

## Blockchain Exp - 4

**AIM:** Hands on Solidity Programming Assignments for creating Smart Contracts.

**Theory:**

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

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

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

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

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

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

### 2. Inputs and Outputs to Functions

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

### 3. Visibility, Modifiers and Constructors

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

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

### 3. Control Flow: if-else, loops

Control flow in Solidity is similar to traditional programming languages:

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

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

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

### 6. Data Locations

Solidity uses three primary data locations for storing variables:

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

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

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

### Implementation:

#### Tutorial - 1 (compile)

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract Counter {
5     uint public count;
6
7     // Function to get the current count
8     function get() public view returns (uint) {    ↗ 2453 gas
9         return count;
10    }
11
12    // Function to increment count by 1
13    function inc() public {    ↗ infinite gas
14        count += 1;
15    }
16
17    // Function to decrement count by 1
18    function dec() public {    ↗ infinite gas
19        count -= 1;
20    }
21 }
```

```
Welcome to Remix 1.5.1

Your files are stored in indexedDB, 2.65 KB / 108.86 GB 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()` in 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: 0x955...0d83c
```

Debug ▾

## Tutorial - 1 (get)

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.execCurrent()` in 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: 0x955...0d83c  
call to Counter.get

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

## Tutorial - 1 (inc)

- 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: 0x955...0d83c  
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() 0xd91...39138 value: 0 wei  
data: 0x371...303c0 logs: 0 hash: 0x3ec...ee5b5  
Debug

## Tutorial - 1 (dec)

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: 0x955...0d83c  
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() 0xd91...39138 value: 0 wei  
data: 0x371...303c0 logs: 0 hash: 0x3ec...ee5b5  
transact to Counter.dec pending ...

[vm] from: 0x5B3...eddC4 to: Counter.dec() 0xd91...39138 value: 0 wei  
data: 0xb3b...cfa82 logs: 0 hash: 0xcc...79d46  
Debug

## Tutorial - 2

LEARNETH

< 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 Next

Well done! No errors.

Compile Home introduction.sol basicSyntax.sol

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than or equal to 0.8.3 and
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract MyContract {
    string public name = "Alice";
}
```

## Tutorial - 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**

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.

Check Answer Show answer Next

Well done! No errors.

Compile Home introduction.sol basicSyntax.sol primitiveDataTypes

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract Primitives {
    bool public boo = true;

    /*
        uint stands for unsigned integer, meaning non negative integers
        different sizes are available
            uint8 ranges from 0 to 2 ** 8 - 1
            uint16 ranges from 0 to 2 ** 16 - 1
            ...
            uint256 ranges from 0 to 2 ** 256 - 1
    */
    uint8 public u8 = 1;
    uint public u256 = 456;
    uint public u = 123; // uint is an alias for uint256

    /*
        Negative numbers are allowed for int types.
        Like uint, different ranges are available from int8 to int256
    */
    int8 public i8 = -1;
    int public i256 = 456;
    int public i = -123; // int is same as int256

    address public addr = 0xCA35b7d915458Ef540aDe6068dFe2F44E8fa733c;

    // Default values
    // Unassigned variables have a default value
    bool public defaultBoo; // false
    uint public defaultUint; // 0
```

## Tutorial - 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**

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.

Explains contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract Variables {
    // State variables are stored on the blockchain.
    string public text = "Hello";
    uint public num = 123;
    uint public blockNumber;

    function doSomething() public {
        // Local variables are not saved to the blockchain
        uint i = 456;

        // Here are some global variables
        uint timestamp = block.timestamp; // Current block timestamp
        address sender = msg.sender; // Address of the caller
        blockNumber = block.number;
    }
}
```

## Tutorial - 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**

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.

Explains contract

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract SimpleStorage {
    // State variable to store a number
    uint public num;
    bool public b = true;

    // You need to send a transaction to write to a state variable
    function set(uint _num) public {
        num = _num;
    }

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

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

## Tutorial - 6

The screenshot shows the LEARNETH platform interface. On the left, the 'Tutorials list' shows '5.2 Functions - View and Pure' as the current section. The main content area displays instructions for calling functions and using inline assembly. It includes a snippet from the Solidity documentation and a note about optimizing code for gas cost. A video tutorial link is also present. On the right, the Solidity compiler interface is shown with the file 'viewAndPure.sol' open. The code defines three functions: 'add' (pure), 'add' (public), and 'addToX2' (public).

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract ViewAndPure {
    uint public x = 1;

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

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

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

## Tutorial - 7

The screenshot shows the LEARNETH platform interface. On the left, the 'Tutorials list' shows '5.3 Functions - Modifiers and Constructors' as the current section. The main content area displays instructions for creating a constructor and using modifiers. It includes a note about initializing variables and a video tutorial link. On the right, the Solidity compiler interface is shown with the file 'modifiersAndConstructors.sol' open. The code defines a contract 'FunctionModifier' with a constructor and a modifier 'onlyOwner'.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract FunctionModifier {
    // We will use these variables to demonstrate how to use
    // modifiers.
    address public owner;
    uint public x = 10;
    bool public locked;

    constructor() {    // 461217 gas 414400 gas
        // 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");
        ;
    }
}
```

## Tutorial - 8

The screenshot shows a web-based Ethereum tutorial interface. On the left, there's a sidebar with "LEARNETH" at the top, followed by "Tutorials list" and "Syllabus". Below that is a section titled "5.4 Functions - Inputs and Outputs" with "8 / 19" next to it. Under this, there's a heading "Input and Output restrictions" with a note about mappings being publicly visible. It also mentions arrays as parameters and return values. A link to a video tutorial on function outputs is provided. A "★ Assignment" section asks to create a function "returnTwo" that returns two values. Below it are "Check Answer" and "Show answer" buttons, and a "Next" link. At the bottom, a green bar says "Well done! No errors."

Code Editor (Solidity):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract Function {
    // Functions can return multiple values.
    function returnMany() payable infinite gas
        public
        pure
        returns (
            uint,
            bool,
            uint
        )
    {
        return (1, true, 2);
    }

    // Return values can be named.
    function named() payable infinite gas
        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.
}
```

## Tutorial - 9

The screenshot shows a web-based Ethereum tutorial interface. On the left, there's a sidebar with "LEARNETH" at the top, followed by "Tutorials list" and "Syllabus". Below that is a section titled "6. Visibility" with "9 / 19" next to it. Under this, there's a heading "external" with a list: "Can be called from other contracts or transactions" and "State variables can not be external". It notes that two contracts, "Base" and "Child", have private functions. When uncommenting "testPrivateFunc", it causes an error because the child contract can't access it. A note says if you compile and deploy both, you can only call them via "privateFunc" and "internalFunc". A link to a video tutorial on visibility is provided. A "★ Assignment" section asks to create a function in the "Child" contract called "testInternalVar" that returns state variables from the "Base" contract. Below it are "Check Answer" and "Show answer" buttons, and a "Next" link. At the bottom, a green bar says "Well done! No errors."

Code Editor (Solidity):

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
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) {
        return "private function called";
    }

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

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

    function testInternalFunc() public pure virtual returns (string memory) {
        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) {
        return "public function called";
    }
}
```

## Tutorial - 10

LEARNETH

7.1 Control Flow - If/Else 10 / 19

In this contract, the `foo` function uses the `else if` statement (line 10) to return `2` if none of the other conditions are met.

**else if**

With the `else if` statement we can combine several conditions.

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

Watch a video tutorial on the If/Else statement.

**Assignment**

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

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

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

Check Answer Show answer Next

Well done! No errors.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
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;
}
```

## Tutorial - 11

LEARNETH

7.2 Control Flow - Loops 11 / 19

executed at least once, before checking on the condition.

**continue**

The `continue` statement is used to skip the remaining code block and start the next iteration of the loop. In this contract, the `continue` statement (line 10) will prevent the second if statement (line 12) from being executed.

**break**

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

Watch a video tutorial on Loop statements.

**Assignment**

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

Check Answer Show answer Next

Well done! No errors.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
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++;
        }
    }
}
```

## Tutorial - 12

LEARNETH

< Tutorials list Syllabus

8.1 Data Structures - Arrays 12 / 19

Using the `pop()` member function, we delete the last element of a dynamic array (line 3). We can use the `delete` operator to remove an element with a specific index from an array (line 42). When we remove an element with the `delete` operator all other elements stay the same, which means that the length of the array will stay the same. This will create a gap in our array. If the order of the array is not 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`.

Check Answer Show answer Next

Well done! No errors.

Compile Home introduction.sol arrays.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
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    uint[3] public arr3 = [0, 1, 2];
11
12    function get(uint i) public view returns (uint) { infinite gas
13        return arr[i];
14    }
15
16    // Solidity can return the entire array.
17    // But this function should be avoided for
18    // arrays that can grow indefinitely in length.
19    function getArr() public view returns (uint[3] memory) { infinite gas
20        return arr3;
21    }
22
23    function push(uint i) public { 46820 gas
24        // Append to array
25        // This will increase the array length by 1.
26        arr.push(i);
27    }
28
29    function pop() public { 29462 gas
30        // Remove last element from array
31        // This will decrease the array length by 1
32        arr.pop();
33    }
34}
```

## Tutorial - 13

LEARNETH

< Tutorials list Syllabus

8.2 Data Structures - Mappings 13 / 19

Setting values

We set a new value for a key by providing the mapping's name and key in brackets and assigning it a new value (line 16).

### Removing values

We can use the `delete` operator to delete a value associated with a key, which will set it to the default value of 0. As we have seen in the arrays section.

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

### ★ Assignment

1. Create a public mapping `balances` that associates the key type `address` with the value type `uint`.
2. Change the functions `get` and `remove` to work with the mapping `balances`.
3. Change the function `set` to create a new entry to the `balances` mapping, where the key is the address of the parameter and the value is the balance associated with the address of the parameter.

Check Answer Show answer Next

Well done! No errors.

Compile Home introduction.sol mappings.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.3;
3 //Ansh Sarfare D20A51
4 contract Mapping {
5     // Mapping from address to uint
6     mapping(address => uint) public balances;
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 balances[_addr];
12    }
13
14    function set(address _addr) public { 25256 gas
15        // Update the value at this address
16        balances[_addr] = _addr.balance;
17    }
18
19    function remove(address _addr) public { 5566 gas
20        // Reset the value to the default value.
21        delete balances[_addr];
22    }
23
24 contract NestedMapping {
25     // Nested mapping (mapping from address to another mapping)
26     mapping(address => mapping(uint => bool)) public nested;
27
28     function get(address _addr1, uint _i) public view returns (bool) { 5566 gas
29         // You can get values from a nested mapping
30         // even when it is not initialized
31         return nested[_addr1][_i];
32     }
33}
```

## Tutorial - 14

The screenshot shows a web-based Solidity IDE interface. On the left, there's a sidebar with 'LEARNETH' branding, a 'Tutorials list' button, and a 'Syllabus' button. The main content area has a title '8.3 Data Structures - Structs' with '14 / 19' below it. A text block explains struct members as parameters in parentheses (line 16) and key-value mapping (line 19). It also describes initializing and updating a struct (lines 20-23). Below this, sections for 'Accessing structs' and 'Updating structs' are shown, each with a brief description and a 'Watch a video tutorial on Structs.' link.

**★ 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.

Code editor:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarface D20A51
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];
    }
}
```

Action buttons at the bottom:

- Check Answer
- Show answer
- Next

Feedback message: Well done! No errors.

## Tutorial - 15

The screenshot shows a web-based Solidity IDE interface. The sidebar has 'LEARNETH' branding, a 'Tutorials list' button, and a 'Syllabus' button. The main content area has a title '8.4 Data Structures - Enums' with '15 / 19' below it. A section titled 'Updating an enum value' explains how to update an enum value by assigning its `uint` representation (line 30) or using the dot operator (line 35). Another section, 'Removing an enum value', describes using the delete operator to set the default value to 0.

**★ Assignment**

- Define an enum type called `Size` with the members `S`, `M`, and `L`.
- Initialize the variable `sizes` of the enum type `Size`.
- Create a getter function `getSize()` that returns the value of the variable `sizes`.

Code editor:

```
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47 }
```

Action buttons at the bottom:

- Check Answer
- Show answer
- Next

Feedback message: Well done! No errors.

## Tutorial - 16

The screenshot shows the LEARNETH platform interface. On the left, the tutorial navigation bar includes 'Tutorials list' and 'Syllabus'. The current section is '9. Data Locations' (16 / 19). A tip message states: 'function f (line 12) and assign it the value of myStruct, changes in myMemStruct3 would not affect the values stored in the mapping myStructs (line 10). As we said in the beginning, when creating contracts we have to be mindful of gas costs. Therefore, we need to use data locations that require the lowest amount of gas possible.' Below this is a 'Assignment' section with four numbered tasks:

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

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

On the right, the Solidity code for `dataLocations.sol` is displayed:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
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( ) internal {
        uint[] storage _arr,
        mapping(uint => address) storage _map,
        MyStruct storage _myStruct
    }
}
```

Buttons at the bottom include 'Check Answer', 'Show answer', and 'Next'. A success message 'Well done! No errors.' is shown at the bottom.

## Tutorial - 17

The screenshot shows the LEARNETH platform interface. On the left, the tutorial navigation bar includes 'Tutorials list' and 'Syllabus'. The current section is '10.1 Transactions - Ether and Wei' (17 / 19). A tip message states: 'To specify a unit of `Ether`, we can add the suffixes `wei`, `gwei`, or `ether` to a literal number.' Below this is a 'Assignment' section with three numbered tasks:

- 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  $10^9$ .

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

On the right, the Solidity code for `etherAndWei.sol` is displayed:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
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;
}
```

Buttons at the bottom include 'Check Answer', 'Show answer', and 'Next'. A success message 'Well done! No errors.' is shown at the bottom.

## Tutorial - 18

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.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
contract Gas {
    uint public i = 0;
    uint public cost = 170367;
}
// Using up all of the gas that you send causes your transaction to revert.
// 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 - 19

If you change the parameter type for the functions `sendViaTransfer` and `sendViaSend` (line 33 and 38) from `payable address` to `address`, you won't be able to use `transfer()` (line 35) or `send()` (line 41).

Watch a video tutorial on *Sending Ether*.

### 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.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
//Ansh Sarfare D20A51
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) {
        return address(this).balance;
    }
}

contract SendEther {
```

**Conclusion:** Through this experiment, the fundamentals of Solidity programming were explored by completing practical assignments in the Remix IDE. Concepts such as data types, variables, functions, visibility, modifiers, constructors, control flow, data structures, and transactions were implemented and understood. The hands-on practice helped in designing, compiling, and deploying smart contracts on the Remix VM, thereby strengthening the understanding of blockchain concepts. This experiment provided a strong foundation for developing and managing smart contracts efficiently.