

Name: Palak Chanchlani

Class: D20A

Roll No: 12

Experiment 4

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

Theory

1. Introduction to Solidity and Smart Contracts

Solidity is a contract-oriented programming language designed for Ethereum. It is high-level, statically typed, and strongly influenced by languages like JavaScript, Python, and C++. Smart contracts written in Solidity allow developers to encode business logic directly on the blockchain, ensuring transparency, immutability, and automation.

Primitive Data Types:

- **uint / int:** Represent unsigned and signed integers of different bit sizes. For example, uint256 is widely used in token contracts to store balances, while int128 may be used in mathematical models.
- **bool:** Stores logical values (true or false). Often used in conditions such as verifying ownership or validating transactions.
- **address:** Holds Ethereum account or contract addresses (20 bytes). This type is essential for identifying participants and enabling contract-to-contract communication.
- **string / bytes:** Used for textual and binary data. Strings are suitable for names or metadata, while bytes are efficient for raw data like cryptographic hashes.

Variable Categories:

- **State Variables:** Permanently stored on-chain. Example: total supply of a token.
- **Local Variables:** Exist only during function execution and vanish afterward.
- **Global Variables:** Predefined values such as msg.sender, msg.value, and block.timestamp that provide transaction context.

Function Types:

- **pure:** Functions that neither read nor modify blockchain state. Example: a function that calculates square of a number.
- **view:** Functions that can read state variables but cannot alter them. Example: fetching a stored balance.

2. Inputs and Outputs in Functions

Functions in Solidity can accept parameters and return values, making contracts interactive.

- **Inputs:** Allow external users or contracts to pass data. Example: a deposit function accepting Ether.
- **Outputs:** Provide results back to the caller. Example: a withdrawal function returning a boolean.
- **Named Returns:** Improve readability and debugging by explicitly naming return variables.

3. Visibility, Modifiers, and Constructors

Visibility:

- **public:** Accessible internally and externally.
- **private:** Restricted to the same contract.
- **internal:** Available within the contract and derived contracts.
- **external:** Callable only from outside.

Modifiers: Reusable logic that alters function behaviour.

Example:

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

Constructor: Special function executed once during deployment. Typically used to initialize critical variables, such as setting the deployer as the contract owner.

4. Control Flow

Solidity supports conditional and iterative structures:

- **if-else:** Enables decision-making, such as checking sufficient balance before transfer.
- **Loops (for, while, do-while):** Allow repetitive tasks like iterating through arrays. Must be used cautiously to avoid excessive gas consumption.

5. Data Structures

- **Arrays:** Ordered collections of elements. Example: storing participant addresses.
- **Mappings:** Key-value pairs for fast lookups. Example: mapping(address => uint) for balances.
- **Structs:** Group related attributes.
Example:

```
struct Player {  
    string name;  
    uint score;  
}
```

- **Enums:** Define a set of constant values for clarity. Example:
enum Status { Pending, Active, Closed }

6. Data Locations

Solidity defines three primary data locations:

- **storage:** Permanent, on-chain data (state variables).
- **memory:** Temporary data during function execution.
- **calldata:** Non-modifiable, efficient storage for external function inputs.

Efficient use of data locations reduces gas costs and improves performance.

7. Transactions: Ether, Wei, Gas, and Transfers

Ether and Wei: Ether is Ethereum's native currency. 1 Ether = (10^{18}) Wei. This precision is vital for financial transactions.

Gas and Gas Price:

- Gas represents computational effort.
- Gas price determines transaction speed and cost. Higher gas prices incentivize miners to process transactions faster.

Sending Transactions:

- **transfer():** Safe, limited gas forwarding.
- **send():** Returns a boolean indicating success or failure.
- **call():** Flexible and widely used in modern Solidity development.

Output -

1. introduction.sol

LEARNEATH

Tutorials list Syllabus

1. Introduction 1 / 19

1. Introduction

Welcome to this interactive Solidity course for beginners.

In this first section, we will give you a short preview of the concepts we will cover in this course, look at an example smart contract, and show you how you can interact with this contract in the Remix IDE.

This contract is a counter contract that has the functionality to increase, decrease, and return the state of a counter variable.

If we look at the top of the contract, we can see some information about the contract like the license (line 1), the Solidity version (line 2), as well as the keyword `contract` and its name, `Counter` (line 4). We will cover these concepts in the next section about the [Basic Syntax](#).

With `uint public count` (line 5) we declare a state variable of the type `uint` with the visibility `public`. We will cover these concepts in our sections about [Variables](#), [Primitive Data Types](#), and [Visibility](#).

We then create a `get` function (line 8) that is defined with the `view` keyword and returns a `uint` type. Specifically, it returns the `count` variable. This contract has two more functions, an `inc` (line 13) and `dec` (line 18) function that increases or decreases our count variable. We will talk about these concepts in our sections about [Functions - Reading and Writing to a State Variable](#) and [Functions - View](#).

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

contract MyContract {
    string public name;

    constructor() {
        infinite gas 101000 gas
        name = "Alice";
    }
}
```

2. basicSyntax.sol

LEARNEATH

Tutorials list Syllabus

2. Basic Syntax 2 / 19

2. Basic Syntax

In this section, we will create our first *smart contract*. This contract only consists of a string that holds the value "Hello World!".

In the first line, we should specify the license that we want to use. You can find a comprehensive list of licenses here: <https://spdx.org/licenses/>.

Using the `pragma` keyword (line 3), we specify the Solidity version we want the compiler to use. In this case, it should be greater than or equal to `0.8.3` but less than `0.9.0`.

We define a contract with the keyword `contract` and give it a name, in this case, `HelloWorld` (line 5).

Inside our contract, we define a *state variable* `greet` that holds the string `"Hello World!"` (line 6).

Solidity is a *statically typed* language, which means that you need to specify the type of the variable when you declare it. In this case, `greet` is a `string`.

We also define the *visibility* of the variable, which specifies from where you can access it. In this case, it's a `public` variable that you can access from inside and outside the contract.

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.

```
pragma solidity ^0.8.3;

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

3. primitiveDataTypes.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and a 'Syllabus' section. The main content area is titled '3. Primitive Data Types' (3 / 19). It contains sections for 'bool', 'uint', 'int', 'address', and a summary. On the right, there's a code editor window titled 'Compile' with tabs for 'introduction.sol', 'basicSyntax.sol', 'primitiveDataTypes.sol', and 'variables.sol'. The 'primitiveDataTypes.sol' tab is active, displaying the following Solidity code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.31;

contract Primitives {
    address public addr = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2;
    address public newAddr = 0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB;

    // New public address (different from addr)
    int public neg = -10;

    // Public negative number
    uint8 public newU = 0;
}
```

4. variables.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and a 'Syllabus' section. The main content area is titled '4. Variables' (4 / 19). It contains sections for 'Global Variables', 'doSomething()', and a tip about reading the current block number. On the right, there's a code editor window titled 'Compiled' with tabs for 'introduction.sol', 'basicSyntax.sol', 'primitiveDataTypes.sol', and 'variables.sol'. The 'variables.sol' tab is active, displaying the following Solidity code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.31;

contract Variables {
    uint public blockNumber;

    function doSomething() public {
        blockNumber = block.number;
    }
}
```

5.1 readAndWrite.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main content area is titled '5.1 Functions - Reading and Writing to a State Variable' (5 / 19). It contains text about functions and state variables, followed by a code editor window. The code editor has tabs for 'variables.sol', '.prettierrc.json', 'readAndWrite.sol', and 'viewAndPure.sol'. The code itself is:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.31;

contract SimpleStorage {
    bool public b = true;

    function get_b() public view returns (bool) {
        return b;
    }
}
```

5.2 viewAndPure.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main content area is titled '5.2 Functions - View and Pure' (6 / 19). It contains text about view and pure functions, followed by a code editor window. The code editor has tabs for 'variables.sol', '.prettierrc.json', 'readAndWrite.sol', and 'viewAndPure.sol'. The code itself is:

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

contract ViewAndPure {
    uint public x = 1;

    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {
        return x + y;
    }

    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {
        return i + j;
    }

    function addToX2(uint y) public {
        x = x + y;
    }
}
```

5.3 modifiersAndConstructor.sol

The screenshot shows the LEARNETH platform interface. On the left, the 'Tutorials list' shows the current section: '5.3 Functions - Modifiers and Constructors'. The main content area displays the following text:

A constructor function is executed upon the creation of a contract. You can use it to run contract initialization code. The constructor can have parameters and is especially useful when you don't know certain initialization values before the deployment of the contract.

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

- 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.
- Make sure that `x` can only be increased.
- The body of the function `increaseX` should be empty.

Tip: Use modifiers.

Buttons at the bottom: 'Check Answer' (blue), 'Show answer' (orange), and 'Next' (green). A green bar at the bottom says 'Well done! No errors.'

On the right, the 'Compiled' tab is selected, showing the following Solidity code:

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

contract FunctionModifier {
    // We will use these variables to demonstrate how to use
    // modifiers.
    address public owner;
    uint public x = 10;
    bool public locked;

    constructor() {
        // set the transaction sender as the owner of the contract.
        owner = msg.sender;
    }

    // Modifier to check that the caller is the owner of
    // the contract.
    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        // Underscore is a special character only used inside
        // a function modifier and it tells Solidity to
        // execute the rest of the code.
        _;
    }

    // Modifiers can take inputs. This modifier checks that the
    // address passed in is not the zero address.
    modifier validAddress(address _addr) {
        require(_addr != address(0), "Not valid address");
        _;
    }
}
```

5.4 inputsAndOutputs.sol

The screenshot shows the LEARNETH platform interface. On the left, the 'Tutorials list' shows the current section: '5.4 Functions - Inputs and Outputs'. The main content area displays the following text:

In this section, we will learn more about the inputs and outputs of functions.

Multiple named Outputs

Functions can return multiple values that can be named and assigned to their name.

The `returnMany` function (line 6) shows how to return multiple values. You will often return multiple values. It could be a function that collects outputs of various functions and returns them in a single function call for example.

The `named` function (line 19) shows how to name return values. Naming return values helps with the readability of your contracts. Named return values make it easier to keep track of the values and the order in which they are returned. You can also assign values to a name.

The `assigned` function (line 33) shows how to assign values to a name. When you assign values to a name you can omit (leave out) the return statement and return them individually.

Deconstructing Assignments

You can use deconstructing assignments to unpack values into distinct variables.

The `destructuringAssignments` function (line 49) assigns the values of the `returnMany`

On the right, the 'Compiled' tab is selected, showing the following Solidity code:

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

contract Function {
    // Functions can return multiple values.
    function returnMany() {
        public
        pure
        returns (
            uint,
            bool,
            uint
        )
    }
    {
        return (1, true, 2);
    }

    // Return values can be named.
    function named() {
        public
        pure
        returns (
            uint x,
            bool b,
            uint y
        )
    }
    {
        return (1, true, 2);
    }

    // Return values can be assigned to their name.
    // In this case the return statement can be omitted.
}
```

6. visibility.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main area is titled '6. Visibility' with '9 / 19' below it. The content area has a dark header '6. Visibility' and a light body. It contains text about the `visibility` specifier and four types of visibilities: `external`, `public`, `internal`, and `private`. Below this is a code editor window titled 'Compiled' showing Solidity code for `visibility.sol`. The code defines a `Base` contract with private and internal functions, and a `test` contract that inherits from `Base` and calls its functions.

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

contract Base {
    // Private function can only be called
    // - inside this contract
    // Contracts that inherit this contract cannot call this function.
    function privateFunc() private pure returns (string memory) {    infinite gas
        return "private function called";
    }

    function testPrivateFunc() public pure returns (string memory) {    infinite gas
        return privateFunc();
    }

    // Internal function can be called
    // - inside this contract
    // - inside contracts that inherit this contract
    function internalFunc() internal pure returns (string memory) {    infinite gas
        return "internal function called";
    }

    function testInternalFunc() public pure virtual returns (string memory) {    infinite gas
        return internalFunc();
    }

    // Public functions can be called
    // - inside this contract
    // - inside contracts that inherit this contract
    // - by other contracts and accounts
    function publicFunc() public pure returns (string memory) {    infinite gas
        return "public function called";
    }
}
```

7.1 ControlFlow.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main area is titled '7.1 Control Flow - If/Else' with '10 / 19' below it. The content area contains text about the `else if` statement and a code editor window titled 'Compiled' showing Solidity code for `ControlFlow.sol`. The code defines a `IfElse` contract with three functions: `foo`, `ternary`, and `evenCheck`.

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

contract IfElse {
    function foo(uint x) public pure returns (uint) {    infinite gas
        if (x < 10) {
            return 0;
        } else if (x < 20) {
            return 1;
        } else {
            return 2;
        }
    }

    function ternary(uint _x) public pure returns (uint) {    infinite gas
        // if (_x < 10) {
        //     return 1;
        // }
        // return 2;
        // shorthand way to write if / else statement
        return _x < 10 ? 1 : 2;
    }

    function evenCheck(uint y) public pure returns (bool) {    infinite gas
        return y%2 == 0 ? true : false;
    }
}
```

7.2 loops.sol

The screenshot shows the LearnETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main content area is titled '7.2 Control Flow - Loops' (11 / 19). It contains sections on 'for', 'while', and 'do while' loops, each with explanatory text and examples. On the right, the Solidity code for 'loops.sol' is displayed in a 'Compiled' tab:

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

contract Loop {
    uint public count;
    function loop() public{    infinite gas
        // for loop
        for (uint i = 0; i < 10; i++) {
            if (i == 5) {
                // skip to next iteration with continue
                continue;
            }
            if (i == 5) {
                // Exit loop with break
                break;
            }
            count++;
        }

        // while loop
        uint j;
        while (j < 10) {
            j++;
        }
    }
}
```

At the bottom, there's a 'Deploy contract' button.

8.1 arrays.sol

The screenshot shows the LearnETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main content area is titled '8.1 Data Structures - Arrays' (12 / 19). It contains sections on 'Data Structures' and 'Arrays'. The 'Arrays' section includes text about reference types and how they differ from value types. On the right, the Solidity code for 'arrays.sol' is displayed in a 'Compiled' tab:

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

contract Array {
    // Several ways to initialize an array
    uint[] public arr;
    uint[] public arr2 = [1, 2, 3];
    // Fixed sized array, all elements initialize to 0
    uint[10] public myFixedSizeArr;
    uint[3] public arr3 = [0, 1, 2];

    function get(uint i) public view returns (uint) {    infinite gas
        return arr[i];
    }

    // Solidity can return the entire array.
    // But this function should be avoided for
    // arrays that can grow indefinitely in length.
    function getArr() public view returns (uint[3] memory) {    infinite gas
        return arr3;
    }

    function push(uint i) public {    46820 gas
        // Append to array
        // This will increase the array length by 1.
        arr.push(i);
    }

    function pop() public {    29462 gas
        // Remove last element from array
        // This will decrease the array length by 1.
        arr.pop();
    }
}
```

8.2 mapping.sol

The screenshot shows the LEARNETH platform interface. On the left, the 'Tutorials list' sidebar indicates '8.2 Data Structures - Mappings' is selected. The main content area displays the title '8.2 Data Structures - Mappings'. Below the title, there is a brief introduction to mappings, mentioning they are collections of key types and corresponding value types. It notes that mappings are similar to arrays but lack iteration support. The text also describes how to retrieve values from a mapping based on a known key.

Creating mappings

Mappings are declared with the syntax `mapping(keyType => valueType) VariableName`. The key type can be any built-in value type or any contract, but not a reference type. The value type can be of any type.

In this contract, we are creating the public mapping `myMap` (line 6) that associates the key type `address` with the value type `uint`.

Accessing values

The right side of the screenshot shows the Solidity code for `mapping.sol`:

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

contract Mapping {
    // Mapping from address to uint
    mapping(address => uint) public balances;

    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];
    }
}

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

    function get(address _addr1, uint _i) public view returns (bool) {
        // You can get values from a nested mapping
        // even when it is not initialized
        return nested[_addr1][_i];
    }
}
```

8.3 structs.sol

The screenshot shows the LEARNETH platform interface. On the left, the 'Tutorials list' sidebar indicates '8.3 Data Structures - Structs' is selected. The main content area displays the title '8.3 Data Structures - Structs'. Below the title, there is a brief introduction to structs, stating they are custom data types consisting of multiple variables.

Defining structs

We define a struct using the `struct` keyword and a name (line 5). Inside curly braces, we can define our struct's members, which consist of the variable names and their data types.

Initializing structs

There are different ways to initialize a struct:

- Positional parameters: We can provide the name of the struct and the values of its members as parameters in parentheses (line 16).
- Key-value mapping: We provide the name of the struct and the keys and values as a mapping inside curly braces (line 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).

The right side of the screenshot shows the Solidity code for `structs.sol`:

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

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: _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);
    }

    // Solidity automatically created a getter for 'todos' so
    // you don't actually need this function.
    function get(uint _index) public view returns (string memory text, bool completed) {
        Todo storage todo = todos[_index];
    }
}
```

8.4 enums.sol

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

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;
}

function getSize() public view returns (Size) {
    return sizes;
}
```

9. dataLocations.sol

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

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);
    }

    function _f(
        uint[] storage _arr,
        mapping(uint => address) storage _map,
        MyStruct storage _mystruct
    ) internal {
    }
}
```

10.1 etherAndWei.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main content area is titled '10.1 Transactions - Ether and Wei' (17 / 19). It contains text about Wei being the smallest subunit of Ether, and how numbers without a suffix are treated as Wei. It also mentions that One gwei (giga-wei) is equal to 1,000,000,000 (10^9) wei, and One ether is equal to 1,000,000,000,000,000,000 (10^18) wei. A tip at the bottom suggests looking at how this is written for gwei and ether in the contract.

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 | Next

Well done! No errors.

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

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;
}
```

10.2 gasAndGasPrice.sol

The screenshot shows the LEARNETH platform interface. On the left, there's a sidebar with 'Tutorials list' and 'Syllabus'. The main content area is titled '10.2 Transactions - Gas and Gas Price' (18 / 19). It contains text about gas prices being denoted in gwei, and introduces the concept of a gas limit. It explains that when sending a transaction, the sender specifies the maximum amount of gas that they are willing to pay for. If they set the limit too low, their transaction can run out of gas before being completed, reverting any changes being made. In this case, the gas was consumed and can't be refunded. A tip at the bottom suggests checking the Remix terminal for transaction details.

Gas limit

When sending a transaction, the sender specifies the maximum amount of gas that they are willing to pay for. If they set the limit too low, their transaction can run out of gas before being completed, reverting any changes being made. In this case, the gas was consumed and can't be refunded.

Learn more about `gas` on [ethereum.org](#).

Watch a video tutorial on `Gas and Gas Price`.

Assignment

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

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

Check Answer | Show answer | Next

Well done! No errors.

```
// 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;
        }
    }
}
```

10.3 sendingEther.sol

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

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
        return address(this).balance;
    }
}

contract SendEther {
```

Conclusion

This experiment provides practical exposure to Solidity programming by covering essential blockchain concepts such as data types, variables, functions, control structures, data storage, and transaction handling. By mastering these fundamentals, developers can design secure, efficient, and scalable smart contracts that form the backbone of decentralized applications.