14th International Conference on Network and Service Management (CNSM)
Rome, Italy, November 9, 2018

Blockchain and Smart Contracts: From Theory to Practice

CNSM Tutorial #3

Bruno Rodrigues[1], Roman Blum[2], Thomas Bocek[2], Burkhard Stiller[1]

[1]Communication Systems Group CSG, Department of Informatics IfI
University of Zurich UZH, Switzerland
[2]Distributed Systems & Ledgers Lab, Hochschule für Technik
Rapperswil HSR, Switzerland

Universität Zürich[UZH]

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

Communication Systems Group

Distributed Systems & Ledgers Lab

Abstract

Blockchains and Smart Contracts determine more recent advances in mechanisms and algorithms on top of distributed systems. Thus, they help building the foundation of a truly distributed digital society. This tutorial provides at first a basic theoretical introduction into blockchains and Smart Contracts and secondly a practical interaction with a blockchain and initial aspects for the development of Smart Contracts.
Hence, the audience is guided through the deployment of an Ethereum Blockchain and the creation of an ERC20 token (Ethereum Request for Comment Number 20) using Smart Contracts. In this regard, besides the practical support of the instructors, the audience will receive a key basis and an introduction to major concepts and issues with respect to blockchains and Smart Contracts. At the end of the tutorial, the audience is expected to be not only able to create blockchains and Smart Contracts, but also to interact on a detailed technical level with these components.

1. Blockchain

While Blockchain have started and are commonly associated with financial applications, many other application areas, use cases, and Blockchain types are emerging with the growth of public attention around the technology. In its purest form, a Blockchain acts like a shared, replicated, append-only database. In the Figure 1, blocks are connected based on a hash value, which links current blocks to previous ones. Each block represents a number of transactional records, which are confirmed and broadcast to the network so as every node in the Blockchain has their own updated version of the chain.

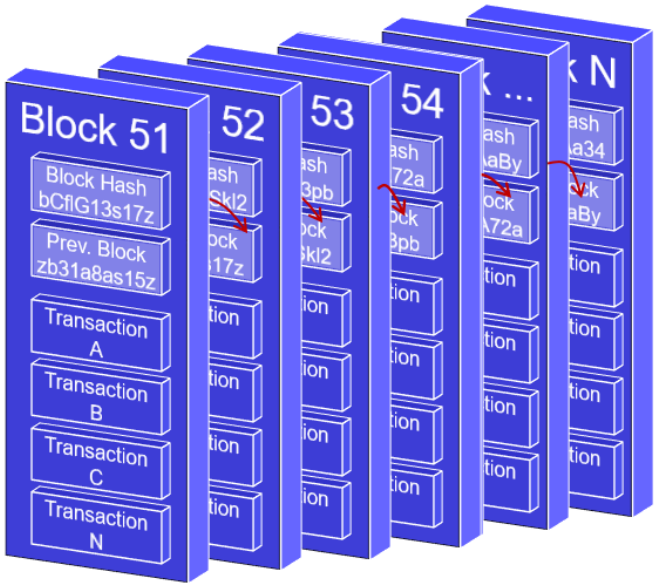


Figure 1 - Blockchain Example

Blockchain participants can distribute write and read permissions and participate in the block-validation process depending on the Blockchain type. Blockchain can be permissionless (*i.e.,* public) where all participants can read and write transactions or permissioned (*e.g.,* fully private or in a consortium-based approach). In these different deployment scenarios, different consensus algorithms (*e.g.,* proof-of-work, proof-of-stake) can be used to ensure that all nodes maintain the same, updated view over the chain and prevent issues such as double spending.

2. Blockchain and Traditional Databases

A traditional database often uses a client-server architecture where entries made by clients are stored in a centralized server and, eventually, replicated. Therefore, the control of the database is maintained by a central authority at the server, which allows for access and read or write permissions. Conversely, a Blockchain uses a fully decentralized model where each participant maintains, calculates and updates new entries that are stored not only in a central entity, but replicated over all nodes in the chain. The consequences of this difference is that blockchains can be better suited as a system of record for certain functions, while a centralized database is entirely appropriate for other functions. A comparison concerning main operations is presented in Table 1:

Table 1 - Comparison of Properties of Blockchains and Traditional Databases

Properties	**Blockchain**	**Traditional Databases**
Operations	Insert, Read	Insert, Read, Update and Delete
Replication	Full Replication	Master-slave
Consensus	Majority of peers agree on outcome of transactions	Distributed transactions
Validation	Any peer can validate transactions	Master node validate transactions

The authors state that a blockchain is not necessary if there is no need to store data or there is only a single entity involved in the use case [1]. However, it is worth to note that even for a single entity that is large enough there might be a lack of trust between its departments or operations in different countries [2], and thus a necessity to validate its operations. Similarly, a blockchain is not recommended when is possible to trust a third party or all involved parties are known and trusted to each other. For cases where a third party is trusted or there is no need for disintermediation, a blockchain-based solution might become a stepping-stone for data confidentiality and performance of transactions. Thus, when parties are known and trusted, a traditional database with shared access is likely to be the most suitable option to address these issues. A summary of advantages with respect to main characteristics is presented in Table 2:

Table 2 Comparison with respect to main characteristics

Characteristic	**Advantage**
Disintermediation	Blockchain
Performance	Traditional Databases
Reliability/Integrity	Blockchain
Data confidentiality	Traditional Databases

Conversely, for all cases that data needs to be stored and multiple entities that do not know to each other are involved, a permissionless (*i.e.*, open and decentralized) blockchain is recommended when it is not possible to trust a central third party. In these cases, goods and services can be exchanged and verified publicly without a central entity managing the membership of these parties. Otherwise, a public permissioned blockchain is necessary when public verifiability is required, but only some pre-defined parties can be trusted. For these cases, a central entity manages the membership of participants, is responsible for granting or denying permissions to a set of participants to read or write in the blockchain. Lastly, a private blockchain is recommended for cases when data is stored by multiple known and trusted parties, and there is no need for public verifiability among them. For example, this type of blockchain could be deployed in a large company that seeks greater transparency and auditability in its internal processes, but do not want to expose these processes to general public.

3. When to use (or avoid) Blockchain

There is no general formula to determine whether a blockchain makes sense or not without a detailed analysis of blockchain properties, requirements and goals of its potential participants, and characteristics of the application itself. However, the following basic requirements are expected to in order to fully explore blockchain capabilities:

- **Database**: there is a need to store state
- **Multiple writers**: blockchain is a shared database with multiple writers.
- **Absence of trust**: there needs to be a certain degree of mistrust between the multiple writers.

- **Disintermediation**: remove intermediaries by enabling databases with multiple non-trusting writers to be modified directly.
- **Immutability**: writers should be uniquely identified and their data should not be modified.

Conversely, the following characteristics need to be observed in order to decide whether to use or not blockchains:

- **Trusted Third-party**: a blockchain is not recommended if it is possible to trust a third party or all involved parties are known and trusted to each other. For cases where a third party can be trusted, or there is no need at all for disintermediation, a blockchain-based solution might become a stepping-stone for data confidentiality and performance of transactions
- **Performance**: blockchains are naturally slower than any centralized database. Besides of the regular operations performed in a traditional database, a blockchain should perform extra procedures to ensure that transactions are correctly recognized, propagated and verified. Also, traditional databases have been deployed, tested and optimized through decades of development.
- **Privacy and confidentiality**: users and data in a blockchain are visible to all nodes (with reading permission) participant in the blockchain. For use cases where a user and data transparency are desired, blockchain is the choice to verify and process every transaction transparently independently. However, if privacy and confidentiality are required, and there is a trusted third party, a blockchain poses no advantage over a centralized database. However, it is still possible to hide confidential information on a blockchain, but hiding critical information requires lots of cryptographic efforts for the nodes in the network.
- **Robustness or reliability**: it is related to the dependability of the database. Blockchains are typically fault-tolerant due to its built-in redundancy, which means that transactions are replicated within the blockchain network and permanently recorded. Some trusted third parties can also provide fault tolerance offering replication techniques, but blockchains can provide high-redundancy by default.

4. Blockchain in Practice

This tutorial will be based on the **Ethereum Blockchain** using the **Geth** client. Ethereum is the main blockchain for learning due to the greater support of its community in relation to other blockchains. This allows students in its first experience with blockchains to find more documentation, including guides and tutorials, that facilitate the learning process. In addition, Ethereum provide greater flexibility concerning the use of different consensus protocols and the definition of smart contracts in contrast to other public blockchains.

4.1 Procedure

The first step of this tutorial is to install Geth Client as shown in Section 4.2. While on platforms like Linux and MacOS it may be simpler, Windows and ARM may require additional configurations. After installing the Geth client, it is necessary to create an account (and know its management procedures) as documented in section 4.3. Then, an introduction to the Geth parameters are detailed in Section 4.4.

4.2 Installing Geth

This sections covers different alternatives to install the Geth client on the OS platforms **Linux**, **MacOS**, **Windows** and **ARM**.

4.2.1 Installing from Binaries

Binaries are available on the official Ethereum website:

```
https://ethereum.github.io/go-ethereum/downloads/
```

Windows [3]: Geth installation is as simple as extracting *geth.exe* from your chosen OS. The download page provides an installer as well as a zip file. The installer puts Geth into your PATH automatically. The zip file contains the command .exe files and can be used without installing.

1. Download zip file
2. Extract geth.exe from the zip file
3. Open a command prompt (Windows key + R and type “cmd”)
4. Excute `geth.exe`

Allow private/public connections in the Windows Firewall for the Geth client (geth.exe). Control Panel -> System and Security -> Windows Defender Firewall -> Allowed apps Further configuration details are provided in [5].

4.2.2 Installing from Package Managers

- **Windows**:

```
https://github.com/ethereum/go-ethereum/wiki/Installation-instructions-for-Windows
```

Windows uses the package manager Chocolatey (http://chocolatey.org) to get the required build tools.

- **Ubuntu**

```
https://github.com/ethereum/go-ethereum/wiki/Installation-Instructions-for-Ubuntu
```

On Linux, installing Geth can be done using apt [3].

```
$ sudo apt-get install software-properties-common
```

```
$ sudo add-apt-repository -y ppa:ethereum/ethereum
$ sudo apt-get update
$ sudo apt-get install ethereum
```

- **MacOS**:

https://github.com/ethereum/go-ethereum/wiki/Installation-Instructions-for-Mac

Brew is recommended to install Geth on MacOS [3]:

```
$ brew update
$ brew upgrade
$ brew tap ethereum/ethereum
$ brew install ethereum
```

There is also an option to build from source code described at the main page:

https://github.com/ethereum/go-ethereum/wiki/Installing-Geth

- **ARM:**

Instructions for other enviroments (ARM, Android, etc):

https://github.com/ethereum/go-ethereum/wiki/Building-Ethereum

4.2.3 Tutorial Virtual Machine

If you do not want to install the Geth client, it is possible to run in inside a the tutorial VM (see below). Download VirtualBox for your OS:

https://www.virtualbox.org/wiki/Downloads

Then (after you downloaded VirtualBox), Connect to the `CNSM-BC-Tutorial` access point. The password is `CNSM18bctutorial`. The VM is available in the link below (alternatively, via USB pen-drive with the instructors).

http://172.10.15.30/install/bc-tutorial.zip

- **Import the VM**: Extract the `bc-tutorial.zip`. On the VirtualBox, click on `File → Import Appliance` and import the `bc-tutorial.ova`.
- **VirtualBox NAT Adapter**: It is necessary to create a NAT between your host OS and the VM on VirtualBox to allow external access. On the VirtualBox, click on `File → Preferences` then create a new NAT network. Then, adjust the VM to use the NAT adapter by clicking on the `VM → Settings → Network → Adapter "Attached to"` and select the NAT Network.
- **OS Firewall**: Remember to enable the "geth" application on your OS firewall and, if necessary, open TCP/UDP ports on 30303 (to synchronize blocks).

4.3 Managing Accounts

Typically, the first step after installing the client is to create an account. In the Geth client this is managed by the **account** command. Also, while creating an account is important to memorize the password, if the password to encrypt your account is lost, it will not be able to access that account again. Other commands are shown in Table 3.

```
$ geth account <command> [options...] [arguments...]
```

Table 3 - Commands for Accounts Management

Description	**Command**
Print summary of existing accounts	*$ list*
Create a new account	*$ new*
Update an existing account	*$ update*
Import a private key into a new account	*$ import*
Info about **subcommands**	*$ [command] --help*

These commands can be executed in both interactive and non-interactive modes. While in the interactive mode users are prompted to insert informations such as passwords, in the non-interactive mode informations can be supplied via a file. Therefore, the non-interactive mode is meant only for scripted use on test networks or known safe environments. For example, unlocking accounts can be performed both in the interactive and non-interactive modes:

IKeys are stored under the `<Datadir>/Keystore`. The keystore is the file that contains the wallet address and keys, which requires backup for safety reasons. To get the full path of the data directory on your specific system you can run the Geth console and type `admin.datadir` and you will see a result like on the screenshot above with the full path.

Example creating an account interactively:

```
$ geth account new
>Your new account is locked with a password. Please give a password. Do not forget
this password.
>Passphrase:
>Repeat Passphrase:
```

If necessary to check your accounts type:

```
$ geth account list
```

4.4 Geth Parameters

Options are generally used to define which blockchain will be used (*e.g.*, official chain, testnet, private chains) and adjust parameters and APIs for its management. In this

sense, these parameters are commonly defined at the moment of the initialization and synchronization of the blockchain.

- **Usage**:

```
$ geth [options] command [command options] [arguments...]
```

Table 4 - Geth Parameters

Description	**Command**
Data directory for the databases and keystore	*$ --datadir "path"*
Directory for the keystore (default = inside the datadir)	*$ --keystore*
Network identifier (integer, 1=Frontier, 2=Morden (disused), 3=Ropsten, 4=Rinkeby) (default: 1)	*$ --networkid value*
Ropsten network: pre-configured proof-of-work test network	*$ --testnet*
Rinkeby network: pre-configured proof-of-authority test network	*$ --rinkeby*
blockchain sync mode ("fast", "full", or "light")	*$ --syncmode "fast"*
Enable the HTTP-RPC server	*$ --rpc*
Change, respectively, the address and port of the HTTP server endpoint	*$ --rpcaddr value* *$ --rpcport value*
Ropsten network: pre-configured proof-of-work test network	*$ --testnet*
Rinkeby network: pre-configured proof-of-authority test network	*$ --rinkeby*
blockchain sync mode ("fast", "full", or "light")	*$ --syncmode "fast"*

Besides the officially exposed APIs, Geth provides the following extra management API namespaces which can be accessed interactively:

- **admin**: Geth node management. Gives access to several non-standard RPC methods, which will allow to have a fine grained control over the Geth instance, including but not limited to network peer and RPC endpoint management.
- **debug**: Geth node debugging. Provides access to several non-standard RPC methods, which allows to inspect, debug and set certain debugging flags during runtime.
- **miner**: Miner. Allows to remote control the node's mining operation and set various mining specific settings.
- personal: Account management. Manages private keys in the keystore.
- **txpool**: Transaction pool inspection. Grants access to several non-standard RPC methods to inspect the contents of the transaction pool containing all the currently pending transactions as well as the ones queued for future processing.

5. Private Blockchain

In this exercise you will connect to the tutorial private blockchain as client, participate in the mining process (gaining some Ethers), and exchange Ethers with other accounts.

5.1 Tutorial Wi-Fi Network

1. Connect to the "`BC-CNSM-Tutorial`" access point. The password is "`CNSM18bctutorial`".

2. On your browser, access server hosting all configuration files:

```
http://172.10.15.30
```

3. The server hosts an Etherpad (shared text editor) that allows to share account addresses. Access in your browser

```
http://172.10.15.30:9001/p/tutorial
```

5.2 Genesis Block

The first step is to download the genesis block. This block i.e., `genesis.json` is the file that defines the "settings" for your blockchain. For example, it determines the chain configuration, type of consensus mechanism (proof-of-work, stake, authority, etc), level of difficulty to mine blocks, accounts to start with a predefined amount of ethers and so on. In this tutorial, it is going to be used a proof-of-work blockchain. The genesis file has to be the same in all nodes of your private blockchain. Therefore, we will use the genesis configuration that is available at the link below:

1. Download the genesis file:

```
http://172.10.15.30/genesis.json
```

2. Remove the previous configuration (if exists):

```
$ geth removedb // confirm with "y"
```

3. Load the genesis file on your nodes type:

```
$ geth init genesis-pow.json
```

5.3 Static Nodes

Geth supports a declaration of nodes that you always want to be connected through the static nodes file, or adding nodes interactively in the Geth client.

1. Download the file from the server:

```
http://172.10.15.30/static-nodes.json
```

2. Copy the file to the `/.ethereum/geth` folder

3. Alternatively, it is possible to connect interactively via the Geth console using:

```
> admin.addPeer(“enode:IP:PORT”)
```

5.4 Connect to the Blockchain

The following Geth parameters are used to connect to the Tutorial Blockchain:

```
# syncmode 'full': will get the entire blockchain data, not only the blockheaders
such as in the "light" mode.
# rpc: enable rpc on port 8545 (default)
# mine: enable the "miner" api
# rpcapi: declares the available APIs in the geth client
# rpccorsdomain "*": open access to your rpc endpoint (default on localhost:8545).
This will enable access to remix to deploy contracts on your nodes (as well as any
other machine in your network)
# networkid: specifies the ID of your private blockchain, which is defined in the
genesis block.
# bootnodes: to connect in a bootstrap node (necessary for private blockchains)
```

1. Open a text editor and paste the following command:

```
geth --syncmode 'full' --rpc --mine --rpcapi 'personal,db,eth,net,web3,admin' --
rpccorsdomain "*" --networkid 15 --bootnodes “enode:IP:PORT” console
```

2. Adjust the `–botnodes “enode:IP:PORT”` using the following `enode`:

```
"enode://c44b7eccf767932eb588b837e1c50735752ccecb81e7040cebcd377168e822c167777d1ed
7709b8338ee15b1ac899cb2493c9fe632362342f87664d44d202a7a@172.10.15.30:30310"
```

3. Paste the command into a terminal (if using Windows use the “geth.exe”) and execute the adjusted command.
    a. **Windows**
        i. Press the key `windows key + r` to open the `run`
        ii. Type `cmd` to open a terminal
        iii. Find the location of the ‘geth.exe’
            1. **dir**: show directories
            2. **cd**: enter directories

5.5 Unlock your Account (and connect to the static node)

1. Unlock your account (in the Geth terminal)

```
> personal.unlockAccount(eth.accounts[0], “your-password”, 0)
```

2. If you do not have an account at this point, create one with the following command:

```
> personal.newAccount()
```

3. Check whether the account was created:

```
> personal.listAccounts
```

4. Make sure you are connected to the tutorial static node

```
>
admin.addPeer("enode://d68bc6efb6045b5d5c0246b806bb8b6da8a12c252e54a6d265b750ec8a1faacffb3011b6b382a89340c8f434fd60d1911f15dd1f1d93b5b6941ecfeda0e60853@[172.10.15.30]:30303")
```

5.6 Start to Mine

1. If necessary, set the Ether base to your account:

```
> personal.listAccounts
> miner.setEtherbase(eth.accounts[0])
```

2. Start the mining process (with 1 Thread initially) using the `mine` API:

```
> miner.start(1)
// more threads, i.e., mining power can be determined e.g., miner.start(24)
```

Wait the DAG (Directed Acyclic Graph) to be created. The DAG algorithm is used in the PoA algorithm of Ethereum to simultaneously achieve memory-hard computation but memory-easy validation.

5.6 Attach another Terminal to Geth

With the blockchain synchronization and the mining process in execution, it is recommended to open another terminal to execute commands. Thus, follow the step below:

1. Open another terminal and connect to the running Geth client by attaching to its RPC interface (the `-rpc` command opens by default the port 8545):

```
$ geth attach

// alternatively

$ geth attach localhost:8545
```

5.8 Share your Account Address

1. Check your account in another Geth terminal:

```
> personal.listAccounts
```

2. Paste your account address into the tutorial's Etherpad. On your browser access the following address:

```
http://172.10.15.30:9001/p/tutorial
```

To easily identify the account owner, write your name before your account's address "name – account".

5.9 Send Funds to other Accounts

1. Before you send funds, check the balance of your account:

```
> web3.fromWei(eth.getBalance(eth.accounts[0]), "ether")
```

2. This commands get the bance from the account "eth.accounts[0]" and converts to "ethers". Also, it is possible to create a variable to your account:

```
> var myaccount = eth.accounts[0]
> web3.fromWei(eth.getBalance(myaccount), "ether")
```

3. Similarly, create a variable with a destination account (account which you are going to send Ethers). Get the an account from your colleagues in the Etherpad.

```
> var receiver = "<address>"
> var amount = web3.toWei(0.25,"ether")
```

4. Send the transaction

```
> eth.sendTransaction({from: myaccount, to: receiver, value: amount})
```

5. Check your account's balance again

```
> web3.fromWei(eth.getBalance(eth.accounts[0]), "ether")
```

5.10 Create a Second Account

To prepare for the second part of the Tutorial (Smart Contracts), create a second account:

```
> personal.newAccount()
```

6. Smart Contracts

Blockchains can run code. While the first blockchains were designed to perform a small set of simple operations – mainly, transactions of a currency-like token – techniques have been developed to allow blockchains to perform more complex operations, defined in full-fledged programming languages.

Because these programs are run on a blockchain, they have unique characteristics compared to other types of software. First, the program itself is recorded *on* the blockchain, which gives it a blockchain's characteristic permanence and censorship resistance. Second, the program can *itself* control blockchain assets – i.e., it can store and transfer amounts of cryptocurrency. Third, the program is executed *by* the blockchain, meaning it will always execute as written and no one can interfere with its operation.

In Ethereum, Smart contracts are applications that run on the Ethereum Virtual Machine. This is a decentralized "world computer" where the computing power is provided by all those Ethereum nodes. Any nodes providing computing power are paid for that resource in Ether tokens. They're named smart contracts because you can write "contracts" that are automatically executed when the requirements are met.

For example, imagine building a Kickstarter-like crowdfunding service on top of Ethereum. Someone could set up an Ethereum smart contract that would pool money to be sent to someone else. The smart contract could be written to say that when $100,000 of currency is added to the pool, it will all be sent to the recipient. Or, if the $100,000 threshold hasn't been met within a month, all the currency will be sent back to the original holders of the currency. Of course, this would use Ether tokens instead of US dollars.

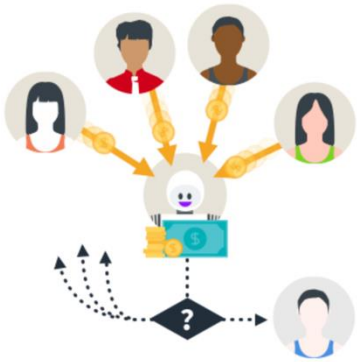


Figure 2 Crowdfunding with Smart Contracts

The Ethereum Foundation main page already contains three examples of smart contracts which can be built with Solidity:

- Create your own cryptocurrency with Ethereum (https://www.ethereum.org/token)
- Kickstart a project with a trustless crowdsale (https://www.ethereum.org/crowdsale)
- Create a democratic autonomous organization (https://www.ethereum.org/dao)

7. Gas

When you send tokens, interact with a contract, send ETH, or do anything else on the blockchain, you must pay for that computation. That payment is calculated in Gas and gas is paid in ETH. The creation of gas units is to separate the cost of computation work in the Ethereum network from Ethereum's volatile market price, as the cost of computation does not change rapidly.

Gas is a **unit of measuring the computational work** of running transactions or smart contracts in the Ethereum network. This system is similar to the use of kilowatts (kW) for measuring electricity in your house; the electricity you use isn't measured in dollars and cents but instead through kWH or Kilowatts per hour.

You are paying for the *computation*, regardless of whether your transaction succeeds or fails. Even if it fails, the miners must validate and execute your transaction *(compute)* and therefore you must pay for that computation just like you would pay for a successful transaction.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
G	200	Paid per byte for a CREATE operation to succeed in placing code into state.

Figure 3 Fee schedule containing the relative costs, in gas, of abstract operations that a transaction may effect

When you hear gas, the person is either talking about:

- Gas Limit: the maximum amount of units of gas you are willing to spend on a transaction
- Gas Price: the price you pay for each unit of gas, you can increase or decrease depending on how quickly your transaction should be mined

The total cost of a transaction (the "TX fee") is the `Gas Limit * Gas Price`.

8. Solidity

Solidity is a JavaScript like a language used to code smart contracts on the Ethereum platform. It compiles into a bytecode format that is understood by the Ethereum Virtual machine (EVM). It's a strongly typed language, supports inheritance, libraries and has the ability to define custom data structures. The best way to try out Solidity right now is using Remix.

This section should provide the essentials of what you need to know about Solidity to solve the exercises. If something is missing here or if you need further information about a topic, pleace conduct the official documentation at https://solidity.readthedocs.io.

8.1 A Simple Smart Contract

A contract in the sense of Solidity is a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

Let us begin with the most basic example. It is fine if you do not understand everything right now, we will go into more detail later.

```
1  pragma solidity ^0.4.24;
2
3  contract SimpleStorage {
4      uint256 storedData;
5
6      // Sets the value of storedData to the value of x
7      function set(uint256 x) public {
8          storedData = x;
9      }
10
11     /*
12      * Returns the value of storedData
13      */
14     function get() public view returns (uint256) {
15         return storedData;
16     }
17 }
```

Let's inspect above code line by line:

- Line 1: Tells that the source code is written for Solidity version 0.4.0 or anything newer that does not break functionality (up to, but not including, version 0.5.0). This is to ensure that the contract does not suddenly behave differently with a new compiler version.
- Line 2: Contracts in Solidity are similar to classes in object-oriented languages. Each contract can contain declarations of State Variables, Functions, Function Modifiers, Events, Struct Types and Enum Types. Furthermore, contracts can inherit from other contracts.
- Line 4: Declares a state variable called `storedData` of type `uint` (unsigned integer of 256 bits). You can think of it as a single slot in a database that can be queried and altered by calling functions of the code that manages the database.
- Line 6: This is a single-line comment
- Line 7 to 9: This function can be used to modify the value of the variable `storedData`.
- Line 11 to 13: This is a multi-line comment
- Line 14 to 16: This function can be used to retrieve the value of the variable `storedData`.

This contract does not do much yet apart from allowing anyone to store a single number that is accessible by anyone in the world without a (feasible) way to prevent you from publishing this number.

8.2 Version Pragma

Source files can (and should) be annotated with a so-called version pragma to reject being compiled with future compiler versions that might introduce incompatible changes.

```
pragma solidity ^0.4.24
```

Such a source file will not compile with a compiler earlier than version `0.4.24` and it will also not work on a compiler starting from version `0.5.0` (this second condition is added by using `^`). The idea behind this is that there will be no breaking changes until version `0.5.0`, so we can always be sure that our code will compile the way we intended it to.

8.3 Comments

Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible.

```
// This is a single-line comment

/*
This is
a multi-line comment.
*/
```

8.4 State Variables

State variables are values which are permanently stored in contract storage.

```
pragma solidity ^0.4.24;

contract SimpleStorage {
    uint storedData; // State variable
}
```

8.5 Types

Solidity is a statically typed language, which means that the type of each variable (state and local) needs to be specified (or at least known - see Type Deduction below) at compile-time. Solidity provides several elementary types which can be combined to form complex types.

```
// BOOLEAN VARIABLES
// The possible values are constants true and false
bool b = true;

// Operators
// !  (logical notation)
// && (logical conjunction, ‘and’) and || (logical disjunction, ‘or’)
// == (equality) and != (inequality)
bool c = !b // Set c to false

// INTEGERS
// Signed and unsigned integers of various sizes, in steps of 8
int8 i1 = 3;
uint256 i2 = 195;
```

```
// Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
uint 256 i3 = 200;
bool d = i2 > i3; // Evaluates to false

// Arithmetic operators: +, -, unary -, unary +, *, /, % (remainder),
// ** (exponentiation), << (left shift), >> (right shift)
uint256 i4 = i3 % i2; // i4 is 5


// ADDRESS
// Holds a 20 byte value (size of an Ethereum address)
address myAddress = 0x123;
myAddress.transfer(10);    // Send 10 Ether to myAddress

// STRING LITERALS
// String literals are written with either double or single-quotes
string foo = "foo"
string bar = 'bar'
```

8.6 Arrays

Arrays can have a compile-time fixed size or they can be dynamic. An array of fixed size k and element type `T` is written as `T[k]`, an array of dynamic size as `T[]`.

```
bytes32[5] nicknames; // static array
bytes32[] names; // dynamic array
uint newLength = names.push("John"); // adding returns new length of the array
```

8.7 Mappings

A mapping is referred to a hash table, which consists of a key type and a value type. We define a mapping like any other variable type.

```
mapping (string => uint) public balances;
balances["charles"] = 1;
console.log(balances["dave"]);    // is 0, all non-set key values return zeroes
console.log(balances["charles"]); // is 1
delete balances["charles"];       // resets the balance of "charles" to default
console.log(balances["charles"]); // is 0
delete balances;                  // sets all elements to 0
```

8.8 Units

A literal number can take a suffix of `wei`, `finney`, `szabo` or `ether` to convert between the subdenominations of Ether, where Ether currency numbers without a postfix are assumed to be Wei, e.g. `2 ether == 2000 finney` evaluates to `true`.

Note that `1 ether == 10**18 wei`, `1 szabo == 10**12 wei`, `1 finney == 10**15 wei`.

Suffixes like `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` after literal numbers can be used to convert between units of time where seconds are the base unit, e.g., `1 minutes == 60 seconds`.

8.9 Globally Available Variables

There are special variables and functions which always exist in the global namespace and are mainly used to provide information about the blockchain or are general-use utility functions.

```
block.number // Type: uint, returns the current block number
gasleft()    // Type: uint, returns the remaining gas
msg.data     // Type: bytes, returns complete calldata
msg.sender   // Type: address, returns the sender of the message (current call)
msg.value    // Type: uint, returns the number of wei sent with the message
```

8.10 Functions

Functions are the executable units of code within a contract. The function below is a payable function that accepts Ether.

```
function bid() public payable { // Payable Function
    // access the amount of Ether with msg.value
}
```

Functions neither declared `pure` nor `view` can read and modify the state.

```
uint256 counter;
function setCounter(uint256 _newValue) public {
    counter = _newValue;
}
```

There are also a few special function types that need further explanation.

8.10.1 View Functions

Functions can be declared `view` in which case they promise not to modify the state.

```
uint256 counter;
function getCounter() public view returns (uint256) {
    return counter;
}
```

8.10.2 Pure Functions

Functions can be declared `pure` in which case they promise not to read from or modify the state.

```
function multiply(uint256 a, uint256 b) public pure returns (uint256) {
    return a * b;
}
```

8.10.3 Fallback Function

A contract can have exactly one unnamed function. This function cannot have arguments and cannot return anything. It is executed on a call to the contract if none of the other functions match the given function identifier (or if no data was supplied at all).

Furthermore, this function is executed whenever the contract receives plain Ether (without data). Additionally, in order to receive Ether, the fallback function must be marked `payable`. If no such function exists, the contract cannot receive Ether through regular transactions.

```
function() public payable {
    // access Ether with msg.value
}
```

8.11 Visibility and Getters

Since Solidity knows two kinds of function calls (internal ones that do not create an actual EVM call (also called a “message call”) and external ones that do), there are four types of visibilities for functions and state variables.

Functions can be specified as being `external`, `public`, `internal` or `private`, where the default is `public`. For state variables, `external` is not possible and the default is `internal`.

- `external`: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works). External functions are sometimes more efficient when they receive large arrays of data.
- `public`: Public functions are part of the contract interface and can be either called internally or via messages. For public state variables, an automatic getter function is generated.
- `internal`: Those functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it), without using `this`.
- `private`: Private functions and state variables are only visible for the contract they are defined in and not in derived contracts.

```
function f(uint a) private returns (uint) { return a + 1; }
function g(uint a) public { data = a; }
function h() external returns (uint) { return data; }
function i(uint a, uint b) internal returns (uint) { return a + b; }
```

Note that everything that is inside a contract is visible to all external observers. Making something `private` only prevents other contracts from accessing and modifying the information, but it will still be visible to the whole world outside of the blockchain.

8.12 Constructors

A constructor is an optional function declared with the `constructor` keyword which is executed upon contract creation. Constructor functions can be either `public` or `internal`. If there is no constructor, the contract will assume the default constructor: `contructor() public {}`.

```
contract A {
    uint256 public a;

    constructor(uint256 _a) public {
        a = _a;
    }
}
```

8.13 Error handling: Assert and Require

Solidity uses state-reverting exceptions to handle errors. Such an exception will undo all changes made to the state in the current call (and all its sub-calls) and also flag an error to the caller. The convenience functions `assert` and `require` can be used to check for conditions and throw an exception if the condition is not met. The `assert` function should only be used to test for internal errors, and to check invariants. The `require` function should be used to ensure valid conditions, such as inputs, or contract state variables are met, or to validate return values from calls to external contracts. If used properly, analysis tools can evaluate your contract to identify the conditions and function calls which will reach a failing `assert`. Properly functioning code should never reach a failing assert statement; if this happens there is a bug in your contract which you should fix.

```
mapping (address => uint256) public balanceOf;

function transfer(address _from, address _to, uint256 _value) public {
    // Check if the sender has sufficient funds
    require(balanceOf[_from] >= value);

    uint256 previousBalances = balanceOf[_from] + balanceOf[_to];

    balanceOf[_from] -= _value; // Subtract from the sender
    balanceOf[_to] += _value;   // Add to the recipient

    // Asserts are used for static analysis to find bugs in your code.
    // They should NEVER fail.
    assert(balance[_from] + balance[_to] == previousBalances);
}
```

Note that this function should not be used in production since a few more checks are needed to guarantee a secure transfer.

9. Smart Contracts in Practice

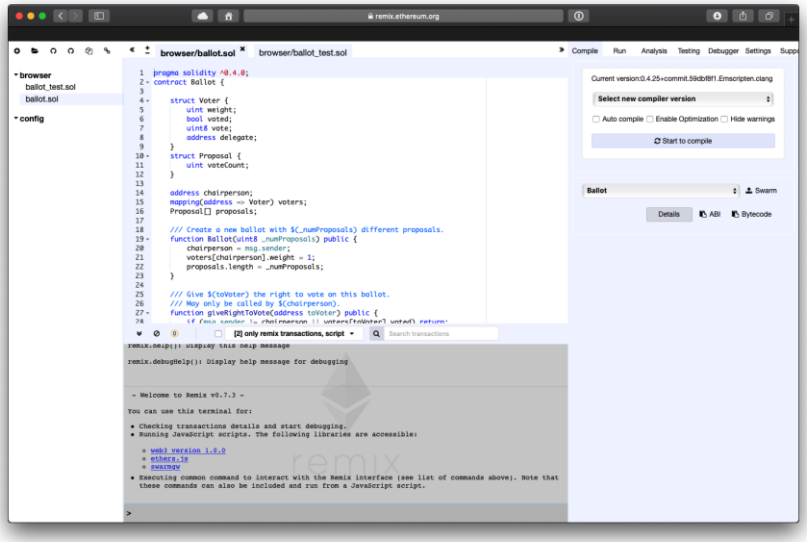
Now that we have a brief understanding of Solidity and how smart contracts work, let's put it into practice.

9.1 Setting up our development environment

This part focuses on using Remix IDE, which is a browser based smart contract IDE. Remix is a good solution if you intend to:

- develop smart contracts (Remix integrates a Solidity editor.)
- debug smart-contract execution
- access the state and properties of a previously-deployed smart contract
- debug a previously-executed transaction
- analyze solidity code to reduce coding mistakes and enforce best practices

The goal of this exercise is to make yourself comfortable with Remix Web IDE and the process of deploying a smart contract and interacting with it. Opening http://remix.ethereum.org/, you should see something like this:



Here's what you're looking at:

- On the left: **File Explorer**. The file explorer lists by default all the files stored in your browser. You can see them in the browser folder. You can always rename, remove or add new files to the file explorer. Note that clearing the browser storage will permanently delete all the solidity files you wrote.
- In the upper middle: **Editor**. The Remix editor recompiles the code each time the current file is changed or another file is selected. It also provides syntax highlighting mapped to solidity keywords and displays opened files as tabs.
- In the lower middle: **Terminal**. It integrates a JavaScript interpreter and the `web3` object. It enables the execution of the JavaScript script which interacts with the current context. It displays important actions made while interacting with the Remix IDE (i.e. sending a new transaction). It also allows searching for the data and clearing the logs from the terminal.
- On the right:
  - **Compile**: Let's you set the compiler version and shows warnings and errors in your code.
  - **Run**: The Run tab is an important section of Remix. It allows you to send transactions to the current environment.
  - **Analysis**: The analysis tab gives detailed information about the contract code. It can help you avoid code mistakes and to enforce best practices.
  - **Debug**: This tab allows you to debug the transaction. It can be used to deploy transactions created from Remix and already mined transactions. (debugging works only if the current environment provides the necessary features).

1: We will set the compiler to "Auto-Compile". This way, Remix triggers a compilation each time the current file is changed or another file is selected. On the right, in the Compile tab, mark the checkbox "Auto-Compile" so that it is ticked.

2: In the Run tab on the right side, choose “Web3 Provider” as your environment. Confirm the suggested endpoint http://localhost:8545.

9.2 Deploying our first contract

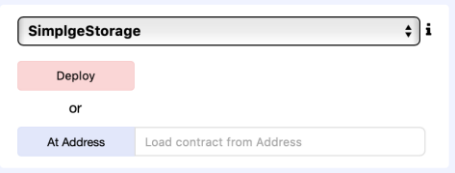
In this exercise we will deploy our first smart contract to our private Ethereum blockchain.

1: Using Remix, create a new file in the file explorer by pressing the +-Button in the upper left corner. Call the file `SimpleStorage.sol`.

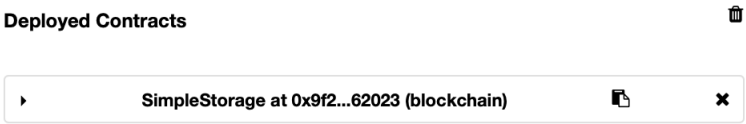
2: In the editor, type in the following contract:

```
pragma solidity ^0.4.24;

contract SimpleStorage {
    uint256 storedData;
}
```

3: Switch to the “Run” tab. Make sure your contract is compiled. The name of the contract should be displayed above the red “Deploy” button. Press the red button to deploy the contract on your private blockchain.



4: It takes a few seconds until its deployed. Once deployed, you should see your contract in the section “Deployed Contracts” (still in the Run tab).



5: You can check the deployment state of your contract in the terminal of Remix. What you also can see is the address, the transaction cost, the logs, etc.

This contract does not do much and was just to make yourself comfortable with writing and deploying a contract on the blockchain.

9.3 Creating your own bank

In this exercise, we will create our own bank that can hold balances of our clients.

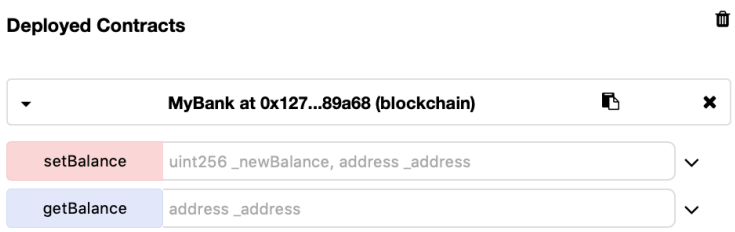
Our code looks like this:

```
pragma solidity ^0.4.24;

contract MyBank {
    // The key is of type address (storing the client's address)
    // The value is of type uint256 (storing the client's balance)
    mapping(address => uint256) private balances;

    // Returns the balance of client with address "_address"
    function getBalance(address _address) view public returns (uint256) {
        return balances[_address];
    }

    // Sets the balance of client with address "_address"
    function setBalance(uint256 _newBalance, address _address) public {
        balances[_address] = _newBalance;
    }
}
```

Let's first examine this contract. As always, we first define our `pragma` to reject being compiled with past compiler versions below `0.4.24` and future compiler versions above `0.5.0`. Next, we create a contract called `MyBank` that holds our variables and functions. Our contract holds a mapping that maps from client's address to client's balance and the two functions are required to get or set the balance by a client's address.

Note that the `getBalance` function is not necessary if you would set the mapping to `public`. While not shown in the editor, a getter-Function would be created automatically.

1: Create a new file in the file explorer called `MyBank.sol`. Write down above contract into the editor and deploy the contract the same way like before. After the deployment, Remix should show you both functions in the "Deployed Contracts" section.



2: In the following exercises, we need our contract address. Recall that you can easily query your wallet's address by running the following command in geth:

```
> personal.listAccounts
["0x59a6e02da587bfe3776d7c1f35a9c837cc2070bc",
"0xd2df134efc0dec93da3344e7f5b565a366e52577"]
```

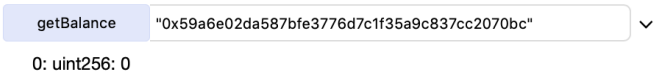
If you have more than one account, the command will show you more addresses.

Also make sure that you have unlocked both of your accounts, otherwise you will get a `authentication needed: password or unlock` error. You can unlock both of your accounts for an indefinitie time by the following geth-commands (replace `"password"` with the password you set before):

```
> personal.unlockAccount(eth.accounts[0], "password", 0)
> personal.unlockAccount(eth.accounts[1], "password", 0)
```

You can also copy your address in the "Run"-tab by clicking the following symbol:

3: Check what your balance is by entering your wallet address into the getBalance field. Make sure that you wrap your address in double quotes. Press the blue button "getBalance", which should return zero.



4: Next, set your balance to 10 by first entering the value to set followed by your wallet address.

5: Last, check your balance again to see if it worked.

6: Great, it worked! However, your Smart Contract has a trivial vulnerability: Look closely in the `setBalance` method. Everbody can get and set each other's balance. We will change this immediately such that only you can get your balance and only the owner of the bank can set balances.

7: We will start with the `getBalance` method. Our goal is that only you, as the caller of the function, can get your balance. We use the globally available variable `msg.sender`, which holds caller's address of the function.

```
// Returns the balance of the caller of the method
function getBalance() view public returns (uint256) {
    return balances[msg.sender];
}
```

8: We have to make sure that only the owner of the bank, i.e., the owner of the contract, can set balances of clients. Our goal is to restrict who can make modifications to your contract's state or call your contract's functions. We do this by setting owner on contract creation, i.e., we set the owner in the constructor of the contract, since the constructor is executed only once during the whole lifetime.

```
// The owner of the bank
address owner;

constructor() public {
    owner = msg.sender; // The current owner is your own account.
}
```

Then we are able to authorize the accessing user of the `setBalance` method by requiring the caller of the function to equal to the owner of the contract. If this is not the case, the call is unauthorized and the transaction is aborted.

```
// Sets the balance of the caller of the address
function setBalance(uint256 _newBalance, address _address) public {
    require(msg.sender == owner, "Unauthorized");
    balances[_address] = _newBalance;
}
```

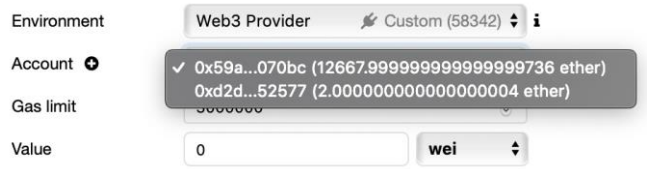
9: In the end, our *secure* bank looks like this:

```
pragma solidity ^0.4.24;

contract MySecureBank {
    // The key is of type address (storing the client's address)
    // The value is of type uint256 (storing the client's balance)
    mapping(address => uint256) private balances;

    // The owner of the bank
    address owner;

    constructor() public {
        owner = msg.sender; // The current owner is your own account.
    }

    // Returns the balance of client with address "_address"
    function getBalance() view public returns (uint256) {
        return balances[msg.sender];
    }

    // Sets the balance of client with address "_address"
    function setBalance(uint256 _newBalance, address _address) public {
        require(msg.sender == owner, "Unauthorized");
        balances[_address] = _newBalance;
    }

    // Allows the current owner to transfer control of the contract to a new owner
    function transferOwnership(address _newOwner) public {
        require(msg.sender == owner, "Unauthorized");
        owner = _newOwner;
    }
```

```
}
```

10: Deploy your secure bank on the blockchain and get your current balance, which should be zero. Switch to your second wallet and check your balance there, which should be zero too.

You can change accounts in the upper right of the "Run" tab.



11: Using your second account, try to set the balance of yourself. Make sure you use the correct address, i.e., the address of the second account. Also make sure that you have enough Ether in your second account to send a transaction to the network. Otherwise, transfer some Ether from your first to your second account with the following command:

```
eth.sendTransaction({from: eth.accounts[0], to: eth.accounts[1], value: web3.toWei(2, "ether")})
```

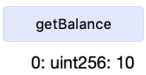
Trying to set your balance should indeed fail. In the logs, you can see the following status of your transaction:

```
0x0 Transaction mined but execution failed
```

This just tells us that our code worked fine and failed when an unauthorized user tried to set the balance of somebody.

12: Switch back to your main account. Since your main account is the owner of the contract and thus the owner of the bank, you can set the balance of everyone. Set the balance of your second wallet to 10. Again, make sure you use the correct address.

13: Now if you switch back to your second account and invoke the `getBalance` method, your balance should be set to 10.



9.4 Creating our own crypto-currency

We are going to create a digital token. Tokens in the Ethereum ecosystem can represent any fungible tradable good: coins, loyalty points, gold certificates, IOUs, in-game items, etc. Since all tokens implement some basic features in a standard way, this also means that your token will be instantly compatible with the Ethereum wallet and any other client or contract that uses the same standards.

The standard ERC20 token contract can be quite complex. But in essence a very basic token boils down to this:

```
contract ERC20 {
    // Public variables of the token
    string public constant name = "Your Token Name";
    string public constant symbol = "YTN";

    // Creates an array with all balances
    mapping (address => uin256) public balanceOf;

    // Holds the total supply of our token
    uint256 public totalSupply;

    // Send coins
    function transfer(address _to, uint256 _value) public returns (bool success) {
        // Code
    }

    // Generates a public event on the blockchain that will notify clients
    event Transfer(address indexed _from, address indexed _to, uint256 _value);
}
```

Your task is now to implement your own ERC20 token, deploy it on your private blockchain, and send exchange tokens between two or more accounts.

In addition to the ERC20 token, include a function which let's the owner of the contract increase the totalSupply. The additional supply should be added to the owner's account. Don't forget to include the possibility to transfer the ownership of the contract.

You can find the solution in the appendix.

10. References

[1] W. Karl, A. Gervais. (2017). Do you need a Blockchain? IACR Cryptology ePrint Archive: 375.
[2] G. Greenspan. (2015). Avoiding the Pointless Blockchain Project. URL: https://www.multichain.com/blog/2015/11/avoiding-pointless-blockchain-project/

[3] Nethereum. Installation and Configuration of the Ethereum client Geth. URL: https://nethereum.readthedocs.io/en/latest/ethereum-and-clients/geth/#installation-and-configuration-of-the-ethereum-client-geth

[4] Salanfe. Setup Your Own Private Proof-of-Authority Ethereum Network with Geth. URL: https://hackernoon.com/setup-your-own-private-proof-of-authority-ethereum-network-with-geth-9a0a3750cda8

[5] Easy Eth. How to Install Geth in Windows. URL> https://www.easyeth.com/how-to-install-geth-in-windows.html

[6] Coindesk. Making Sense of Smart Contract. URL: https://www.coindesk.com/making-sense-smart-contracts/

[7] Ethereum Foundation. Solidity Documentation. URL: https://solidity.readthedocs.io/en/v0.4.24/index.html

[8] Ethereum Foundation. Remix Documentation. URL: https://remix.readthedocs.io/en/latest/

[9] MyEtherWallet. What is Gas in Ethereum? URL: https://myetherwallet.github.io/knowledge-base/gas/what-is-gas-ethereum.html.

[10] Thomas Bocek, Burkhard Stiller: Smart Contracts — Blockchains in the Wings; in: Claudia Linnhoff-Popien, Ralf Schneider, Michael Zaddach (Edts.), “Digital Marketplaces Unleashed”, Springer, Heidelberg, Germany, January 2017, ISBN 978-3-662-49274-1, pp 169–184, https://link.springer.com/chapter/10.1007/978-3-662-49275-8_19.

[11] Sina Rafati, Florian Schüpfer, Thomas Bocek, Burkhard Stiller: A Peer-to-peer Purchase and Rental Smart Contract-based Application (PuRSCA); it — Information Technology, De Gruyter, Vol. 36, No. 7, October 2018, ISSN 2196-7032, pp 1–14. doi:10.1515/itit-2017-0036, https://www.degruyter.com/view/j/itit.ahead-of-print/itit-2017-0036/itit-2017-0036.xml.

[11] Bruno Rodrigues, Thomas Bocek, Burkhard Stiller: The Use of Blockchains: Application-Driven Analysis of Applicability; in: Pethuru Raj, Ganesh Deka (Edts.), “Blockchain Technology: Platforms, Tools and Use Cases, Volume 111 (Advances in Computers)”, Springer, Waltham, Massachusetts, U.S.A., No. 111, September 2018, ISBN 978-0-128-13852-6, pp 163–198, https://www.sciencedirect.com/science/article/pii/S006524581830024X

11. Appendix

An example of an ERC20 token (including the additional requirement to be able to raise the total supply) could look like this:

```
pragma solidity ^0.4.24;

contract ERC20 {
    // Public variables of the token
    string public constant name = "Your Token Name";
    string public constant symbol = "YTN";

    // Creates an array with all balances
    mapping (address => uint256) public balanceOf;

    // Holds the total supply of our token
    uint256 public totalSupply = 1000;

    // The owner of the contract
    address owner;

    // Generates a public event on the blockchain that will notify clients
    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    // The constructor is called at contract creation and only once
    constructor() public {
        // The creator of the contract is the owner
        owner = msg.sender;
        // Add the total supply to the owner of the contract
        balanceOf[msg.sender] = totalSupply;
    }

    // Send coins
    function transfer(address _to, uint256 _value) public returns (bool success) {
        // Prevent transfer to 0x0 address
        require(_to != 0x0);
        // Check if sender has sufficient funds
        require(balanceOf[msg.sender] >= _value);
        // Save this for an assertion in the future
        uint256 previousBalances = balanceOf[msg.sender] + balanceOf[_to];
        // Subtract from the sender
        balanceOf[msg.sender] -= _value;
        // Add the same to the recipient
        balanceOf[_to] += _value;
        // Emit the event about transferred value
        emit Transfer(msg.sender, _to, _value);
        // Assert that the total involved amount stayed the same
        assert(balanceOf[msg.sender] + balanceOf[_to] == previousBalances);
        return true;
    }

    // Raise the total supply
    function raiseTotalSupply(uint256 _amount) public {
        // Authorization check
        require(msg.sender == owner, "Unauthorized");
        // Check for overflows
        require(totalSupply + _amount >= totalSupply, "Overflow detected");
        // Check if amount is not negative
```

```
        require(_amount > 0, "Total supply can only be raised");
        // Add the amount to the total supply
        totalSupply += _amount;
        // Add the amount to the balance of the owner
        balanceOf[owner] += _amount;
    }

    // Transfer the ownership of this token
    function transferOwnership(address _newOwner) public {
        // Authorization check
        require(msg.sender == owner, "Unauthorized");
        // Replace the old owner with the new owner
        owner = _newOwner;
    }
}
```