

Aim: Hands on Solidity Programming Assignments for creating Smart Contracts

Theory:

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

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

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

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

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

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

2. Inputs and Outputs to Functions

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

3. Visibility, Modifiers and Constructors

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

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

3. Control Flow: if-else, loops

Control flow in Solidity is similar to traditional programming languages:

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

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

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

6. Data Locations

Solidity uses three primary data locations for storing variables:

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

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

- **Ether and Wei:** Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether = 10^{18} Wei). This ensures high precision in financial transactions.
- **Gas and Gas Price:** Every transaction consumes gas, which represents computational effort. The gas price determines how much Ether is paid per unit of gas. A higher gas price incentivizes miners to prioritize the transaction.
- **Sending Transactions:** Transactions are used for transferring Ether or interacting with contracts. Functions like transfer() and send() are commonly used, while call() provides more flexibility. Each transaction requires gas, making efficiency in contract design very important.

Output:

- Tutorial no. 1 – Compile the code

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract Counter {
5     uint public count;
6
7     // Function to get the current count
8     function get() public view returns (uint) { 2453 gas
9         return count;
10    }
11    // Ronak 29
12    // Function to increment count by 1
13    function inc() public { infinite gas
14        count += 1;
15    }
16
17    // Function to decrement count by 1
18    function dec() public { infinite gas
19        count -= 1;
20    }
21 }
```

- Tutorial no. 1 – Deploy the contract

The screenshot shows a web-based Ethereum development environment. On the left, there's a sidebar titled "DEPLOY & RUN TRANSACTIONS" with tabs for "At Address" and "Load contract from Address". Below these are sections for "Transactions recorded" and "Deployed Contracts". Under "Deployed Contracts", there's a list for "COUNTER AT 0X7EF...8CB47" which includes a "Balance: 0 ETH" and four buttons: "dec", "inc", "count", and "get". A callout box points to the "get" button. On the right, the "Compile" tab is selected, showing the Solidity code for the "Counter" contract:

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

contract Counter {
    uint public count;

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

    // Ronak 29
    // Function to increment count by 1
    function inc() public {
        infinite gas
        count += 1;
    }

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

- Tutorial no. 1 – get

This screenshot shows the same interface as the previous one, but the "get" button for the "COUNTER" contract is now highlighted with a callout box pointing to it. The callout box contains the text "get - call". Below the callout, the result of the call is shown as "0: uint256: 0".

- Tutorial no. 1 – Increment

Deployed Contracts 1

▼ COUNTER AT 0X7EF..8CB47 (M) ⌂ ⚡ ✎

Balance: 0 ETH

dec

inc inc - transact (not payed)

count

get

0: uint256: 2

Low level interactions i

CALldata

Transact

0: uint256: 2

- Tutorial no. 1 – Decremen

Deployed Contracts 1

▼ COUNTER AT 0X7EF..8CB47 (M) ⌂ ⚡ ✎

Balance: 0 ETH

dec

inc inc - transact (not payed)

count

get

0: uint256: 1

Low level interactions i

CALldata

0: uint256: 1

- Tutorial no. 2

LEARNETH

Compile Home intro

< Tutorials list Syllabus

2. Basic Syntax 2 / 19 >

following sections.

To help you understand the code, we will link in all following sections to video tutorials from the creator of the Solidity by Example contracts.

Watch a video tutorial on Basic Syntax.

★ Assignment

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.

Check Answer Show answer

Next

Well done! No errors.

// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3
pragma solidity ^0.8.3;
// Ronak 29
contract MyContract {
 string public name = "Alice";
}

Explain contract

0 Listed

- Tutorial no. 3

< Tutorials list

Syllabus

3. Primitive Data Types
3 / 19

and **Structs**.

Watch a video tutorial on [Primitive Data Types](#).

Assignment

- 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`.

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.

...
uint256 ranges from 0 to $2^{256} - 1$

```
*/  
uint8 public u8 = 1;  
uint public u256 = 456;  
uint public u = 123; // uint is an alias for uint256  
uint public newU = 0;  
// Ronak 29  
/*  
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  
int public neg=-29;  
  
address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;  
address public newAddr = 0x000000000000000000000000000000000000000000000000000000000000000;
```

[Explain contract](#)

0 Listen on all transactions Filter with transaction hash or address

data: 0x6d4...ce63c

- Tutorial no. 4

LEARNETH

[Tutorials list](#) [Syllabus](#)

4. Variables

4 / 19

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

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

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

Assignment

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

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

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

Next

Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
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
10    function doSomething() public {
11        // Local variables are not saved to the blockchain.
12
13        uint i = 456;
14        // Ronak 29
15        // Here are some global variables
16        uint timestamp = block.timestamp; // Current block timestamp
17        address sender = msg.sender; // address of the caller
18        blockNumber = block.number;
19    }
20 }
```

Explain contract

0 Listen on all transactions [Filter with transaction hash or a](#)

data: 0x6d4...ce63c

- Tutorial no. 5

LEARNETH

[Tutorials list](#) [Syllabus](#)

5.1 Functions - Reading and Writing to a State Variable

5 / 19

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

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.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract SimpleStorage {
5     // State variable to store a number
6     uint public num;
7     bool public b=true;
8
9     function get_b() public view returns (bool) {
10         return b;
11     }
12     // Ronak 29
13     // You need to send a transaction to write to a state variable.
14     function set(uint _num) public {
15         num = _num;
16     }
17
18     // You can read from a state variable without sending a transaction
19     function get() public view returns (uint) {
20         return num;
21     }
}
```

Explain contract

0 Listen on all transactions [Filter with transaction hash or a](#)

data: 0x6d4...ce63c

- Tutorial no. 6

LEARNETH

< Tutorials list Syllabus

5.2 Functions - View and Pure

6 / 19

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**

Next

Well done! No errors.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3
4 contract ViewAndPure {
5     uint public x = 1;
6
7     // Promise not to modify the state.
8     function addToX(uint y) public view returns (uint) {
9         return x + y;
10    }
11    // Ronak 29
12    function addToX2(uint y) public {
13        x=x+y;
14    }
15
16    // Promise not to modify or read from the state.
17    function add(uint i, uint j) public pure returns (uint) {
18        return i + j;
19    }
20 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

data: 0x6d4...ce63c

- Tutorial no. 7

LEARNETH

< Tutorials list Syllabus

5.3 Functions - Modifiers and Constructors

7 / 19

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**

Next

Well done! No errors.

```
43     locked = true;
44     ;
45     locked = false;
46 }
47
48 function decrement(uint i) public noReentrancy { infinite gas
49     x -= i;
50
51     if (i > 1) {
52         decrement(i - 1);
53     }
54 }
55 // Ronak 29
56 function increaseX(uint i) public noReentrancy { infinite gas
57     x += i;
58
59     if (i < 1) {
60         increaseX(i + 1);
61     }
62 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

data: 0x6d4...ce63c

- Tutorial no. 8

LEARNETH

< Tutorials list Syllabus

5.4 Functions - Inputs and Outputs

8 / 19

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.

Code Editor:

```

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
    
```

Compiler Output:

Compile Constructors.sol

InputsAndOutputs.sol

Visibility.sol

visibility_answer.sol

0 Listen on all transactions

data: 0x6d4...ce63c

Explain contract

- Tutorial no. 9

LEARNETH

< Tutorials list Syllabus

6. Visibility

9 / 19

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.

Code Editor:

```

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     function testInternalFunc() public pure override returns (string memory) {
64         return internalFunc();
65     }
66     // ronak 29
67     function testInternalVar() public view returns (string memory, string memory) {
68         return (internalVar, publicVar);
69     }
70
    
```

Compiler Output:

Compile Constructors.sol

InputsAndOutputs.sol

Visibility.sol

visibility_answer.sol

0 Listen on all transactions

data: 0x6d4...ce63c

Explain contract

AI copilot

Tutorial no. 10

Tutorials list Syllabus

7.1 Control Flow - If/Else

10 / 19

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

Watch a video tutorial on the If/Else statement.

Assignment

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

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

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

Check Answer **Show answer**

Next

Well done! No errors.

```

10 } else {
11     return 2;
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 // Ronak 29
25 function evenCheck(uint y) public pure returns (bool) { // infinite gas
26     return y%2 == 0 ? true : false;
27 }

```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

data: 0x6d4...ce63c

Tutorial no. 11

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

- 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**

Next

Well done! No errors.

```

4 contract Loop {
5     uint public count;
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         // Ronak 29
20         // while loop
21         uint j;
22         while (j < 10) {
23             j++;
24         }

```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

data: 0x6d4...ce63c

Tutorial no. 12

Tutorials list Syllabus

8.1 Data Structures - Arrays

12 / 19

important, then we can move the last element of 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

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

Check Answer **Show answer**

Next

Well done! No errors.

```

14 }
15 }
16 // Solidity can return the entire array.
17 // But this function should be avoided for
18 // arrays that can grow indefinitely in length.
19 function getArr() public view returns (uint[3] memory) { // infinite gas
20     return arr3;
21 }
22 // Ronak 29
23
24 function push(uint i) public { // 46820 gas
25     // Append to array
26     // This will increase the array length by 1.
27     arr.push(i);
28 }
29
30 function pop() public { // 29462 gas
31     // Remove last element from array
32 }

```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

data: 0x6d4...ce63c

Tutorial no. 13

8.2 Data Structures - Mappings
13 / 19

We can use the delete operator to delete a value associated 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.

```
contract Mapping {
    // Mapping from address to uint
    mapping(address => uint) public balances;
    // Ronak 29
    function get(address _addr) public view returns (uint) {
        // Mapping always returns a value.
        // If the value was never set, it will return the default value.
        return balances[_addr];
    }

    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];
    }
}
```

Explain contract **AI copilot**

0 Listen on all transactions Filter with transaction hash or ad...

data: 0x6d4...ce63c

Tutorial no. 14

8.3 Data Structures - Structs
14 / 19

its member by assigning it a new value (line 23).

Accessing structs

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

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.

```
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52

}
// update text
function update(uint _index, string memory _text) public {
    Todo storage todo = todos[_index];
    todo.text = _text;
}

// update completed
function toggleCompleted(uint _index) public {
    Todo storage todo = todos[_index];
    todo.completed = !todo.completed;
}
// Ronak 29

function remove(uint _index) public {
    delete todos[_index];
}
```

Explain contract **AI copilot**

0 Listen on all transactions Filter with transaction hash or ad...

data: 0x6d4...ce63c

Tutorial no. 15

8.4 Data Structures - Enums
15 / 19

Another way to update the value is using the dot operator by providing the name of the enum 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.

```
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

Status public status,
Size public sizes;

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

function getSize() public view returns (Size) {
    return sizes;
}
// Ronak 29

// Update status by passing uint into input
function set(Status _status) public {
    status = _status;
}

// You can update to a specific enum like this
function cancel() public {
    status = Status.S;
}
```

Explain contract **AI copilot**

0 Listen on all transactions Filter with transaction hash or ad...

data: 0x6d4...ce63c

Tutorial no. 16

9. Data Locations
16 / 19

Assignment

- 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`.

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

Check Answer **Show answer**

Well done! No errors.

```

24 myMemStruct3.foo = 3;
25     return (myStruct, myMemStruct2, myMemStruct3);
26 }
27
28 function _f() {
29     uint[] storage _arr,
30     mapping(uint => address) storage _map,
31     MyStruct storage _myStruct
32     internal {
33         // do something with storage variables
34     }
35     // ronak 29
36
37     // You can return memory variables
38     function g(uint[] memory arr) public returns (uint[] memory) {
39         // do something with memory array
40         arr[0] = 1;
41     }
42 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash or ad... **AI copilot**

data: 0x6d4...ce63c

Tutorial no. 17

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

- Create a `public uint` called `oneGwei` and set it to 1 `gwei`.
- 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**

Well done! No errors.

```

3 contract EtherUnits {
4     uint public oneWei = 1 wei;
5     // 1 wei is equal to 1
6     bool public isOneWei = 1 wei == 1;
7
8     uint public oneEther = 1 ether;
9     // 1 ether is equal to  $10^{18}$  wei
10    bool public isOneEther = 1 ether == 1e18;
11    // Ronak 29
12    uint public oneGwei = 1 gwei;
13    // 1 ether is equal to  $10^9$  wei
14    bool public isOneGwei = 1 gwei == 1e9;
15 }
16 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash or ad... **AI copilot**

data: 0x6d4...ce63c

Tutorial no. 18

10.2 Transactions - Gas and Gas Price
18 / 19

run out of `gas` before being completed, reverting any changes being made. In this case, the `gas` was consumed and can't be refunded.
Learn more about `gas` on ethereum.org.
Watch a video tutorial on Gas and Gas Price.

Assignment

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

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

Check Answer **Show answer**

Well done! No errors.

```

3 contract Gas {
4     uint public i = 0;
5     uint public cost = 170367;
6     // Ronak 29
7     // Using up all of the gas that you send causes your transaction to fail.
8     // State changes are undone.
9     // Gas spent are not refunded.
10    function forever() public {
11        // Here we run a loop until all of the gas are spent
12        // and the transaction fails
13        while (true) {
14            i += 1;
15        }
16    }
17 }
18 }
```

Explain contract

0 Listen on all transactions Filter with transaction hash or ad... **AI copilot**

data: 0x6d4...ce63c

Tutorial no. 19

The screenshot shows the Remix IDE interface for a Solidity contract named "Charity". The code is as follows:

```
52
53 contract Charity {
54     address public owner;
55
56     constructor() { // 165452 gas 141000 gas
57         owner = msg.sender;
58     }
59
60     function donate() public payable {} // 141 gas
61     // ronak 29
62     function withdraw() public {
63         infinite gas
64         uint amount = address(this).balance;
65
66         (bool sent, bytes memory data) = owner.call{value: amount}("");
67         require(sent, "Failed to send Ether");
68     }
}
```

The interface includes tabs for "Solidity Compiler", "Assignment", and "Explain contract". The "Assignment" tab contains instructions to build a charity contract that receives Ether and can be withdrawn by a beneficiary. It lists four steps: creating a contract, adding a public state variable, creating a donate function, and creating a withdraw function. A tip suggests testing the contract by deploying it and sending Ether from another account. Below the code editor are buttons for "Check Answer", "Show answer", and "Next". A green message bar at the bottom says "Well done! No errors."

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.