

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 contract
- **Modifiers** are reusable code blocks that change the behavior of functions. They are often used for access control, such as restricting sensitive functions to the contract owner (`onlyOwner`).

- **Constructors** are special functions executed only once during contract deployment. They initialize important values, such as setting the deploying account as the owner of the contract.

4. Control Flow: if-else, loops

Control flow in Solidity is similar to traditional programming languages:

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

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

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

6. Data Locations

Solidity uses three primary data locations for storing variables:

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

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

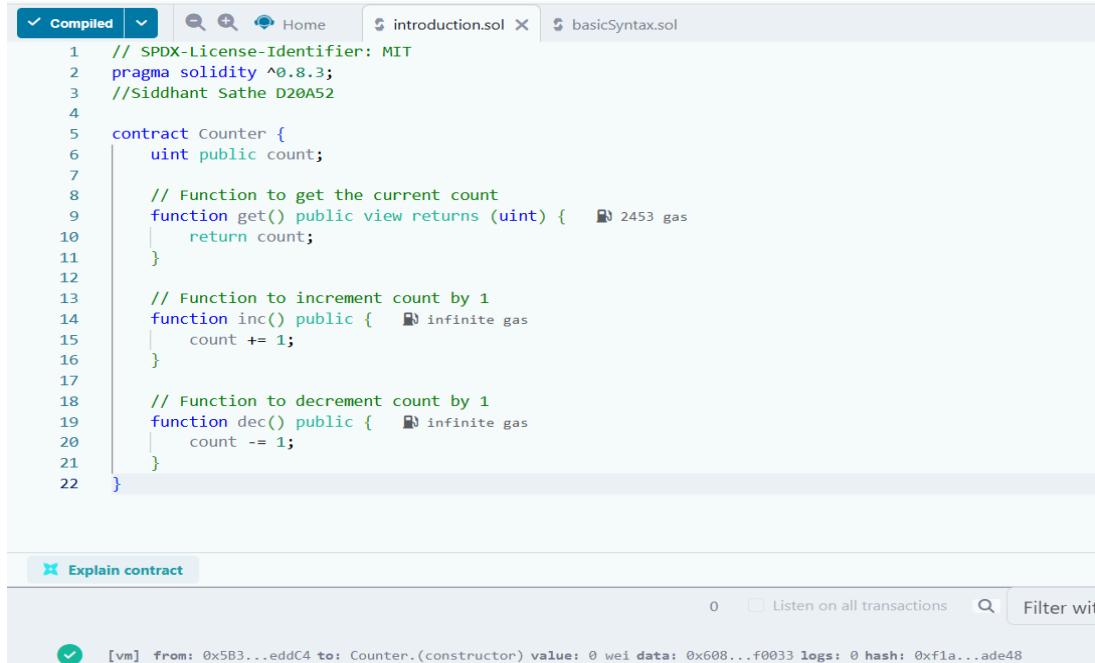
- **Ether and Wei**: Ether is the main currency in Ethereum. All values are measured in Wei, the smallest unit (1 Ether = 10^{18} Wei). This ensures high precision in financial transactions.
- **Gas and Gas Price**: Every transaction consumes gas, which represents computational

effort. The gas price determines how much Ether is paid per unit of gas. A higher gas price incentivizes miners to prioritize the transaction.

- **Sending Transactions:** Transactions are used for transferring Ether or interacting with contracts. Functions like transfer() and send() are commonly used, while call() provides more flexibility. Each transaction requires gas, making efficiency in contract design very important.

Implementation:

- Tutorial no. 1 – Compile the code



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52

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;
    }
}
```

Explain contract

0 Listen on all transactions Filter with transaction hash or address

[vm] from: 0x5B3...edc4 to: Counter.(constructor) value: 0 wei data: 0x608...f0033 logs: 0 hash: 0xf1a...ade48

- Tutorial no. 1 – get



Deployed Contracts 1

COUNTER AT 0xD91...39138 (0 ETH)

Balance: 0 ETH

dec

inc

count

get - call

0: uint256: 0

Low level interactions 1

CALLDATA

Transact

7 // Function to get the current count

Explain contract

0 Listen on all transactions Filter with transaction hash or address

decoded output: - 0

logs: [] 0

raw logs: [] 0

call to Counter.get: [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

call to Counter.get: [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c

Debug

Debug

Tutorial no. 1 – Increment

Deployed Contracts 1

COUNTER AT 0x5B3...9fBe8 (M) Balance: 0 ETH

dec
inc
count
get

0: uint256: 1

Explain contract

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0x5ed...791c8
call to Counter.get

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

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0xb22...2e5b3

● Tutorial no. 1 – Decrement

Deployed Contracts 1

COUNTER AT 0x5B3...9fBe8 (M) Balance: 0 ETH

dec
inc
count
get

0: uint256: 1

Explain contract

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

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0xb22...2e5b3
transact to Counter.dec pending ...

[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xf8e...9fBe8 value: 0 wei data: 0xb3b...cfa82 logs: 0 hash: 0x8ed...336ac

● Tutorial no. 2

LEARNEETH

Tutorials list

Syllabus

2. Basic Syntax 2 / 19

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.

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

Well done! No errors.

Compile basicSyntax.sol

1 // SPDX-License-Identifier: MIT
2 //Siddhant Sathe D20A52
3 pragma solidity ^0.8.3;
4
5 contract MyContract {
6 string public name = "Alice";
7 }

Explain contract

call [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() 0xf8e...9fBe8
transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8
hash: 0xb22...2e5b3
transact to Counter.dec pending ...

[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xf8e...9fBe8
hash: 0x8ed...336ac

● Tutorial no. 3

LEARNETH

< Tutorials list Syllabus

3. Primitive Data Types 3 / 19

You can learn more about these data types as well as *Fixed Point Numbers*, *Byte Arrays*, *Strings*, and more in the [Solidity documentation](#).

Later in the course, we will look at data structures like **Mappings**, **Arrays**, **Enums**, 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**

Well done! No errors.

Compiled primitiveDataTypes.sol × primitiveDataTypes_answer.sol

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
4
5 contract Primitives {
6     bool public boo = true;
7
8     /*
9      uint stands for unsigned integer, meaning non negative integers
10     different sizes are available
11     uint8 ranges from 0 to 2 ** 8 - 1
12     uint16 ranges from 0 to 2 ** 16 - 1
13     ...
14     uint256 ranges from 0 to 2 ** 256 - 1
15   */
16   uint8 public u8 = 1;
17   uint public u256 = 456;
18   uint public u = 123; // uint is an alias for uint256
19
20   /*
21   Negative numbers are allowed for int types

```

Explain contract

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

call [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0xd4...ce63c transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0xb22...2e5b3 transact to Counter.dec random

● Tutorial no. 4

LEARNETH

< Tutorials list Syllabus

4. Variables 4 / 19

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

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

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

Check Answer **Show answer**

Well done! No errors.

Compiled variables.sol 3 × variables_answer.sol 3

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
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 { 22334 gas
11        // Local variables are not saved to the blockchain.
12        uint i = 456;
13
14        // Here are some global variables
15        uint timestamp = block.timestamp; // Current block timestamp
16        address sender = msg.sender; // Address of the caller
17        blockNumber = block.number;
18    }

```

Explain contract

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

call [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0xd4...ce63c transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0xb22...2e5b3 transact to Counter.dec random

● Tutorial no. 5

LEARNETH

< Tutorials list Syllabus

5.1 Functions - Reading and Writing to a State Variable 5 / 19

To define a function, use the `function` keyword followed by a unique name.

If the function takes inputs like our `set` function (line 9), you must specify the parameter types and names. A common convention is to use an underscore as a prefix for the parameter name to distinguish them from state variables.

You can then set the visibility of a function and declare them `view` or `pure` as we do for the `get` function if they don't modify the state. Our `get` function also returns values, so we have to specify the return types. In this case, it's a `uint` since the state variable `num` that the function returns is a `uint`.

We will explore the particularities of Solidity functions in more detail in the following sections.

Watch a video tutorial on [Functions](#).

Assignment

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

Check Answer **Show answer**

Well done! No errors.

Compiled readAndWrite.sol ×

```

2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
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    }

```

Explain contract

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

call [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0xd4...ce63c transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 logs: 0 hash: 0xb22...2e5b3 transact to Counter.dec random

● Tutorial no. 6

LEARNETH

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

5.2 Functions - View and Pure 6 / 19

msg.sender and msg.data.

4. Calling any function not marked pure.
5. Using inline assembly that contains certain opcodes."

From the Solidity documentation.

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 `addToX` that takes the parameter `i` 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.

Compiled

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A5Z
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    function addToX2(uint y) public {
18        x = x + y;
19    }
20 }
```

Explain contract

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

CALL [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce63c
transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c0 logs: 0
hash: 0xb22...2e5b3
transact to Counter.dec pending ...

● Tutorial no. 7

LEARNETH

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

5.3 Functions - Modifiers and Constructors 7 / 19

Constructor

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

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.

Compiled

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A5Z
4 contract FunctionModifier {
5     // We will use these variables to demonstrate how to use
6     // modifiers.
7     address public owner;
8     uint public x = 10;
9     bool public locked;
10
11    constructor() {    461217 gas 414400 gas
12        // Set the transaction sender as the owner of the contract.
13        owner = msg.sender;
14    }
15
16    // Modifier to check that the caller is the owner of
17    // the contract.
18    modifier onlyOwner() {
19        require(msg.sender == owner, "Not owner");
20        // Underscore is a special character only used inside
21        // a function modifier and it tells Solidity to
```

Explain contract

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

CALL [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4
transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...3
hash: 0xb22...2e5b3
transact to Counter.dec pending ...

● Tutorial no. 8

LEARNETH

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

5.4 Functions - Inputs and Outputs 8 / 19

Input and Output restrictions

There are a few restrictions and best practices for the input and output parameters of contract functions.

"[Mappings] cannot be used as parameters or return parameters of contract functions that are publicly visible." From the [Solidity documentation](#).

Arrays can be used as parameters, as shown in the function `arrayInputs` (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 `true` and `true` without using a return statement.

Check Answer **Show answer**

Next

Well done! No errors.

Compiled

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A5Z
4 contract Function {
5     // Functions can return multiple values.
6     function returnMany()    infinite gas
7         public
8         pure
9         returns (
10             uint,
11             bool,
12             uint
13         )
14     {
15         return (1, true, 2);
16     }
17
18     // Return values can be named.
19     function named()    infinite gas
20         public
21         pure
22 }
```

Explain contract

0 Listen on all transactions

CALL [call] from: 0x5B380a6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4
transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...3
hash: 0xb22...2e5b3
transact to Counter.dec pending ...

● Tutorial no. 9

LEARNETH

6. Visibility 9 / 19

Syllabus

EXERCISE

- Can be called from other contracts or transactions
- State variables can not be `external`

In this example, we have two contracts, the `base` contract (line 4) and the `child` contract (line 55) which inherits the functions and state variables from the `base` contract.

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 `testInternalValue` that returns the values of all state variables from the `base` contract that are possible to return.

Check Answer **Show answer**

Next

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
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";
    }
}
```

Explain contract

0 Listen on all transactions Q Filter with transaction hash or address...

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56bedd4 to: Counter.get() data: 0x6d4...ce63c
transact to Counter.inc pending ...

✓ [vm] from: 0x5B3...eddc4 to: Counter.inc() 0xf8e8 value: 0 wei data: 0x371...303c0 logs: 0
hash: 0xb22...2e3b3

● Tutorial no. 10

LEARNETH

7.1 Control Flow - If/Else 10 / 19

Syllabus

7.1 Control Flow - If/Else

Solidity supports different control flow statements that determine which parts of the contract will be executed. The conditional `If/Else statement` enables contracts to make decisions depending on whether boolean conditions are either `true` or `false`.

Solidity differentiates between three different If/Else statements: `if`, `else`, and `else if`.

if

The `if` statement is the most basic statement that allows the contract to perform an action based on a boolean expression.

In this contract's `foo` function (line 5) the if statement (line 6) checks if `x` is smaller than `10`. If the statement is true, the function returns `0`.

else

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
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;
        // }
    }
}
```

Explain contract

0 Listen on all transactions Q Filter with transaction...

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56bedd4 to: Counter.get() data: 0x6d4...ce63c
transact to Counter.inc pending ...

● Tutorial no. 11

LEARNETH

7.2 Control Flow - Loops 11 / 19

Syllabus

7.2 Control Flow - Loops

Solidity supports iterative control flow statements that allow contracts to execute code repeatedly.

Solidity differentiates between three types of loops: `for`, `while`, and `do while` loops.

for

Generally, `for` loops (line 7) are great if you know how many times you want to execute a certain block of code. In solidity, you should specify this amount to avoid transactions running out of gas and failing if the amount of iterations is too high.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
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++;
        }
    }
}
```

Explain contract

● Tutorial no. 12

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

8.1 Data Structures - Arrays
12 / 19

8.1 Data Structures - Arrays

In the next sections, we will look into the data structures that we can use to organize and store our data in Solidity.

Arrays, mappings and structs are all *reference types*. Unlike *value types* (e.g. booleans or integers) reference types don't store their value directly. Instead, they store the location where the value is being stored. Multiple reference type variables could reference the same location, and a change in one variable would affect the others, therefore they need to be handled carefully.

In Solidity, an array stores an ordered list of values of the same type that are indexed numerically.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
4 contract Array {
5     // Several ways to initialize an array
6     uint[] public arr;
7     uint[] public arr2 = [1, 2, 3];
8     // Fixed sized array, all elements initialize to 0
9     uint[10] public myFixedSizeArr;
10
11    function get(uint i) public view returns (uint) {   // infinite gas
12        return arr[i];
13    }
14
15    // Solidity can return the entire array.
16    // But this function should be avoided for
17    // arrays that can grow indefinitely in length.
18    function getArr() public view returns (uint[]) memory {   // infinite gas

```

[Explain contract](#)

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

● Tutorial no. 13

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

8.2 Data Structures - Mappings
13 / 19

8.2 Data Structures - Mappings

In Solidity, *mappings* are a collection of key types and corresponding value type pairs.

The biggest difference between a mapping and an array is that you can't iterate over mappings. If we don't know a key we won't be able to access its value. If we need to know all of our data or iterate over it, we should use an array.

If we want to retrieve a value based on a known key we can use a mapping (e.g. addresses are often used as keys). Looking up values with a mapping is easier and cheaper than iterating over arrays. If arrays become too large, the gas cost of iterating over it could become too high and cause the transaction to fail.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
4 contract Mapping {
5     // Mapping from address to uint
6     mapping(address => uint) public myMap;
7
8     function get(address _addr) public view returns (uint) {   // 2872 gas
9         // Mapping always returns a value.
10        // If the value was never set, it will return the default value.
11        return myMap[_addr];
12    }
13
14    function set(address _addr, uint _i) public {   // 22842 gas
15        // Update the value at this address
16        myMap[_addr] = _i;
17    }
18

```

[Explain contract](#)

● Tutorial no. 14

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

8.3 Data Structures - Structs
14 / 19

8.3 Data Structures - Structs

In Solidity, we can define custom data types in the form of *structs*. Structs are a collection of variables that can consist of different data types.

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.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
4 contract Todos {
5     struct Todo {
6         string text;
7         bool completed;
8     }
9
10    // An array of 'Todo' structs
11    Todo[] public todos;
12
13    function create(string memory _text) public {   // infinite gas
14        // 3 ways to initialize a struct
15        // - calling it like a function
16        todos.push(Todo(_text, false));
17
18        // key value mapping

```

[Explain contract](#)

● Tutorial no. 15

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

8.4 Data Structures - Enums
15 / 19

8.4 Data Structures - Enums

In Solidity *enums* are custom data types consisting of a limited set of constant values. We use enums when our variables should only get assigned a value from a predefined set of values.

In this contract, the state variable `status` can get assigned a value from the limited set of provided values of the enum `Status` representing the various states of a shipping status.

Defining enums

We define an enum with the `enum` keyword, followed by the name of the custom type we want to create (line 6). Inside the curly braces, we define all available members of the

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
4 contract Enum {
5     // Enum representing shipping status
6     enum Status {
7         Pending,
8         Shipped,
9         Accepted,
10        Rejected,
11        Canceled
12    }
13
14    // Default value is the first element listed in
15    // definition of the type, in this case "Pending"
16    Status public status;
17
18    // Returns uint

```

[Explain contract](#)

● Tutorial no. 16

[◀ Tutorials list](#) [Syllabus](#)

9. Data Locations
16 / 19

9. Data Locations

The values of variables in Solidity can be stored in different data locations: *memory*, *storage*, and *calldata*.

As we have discussed before, variables of the value type store an independent copy of a value, while variables of the reference type (array, struct, mapping) only store the location (reference) of the value.

If we use a reference type in a function, we have to specify in which data location their values are stored. The price for the execution of the function is influenced by the data location; creating copies from reference types costs gas.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
contract DataLocations {
    uint[] public arr;
    mapping(uint => address) map;
    struct MyStruct {
        uint foo;
    }
    mapping(uint => MyStruct) myStructs;
}

function f() public {
    // call _f with state variables
    _f(arr, map, myStructs[1]);
    // get a struct from a mapping
    MyStruct storage myStruct = myStructs[1];
    // create a struct in memory
}
```

[Explain contract](#)

● Tutorial no. 17

[◀ Tutorials list](#) [Syllabus](#)

10.1 Transactions - Ether and Wei
17 / 19

number.

wei
Wei is the smallest subunit of Ether, named after the cryptographer Wei Dai. Ether numbers without a suffix are treated as `wei` (line 7).

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](#)

Next

Well done! No errors.

● Tutorial no. 18

[◀ Tutorials list](#) [Syllabus](#)

10.2 Transactions - Gas and Gas Price
18 / 19

Gas prices are denoted in gwei.

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.

```
pragma solidity ^0.8.3;
//Siddhant Sathe D20A52
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;
        }
    }
}
```

[Explain contract](#)

0 Listen on all transactions Filter with transaction hash

CALL [call] from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: Counter.get() data: 0x6d4...ce6 transact to Counter.inc pending ...

[vm] from: 0x5B3...eddC4 to: Counter.inc() 0xf8e...9fBe8 value: 0 wei data: 0x371...303c01 hash: 0xb22...2e5b3 transact to Counter.dec pending ...

[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xf8e...9fBe8 value: 0 wei data: 0xb3b...cfa82 1

● Tutorial no. 19

The screenshot shows a web-based tutorial interface. At the top, there are navigation buttons for 'Tutorials list' and 'Syllabus'. Below that, the title '10.3 Transactions - Sending Ether' is displayed, along with a page number '19 / 19'. The main content area contains a heading '10.3 Transactions - Sending Ether' and a paragraph explaining that the section covers how a contract can send and receive Ether. It lists three options: `transfer()`, `send()`, and `call()`. A 'transfer' example is shown with the code: `<address payable>.transfer(uint256 amount)`. A note states: '• `transfer()` throws an exception on failure'. To the right of this content is a block of Solidity code:

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Siddhant Sathe D20A52
4 contract ReceiveEther {
5     /*
6      * Which function is called, fallback() or receive()?
7      */
8     send Ether
9     |
10    msg.data is empty?
11    / \
12    yes no
13    / \
14    receive() exists? fallback()
15    / \
16    yes no
17    / \
18    receive() fallback()
```

Conclusion : Through this experiment, the fundamentals of Solidity programming were explored by completing practical assignments in Remix IDE. This hands-on approach improved understanding of smart contract creation, deployment, and execution on the Ethereum blockchain.