MODULE III

CHAPTER 3

Permissionless Blockchain : Ethereum

University Prescribed Syllabus w.e.f Academic Year 2022-2023

Introduction to Ethereum, Ethereum 1.0 and 2.0 , Turing completeness EVM and compare with bitcoin Basics of Ether Units, Ethereum Wallets Working with Metamask EOA and Contracts Transaction:: Structure of Transaction, Transaction Nonce, Transaction GAS, Recipient, Values and Data, Transmitting Values to EOA and Contracts

Smart Contracts and Solidity

Development environment and client , Basic of Solidity and Web 3 Life cycle of Smart contract, Smart Contract programming using solidity, Metamask (Ethereum. Wallet), Setting up development environment, Use cases of Smart Contract, Smart Contracts: Opportunities and Risk. **Smart Contract Deployment :** Introduction to Truffle, Use of Remix and test networks for deployment

Self-learning Topics: Smart contract development using Java or Python

3.1 INTRODUCTION TO ETHEREUM

- Ethereum is often described as "the world computer." From a computer science perspective, Ethereum is a deterministic but practically unbounded state machine, consisting of a globally accessible singleton state and a virtual machine that applies changes to that state.
- From a more practical perspective, Ethereum is an open source, globally decentralized computing infrastructure that executes programs called smart contracts. It uses a blockchain to synchronize and store the system's state changes, along with a cryptocurrency called ether to meter and constrain execution resource costs.
- The Ethereum platform enables developers to build powerful decentralized applications with built-in economic functions. While providing high availability, auditability, transparency, and neutrality, it also reduces or eliminates censorship and reduces certain counterparty risks.
- Ethereum is a blockchain with a computer embedded in it. It is the foundation for building apps and organizations in a decentralized, permissionless, censorship-resistant way.
- In the Ethereum universe, there is a single, canonical computer (called the Ethereum Virtual Machine, or EVM) whose state everyone on the Ethereum network agrees on. Everyone who participates in the Ethereum network (every Ethereum node) keeps a copy of the state of this computer.
- Additionally, any participant can broadcast a request for this computer to perform arbitrary computation. Whenever such a request is broadcast, other participants on the network verify, validate, and carry out ("execute") the computation. This execution causes a state change in the EVM, which is committed and propagated throughout the entire network.
- Requests for computation are called transaction requests; the record of all transactions and the EVM's present state gets stored on the blockchain, which in turn is stored and agreed upon by all nodes.

3.2 ETHEREUM 1.0 AND 2.0

3.2.1 Ethereum 1.0

- Projects that run on Ethereum are decentralized applications (DApps) that work just like the apps we all know and use every single day. However, there's one big difference: they don't have a single point of authority. By taking advantage of Ethereum's blockchain, they can store transaction history and other data immutably (without having to build their own blockchain – which is expensive).

- Dapps development is a rapidly evolving field since the rise of Ethereum, and today thousands of projects have DApps running on the network across technology, gaming, art and collectibles, and – naturally – financial services. Dapps are sometimes considered as part of Web3, the next iteration of the internet following the rise of blockchain.
- And it's all thanks to Ethereum's platform and its ability to create and issue tokens. Projects can launch their own tokens with the help of Ethereum's ERC-20 tokens standard. These tokens will be running on Ethereum's blockchain but can also be traded on the open markets together with other cryptocurrencies. Many of the top 100 crypto tokens actually run on the Ethereum network, such as Tether, Uniswap, or Chainlink.
- The popularity of Ethereum as a blockchain for developers landed it at the forefront of the decentralized finance revolution. The Ethereum network is also heavily used by stable coins – cryptocurrencies pegged to the value of a real-world currency like the US dollar. According to a report from ConsenSys, three-quarters of all stable coins now run on Ethereum, and the network handled more than $1 trillion worth of transactions in 2020.

Key Problems of Ethereum 1.0

- Ethereum has experienced its fair share of setbacks during the past few years. For example, a hack 2016 exploited the weakness in one of the projects built on top of Ethereum, resulting in $50 million worth of the coin being stolen.
- This was when the founders decided to split the blockchain in order to recover the stolen funds. That fork is known as the Ethereum Classic, and it operates until today.
- However, since Ethereum has become so popular across many projects in the crypto scene, its network is now struggling to handle all of the traffic. This problem has become so painful that the network becomes unstable at times.
- Transaction speed is an issue because the network can handle only 15 transactions per second. The fees paid to get a transaction executed can be incredibly high at times of high demand.
- In other words, Ethereum has become the victim of its own success. Since so many users were attracted to it in a short time, its popularity affected its performance.
- Naturally, the crypto scene reacted to this problem and offered alternative projects that could serve as alternatives to Ethereum – for example, Charles Hoskinson's Cardano and Polkadot developed by Gavin Wood. These developer blockchains are similar to Ethereum's design but have a much higher transaction capacity and can handle much more traffic.

3.2.2 Ethereum 2.0

The goal for Ethereum 2.0 is as follows:

- To bring Ethereum into the mainstream and serve all of humanity, we have to make Ethereum more scalable, secure, and sustainable.
- This quote from the team pretty much confirms that the network isn't in good shape today. The process of upgrading Ethereum to the 2.0 version is going to be a long one. It's divided into three stages. But before we get into that, let's understand which blockchain consensus mechanism Ethereum is planning to use.
- Proof of work blockchain comes with its problems, and to avoid inefficiency, Ethereum is now moving towards a proof of stake (PoS) blockchain. In this type of blockchain, a consensus is achieved in a much more efficient manner. The nodes that want to mine new blocks and claim the reward can stake their crypto for a chance of becoming a validator. It works more or less like a lottery. The more tickets you buy, the greater your chances of winning.
- Validators are chosen at random to mine a new block and claim reward, which is usually a cut of all the fees paid for transactions inside that block. This method of achieving consensus means that you no longer need multiple miners using lots of power to be allowed to mine a new block. But switching to the system is going to be a little more complex than you might expect.

Stage I : The Beacon Chain

The first stage of the switch went live in December 2020 and generated a lot of excitement in the Ethereum community. The Beacon Chain focuses on allowing staking to take place on Ethereum. This will allow stakers to run validator software. The mechanism ensures that the control over the network isn't concentrated among a few validator nodes.

Ethereum hopes that this type of decentralization will also address the problem of the network's security. The functionality paves the way for the next stage of ETH 2.0 that will rely on the PoS system.

Stage II : Shard Chains

To improve the scalability of Ethereum and allow the network to handle more transactions, the team plans to introduce additional changes known as shard chains. These chains will offload the main chain. Ultimately, Ethereum aims to have 64 shard chains running in parallel to increase the amount of traffic the network can handle.

Shard chains will be assigned validators by the Beacon Chain to further increase network security since no two validators will be able to collude to take over the shard. Spreading the network over shard chains will improve speed and security but also allow users to run clients from a laptop or smartphone to secure the network even more.

Sharding is going to be released sometime this year, depending on how quickly the work progresses after the launch of the Beacon Chain. Once established, the time will come for the final stage of ETH 2.0.

- **Stage III : Docking**

The Beacon Chain and shard chains will run separately from the main chain. The latter will continue using the proof of work consensus. The docking is then going to join the mainnet with the Beacon Chain and shard chains to finally move the entire Ethereum network to proof of state consensus.

3.3 TURING COMPLETENESS ETHEREUM VIRTUAL MACHINE (EVM)

- Ethereum, they say, unlike Bitcoin, is Turing complete. The term refers to English mathematician Alan Turing, who is considered the father of computer science.
- In 1936 he created a mathematical model of a computer consisting of a state machine that manipulates symbols by reading and writing them on sequential memory (resembling an infinite-length paper tape).
- With this construct, Turing went on to provide a mathematical foundation to answer (in the negative) questions about universal computability, meaning whether all problems are solvable. He proved that there are classes of problems that are uncomputable.
- Specifically, he proved that the halting problem (whether it is possible, given an arbitrary program and its input, to determine whether the program will eventually stop running) is not solvable.
- Alan Turing further defined a system to be Turing complete if it can be used to simulate any Turing machine. Such a system is called a Universal Turing Machine (UTM).
- Ethereum's ability to execute a stored program, in a state machine called the Ethereum Virtual Machine, while reading and writing data to memory makes it a Turing-complete system and therefore a UTM.
- Ethereum can compute any algorithm that can be computed by any Turing machine, given the limitations of finite memory.
- Ethereum's ground breaking innovation is to combine the general-purpose computing architecture of a stored-program computer with a decentralized blockchain, thereby creating a distributed single-state (singleton) world computer.
- Ethereum programs run "everywhere," yet produce a common state that is secured by the rules of consensus.

Turing Completeness as a "Feature"

- Hearing that Ethereum is Turing complete, you might arrive at the conclusion that this is a feature that is somehow lacking in a system that is Turing incomplete. Rather, it is the opposite.
- Turing completeness is very easy to achieve; in fact, the simplest Turing-complete state machine known has 4 states and uses 6 symbols, with a state definition that is only 22 instructions long. Indeed, sometimes systems are found to be "accidentally Turing complete."
- However, Turing completeness is very dangerous, particularly in open access systems like public blockchains, because of the halting problem we touched on earlier. For example, modern printers are Turing complete and can be given files to print that send them into a frozen state.
- The fact that Ethereum is Turing complete means that any program of any complexity can be computed by Ethereum. But that flexibility brings some thorny security and resource management problems. An unresponsive printer can be turned off and turned back on again. That is not possible with a public blockchain.

Implications of Turing Completeness

- Turing proved that you cannot predict whether a program will terminate by simulating it on a computer. In simple terms, we cannot predict the path of a program without running it.
- Turing-complete systems can run in "infinite loops," a term used (in oversimplification) to describe a program that does not terminate. It is trivial to create a program that runs a loop that never ends. But unintended never-ending loops can arise without warning, due to complex interactions between the starting conditions and the code.
- In Ethereum, this poses a challenge: every participating node (client) must validate every transaction, running any smart contracts it calls. But as Turing proved, Ethereum can't predict if a smart contract will terminate, or how long it will run, without actually running it (possibly running forever).
- Whether by accident or on purpose, a smart contract can be created such that it runs forever when a node attempts to validate it. This is effectively a DoS attack. And of course, between a program that takes a millisecond to validate and one that runs forever are an infinite range of nasty, resource-hogging, memory-bloating, CPU-overheating programs that simply waste resources. In a world computer, a program that abuses resources gets to abuse the world's resources.

- Ethereum introduces a metering mechanism called gas. As the EVM executes a smart contract, it carefully accounts for every instruction (computation, data access, etc.). Each instruction has a predetermined cost in units of gas.
- When a transaction triggers the execution of a smart contract, it must include an amount of gas that sets the upper limit of what can be consumed running the smart contract.
- The EVM will terminate execution if the amount of gas consumed by computation exceeds the gas available in the transaction. Gas is the mechanism Ethereum uses to allow Turing-complete computation while limiting the resources that any program can consume.
- To pay for computation on the Ethereum world computer. You won't find gas on any exchanges. It can only be purchased as part of a transaction, and can only be bought with ether.
- Ether needs to be sent along with a transaction and it needs to be explicitly earmarked for the purchase of gas, along with an acceptable gas price. Just like at the pump, the price of gas is not fixed. Gas is purchased for the transaction, the computation is executed, and any unused gas is refunded back to the sender of the transaction.

3.4 COMPARISON WITH BITCOIN

- Many people will come to Ethereum with some prior experience of cryptocurrencies, specifically Bitcoin. Ethereum shares many common elements with other open blockchains: a peer-to-peer network connecting participants, a Byzantine fault–tolerant consensus algorithm for synchronization of state updates (a proof-of-work blockchain), the use of cryptographic primitives such as digital signatures and hashes, and a digital currency (ether).
- Yet in many ways, both the purpose and construction of Ethereum are strikingly different from those of the open blockchains that preceded it, including Bitcoin.
- Ethereum's purpose is not primarily to be a digital currency payment network. While the digital currency ether is both integral to and necessary for the operation of Ethereum, ether is intended as a utility currency to pay for use of the Ethereum platform as the world computer.
- Unlike Bitcoin, which has a very limited scripting language, Ethereum is designed to be a general-purpose programmable blockchain that runs a virtual machine capable of executing code of arbitrary and unbounded complexity. Where Bitcoin's Script language is, intentionally, constrained to simple true/false evaluation of spending conditions, Ethereum's language is Turing complete, meaning that Ethereum can straightforwardly function as a general-purpose computer.

3.5 ETHEREUM WALLETS WORKING WITH METAMASK EXTERNALLY OWNED ACCOUNTS (EOA) AND CONTRACTS TRANSACTION : STRUCTURE OF TRANSACTION

3.5.1 Ethereum Wallets

There are only two types of ways to manage funds on Ethereum: externally owned accounts (EOAs) and contract accounts. These both allow Ethereum users to store ETH, ERC20s and NFTs while also enabling interaction with the Ethereum network.

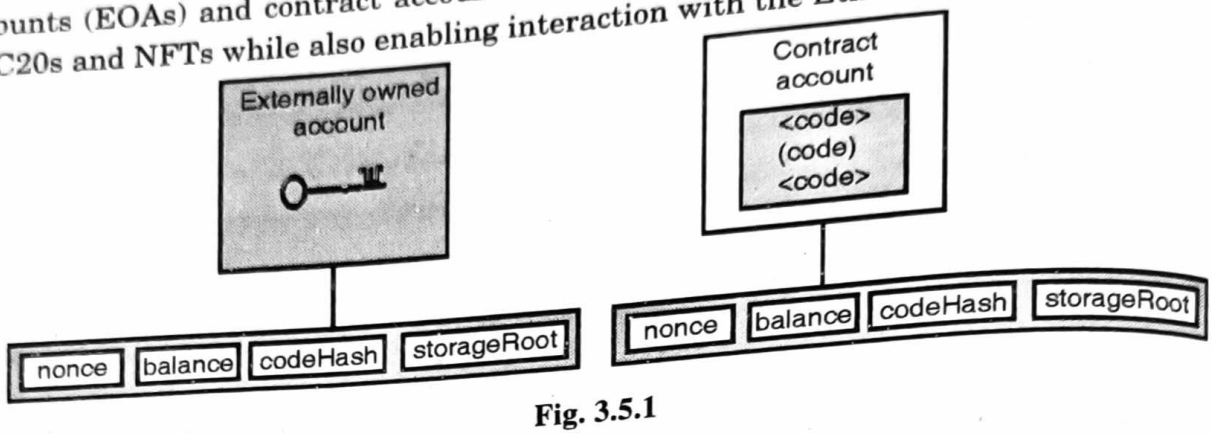


Fig. 3.5.1

3.5.2 Externally Owned Accounts (EOAs)

- An account is made up of a cryptographic pair of keys: public and private. They help prove that a transaction was actually signed by the sender and prevent forgeries. Your private key is what you use to sign transactions, so it grants you custody over the funds associated with your account. You never really hold cryptocurrency, you hold private keys – the funds are always on Ethereum's ledger.
- They can be lumped into the following categories:
  - EOA software wallets
  - EOA hardware wallets

EOA software wallets

- To date, most accounts created on the Ethereum network fall under the category of externally owned accounts. EOAs are free to generate and after creation the user is given a public key (0x...) that they can use to send and receive funds. To recover an EOA, users are given a private key normally in the form of a 12 word seed phrase.
- Users must make sure this seed phrase is backed up or not comprised, as it has complete control over the account. Many wallets that deal with EOAs allow users to create and manage as many accounts as they'd like.

EOA hardware wallets

- Hardware wallets are simply externally owned accounts whose private keys are never exposed to the internet. When a user gets a hardware wallet and generates an account, the keys are generated on device behind a secure enclave. When the user wants to make a transaction, they sign that transaction on the device and it is then broadcast to the network. The private keys are never exposed in the process. While this is a great solution for cold storage, hardware wallets are not as portable or easily accessible as an EOA in your browser.
- MetaMask is a free crypto wallet software that can be connected to virtually any Ethereum-based platform. MetaMask is by far the most popular with over 21 million monthly active users – up by 38x since 2020. To put it simply, MetaMask is a free hot wallet service available as a smartphone app or web browser extension. This means you can download it directly onto your phone or the Google Chrome, Mozilla Firefox, Brave or Edge browser similar to an ad blocker extension.
- The "hot" part simply means it's permanently connected to the internet so that you can easily move your crypto assets around at any time. One of the key reasons MetaMask is so popular among new and existing crypto users is its interoperability with virtually all Ethereum-based platforms. MetaMask allows users to connect to more than 3,700 different decentralized applications and Web 3 services.

3.5.3 Contracts Transaction : Structure of Transaction

- Transactions are signed messages originated by an externally owned account, transmitted by the Ethereum network, and recorded on the Ethereum blockchain. This basic definition conceals a lot of surprising and fascinating details. Another way to look at transactions is that they are the only things that can trigger a change of state, or cause a contract to execute in the EVM.
- Ethereum is a global singleton state machine, and transactions are what make that state machine "tick," changing its state. Contracts don't run on their own. Ethereum doesn't run autonomously. Everything starts with a transaction

3.6 STRUCTURE OF A TRANSACTION

- First let's take a look at the basic structure of a transaction, as it is serialized and transmitted on the Ethereum network. Each client and application that receives a serialized transaction will store it in-memory using its own internal data structure, perhaps embellished with metadata that doesn't exist in the network serialized transaction itself. The network-serialization is the only standard form of a transaction.

- A transaction is a serialized binary message that contains the following data:
  - **Nonce** : A sequence number, issued by the originating EOA, used to prevent message replay
  - **Gas price** : The price of gas (in wei) the originator is willing to pay
  - **Gas limit** : The maximum amount of gas the originator is willing to buy for this transaction
  - **Recipient** : The destination Ethereum address
  - **Value** : The amount of ether to send to the destination
  - **Data** : The variable-length binary data payload
  - **v, r, s** : The three components of an ECDSA digital signature of the originating EOA
- The transaction message's structure is serialized using the Recursive Length Prefix (RLP) encoding scheme, which was created specifically for simple, byte-perfect data serialization in Ethereum. All numbers in Ethereum are encoded as big-endian integers, of lengths that are multiples of 8 bits.
- the field labels (to, gas limit, etc.) are shown here for clarity, but are not part of the transaction serialized data, which contains the field values RLP-encoded. In general, RLP does not contain any field delimiters or labels. RLP's length prefix is used to identify the length of each field. Anything beyond the defined length belongs to the next field in the structure.
- While this is the actual transaction structure transmitted, most internal representations and user interface visualizations embellish this with additional information, derived from the transaction or from the blockchain.
- For example, you may notice there is no "from" data in the address identifying the originator EOA. That is because the EOA's public key can be derived from the v, r, s components of the ECDSA signature. The address can, in turn, be derived from the public key. When you see a transaction showing a "from" field, that was added by the software used to visualize the transaction. Other metadata frequently added to the transaction by client software includes the block number (once it is mined and included in the blockchain) and a transaction ID (calculated hash). Again, this data is derived from the transaction, and does not form part of the transaction message itself.

3.7 TRANSACTION NONCE

- The nonce is one of the most important and least understood components of a transaction.
- the nonce is an attribute of the originating address; that is, it only has meaning in the context of the sending address. However, the nonce is not stored explicitly as part of an account's state on the blockchain. Instead, it is calculated dynamically, by counting the number of confirmed transactions that have originated from an address.

- There are two scenarios where the existence of a transaction-counting nonce is important: the usability feature of transactions being included in the order of creation, and the vital feature of transaction duplication protection. The examples are
- Imagine you wish to make two transactions. You have an important payment to make of 6 ether, and also another payment of 8 ether. You sign and broadcast the 6-ether transaction first, because it is the more important one, and then you sign and broadcast the second, 8-ether transaction. Sadly, you have overlooked the fact that your account contains only 10 ether, so the network can't accept both transactions: one of them will fail.
- Because you sent the more important 6-ether one first, you understandably expect that one to go through and the 8-ether one to be rejected. However, in a decentralized system like Ethereum, nodes may receive the transactions in either order; there is no guarantee that a particular node will have one transaction propagated to it before the other. As such, it will almost certainly be the case that some nodes receive the 6-ether transaction first and others receive the 8-ether transaction first.
- Without the nonce, it would be random as to which one gets accepted and which rejected. However, with the nonce included, the first transaction you sent will have a nonce of, let's say, 3, while the 8-ether transaction has the next nonce value (i.e., 4). So, that transaction will be ignored until the transactions with nonces from 0 to 3 have been processed, even if it is received first. Phew!
- Now imagine you have an account with 100 ether. Fantastic! You find someone online who will accept payment in ether for a mcguffin-widget that you really want to buy. You send them 2 ether and they send you the mcguffin-widget. Lovely. To make that 2-ether payment, you signed a transaction sending 2 ether from your account to their account, and then broadcast it to the Ethereum network to be verified and included on the blockchain.
- Now, without a nonce value in the transaction, a second transaction sending 2 ether to the same address a second time will look exactly the same as the first transaction. This means that anyone who sees your transaction on the Ethereum network (which means everyone, including the recipient or your enemies) can "replay" the transaction again and again and again until all your ether is gone simply by copying and pasting your original transaction and resending it to the network.
- However, with the nonce value included in the transaction data, *every single transaction is unique*, even when sending the same amount of ether to the same recipient address multiple times. Thus, by having the incrementing nonce as part of the transaction, it is simply not possible for anyone to "duplicate" a payment you have made.
- It is important to note that the use of the nonce is actually vital for an *account-based* protocol, in contrast to the "Unspent Transaction Output" (UTXO) mechanism of the Bitcoin protocol.

3.8 TRANSACTION GAS

- Gas is the fuel of Ethereum. Gas is not ether—it's a separate virtual currency with its own exchange rate against ether. Ethereum uses gas to control the amount of resources that a transaction can use, since it will be processed on thousands of computers around the world. The open-ended (Turing-complete) computation model requires some form of metering in order to avoid denial-of-service attacks or inadvertently resource-devouring transactions.
- Gas is separate from ether in order to protect the system from the volatility that might arise along with rapid changes in the value of ether, and also as a way to manage the important and sensitive ratios between the costs of the various resources that gas pays for (namely, computation, memory, and storage).
- The gasPrice field in a transaction allows the transaction originator to set the price they are willing to pay in exchange for gas. The price is measured in wei per gas unit. For example, in the sample transaction in your wallet set the gasPrice to 3 gwei (3 gigawei or 3 billion wei).
- Wallets can adjust the gasPrice in transactions they originate to achieve faster confirmation of transactions. The higher the gasPrice, the faster the transaction is likely to be confirmed.
- Conversely, lower-priority transactions can carry a reduced price, resulting in slower confirmation. The minimum value that gasPrice can be set to is zero, which means a fee-free transaction. During periods of low demand for space in a block, such transactions might very well get mined.
- The second important field related to gas is gasLimit. In simple terms, gasLimit gives the maximum number of units of gas the transaction originator is willing to buy in order to complete the transaction. For simple payments, meaning transactions that transfer ether from one EOA to another EOA, the gas amount needed is fixed at 21,000 gas units.

3.9 TRANSACTION RECIPIENT

- The recipient of a transaction is specified in the to field. This contains a 20-byte Ethereum address. The address can be an EOA or a contract address.
- Ethereum does no further validation of this field. Any 20-byte value is considered valid.
- If the 20-byte value corresponds to an address without a corresponding private key, or without a corresponding contract, the transaction is still valid. Ethereum has no way of knowing whether an address was correctly derived from a public key (and therefore from a private key) in existence.

- Sending a transaction to the wrong address will probably *burn* the ether sent, rendering it forever inaccessible (unspendable), since most addresses do not have a known private key and therefore no signature can be generated to spend it. It is assumed that validation of the address happens at the user interface level.
- In fact, there are a number of valid reasons for burning ether - for example, as a disincentive to cheating in payment channels and other smart contracts - and since the amount of ether is finite, burning ether effectively distributes the value burned to all ether holders (in proportion to the amount of ether they hold).

3.10 TRANSACTION VALUES AND DATA

- The main "payload" of a transaction is contained in two fields: value and data. Transactions can have both value and data, only value, only data, or neither value nor data. All four combinations are valid.
- A transaction with only value is a *payment*. A transaction with only data is an *invocation*. A transaction with both value and data is both a payment and an invocation. A transaction with neither value nor data well that's probably just a waste of gas! But it is still possible.

3.11 TRANSMITTING VALUES TO EXTERNALLY OWNED ACCOUNTS AND CONTRACTS

- When you construct an Ethereum transaction that contains a value, it is the equivalent of a *payment*. Such transactions behave differently depending on whether the destination address is a contract or not.
- For EOA addresses, or rather for any address that isn't flagged as a contract on the blockchain, Ethereum will record a state change, adding the value you sent to the balance of the address. If the address has not been seen before, it will be added to the client's internal representation of the state and its balance initialized to the value of your payment.
- If the destination address (to) is a contract, then the EVM will execute the contract and will attempt to call the function named in the data payload of your transaction. If there is no data in your transaction, the EVM will call a *fallback* function and, if that function is payable, will execute it to determine what to do next. If there is no fallback function, then the effect of the transaction will be to increase the balance of the contract, exactly like a payment to a wallet.
- A contract can reject incoming payments by throwing an exception immediately when a function is called, or as determined by conditions coded in a function. If the function terminates successfully (without an exception), then the contract's state is updated to reflect an increase in the contract's ether balance.

3.12 SMART CONTRACTS AND SOLIDITY

- The term smart contract has been used over the years to describe a wide variety of different things. In the 1990s, cryptographer Nick Szabo coined the term and defined it as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises."
- Since then, the concept of smart contracts has evolved, especially after the introduction of decentralized blockchain platforms with the invention of Bitcoin in 2009. In the context of Ethereum, the term is actually a bit of a misnomer, given that Ethereum smart contracts are neither smart nor legal contracts, but the term has stuck. In this book, we use the term "smart contracts" to refer to immutable computer programs that run deterministically in the context of an Ethereum Virtual Machine as part of the Ethereum network protocol i.e., on the decentralized Ethereum world computer.

3.12.1 Development environment and client

- To develop in Solidity, you can use any text editor and solc on the command line. However, you might find that some text editors designed for development, such as Emacs, Vim, and Atom, offer additional features such as syntax highlighting and macros that make Solidity development easier. There are also web-based development environments, such as Remix IDE and EthFiddle.
- Use the tools that make you productive. In the end, Solidity programs are just plain text files. While fancy editors and development environments can make things easier, you don't need anything more than a simple text editor, such as nano (Linux/Unix), TextEdit (macOS), or even Notepad (Windows). Simply save your program source code with a .sol extension and it will be recognized by the Solidity compiler as a Solidity program.

Writing a Simple Solidity Program

We wrote our first Solidity program. When we first built the Faucet contract, we used the Remix IDE to compile and deploy the contract. In this section, we will revisit, improve, and embellish Faucet.

Example 1. Faucet.sol: A Solidity contract implementing a faucet

```
link:code/Solidity/Faucet.sol[]
```

Compiling with the Solidity Compiler (solc)

Now, we will use the Solidity compiler on the command line to compile our contract directly. The Solidity compiler solc offers a variety of options, which you can see by passing the --help argument.

We use the --bin and --optimize arguments of solc to produce an optimized binary of our example contract:

```
$ solc --optimize --bin Faucet.sol
======= Faucet.sol:Faucet =======
Binary
6060604052341561000f57600080fd5b60cf8061001d6000396000f300606060405260043610603e5
763ffffffff7c010000000000000000000000000000000000000000000000000000006000350416
632e1a7d4d81146040575b005b3415604a57600080fd5b603e60043567016345785d8a00008111156
0835760008080fd5b73ffffffffffffffffffffffffffffffffffffffff331681156108fc0282604051
6000604051808303818588881935050505015156a057600080fd5b505600a165627a7a723058203
556d79355f2da19e773a9551e95f1ca7457f2b5fbbf4eacf7748ab59d2532130029
```

The result that solc produces is a hex-serialized binary that can be submitted to the Ethereum blockchain.

3.12.2 Basics of Solidity

- Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state. Solidity is a curly-bracket language designed to target the Ethereum Virtual Machine (EVM). It is influenced by C++, Python and JavaScript.
- Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features. With Solidity can create contracts for uses such as voting, crowdfunding, blind auctions, and multi-signature wallets.

3.12.3 Lifecycle of a Smart Contact

- Smart contracts are typically written in a high-level language, such as Solidity. But in order to run, they must be compiled to the low-level bytecode that runs in the EVM.
- Once compiled, they are deployed on the Ethereum platform using a special *contract creation* transaction, which is identified as such by being sent to the special contract creation address, namely 0x0 .
- Each contract is identified by an Ethereum address, which is derived from the contract creation transaction as a function of the originating account and nonce. The Ethereum address of a contract can be used in a transaction as the recipient, sending funds to the contract or calling one of the contract's functions.
- There are no keys associated with an account created for a new smart contract. As the contract creator, you don't get any special privileges at the protocol level (although you can explicitly code them into the smart contract). You certainly don't receive the private key for the contract account, which in fact does not exist we can say that smart contract accounts own themselves.

- Importantly, contracts *only run if they are called by a transaction*. All smart contracts in Ethereum are executed, ultimately, because of a transaction initiated from an EOA. A contract can call another contract that can call another contract, and so on, but the first contract in such a chain of execution will always have been called by a transaction from an EOA.
- Contracts never run "on their own" or "in the background." Contracts effectively lie dormant until a transaction triggers execution, either directly or indirectly as part of a chain of contract calls. It is also worth noting that smart contracts are not executed "in parallel" in any sense the Ethereum world computer can be considered to be a single-threaded machine.
- Transactions are *atomic*, regardless of how many contracts they call or what those contracts do when called. Transactions execute in their entirety, with any changes in the global state (contracts, accounts, etc.) recorded only if all execution terminates successfully.
- Successful termination means that the program executed without an error and reached the end of execution. If execution fails due to an error, all of its effects (changes in state) are "rolled back" as if the transaction never ran. A failed transaction is still recorded as having been attempted, and the ether spent on gas for the execution is deducted from the originating account, but it otherwise has no other effects on contract or account state.
- it is important to remember that a contract's code cannot be changed. However, a contract can be "deleted," removing the code and its internal state (storage) from its address, leaving a blank account.
- Any transactions sent to that account address after the contract has been deleted do not result in any code execution, because there is no longer any code there to execute. To delete a contract, you execute an EVM opcode called SELFDESTRUCT. That operation costs "negative gas," a gas refund, thereby incentivizing the release of network client resources from the deletion of stored state.
- Deleting a contract in this way does not remove the transaction history (past) of the contract, since the blockchain itself is immutable. It is also important to note that the SELFDESTRUCT capability will only be available if the contract author programmed the smart contract to have that functionality. If the contract's code does not have a SELFDESTRUCT opcode, or it is inaccessible, the smart contract cannot be deleted.

3.12.4 Smart Contract Programming using Solidity

- The EVM is a virtual machine that runs a special form of code called EVM bytecode, analogous to your computer's CPU, which runs machine code such as x86_64. While any high-level language could be adapted to write smart contracts, adapting an arbitrary language to be compilable to EVM bytecode is quite a cumbersome exercise

and would in general lead to some amount of confusion. Smart contracts operate in a highly constrained and minimalistic execution environment (the EVM).

- In addition, a special set of EVM-specific system variables and functions needs to be available. As such, it is easier to build a smart contract language from scratch than it is to make a general-purpose language suitable for writing smart contracts. As a result, a number of special-purpose languages have emerged for programming smart contracts.
- Ethereum has several such languages, together with the compilers needed to produce EVM-executable bytecode. In general, programming languages can be classified into two broad programming paradigms: *declarative* and *imperative*, also known as *functional* and *procedural*, respectively. In declarative programming, we write functions that express the *logic* of a program, but not its *flow*.

Data Types

First, let's look at some of the basic data types offered in Solidity:

- **Boolean (bool)**

  Boolean value, true or false, with logical operators ! (not), && (and), || (or), == (equal), and != (not equal).

- **Integer (int, uint)**

  Signed (int) and unsigned (uint) integers, declared in increments of 8 bits from int8 to uint256. Without a size suffix, 256-bit quantities are used, to match the word size of the EVM.

- **Fixed point (fixed, ufixed)**

  Fixed-point numbers, declared with (u)fixedMxN where M is the size in bits (increments of 8 up to 256) and N is the number of decimals after the point (up to 18); e.g., ufixed32x2.

- **Address**

  A 20-byte Ethereum address. The address object has many helpful member functions, the main ones being balance (returns the account balance) and transfer (transfers ether to the account).

- **Byte array (fixed)**

  Fixed-size arrays of bytes, declared with bytes1 up to bytes32.

- **Byte array (dynamic)**

  Variable-sized arrays of bytes, declared with bytes or string.

- **Enum**

  User-defined type for enumerating discrete values:

  enum NAME {LABEL1, LABEL 2, ...}.

- **Arrays**

  An array of any type, either fixed or dynamic: uint32[][5] is a fixed-size array of five dynamic arrays of unsigned integers.

- **Struct**

  User-defined data containers for grouping variables: struct NAME {TYPE1 VARIABLE1; TYPE2 VARIABLE2; ...}.

- **Mapping**

  Hash lookup tables for *key* => *value* pairs: mapping(KEY_TYPE => VALUE_TYPE) NAME.

  In addition to these data types, Solidity also offers a variety of value literals that can be used to calculate different units:

- **Time units**

  The units seconds, minutes, hours, and days can be used as suffixes, converting to multiples of the base unit seconds.

- **Ether units**
  - The units wei, finney, szabo, and ether can be used as suffixes, converting to multiples of the base unit wei.
  - In our Faucet contract example, we used a uint (which is an alias for uint256) for the withdraw_amount variable. We also indirectly used an address variable, which we set with msg.sender.
  - Let's use one of the unit multipliers to improve the readability of our example contract. In the withdraw function we limit the maximum withdrawal, expressing the limit in wei, the base unit of ether:
  - require(withdraw_amount <= 100000000000000000);
  - That's not very easy to read. We can improve our code by using the unit multiplier ether, to express the value in ether instead of wei:

    require(withdraw_amount <= 0.1 ether);

3.12.5 MetaMask (Ethereum Wallet)

- The decentralized internet, Web3, is built on a foundation of cryptocurrencies and decentralized applications (DApps). But in order to use them, you need a user interface. Ideally, an elegant, intuitive, easy to use interface. MetaMask is one of the leading crypto wallets, and relies on browser integration and good design to serve as one of the main gateways to the world of Web3, decentralized finance (DeFi) and NFTs.

- MetaMask is a browser plugin that serves as an Ethereum wallet, and is installed like any other browser plugin. Once it's installed, it allows users to store Ether and other ERC-20 tokens, enabling them to transact with any Ethereum address.
- By connecting to MetaMask to Ethereum-based DApps, users can spend their coins in games, stake tokens in gambling applications, and trade them on decentralized exchanges (DEXs). It also provides users with an entry point into the emerging world of decentralized finance, or DeFi, providing a way to access DeFi apps such as Compound and PoolTogether.

3.12.6 Use Cases of Smart Contract

Smart contract use case can vary from sector to sector based on where companies are using them.

- **Digital Identity**

  One of the most obvious smart contract use cases is Digital Identity. Individual identity is one of the biggest assets for that individual. It contains reputation, data, and digital assets. The digital identity, if used rightly, can bring new opportunities to the person. Also, digital identity can also help protect the identity from counterparties and enable him to share it with companies that he intends.

  For now, the internet allows you to connect to multiple services, but at the same time, unknowingly sharing your identity with the companies and they're associated with having your identity mapped.

  In this case, smart contracts can help counterparties learn about the individual without knowing their true identity or verify transactions. This frictionless KYC can help improve interoperability, resilience, and compliance — all with the use of smart contracts.

- **High Securities**

  Another one of the useful smart contract real use cases include securities. With smart contracts, capitalization table management can be simplified and improved. This means that there are no intermediaries between the parties, including security custody chains.

  It can also be used for dividends, automatic payments, liability management, and stock splits. Also, smart contracts can help reduce operational risk and make workflows digitized.

- **Cross-border Payments**

  Trade Finance can also be revolutionized with the help of smart contracts. There is no doubt that it can help in international goods transfer and trade payment initiations with the use of a Letter of Credit.

Clearly, using smart contracts will improve the liquidity of the financial assets, in return, improving the suppliers, buyers, and institutions' financial efficiencies.

To make smart contracts work in Trade finance, especially in cross-border payments and international trade, it is necessary to find an industry standard and implement it accordingly.

With proper integration, it can surely solve legal complications and offers a better way to solve disputes among parties.

- **Loans and Mortgages**

  Smart contracts can also help improve financial services, including mortgages and loans. To do so, it can connect the parties and ensure that the whole process can be completed in a friction-less way. Moreover, it also provides an error-free process. For instance, the smart contract set up to handle a mortgage can manage it by tracking the payments and releasing the property when the whole loan is paid off.

  One more benefit from using smart contracts in financial services is visibility to all the involved parties.

- **Financial Data Recording**

  Financial data is very important for any organization. And, this is where the smart contracts come in. They provide the necessary way to data records for a more accurate and transparent financial data collection. With smart contracts, it is easy to manage the uniform recording of data across an organization resulting in reduced auditing costs and reporting.

  Finally, it also results in reduced accounting costs and better interoperability among legacy networks and distributed ledger networks.

- **Government**

  Smart contracts help automate. That's where it can help the government to manage operations. One of those operations includes land title recording where the government can use to do property transfers.

  Land Title Recording requires parties to transfer property with efficiency and transparency. Smart contracts can help do so. Also, using it will reduce auditing costs and also improve transparency within the whole system.

  Another use-case for government, including electronic elections, the digital identity that we discussed earlier, and electronic record filing.

- **Supply Chain Management**

  Supply chain management is a great, blockchain smart contract use case. By using smart contracts, the supply chain can be improved manifold. For example, it can be used to track items within the supply chain with full visibility and transparency. A business can use smart-contract-powered supply chains and improve its inventory tracking to a granular level.

It also improves other aspects of the business, which are directly connected with the supply chain. Moreover, using smart contracts also means a reduction in verification, and enhanced tracing results in fewer frauds and thefts. However, to make it work, the institutions need to add additional equipment, including sensors, to their supply chain. More so, it's a smart contract application example.

- **Insurance**

Insurance has always been one of the most use-cases of smart contracts. It is a known fact that most of the disputes happen in the insurance sector. For example, let's take auto insurance as an example. Here, smart contracts can be used to settle the insurance as soon as possible.

To do so, smart contracts need to utilize a lot of technology, including Internet-of-Things, to facilitate themselves. The smart contract will facilitate the policy and make sure that it has all the proper documentation, including driver reports and driving records, with the use of the technology. If the smart contract is set up with the right policy, documents, and ways to capture data, it can execute itself shortly after the accident. Also, smart contract execution is only done based on the collected data, which ensures that no fraud is done in the process.

- **Clinical Trials**

Clinical trials can also improve with smart contracts as they can improve cross-institutional visibility. It can also automate the data sharing between institutions, thanks to the automation and privacy-preserving computations it can do. This is one of the real world examples of smart contracts. Moreover, smart contracts can also be used to automate the trials and share the information across-industry. To be precise, it can help in identity, authorization, and authentication of the data.

- **Escrow**

The last use-case that we are going to discuss is escrow. Escrows are the process of storing value between the parties when the contract is still active. For this, the action is taken by the payer to release the funds. However, in the case of smart contract usage, it will be possible to automate the whole thing as soon as the service provider submits its work and authenticates it. The smart contracts can be very useful for platforms such as Upwork or other freelancing platforms where the platform's escrow amount is held. There are many companies using smart contracts for this purpose.

- **Record Storage**

Smart contract database can be used to record information and also do digitization of real-world assets. You can use a smart contract database to store the records and renew them and release them according to the set parameters. All of these can be done automatically. This is one of the real world examples of smart contracts.

- **Trading Activities**

  Another use case of smart contracts in trade finance is trading activities. In this case, the middleman or broker is removed, and his work is automated with the smart contract. This removes the additional cost related to them. There are many companies using smart contracts for this purpose.

- **Mortgage System**

  Smart contracts can be effectively used in the mortgage system. It enables mortgages to be automated and ease both the owner and the buyer. To make all of these happen, smart contracts need to be coded according to the mortgage agreement. Once done, the smart contracts can be set into motion, and each step in the process can be automatically executed. The whole process is fast, cheap, and easy. This is one of the real world examples of smart contracts.

3.12.7 Smart Contracts : Opportunities and Risk

3.12.7(A) Opportunities

1. Government Voting System

- Smart contracts provide a secure environment by making the voting system less susceptible to manipulation. Votes using smart contracts would be protected by the ledger, which is extremely difficult to crack.
- Additionally, smart contracts could increase voter turnover, which is historically low due to the inefficient system that forces voters to line up, show their IDs and fill out forms. If transferred online via smart contracts, voting can increase the number of participants in a voting system.

2. Healthcare

- The smart contract can also be helpful for the healthcare industry. It can keep the patient's records safe and secure with private keys and only the person with access. Apart from that, one can research using the data but confidentiality.
- Moreover, the smart contract will also help provide proof of service to the insurance companies & even the same ledger can be useful in managing the medical supplies of the patients and supervising drugs.

3. Supply Chain

- Usually, the supply chains have to go through the suffering of a paper works system where each work has to pass through multiple pipelines. The paperwork process can be the reason for human errors and increases the risk of fraud or loss of documents. Blockchain has 100% capability to reduce the risk of fraud or any human error.

- Most industries adopted the blockchain industry because of its feature of ultimate protection for each crucial document and can only be accessible by those who have the allowance.
- Smart contracts can be used for inventory management and automating payments and tasks.

4. Financial Services

- Smart contracts allow for changing traditional financial services in numerous ways. The insurance claims section executes error checking, routing, and transfer costs to the user if everything is found appropriate and suitable as per the norms.
- Smart contracts contain crucial tools for bookkeeping and exclude the probability of infiltration of accounting journals. It also allows shareholders to participate in decision-making transparently and more actively. Also, they assist in business clearing, where stores are exchanged once the sums of exchange settlements are calculated.
- A well-publicized advantage of smart contracts is that payments can be automated without the need for reminders or other collection costs and without going to court to get a payment decision.
- This is true for simple use cases, but it can be less accurate for complex business relationships. In reality, the parties are constantly shifting funds around their business rather than "holding" the total amount to be paid for long-term contracts in anticipation of future payment requirements.
- Similarly, the person receiving the loan rarely holds the full amount of the loan in a particular wallet linked to a smart contract.
- Rather, the borrower uses these funds to fund the required repayments. If the party is encouraged by the Smart Convention, the Wallet is not in time an intelligent contract is to transfer from this wallet to this wallet.
- The problem will not be solved by implementing another layer in the process and withdrawing from other wallets or with a smart contract to draw "self" from other wallets.

3.12.7(B) Risks

- **Security**

For any blockchain software development company, the major concern or challenge is to make the deal or exchange safe from hackers. Blockchain technology is safe from such threats, but still, hackers try to break the walls and always try new tricks. Technology is improving every day, so security is also getting advanced.

- **Integrity**

  An oracle (a gushing information source that sends occasion overhauls) must watch against occasion spoofing programmers that trigger savvy contracts to execute when they shouldn't. It must be modified to survey occasions precisely, which can be troublesome for complex scenarios.

- **Alignment**

  Smart contracts can speed up forms, including numerous parties. But this capability can increase the harm when times get out of control, particularly when there's no way to halt or rectify unintended behavior. Gartner is famous for making versatility and reasonability issues for keen contracts that must be completely addressed.

- **Management**

  Smart contracts are complex to actualize and oversee. They are regularly set up to be troublesome or inconceivable to alter. While this may be considered a security advantage, parties can't change a savvy contract assertion or incorporate unused points of interest without drawing up a modern contract.

3.13 SMART CONTRACT DEPLOYMENT

3.13.1 Introduction to Truffle

A world class development environment, testing framework and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM), aiming to make life as a developer easier. With Truffle, you get:

- Built-in smart contract compilation, linking, deployment and binary management.
- Advanced debugging with breakpoints, variable analysis, and step functionality.
- Deployments and transactions through MetaMask to protect your mnemonic.
- External script runner that executes scripts within a Truffle environment.
- Interactive console for direct contract communication.
- Automated contract testing for rapid development.
- Scriptable, extensible deployment & migrations framework.
- Network management for deploying to any number of public & private networks.
- Package management with NPM, using the ERC190 standard.
- Configurable build pipeline with support for tight integration.

Truffle is a world-class development environment, testing framework, and asset pipeline for blockchains using the Ethereum Virtual Machine (EVM). By creating a Truffle project and editing a few configuration settings you can easily deploy your project on Celo.

To deploy on Celo using Truffle, you should have Celo set up Celo in your local environment. If you prefer to deploy without a local environment, you can deploy using Remix or Replit.

- Using Windows
- Using Mac
- Using Replit

Setup Project Folder

- Open your terminal window, create a project directory, and navigate into that directory.

  mkdir myDapp && cd myDap
- Install hdwallet-provider

  From root truffle project directory, install truffle/hdwallet-provider. This allows you to sign transactions for addresses derived from a mnemonic. use this to connect to Celo in your truffle configuration file.

  npm install @truffle/hdwallet-provider --save
- Initialize Truffle

  Initializing truffle creates the scaffolding for your truffle project.

  truffle init
- Open Project

  Open your project in Visual Studio code or your preferred IDE.

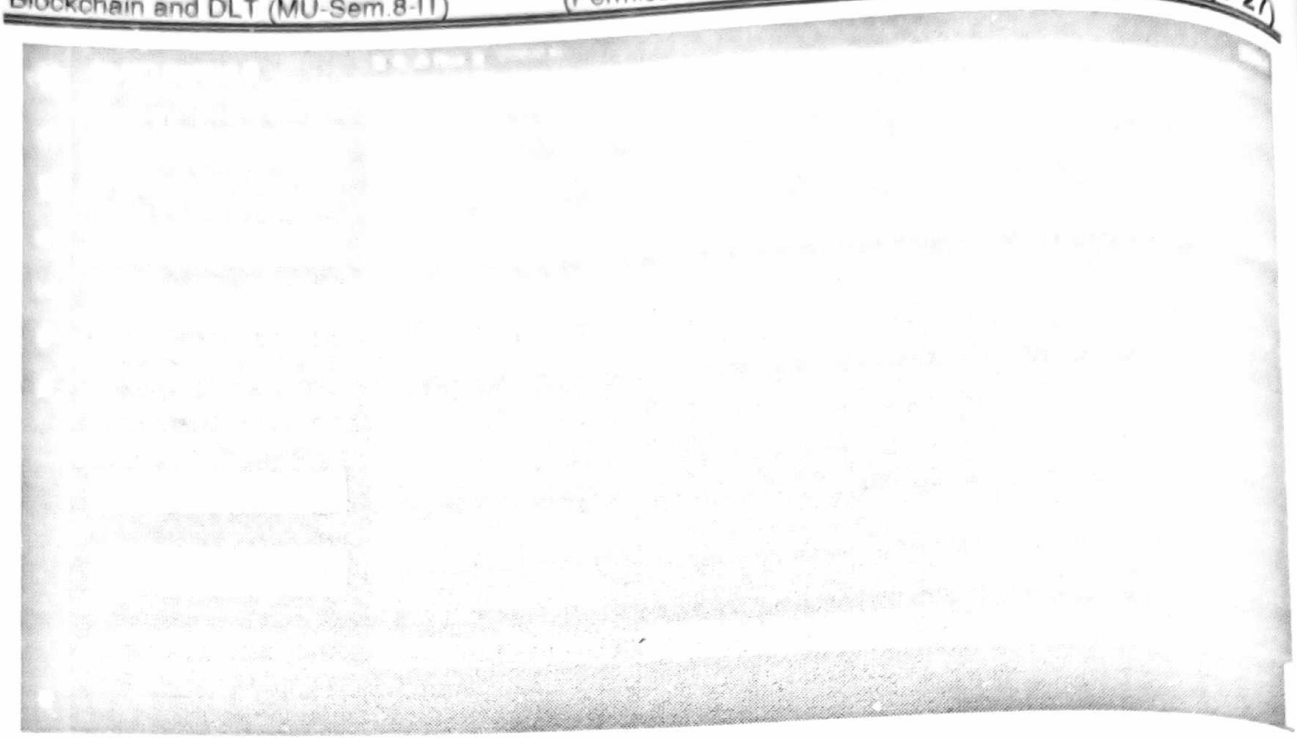
  code .

3.13.2 Use of Remix and Test Networks for Deployment

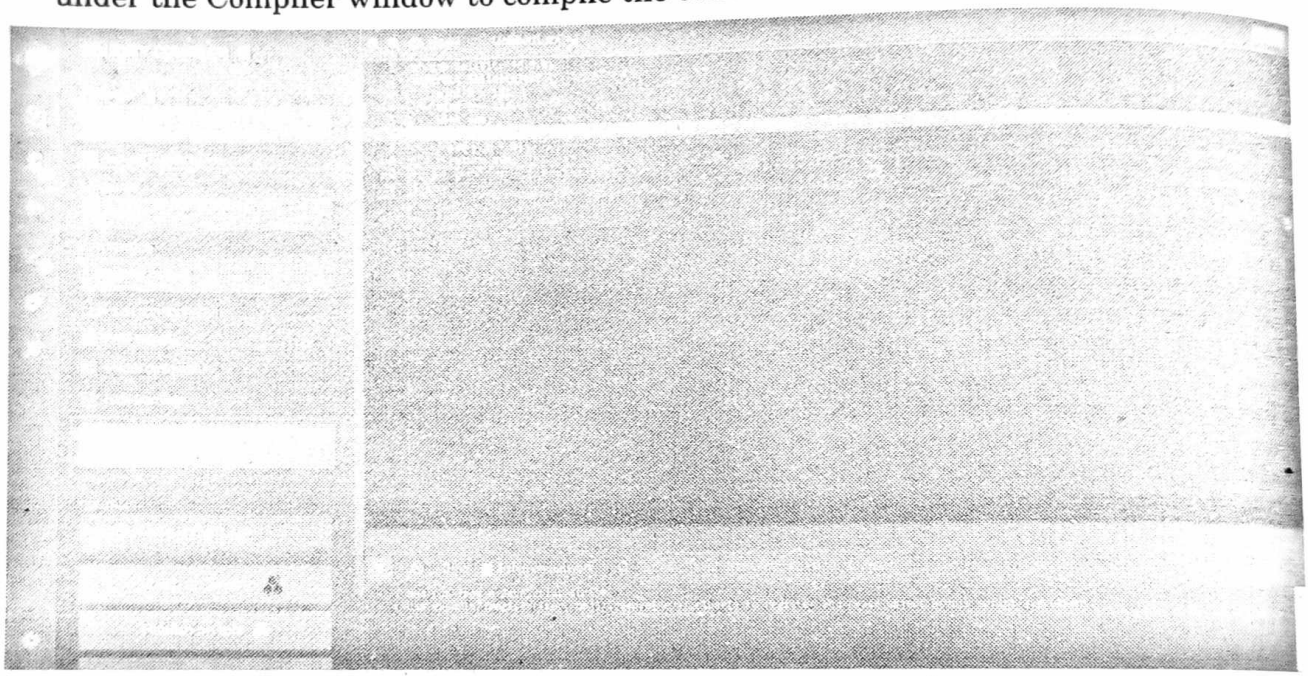
Remix IDE is used for the entire journey of smart contract development by users at every knowledge level. It requires no setup, fosters a fast development cycle and has a rich set of plugins with intuitive GUIs. The IDE comes in 2 flavors (web app or desktop app) and as a VSCode extension.

Remix IDE is generally used to compile and run Solidity smart contracts. Below are the steps for the compilation, execution, and debugging of the smart contract.

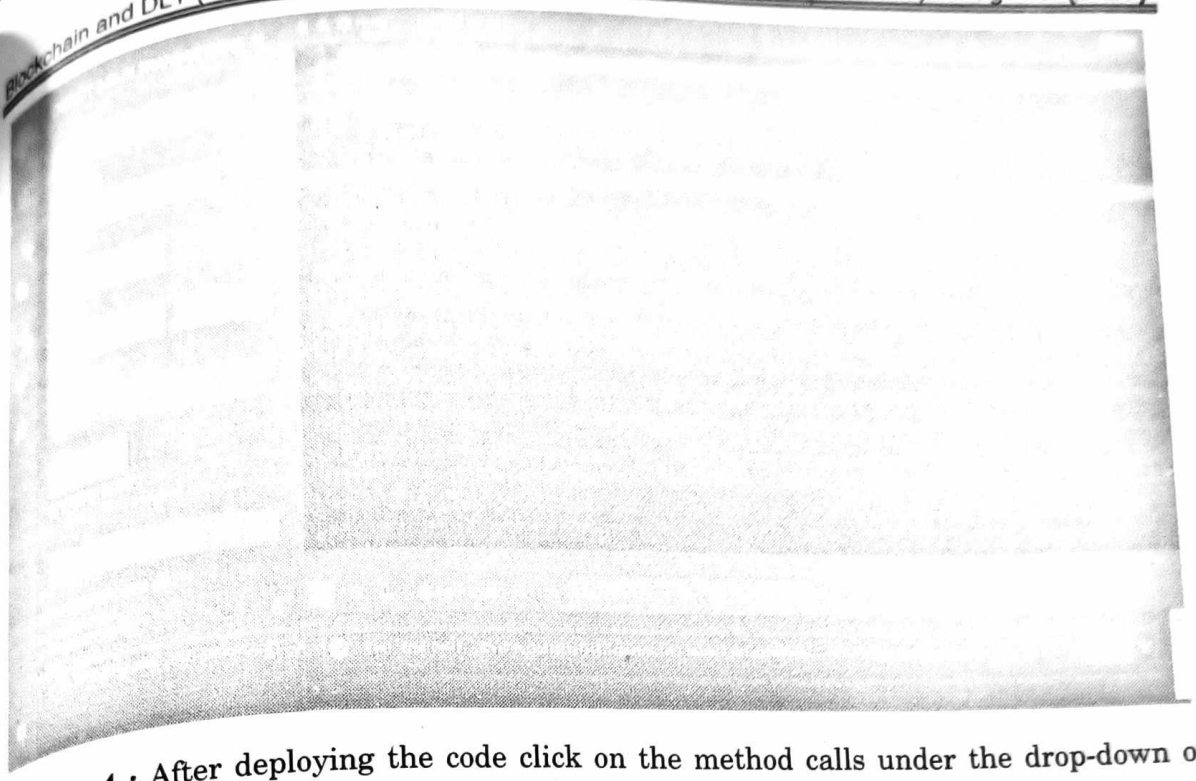
- **Step 1:** Open Remix IDE on any of your browsers, select on the *New File* and click on *Solidity* to choose the environment.



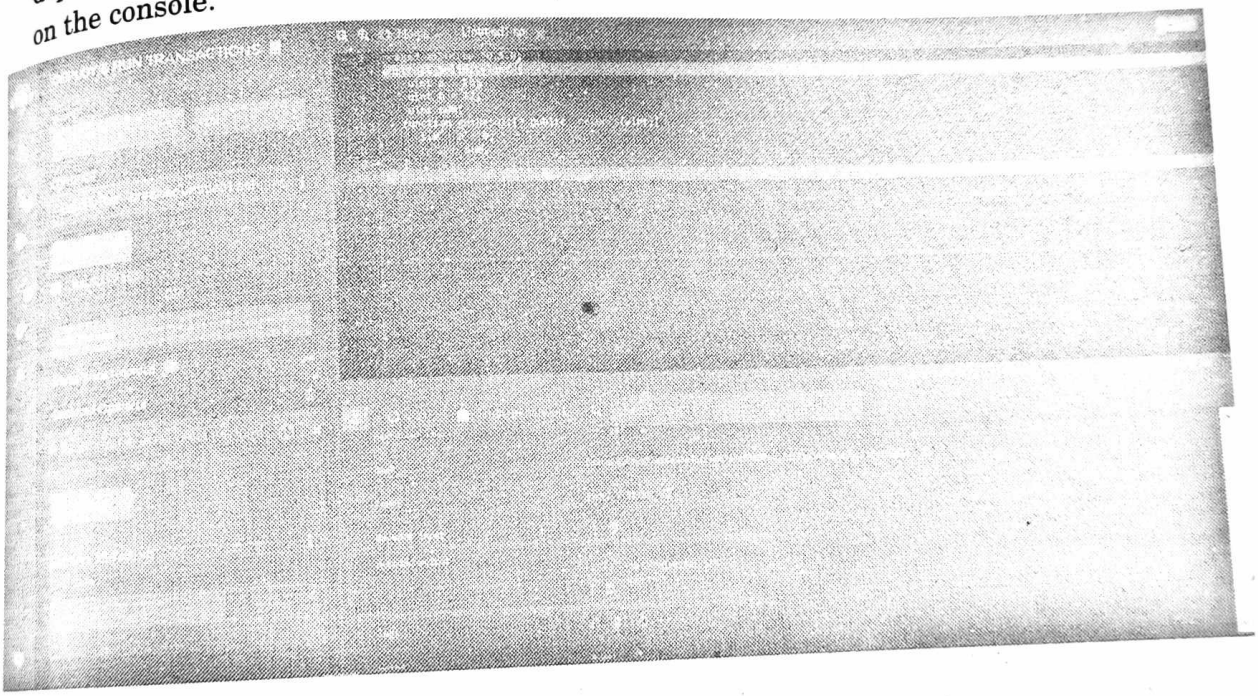
- **Step 2:** Write the Smart contract in the code section, and click the *Compile button* under the Compiler window to compile the contract.



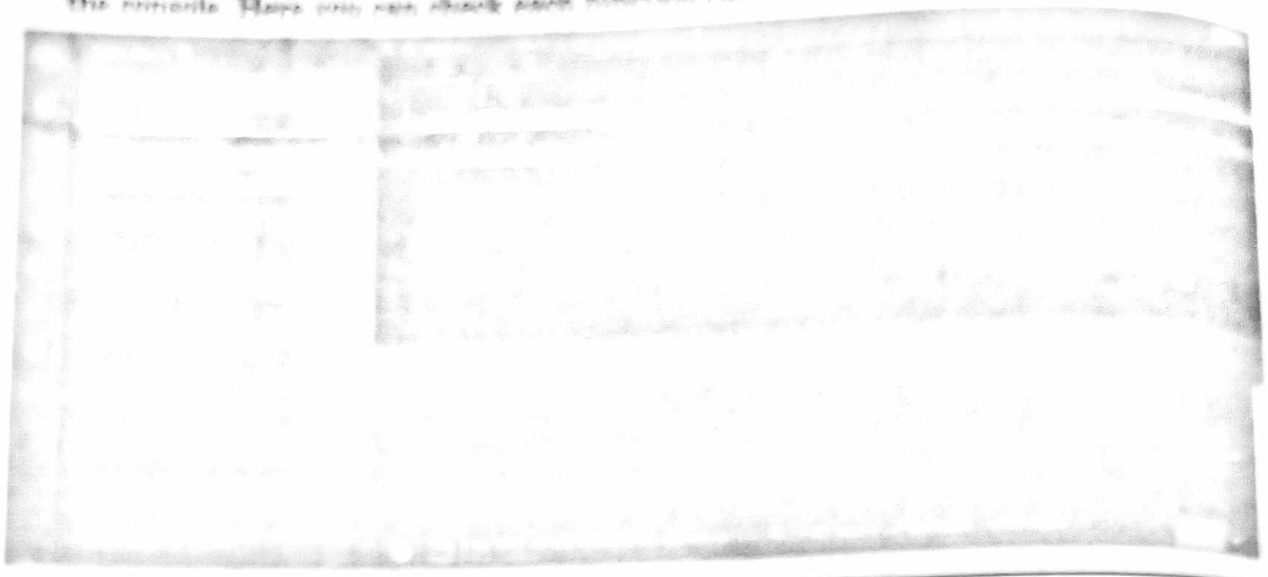
- **Step 3:** To execute the code, click on the *Deploy button* under Deploy and Run Transactions window.



Step 4 : After deploying the code click on the method calls under the drop-down of deployed contracts to run the program, and for output, check to click on the drop-down on the console.



- **Step 5:** For debugging click on the *Debug* button corresponding to the method call in the console. Here you can check each function call and variable assignments



Chapter Ends