

Experiment No. 4

Name: Atharva Prabhu

Batch: B

Div: D20A

Roll No: 45

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:
 - o **public**: available both inside and outside the contract.
 - o **private**: only accessible within the same contract.

- o internal: accessible within the contract and its child contracts.
 - o external: can be called only by external accounts or other contract
- **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.

Implementation:

- Tutorial no. 1 – Compile the code

The screenshot shows the Remix IDE interface. On the left, there's a sidebar for 'LEARNETH' with sections like 'Tutorials list', 'Syllabus', and '1. Introduction'. The main area displays a Solidity contract named 'Counter'. The code is as follows:

```
1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Counter {
6     uint public count;
7
8     // Function to get the current count
9     function get() public view returns (uint) { 2453 gas
10         return count;
11     }
12
13     // Function to increment count by 1
14     function inc() public { infinite gas
15         count += 1;
16     }
17
18     // Function to decrement count by 1
19     function dec() public { infinite gas
20         count -= 1;
21     }
22 }
```

At the bottom, there are status indicators: 'Scam Alert', 'Initialize as git repo', 'Did you know? To prototype on a uniswap v4 hooks, you can create a Multi Sig Swap Hook workspace. Template created by the cookbook team.', and 'RemixAI Copilot (enabled)'.

- Tutorial no. 1 – Deploy the contract

The screenshot shows the Remix IDE interface. On the left, the sidebar includes options for environment (Remix VM - Osaka), account (0x5B3...eddC4), gas limit (3000000), and value (0 Wei). The main area displays the Solidity code for a Counter contract:

```
// SPDX-License-Identifier: MIT
// Atharva Prabhu D20A
pragma solidity ^0.8.3;

contract Counter {
    uint public count;

    // Function to get the current count
    function get() public view returns (uint) {
        return count;
    }

    // Function to increment count by 1
    function inc() public {
        count += 1;
    }

    // Function to decrement count by 1
    function dec() public {
        count -= 1;
    }
}
```

Below the code, there is an "Explain contract" section and a deployment status bar indicating "Deploying...".

- Tutorial no. 1 – get

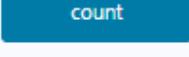
The screenshot shows the "Deployed Contracts" interface. It lists a single deployed contract: "COUNTER AT 0xD91...39138 (MEMO)". The contract has a balance of "0 ETH". Below the contract, four buttons represent its functions: "dec", "inc", "count", and "get". The "get" button is highlighted in blue. The result of the "get" call is displayed as "0: uint256: 0". At the bottom, there is a "Transact" button for interacting with the contract.

- Tutorial no. 1 – Increment

Deployed Contracts 1

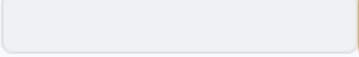
COUNTER AT 0xD91...39138 (MEMO)   

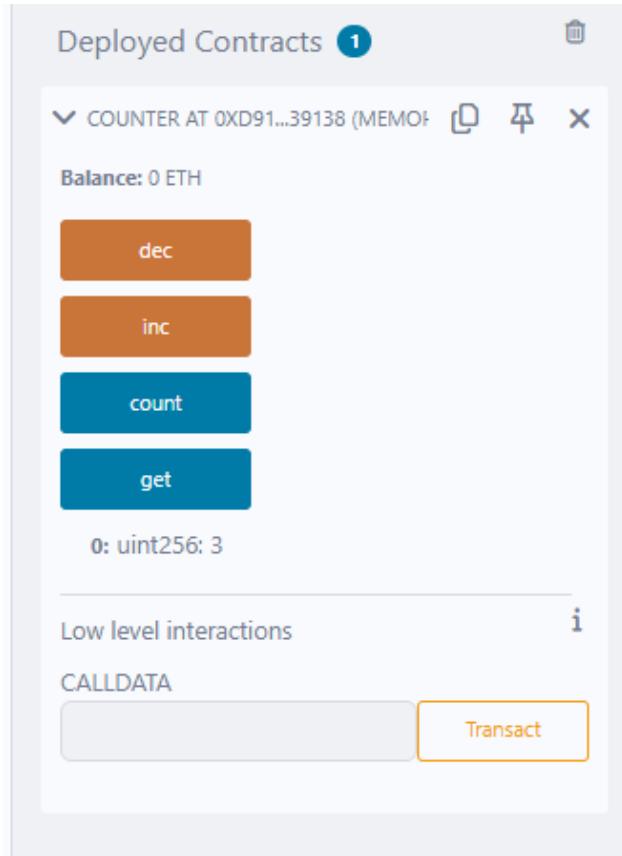
Balance: 0 ETH

0: uint256: 3

Low level interactions 

CALldata  

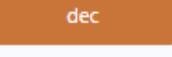
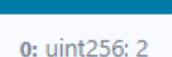


- Tutorial no. 1 – Decrement

Deployed Contracts 1

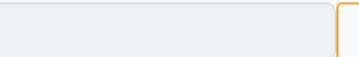
COUNTER AT 0xD91...39138 (MEMO)   

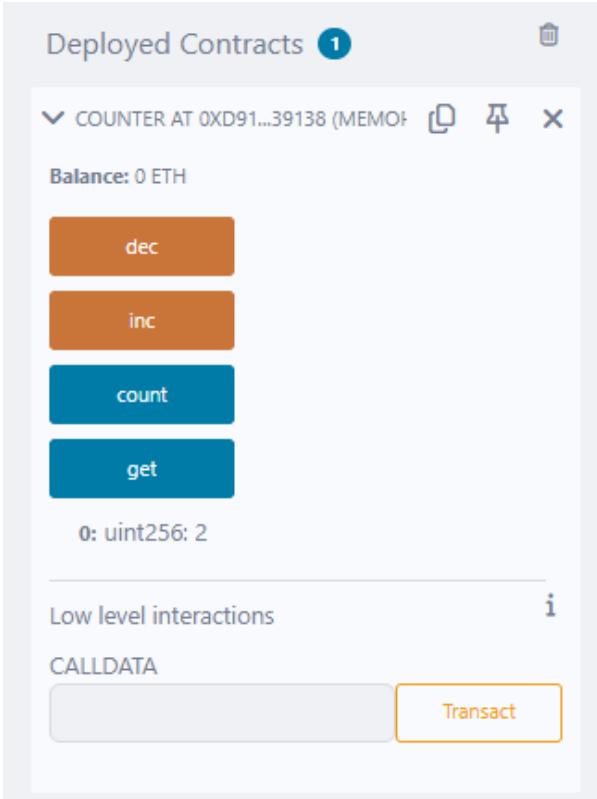
Balance: 0 ETH

0: uint256: 2

Low level interactions 

CALldata  



● Tutorial no. 2

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, a search bar, and a "Tutorials list" section. The main content area displays a "Syllabus" for "2. Basic Syntax". It includes a note about visibility, data types, and state variables, followed by a note linking to video tutorials for contracts. Below this is a "Assignment" section with the following tasks:

1. Delete the HelloWorld contract and its content.
2. Create a new contract named "MyContract".
3. The contract should have a public state variable called "name" of the type string.
4. Assign the value "Alice" to your new variable.

At the bottom of the assignment section are "Check Answer", "Show answer", and "Next" buttons. A green banner at the bottom says "Well done! No errors.".

On the right side of the interface, the code editor shows the following Solidity code:

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and less than 0.9.0
// Atharva Prabhu D20A
pragma solidity ^0.8.3;

contract MyContract {
    // We declare a public string variable named 'name'
    // and assign it the value "Alice"
    string public name = "Alice";
}
```

● Tutorial no. 3

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, a search bar, and a "Tutorials list" section. The main content area displays a "Syllabus" for "3. Primitive Data Types". It includes a note about structs, followed by a note linking to a video tutorial on primitive data types. Below this is a "Assignment" section with the following tasks:

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`.

Below the assignment section is a tip: "Tip: Look at the other address in the contract or search the internet for an Ethereum address." At the bottom of the assignment section are "Check Answer", "Show answer", and "Next" buttons. A green banner at the bottom says "Well done! No errors.".

On the right side of the interface, the code editor shows the following Solidity code:

```
// SPDX-License-Identifier: MIT
// Atharva Prabhu D20A
pragma solidity ^0.8.3;

contract Primitives {
    string public name = "Alice";

    // 1. A public address variable with a unique value
    address public newAddr = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;

    // 2. A signed integer to hold a negative number
    // int8 is sufficient for small negative numbers and saves space
    int8 public neg = -50;

    // 3. The smallest uint size (uint8) and smallest value (0)
    uint8 public newU = 0;
}
```

● Tutorial no. 4

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

- Create a new public state variable called `blockNumber`.
- 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.

Code Snippet:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Variable {
6     uint256 public blockNumber;
7     string public name;
8     uint public n;
9 }
10
11 uint public blockNumber;
12
13 function doSomething() public {
14     // Local variable (Exists only during function execution)
15     uint i = 456;
16
17     // 2. Use a Global Variable to get the current block number
18     // 'block.number' is the global variable you're looking for!
19     blockNumber = block.number;
20
21     // Other examples of global variables:
22     uint timestamp = block.timestamp; // Current block timestamp
23     address sender = msg.sender; // Address of the person calling this function
}

```

● Tutorial no. 5

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

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

Code Snippet:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract SimpleStorage {
6     // State variable to store a number
7     uint public num;
8
9     // 1. Create a public state variable 'b' initialized to true
10    bool public b = true;
11
12    // Function that changes the state (Writing)
13    function set(uint _num) public {
14        num = _num;
15    }
16
17    // 2. Create a public function 'get_b' that returns the value of 'b'
18    // We use 'view' because we are reading, not changing, the state.
19    function get_b() public view returns (bool) {
20        return b;
21    }
}

```

● Tutorial no. 6

You can declare a pure function using the keyword `pure`. In this contract, `add` (line 13) is a pure function. This function takes the parameters `i` and `j`, and returns the sum of them. It neither reads nor modifies 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**

Well done! No errors.

https://remix.ethereum.org/plugins/learneth/index.html#/home

Did you know? You can use the help of AI for Solidity error, click on 'Ask RemixAI'.

RemixAI Copilot (enabled)

● Tutorial no. 7

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

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**

Well done! No errors.

⚠ Scan Alert Initialize as git repo

Did you know? You can use the help of AI for Solidity error, click on 'Ask RemixAI'.

RemixAI Copilot (enabled)

● Tutorial no. 8

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, search, and navigation. The main area displays a tutorial titled "5.4 Functions - Inputs and Outputs" (3/19). The content discusses arrays as parameters and return values. A note about gas consumption is present. Below the text is a section titled "Assignment" with a task: "Create a new function called `returnTwo` that returns the values `-2` and `true` without using a return statement." It includes "Check Answer" and "Show answer" buttons. A message at the bottom says "Well done! No errors." At the top right, there are tabs for "learneth tutorials" and "inputsAndOutputs.sol", along with "Compile" and "Run" buttons. The status bar at the bottom shows "Scan Alert", "Initialize as git repo", "Did you know?", and "RemixAI Copilot (enabled)".

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Function {
6     // Functions can return multiple values.
7     function returnMany() payable infinite gas
8         public
9         pure
10        returns (
11             uint,
12             bool,
13             uint
14         )
15     {
16         return (1, true, 2);
17     }
18
19     // Return values can be named.
20     function named() payable infinite gas
21         public
22         pure
23         returns (
24             uint x,
25             bool b,
26             uint y
27         )
28 }

```

● Tutorial no. 9

The screenshot shows the REMIX IDE interface. The left sidebar has icons for file operations, search, and navigation. The main area displays a tutorial titled "6. Visibility" (9/19). It explains the `visibility` specifier and its four types: `external`, `public`, `internal`, and `private`. It notes that they regulate access from contracts, child contracts, and other contracts. Below this, there are sections for `private`, `internal`, and `public`, each with bullet points. A note at the bottom says "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." At the top right, there are tabs for "learneth tutorials" and "visibility.sol", along with "Compile" and "Run" buttons. The status bar at the bottom shows "Scan Alert", "Initialize as git repo", "Did you know?", and "RemixAI Copilot (enabled)".

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Base {
6     // Private function can only be called
7     // - inside this contract
8     // Contracts that inherit this contract cannot call this function.
9     function privateFunc() private pure returns (string memory) {
10         return "private function called";
11     }
12
13     function testPrivateFunc() public pure returns (string memory) {
14         return "private function called";
15     }
16
17     // Internal function can be called from 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 }

```

● Tutorial no. 10

REMX 1.5.1

learneth tutorials

7.1 Control Flow - If/Else
10 / 19

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**

Well done! No errors.

Scam Alert Initialize as git repo Did you know? To prototype on a uniswap v4 hooks, you can create a Multi Sig Swap Hook workspace. Template created by the cookbook team. RemixAI Copilot (enabled)

● Tutorial no. 11

REMX 1.5.1

learneth tutorials

7.2 Control Flow - Loops
11 / 19

Watch a video tutorial on Loop statements.

Assignment

- Create a public `uint` state variable called `count` in the `Loop` contract.
- At the end of the for loop, increment the `count` variable by 1.
- 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**

Well done! No errors.

Scam Alert Initialize as git repo Did you know? To prototype on a uniswap v4 hooks, you can create a Multi Sig Swap Hook workspace. Template created by the cookbook team. RemixAI Copilot (enabled)

● Tutorial no. 12

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar titled "LEARNETH" with sections like "Tutorials list", "Syllabus", and "8.1 Data Structures - Arrays". The main content area displays a Solidity code editor with the following code:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.0;
4
5 contract Array {
6     // contract Array is Array
7     // uint
8     uint
9     remmix-project-org/remix-workshops/8.1 Data Structures - Arrays/arrays.sol 4
10    //
11    // Note: We use "pure" if we weren't reading state,
12    // but since we are reading arr3, we use "view".
13    function getArr() public view returns (uint[3] memory) {
14        return arr3;
15    }
16 }

```

Below the code editor, there are two buttons: "Check Answer" and "Show answer". A green bar at the bottom says "Well done! No errors.".

● Tutorial no. 13

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar titled "LEARNETH" with sections like "Tutorials list", "Syllabus", and "8.2 Data Structures - Mappings". The main content area displays a Solidity code editor with the following code:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
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) {
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 {
16         // Update the value at this address
17         balances[_addr] = _addr.balance;
18     }
19
20     function remove(address _addr) public {
21         // Reset the value to the default value.
22         delete balances[_addr];
23     }
24 }
25
26 contract NestedMapping {
27     // Nested mapping (mapping from address to another mapping)
28 }

```

Below the code editor, there are two buttons: "Check Answer" and "Show answer". A green bar at the bottom says "Well done! No errors.".

● Tutorial no. 14

The screenshot shows the REMIX IDE interface. On the left, the LEARNETH tutorial sidebar for '8.3 Data Structures - Structs' is visible, showing sections like 'Accessing structs' and 'Updating structs'. The main workspace displays a Solidity code editor with the following code:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Todos {
6     struct Todo {
7         string text;
8         bool completed;
9     }
10
11     // An array of 'Todo' structs
12     Todo[] public todos;
13
14     function create(string memory _text) public {
15         // 3 ways to initialize a struct
16         // - calling it like a function
17         todos.push(Todo(_text, false));
18
19         // key value mapping
20         todos.push(Todo({_text, completed: false}));
21
22         // initialize an empty struct and then update it
23         Todo memory todo;
24         todo.text = _text;
25         // todo.completed initialized to false
26
27         todos.push(todo);
    }
  
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A message says 'Well done! No errors.' and a note about initializing as a git repo.

● Tutorial no. 15

The screenshot shows the REMIX IDE interface. On the left, the LEARNETH tutorial sidebar for '8.4 Data Structures - Enums' is visible, showing sections like 'Removing an enum value'. The main workspace displays a Solidity code editor with the following code:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Enum {
6     // Enum representing shipping status
7     enum Status {
8         Pending,
9         Shipped,
10        Accepted,
11        Rejected,
12        Canceled
13    }
14
15    enum Size {
16        S,
17        M,
18        L
19    }
20
21    // Default value is the first element listed in
22    // definition of the type, in this case "Pending"
23    Status public status;
24    Size public sizes;
25
26    function get() public view returns (Status) {
27        return status;
    }
  
```

At the bottom, there are buttons for 'Check Answer', 'Show answer', and 'Next'. A message says 'Well done! No errors.' and a note about initializing as a git repo.

● Tutorial no. 16

The screenshot shows the Remix IDE interface. On the left, the LEARNETH syllabus is displayed with the current section being "9. Data Locations". The main workspace contains the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract DataLocations {
6     uint[] public arr;
7     mapping(uint => address) map;
8     struct MyStruct {
9         uint foo;
10    }
11    mapping(uint => MyStruct) public myStructs;
12
13 function f() public returns (MyStruct memory, MyStruct memory, MyStruct memory){
14     // call _f with state variables
15     _f(arr, map, myStructs[1]);
16     // get a struct from a mapping
17     MyStruct storage myStruct = myStructs[1];
18     myStruct.foo = 4;
19     // create a struct in memory
20     MyStruct memory myMemStruct = MyStruct(0);
21     MyStruct memory myMemStruct2 = myMemStruct;
22     myMemStruct2.foo = 1;
23
24     MyStruct memory myMemStruct3 = myStruct;
25     myMemStruct3.foo = 3;
26     return (myStruct, myMemStruct2, myMemStruct3);
27 }

```

Below the code, there are two buttons: "Check Answer" and "Show answer". A tip at the bottom says: "Tip: Make sure to create the correct return types for the function `f`". A message at the bottom of the screen says: "Well done! No errors."

● Tutorial no. 17

The screenshot shows the Remix IDE interface. On the left, the LEARNETH syllabus is displayed with the current section being "10.1 Transactions - Ether and Wei". The main workspace contains the following Solidity code:

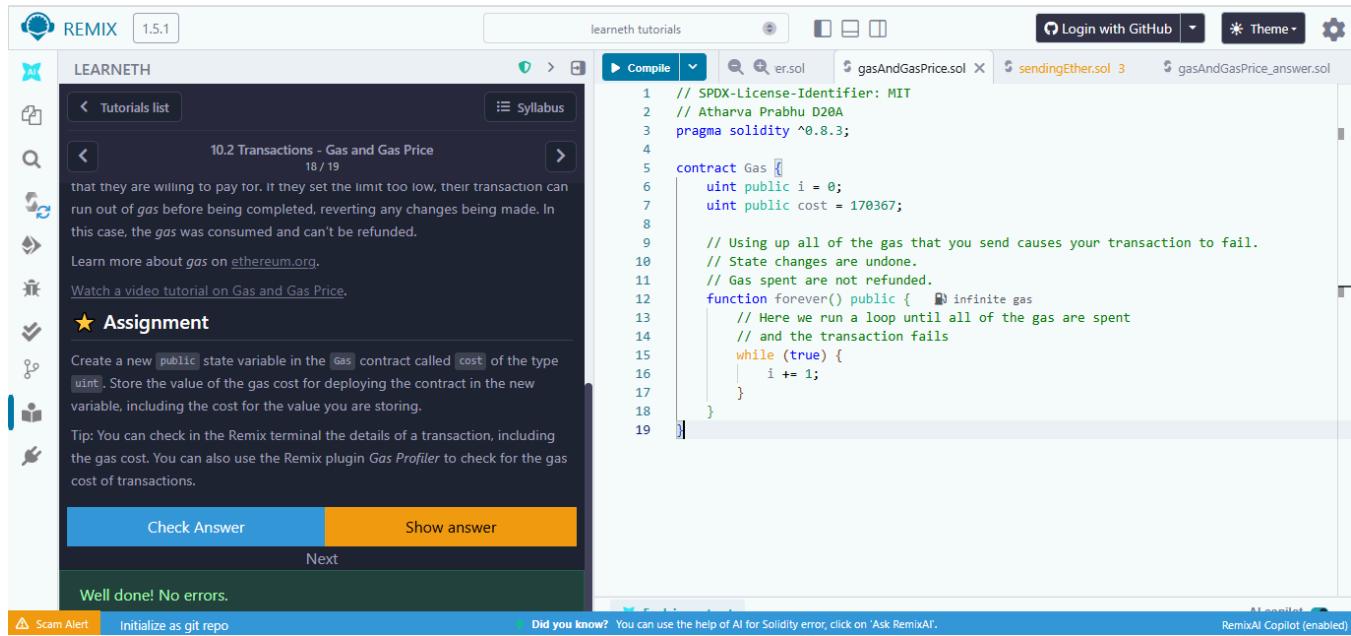
```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract EtherUnits {
6     uint public oneWei = 1 wei;
7     // 1 wei is equal to 1
8     bool public isOneWei = 1 wei == 1;
9
10    uint public oneEther = 1 ether;
11    // 1 ether is equal to 10^18 wei
12    bool public isOneEther = 1 ether == 1e18;
13
14    uint public oneGwei = 1 gwei;
15    // 1 ether is equal to 10^9 wei
16    bool public isOneGwei = 1 gwei == 1e9;
17 }

```

Below the code, there are two buttons: "Check Answer" and "Show answer". A tip at the bottom says: "Tip: Look at how this is written for `gwei` and `ether` in the contract". A message at the bottom of the screen says: "Well done! No errors."

● Tutorial no. 18



The screenshot shows the REMIX IDE interface. On the left, there's a sidebar for 'LEARNETH' with a 'Tutorials list' section. The main area displays a tutorial titled '10.2 Transactions - Gas and Gas Price' (18 / 19). The text explains that if the gas limit is set too low, the transaction will run out of gas before completing, reverting any changes. It also links to more information on Ethereum.org and provides a video tutorial.

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.

Buttons at the bottom: 'Check Answer' (blue), 'Show answer' (orange), and 'Next'.

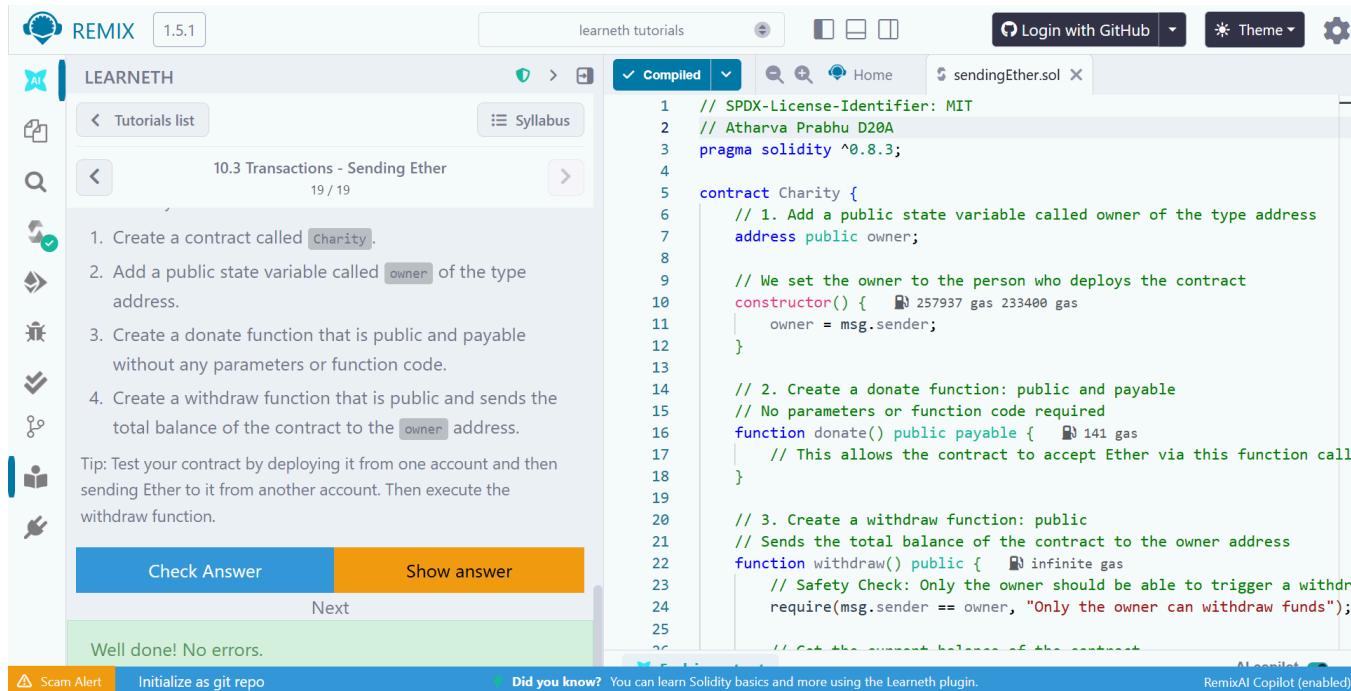
Code editor on the right shows the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Gas {
6     uint public i = 0;
7     uint public cost = 170367;
8
9     // Using up all of the gas that you send causes your transaction to fail.
10    // State changes are undone.
11    // Gas spent are not refunded.
12    function forever() public {
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 }
```

At the bottom of the interface, there are buttons for 'Scan Alert', 'Initialize as git repo', 'Did you know?', and 'RemixAI Copilot (enabled)'.

● Tutorial no. 19



The screenshot shows the REMIX IDE interface. On the left, there's a sidebar for 'LEARNETH' with a 'Tutorials list' section. The main area displays a tutorial titled '10.3 Transactions - Sending Ether' (19 / 19).

Assignment:

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.

Buttons at the bottom: 'Check Answer' (blue), 'Show answer' (orange), and 'Next'.

Code editor on the right shows the following Solidity code:

```

1 // SPDX-License-Identifier: MIT
2 // Atharva Prabhu D20A
3 pragma solidity ^0.8.3;
4
5 contract Charity {
6     // 1. Add a public state variable called owner of the type address
7     address public owner;
8
9     // We set the owner to the person who deploys the contract
10    constructor() {
11        owner = msg.sender;
12    }
13
14    // 2. Create a donate function: public and payable
15    // No parameters or function code required
16    function donate() public payable {
17        // This allows the contract to accept Ether via this function call
18    }
19
20    // 3. Create a withdraw function: public
21    // Sends the total balance of the contract to the owner address
22    function withdraw() public {
23        // infinite gas
24        // Safety Check: Only the owner should be able to trigger a withdraw
25        require(msg.sender == owner, "Only the owner can withdraw funds");
26        // Get the current balance of the contract
27    }
28 }
```

At the bottom of the interface, there are buttons for 'Scan Alert', 'Initialize as git repo', 'Did you know?', and 'RemixAI Copilot (enabled)'.

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.