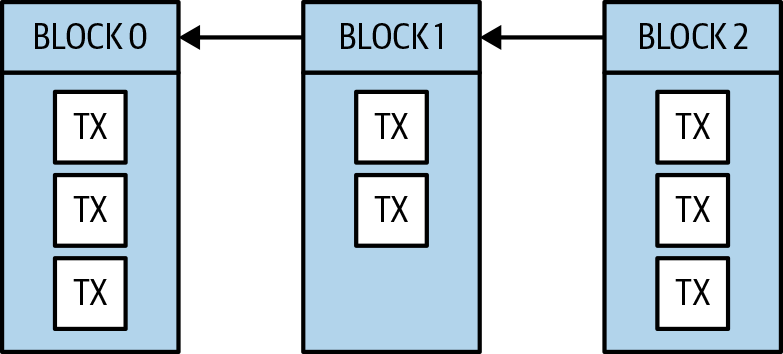
Fundamentally, a blockchain is a data structure. It is a linked list, or chain, of unique “blocks.” Each block points to the previous one, and is itself a list of transactions. On top of this relatively simple list-of-lists data structure is laid the key innovation that blockchains have given us: a protocol for how blocks are added to the chain without any central authority.

Ethereum provides a cryptographically secure platform for storing, updating, and removing data from a blockchain using what is referred to as “smart contracts.”



 Solidity is a popular programming language for developing smart contracts, and was designed to run on the Ethereum Virtual Machine (EVM).

Transactions of value exchange enter a peer-to-peer network, and are periodically grouped into “blocks,” or lists. When a “block” of transactions is persisted, it is “chained” to the previous block. This append-only data structure and the protocol that constructs it create an immutable record of transactions.

Blockchain prodigy Vitalik Buterin made the ambitious decision to stop trying to extend Bitcoin and instead create a more general-purpose protocol from scratch. In 2013, Vitalik wrote the [Ethereum white paper](https://github.com/ethereum/wiki/wiki/White-Paper).

 The term “smart contract” was coined by [Nick Szabo in 1994](http://bit.ly/szabo-sc). The idea then was that many legal contracts, notaries, and other analog agreements could be enforced nearly automatically using digital protocols and cryptographic signatures.

The virtual machine as defined by the Ethereum protocol is Turing-complete. This means that as long as you can fit your computations within the limitations of a single block, smart contract developers have few other constraints to contend with beyond their own imagination.

Many software developers will have worked with a technology stack that includes 1) a native mobile user interface and/or web user interface 2) with a server-side programming language that ultimately interacts with 3) a database. In the most basic versions of these systems, the interactions with the database are essentially instantaneous and permanent.

When we refer to “nodes” in the context of a blockchain, we are referring to software that someone has installed on a computer and connected to the blockchain network. Just like there are many different software implementations of the HTTP protocol (Apache, NGINX) called “web servers,” there are many software implementations of the Ethereum protocol (geth, Parity) called Ethereum nodes.

Blockchains were designed to run over decentralized networks. People and companies run nodes in the network, and, in many respects, all nodes are peers. This is possible because every node contains the full history of every transaction that ever occurred on the blockchain. With every node as a self-sufficient, independent entity, there is no center of the network. All nodes validate transactions and propagate them to their peers.

On top of this, some nodes also participate in the block creation process, and are financially incentivized to do so via receiving a “block reward” in the blockchain’s native cryptocurrency. Unlike our database example, any node in a decentralized network can join or drop at will. No special permission or rights are needed to read or write to the blockchain as long as the protocol is followed.

 Every major Ethereum node software has a hardcoded list of “bootnodes.” These are well-known, relatively reliable IP addresses that can be used to create a sufficiently large pool of network peer connections. If for some reason those nodes aren’t available or are no longer trustworthy, the node software can be given a custom list of bootnodes to use.

In 2019, the global, public Ethereum network had more nodes than any other blockchain network. With over 12,000 nodes, it’s unlikely that any two nodes will have the exact same set of peers. As a transaction enters the network through a single peer, it spreads quickly around the world to every node. Similarly, when a new block is added to the blockchain, news of this addition spreads quickly. Unfortunately, quickly is not the same as instantaneously, meaning that when two blocks are created at the same time by two different nodes, the respective peers of those diverging nodes will be operating on two different versions of the blockchain. This is called a temporary fork. In general, temporary forks are resolved by granting precedence to whichever fork adds the next block, summarized as “longest chain wins.” These rules are determined by a blockchain’s consensus protocol.

CONSENSUS PROTOCOLS

 A consensus protocol is simply a system of agreement across a blockchain. There are three protocols that we need to cover, and we’ll start with Proof-of-Work, which Ethereum used when they launched their blockchains.

**PROOF-OF-WORK**

When you hear about “mining ether” or “cryptomining,” this terminology is due to the nature of the Proof-of-Work (PoW) protocol. Like a miner in search of gold, the PoW protocol for creating a block requires considerable effort, and the more effort you expend, the more likely you are to “mine a block.”

Each Ethereum block contains a block reward in Ethereum’s currency, known as “ether.” This reward is granted to the miner who successfully adds a valid block to the blockchain before any other miners. The miner succeeds by successively “hashing” the block data in an attempt to find a cryptographic hash that has specific characteristics. The protocol determines the rarity of these “hashes.” This rarity is called the “difficulty,” the number that determines the relative effort that miners have to expend to “mine a block.”

Proof-of-Work uses a difficulty variable to maintain a stable block time as miners come and go, adding to or subtracting from the overall mining power (or hashrate) of the network. For example, if the Ethereum network had a cumulative hashrate of 100 tera hashes per second, and a large mining pool went offline, lowering the cumulative hashrate to 90 TH/s, blocks would suddenly take, on average, 10% longer to mine.

Ethereum’s protocol adjusts the difficulty to ensure that the correct block time is targeted. In this case, the protocol would adjust by making finding blocks about 10% easier. This process means that in the event of significant changes in the network’s hashrate, the block times will be affected.

### PROOF-OF-STAKE

Proof-of-Stake (PoS) has always been on Ethereum’s roadmap. If designed correctly, PoS has significant benefits over PoW. First, there is no need to burn large amounts of electricity to find a valid block. As the respective names suggest, there is no “work” to do, only “stake” to lose. Second, the downside of being a bad actor in a PoS system can be much more severe than in PoW.

Block creators stake ether in order to participate, and if they are proven to be operating maliciously, that stake can be taken away (“slashed”). In PoW, malicious operators can be ignored, but we can’t take away their hardware. That asset lives to corrupt another day. Finally, there are less economies of scale in PoS. When a PoW miner earns a block reward, they can use that money to achieve non-linear advantages over other miners, such as faster network connectivity. In PoS, while people with more stakes will earn more ether, that advantage is linear. The rich still get richer, but not exponentially so. That said, in PoW in order to obtain currency, you can buy hardware and connect to the network, whereas in PoS you must buy from existing currency holders, which gives newcomers a disadvantage.

### PROOF-OF-AUTHORITY

In some situations, it may make sense to use a blockchain while restricting block creation to only certain entities. These situations cut against the typical use case of having it be open to the public. This sort of restricted block creator status is provided by the Proof-of-Authority (PoA) protocol. PoA is commonly used for private blockchains that are hidden within internal networks. The block creators simply take turns appending the next block at the correct frequency. Due to the round-robin nature of PoA, temporary forks are far less likely.

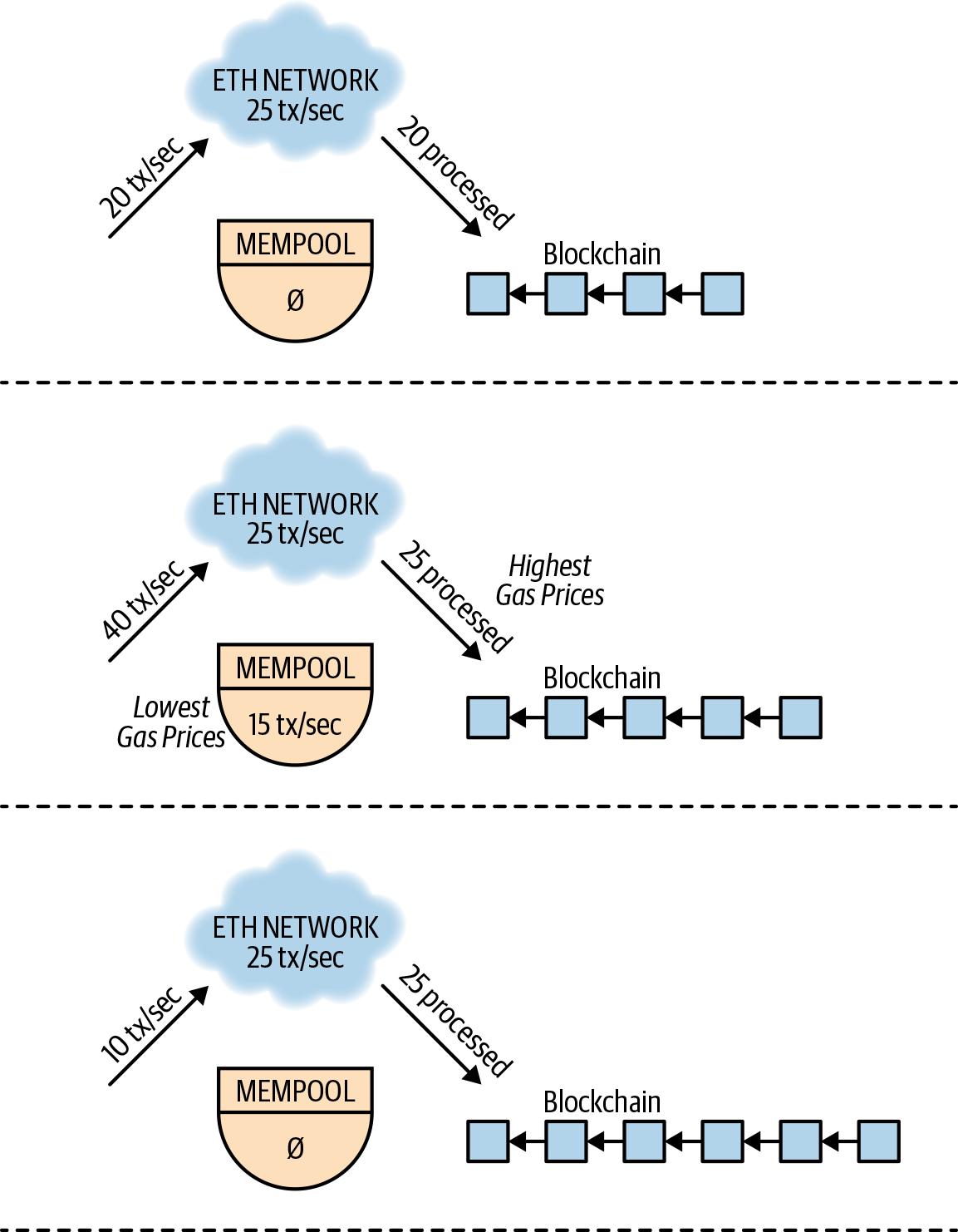
Regardless of which protocol is used, there is a reward for any node that adds a new block to the blockchain. Block creators receive a block reward as well as the sum of all transaction fees in the block. In Ethereum, transaction fees are referred to as “gas.”

## Transaction Processing

This wide variation in block times is due to the random nature of Proof-of-Work. As Ethereum transitions to Proof-of-Stake, we expect to see block times and block time variability decrease.

While block times are regulated by the protocol, the actual time that a transaction takes, from the moment it is first broadcast on the network to its execution in a block, can vary from a few seconds to a few hours.

While block times are regulated by the protocol, the actual time that a transaction takes, from the moment it is first broadcast on the network to its execution in a block, can vary from a few seconds to a few hours. This variability is due to the limitations of the current Ethereum protocol combined with its popularity. For example, if Ethereum can process around 25 transactions per second, but there are over 30 transactions entering the network every second, then there will be transactions that will remain unexecuted until network demand slows down to under 25 transactions per second. These pending transactions are held in-memory by each Ethereum node in what is called a “mempool.” The most reliable way to quickly get a transaction out of the mempool and into a block is to pay more fees to the block creator. In Ethereum, these fees are called “gas,” and as a smart contract developer, you’ll have to frequently consider “gas prices.”



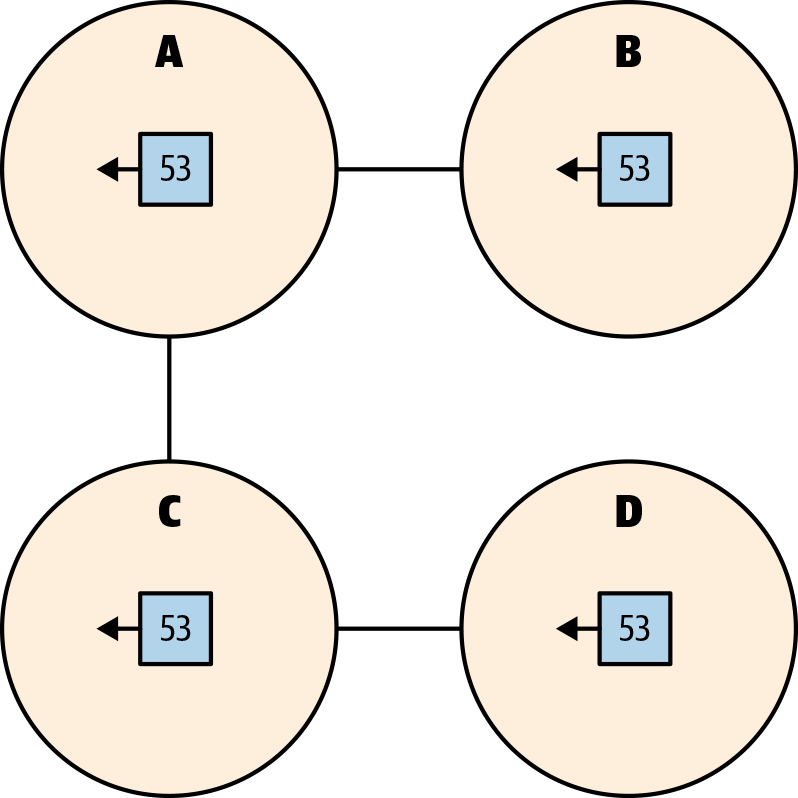
*Figure 1-2. In step 1, just 20 transactions/second are broadcasted on the network. Assuming the network can currently handle 25 transactions/second, all transactions are mined into the next block. In step 2, 40 transactions/second are broadcasted and the network falls behind, creating a mempool of pending transactions. In step 3, few transactions are sent—just 10 per second—and the mempool drains as the network catches up. The fewer transactions in step 3 would have likely been caused by rising gas prices.*

If the same account has two transactions in the mempool, the block creators know which transaction to choose first based on its “nonce,” an account-specific counter that is incremented with each transaction. This means that while the mempool is generally a heap of transactions that block creators can choose from, it’s actually a heap of queues in the cases where accounts have sent multiple transactions before seeing them succeed.

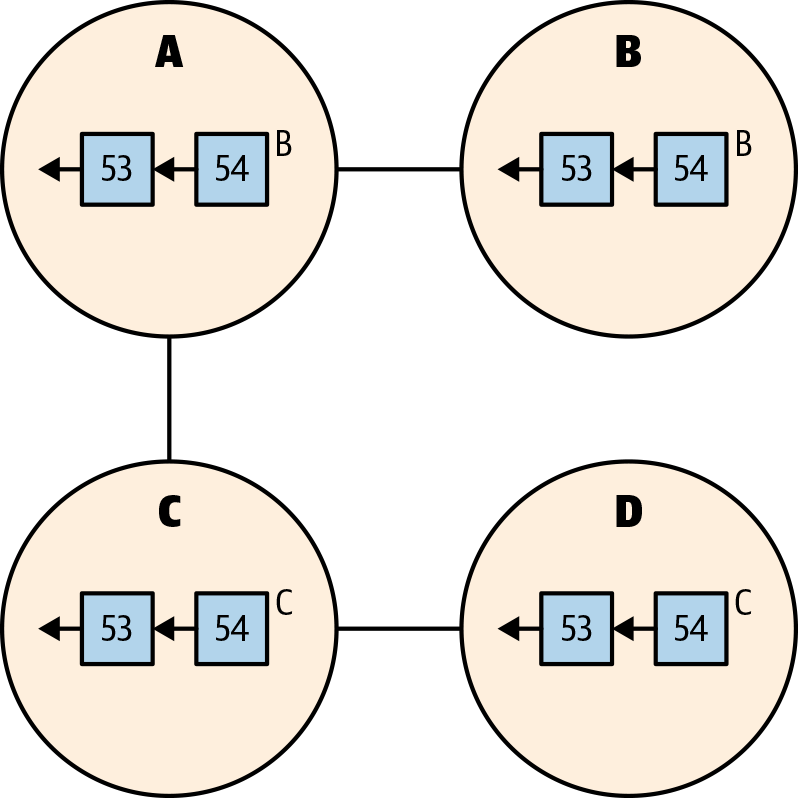
Regardless of whether a transaction is delayed by the nature of the protocol or due to network congestion, as a developer, you need to consider the unpredictable and asynchronous nature of your users’ experiences with your software system. In a world where financial, commerce, and social media software have been tuned to accept transactions in less than a second, systems backed by smart contracts have significant user experience challenges.

## Transaction Finality

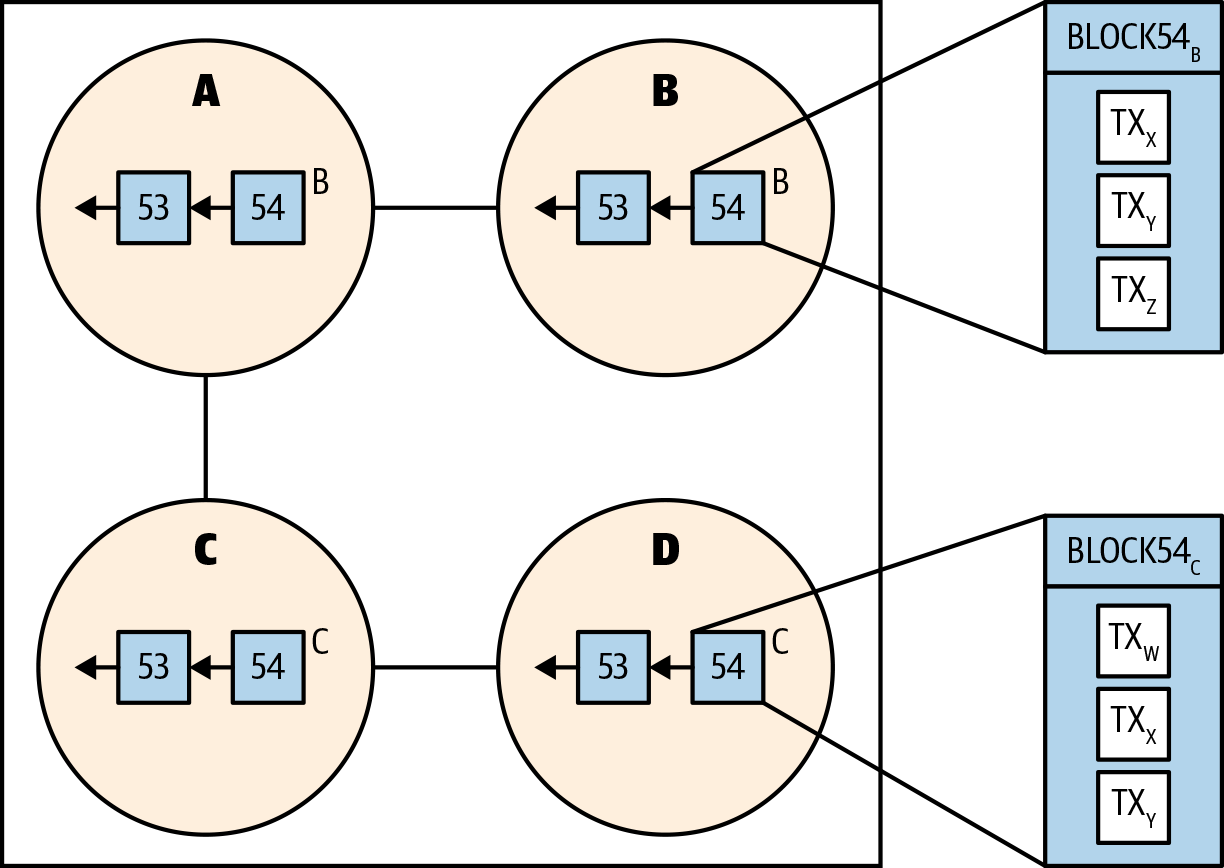
When you successfully execute a transaction in a typical database, developers assume that its effects will never be rolled back. In the case of blockchains, though, it’s possible that a user could see a successful transaction included in a block, only to see that block promptly orphaned and replaced with a different block. This block may not have included the previously successful transaction! This lack of finality with blockchain transactions is due to the decentralized nature of the network. It’s possible for two different nodes to create blocks nearly simultaneously, and for their respective peers to be on different “forks” of the blockchain, as mentioned earlier. Whichever chain creates the next block first will beat the other fork, and previously successful transactions can disappear in the process. From the user’s standpoint, it appears that their once successful transaction gets reverted. Figures [1-4](https://learning.oreilly.com/library/view/Hands-On+Smart+Contract+Development+with+Solidity+and+Ethereum/9781492045250/ch01.html#temp-fork-image1) through [1-6](https://learning.oreilly.com/library/view/Hands-On+Smart+Contract+Development+with+Solidity+and+Ethereum/9781492045250/ch01.html#temp-fork-tx-image3) illustrate the issue.



###### *Figure 1-4. Illustrating a temporary fork, step 1: all four nodes have the same block 53 and consensus with each other.*



###### *Figure 1-5. Illustrating a temporary fork, step 2: nodes B and C have forked from each other due to mining (or receiving) two different block 54s simultaneously. Node D has received Node C’s block 54, while Node A has received Node B’s block 54. There are now two divergent blockchains.*



###### *Figure 1-6. Illustrating a temporary fork, step 3: note that the competing blocks of a temporary fork can contain different transactions. Once this fork is resolved, either TXw or TXz will no longer exist on the blockchain.*

This problem can be solved by waiting for additional blocks to be added to the chain before considering the transaction settled. Each application will need to have its own threshold for considering a transaction final. Popular cryptocurrency exchanges will wait for up to 50 blocks for a transaction to be treated as “confirmed” on the Ethereum network. When the stakes are lower, it’s less necessary to wait that long. Each application should consider how long it wants to wait to consider transactions final. **In some cases, this “confirmation” will be built into the user experience, and in others, it may just be a general warning that not all transactions are final.**

Ether Gas

The Ethereum protocol has its own currency, called ether. The fundamental use of this currency is to pay block creators to include transactions in blocks. Much like the US dollar, ether is divisible, though to much smaller fractions than a cent. The smallest unit of ether is called a wei, which is a quintillionth of an ether (for perspective, a quintillion is a billion billions, or 1018). Due to a wei’s small size in proportion to ether, you’ll often see ether denominated by Gwei, particularly when it comes to gas prices. A Gwei is a billion wei, and a billion Gwei is one ether.

1 ether = 10^9 Gwei

1 Gwei = 10^9 wei

The ether on the public Ethereum network is known as **ETH** and it has real-world value. The public Ethereum network is referred to as “**mainnet**” by developers. There are also public test networks or “**testnets**” that the community uses as staging environments. These test networks typically have “**faucets**,” or mechanisms to give developers free ether in order to test their smart contracts. Every network running the Ethereum protocol has ether, but mainnet’s ETH is the one that has actual value. **To deploy and execute smart contracts on mainnet, we need an account that holds ETH.** To acquire ETH you will need to purchase it via an exchange, receive it from a friend, earn it from a business, or mine it yourself.

**Smart contract code, such as Solidity, is compiled to bytecode, which provides a series of opcodes to the EVM. An opcode is an instruction such as PUSH1 or MLOAD that is interpreted by the EVM. Each of these opcodes has an associated “gas cost.”**

Each Ethereum transaction has to include gas and gasprice attributes, which when multiplied will set the maximum transaction fee for the transaction, denominated in wei.

Each Ethereum transaction has to include gas and gasprice attributes, which when multiplied will set the maximum transaction fee for the transaction, denominated in wei. This gas attribute sets a limit to how many computations the transaction can perform. If that limit is reached, the smart contract execution reverts, but the transaction is still written to the blockchain, and the fees are consumed by the block creator. If the smart contract call completes with leftover gas, the gas is returned to the transaction creator. The sum of the gas used across all transactions in a block cannot exceed the block’s specified gaslimit. This also means that no single transaction’s gas usage can exceed the block’s gaslimit.

## Accounts

**The most basic Ethereum transactions are just externally owned addresses (EOAs) sending ether to each other.** Additionally, Ethereum transactions can be sent from an EOA to a smart contract. Both EOAs and smart contracts are identified by an Ethereum address like the following:

0x52bc44d5378309ee2abf1539bf71de1b7d7be3b5

**An address is represented by a hexadecimal number. Given the magnitude of the numbers involved in generating an address, it is practically impossible to generate two identical addresses.** There is no way to distinguish an address for a smart contract from an EOA without inspecting the blockchain. While accounts and contracts have undistinguishable addresses, they have some important differences.

Every transaction in the Ethereum blockchain is initiated by an EOA. Smart contracts can’t spontaneously perform an action. They can call other smart contracts, but every transaction originates from an EOA. When contracts are called, they can emit events, store data, receive ether, send ether to EOAs, or send data or ether to other contracts. Besides initiating transactions, EOAs, on the other hand, can only receive ether. They cannot react to any transactions that involve them the way that a smart contract does.

## Contracts

Contracts in Solidity are organized in an object-oriented style similar to the Java programming language. In object-oriented parlance, a contract is really a class, or a collection of state variables and functions. In order to reuse common functionality, object-oriented languages allow a class to inherit from another class (or contracts, in this case). Since Solidity uses function as a keyword, we will refer to as “functions” what most object-oriented languages refer to as “methods.” Solidity functions can be separated into two distinct types: write-only and read-only.

Read-only Solidity functions are denoted with the pure and view keywords.

 Read-only functions cannot change the state of the contract or emit events. Because no updates are needed on-chain, read-only functions are instantaneous—they are very similar to web API calls, particularly GET requests. It’s important to note that being able to skip on-chain updates means that read-only functions can be called without paying any gas costs, and there will be no transaction created.

Write-only functions are the default in Solidity so they require no additional keywords. Despite being “write-only,” they can actually return data, but due to the asynchronous nature of Ethereum, the return data is practically useless, hence “write-only.”

These functions are the workhorses of Ethereum, and their data must be sent via transaction and included in a block in order for the function to be executed. An unsuccessful write-only method will revert, either because it has run out of gas or reached an invalid EVM state, or because of an explicit statement in the contract, such as failing a require statement. A successful write-only method doesn’t actually have to change anything, but it typically changes something and often emits one or more events in the process.

**The purpose of events in Ethereum is generally twofold: to provide a custom historical log of what has occurred in the contract, and to allow observers to subscribe to real-time updates**.

## Blocks and Transactions

Only Ethereum block creators determine the attributes of a block, such as which transactions are included. Similarly, only Ethereum users determine the attributes of a transaction, such as which contract to send data to.

Solidity exposes the following transaction (tx) attributes:

*gasprice*

The gas price (in wei) set by the EOA that created the transaction.

*origin*

The address of the EOA that created the transaction. Surprisingly, this address is rarely useful and often insecure.

A transaction can have an arbitrary number of contracts involved in its execution, provided that its execution fits within the constraints of the block’s gaslimit. Solidity exposes a number of other transaction-related attributes, but groups them into a message (msg) abstraction. Messages refer to the communication between contracts and anything that can call them, such as other contracts. For example, a contract function call will always have a msg.sender. That msg.sender could be equal to the tx.origin or the creator of the transaction, or it could be the address of an intermediary contract.

Solidity exposes the following block attributes:

*number*

Each block increments this number. The genesis block was block 0.

*timestamp*

The time in seconds since epoch that the block was created. You may also see code that uses its alias, now.

*blockhash*

In addition to its sequential block number, each block is uniquely identified by its hash. A hash is a hexadecimal number such as 0x88e96d4537bea4d9c05d12549​907b32561d3bf31f45aae734cdc119f13406cb6. The hash of the current block is not available, but by providing a block number, you can get the hash of any block within the past 256 blocks.

*difficulty*

The mining difficulty level of the current block.

*gaslimit*

The maximum amount of gas that can be consumed by this block. This is set by the block creator.

*coinbase*

The address of the block creator.

The events emitted by this function would all have the same exact time attribute. That time attribute, set by block.timestamp, is the time that the block was added to the blockchain. For every transaction in that block, the block.timestamp attribute will be identical.

When designing smart contracts, it is also important to keep in mind that that block creators can manipulate the time a block is created as well as the ordering of the transactions to their advantage.

A private key is the secret counterpart to a public key. The address of an EOA is a truncation of the hash of its public key.[3](https://learning.oreilly.com/library/view/hands-on-smart-contract/9781492045250/ch01.html#idm44814751614936)

When we send an Ethereum transaction using any of the Web3 libraries, the cryptographic signature happens in the background. The following transaction attributes are concatenated, encoded, and then signed with the configured private key:

*nonce*

Sequence number of this transaction for this EOA.

*gasPrice*

Amount of wei that this transaction is paying per unit of gas.

*gas*

Amount of gas that this transaction is willing to spend.

*to*

Recipient address of this transaction. It could be an EOA or a contract.

*value*

Amount of wei (if any) that this transaction is sending to the recipient.

*data*

In the case of a contract call, this contains the function name and all parameters. In the case of a contract deployment, this contains the contract bytecode. If no contracts are involved, this is generally blank.

*chainId*

Each public Ethereum network has a chainId. Mainnet is 1, the Kovan testnet is 42, etc.

Once those attributes are signed, the signature itself is included in the transaction so that Ethereum nodes can validate that the sender is legitimate. In order to validate this, nodes use the sender’s address to validate the signature. If someone were to try to send a transaction with a bad signature, the nodes would reject it.

We will start by installing an Ethereum client, which is software that can be used to interact with the Ethereum blockchain. We will then install Node.js, which provides the JavaScript environment for Truffle. Lastly, we will install Truffle and Ganache from the Truffle Suite. Truffle provides a fantastic set of utilities used for testing and deploying our contracts, while Ganache gives us a local blockchain environment to run our application locally.

When you install an Ethereum client, you are installing software that will allow you to run an Ethereum node on your machine. This software comes with a command-line interface (CLI), which allows you to create accounts or launch an interactive console that preloads Web3 (more on that in later chapters). Additionally, this software will run a server to expose the Ethereum JSON RPC API.

If you are unfamiliar with JSON RPC, it is a lightweight remote procedure call (RPC) protocol. This means we use JSON to send a request to a server, which then executes some predefined operation. It is through this JSON RPC that we will be interacting with the blockchain.

There are a number of Ethereum clients you can use, including cpp-ethereum, go-ethereum (otherwise known as Geth), and a few others, but the one we will be using is called Parity. Parity is a client written in Rust and provides one of the faster syncing options of the available clients.

Goerli, which is a network designed to work with several different Ethereum clients such as Geth and Parity.

# Installing MetaMask

If your application required users to download and run a full Ethereum client such as Parity or Geth, you just lost a lot of potential users. Asking a user to install and run a full Ethereum node is a bit much for all but the devoted crypto enthusiast.

 The MetaMask software provides users with the ability to create accounts, and loads a preconfigured instance of Web3 into the browser that is used to interact with the blockchain via JSON RPC.

It also provides the user with a 12-word mnemonic that they can store to later recover their accounts should they change computers, browsers, or lose their password.

## Truffle

Though Truffle is often referred to as a framework, it is really more of a blockchain utility belt. It provides tools that make compiling, testing, deploying, and packaging your application as easy as possible. These utilities rely on the generated directory structure, which is where the framework aspect comes into play, and the Truffle CLI.

Truffle is distributed as a npm package. To install it, run the following command:

**$** npm install -g truffle

Since we used the -g flag when running the install command, the output lets us know where to find the symlink that is creating the globally available executable.

## Ganache

Ganache is your very own blockchain. In many regards, it is very much like the Ethereum client we downloaded earlier. It provides tools for creating accounts and runs a JSON RPC API server for you to connect and read/write to the blockchain. The main difference is that it doesn’t actually connect to the Ethereum network; in fact, older versions would destroy all data when you shut down so you had a clean slate when starting up again.

# Setup

As we get set up, we will need a directory to hold our new application. Let’s first create a directory called greeter and change into our new directory. Open your terminal and use the following commands:

**$** mkdir greeter

**$** cd greeter

We are now going to initialize a new Truffle project as follows:

**$** truffle init

This command will generate the following output:

✔ Preparing to download

✔ Downloading

✔ Cleaning up temporary files

✔ Setting up box

Unbox successful. Sweet!

Commands:

Compile: truffle compile

Migrate: truffle migrate

Test contracts: truffle test

Our greeter directory should now include the following files:

greeter

├── contracts

│   └── Migrations.sol

├── migrations

│   └── 1\_initial\_migration.js

├── test

└── truffle-config.js

Notice that the commands specified in the output line up well with the directory structure generated when initializing our application. truffle compile will compile all the contracts in the contracts directory, truffle migrate will deploy our compiled contracts by running the scripts in our migrations directory, and lastly, truffle test will run the tests in our test directory.

 Every time we run the truffle test command, Truffle first compiles our contracts, and then deploys them to a test network. In order to deploy our contract, we need to turn to another tool provided by the Truffle toolbelt: migrations.

Migrations are scripts written in JavaScript that are used to automate the deployment of our contracts. The default Migrations contract found in contracts/Migrations.sol is the contract that is deployed by migrations/1\_initial\_migration.js and is currently the only contract that has made its way to the test network.

**const** GreeterContract = artifacts.require("Greeter");

module.exports = **function**(deployer) {

deployer.deploy(GreeterContract);

}

# Saying Hello

This is normally the point where we would use a call to some variant of printf or println to output our greeting, but in Solidity we do not have access to standard out, or the file system, the network, or any other input/output (I/O). What we do have are functions.

Once deployed, our smart contract will be stored on the Ethereum network at a specific address. It will be dormant until a request comes in asking it to perform some work and the work our contract can do is defined by our functions

**external function. This means that it is part of our contract’s interface and can be called from other contracts, or from transactions, but cannot be called from within the contract** or at least not without an explicit reference to the object it is being called on.

**public functions are also part of the interface, meaning they can be called from other contracts or transactions, but additionally they can be called internally**. This means you can use an implicit receiver of the message when invoking the method inside of a method.

**internal and private functions must use the implicit receiver or, in other words, cannot be called on an object or on this.**The main difference between these two modifiers is that private functions are only visible within the contract in which they are defined, and not in derived contracts.

**Functions that will not alter the state of the contract’s variables can be marked as either pure or view.** **pure functions do not read from the blockchain. Instead, they operate on the data passed in or, as in our case, data that did not need any input at all. view functions are allowed to read data from the blockchain, but again they are restricted in that they cannot write to the blockchain.**

We also indicate that this is a value that is not referencing anything located in our contract’s persisted storage by using the keyword memory.

Because this function is being called from the outside world, the data being passed in as a parameter is not part of the contract’s persisted storage, but is included as part of the calldata and must be labeled with the data location calldata. The calldata location is only needed when the function is declared as external and when the data type of the parameter is a reference type such as a mapping, struct, string, or array. Using value types like int or address do not require this label.

State variables will be available to all functions defined inside of a contract, similar to instance variables or member variables of other object-oriented languages. They are also where we will store data that will exist for the entire lifetime of our contract. Like functions, state variables can be declared with different levels of visibility modifiers, including public, internal, and private. In our previous example, we used the private modifier, which means this variable is accessible in our Greeter contract.

# Making the Greeter Ownable

As it stands now, anyone can change the message of our Greeter contract. This may be fine in some cases, but it could also lead to someone changing the message to something less than welcoming. To prevent this, we will now add the idea of ownership to the contract, and then restrict the ability to change the greeting to the owner.

In order to do this, we want to set the owner of the Greeter contract to the address that deployed the contract. This means we’ll need to store the address during initialization, and for that, we’ll need to write a constructor function. We will also need to access some information from the msg object. The msg object is globally available and includes the calldata, sender of the message, the signature of the function being called, and the value (how much wei was sent).

**The Solidity language provides two address types: one is address and the other is address payable**. The difference between them is that address payable gives access to the transfer and send methods, and variables of this type can also receive ether. We are not sending ether to this address and we can use the address type for our purposes.

Now that we know who created the contract, we can create a restriction that only the owner can update the greeting. This type of access control is normally done with a function modifier.

Function modifiers allow us to extend a function with code that can run before and/or after a function. These typically take the form of a guard clause and will prevent the function from being invoked if the clause is not met, which is exactly what we want for our Greeter contract.

Let’s create and apply our modifier to fix this right up. Back in our Greeter contract, after the constructor, add the following code:

**modifier** onlyOwner() {

require(

msg.sender == \_owner,

"Ownable: caller is not the owner"

);

**\_**;

}

Here, our modifier is using the require function, where the first argument is an expression that will evaluate to a boolean. When this expression results in a false, the transaction is completely reverted, meaning all state changes are reversed and the program stops execution. The revert function also takes an optional string parameter that can be used to give more information to the caller as to why the operation failed.

The last part of our modifier function is the \_; line. This line is where the function that is being modified will be called. If you put anything after this line, it will be run after the function body completes.

Now let’s update our setGreeter function to use the modifier by adding the onlyOwner declaration after external in the function definition:

**function** setGreeting(**string** calldata greeting) **external** onlyOwner {

\_greeting = greeting;

}

OpenZeppelin and how they have created contracts that can be used as the basis for creating tokens. Well, they also have contracts that implement the idea of ownership, and we are duplicating some of that behavior. Instead of duplicating it, we will update our Greeter contract to leverage their implementation.

Back in our terminal, in the root directory of our application, enter the following command:

**$** npm install openzeppelin-solidity

Once that has completed, update the top of the Greeter contract to look like [Example 4-16](https://learning.oreilly.com/library/view/hands-on-smart-contract/9781492045250/ch04.html#GREETER8).

##### ***Example 4-16. Inheriting from Ownable***

**pragma solidity** >= 0.4.0 < 0.7.0;

**import** "openzeppelin-solidity/contracts/ownership/Ownable.sol";

**contract** Greeter **is** Ownable {

...rest of file...

Here, we added an import statement that will pull in all the global symbols from the imported file, such as Ownable, and make them available in the current scope. The next thing to notice is that our Greeter contract now inherits from Ownable through the is syntax. Solidity supports multiple inheritance much like Python or even C++. The inheriting classes are listed with a comma separating them.

With this in place, we can remove our implementations of the onlyOwner modifier, the owner getter function, and the constructor function since Ownable provides these definitions. This may seem like overkill, and normally I would agree with that thought. However, using code that has been through a thorough audit and well tested is prudent when working with smart contracts since security of smart contracts is critically important.

If we want to use our contracts with something other than our tests, we will need to get our contract onto a network that runs outside of the test process. To do this, we use the truffle migrate command. Unlike truffle test, we will need to specify which network we are going to deploy our contracts using the --network flag. Truffle will then look up the configuration details for the corresponding network in our truffle-config.js file and deploy our contracts based on those settings.

Let’s go ahead and compile our contracts. Back in our *greeter* directory, open a terminal and use the following command:

**$** truffle compile

We should see output that indicates the contracts that have been compiled and the location, relative to our project directory, where the files have been saved.

**When we deploy a contract, we submit a transaction to the Ethereum network. Unlike a normal transaction that contains both a sender (from) and receiver (to) address, our deployment transaction will need to set the receiving address to the 0x0 address.** Our deployment transaction will also need to include the bytecode mentioned earlier, which will be sent as the transaction data. With our contract being sent as a transaction, it has to be mined before we will be able to interact with it. When the contract is mined, it will execute the code in the constructor, setting the initial state for the contract.