Anshi Tiwari D20C Roll no: 60

Experiment 4

Aim: Hands on Solidity Programming Assignments for creating Smart Contracts

Theory:

Solidity is a contract-oriented programming language used to write smart contracts on the Ethereum blockchain. Smart contracts are self-executing programs that run on the Ethereum Virtual Machine (EVM) and automatically enforce rules defined within them.

1. Primitive Data Types, Variables and Functions:

Solidity provides several primitive data types that are essential for contract development:

Data Type	Description	Example
uint / int	Unsigned and signed integers (e.g., uint256, int128)	uint256 age = 20;
bool	Logical values (true/false)	bool isActive = true;
address	Stores Ethereum account or contract address	address owner;
string / bytes	Text and raw byte data	string name = "Alice";

Types of Variables:

- **State Variables** – Stored permanently on the blockchain.
- **Local Variables** – Declared inside functions; temporary.
- **Global Variables** – Built-in variables such as msg.sender, msg.value, block.timestamp.

Function Type	Description
pure	Cannot read or modify state variables. Used for computation only.
view	Can read state variables but cannot modify them.
Regular Function	Can read and modify state variables.

2. Function Inputs and Outputs:

Solidity functions can accept parameters and return single or multiple values.

Key Features:

- Input parameters allow users to pass data.
- Return values provide results after execution.
- Named return variables enhance clarity.
- Multiple returns are supported.

Example concept: function add(uint a, uint b) public pure returns (uint) {return a + b;}

3. Visibility, Modifiers and Constructors

A. Function Visibility

Visibility	Accessibility
public	Accessible internally and externally
private	Accessible only within the contract
internal	Accessible within contract and derived contracts
external	Callable only from outside the contract

B. Modifiers: Modifiers are reusable code blocks that enforce conditions before function execution.

Example usage:

- Restricting access to contract owners.
- Validating inputs.
- Checking balances.

C. Constructors

- Executed only once during deployment.
- Used to initialize state variables.
- Commonly assigns the deployer as contract owner.

4. Control Flow Statements

Solidity supports decision-making and looping mechanisms similar to other programming languages. Conditional Statements: if, if-else. Loops: for, while, do-while

5. Data Structures:

Solidity offers powerful data structures to organize and manage data.

Data Structure	Description	Example Usage
Arrays	Ordered collection of elements	List of user addresses
Mappings	Key-value storage	mapping(address => uint)
Structs	Custom grouped data type	struct Student {string name; uint marks;}
Enums	Predefined constant values	enum Status {Pending, Active, Closed}

6. Data Locations:

Understanding data locations is crucial for gas optimization.

Location	Storage Duration	Modifiable	Gas Cost

storage	Permanent (Blockchain)	Yes	High
memory	Temporary (Function execution)	Yes	Medium
calldata	Temporary (External inputs)	No	Low

7. Transactions and Gas Concepts

Ether and Wei

- Ether is the cryptocurrency of Ethereum.
- 1 Ether = 10^{18} Wei.
- Wei ensures precision in financial calculations.

Gas and Gas Price

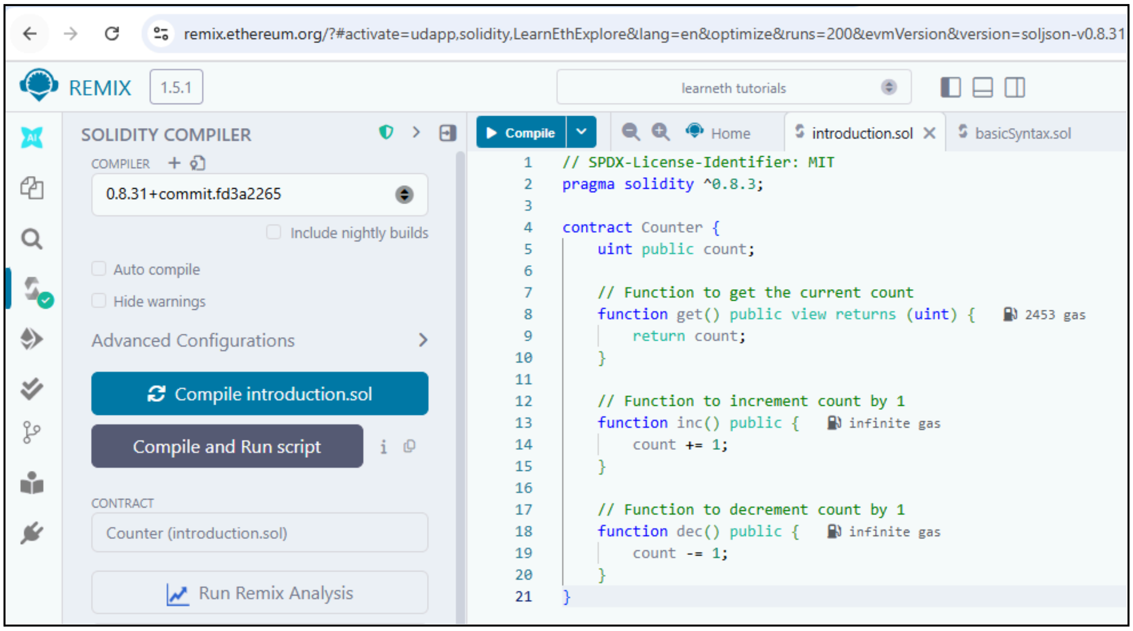
- Gas represents computational effort.
- Gas price determines transaction priority.
- Higher gas price → Faster processing.

Sending Ether
Common methods:

- transfer()
- send()
- call() (more flexible and recommended in modern contracts)

Implementation

Tutorial no. 1 – Compile the code



remix.ethereum.org/?#activate=udapp,solidity,LearnEthExplore&lang=en&optimize&runs=200&evmVersion&version=soljson-v0.8.31

SOLIDITY COMPILER

COMPILER 0.8.31+commit.fd3a2265

□ Include nightly builds

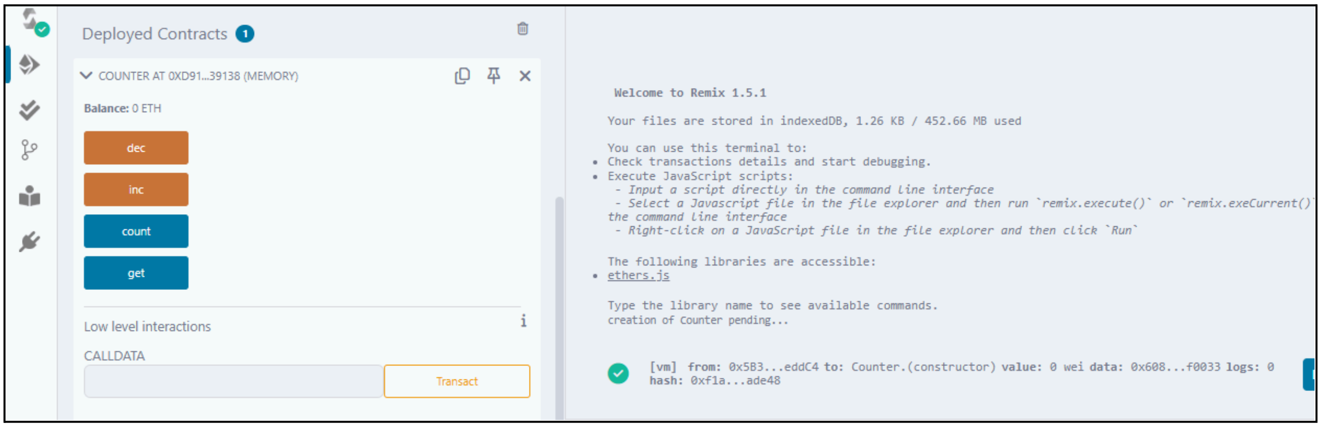
□ Auto compile

□ Hide warnings

CONTRACT Counter (introduction.sol)

```
// 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) {    2453 gas
        return count;
    }

    // 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 – Deploy the contract

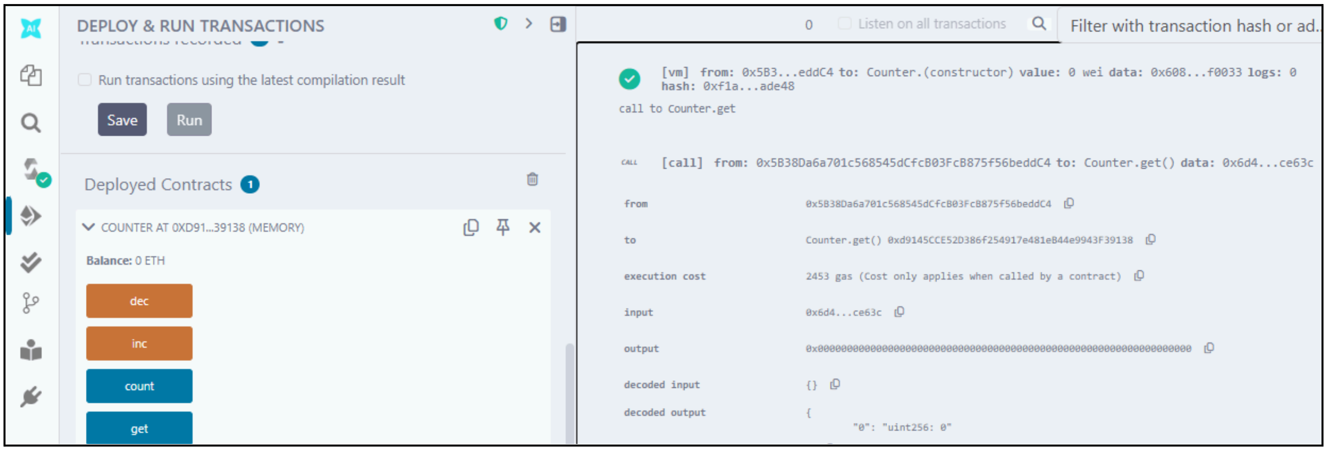


COUNTER AT 0XD91...39138 (MEMORY)

Balance: 0 ETH

```
 Welcome to Remix 1.5.1

 Your files are stored in indexedDB, 1.26 KB / 452.66 MB used

 You can use this terminal to:
• Check transactions details and start debugging.
• Execute JavaScript scripts:
  - Input a script directly in the command line interface
  - Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.exeCurrent()`
 the command line interface
  - Right-click on a JavaScript file in the file explorer and then click `Run`

 The following libraries are accessible:
• ethers.js

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

 [vm] from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei data: 0x608...f0033 logs: 0
 hash: 0xf1a...ade48
```

Tutorial no. 1 – get


DEPLOY & RUN TRANSACTIONS
Save
Run
Deployed Contracts 1
dec
inc
count
get
0
Listen on all transactions
Filter with transaction hash or ad..

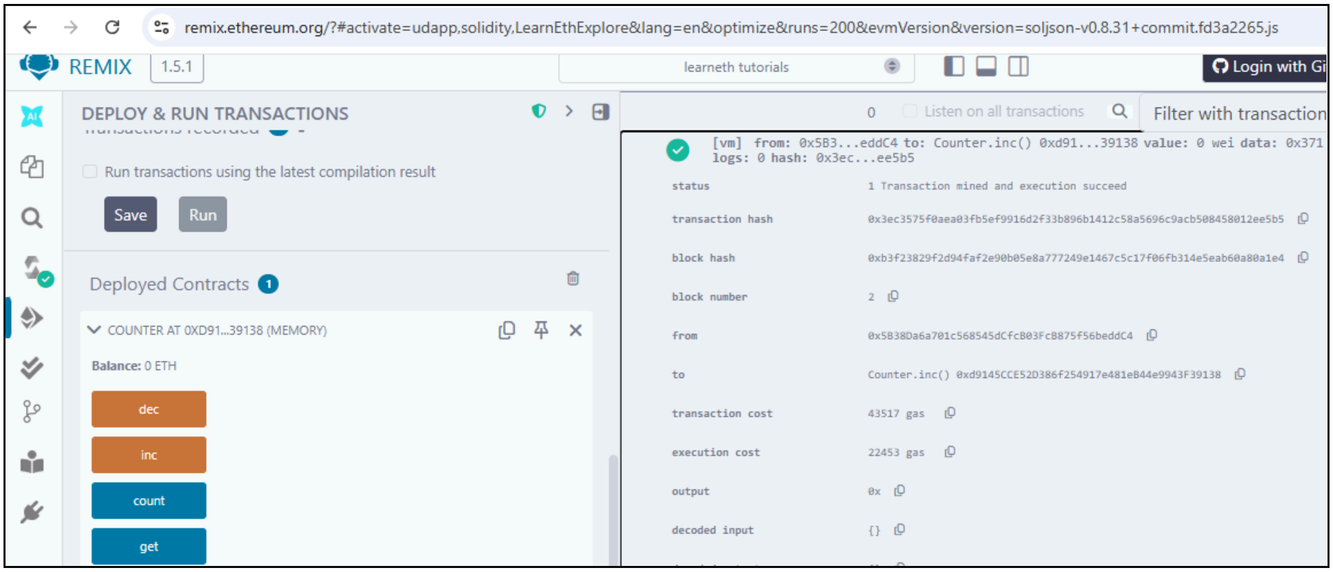
□ Run transactions using the latest compilation result

COUNTER AT 0XD91...39138 (MEMORY)

Balance: 0 ETH

```
 [vm] from: 0x5B3...eddC4 to: Counter.(constructor) value: 0 wei data: 0x608...f0033 logs: 0
 hash: 0xf1a...ade48
call to Counter.get

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

 from              0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
 to                Counter.get() 0xd9145CCE52D386f254917e481eB44e9943F39138
 execution cost    2453 gas (Cost only applies when called by a contract)
 input             0x6d4...ce63c
 output            0x0000000000000000000000000000000000000000000000000000000000000000
 decoded input     {}
 decoded output    {
                        "0": "uint256: 0"
```

Tutorial no. 1 – Increment



remix.ethereum.org/?#activate=udapp,solidity,LearnEthExplore&lang=en&optimize&runs=200&evmVersion&version=soljson-v0.8.31+commit.fd3a2265.js

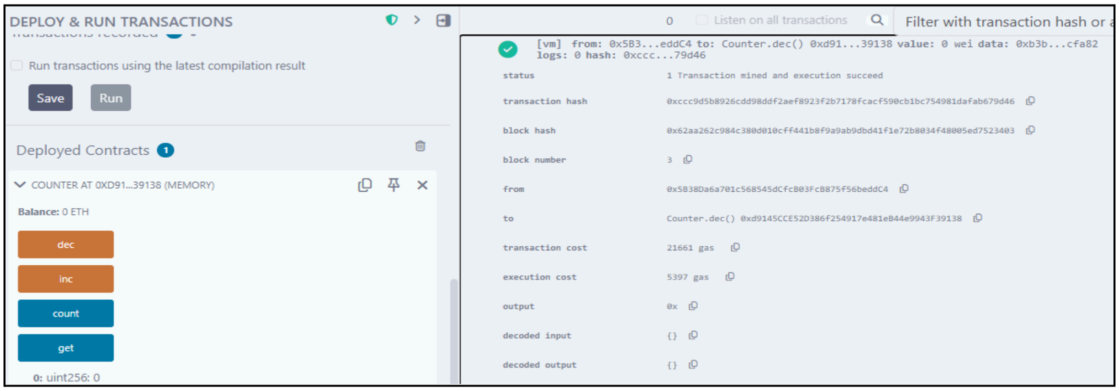
□ Run transactions using the latest compilation result

COUNTER AT 0XD91...39138 (MEMORY)

Balance: 0 ETH

```
 [vm] from: 0x5B3...eddC4 to: Counter.inc() 0xd91...39138 value: 0 wei data: 0x371
 logs: 0 hash: 0x3ec...ee5b5
 status             1 Transaction mined and execution succeed
 transaction hash   0x3ec3575f0aea03fb5ef9916d2f33b896b1412c58a5696c9acb508458012ee5b5
 block hash         0xb3f23829f2d94faf2e90b05e8a777249e1467c5c17f06fb314e5eab60a80a1e4
 block number       2
 from               0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
 to                 Counter.inc() 0xd9145CCE52D386f254917e481eB44e9943F39138
 transaction cost   43517 gas
 execution cost     22453 gas
 output             0x
 decoded input      {}
```

Tutorial no. 1 – Decrement



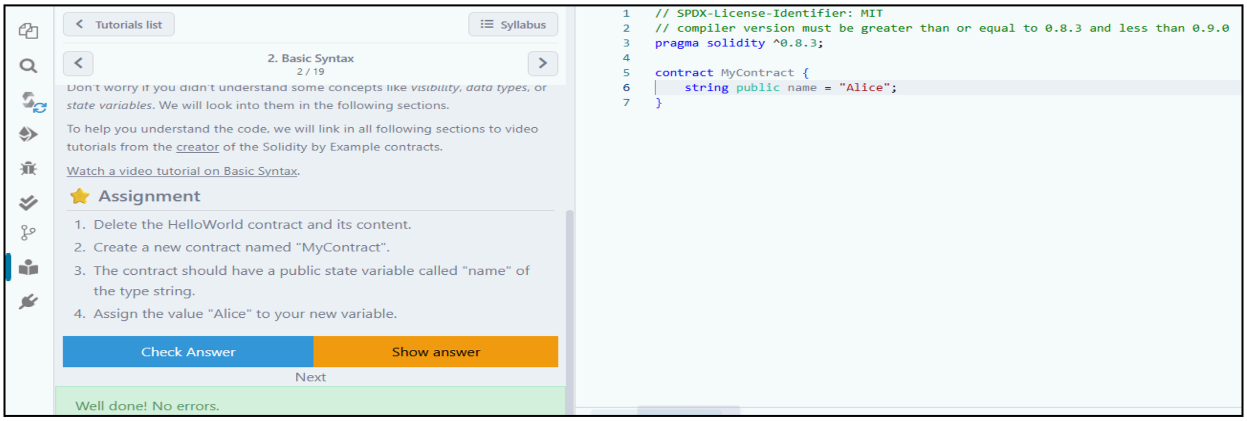
DEPLOY & RUN TRANSACTIONS
□ Run transactions using the latest compilation result
Save
Run
Deployed Contracts 1
COUNTER AT 0XD91...39138 (MEMORY)
dec
inc
count
get
0
□ Listen on all transactions
Filter with transaction hash or a

Balance: 0 ETH

0: uint256: 0

```
[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei data: 0xb3b...cfa82
logs: 0 hash: 0xccc...79d46
status              1 Transaction mined and execution succeed
transaction hash    0xccc9d5b8926cdd98ddf2aef8923f2b7178fcacf590cb1bc754981dafab679d46
block hash          0x62aa262c984c380d010cff441b8f9a9ab9dbd41f1e72b8034f48005ed7523403
block number        3
from                0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                  Counter.dec() 0xd9145CCE52D386f254917e481eB44e9943F39138
transaction cost    21661 gas
execution cost      5397 gas
output              0x
decoded input       {}
decoded output      {}
```

Tutorial 2: Basic Syntax



Don't worry if you didn't understand some concepts like *visibility*, *data types*, or *state variables*. We will look into them in the 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.

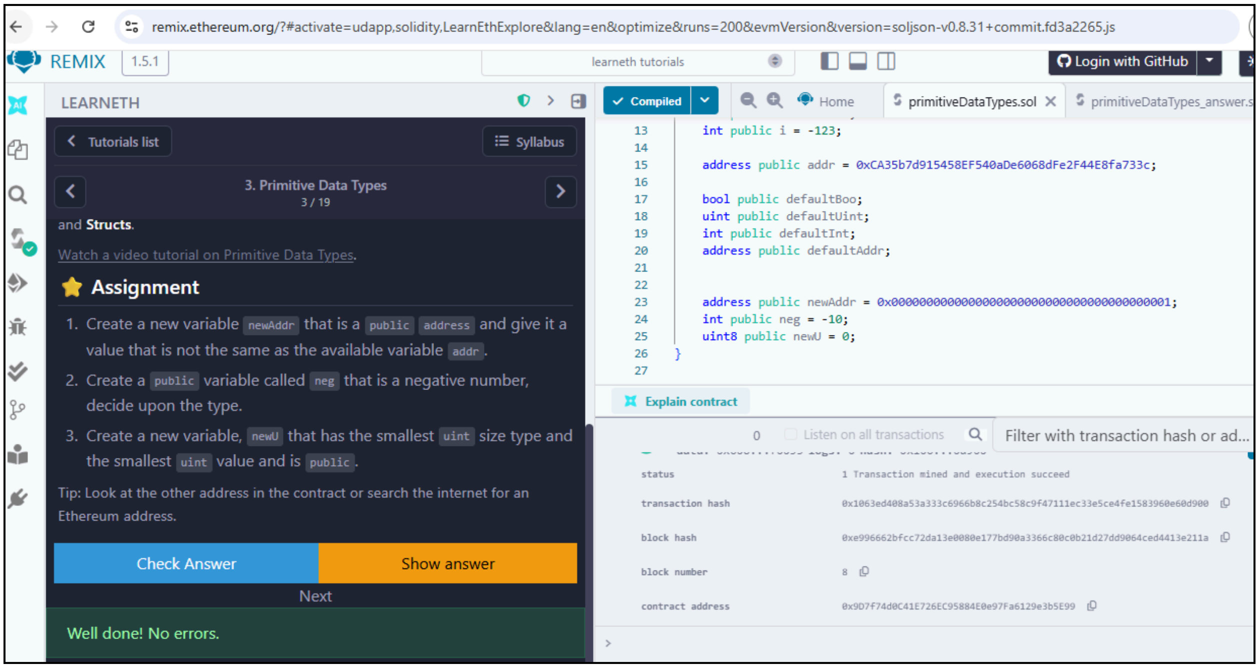
Assignment

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.

Well done! No errors.

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

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

Tutorial no. 3: Primitive Datatype



remix.ethereum.org/?#activate=udapp,solidity,LearnEthExplore&lang=en&optimize&runs=200&evmVersion&version=soljson-v0.8.31+commit.fd3a2265.js
REMIX
1.5.1
learneth tutorials
Login with GitHub
LEARNETH
Compiled
Home
primitiveDataTypes.sol
primitiveDataTypes_answer.s
Tutorials list
Syllabus
3. Primitive Data Types
3 / 19
Check Answer
Show answer
Next
Explain contract
0
□ Listen on all transactions
Filter with transaction hash or ad...

and **Structs**.

Watch a video tutorial on Primitive Data Types.

Assignment

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

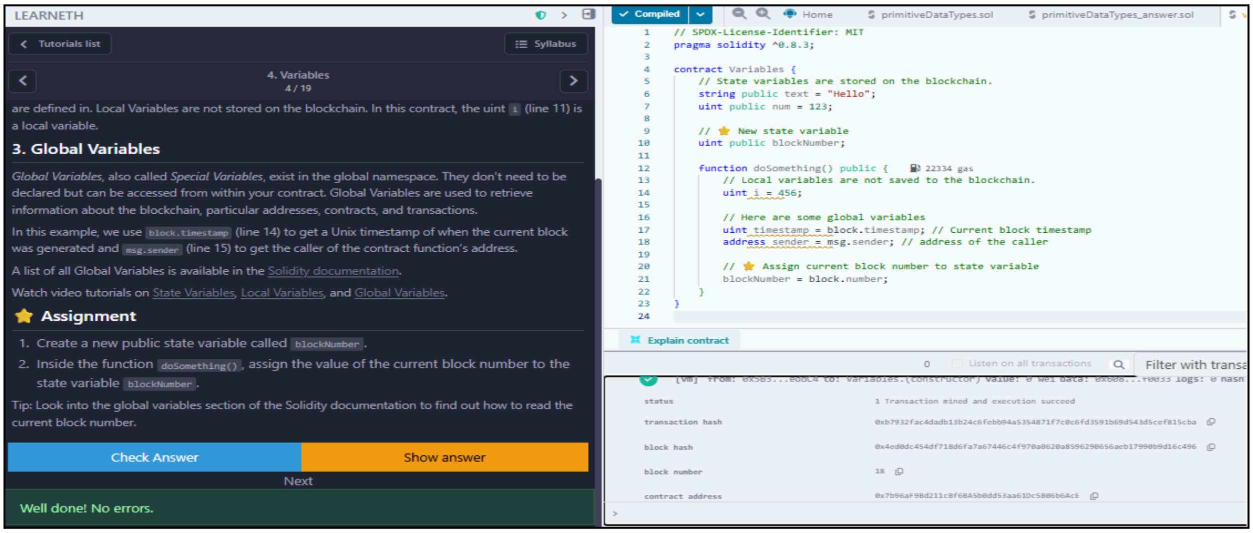
Tip: Look at the other address in the contract or search the internet for an Ethereum address.

Well done! No errors.

```
    int public i = -123;

    address public addr = 0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c;

    bool public defaultBoo;
    uint public defaultUint;
    int public defaultInt;
    address public defaultAddr;


    address public newAddr = 0x0000000000000000000000000000000000000001;
    int public neg = -10;
    uint8 public newU = 0;
}

```

```
status              1 Transaction mined and execution succeed
transaction hash    0x1063ed408a53a333c6966b8c254bc58c9f47111ec33e5ce4fe1583960e60d900
block hash          0xe996662bfcc72da13e0080e177bd90a3366c80c0b21d27dd9064ced4413e211a
block number        8
contract address    0x9D7f74d0C41E726EC95884E0e97Fa6129e3b5E99
```

Tutorial No: 4: Variables



4. Variables
4 / 19

are defined in. Local Variables are not stored on the blockchain. In this contract, the uint `i` (line 11) is a local variable.

3. Global Variables

Global Variables, also called *Special Variables*, exist in the global namespace. They don't need to be declared but can be accessed from within your contract. Global Variables are used to retrieve information about the blockchain, particular addresses, contracts, and transactions.

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

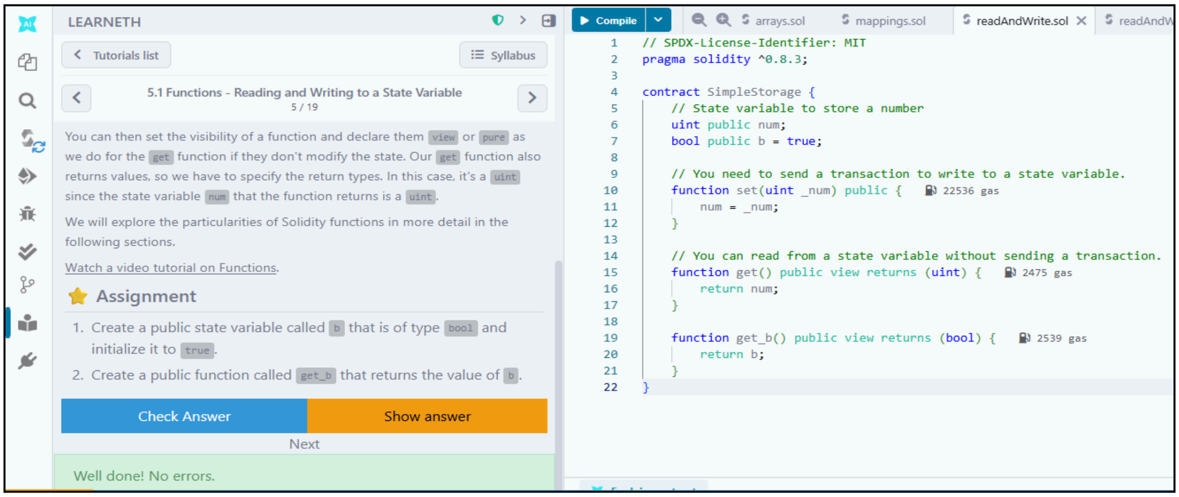
Next

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
9       // ⭐ New state variable
10      uint public blockNumber;
11
12      function doSomething() public {      22334 gas
13          // Local variables are not saved to the blockchain.
14          uint i = 456;
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          // ⭐ Assign current block number to state variable
21          blockNumber = block.number;
22      }
23  }
24
```

```
[vm] from: 0x5B3...eddC4 to: Variables.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash
status              1 Transaction mined and execution succeed
transaction hash    0xb7932fac4dadb13b24c6febb94a5354871f7c8c6fd3591b69d543d5cef815cba
block hash          0x4ed0dc454df718d6fa7a67446c4f970a0620a8596290656acb17990b9d16c496
block number        18
contract address    0x7b96aF98d211c8f68A5b0dd53aa61Dc5806b6AcE
```

Tutorial no. 5: Functions - Reading and Writing to a State variable



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`.

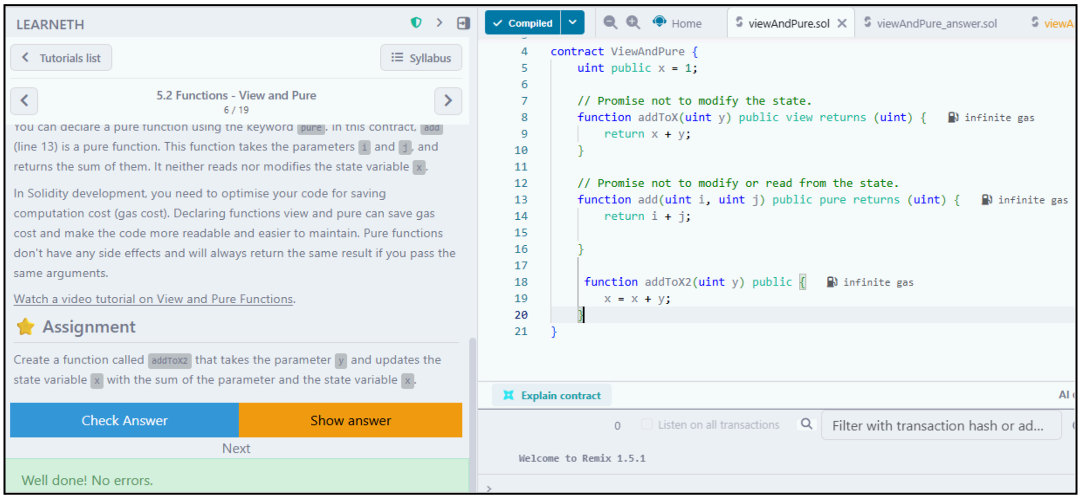
Check Answer | Show answer

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       // You need to send a transaction to write to a state variable.
10      function set(uint _num) public {      22536 gas
11          num = _num;
12      }
13
14      // You can read from a state variable without sending a transaction.
15      function get() public view returns (uint) {      2475 gas
16          return num;
17      }
18
19      function get_b() public view returns (bool) {      2539 gas
20          return b;
21      }
22  }
```

Tutorial no. 6 : Functions- View and Pure



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

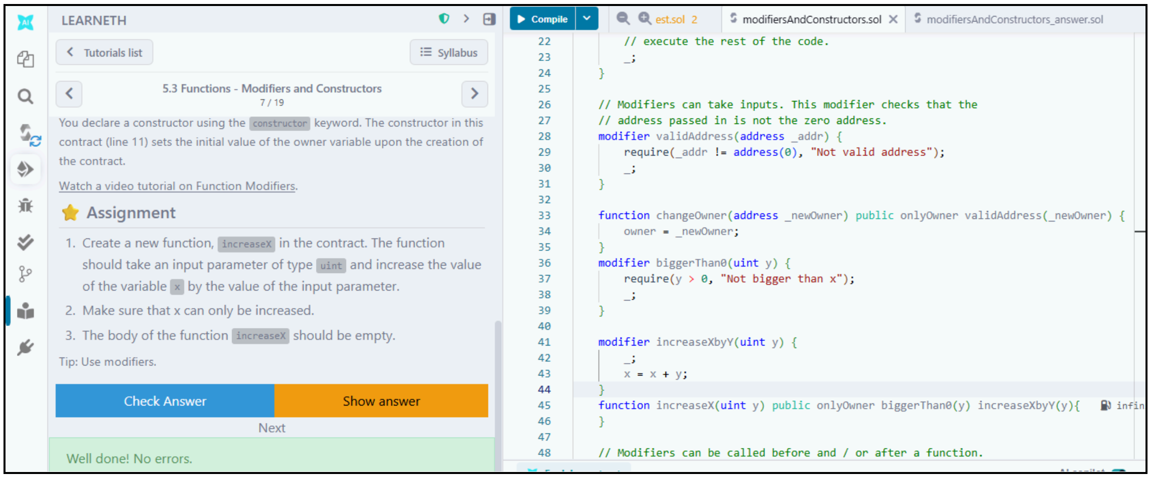
Next

Well done! No errors.

```
4   contract ViewAndPure {
5       uint public x = 1;
6
7       // Promise not to modify the state.
8       function addToX(uint y) public view returns (uint) {      infinite gas
9           return x + y;
10      }
11
12      // Promise not to modify or read from the state.
13      function add(uint i, uint j) public pure returns (uint) {      infinite gas
14          return i + j;
15
16      }
17
18       function addToX2(uint y) public {      infinite gas
19          x = x + y;
20      }
21  }
```

```
0
Welcome to Remix 1.5.1
```

Tutorial no.7 : Functions- Modifiers and Constructors


LEARNETH
Tutorials list
Syllabus
Compile
est.sol 2
modifiersAndConstructors.sol
modifiersAndConstructors_answer.sol

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.

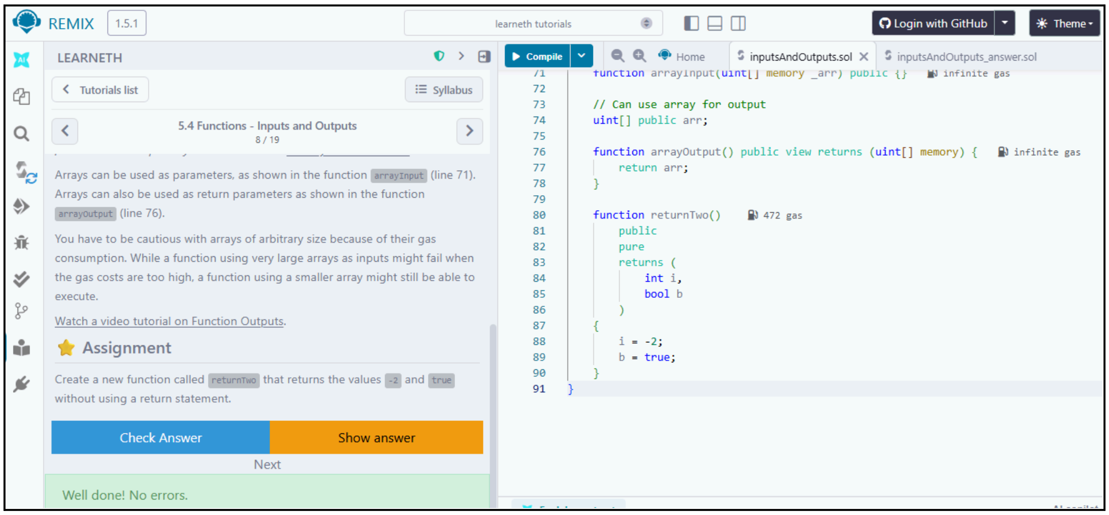
Check Answer | Show answer

Next

Well done! No errors.

```
22          // execute the rest of the code.
23          _;
24      }
25
26      // Modifiers can take inputs. This modifier checks that the
27      // address passed in is not the zero address.
28      modifier validAddress(address _addr) {
29          require(_addr != address(0), "Not valid address");
30          _;
31      }
32
33      function changeOwner(address _newOwner) public onlyOwner validAddress(_newOwner) {
34          owner = _newOwner;
35      }
36      modifier biggerThan0(uint y) {
37          require(y > 0, "Not bigger than x");
38          _;
39      }
40
41      modifier increaseXbyY(uint y) {
42          _;
43          x = x + y;
44      }
45      function increaseX(uint y) public onlyOwner biggerThan0(y) increaseXbyY(y){    infin
46      }
47
48      // Modifiers can be called before and / or after a function.
```

Tutorial no. 8 : Functions- Inputs and outputs



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.

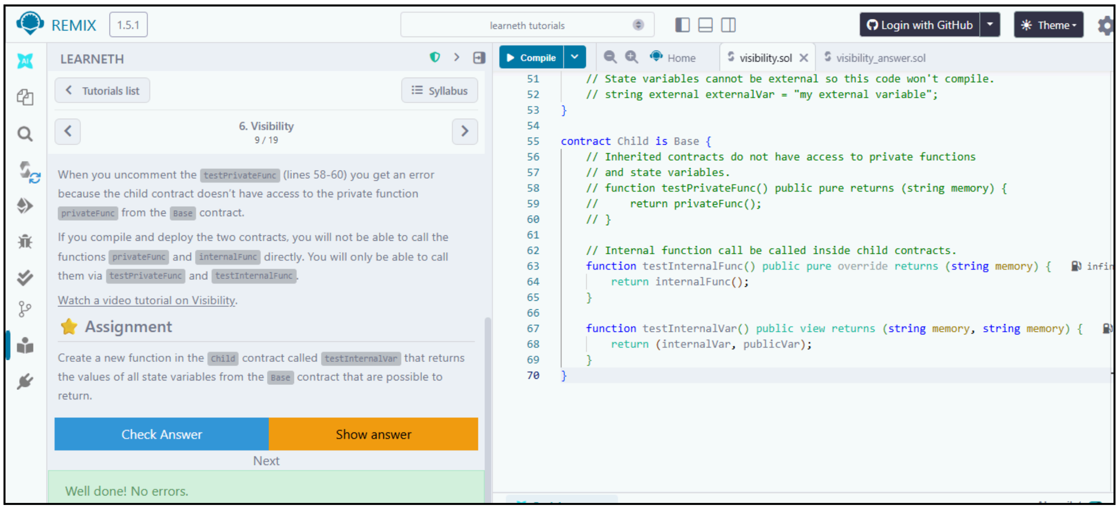
Check Answer | Show answer

Next

Well done! No errors.

```
71      function arrayInput(uint[] memory _arr) public {}    infinite gas
72
73      // Can use array for output
74      uint[] public arr;
75
76      function arrayOutput() public view returns (uint[] memory) {    infinite gas
77          return arr;
78      }
79
80      function returnTwo()    472 gas
81          public
82          pure
83          returns (
84              int i,
85              bool b
86          )
87      {
88          i = -2;
89          b = true;
90      }
91  }
```

Tutorial no. 9: Visibility



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.

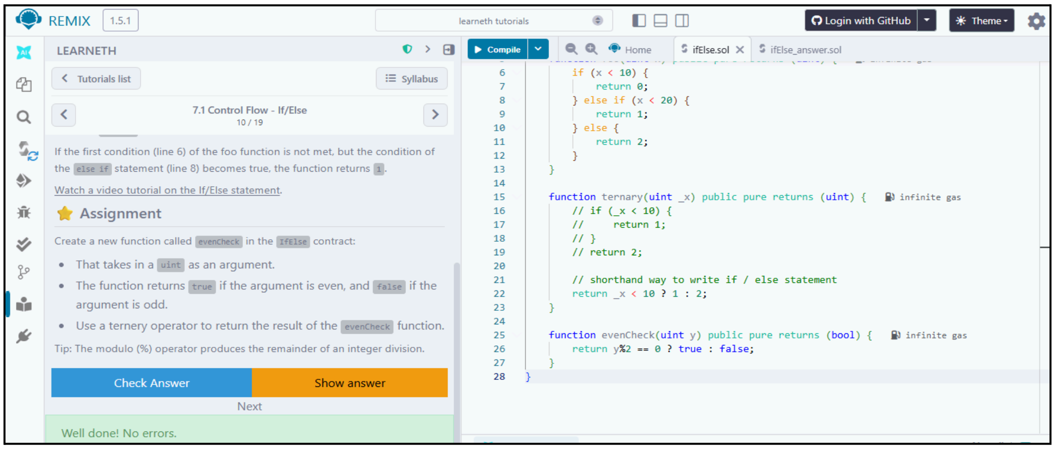
Check Answer | Show answer

Next

Well done! No errors.

```
51      // State variables cannot be external so this code won't compile.
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) {    infin
64          return internalFunc();
65      }
66
67      function testInternalVar() public view returns (string memory, string memory) {
68          return (internalVar, publicVar);
69      }
70  }
```

Tutorial no. 10: Control Flow - If/Else



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:

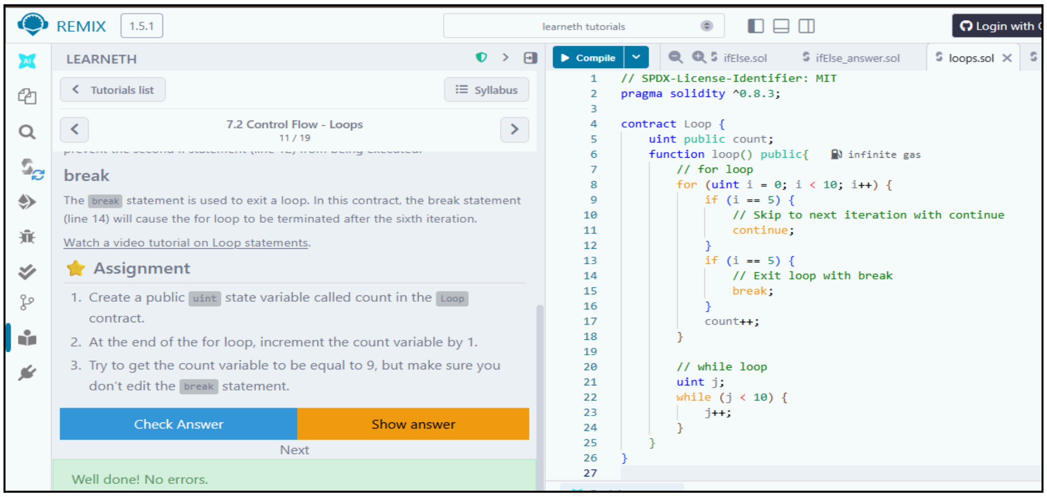
- That takes in a `uint` as an argument.
- The function returns `true` if the argument is even, and `false` if the argument is odd.
- Use a ternery operator to return the result of the `evenCheck` function.

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

Well done! No errors.

```
6          if (x < 10) {
7              return 0;
8          } else if (x < 20) {
9              return 1;
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
25     function evenCheck(uint y) public pure returns (bool) {    infinite gas
26         return y%2 == 0 ? true : false;
27     }
28 }
```

Tutorial no. 11: Control Flow - loops



7.2 Control Flow - Loops
11 / 19

break

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

Watch a video tutorial on Loop statements.

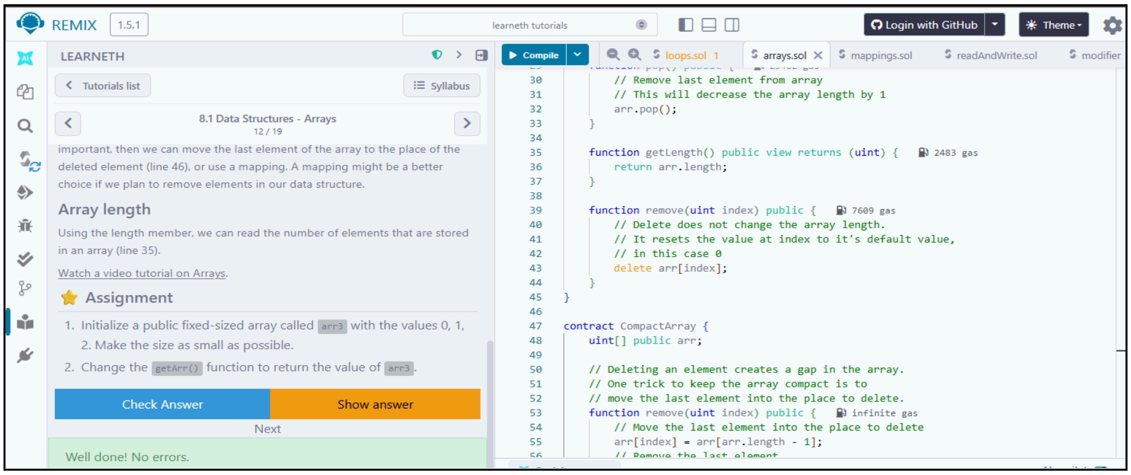
Assignment

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

Well done! No errors.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.3;
3
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
20         // while loop
21         uint j;
22         while (j < 10) {
23             j++;
24         }
25     }
26 }
27
```

Tutorial no. 12 : Data Structures-Arrays



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.

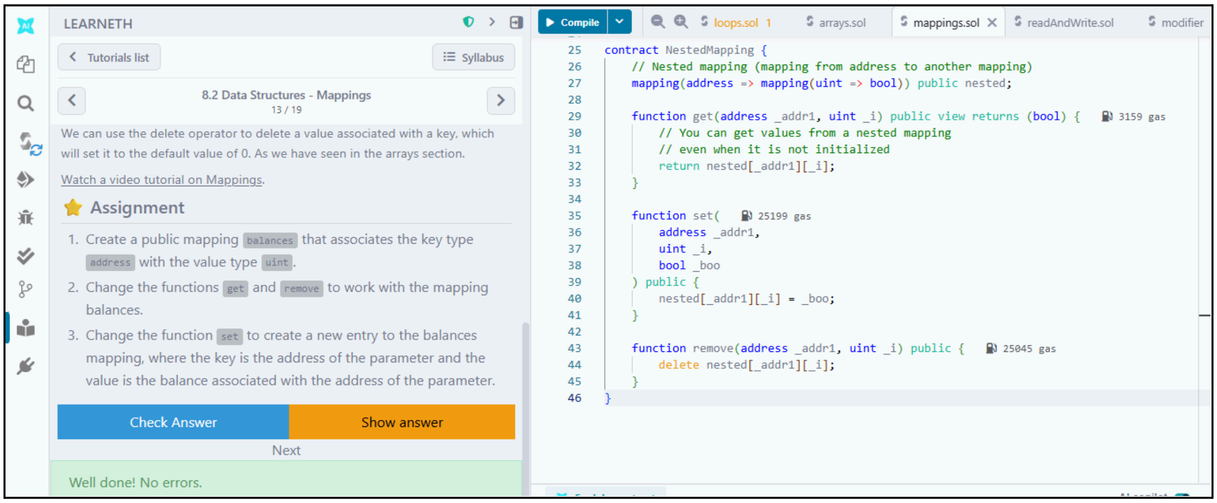
Assignment

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

Well done! No errors.

```
30         // Remove last element from array
31         // This will decrease the array length by 1
32         arr.pop();
33     }
34
35     function getLength() public view returns (uint) {    2483 gas
36         return arr.length;
37     }
38
39     function remove(uint index) public {    7609 gas
40         // Delete does not change the array length.
41         // It resets the value at index to it's default value,
42         // in this case 0
43         delete arr[index];
44     }
45 }
46
47 contract CompactArray {
48     uint[] public arr;
49
50     // Deleting an element creates a gap in the array.
51     // One trick to keep the array compact is to
52     // move the last element into the place to delete.
53     function remove(uint index) public {    infinite gas
54         // Move the last element into the place to delete
55         arr[index] = arr[arr.length - 1];
56         // Remove the last element
```

Tutorial no. 13 : Data Structures- Mappings



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.

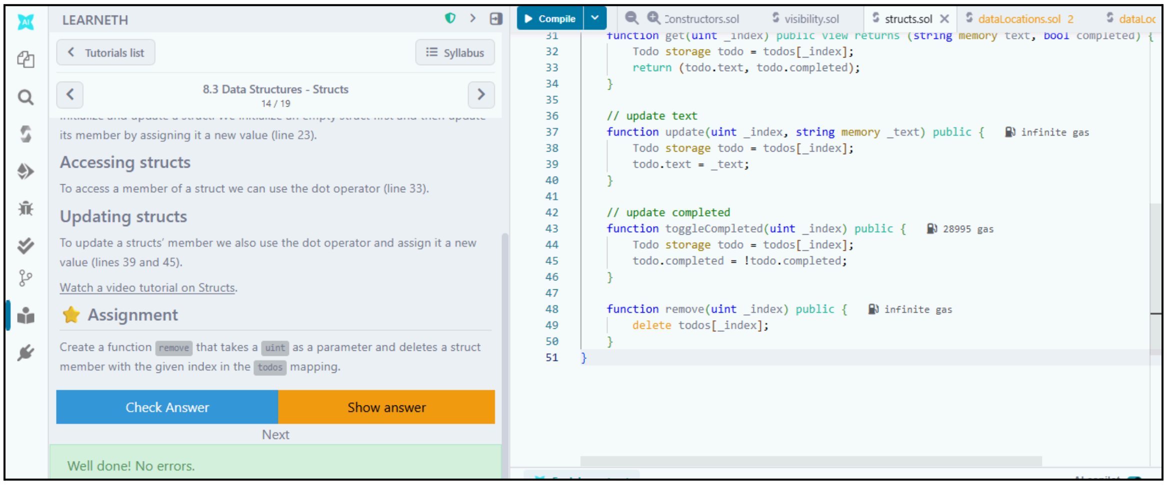
⭐ Assignment

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.

Well done! No errors.

```
25  contract NestedMapping {
26      // Nested mapping (mapping from address to another mapping)
27      mapping(address => mapping(uint => bool)) public nested;
28
29      function get(address _addr1, uint _i) public view returns (bool) {    3159 gas
30          // You can get values from a nested mapping
31          // even when it is not initialized
32          return nested[_addr1][_i];
33      }
34
35      function set(    25199 gas
36          address _addr1,
37          uint _i,
38          bool _boo
39      ) public {
40          nested[_addr1][_i] = _boo;
41      }
42
43      function remove(address _addr1, uint _i) public {    25045 gas
44          delete nested[_addr1][_i];
45      }
46  }
```

Tutorial no. 14 : Data Structures- Structs



8.3 Data Structures - Structs
14 / 19

initialize and update a struct. We initialize an empty struct first and then update 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.

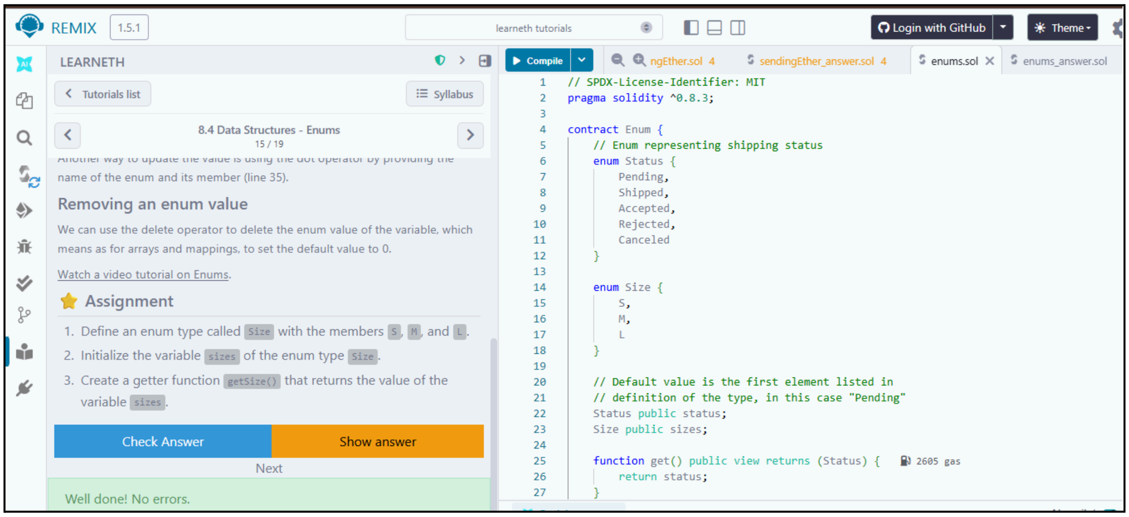
⭐ Assignment

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

Well done! No errors.

```
31      function get(uint _index) public view returns (string memory text, bool completed) {
32          Todo storage todo = todos[_index];
33          return (todo.text, todo.completed);
34      }
35
36      // update text
37      function update(uint _index, string memory _text) public {    infinite gas
38          Todo storage todo = todos[_index];
39          todo.text = _text;
40      }
41
42      // update completed
43      function toggleCompleted(uint _index) public {    28995 gas
44          Todo storage todo = todos[_index];
45          todo.completed = !todo.completed;
46      }
47
48      function remove(uint _index) public {    infinite gas
49          delete todos[_index];
50      }
51  }
```

Tutorial no. 15 : Data Structures- Enums



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.

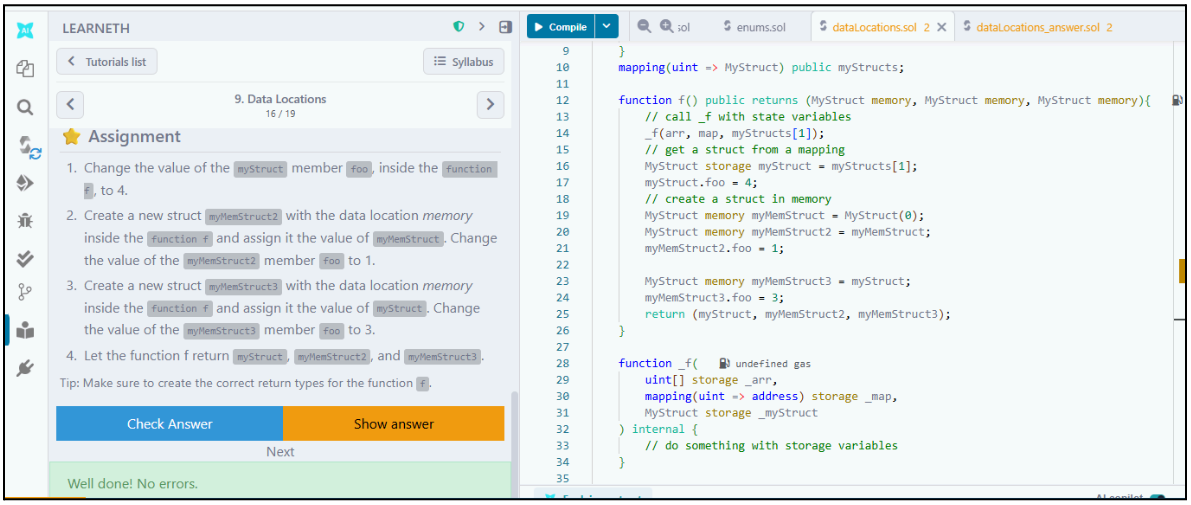
⭐ Assignment

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`.

Well done! No errors.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.3;
3
4   contract Enum {
5       // Enum representing shipping status
6       enum Status {
7           Pending,
8           Shipped,
9           Accepted,
10          Rejected,
11          Canceled
12      }
13
14      enum Size {
15          S,
16          M,
17          L
18      }
19
20      // Default value is the first element listed in
21      // definition of the type, in this case "Pending"
22      Status public status;
23      Size public sizes;
24
25      function get() public view returns (Status) {    2605 gas
26          return status;
27      }
```

Tutorial no. 16: Data Locations



9. Data Locations
16 / 19

Assignment

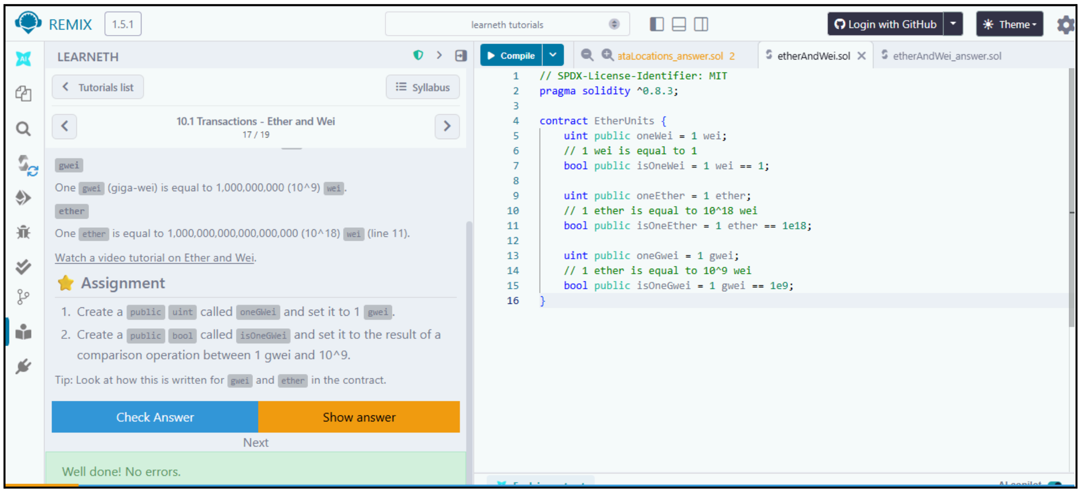
1. Change the value of the `myStruct` member `foo`, inside the `function f`, to 4.
2. Create a new struct `myMemStruct2` with the data location *memory* inside the `function f` and assign it the value of `myMemStruct`. Change the value of the `myMemStruct2` member `foo` to 1.
3. Create a new struct `myMemStruct3` with the data location *memory* inside the `function f` and assign it the value of `myStruct`. Change the value of the `myMemStruct3` member `foo` to 3.
4. Let the function f return `myStruct`, `myMemStruct2`, and `myMemStruct3`.

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

Well done! No errors.

```
    }
    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);
    }

    function _f(    undefined gas
        uint[] storage _arr,
        mapping(uint => address) storage _map,
        MyStruct storage _myStruct
    ) internal {
        // do something with storage variables
    }

```

Tutorial no. 17: Transactions- Ether and Wei



10.1 Transactions - Ether and Wei
17 / 19

`gwei`

One `gwei` (giga-wei) is equal to 1,000,000,000 (10^9) `wei`.

`ether`

One `ether` is equal to 1,000,000,000,000,000,000 (10^18) `wei` (line 11).

Watch a video tutorial on Ether and Wei.

Assignment

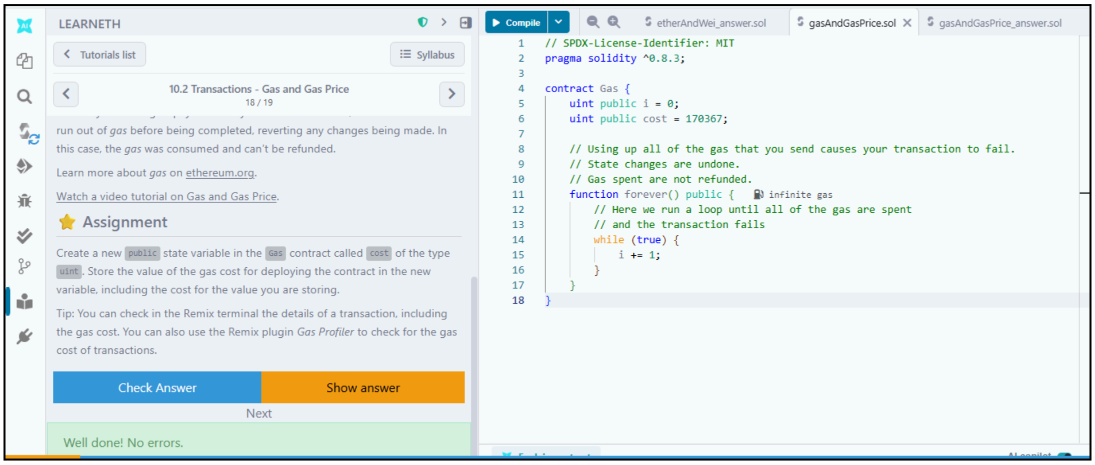
1. Create a `public` `uint` called `oneGWei` and set it to 1 `gwei`.
2. Create a `public` `bool` called `isOneGWei` and set it to the result of a comparison operation between 1 gwei and 10^9.

Tip: Look at how this is written for `gwei` and `ether` in the contract.

Well done! No errors.

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

Tutorial no.18: Transactions-Gas and Gas Price



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

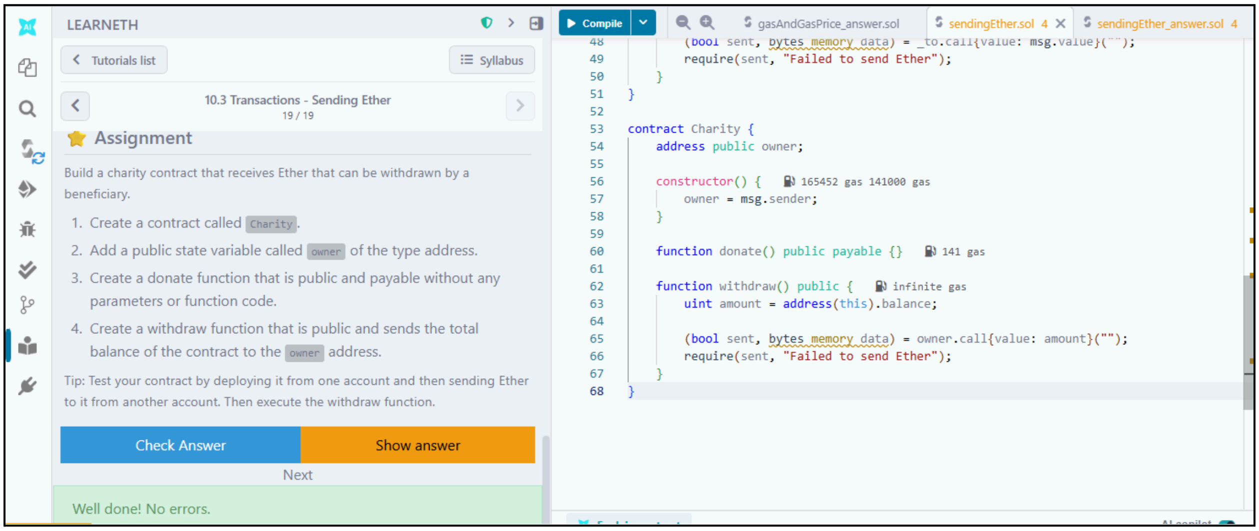
Create a new `public` state variable in the `Gas` contract called `cost` of the type `uint`. Store the value of the gas cost for deploying the contract in the new variable, including the cost for the value you are storing.

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.

Well done! No errors.

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

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;
        }
    }
}
```

Tutorial no. 19: Transactions- Sending Ether



10.3 Transactions - Sending Ether
19 / 19

Assignment

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

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

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

Check Answer | Show answer

Next

Well done! No errors.

```
        (bool sent, bytes memory data) = _to.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
    }
}

contract Charity {
    address public owner;

    constructor() {    165452 gas 141000 gas
        owner = msg.sender;
    }

    function donate() public payable {}    141 gas

    function withdraw() public {    infinite gas
        uint amount = address(this).balance;

        (bool sent, bytes memory data) = owner.call{value: amount}("");
        require(sent, "Failed to send Ether");
    }
}
```

Conclusion

In this experiment, practical exposure to Solidity programming was achieved through multiple hands-on tutorials executed in the Remix IDE environment. Core blockchain concepts such as state management, function types, visibility control, modifiers, constructors, and structured data handling were implemented and tested.

The experiment provided a deeper understanding of how smart contracts operate on the Ethereum blockchain, including how transactions consume gas and how Ether transfers are handled securely. By compiling, deploying, and interacting with contracts, theoretical concepts were reinforced through real-time execution.

Additionally, the use of arrays, mappings, structs, and enums demonstrated how complex decentralized applications can be structured efficiently. Awareness of data locations and gas optimization strategies further strengthened contract design skills.

Overall, this experiment established a strong conceptual and practical foundation in Solidity, enabling the design, deployment, and management of secure and efficient smart contracts for decentralized applications.