

Batch: A2 Roll No.: 1911031

Experiment No. 04

Title: Development of smart contract for any simple application – I

Objective: To develop a smart contract that will be used for transferring ethers from one account to another via a third party.

Expected Outcome of Experiment:

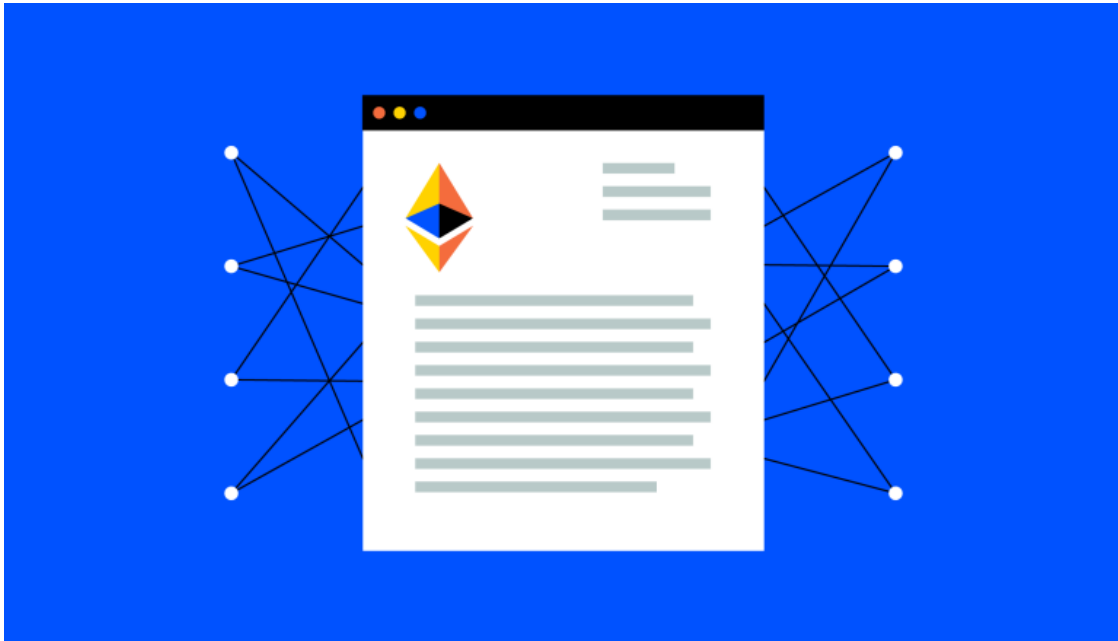
CO	Outcome
CO2	Learn Solidity language & Multiple Technology-based developments
CO3	Apply the algorithm and techniques used in Blockchain

Books/ Journals/ Websites referred:

1. <https://remix.ethereum.org/>
2. <https://www.coinbase.com/learn/crypto-basics/what-is-a-smart-contract>
3. <https://www.realvision.com/blog/how-to-create-a-smart-contract-on-ethereum>
4. <https://corporatefinanceinstitute.com/resources/valuation/smart-contracts/>

Abstract:-

What is a smart contract?



Definition

A smart contract, like any contract, establishes the terms of an agreement. But unlike a traditional contract, a smart contract's terms are executed as code running on a blockchain like Ethereum. Smart contracts allow developers to build apps that take advantage of blockchain security, reliability, and accessibility while offering sophisticated peer-to-peer functionality — everything from loans and insurance to logistics and gaming.

Just like any contract, smart contracts lay out the terms of an agreement or deal. What makes smart contracts “smart,” however, is that the terms are established and executed as code running on a blockchain, rather than on paper sitting on a lawyer's desk. Smart contracts expand on the basic idea behind Bitcoin — sending and receiving money without a “trusted intermediary” like a bank in the middle — to make it possible to securely automate and decentralize virtually any kind of deal or transaction, no matter how complex. And because they run on a blockchain like Ethereum, they offer security, reliability, and borderless accessibility.

Why are smart contracts important?

Smart contracts allow developers to build a wide variety of decentralized apps and tokens. They're used in everything from new financial tools to logistics and game experiences, and they're stored on a blockchain like any other crypto transaction. Once

a smart-contract app has been added to the blockchain, it generally can't be reversed or changed (although there are some exceptions).

Smart-contract-powered apps are often referred to as “decentralized applications” or “dapps” – and they include decentralized finance (or DeFi) tech that aims to transform the banking industry. DeFi apps allow cryptocurrency holders to engage in complex financial transactions — saving, loans, insurance — without a bank or other financial institution taking a cut and from anywhere in the world. Some of the more popular current smart-contract powered applications include:

- **Uniswap:** A decentralized exchange that allows users, via smart contract, to trade certain kinds of crypto without any central authority setting the exchange rates.
- **Compound:** A platform that uses smart contracts to let investors earn interest and borrowers to instantly get a loan without the need for a bank in the middle.
- **USDC:** A cryptocurrency that is pegged via smart contract to the US dollar, making one USDC worth one U.S. dollar. USDC is part of a newer category of digital money known as stablecoins.

So how would you use these smart contract-powered tools? Imagine you're holding some Ethereum that you'd like to trade for USDC. You could put some Ethereum into Uniswap, which, via smart contract, can automatically find you the best exchange rate, make the trade, and send you your USDC. You could then put some of your USDC into Compound to lend to others and receive an algorithmically determined rate of interest — all without using a bank or other financial institution.

In traditional finance, swapping currencies is expensive and time consuming. And it isn't easy or secure for individuals to loan out their liquid assets to strangers on the other side of the world. But smart contracts make both of those scenarios, and a vast variety of others, possible.

How do smart contracts work?

Smart contracts were first proposed in the 1990s by a computer scientist and lawyer named Nick Szabo. Szabo famously compared a smart contract to a vending machine. Imagine a machine that sells cans of soda for a quarter. If you put a dollar into the machine and select a soda, the machine is hardwired to either produce your drink and 75 cents in change, or (if your choice is sold out) to prompt you to make another selection or get your dollar back. This is an example of a simple smart contract. Just like a soda machine can automate a sale without a human intermediary, smart contracts can automate virtually any kind of exchange.

Currently, Ethereum is the most popular smart contract platform, but many other cryptocurrency blockchains (including EOS, Neo, Tezos, Tron, Polkadot, and

Algorand) can run them. A smart contract can be created and deployed to a blockchain by anyone. Their code is transparent and publicly verifiable, which means that any interested party can see exactly what logic a smart contract follows when it receives digital assets.

- Smart contracts are written in a variety of programming languages (including Solidity, Web Assembly, and Michelson). On the Ethereum network, each smart contract's code is stored on the blockchain, allowing any interested party to inspect the contract's code and current state to verify its functionality.
- Each computer on the network (or "node") stores a copy of all existing smart contracts and their current state alongside the blockchain and transaction data.
- When a smart contract receives funds from a user, its code is executed by all nodes in the network in order to reach a consensus about the outcome and resulting flow of value. This is what allows smart contracts to securely run without any central authority, even when users are making complex financial transactions with unknown entities.
- To execute a smart contract on the Ethereum network, you will generally have to pay a fee called "gas" (so named because these fees keep the blockchain running).
- Once deployed onto a blockchain, smart contracts generally can't be altered, even by their creator. (There are exceptions to this rule.) This helps ensure that they can't be censored or shut down.

Related Theory: -

How Ethereum Blockchain Platform executes Smart Contracts?

Before discussing how to create a smart contract on the Ethereum platform, you need to understand the Ethereum blockchain and how it runs smart contracts. So, let's understand the execution environment first.

- **Ethereum Virtual Machine (EVM):** The purpose of EVM is to serve as a runtime environment for smart contracts built on Ethereum. Consider it as a global supercomputer that executes all the smart contracts. As the name indicates, Ethereum Virtual Machine is not physical but a virtual machine. The functionality of EVM is restricted to virtual machines.
- **Gas:** In the Ethereum Virtual Machine, gas is a measurement unit used for assigning fees to each transaction with a smart contract. Each computation

happening in the EVM needs some amount of gas. The more complex the computation is, the more the gas is required to run the smart contracts.

- **Solidity:** Solidity is a smart contract programming language on Ethereum. Developed on the top of the EVM, it is similar to the object-oriented programming language that uses class and methods. It allows you to perform arbitrary computations, but it is used to send and receive tokens and store states. When it comes to syntax, Solidity is greatly influenced by C++, Python, and Javascript so that developers can understand its syntax quickly.

How to create a Smart contract

Step 1: Connect to the Ethereum network.

You can connect to the Ethereum mainnet by downloading and installing a MetaMask wallet on your Chrome browser and enabling it. Once you have completed the setup, proceed to connect it to the Ethereum mainnet.

Step 2: Choose a test network.

You will find a list of test networks in your Metamask wallet. Select one, which will be used to test the smart contract you are building. The test networks include:

- Robsten Test network
- Rinkeby Test network
- Kovan Test network
- Goerli Test network.

Step 3: Fund your wallet with Testnet ETH.

Eventually, when you are ready to test your smart contract, you will need to have Testnet ETH in your wallet. Fortunately, the process of adding Testnet ETH in MetaMask is straightforward. Click on the “Deposit” and “Get Ether” buttons under the Test Faucet and proceed with the instructions.

Step 4: Use the Remix browser to write your smart contract.

You can use the editor in **Remix browser IDE** to write your smart contract in Solidity. Remix browser is the best option for writing smart contracts as it comes with several features and is usually used to write basic smart contracts.

Step 5: Create a .sol extension file.

Open the Remix Browser and click on the (+) icon on the left side to create a .sol extension. This makes any programmed file solidity-compatible.

Step 6: Complete your smart contract code.

Choose a version of the compiler from the Remix browser and compile the solidity smart contract code.

Step 7: Deploy the smart contract.

Deploy the smart contract on your selected Ethereum test network by clicking on the deploy button on the Remix browser. Once the transaction is complete, the address of the smart contract will appear on the right-hand side of the Remix browser.

Step 8: Make smart contract ready to go live.

To prepare a smart contract to go live on a blockchain, procedures such as testing or auditing are strongly recommended, besides other considerations. Once reassured that the smart contract is ready to hit the market, the smart contract can be deployed to the Ethereum mainnet.

Related Theory (contd...): -

Uses of Smart Contracts

Smart contracts can be used in a variety of fields, from healthcare to supply chain to financial services. Some examples are as follows:

1. Government voting system

Smart contracts provide a secure environment making the voting system less susceptible to manipulation. Votes using smart contracts would be ledger-protected, which is extremely difficult to decode.

Moreover, smart contracts could increase the turnover of voters, which is historically low due to the inefficient system that requires voters to line up, show identity, and complete forms. Voting, when transferred online using smart contracts, can increase the number of participants in a voting system.

2. Healthcare

Blockchain can store the encoded health records of patients with a private key. Only specific individuals would be granted access to the records for privacy concerns. Similarly, research can be conducted confidentially and securely using smart contracts.

All hospital receipts of patients can be stored on the blockchain and automatically shared with insurance companies as proof of service. Moreover,

the ledger can be used for different activities, such as managing supplies, supervising drugs, and regulation compliance.

3. Supply chain

Traditionally, supply chains suffer due to paper-based systems where forms pass through multiple channels to get approvals. The laborious process increases the risk of fraud and loss.

Blockchain can nullify such risks by delivering an accessible and secure digital version to parties involved in the chain. Smart contracts can be used for inventory management and the automation of payments and tasks.

4. Financial services

Smart contracts help in transforming traditional financial services in multiple ways. In the case of insurance claims, they perform error checking, routing, and transfer payments to the user if everything is found appropriate.

Smart contracts incorporate critical tools for bookkeeping and eliminate the possibility of infiltration of accounting records. They also enable shareholders to take part in decision making in a transparent way. Also, they help in trade clearing, where the funds are transferred once the amounts of trade settlements are calculated.

Benefits of Smart Contracts

1. Autonomy and savings

Smart contracts do not need brokers or other intermediaries to confirm the agreement; thus, they eliminate the risk of manipulation by third parties. Moreover, the absence of intermediary in smart contracts results in cost savings.

2. Backup

All the documents stored on blockchain are duplicated multiple times; thus, originals can be restored in the event of any data loss.

3. Safety

Smart contracts are encrypted, and cryptography keeps all the documents safe from infiltration.

4. Speed

Smart contracts automate tasks by using computer protocols, saving hours of various business processes.

5. Accuracy

Using smart contracts results in the elimination of errors that occur due to manual filling of numerous forms.

Limitations of Smart Contracts

1. Difficult to change

Changing smart contract processes is almost impossible, any error in the code can be time-consuming and expensive to correct.

2. Possibility of loopholes

According to the concept of good faith, parties will deal fairly and not get benefits unethically from a contract. However, using smart contracts makes it difficult to ensure that the terms are met according to what was agreed upon.

3. Third party

Although smart contracts seek to eliminate third-party involvement, it is not possible to eliminate them. Third parties assume different roles from the ones they take in traditional contracts. For example, lawyers will not be needed to prepare individual contracts; however, they will be needed by developers to understand the terms to create codes for smart contracts.

4. Vague terms

Since contracts include terms that are not always understood, smart contracts are not always able to handle terms and conditions that are vague.

Code:

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

contract exp4 {
    address payable public payer;
    address payable public payee;
    address payable public thirdParty;
    uint256 public amount;

    constructor (address payable _payer, address payable _payee, uint256
_amount) payable {
        payer = _payer;
        payee = _payee;
        amount = (_amount);
        thirdParty = payable(msg.sender);
    }

    function deposit() public payable {
        require(payer == msg.sender, "Sender is not the payer");
        require(amount*(1 ether) >= address(this).balance, "Cant send
more than specified escrow amount");
    }

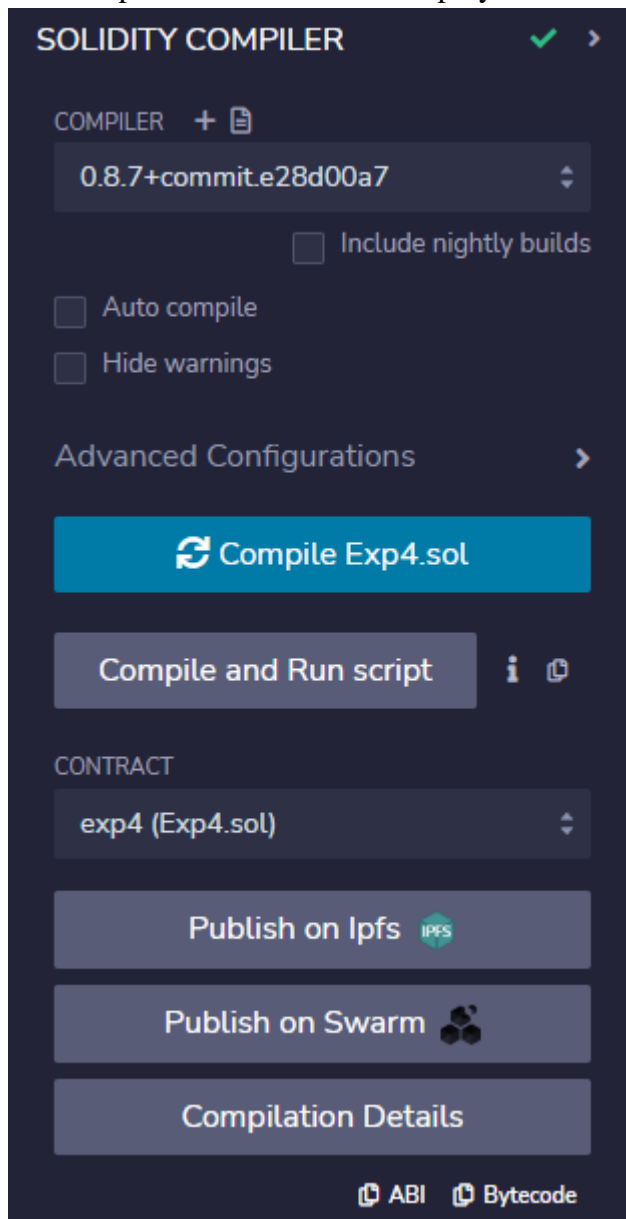
    function release() public payable {
        require(amount*(1 ether) == address(this).balance, "Cannot
release, fund insufficient");
        require(msg.sender == thirdParty, "Sender Not thirdParty");
        payee.transfer(address(this).balance);
    }

    function balanceOf() public view returns (uint256) {
        return address(this).balance;
    }
}
```


Implementation Details:

1. Enlist all the Steps followed and various options explored

- Open remix ide on the web to start writing some smart contract code
- Create a new file to write code in it
- Now Write your business logic
- Compile the code and then deploy it



DEPLOY & RUN TRANSACTIONS ✓ >

ENVIRONMENT 

Remix VM (London) ⌵ i

VM

ACCOUNT +

0x5B3...eddC4 (100 ether) ⌵ 📋 🔗

GAS LIMIT

3000000

VALUE

0 Wei ⌵

CONTRACT (Compiled by Remix)

exp4 - contracts/Exp4.sol ⌵

Deploy address _payer, address _payee ⌵

☐ Publish to IPFS

OR

At Address Load contract from Address

Transactions recorded 0 i >

Deployed Contracts

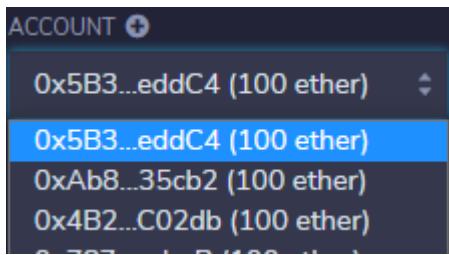
Currently you have no contract instances to interact with.

0x5B3...eddC4 (100 ether)

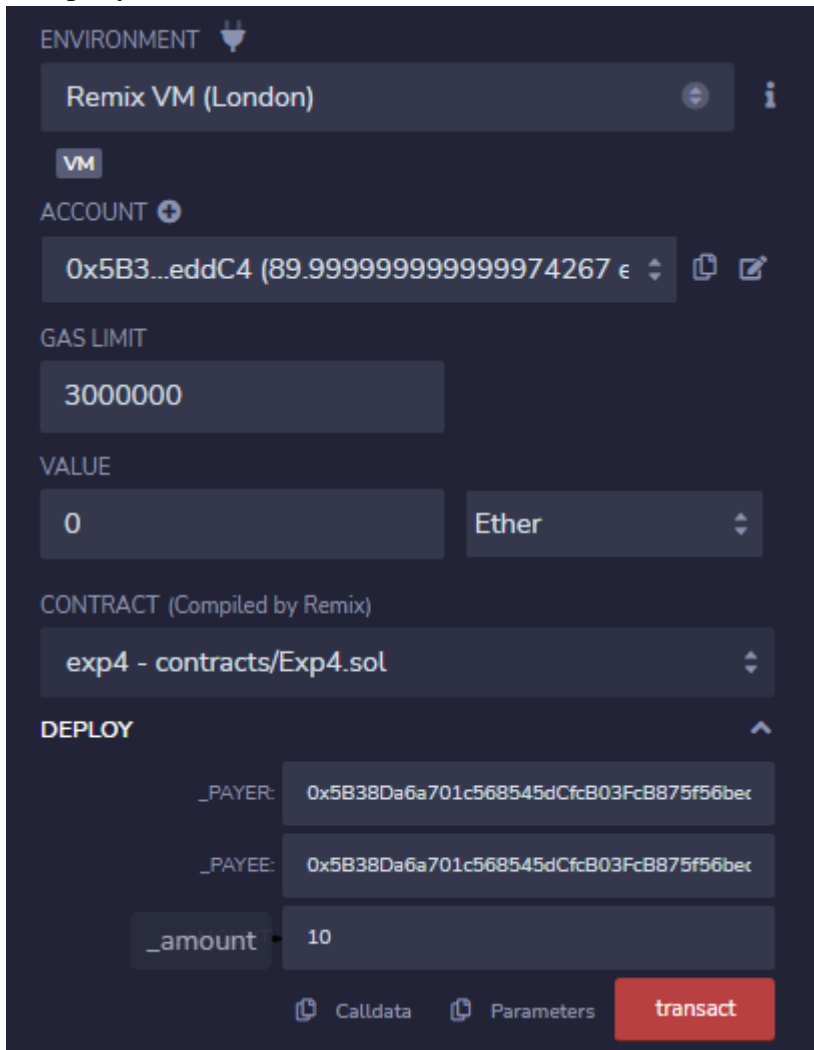
0xA8b...35cb2 (100 ether)

0x4B2...C02db (100 ether)

- Now we have a sender a receiver and an escrow account



- Now add the required data to the constructor and deploy the contract using the thirdparty escrow account.

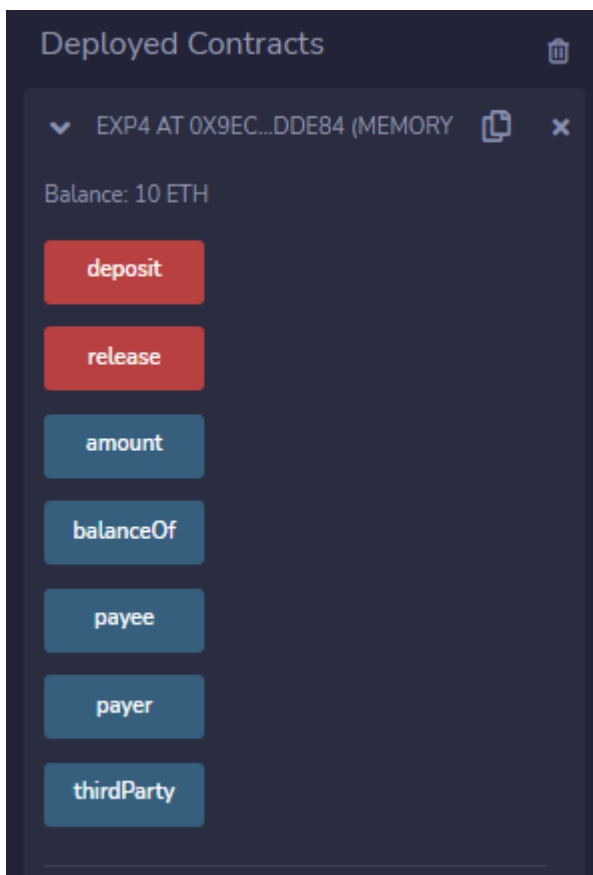


- Once the contract is deployed we get a response as such which indicates that the contract is deployed successfully

```
✓ [vm] from: 0x482...C02db to: exp4.(constructor) value: 1000000000000000000 wei data: 0x608...00005 logs: 0 hash: 0xad1...712b8

status      true Transaction mined and execution succeed
transaction hash  0xad114cd733c2a3443695390872696a111a17a6ed65a29368913038f9604712b8 ⓘ
from        0x48209938c481177ec7E8f571ceCaE8A9e22C02db ⓘ
to          exp4.(constructor) ⓘ
gas         680411 gas ⓘ
transaction cost 591661 gas ⓘ
execution cost  591661 gas ⓘ
input        0x608...00005 ⓘ
decoded input  {
  "address _payer": "0x58380a6a701c568545dcfc803fc8875f56beddc4",
  "address _payee": "0xab8483f64d9c6d1ecf9b849ae677d03315835cb2",
  "uint256 _amount": "5"
} ⓘ
decoded output - ⓘ
logs          [] ⓘ ⓘ
val          10000000000000000000 wei ⓘ
```

- This is how the contract looks after it is deployed



- Now on pressing the different values we get what they hold as values



- Now if the sender is not the payer then we get an error and the contract does not execute any function

```
[x] [vm] from: 0x4B2...C02db to: exp4.deposit() 0x9ec...dde84 value: 0 wei data: 0xd0e...30db0 logs: 0 hash: 0x789...d1718

status           false Transaction mined but execution failed
transaction hash  0x789ab0b0f8bcf12251147117b5989313a7e1b8496bc8a662116098beba8d1718
from             0x4B209938c481177ec7E8f571ceCaE8A9e22C02db
to              exp4.deposit() 0x9ecEA68DE55F3168702f27eE389010C2EE0dde84
gas             3000000 gas
transaction cost  23684 gas
execution cost   23684 gas
input           0xd0e...30db0
decoded input    {}
decoded output   {}
logs            []
val             0 wei

transact to exp4.deposit errored: VM error: revert.

revert
  The transaction has been reverted to the initial state.
  Reason provided by the contract: "Sender is not the payer".
  Debug the transaction to get more information.
```

- If the sender is the payer then the contracts executes and the ethers are then sent to the escrow from where they can be released, The contract balance decreases by the amount we want to transfer to the payee.

```
[x] [vm] from: 0x5B3...eddC4 to: exp4.deposit() 0x9ec...dde84 value: 0 wei data: 0xd0e...30db0 logs: 0 hash: 0xa19...0a73b

status           true Transaction mined and execution succeed
transaction hash  0xa19386f98fde1703337e4ff4bce63ec58590cd8a7697cc06e1141e145ce0a73b
from             0x5B380a6a701c568545dcfcB03Fc8875f56beddC4
to              exp4.deposit() 0x9ecEA68DE55F3168702f27eE389010C2EE0dde84
gas             43221 gas
transaction cost  37583 gas
execution cost   37583 gas
input           0xd0e...30db0
decoded input    {}
decoded output   {}
logs            []
val             0 wei
```



- If we try to release the funds from an account other than the thirdparty/escrow then we get an error stating that funds can be released only by the escrow account


```
✖ [vm] from: 0x5B3...eddC4 to: exp4.release() 0x9ec...dde84 value: 0 wei data: 0x86d...1a69f logs: 0 hash: 0x32c...e1524

status          false Transaction mined but execution failed
transaction hash 0x32c63025fdab39ec26f5b58666c9e65cc3c6838c664e699768f2217e347e1524 ⓘ
from            0x5B380a6a701c568545dCfc803Fc8875f56beddC4 ⓘ
to             exp4.release() 0x9ecEA680E55F3168702f27eE389D10C2EE0dde84 ⓘ
gas            3000000 gas ⓘ
transaction cost 26023 gas ⓘ
execution cost  26023 gas ⓘ
input          0x86d...1a69f ⓘ
decoded input   {} ⓘ
decoded output  {} ⓘ
logs           [] ⓘ ⓘ
val            0 wei ⓘ

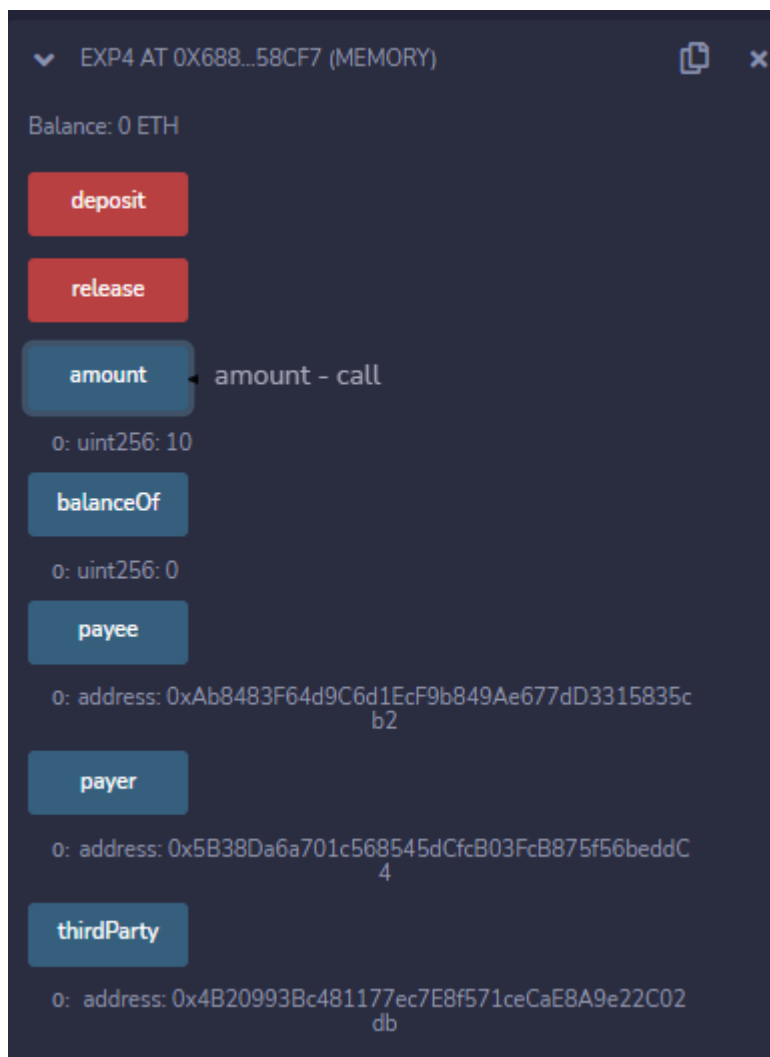
transact to exp4.release errored: VM error: revert.

revert
  The transaction has been reverted to the initial state.
  Reason provided by the contract: "Sender Not thirdParty".
  Debug the transaction to get more information.
```

- The amount can now be released from the escrow account.

```
✔ [vm] from: 0x4B2...C02db to: exp4.release() 0x9ec...dde84 value: 0 wei data: 0x86d...1a69f logs: 0 hash: 0xa16...4c670

status          true Transaction mined and execution succeed
transaction hash 0xa16ffe840b33505647b8279dd2375032cb1e03e673eaad67d667be0c4864c670 ⓘ
from            0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db ⓘ
to             exp4.release() 0x9ecEA680E55F3168702f27eE389D10C2EE0dde84 ⓘ
gas            43222 gas ⓘ
transaction cost 37584 gas ⓘ
execution cost  37584 gas ⓘ
input          0x86d...1a69f ⓘ
decoded input   {} ⓘ
decoded output  {} ⓘ
logs           [] ⓘ ⓘ
val            0 wei ⓘ
```



- After the transaction is done we can now see what are the balances of the payer, payee and the escrow.

```
0x5B3...eddC4 (89.999999999999948534 ether)
0xab8...35cb2 (110 ether)
0x4B2...C02db (99.999999999999805452 ether)
```

The amount of 10 ethers has been deducted from the sender and added to the receiver.

2. Explain your program logic, classes and methods used.

- Created a contract named Exp4 which had the following data variables
 - Public payable address payer
 - Public payable address payee
 - Public payable address thirdparty
 - Public uint256 amount
- Now we create a function named deposit which has the following functionality

- Has to be invoked by the sender
- If not invoked by sender then error pops up
- If invoked by the sender then we check if the amount to be transferred is less than or equal to the balance the contract holds
- If true then we proceed and complete the transaction from the sender to the contract balance and then to the thirdparty or escrow account.
- Now we create a release function
 - This function checks whether the invoker is the thirdparty or not as our logic states that the amount can only be released by the thirdparty
 - Once this is done we check if there is sufficient balance that the contract holds
 - If yes then we transfer the amount to the payee.

Methods used:

- 1) payable(): The payable allows someone to send ether to a contract and run code to account for this deposit. This code could potentially log an event, modify storage to record the deposit, or it could even revert the transaction if it chooses to do so. It is used for transferring amount between different accounts.
- 2) require(): Require is an error handling global function in solidity which is basically operates in the manner that if the condition within require comes out to be true then the compiler will execute the piece of code written beneath it. This function validates the users and the amount to be transferred.
- 3) transfer(): This method is generally used for transferring ethers from one account to another.

3. Explain the Importance of the approach followed by you

1. **Solidity Static Analysis:** Static code analysis — to debug the code by examining it statically without executing the code. This plugin checks for security vulnerabilities and bad development practices, among other issues, once the smart contracts are compiled. This analyzer allows the user to select the modules to be analyzed and can run the analysis for the last compiled contract by clicking on RunThe Auto run option is provided to conduct the analysis each time a contract is compiled.
2. **Etherscan Contract Verification:** A tool to verify your code with a deployed contract and to publish your code on Etherscan. To update token information on Etherscan, the token contract address for the token must be verified. This is to

ensure that the contract code is exactly what is being deployed onto the blockchain and also allows the public to audit and read the contract. Etherscan ensures that all token contracts must be verified before they can be updated with information submitted by the contract owner.

3. **Control Flow Graph:** A tool to visualize the control flow of the execution of a smart contract. It generated Control Flow Graphs (CFGs) from the bytecode of a smart contract and also highlights its execution trace.
4. **Sol2UML:** Unified Modeling Language (UML) class diagram generator for Solidity contracts. sol2uml uses @solidity-parser/parser which is maintained by the Solidity tool community lead by Franco Victorio (@fvictorio). The diagrams are generated using viz.js which uses Graphviz to render a Scalable Vector Graphics (SVG) file. Graphviz Online allows dotfiles to be edited and rendered into an SVG dynamically.
5. **DGIT — A decentralized git plugin for Remix:** This tool stores the files in Remix as a git repository in IPFS. The repository is stored in the browser but can be exported and imported from IPFS using the plugin. Each export of the repository will generate a new unique IPFS hash. The committed browser's files from Remix and the git repository itself (.git) can be found in IPFS. These hashes are stored in the browser's local storage and can be exported to Pinata Cloud. To ensure data persistence on IPFS against processes like garbage collection, data can be pinned to one or more IPFS nodes indefinitely. Pinata enables free pinning up to 1GB of data (see Pinata Cloud)

Conclusion: - Understood how a smart contract works and how it can be used to fulfil different business logics via the amount of flexibility that solidity and Ethereum smart contracts provide.