

Experiment :04
Blockchain Lab

Aim: Hands on Solidity Programming Assignments for creating Smart Contracts

Theory:

1. Primitive Data Types and Variables in Solidity

Solidity provides several primitive data types that serve as the building blocks for smart contract development:

- **Integer Types (uint, int)**

Used to store numeric values. uint stores unsigned integers (non-negative), while int stores signed integers. Commonly used sizes include uint256, which is optimized for Ethereum Virtual Machine (EVM).

- **Boolean (bool)**

Stores logical values true or false, typically used in conditions and decision-making.

- **Address (address)**

Represents a 20-byte Ethereum account or contract address. It is commonly used for identifying users, transferring Ether, and enforcing ownership.

- **Bytes and String (bytes, string)**

bytes is used for raw binary data, while string is used for storing textual information.

Variables in Solidity are classified as:

- **State Variables:** Stored permanently on the blockchain.

- **Local Variables:** Declared inside functions and exist temporarily.

- **Global Variables:** Predefined variables like msg.sender, msg.value, and block.timestamp that provide blockchain-related information.

2. Functions and Function Types

Functions define the logic of a smart contract. They can accept inputs, perform computations, and return outputs.

Special function types include:

- **pure functions:** Do not read or modify state variables. They only use input parameters and internal logic.

- **view functions:** Can read state variables but cannot modify them.

Using pure and view functions helps reduce gas costs and improves contract efficiency.

3. Inputs and Outputs to Functions

Solidity functions can take one or more input parameters and return one or more output values. Inputs allow data to be passed into the contract, while outputs return computed results to the caller.

Functions can also use **named return variables**, which enhance code readability and simplify debugging. This feature is especially useful when returning complex data structures.

4. Function Visibility, Modifiers, and Constructors

Function Visibility

Solidity provides access control through visibility specifiers:

- **public:** Can be accessed both internally and externally.
- **private:** Accessible only within the same contract.
- **internal:** Accessible within the contract and derived contracts.
- **external:** Can only be called from outside the contract.

Modifiers

Modifiers are reusable pieces of code used to alter function behavior. They are commonly used to enforce conditions such as ownership or authorization (e.g., `onlyOwner`).

Constructors

A constructor is a special function executed only once during contract deployment. It is typically used to initialize state variables, such as assigning the deployer as the contract owner.

5. Control Flow Statements

Solidity supports standard control flow mechanisms:

- **Conditional Statements (if-else)**

Used to execute code based on conditions, such as validating balances before transactions.

- **Loops (for, while, do-while)**

Used to repeat execution of code blocks. However, excessive looping increases gas consumption, so loops must be used carefully in smart contracts.

6. Data Structures in Solidity

Solidity provides several data structures for efficient data management:

- **Arrays**

Store ordered collections of elements. Arrays can be fixed-size or dynamic.

- **Mappings**

Store key-value pairs for fast lookups, such as mapping(address => uint) for account balances. Mappings do not support iteration.

- **Structs**

Group related variables into a single custom data type, improving code organization.

- **Enums**

Define a set of named constants, enhancing code readability and reducing logical errors.

7. Data Locations

Solidity uses three data locations to manage variable storage:

- **storage**: Permanent data stored on the blockchain (state variables).

- **memory**: Temporary data used during function execution.

- **calldata**: Read-only and gas-efficient location used for external function parameters.

Choosing the correct data location is crucial for optimizing gas usage and contract performance.

8. Transactions, Ether, and Gas

- **Ether and Wei**

Ether is the native cryptocurrency of Ethereum. All values are internally handled in Wei, where

1 Ether = 10^{18} Wei, allowing precise financial calculations.

- **Gas and Gas Price**

Gas measures the computational effort required to execute transactions or smart contracts. The gas price determines how much Ether is paid per unit of gas.

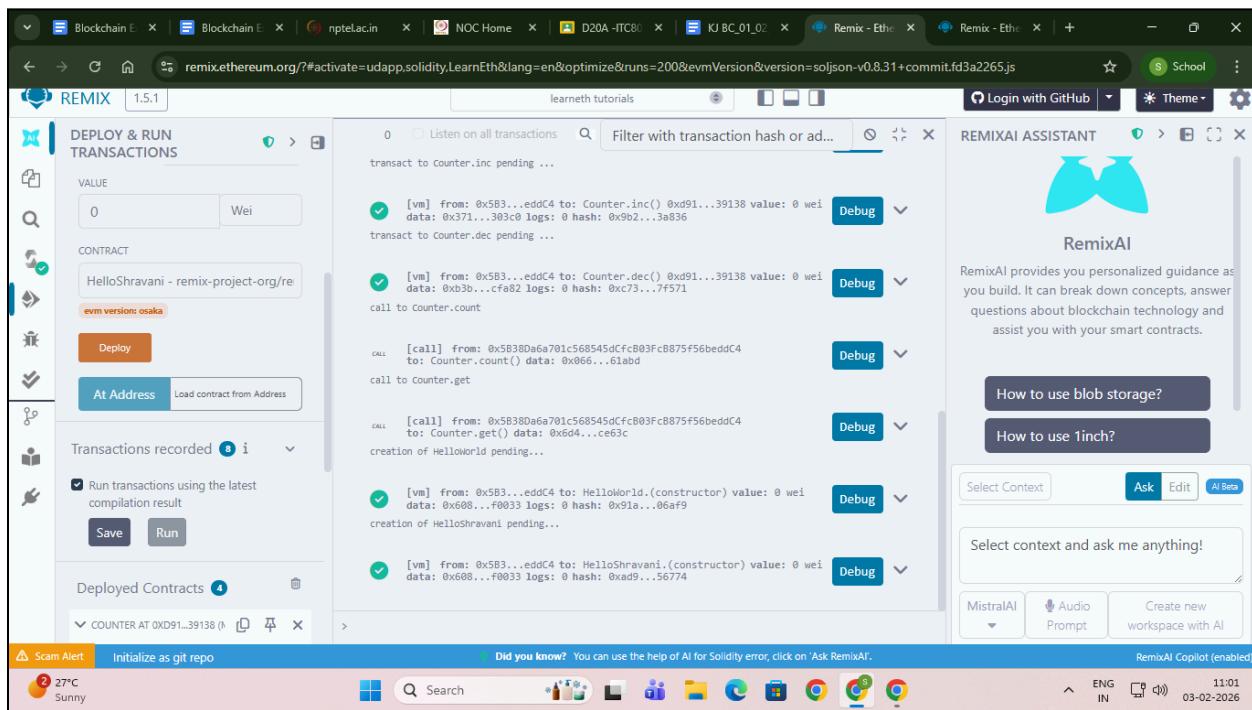
- **Sending Transactions**

Transactions are used to transfer Ether or interact with smart contracts. Methods like transfer(), send(), and call() are used, with call() offering greater flexibility and control.

Efficient contract design minimizes gas consumption and ensures secure execution.

1. basic.sol

```
2  //Shravani Patil - 40
3  // compiler version must be greater than or equal to 0.
4  pragma solidity ^0.8.3;
5
6  contract HelloWorld {
7      string public greet = "Hello World!";
8  }
9
10 contract HelloShravani {
11     string public student ="Shravani Patil-40";
12 }
```



2. introduction.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
// SPDX-License-Identifier: MIT
//Shravani Patil - 40
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;
    }
}
```

The screenshot shows the Remix Ethereum IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is open, showing the deployed contract 'Counter' at address 0x0D91...39138. It has three pending transactions: 'inc()', 'dec()', and 'get()'. The 'inc()' transaction is successful, showing a log message. The 'dec()' and 'get()' transactions are pending. Below these, there are sections for 'Transactions recorded' and 'Deployed Contracts'. On the right, the main workspace shows the Solidity code of the Counter contract. A 'REMIXAI ASSISTANT' panel is visible on the far right, providing personalized guidance and AI assistance.

3. PrimitiveDataTypes.col

The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various icons and a search bar. The main area displays a tutorial titled "3. Primitive Data Types" from "LEARNETH". Below the title, it says "Watch a video tutorial on Primitive Data Types." and "Assignment". The assignment tasks are:

- 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`.
- Create a `public` variable called `neg` that is a negative number, decide upon the type.
- Create a new variable, `newU` that has the smallest `uint` size type and the smallest `uint` value and is `public`.

A tip at the bottom suggests looking at other addresses in the contract or searching the internet for Ethereum addresses. At the bottom of the assignment section, there are "Check Answer" and "Show answer" buttons, and a message "Well done! No errors."

On the right side of the interface, there is a code editor with Solidity code, a compiler status bar, and a transaction history pane. The code editor contains the following Solidity code:

```
/*
Negative numbers are allowed for int types.
Like uint, different ranges are available from int8 to int256
*/
int8 public i8 = -1;
int public i256 = 456;
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 = 0xb3b...cfab2;
int8 public neg = -1;
uint public newU = 0;
```

The transaction history pane shows several transactions, including calls to `Counter` and `HelloWorld` contracts, with details like from address, to address, value, data, logs, and hash.

This screenshot shows the "DEPLOY & RUN TRANSACTIONS" section of the Remix IDE. On the left, there are buttons for "Deploy" and "At Address", and a dropdown for "Transactions recorded". Below these are buttons for "Save" and "Run".

The main area displays a list of deployed contracts and their transaction details. The contracts listed are:

- `COUNTER` at address `0xD91...39138`
- `Primitives` (remix-project-org/remix)
- `HelloWorld`
- `Primitives`
- `COUNTER` at address `0xD91...39138`

Each contract entry shows a list of transactions with details such as from address, to address, value, data, logs, and hash. There are "Debug" buttons next to each transaction entry.

At the bottom, there are "Scan Alert" and "Initialize as git repo" buttons, along with a weather widget showing "27°C Sunny".

4. Variables.sol

The screenshot shows the Remix IDE interface. On the left, there's a sidebar for 'LEARNETH' with a 'Tutorials list' and a section for '4. Variables'. It contains text about using `block.timestamp` and `msg.sender`, a list of global variables, and an assignment section with two tasks. A tip about reading the current block number is also present. Below this is a 'Check Answer' button, a 'Show answer' button, and a 'Next' link. A green banner at the bottom says 'Well done! No errors.' On the right, the code editor displays the `Variables.sol` contract:

```
4 contract Variables {
5     // State variables are stored on the blockchain.
6     string public text = "Hello";
7     uint public num = 123;
8     uint public blockNumber;
9     string public name ="Shravani Anil Patil-40";
10
11    function doSomething() public {
12        // Local variables are not saved to the blockchain.
13        uint _i = 456;
14        blockNumber= block.number;
15
16        // Here are some global variables
17        uint timestamp = block.timestamp; // Current block timestamp
18        address sender = msg.sender; // address of the caller
19
20    }
21
22
23 }
```

At the bottom, there are 'Scan Alert' and 'Initialize as git repo' buttons, a 'Did you know?' section, and an 'Ask RemixAI' button. The system status bar shows it's 29°C, sunny, and the date is 03-02-2026.

This screenshot shows the Remix IDE's deployment and transaction history. The left sidebar has a 'DEPLOY & RUN TRANSACTIONS' section with fields for 'GAS LIMIT' (set to 'Estimated Gas'), 'VALUE' (set to 0 Wei), and a 'CONTRACT' section for 'Variables - remix-project-org/variables'. It includes an 'evm version: osaka' dropdown and a 'Deploy' button. Below these are 'Transactions recorded' (11) and options to 'Run transactions using the latest compilation result', 'Save', and 'Run'. At the bottom, there are 'Scan Alert' and 'Initialize as git repo' buttons, a 'Did you know?' section, and an 'Ask RemixAI' button. The system status bar shows it's 29°C, sunny, and the date is 03-02-2026.

The main pane shows a list of transactions:

- call to Counter.get
- [call] from: 0x5B3...eddC4 to: Counter.get() data: 0x604...ce63c creation of HelloWorld pending...
- [vm] from: 0x5B3...eddC4 to: HelloWorld.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x91a...06af9 creation of HelloWorld pending...
- [vm] from: 0x5B3...eddC4 to: HelloShravani.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xad9...56774 creation of Primitives pending...
- [vm] from: 0x5B3...eddC4 to: Primitives.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x910...06203 creation of Primitives pending...
- [vm] from: 0x5B3...eddC4 to: Variables.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x47a...28909 creation of Variables pending...
- [vm] from: 0x5B3...eddC4 to: Variables.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xe8c...4c525 creation of Variables pending...

Each transaction entry has a 'Debug' button to its right.

5. readAndWrite.sol

The screenshot shows the Remix IDE interface. On the left, there's a sidebar for 'LEARNETH' with sections like 'Tutorials list', 'Syllabus', and an 'Assignment' section containing two tasks. The main area displays the Solidity code for 'readAndWrite.sol'. The code defines a state variable 'num' and functions to set and get its value, along with a function to return the owner's name. Below the code, there's an 'Explain contract' button and a transaction log table.

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

contract SimpleStorage {
    // State variable to store a number
    uint public num;
    bool public b = true;
    string public name="Shravani Anil Patil-40";
    // You need to send a transaction to write to a state variable.
    function set(uint _num) public {
        num = _num;
    }

    // You can read from a state variable without sending a transaction.
    function get() public view returns (uint) {
        return num;
    }

    function get_b() public view returns (bool) {
        return b;
    }
    // function getName() public view returns (string) {
    //     return name;
    }
}
```

This screenshot shows the Remix IDE after deploying the 'SimpleStorage' contract. The left sidebar shows the deployed contract at address 0x5B3...eddC4. The main pane displays the transaction logs for the deployment and subsequent calls to the contract's functions. The logs show the creation of the HelloWorld, HelloShravani, Primitives, Variables, and Simplestorage contracts, along with their respective constructor parameters and gas usage.

creation of HelloWorld pending...

[vm] from: 0x5B3...eddC4 to: HelloWorld.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x91a...06af9

creation of HelloShravani pending...

[vm] from: 0x5B3...eddC4 to: HelloShravani.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xad9...56774

creation of Primitives pending...

[vm] from: 0x5B3...eddC4 to: Primitives.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x910...06203

creation of Variables pending...

[vm] from: 0x5B3...eddC4 to: Variables.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0x47a...28909

creation of Simplestorage pending...

[vm] from: 0x5B3...eddC4 to: SimpleStorage.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xe8c...4c525

6. ViewAndPure.sol

The screenshot shows the Remix IDE interface with the `ViewAndPure.sol` contract deployed and interacting with it.

Deployed Contracts: `VIEWANDPURE AT 0x534...0B6`

Balance: 0 ETH

ADDTOX2: `y: "7"`

ADD: `i: "7"`, `j: "8"`

Contract Code:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract ViewAndPure {
5     uint public x = 1;
6     string public name = "Shravani Anil Patil";
7
8     // Promise not to modify the state.
9     function addToX(uint y) public view returns (uint) {
10         return x + y;
11     }
12
13     // Promise not to modify or read from the state.
14     function add(uint i, uint j) public pure returns (uint) {
15         return i + j;
16     }
17     function addToX2(uint y) public {
18         x = x+y;
19     }
20 }
21 
```

Explain contract:

0 Listen on all transactions

7. ModifiersandConstructors.sol

```
require(!locked, "No reentrancy");

locked = true;
_;
locked = false;

function decrement(uint i) public noReentrancy {
    x -= i;

    if (i > 1) {
        decrement(i - 1);
    }
}

function increaseX(uint y) public{
    x=x+y;
}
```

8. inputsandOutputs.sol

```
// Can use array for output
uint[] public arr;

function arrayOutput() public view returns (uint[] memory) {
    return arr;
}

function returnTwo() public pure
returns (
    int i,
    bool b
) {
    i = -2;
    b = true;
}
```

9. Visibility.sol

The screenshot shows the Truffle IDE interface. The top navigation bar includes tabs for 'tors.sol', 'inputsAndOutputs.sol', 'inputsAndOutputs_answer.sol', and 'visibility.sol'. The main area displays the Solidity code for 'tors.sol'.

```
// and state variables.  
// function testPrivateFunc() public pure returns (string memory) {  
//     return privateFunc();  
// }  
  
// Internal function call be called inside child contracts.  
function testInternalFunc() public pure override returns (string memory) {  
    return internalFunc();  
}  
//Shravani Anil Patil-40  
function testInternalVar() public view returns (string memory, string memory) {  
    return (internalVar, publicVar);  
}
```

The left sidebar shows deployed contracts and their balances. It includes buttons for interacting with the contract: 'externalFunc', 'publicFunc', 'publicVar', 'testInternalFunc', 'testPrivateFunc', and log outputs for each function call.

Logs for 'externalFunc':
0: string: my public variable

Logs for 'testInternalFunc':
0: string: internal function called

Logs for 'testPrivateFunc':
0: string: private function called

10.IfElse.sol

The screenshot shows the Truffle UI interface for the `IfElse.sol` contract. On the left, there are two tabs: `EVENCHECK` and `FOO`. The `EVENCHECK` tab has a parameter `y: "8"` and a call button. The `FOO` tab has a parameter `x: "7"` and a call button. Below these tabs, the state is shown as `0: bool: true` and `0: uint256: 0`. A dropdown menu is open, showing `ternary` selected, with a sub-menu option `uint256 _x`. At the bottom, there is a `Transact` button. The right side of the screen displays the Solidity code for the `IfElse.sol` contract:

```
11     return z;
12 }
13 }
14
15 function ternary(uint _x) public pure returns (uint) {    infinite gas
16     // if (_x < 10) {
17     //     return 1;
18     // }
19     // return 2;
20
21     // shorthand way to write if / else statement
22     return _x < 10 ? 1 : 2;
23 }
24 //Shravani Anil Patil-40
25 function evenCheck(uint y) public pure returns (bool) {    infinite gas
26     return y%2 == 0 ? true : false;
27 }
```

11. Loops.sol

The screenshot shows the Truffle UI interface for the `Loops.sol` contract. On the left, there is a `DEPLOY & RUN TRANSACTIONS` section with a `Deploy` button and an `At Address` dropdown set to `Load contract from Address`. Below this, there is a `Transactions recorded` section with a count of `29` and a `i` variable value of `29`. Further down is a `Deployed Contracts` section with a count of `1`, showing a deployed contract named `LOOP AT 0X36C...CA07B (MEM)`. The `Balance: 0 ETH` is shown. A `loop` button is highlighted in orange, and a `count` button is shown in blue. The state is indicated as `0: uint256: 9`. At the bottom, there is an `Explain contract` button. The right side of the screen displays the Solidity code for the `Loops.sol` contract:

```
6     function loop() public{    infinite gas
7         // for loop
8         for (uint i = 0; i < 10; i++) {
9             if (i == 5) {
10                 // Skip to next iteration with continue
11                 continue;
12             }
13             if (i == 5) {
14                 // Exit loop with break
15                 break;
16             }
17             count++;
18         }
19 // Shravani Patil-40
20 // while loop
21         uint j;
22         while (j < 10) {
23             j++;
24         }
25     }
26 }
27 }
```

12. Arrays.sol

The screenshot shows the REMIX IDE interface. On the left, the contract interface for `ARRAY AT 0xD20...D2358` is displayed, showing functions like `pop`, `push` (with parameter `i: 0`), `remove` (with parameters `uint256 index` and `arr`), `getArr` (with parameter `uint256 i`), `getLength` (returning `0: uint256[3]: 0,1,2`), and `myFixedSizeArr` (with parameter `uint256`).

On the right, the Solidity code for `Arrays.sol` is shown:

```
//Shravani Patil-40
function test() public {
    infinite gas
    arr.push(1);
    arr.push(2);
    arr.push(3);
    arr.push(4);
    // [1, 2, 3, 4]

    remove(1);
    // [1, 4, 3]

    remove(2);
    // [1, 4]
}
```

13. Mappings.sol

The screenshot shows the REMIX IDE interface. On the left, the assignment interface for "8.2 Data Structures - Mappings" is displayed, showing instructions to create a mapping, change `get` and `remove` functions, and change `set` to update the mapping.

On the right, the Solidity code for `Mappings.sol` is shown:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Shravani Patil-40
contract Mapping {
    // Mapping from address to uint
    mapping(address => uint) public balances;

    function get(address _addr) public view returns (uint) {
        // 2872 gas
        remix-project-org/remix-workshops/8.2 Data Structures - Mappings/mappings.sol:10
    }

    function set(address _addr) public {
        // Update the value at this address
        balances[_addr] = _addr.balance;
    }

    function remove(address _addr) public {
        // Reset the value to the default value.
        delete balances[_addr];
    }
}

contract NestedMapping {
    // Nested mapping (mapping from address to another mapping)
    mapping(address => mapping(uint => bool)) public nested;
}
```

14.Struct.sol

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with navigation links like 'Tutorials list' and 'Syllabus'. The main area displays a Solidity contract named 'Todos'. The code defines a struct 'Todo' with fields 'text' (string) and 'completed' (bool). It then declares an array of 'Todo' structs named 'todos'. A function 'create' takes a string parameter '_text' and pushes a new 'Todo' object to the 'todos' array. Another function initializes an empty struct and updates it. At the bottom, there are buttons for 'Check Answer' and 'Show answer'.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Shravani Patil
contract Todos {
    struct Todo {
        string text;
        bool completed;
    }
    // An array of 'Todo' structs
    Todo[] public todos;
}

function create(string memory _text) public {
    // 3 ways to initialize a struct
    // - calling it like a function
    todos.push(Todo(_text, false));
    // key value mapping
    todos.push(Todo({_text, completed: false}));
    // initialize an empty struct and then update it
    Todo memory todo;
    todo.text = _text;
    // todo.completed initialized to false
    todos.push(todo);
}
```

15.enums.sol

The screenshot shows the REMIX IDE interface. The sidebar has a link to '8.4 Data Structures - Enums'. The main area shows a Solidity contract named 'Enum'. It defines two enums: 'Status' with values 'Pending', 'Shipped', 'Accepted', 'Rejected', and 'Canceled'; and 'Size' with values 'S', 'M', and 'L'. It also defines a variable 'status' of type 'Status' and an array 'sizes' of type 'Size'. A function 'get()' returns the value of 'status'. Buttons for 'Check Answer' and 'Show answer' are at the bottom.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Shravani Patil
contract Enum {
    // Enum representing shipping status
    enum Status {
        Pending,
        Shipped,
        Accepted,
        Rejected,
        Canceled
    }
    enum Size {
        S,
        M,
        L
    }
    // Default value is the first element listed in
    // definition of the type, in this case "Pending"
    Status public status;
    Size public sizes;
}

function get() public view returns (Status) {
    return status;
}
```

16. dataLocations.sol

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, search, and a syllabus. The main area displays a tutorial titled "9. Data Locations" (16 / 19). A yellow star icon indicates an assignment. The assignment tasks are:

- Change the value of the `myStruct` member `foo`, inside the function `f`, to 4.
- Create a new struct `myMemStruct2` with the data location `memory` inside the function `f` and assign it the value of `myStruct`. Change the value of the `myMemStruct2` member `foo` to 1.
- 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.
- Let the function `f` return `myStruct`, `myMemStruct2`, and `myMemStruct3`.

A tip at the bottom says: "Tip: Make sure to create the correct return types for the function `f`".

On the right, the code editor shows the `dataLocations.sol` file with the following content:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Shravani Patil-40
contract DataLocations {
    uint[] public arr;
    mapping(uint => address) map;
    struct MyStruct {
        uint foo;
    }
    mapping(uint => MyStruct) public myStructs;

    function f() public returns (MyStruct memory, MyStruct memory, MyStruct memory){
        // call _f with state variables
        _f(arr, map, myStructs[1]);
        // get a struct from a mapping
        MyStruct storage myStruct = myStructs[1];
        myStruct.foo = 4;
        // create a struct in memory
        MyStruct memory myMemStruct = MyStruct(0);
        MyStruct memory myMemStruct2 = myMemStruct;
        myMemStruct2.foo = 1;

        MyStruct memory myMemStruct3 = myStruct;
        myMemStruct3.foo = 3;
        return (myStruct, myMemStruct2, myMemStruct3);
    }
}
```

The status bar at the bottom shows: Scan Alert, Initialize as git repo, Did you know?, RemiAI Copilot (enabled), 26°C Sunny, ENG IN, 11:07, 10-02-2026.

17.etherandWei.sol

The screenshot shows the REMIX IDE interface. On the left, there's a sidebar with icons for file operations, search, and a syllabus. The main area displays a tutorial titled "10.1 Transactions - Ether and Wei" (17 / 19). A yellow star icon indicates an assignment. The assignment tasks are:

- 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 .

A tip at the bottom says: "Tip: Look at how this is written for `gwei` and `ether` in the contract."

On the right, the code editor shows the `etherAndWei.sol` file with the following content:

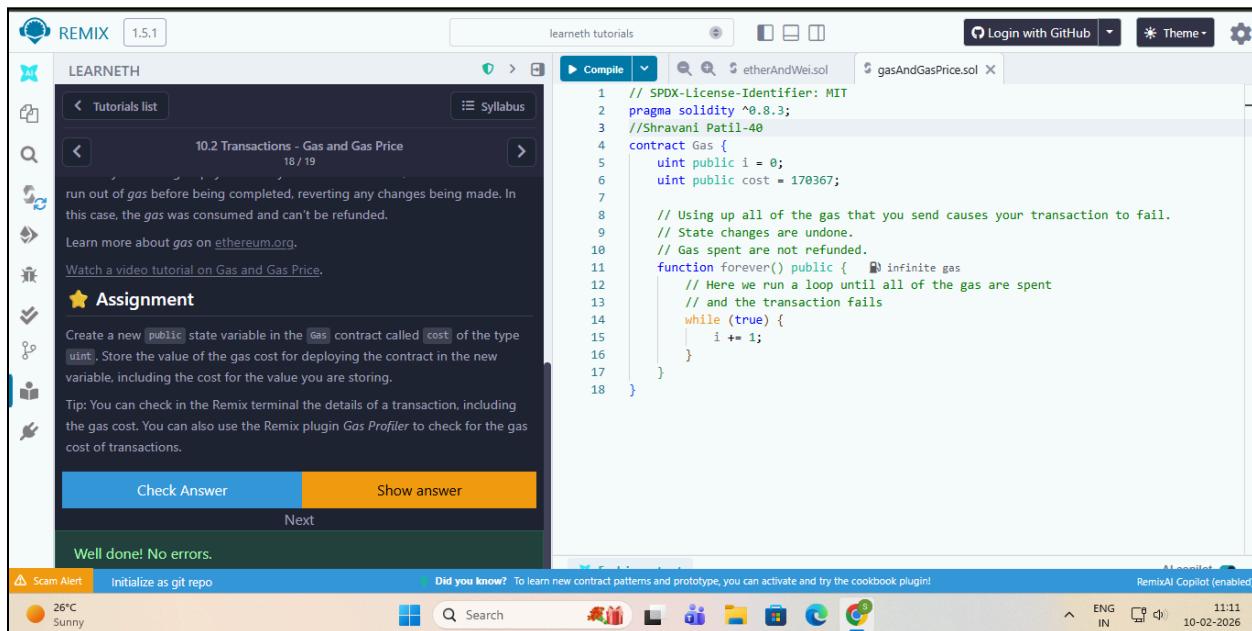
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Shravani Patil-40
contract EtherUnits {
    uint public oneWei = 1 wei;
    // 1 wei is equal to 1
    bool public isOneWei = 1 wei == 1;

    uint public oneEther = 1 ether;
    // 1 ether is equal to  $10^{18}$  wei
    bool public isOneEther = 1 ether == 1e18;

    uint public oneGwei = 1 gwei;
    // 1 ether is equal to  $10^9$  wei
    bool public isOneGwei = 1 gwei == 1e9;
}
```

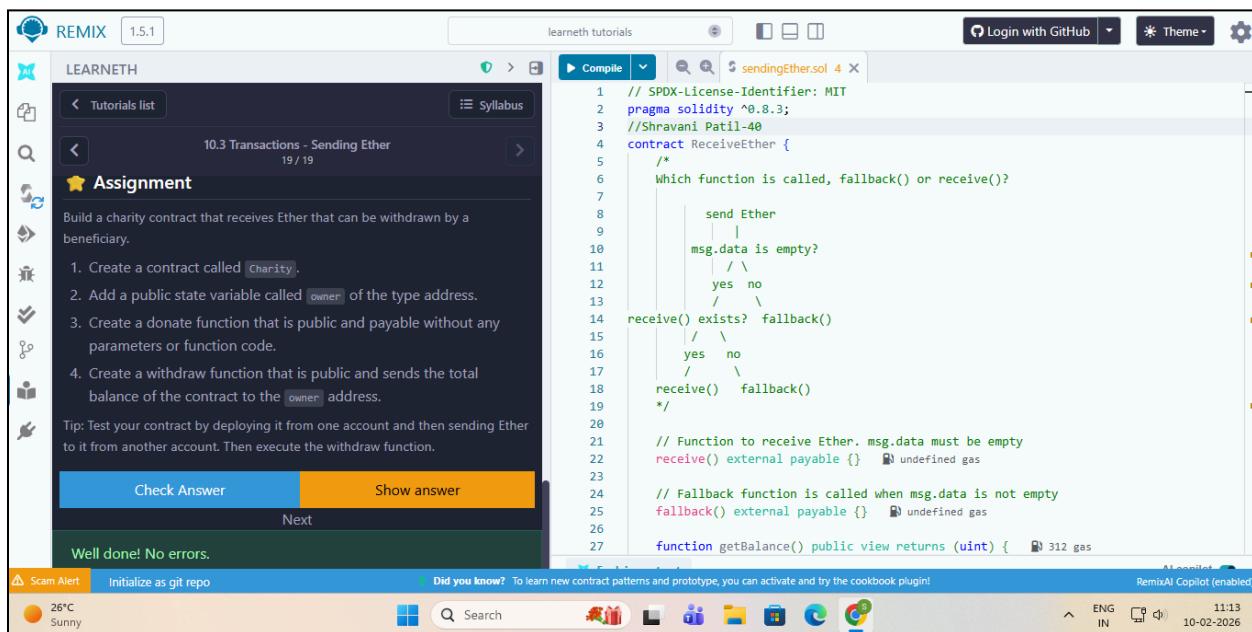
The status bar at the bottom shows: Scan Alert, Initialize as git repo, Did you know?, 10 February 2026, 26°C Sunny, ENG IN, Tue 11:08 (Local time), 10-02-2026.

18.gasandGasPrice.sol



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Shravani Patil-40
contract Gas {
    uint public i = 0;
    uint public cost = 170367;
    // Using up all of the gas that you send causes your transaction to fail.
    // State changes are undone.
    // Gas spent are not refunded.
    function forever() public {
        infinite gas
        // Here we run a loop until all of the gas are spent
        // and the transaction fails
        while (true) {
            i += 1;
        }
    }
}
```

19. SendingEther.sol



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Shravani Patil-40
contract ReceiveEther {
    /*
    Which function is called, fallback() or receive()?
    */
    send Ether
    |
    msg.data is empty?
    / \
    yes no
    / \
    receive() exists? fallback()
    / \
    yes no
    / \
    receive() fallback()
    */

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}  undefined gas

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}  undefined gas

    function getBalance() public view returns (uint) {  312 gas
}
```

Conclusion:

This experiment provides hands-on experience in Solidity programming by exploring core blockchain concepts, data types, control structures, and transaction handling. Understanding these fundamentals enables the design and deployment of secure, efficient, and reliable smart contracts on the Ethereum blockchain.