol

inputsAndOutputs.sol

visibility.sol

ifElse.sol

loop

```

4  contract Array {
5      // Several ways to initialize an array
6      //Aryan Patankar D20A 38
7      uint[] public arr;
8      uint[] public arr2 = [1, 2, 3];
9      uint[3] public arr3 = [0, 1, 2];
10
11      // Fixed sized array, all elements initialize to 0
12      uint[10] public myFixedSizeArr;
13
14      function get(uint i) public view returns (uint) {
15          return arr[i];
16      }
17
18
19      // Solidity can return the entire array.
20      // But this function should be avoided for
21      // arrays that can grow indefinitely in length.
22      function getArr() public view returns (uint[3] memory) {
23          return arr3;
24      }
25

```

Explain contract

0 ☐ Listen on all transactions

Filter with transaction hash

Type the library name to see available commands.

## • Tutorial no.13

LEARNETH

Tutorials list

Syllabus

8.2 Data Structures - Mappings

13 / 19

with a key, which will set it to the default value of 0. As we have seen in the arrays section.

Watch a video tutorial on [Mappings](#).

★ Assignment

1. Create a public mapping `balances` that associates the key type `address` with the value type `uint`.

2. Change the functions `get` and `remove` to work with the mapping balances.

3. Change the function `set` to create a new entry to the balances mapping, where the key is the address of the parameter and the value is the balance associated with the address of the parameter.

Check Answer

Show answer

Next

Well done! No errors.

Compile

inputsAndOutputs.solvisibility.solifElse.soloops.solarrays.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
4
5 contract Mapping {
6     // Mapping from address to uint
7     mapping(address => uint) public balances;
8
9     function get(address _addr) public view returns (uint) { 2872 gas
10         // Mapping always returns a value.
11         // If the value was never set, it will return the default value.
12         return balances[_addr];
13     }
14
15     function set(address _addr) public { 25256 gas
16         // Update the value at this address
17         balances[_addr] = _addr.balance;
18     }
19
20     function remove(address _addr) public { 5566 gas
21         // Reset the value to the default value.
22         delete balances[_addr];
23     }
24 }
```

Explain contract

0 ☐ Listen on all transactions

Type the library name to see available commands.

## • Tutorial no.14

LEARNETH

Tutorials list

Syllabus

8.3 Data Structures - Structs

14 / 19

23).

Accessing structs

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

Updating structs

To update a structs' member we also use the dot operator and assign it a new value (lines 39 and 45).

Watch a video tutorial on [Structs](#).

★ Assignment

Create a function `remove` that takes a `uint` as a parameter and deletes a struct member with the given index in the `todos` mapping.

Check Answer

Show answer

Next

Well done! No errors.

Compile

its.solvisibility.solifElse.soloops.solarrays.solmappings.sol

```
35
36
37 // update text
38 function update(uint _index, string memory _text) public { infinite gas
39     Todo storage todo = todos[_index];
40     todo.text = _text;
41 }
42
43 // update completed
44 function toggleCompleted(uint _index) public { 28995 gas
45     Todo storage todo = todos[_index];
46     todo.completed = !todo.completed;
47 }
48 //Aryan Patankar D20A 38
49 function remove(uint _index) public { infinite gas
50     delete todos[_index];
51 }
52 }
```

Explain contract

0 ☐ Listen on all transactions

Type the library name to see available commands.

- Tutorial no.15

LEARNETH

Tutorials list

Syllabus

8.4 Data Structures - Enums

15 / 19

and its member (line 35).

Removing an enum value

We can use the delete operator to delete the enum value of the variable, which means as for arrays and mappings, to set the default value to 0.

Watch a video tutorial on Enums.

★ Assignment

1. Define an enum type called `Size` with the members `S`, `M`, and `L`.

2. Initialize the variable `sizes` of the enum type `Size`.

3. Create a getter function `getSize()` that returns the value of the variable `sizes`.

Check Answer

Show answer

Next

Well done! No errors.

Compile

ility.sol ifElse.sol loops.sol arrays.sol mappings.sol

```
14 // Default value is the first element listed in
15 // definition of the type, in this case "Pending"
16 Status public status;
17
18 //Aryan Patankar D20A 38
19 enum Size {
20     S,
21     M,
22     L
23 }
24
25 Size public sizes;
26
27 function getSize() public view returns (Size) {
28     return sizes;
29 }
30
31 // Returns uint
32 // Pending - 0
33 // Shipped - 1
34 // Accepted - 2
35 // Rejected - 3
```

2633 gas

Explain contract

0 ☐ Listen on all transactions

Type the library name to see available commands.

- Tutorial no.16

LEARNETH

Tutorials list

Syllabus

9. Data Locations

16 / 19

to be used in the beginning, when creating contracts we have to be mindful of gas costs. Therefore, we need to use data locations that require the lowest amount of gas possible.

★ Assignment

1. Change the value of the `myStruct` member `foo`, inside the `function f`, to 4.

2. Create a new struct `myMemStruct2` with the data location `memory` inside the `function f` and assign it the value of `myMemStruct`. Change the value of the `myMemStruct2` member `foo` to 1.

3. Create a new struct `myMemStruct3` with the data location `memory` inside the `function f` and assign it the value of `myStruct`. Change the value of the `myMemStruct3` member `foo` to 3.

4. Let the function `f` return `myStruct`, `myMemStruct2`, and `myMemStruct3`.

Check Answer

Show answer

Tip: Make sure to create the correct return types for the function `f`.

Compile

ifElse.sol loops.sol arrays.sol mappings.sol structs.sol enums.sol dataLocations.sol

```
12 //Aryan Patankar D20A 38
13 function f() public returns (MyStruct memory, MyStruct memory, MyStruct memory) {
14     // call f with state variables
15     _f(arr, map, myStructs[1]);
16
17     // 1. get a struct from a mapping (Storage reference)
18     MyStruct storage myStruct = myStructs[1];
19     // Change the value of the myStruct member foo to 4
20     myStruct.foo = 4;
21
22     // 2. create a struct in memory
23     MyStruct memory myMemStruct = MyStruct(0);
24
25     // Create myMemStruct2 (Memory reference)
26     MyStruct memory myMemStruct2 = myMemStruct;
27     // Change myMemStruct2 member foo to 1
28     myMemStruct2.foo = 1;
29
30     // 3. Create myMemStruct3 (Independent copy from storage)
31     MyStruct memory myMemStruct3 = myStruct;
32     myMemStruct3.foo = 3;
33
34     // 4. Return all three structs
35     return (myStruct, myMemStruct2, myMemStruct3);
36 }
```

infinite gas

Explain contract

0 ☐ Listen on all transactions

Type the library name to see available commands.

- Tutorial no.17

LEARNETH

Tutorials list

Syllabus

10.1 Transactions - Ether and Wei

17 / 19

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

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.

Compile

`.sol` `arrays.sol` `mappings.sol` `structs.sol` `enums`

```
1 // SPDX-License-Identifier: MIT
2 //Aryan Patankar D20A 38
3 pragma solidity ^0.8.3;
4
5 contract EtherUnits {
6     uint public oneWei = 1 wei;
7     bool public isOneWei = 1 wei == 1;
8
9     uint public oneEther = 1 ether;
10    bool public isOneEther = 1 ether == 1e18;
11
12    //1. Create oneGWei and set it to 1 gwei
13    uint public oneGWei = 1 gwei;
14
15    // 2. Compare 1 gwei to 10^9 (10**9)
16    bool public isOneGWei = 1 gwei == 10**9;
17 }
18
```

✖ Explain contract

0 ☐ Listen on all transactions

`ETHER 3.0.2`

Type the library name to see available commands.

>

## • Tutorial no.18

**LEARNETH**

Tutorials list

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](https://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.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Aryan Patankar D20A 38
4
5 contract Gas {
6     uint public i = 0;
7
8
9     uint public cost = 170367;
10
11     function forever() public {
12         while (true) {
13             i += 1;
14         }
15     }
16 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash

• ethers.js

Type the library name to see available commands.  
creation of Gas pending...

[vm] from: 0x5B3...eddC4 to: Gas.(constructor) value: 0 wei data: 0x608...f0033 logs: 0  
hash: 0x199...729d7

status 1 Transaction mined and execution succeed

transaction hash 0x199c8f2705821ca54539f0a44c58e6fb49d36ae0919903b941bc9cc75cc729d7

## • Tutorial no.19

**LEARNETH**

Tutorials list

10.3 Transactions - Sending Ether  
19 / 19

If you change the parameter type for the functions `sendViaTransfer` and `sendViaSend` (line 33 and 38) from `payable address` to `address`, you won't be able to use `transfer()` (line 35) or `send()` (line 41).

Watch a video tutorial on Sending Ether.

### ★ Assignment

Build a charity contract that receives Ether that can be withdrawn by a beneficiary.

1. Create a contract called `Charity`.
2. Add a public state variable called `owner` of the type `address`.
3. Create a donate function that is public and payable without any parameters or function code.
4. Create a withdraw function that is public and sends the total balance of the contract to the `owner` address.

Tip: Test your contract by deploying it from one account and then sending Ether to it from another account. Then execute the withdraw function.

Check Answer Show answer

Next

Well done! No errors.

```
53 }
54 //Aryan Patankar D20A 38
55 contract Charity {
56     address public owner;
57
58     constructor() {
59         owner = msg.sender;
60     }
61
62     function donate() public payable {}
63
64     function withdraw() public {
65         uint amount = address(this).balance;
66
67         (bool sent, bytes memory data) = owner.call{value: amount}("");
68         require(sent, "Failed to send Ether");
69     }
70 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash

• ethers.js

Type the library name to see available commands.  
creation of Gas pending...

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

## **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.