BLOCKCHAIN
TRAINING ALLIANCE

Thomas Wiesner, Ernesto Lee

Certified Blockchain Developer Ethereum (CBDE)

Official Exam Study Guide

This exam study guide covers exam objectives, including how to plan and prepare production ready applications for the Ethereum blockchain. The guide includes how to use the essential tooling and systems needed to work with the Ethereum ecosystem.

Exam guide includes 100+ Exam Practice Questions

Certified Blockchain Developer Ethereum (CBDE), Official Exam Study Guide
By: Thomas Wiesner, Ernesto Lee

Book is published by Blockchain Training Alliance, Inc.

Copyright © 2018

Limit of Liability / Disclaimer of Warranty / Terms of Use

Table of Contents

Chapter 1: Ethereum Basics

Introduction

In this chapter we are covering all the basics for Ethereum and the Ethereum Virtual Machine (EVM).

What are "DApps"?

Distributed Applications, or simply "DApps", are applications running directly on the blockchain. This way they are decentralized. Their purpose is multi-fold, reaching from simple logic for p2p value transfers to complicated token structures and much more.

Because the economics and the logic are on the same layer it makes value transfer extremely easy. Additionally, you can save hardware and configuration by deploying the logic directly on the blockchain.

Ethereum and EVM Basics

The EVM is basically a sandboxed virtual machine running on every single node. It is Turing complete and a transaction-based state machine. The nodes reach consensus by executing all transactions. Only the miner node gets the block reward, all other nodes are just checking if the miner was "honest".

The blockchain is also deterministic, which means: The same input will produce the same output.

Hashing

With hashing you can map data of arbitrary size to data of fixed size. There is a plethora of hash functions available, but Ethereum uses the so-called Keccak-256, which is internally almost identical to SHA256, but differs slightly on the input padding, so they are not interchangeable.

Previous to the current Keccak-256 hashing, a hybrid method called Dagger-Hashimoto was used that combined the DAG approach to generate data and used the approach of hash, shift and modulo of Hashimoto. This Dagger-Hashimoto method has evolved into what we now use as Ethash which is the proof-of-work algorithm used in Ethereum.

Hashing is deterministic: Same input produces the same output.

Ethereum Denominations

In Ethereum several units are important. 1 Ether is probably the most well-known denomination on the Ethereum blockchain. 1 Ether is 10^{18} Wei, which is the smallest unit. But in between there are other denominations.

Here is a whole list of the important units on the Ethereum Blockchain:

1 Ether = 1000000000000000000 Wei (10^18)
1 Ether = 1000000000000000 Kwei, Ada, Femtoether (10^15)
1 Ether = 1000000000000 Mwei, Babbage, Picoether (10^12)
1 Ether = 1000000000 Gwei, Shannon, Nanoether, Nano (10^9)
1 Ether = 1000000 Szabo, Microether, Micro (10^6)
1 Ether = 1000 Finney, Milliether, Milli (10^3)
1 Ether = 0.001 Kether, Grand, Einstein (10^-3)
1 Ether = 0.000001 Mether (10^-6)
1 Ether = 0.000000001 Gether (10^-9)
1 Ether = 0.000000000001 Tether (10^-12)

Most widely used are Wei, Gwei, Finney and Ether. With the tool https://etherconverter.online/ you can easily convert different units.

Private Keys, Public Keys and Accounts

An Ethereum address is bound to a private key. The private key is nothing other than a 64 character (32 bytes) long hex value. From this private key the public key is derived and from the public key the address.

Proof of Work (PoW) vs. Proof of Stake (PoS) vs. Proof of Authority (PoA)

In a proof of work environment all mining nodes are trying to find the solution to a mathematical riddle. Basically, the input parameters to a hashed output has to be found. They do this in a "brute force" way by continuously calculating the hash. This takes a lot of computing power and electricity.

In a proof of stake environment some stakeholders lock-in some funds and are randomly chosen to mine and seal a new block. This way little electricity is spent, and mining doesn't need a lot of computer power.

In both networks, PoW and PoS, basically anyone can become a miner. In a proof of authority network some nodes are pre-selected and become the miner nodes.

Externally Owned Accounts (EOA) vs. Smart Contracts

In Ethereum we are talking about two different types of accounts. One type is controlled by a private key – or an external person. This is a so-called Externally Owned Account: an account which is controlled by a private key.

When a smart contract gets deployed it then becomes its own address on the blockchain. This way an address is self-governing through the code. This is now a smart contract and not controlled through a private key from the outside world.

Transactions

If you send off a transaction, then several fields have to be set. These include:

- nonce: It is a sequence number for the sending account which counteracts replay attacks
- gasprice: price offered to pay per gas
- startgas: upper limit for the gas consumption
- to: destination address (EoA or contract address)
- value: Ether to transfer
- data: Data to transfer
- v, r, s: ECDA signature

Chapter 1 Quiz

1. Which is the right order for Denominations?
a. Wei, Finney, Szabo, Ether, Tether
b. Finney, Szabo, Mether, Gwei
c. Gwei, Szabo, Finney, Ether

2. What is the nonce-field in a transaction?
a. To protect against replay attacks
b. To have an additional checksum for transactions
c. To sum up all ethers sent from that address

3. Select which statement is true about the EVM
a. While the EVM is Sandboxed, it isn't as powerful as the Bitcoin Network, because it's not Turing Complete
b. The EVM can't access hardware layers or anything outside a blockchain node because it's sandboxed
c. The EVM is extremely powerful, turing complete and perfect for doing computational intensive things, because of the direct access to the graphics card.

4. DApps are…
a. Great, because they cutout the middle man, run on a trusted platform, apply logic to the blockchain where already economic assets are running and thus allow peer to peer trade.
b. An amazing way to create new applications. Those applications run entirely separated from other applications on the platform and allow for logical interactions. To add an additional layer of trust they can't access any funds.
c. A new way of applying logical operations for banks and big financial institutions. This way they can reduce staff while operating at increased security.

5. To get most out of the blockchain it is best...
a. To use it for all business logic. It's always best to have everything in once place.
b. To use it only for those things which need the benefits of the blockchain

6. What does a Hashing Algorithm is deterministic mean?
a. Given the same input it always produces the same output
b. Given a long input it uses equally distributed data to produce the output
c. Given the output it shouldn't be possible to re-generate the input

7. What's the correct scientific notation?
a. 1 Ether = 10^{18} wei, 10^9 Gwei, 10^3 Finney
b. 1 Ether = 10^{19} wei, 10^{13} Gwei, 10^3 Finney
c. 1 Ether = 10^{16} wei, 10^{13} Gwei, 10^3 Finney
d. 1 Ether = 10^{18} wei, 10^6 Gwei, 10^6 Finney

8. What Are Private Keys for?
a. To Protect the Public Keys by being cryptographically significant
b. To Sign Transactions and to derive a public key from which an address is determined
c. To Generate an Address which can sign transactions

9. Public Keys vs. Private Keys, select which is true:
a. The Public Key is for Signing Transactions, the Private Key must be given out to verify the signature
b. The Private Key signs transactions, the public key can verify the signature
c. The Private Key is to generate a public key. The public key can sign transactions, the address is here to verify the transactions

10. Proof of Work (PoW) vs. Proof of Stake.
a. PoW is computationally intensive which requires lots of energy. On the other hand, miners earn straightforward a reward for mining a block and incorporating transactions.
b. PoW is better than PoS, because with PoS we increase the amount of energy spent on the network.
c. PoS is mining with specialized new hardware that has to be purchased with a stack of Ether in the network. Hence the Name: Proof of Stake, which derives from Stack.

11. Externally Owned Accounts

a. Can be destroyed using the selfdestruct keyword. This way all remaining ether will be sent to the receiver address, regardless if they have a fallback function or not.

b. Are bound to a private key which is necessary to sign transactions outgoing from that account

c. Are logical opcodes running on the ethereum blockchain very similar to smart contracts.

12. Transactions

a. Transactions containing the same data to create the same smart contract always have the same signature

b. Transactions containing the same data to create the same smart contract are having a different signature because of the nonce which changes upon every transaction

13. Sending one Ether

a. Is internally translated to Wei, so it will send the equivalent of 10^{18} Wei

b. Is internally translated to Finney, so it will send the equivalent of 10^3 Finney

c. Is internally translated to Szabo, so it will send the equivalent of 10^6 Szabo

14. Hashing

a. Mining uses Keccack256 while internally to hash values it's easy to use the Dagger-Hashimoto to create a meaningful hash

b. Mining uses the Dagger-Hashimoto hashing while internally the EVM uses SHA256 which is an alias for Keccack256

c. Mining uses the Dagger-Hashimoto hashing while internally the EVM uses Keccack256 which is similar to SHA256, but has different padding so it produces different hashes

15. PoS

a. PoS would be better, because it can reduce the amount of energy needed for mining

b. PoS would be worse, because it would increase the amount of energy needed for mining

16. EoA

a. Externally Owned Accounts change their address every time a Transaction is sent because of the nonce

b. Externally Owned Accounts keep their address, but on the blockchain a nonce is increased every time they send a transaction to avoid replay attacks

Solution for the Quiz:

Q1: C	Q7: A	Q12: B
Q2: A	Q8: B	Q13: A
Q3: B	Q9: B	Q14: C
Q4: A	Q10: A	Q15: A
Q5: B	Q11: B	Q16: B
Q6: A		

Chapter 2: Ethereum Nodes and Networks

Ethereum Protocol

Ethereum itself is only a protocol defining how the communication should work. It is neither a proprietary software, nor patent. Instead it is open and there are several different implementations of the Ethereum protocol.

Two of the most popular implementations are "Go-Ethereum" or simply "GETH", written in GO and the other one is "Parity" written in Rust.

Communication of Ethereum Nodes with clients

Ethereum nodes communicate with each other using the Ethereum protocol. But, as with MySQL for example, you can also connect to Ethereum nodes from the outside world. There are different ways to connect to an Ethereum node.

Usually you can, at least, connect via HTTP and IPC. Sometimes also WebSockets connection are supported. With HTTP, for example, the Ethereum node accepts requests in a **JSON-RPC** format. This is a standardized way of communicating with Ethereum nodes from clients.

This way, any software that implements the JSON-RPC calls, can connect to the blockchain via an Ethereum node. In other words: Every Ethereum node must implement also a JSON-RPC interface.

Blockchain Networks

In Ethereum a blockchain is defined through a genesis block (which is the first block) and every block thereafter through mining. Every time a new block is mined it contains a part of the previous block signature and the transactions that were mined through the miner.

In Ethereum everyone can start their own blockchain based on the Ethereum protocol. This means that the Main-Net can be replicated in a way so that it works exactly the same way but doesn't cost any real money.

These networks are usually Test-Networks which are very similar to the Main-Network. They are either a replication to test distributed Applications, or they are created to test new network functionalities, such as Proof of Stake.

As happens often with cutting edge software, there is no official list of Network-Ids and Network Names, but in a stack-overflow thread an overview is given:

- 0: Olympic, Ethereum public pre-release testnet
- 1: Frontier, Homestead, Metropolis, the Ethereum public main network
- 1: Classic, the (un)forked public Ethereum Classic main network, chain ID 61
- 1: Expanse, an alternative Ethereum implementation, chain ID 2
- 2: Morden, the public Ethereum testnet, now Ethereum Classic testnet
- 3: Ropsten, the public cross-client Ethereum testnet
- 4: Rinkeby, the public Geth PoA testnet
- 8: Ubiq, the public Gubiq main network with flux difficulty chain ID 8
- 42: Kovan, the public Parity PoA testnet
- 77: Sokol, the public POA Network testnet
- 99: Core, the public POA Network main network
- 7762959: Musicoin, the music blockchain

Souce: https://ethereum.stackexchange.com/questions/17051/how-to-select-a-network-id-or-is-there-a-list-of-network-ids/17101#17101

Blockchain Nodes and Mode of Operation

There are several redundant implementations of the Ethereum protocol to ensure the correctness of the implementation. Additionally, not all blockchain nodes operate the same way. Some are purely for developing and hold a blockchain in-memory and just simulate the mining.

Real Blockchain Nodes:

1. Cpp-ethereum
2. Go-Etheruem (GETH)
3. Parity

In-Memory Blockchain simulations for rapid development:

1. TestRPC
2. Ganache
3. Truffle Developer Console

Clients to access the blockchain in a convenient way:

1. MetaMask browser Plugin through Infura
2. Status.IM Android/iOS app through Infura
3. MIST DApp Browser with integrated GETH

Private, Consortium and Public Networks

A Private Network is a network that is not running publicly. This means access can be more tightly controlled. Writing to the blockchain is also usually limited to one or a few nodes. Not everybody can join.

Consortium Networks are very similar to private networks, but certain parts might be public. Consensus and writing is usually still restricted to several pre-defined nodes in the network. It's also called a "partially decentralized" system.

The typical Main-Net is a Public Network. Everybody can join. Everybody can become a miner. And Everybody can send transactions.

How consensus works

If you send a transaction to the network, then one miner-node will at some point pick it up. The Miner will run the transaction and add the result to the next block.

Now, this doesn't imply consensus yet. By design all nodes don't trust each other. Each node must verify that the transaction the miner added to the block is really valid.

This means, consensus is reached by having every node running the same transactions again and verifying that the result is correct. Plus, the results are verified in a cryptographic manner.

A block on the blockchain

At the time of writing these lines, the Ethereum Blockchain still runs on Proof of Work. When a block is mined, the miner node selects some transactions from a pool of pending transactions. Usually they are sorted by how much gas they would bring in.

These transactions are executed and incorporated in the new block.

But a block also contains two very important parameters: a difficulty and a timestamp. The difficulty regulates how hard it is to find a block by the miner. The mining time is set to be between 10 and 20 seconds. If it's beyond 20 seconds, the difficulty is too high and will be automatically lowered going forward. If the mining happens below 10 seconds, then the difficulty increases.

The timestamp is the time when a miner found the block. It is not automatically derived, rather it is set by the miner itself and can thus be influenced to a certain degree. The timestamp does not depend on the time zone, as it's the standard Unix timestamp.

Functions of a Miner

There are some functions of a miner during mining to keep the blockchain "alive". Let's assume the miner found the next block, so here is what will happen:

1. It will listen to transactions and include as much pending transactions as it can in the block
2. It will determine and include uncles (ommers)
3. It will update the account-balance of the coinbase/etherbase, which is the account set to be rewarded from the mining

Chapter 2 Quiz

1. What are Ethereum Nodes?
a) Programs implementing the Ethereum Protocol to talk to each other and JSON-RPC interfaces to talk to the outside world
b) A Java-Script library to compile and run Solidity Code
c) A Framework for deploying and running smart contract in a decentralized way

2. To communicate with an Ethereum node via JavaScript
a) The library you use should make use of the JSON-RPC Interface of an Ethereum Node
b) Must Implement the Ethereum Protocol to connect to other Ethereum Nodes
c) Must use Web3.js, which is closed source to communicate to other Ethereum Nodes

3. It's possible to access the blockchain via an Ethereum Node
a) Only via JavaScript because there is the proprietary Web3.js library
b) by any programming language, IPC or Web Socket, if it adheres to the JSON-RPC standard

4. A Private Network is
a) A side Channel to the Ethereum Main Net which costs less gas to run smart contracts
b) An exact clone of the Rinkeby Test-Network which can be started as virtual machine in the Azure Cloud
c) A Network running only in a private area, where people cannot join freely and openly

5. For Rapid Development Cycles it's good
a) To deploy to the main-network as quickly as possible
b) To use in-memory blockchain simulations, because mining works instantaneously
c) To use a private network at all times, because this is the closest you get to the real network

6. Go-Ethereum vs. Ganache
a) Both are the same, just implemented in a different language
b) With Go-Ethereum you get a real blockchain node where you can create your own local private network, connect to Test-Networks or the Main-Net, while with Ganache you get an in-memory blockchain simulation
c) With Ganache you get a real blockchain node where you can connect to the Test-Networks Rinkeby and Ropsten

7. Topic: Block Timestamp
a) Because the timestamp is based on the time zone of the miner it changes the difficulty continuously to reflect network latency.
b) The timestamp can't be influenced by a miner and is generally considered safe to be used for randomness on the blockchain.
c) The timestamp can be influenced by a miner to a certain degree but it's always independent from the time-zone.

8. Block Difficulty

a) The Block Difficulty is determined by the Ethereum Committee every fortnight to reflect the average amount of transaction and it cannot be influenced by the network itself.

b) The Block Difficulty increases when the time between mined blocks is below 10 seconds, while it decreases when the time is above 20 seconds.

c) The Block Difficulty increases when the time between mined blocks is below 20 seconds, while it decreases when the tie is above 60 seconds

9. Ethereum Nodes

a) Must implement the Ethereum protocol and external access can only be done via the proprietary Ethereum Libraries like Web3.js

b) Must Implement the Ethereum Protocol and a JSON-RPC to talk with clients

c) Must implement Web3.js to interact with Websites

10. When a new block is mined

a) A list of transactions as well as uncles is incorporated in the block. All gas that is used during those transactions is added to the miners' balance. Also, the block reward is added to the miner. Then the same transactions are run again by every participating node in the network to achieve consensus.

b) A list of transactions is incorporated in that block. Gas used during the execution is attached to the executing contracts while the block reward is automatically spread across the mining pool to ensure a fair spread. Consensus is reached by a special form of hash code.

11. A Blockchain Node

a) Can never become a mining node

b) Can always become a mining node

c) Can become a mining node, depending if the implementation has the functionality implemented

12. On a consortium network

a) Everybody can become a miner, everybody can send transactions, and everything is public

b) Usually only a few selected nodes can be miners. Transactions can be further limited

13. The JSON-RPC Protocol

a) Is used to communicate between blockchain nodes

b) Is used to ensure safe communication between miners

c) Is a means of dumping the blockchain data in a so-called consensus export

d) Is used to communicate between the blockchain node and externally running applications

14. GETH

a) Is the reference implementation of the Ethereum protocol and every other node implementation internally uses the closed-source from Geth

b) Is the library that is used for the blockchain node Go-Ethereum. It is also used by Parity is parts, because it's closed source

c) Is one of the many blockchain nodes that implement the Ethereum Protocol. It's open source and everyone can contribute.

15. Consensus is reached

a) By the miner nodes which make sure that a transaction is valid

b) By every single node in the blockchain network executing the same transaction

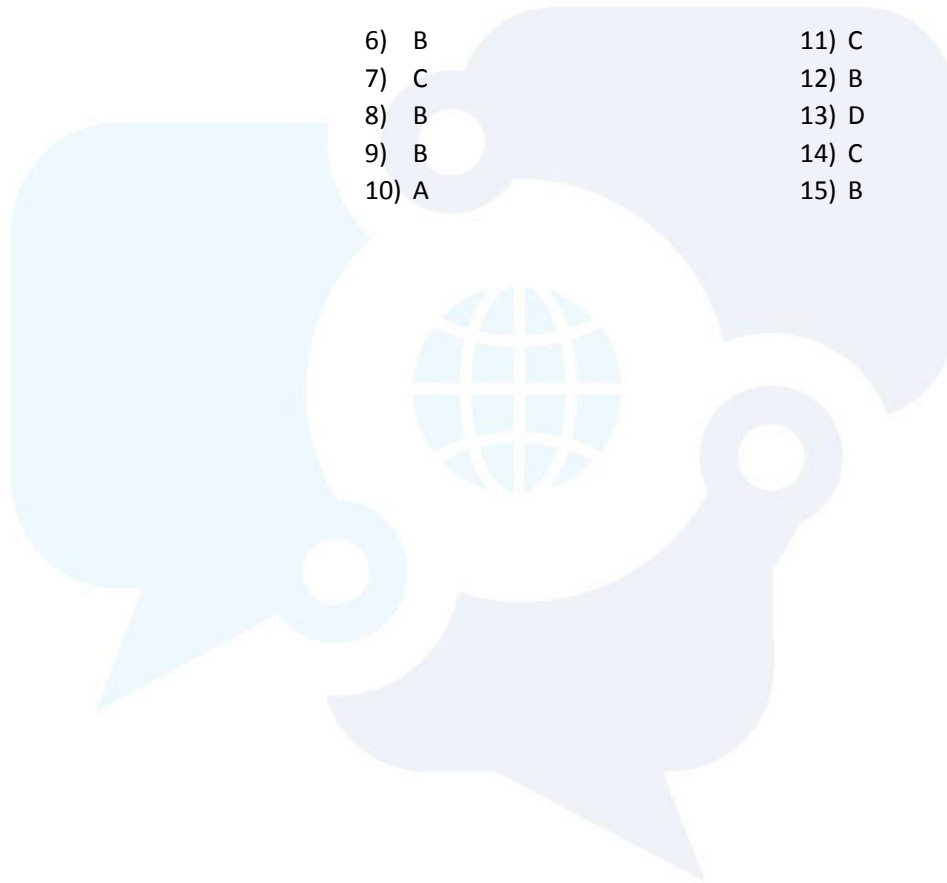c) By a cryptographic secure signature algorithm called ECDSA which makes sure that cheating is impossible

Chapter 2 Quiz Solution

1) A	6) B	11) C
2) A	7) C	12) B
3) B	8) B	13) D
4) C	9) B	14) C
5) B	10) A	15) B

Chapter 3: Ethereum Programming Basics

Different Programming Languages

Ethereum Smart Contracts run on compiled bytecode, which means that there can be several high-level languages which code can be written in.

In particular, Ethereum has a number of languages available:

1. Solidity – the most popular language now (2018).
2. Vyper – A Language by Vitalik Buterin with an emphasis on security
3. LLL – "Low Level Lisp-like Language"
4. Mutan – Golang-like, deprecated in 2015
5. Serpend, Python-like, but *seems* to be no longer maintained
6. Bamboo

What exactly is a "smart contract" technically?

A smart contract is a state-machine like execution of bytecode. It runs on the Ethereum blockchain and gets executed by all nodes in the network. It is *usually* written in high-level languages which compile the code down to assembly and bytecode.

EVM Assembly and Opcodes

The Ethereum virtual machine understands several opcodes in EVM assembly, which are represented by bytecodes. These are instructions for the EVM to read, write, jump, store etc. It can read and write from/to a stack or to storage.

A good list of available opcodes is listed in this stackexchange thread:
https://ethereum.stackexchange.com/questions/119/what-opcodes-are-available-for-the-ethereum-evm

And can also be found in the Ethereum Yellow paper https://ethereum.github.io/yellowpaper/paper.pdf

Gas and Gas Requirements

Each operation on the Ethereum blockchain is done by following the bytecode or opcodes. Each opcode costs a certain amount of gas.

The different gas requirements can be found on page 26 in the Ethereum Yellow paper (screenshots below): https://ethereum.github.io/yellowpaper/paper.pdf

In addition, the maximum amount of gas available has an upper ceiling of currently around 6 million. In other words, the execution is limited by the maximum amount of gas a user can provide and a miner would accept.

This leads to multiple problems "normal" software developers face.

1. During development of the smart contract you might have only a few users, but then in production, when things start to "scale" the contract suddenly runs into out-of-gas exceptions.
2. Loops are the biggest trap. **You should never use loops if you don't know how much gas it consumes at all times**.

Gas-Costs Economics on the Blockchain

Gas is calculated by how complex the transaction is to be executed. And the reason why you pay in "gas" rather than "Ether" is that in this way the actual $ costs can stay constant even though Ether might get much more valuable.

Let's run an example:

A transaction which interacts with a contract needs 4 million gas to be executed. You *offer* 2 GWei per gas, which is $2*10^{-9}$ Ether $* 4 * 10^6 => 2*4*10^{-3} = 0.008$ Ether, or at the time of writing, this would cost you around USD5.

During network congestion the gas price can rise. The miners decide which transactions to include in the next block and there is not unlimited amount in one block. So transactions compete by gas and gas price, basically the reward the miner gets.

If you decide, for some reason, to offer 50 Gwei for the same transaction to speed it up, you'd pay:

$50 * 10^{-9}$ Ether $* 4 * 10^6 = 4*50*10^{-3} = 0.2$ Ether, or USD140. A big difference, as you can see.

Data Storage in a Merkle Patricia Trie

Ethereum has its own, efficient, system to store information. This system has to be cryptographically secure. All key, value bindings in Ethereum are stored in a Merkle Patricia Trie, which is held in a State Trie, a Storage Trie, a Transaction Trie and a Receipts Trie.

Privacy on the Ethereum Blockchain

On of the biggest issues on the Ethereum blockchain is the privacy issue. There are zk-SNARKs now, but one of the biggest misconceptions is that private variables in Solidity are "private". All data, in general, is public. Defining a variable "private" just means that it can't be accessed from other smart contracts in a programmatic way. Any decent programmer can access all private variables information within 10 minutes, give or take.

To ensure privacy on the blockchain an additional layer has to be created that encrypts information. This is still a matter of active research.

Chapter 3 Quiz

1. Smart Contracts can be written in
    a) Java, C++, Solidity and JavaScript, because the Ethereum Blockchain is completely language agnostic and cross compilers for every major language exist.
    b) Solidity, Viper, LLL and Serpent, because those are high level languages that are compiled down to bytecode
    c) Solidity and JavaScript, because those are the official first implementations for Distributed applications and the Blockchain fully supports those languages

2. Solidity gets compiled to
a) Bytecode that can't be understood by humans
b) Bytecodes which are essentially opcodes running instruction by instruction

3. Gas is used
a) Depending on the instruction/opcode run by the Ethereum Blockchain
b) Is a fixed amount for the length of your smart contract

4. To store almost all data in the Ethereum Blockchain
a) A Linked List with pointers to previous blocks hashes is used
b) A Merkle Patricia Trie is used
c) A Radix Trie is used because the Merkle Patricia Trie is too inefficient

5. You interact with a smart contract and see a gas usage of 50,000 gas with a gas cost of 15Gwei, how much Ether would you have to pay to the miner?
a) 750,000,000,000,000 Wei
b) 750,000,000,000 Wei
c) 750,000,000 Wei
d) A flat fee of 1 Ether

6. Checking the balance of an address inside a loop of a smart contract constantly
a) Doesn't cost any gas
b) Cost gas every time we check the balance

7. Gas costs accrue on
a) Sending a transaction no matter the content
b) Sending a transaction only with a new smart contract deployment
c) Sending a transaction only interacting with an already deployed smart contract

8. EVM assembly
a) Is much better than Solidity and a viable alternative
b) Is what every high-level language gets compiled to
c) Is another language similar to LLL, more secure than Solidity.

9. Smart Contracts
a) Are always living on the same address, because the blockchain is deterministic. So, one account can always have one smart contract
b) Are having the same address as the EOA
c) Are sitting on their own address. The Address is created from the nonce and the EOA address and could be known in advance before deploying the smart contract.
d) The address of the smart contract is a random address which gets generated by the miner who mines the contract-creation transaction.

Chapter 3 Quiz Solution

1) B
2) B
3) A
4) B
5) A

6) B
7) A
8) B
9) C

Chapter 4: Solidity Basics

Foreword

While the questions are very comprehensive, this "cheat sheet" is no substitute for the official solidity documentation, but should rather highlight certain aspects.

Layout of a Solidity File

A solidity file, in general, ends with ".sol".

It can contain multiple contracts, which do not have to have the same name as the filename.

It starts with the pragma-line, which defines for which compiler version the contract(s) was/were written.

Example:

```
pragma solidity ^0.4.24;
contract MyContract {}
```

Comments in Solidity

Solidity support line and block comments, as well as natspec-style comments:

```
// line comment
/// line comment
/*
* Block comment
*/
/**
 * @author Thomas Wiesner
 * Natspec style
 */
```

Importing of other files

It is possible to import other contracts from other files. This can be done in various ways from different sources.

In Remix it is possible to import local files and files from GitHub.

In Truffle it is *not* possible to import files from GitHub.

If you import contracts from other files you can either import the whole file with all containing contracts or just some contracts from the file. Directly from the Solidity documentation:
http://solidity.readthedocs.io/en/latest/layout-of-source-files.html?#importing-other-source-files

The important parts:

```
import "filename"; //imports everything from "filename"


import * as symbolName from "filename"; // import * as symbolName from
"filename";


import {symbol1 as alias, symbol2} from "filename"; // creates new global
symbols  alias  and  symbol2  which  reference  symbol1  and  symbol2  from
"filename"


import "filename" as symbolName; // same as import * as symbolName from
"filename";
```

Important aspects of Value-Types

Most of the value types are not copy by reference. There are all the typical primitive value types, such as integers, unsigned integers, Booleans.

1. No floats
2. Integers are truncated, so they are *always rounded down*

In addition to that there are addresses, which have members:

1. They have a balance
2. They have several methods for interaction:
    a. Address.transfer(ether_amount) sends ether to the "Address". If an exception happens, it is cascaded. 2300 gas is capped.
    b. Address.send(ether_amount) sends ether to the "Address". If an exception happens, a boolean "false" is returned. 2300 gas is capped.
    c. Address.call.value()() let's you interact and send ether to the "Address". If an exception happens, a boolean (false) is returned. All gas is forwarded.
    d. Then there is also Address.delegatecall which uses the scope of the current calling contract and is mostly used for libraries.
    e. Member variable: Address.balance which gives the balance in wei of the "Address".

Addresses hold 20 bytes of values.

The following are value types:

Boolean, Integer, Enums, Fixed Point Numbers, Address, Fixed-size byte arrays, Dynamically-sized byte arrays, Address literals, Rational and Integer literals, String literals, Hexadecimal literals and function types.

Strings in Solidity

You cannot compare strings directly in Solidity. So,

```
string1 == string2
```

doesn't work. There are some workarounds for that. For example

```
keccak256(string1) == keccak256(string2)
```

Arrays, Structs and Mappings

Mappings are accessed in Solidity like arrays in other languages:

```
myMapping[keyVal] = value;
```

1. Unlike arrays, in a mapping, every possible key is pre-initialized
2. That means, a mapping doesn't have a "size" or "length". You can't iterate through a mapping without any additional counter variable.

Structs are a convenient way to introduce a new data-type. If you have object oriented programming background, you'd create a new class for every model, etc. In Solidity you better create a struct, as it takes less overhead.

In a struct also every value is pre-initialized with its default value.

Functions and Variable Visibility

If a variable is declared public then the solidity compiler automatically generates a getter-function in the background.

If a variable is declared private it means it cannot be accessed programmatically through another smart contract instantiating the declaring contract.

Internal Functions cannot be accessed from instantiating contracts.

External functions must be called from instantiating contracts *or via the keyword "this.internalFunction"*.

Private functions can only be called from the declaring contract.

If a visibility is omitted it will throw a warning and an error in the future. It will automatically assume (at the moment) that the function is public.

Sometimes "view" and "pure" functions are considered as function visibility, because they make a writing function to a read-only function.

Function Modifiers

Function modifiers can be inherited and are thus available throughout the inheritance tree.

View/Pure Functions

A View function can read from state but not write.

A pure function can neither read from nor write to the state.

Fallback Functions

A fallback function should be defined if the contact can be called (or receive ether) without a dedicated (named) function call. The fallback function is an anonymous function that gets called when a transaction to a smart contract contains no or an invalid function identifier and thus no other function matches.

If the contract should be able to receive ether via the fallback function it shouldn't consume more than 2300 gas. Basically, emitting an event.

If there is no fallback function and during a transaction no other function matches then an exception is triggered.

Global objects: msg. and tx.

Msg.sender and tx.origin both contain an address. If, let's say, a transaction is started toward contract A that contract is calling another contract B then in contract B:

1. Msg.sender will contain the address of contract A
2. Tx.origin will contain the address of the person who initiated the transaction

Furthermore msg.timestamp contains the block timestamp but is not meant to be used for random variables.

Loops

Solidity supports all the well-known common loops. But using loops in solidity can be dangerous if you don't know the upper bounds of the loop.

Smart Contract execution runs on gas and every operation consumes gas. During development you might have little to no actual data in a smart contract and the loops run just fine. But in production the loops might consume all the gas.

So, try to avoid loops. Only use loops if you know the upper bounds.

Events

Events are a great way for the off-chain part of DApps to react to certain situations. Events cannot be consumed by smart contracts and are running on some sort of "side chain".

According to the style guide Events should be capitalized. In earlier versions of Solidity, you should also prefix the event name to avoid confusion with function names. In newer versions of Solidity, you have to use the keyword "emit" to emit an event, which circumvents this confusion problem.

Official Style Guide

There is an official style guide which defines a range of things.

http://solidity.readthedocs.io/en/latest/style-guide.html

A few important things in particular:

Functions should be in this order:

- constructor
- fallback function (if exists)
- external
- public
- internal
- private

Strings should be quoted with double-quotes instead of single-quotes.

CapWord Style:

- Contract and Library Names
- Structs
- Events
- Enums

mixedCase Style:

- Functions
- Modifiers

Constants should be all capitalized with underscores (SOME_CONSTANT).

Keeping lines under the PEP 8 recommendation to a maximum of 79 (or 99) characters helps readers easily parse the code.

The Difference between address.transfer(), address.send(), address.call.value()(), address.delegatecall() and address.callcode().

There is a confusion to what these do, because they all look like they provide the same functionality to beginners. Let's quickly discuss them here.

Both, address.transfer() and address.send() can be used to send funds from ContractA to an "address". They are both considered safe against re-entrancy because they send only the gas-stipend of 2300 gas along. So, if there is a contract running on "address" it can't do much, because it doesn't have enough gas to operate any meaningful logic, except to emit an event. "address.transfer()" is a high-level function. It is

newer than "address.send()" and was only recently introduced. That means there is one major outstanding benefit in using "address.transfer()" – it will cascade Exceptions, while with "address.send()", if there is an exception happening during the transfer, it will only return false. In short: address.transfer() will delegate exceptions and with address.send() you have to check the return value if its false or true.

Sometimes it's not enough that a receiving contract has only 2300 gas. This is where address.call.value()() comes in. With address.call.value()() you can forward all gas to the receiving contract so that it can do a lot more than just with the gas-stipend. That means it's both dangerous and useful. Dangerous because it opens the door to re-entrancy attacks when not coded correctly, following the Checks-Effects-Interactions pattern.

Address.delegatecall() works almost the same as address.call, with the major difference that it will keep the scope in the calling contract. It's used for libraries. So in a library you could do "this.myVariable" and actually speak to the calling contract's variable.

Address.callcode() is deprecated and shouldn't be used anymore.

Address.call.value()(), address.delegatecall(), address.callcode() and address.send() are considered low-level functions return a Boolean (true/false) if an error happens during execution.

We will talk about this in more detail in the Chapter "Solidity Advanced".

Chapter 4 Quiz

1. Solidity code…
   a) Can't be split across multiple files, everything should be in one single file
   b) Can be split across multiple files, but every contract must be in a file with the same name as the contract itself
   c) Can be spread across multiple files. To import all contract from a file you can use "import 'myfile.sol'. To import Contract MyContract from myfile.sol you use "import {MyContract as SomeContract} from 'myfile.sol';".

2. Files can be…
   a) Imported using relative and absolute paths, where the "." And the ".." depict that it's a relative path.
   b) Imported only via GitHub using the Repository and Username.
   c) Imported using the special requirefile(…) statement, which looks in a specific library path to import files.

3. Importing from GitHub…
   a) Works across all compilers and platforms the same way
   b) Is generally possible, but currently works only in Remix, but doesn't work in Truffle

4. Single line comments in Solidity are working with
   a) Either // or ///
   b) With /* comment */ or /** @.. natspec style */
   c) Are not possible, all comments must be multi-line

5. Multi-Line Comments in Solidity work with
   a) Either // or ///
   b) With /* comment */ or /** @.. natspec style */
   a) Are not possible, all comments must be single-line

6. The following are value types in Solidity
   a) Integer, Boolean, Struct, Mapping and Enum
   b) Integer, Boolean, Enum and Addresses
   c) Integer, Boolean, Structs and Fixed Point Numbers

7. To compare a String in Solidity you use
   a) String1 == string2
   b) The internal function "str_compare(str1,str2)"
   c) You can't directly compare two strings, but one method would be to hash both strings and compare the hashes
   d) Bytes32(string1) == bytes32(string2)

8. If we divide two integers: 5/2, the result is
   a) 2, because the decimal is truncated
   b) 3, because it's always rounded
   c) 2.5, because it's automatically converted into a float.

9. A Struct is a great way
   a) To define a new datatype in Solidity, so you don't need to use objects of another contract
   b) To hold instances of other contracts
   c) To implement pointers to other contracts that can hold new datatypes

10. A Mapping consists of keys and value.
   a) The Keys can be anything, but the value can't be another mapping or struct
   b) The Value can be anything, but the key cannot be another mapping, struct, integer or Boolean
   c) The value can be anything, but the key cannot be another mapping, struct, enum or dynamically sized array

11. To Iterate through a mapping you
   a) Can use the length parameter of the mapping
   b) You need an external helper variable
   c) You cannot iterate any mapping to make the overall language design more safe

12. Function and Variable Visibility:
   a) A function marked as internal cannot be called by other contracts, unless the function is used by a derived contract. Private Functions cannot be called by any other outside contract and public variables are generating automatically a getter function.
   b) A function that is marked as external can never be called internally. Private functions can also be called by derived contracts using inheritance. Private variables are accessible also in derived contracts.

13. View and Pure Functions:

a) A function marked as pure can change the state, while a view function can only return static calls

b) A function marked as view can never access state variables, while pure functions are here to return only one value

c) A view function can access state variables, but not write to them. A Pure function cannot modify or read from state.

14. View and Pure Functions

a) Can only be accessed during calls

b) Can be accessed during transactions and calls

15. The Fallback function

a) Cannot receive Ether, not even by adding the payable modifier

b) Can contain as much logic as you want, but it's better to keep it short and not exceed the gas stipend of 2300 gas

c) Can be used to avoid receiving ether.

16. To get the address that initiated the transaction you need to use

a) tx.origin

b) msg.sender

17. If a User calls contract A and that calls Contract B, then msg.sender in Contract B will contain the address of

a) The User

b) Contract A

18. Loops in Solidity

a) Are a great way to circumvent gas requirements, because a loop will only consume gas once

b) Are preferable when used with datastructures that grow, such as arrays or mapping, because it's easy to estimate the gas requirements

c) Should be avoided where possible, because of unknown side-effects on the gas requirements

19. Events

a) Are stored on chain and are a great way to get a return value when a contract calls another contract

b) Are stored in something like a side-chain and cannot be accessed by contracts

c) Are used primarily for debugging exceptions in solidity

20. According to the official Style Guide

a) You should capitalize function names, events and contract names, to avoid confusion with JavaScript. You should use Tabs to indentation and a maximum of 80 characters per line.

b) Contract names should be capitalized, while functions should be mixedCase. You should use 4 spaces as indentation and a maximum of 79 (or 99) characters per line.

c) Contract should be mixedCase, as well as function names. Events should be capitalized. 2 spaces should be used as indentation and a maximum of 120 characters per line.

21. A version pragma is a great way

a) To make it clear for which compiler version a smart contract was developed. It helps to avoid breaking changes.

b) To make it clear for which blockchain a smart contract was developed. It helps to avoid confusion with beta-customers.

c) To make it clear for which blockchain node a smart contract was developed. It helps to avoid mixing up different versions of go-ethereum.

22. Variables of the type address store

a) A 20 bytes value

b) A 32 bytes value

c) A string

d) A 20 characters long hex number

23. Address.send()

a) Will cascade exceptions and address.transfer() will return a false on error

b) Will return false on error while address.transfer() will cascade transactions

24. Address.call.value()

a) Sends the gas stipend of 2300 gas and returns a false on error

b) Sends all the gas along and cascades exceptions

c) Sends all the gas along and returns a false on error

d) Sends the gas stipend of 2300 gas and cascades exceptions

25. Address.send() and address.transfer()

a) Are considered safe against re-entrancy because of the small gas stipend of 2300 gas

b) Are considered dangerous because they send all gas along, it's better to use address.call.value()()

26. When defining a new datatype

a) It's best to use a contract with public storage variables, so it can be used like a class

b) It's best to use a struct, which is cheaper than deploying a new contract

c) It's not possible to generate new datatypes in Solidity

Chapter 4 Quiz Solution

1) C	9) A	17) B
2) A	10) C	18) B
3) B	11) B	19) B
4) A	12) A	20) B
5) B	13) C	21) A
6) B	14) B	22) A
7) C	15) B	23) B
8) A	16) A	24) C
		25) A
		26) B

Chapter 5: ERC and EIP

What is an ERC?

ERC stands for Ethereum Request for Comments.

It is basically a GitHub issue tracker where developers can ask for comments on contract (and other) proposals. It started with #1 and increments every time a new issue is opened.

For example, the ERC20 token is issue #20.

What is an EIP?

EIP stands for Ethereum Improvement Proposal.

It has the same format as ERC but are used to propose changes in the Ethereum Protocol.

EIP also start with EIP #1 and increment every time a new issue is opened.

What exactly is the ERC20 Token Contract?

The ERC20 Token contract is a "standard" template to deploy fungible tokens on the Ethereum blockchain as a smart contract. It is basically a standard interface and a sample implementation of the functions necessary to create and operate an ERC20 Token.

The Interface is defined in the following way, taken from the Ethereum Wiki:

```
// ----------------------------------------------------------------------
// ERC Token Standard #20 Interface
// https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-
standard.md
// ----------------------------------------------------------------------
contract ERC20Interface {
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint
balance);
    function allowance(address tokenOwner, address spender) public constant
returns (uint remaining);
    function transfer(address to, uint tokens) public returns (bool
success);
    function approve(address spender, uint tokens) public returns (bool
success);
    function transferFrom(address from, address to, uint tokens) public
returns (bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender,
uint tokens);
}
```

These are the functions *a contract has to include to be ERC20 compatible.*

Wallets and other software use this interface to directly interact with any ERC20 compatible token contract so that the ABI array is not necessary anymore.

What exactly is the ERC721 Token Contract?

The ERC721 comes out of the Ethereum Request for Comments #721. It is a standard for non fungible tokens and is very similar to the ERC20 token contract. The main difference is that every single token has a non-interchangeable number.

It became famous through the crypto-kitties contract. It's very often used for collectibles where a collectible is of a specific type (the ERC721 token name for example), but each token has a different value.

The Standard Interface for the ERC721 Token Contract is the following, directly taken from the http://erc721.org/ website:

```
contract ERC721 {

        event Transfer(address indexed _from, address indexed _to,
uint256 _tokenId);


        event Approval(address indexed _owner, address indexed
_approved, uint256 _tokenId);


        event ApprovalForAll(address indexed _owner, address indexed
_operator, bool _approved);

        function balanceOf(address _owner) external view returns
(uint256);

        function ownerOf(uint256 _tokenId) external view returns
(address);


        function safeTransferFrom(address _from, address _to, uint256
_tokenId, bytes data) external payable;


        function safeTransferFrom(address _from, address _to, uint256
_tokenId) external payable;


        function transferFrom(address _from, address _to, uint256
_tokenId) external payable;


        function approve(address _approved, uint256 _tokenId) external
payable;

        function setApprovalForAll(address _operator, bool _approved)
external;
```

```
        function getApproved(uint256 _tokenId) external view returns
(address);

        function isApprovedForAll(address _owner, address _operator)
external view returns (bool);

        function supportsInterface(bytes4 interfaceID) external view
returns (bool);
}
```

Chapter 5 Quiz

1. What's the difference between ERC and EIP?
   a) Ethereum Request for Comments (ERC) are used to define standards for the usage of Ethereum. Ethereum Improvement Proposals (EIP) are used to improve the Ethereum Protocol itself.
   b) Ethereum Request for Comments (ERC) are used to propose new distributed applications on top of the Ethereum layer, while Ethereum Improvement Proposals (EIP) are used to improve existing mining software.
   c) Ethereum Request for Comments (ERC) are an open platform to discuss continuous forking of the Ethereum platform. Successful forks are then incorporated in the Ethereum Improvement Proposals (EIP) for further voting by the Ethereum Consortium.

2. What is the difference between ERC20 and ERC721 Tokens in simple terms?
   a) The tokens of a certain ERC20 symbol are all the same, the tokens of an ERC721 symbol are all different. So, ERC20 tokens are fungible, while ERC721 tokens are non-fungible.
   b) The tokens of a certain ERC20 symbol are all different, the tokens of an ERC721 symbol are all the same. So, ERC20 tokens are non-fungible while ERC721 tokens are fungible.
   c) The ERC20 token standard was the first standard token contract out there which was superseded by ERC721 tokens in order to support different token standards. One of the token standards that was necessary was non-fungible tokens. Since ERC721 tokens exist there is no need for ERC20 tokens anymore.

3. In order to implement an ERC20 token contract, you'd need at least to implement the following functions and events in order to fulfill the interface requirements:
   a) totalSupply(), balanceOf(address), allowance(address,address), transfer(address,uint256), approve(address,uint256), transferFrom(address,address,uint256). Events: Transfer(address,address,uint256), Approval(address,address,uint256)
   b) name(), symbol(), totalSupply(), balanceOf(address), ownerOf(uint26),approve(address,uint256), takeOwnership(uint256),transfer(address,uint256),Events: Transfer(address,address,uint256), Approval(address,address,uint256)

4. Why is it important to follow the same Interfaces?
   a) Because websites that try to interface with the Token would have to know the exact ABI. With the standard Interfaces it's clear how the interaction has to be defined.
   b) By following the standard interface the Ethereum Foundation can easily validate the Tokens and approve any audits
   c) The Interface is a mere suggestion for the developer. The actual Token contract can look totally different and would still be considered an ERC20/ERC721 token.

Chapter 5 Quiz Solution

1.) A
2.) A
3.) A
4.) A

Chapter 6: Solidity Advanced

Inheritance in Solidity

In Solidity it's possible to use inheritance, including abstract contracts as well as interfaces.

In addition to that, Solidity also supports inheritance from multiple contracts. The inheritance uses the keyword "is". So, "SomeContract is AnotherContract" where SomeContract extends AnotherContract. Let's see an example:

Simple example:

```solidity
contract MyContract {
    function one() {
        //the code
    }
}
contract MyOtherContract is MyContract {
    function two() {
        //code
    }
}
```

You could instantiate MyOtherContract and use function "one" on it:

```solidity
MyOtherContract myOtherContract = new MyOtherContract();
myOtherContract.one();
```

This applies to both, properties and functions.

The keyword "private" on functions, however, makes functions unavailable in the inheritance graph.

As stated above, multiple inheritance is also possible: "SomeContract is ContractA, ContractB, ContractC { }".

If you want to know more about inheritance, jump directly into the chapter in the solidity docs: https://solidity.readthedocs.io/en/v0.4.24/contracts.html?#inheritance

What exactly is Web3.js?

Web3.js is often confused with the term "Web 3.0". Web3.js is the JavaScript library to conveniently interact with Blockchain nodes.

If you want to interact with blockchain nodes, you need to talk to a JSON RPC Interface. It doesn't matter which language you are trying to use. Be it C, C++, C#, Java, JavaScript, PHP, you need to implement *something* that can talk to this JSON RPC Interface of the blockchain node.

In JavaScript you can go all-low level and encode the requests yourself. Do the proper padding of Input/Output Parameters, decode the output, handle the communication. *OR* you can use Web3.js which does all of that.

There are two major versions (at the moment, June 2018). Version 0.2 is the current "stable" version. It's highly in use by all kinds of libraries building on top of it. Version 1 is a major re-write, supporting Promises.

So, long story short: Web3.js is a JavaScript library to conveniently interact with JSON RPC Interfaces of Blockchain nodes. It does all the raw encoding, formatting, padding and decoding of the output.

What is the MetaData and the ABI Array?

When you write a contract in solidity and deploy it on the blockchain nobody really knows how to interact with that contract. From the outside, you just see bytecode and you can't guess how input-parameters have to be padded, output has to be decoded. What are the functions? And their names?

Let's have a quick look at a sample contract:

```solidity
pragma solidity ^0.4.23;

contract MyContract {
    uint myStorageVar;
    function myFunction(uint someArg) public {
        myStorageVar = someArg;
    }
}
```

The ABI, the Application Binary Interface, is a JSON-representation of the expected input/output and functions of the contract:

```
[
    {
        "constant": false,
        "inputs": [
            {
                "name": "someArg",
                "type": "uint256"
            }
        ],
        "name": "myFunction",
        "outputs": [],
        "payable": false,
```

```
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
```

This way, anyone who externally wants to interact with the smart contract knows how to "talk to the contract".

But what about the MetaData. The MetaData is an even bigger array containing all sorts of information. From the compiler version, to *eventually* the address of the contract when it's deployed.

```
{"compiler":{"version":"0.4.24+commit.e67f0147"},"language":"Solidity","
output":{"abi":[{"constant":false,"inputs":[{"name":"someArg","type":"ui
nt256"}],"name":"myFunction","outputs":[],"payable":false,"stateMutabili
ty":"nonpayable","type":"function"}],"devdoc":{"methods":{}},"userdoc":{
"methods":{}}},"settings":{"compilationTarget":{"browser/MyContract1.sol
":"MyContract"},"evmVersion":"byzantium","libraries":{},"optimizer":{"en
abled":false,"runs":200},"remappings":[]},"sources":{"browser/MyContract
1.sol":{"keccak256":"0x8a615f3d0dd31c268aaa12ef557972c4c0b51231bbc67d806
31994426ea6d6d5","urls":["bzzr://cb9968f84246d4ac7705652db44578dd1dc27c3
433c298d6dd7a8f528d8137dc"]}},"version":1}
```

Difference between address.send and address.transfer

One thing I want to highlight is the difference between "someAddress.send()" and "someAddress.transfer()" as it's not 100% clear to a lot of Ethereum beginners just starting now with Ethereum development.

Both functions can be used to transfer funds out of a smart contract to the address "someAddress". One is cascading exceptions, the other one is a low-level function returning a Boolean on error.

The problem with this Boolean is that it led to programming errors, where developers forgot to check the return value and locking up funds in a smart contract.

So, whenever possible, use "someAddress.transfer()", which is a high-level function cascading exceptions. "someAddress.send()" is a low-level function which returns "false" if the transfer failed, ran out of gas, the calling contract threw an exception, and so on.

Both functions send only the gas stipend of 2300 gas along.

Exceptions with Solidity

There are exceptions in Solidity, but you can't catch them nor can't they return an error message (yet).

Exceptions are thrown upon errors, like out of gas exceptions. But they can be manually triggered as well.

There are three keywords to trigger an exception "require", "assert" and "revert". There is another keyword "throw" which is deprecated.

I want to give you a quick overview of the usage. A thorough overview can be found in the Solidity documentation here: https://solidity.readthedocs.io/en/v0.4.24/control-structures.html?#error-handling-assert-require-revert-and-exceptions

Earlier you had to do this:

```
if(myVar <= 50) {

    throw;

}
```

As stated above, this is deprecated now and becomes:

```
require(myVar <= 50);
```

What is the difference between require and assert?

Require is used to check input parameters from users. It's returning all remaining gas.

Assert is used to check internal states that *should never happen*. It's used to check invariants. It's consuming all remaining gas.

Why avoid .call.value()()?

When interacting with other smart contracts and sending funds, it's often not enough to send only the 2300 gas along.

With address.call.value()() you can interact with other smart contracts and *all gas will be sent along*.

This is potentially dangerous, because the called contract can make calls back and, if the call doesn't follow the checks-effects-interactions pattern, it can lead to reentrancy attacks. http://solidity.readthedocs.io/en/v0.4.21/security-considerations.html#re-entrancy

Additionally, it's a low-level function and doesn't propagate errors. So, if the called contract throws an exceptions it will return a Boolean "false".

One way to send only a specific amount of gas along *and* to make it clear what is meant when calling an external contract, is to use named calls:

https://solidity.readthedocs.io/en/latest/control-structures.html?#external-function-calls

In this case you would do something like this:

MyContractInstance myInstance = MyContractInstance(addressOfMyContract);

myInstance.contractFunction.value(1000).gas(50000);

The problem with this is that it will also use the call.value()() low level function, but cascades exceptions.

Whatever you do, be especially careful with external contract calls. The warning from the solidity docs can be read here:

Warning

Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into

Low-Level Assembly and Solidity

Sometimes, in high level Solidity, it's not possible to write the code you want in the way you want it. Sometimes you need more fine-grained access.

In Solidity you can add in Assembly in-line and thus have more control over the program flow. It specifies an assembly language that is easier to use, especially with regards to special situations within the Ethereum Virtual Machine. One example would be finding the right stack slot, which is easier with the inline-assembly "overlay" in Solidity. Read more: https://solidity.readthedocs.io/en/latest/assembly.html

Chapter 6 Quiz

1. If contract MyContractA is derived from Contract MyContractB, then this would be the right syntax:
   a) contract MyContractA is MyContractB { … }
   b) contract MyContractA inherit (MyContractB) {…}
   c) contract MyContractA extends MyContractB {…}
   d) contract MyContractB derives MyContractA {…}

2. Inheritance is useful, because a contract that is derived from another contract can make use of:
   a) all public state variables and properties, public and internal functions and modifiers.
   b) all public and private state variables, public, internal and external functions, but not modifiers.
   c) all public state variables and properties, public functions and modifiers, but not internal, external or private ones.

3. Finish the sentence: The Library Web3.js is …
   a) useful when developing distributed applications with HTML and JavaScript, because it already implements the abstraction of the JSON-RPC interface of Ethereum Nodes.
   b) necessary when developing distributed applications with HTML and JavaScript, because the proprietary JSON-RPC interface of Ethereum Nodes is closed source.
   c) a great way to start with a boilerplate distributed application. Web3.js gives you a lot of options to start either with React or Vue.js apps.

4. When solidity is compiled then also Metadata is generated
   a) The Metadata contains the ABI Array, which defines the Interface to interact with the Smart Contract. Metadata can also contain the address of the smart contract when it gets deployed.
   b) Metadata contains the address, and the size of the smart contract. The ABI Array is generated externally upon deploying the smart contract.
   c) The ABI array and the Metadata are not generated when solidity is compiled to bytecode, they are generated by migration software which deploys the smart contract on the blockchain.

5. The difference between address.send() and address.transfer() is
   a) .send returns a Boolean and .transfer throws an exception on error. Both just forward the gas-stipend of 2300 gas and are considered safe against re-entrancy.
   b) .send throws an exception and .transfer returns a Boolean on error. Both just forward the gas-stipend of 2300 gas and considered safe against re-entrancy.
   c) .send returns a Boolean and .transfer throws an exception on error. .send is considered dangerous, because it sends all gas along, while .transfer only sends the gas stipend of 2300 gas along.
   d) .send and .transfer are both considered low-level functions which are dangerous, because they send all gas along. It's better to use address.call.value()() to control the gas-amount.

6. All low-level functions on the address, specifically address.send(), address.call.value()(), address.callcode() and address.delegatecall()
   a) Interrupt execution on error, because they throw an exception.
   b) Continue execution on error silently, which is the reason why they are so dangerous.
   c) Return Booleans to indicate an error during execution
   d) .send() throws an exception, while the other functions return Booleans during execution to indicate an error.

7. When using assert to check invariants and it evaluates to false
   a) All gas is consumed
   b) All remaining gas is returned

8. When using require to check input parameters and it evaluates to false
   a) All gas is consumed
   b) All remaining gas is returned

9. To send ether to a contract without a function call:
   a) A fallback function must be declared and it must be made payable. If there is no fallback function or the fallback function is not payable it will throw an exception.
   b) Either a fallback function which is payable exists, or no fallback function at all exists.
   c) You cannot send ether to a contract without explicitly calling a function. The fallback function can never receive ether.

10. If you need more fine-grained functionality than solidity offers out of the box
   a) You can incorporate inline-assembly to get better controls
   b) You must import pre-compiled assembly files which are then hard-copied into the bytecode of the compiled solidity file
   c) You can use Viper, the experimental assembly like language specifically to offer more flexibility

11. Address.Call() vs. Address.Delegatecall():
a) Address.call() is used for calling other contracts using the scope of the called contract in terms of storage variables. Address.delegatecall() is used for libraries, which uses the storage variables of the contract who called. Libraries are a great way to re-use already existing code and delegatecall() can make sure that no storage is used from the library, instead it looks like the code is directly copied into the calling contract.

b) Address.delegatecall() is used for calling other contracts using the scope of the called contract in terms of storage variables. Address. call() is used for libraries, which uses the storage variables of the contract who called. Libraries are a great way to re-use already existing code and call() can make sure that no storage is used from the library, instead it looks like the code is directly copied into the calling contract.

12. In Solidity it's not possible to use inheritance from multiple sources
a) True
b) False

13. Assert is used
a) To check internal states that should never happen
b) To check input arguments from users

14. Require is used
a) To check internal states that should never happen
b) To check input arguments from users

Chapter 6 Quiz Solution

1. A	6. C	11. A
2. A	7. A	12. B
3. A	8. B	13. A
4. A	9. A	14. B
5. A	10. A	

Chapter 7: Truffle

What is truffle?

Truffle is a framework for DApp development. It's not to be confused with truffle-contract, a library and abstraction for smart contract interaction, similar to Web3.js.

Truffle is a great, because it can be the foundation to the distributed app development with Unit Testing and continuous integration style workflows.

The Website: http://truffleframework.com/

It has three things in particular that are worth highlighting:

1. Built-In Smart Contract Compilation, Linking, Deployment and Binary Management.
   You will very quickly realize in Ethereum, that there are multiple compiler implementations for solidity. And if you are having multiple contracts which need to be linked together, a team that is working on them on a development chain, then a set path for deployment and compilation is what you are actually looking for.

2. Automated Contract Testing for Rapid Development
   With immutable smart contracts on the chain the one thing you need to do is test, test and test again. Truffle makes unit-testing with the built in testing framework including the development chain a breeze. **You can test in JavaScript and Solidity**.

3. Network Management for Deployment
   With multiple blockchains and multiple developers it's always good

How does truffle work?

Truffle is a javascript library that's based on NodeJs.

So, to install and run truffle you'd need to install Node and the Node Package Manager (NPM) first. You can find Node for all platforms on this website: https://nodejs.org/en/

After installing Node you can go to any command line interface (Terminal on Mac/Linux or PowerShell on Windows) and interact with npm.

Install truffle by typing in

```
npm install –g truffle
```

This should install truffle globally. Remember, truffle and npm don't have any automated update mechanism, so you might have to update truffle manually from time to time. This usually means uninstalling and re-installing truffle.

It should output something like this:

```
PS C:\Users\thoma> npm install -g truffle
C:\Users\thoma\AppData\Roaming\npm\truffle -> C:\Users\thoma\AppData\Roaming\npm\node_modules\truffle\build\cli.bundled.
js
+ truffle@4.1.11
removed 11 packages and updated 10 packages in 16.269s
PS C:\Users\thoma>
```

As you can see, the version that got installed is "4.1.11". Sometimes, when things are not working as expected, it's a great idea to roll-back to a different version. This can be done by specifying the version number during installation:

```
npm install –g truffle@4.0.0
```

would install truffle version 4.

Let's quickly discuss the truffle directories:

When you initialize a new project with

```
truffle init
```

in an empty folder, then truffle will download a scaffolding project from GitHub and unpack it.

The project will be initialized with a default structure, these are the important folders in *every* truffle project:

Name	Date modified	Type	Size
contracts	5/27/2018 9:05 AM	File folder	
migrations	5/27/2018 9:05 AM	File folder	
test	5/27/2018 9:05 AM	File folder	
truffle.js	5/27/2018 9:05 AM	JavaScript File	1 KB
truffle-config.js	5/27/2018 9:05 AM	JavaScript File	1 KB

In the "contracts" folder is where the smart contracts are located. These are the Solidity files.

In the "migrations" folder are the rules and scripts to deploy the contracts in the "contracts" folder. Contracts are not deployed automatically, you must write a rule for deployment of each and every solidity file.

In the "test" folder is both, the JavaScript and Solidity test files.

Then there are two js-files, the "truffle.js" and the "truffle-config.js" file. Both are identical, the "truffle-config.js" file is sometimes necessary on windows, because it has naming conflicts with the executable file "truffle".

Truffle Box

One project from the truffleframework worth mentioning is "Truffle Box". These are pre-configured "mini scaffolding projects" which make starting a new distributed application a breeze.

You can find a list of truffle boxes here:

http://truffleframework.com/boxes/

If you start with vanilla javascript or Vue, React, Angular, you can start off with a truffle box.

To get a truffle box, for example the webpack box, just type in

```
truffle unbox webpack
```

This will download, unpack and initialize the GitHub repository here: https://github.com/truffle-box/webpack-box

It's basically the same as cloning the repository and running "npm init".

How to write unit tests in truffle?

Writing tests in truffle is very easy and should be done for every function and every smart contract you develop.

Truffle supports both, JavaScript tests and Solidity tests.

The JavaScript tests can test the interaction with smart contracts "from the outside world", while the Solidity tests test the smart contracts as interaction from other smart contracts.

Let's assume for a moment you unboxed the webpack box from truffle as mentioned above. Let's have a look at the provided sample-test in JavaScript first. I'm going to take the tests 1:1 and go with you through the test line by line!

Start with the "tests/metacoin.js" file:

```javascript
var MetaCoin = artifacts.require("./MetaCoin.sol");

contract('MetaCoin', function(accounts) {
  it("should put 10000 MetaCoin in the first account", function() {
    return MetaCoin.deployed().then(function(instance) {
      return instance.getBalance.call(accounts[0]);
    }).then(function(balance) {
      assert.equal(balance.valueOf(), 10000, "10000 wasn't in the first
account");
    });
  });
  it("should call a function that depends on a linked library", function()
{
    var meta;
    var metaCoinBalance;
    var metaCoinEthBalance;

    return MetaCoin.deployed().then(function(instance) {
      meta = instance;
      return meta.getBalance.call(accounts[0]);
    }).then(function(outCoinBalance) {
      metaCoinBalance = outCoinBalance.toNumber();
      return meta.getBalanceInEth.call(accounts[0]);
    }).then(function(outCoinBalanceEth) {
      metaCoinEthBalance = outCoinBalanceEth.toNumber();
    }).then(function() {
      assert.equal(metaCoinEthBalance, 2 * metaCoinBalance, "Library
function returned unexpeced function, linkage may be broken");
    });
  });
```

```
  it("should send coin correctly", function() {
    var meta;

    //    Get initial balances of first and second account.
    var account_one = accounts[0];
    var account_two = accounts[1];

    var account_one_starting_balance;
    var account_two_starting_balance;
    var account_one_ending_balance;
    var account_two_ending_balance;

    var amount = 10;

    return MetaCoin.deployed().then(function(instance) {
      meta = instance;
      return meta.getBalance.call(account_one);
    }).then(function(balance) {
      account_one_starting_balance = balance.toNumber();
      return meta.getBalance.call(account_two);
    }).then(function(balance) {
      account_two_starting_balance = balance.toNumber();
      return meta.sendCoin(account_two, amount, {from: account_one});
    }).then(function() {
      return meta.getBalance.call(account_one);
    }).then(function(balance) {
      account_one_ending_balance = balance.toNumber();
      return meta.getBalance.call(account_two);
    }).then(function(balance) {
      account_two_ending_balance = balance.toNumber();

      assert.equal(account_one_ending_balance, account_one_starting_balance
- amount, "Amount wasn't correctly taken from the sender");
      assert.equal(account_two_ending_balance, account_two_starting_balance
+ amount, "Amount wasn't correctly sent to the receiver");
    });
  });
});
```

The test itself uses the Mocha framework and Chai assertions.

With the Node-typical imports you can import the artifacts from your smart contracts. Artifacts are basically the same as the ABI Array and contain more Meta-Data that let's you interact with your smart contract via truffle.

Then you describe the contract with the keyword "contract('ContractName, function…" which receives one function with the accounts of the blockchain node it connects to. This way you have access to the accounts and from there you can start your tests. The testing framework will automatically run through the tests one by one, but not necessarily keeping the order. That's quite important to remember so don't rely on the order of the functions.

Truffle always starts with a fresh deployment of your smart contracts to avoid having "unknown initial states". So, you start with a fresh state and can initialize everything from there!

The functions you call within the tests usually rely heavily on JavaScript Promises. Promises are, if you haven't heard of them yet, a way to deal with concurrency. Basically, you'd have to wait for a transaction to be mined and with promises you can wait and when it's resolved it will continue in the ".then(…)" block, one by one.

The assertions are rather simple. I've never needed more than "assert.equal", but there is a great list in case you find the "equal" assertion non-sufficient: http://www.chaijs.com/guide/styles/#assert

Let's dig into the Solidity Tests!

Here is the full file from "tests/TestMetacoin.sol":

```solidity
pragma solidity ^0.4.2;

import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/MetaCoin.sol";

contract TestMetacoin {

  function testInitialBalanceUsingDeployedContract() {
    MetaCoin meta = MetaCoin(DeployedAddresses.MetaCoin());

    uint expected = 10000;

    Assert.equal(meta.getBalance(tx.origin), expected, "Owner should have
10000 MetaCoin initially");
  }

  function testInitialBalanceWithNewMetaCoin() {
    MetaCoin meta = new MetaCoin();

    uint expected = 10000;

    Assert.equal(meta.getBalance(tx.origin), expected, "Owner should have
10000 MetaCoin initially");
  }

}
```

As you can see it's a normal Solidity file, starting with a pragma line and containing a smart contract. The difference is the import: It imports "Assert.sol" and "DeployedAddresss.sol" which is necessary to interact with the MetaCoin Smart Contract from your migrations.

All the tests are then wrapped in "testTestName(…)" functions. And they can also use the "Assert.equal(…)" functions to make assertions.

The significant difference is the point of view. While the JavaScript tests are testing the smart contract from "outside" the blockchain, the Solidity Tests are testing the smart contract from "within" the blockchain, testing the contract as "external source".

The tests itself can be initiated with

```
truffle test
```

Again, it is imperative to unit test your code as good as you can. Test everything, but don't purely rely on tests. Manage the money at risk and eventually have a "pause" option for your smart contracts to give you some time to figure out what happened and how you can resolve it!

Chapter 7 Quiz

1. Truffle:
a) Is a framework that helps developers with Testing, Deployment and Management of Smart Contracts and Distributed Applications.
b) Is a library that helps developers to connect to Ethereum nodes, because it abstracts the JSON-RPC interface.
c) Is a framework for Java, similar to Web3.js for JavaScript. It's a great way to develop distributed Java enterprise applications.

2. Unit-Testing on a local chain is important, because it helps you to
a) Run tests quickly and especially for free, compared to continuous deployment on the Main-Network. This way you save a lot of fees, time and costs.
b) Run tests in an environment where logging is activated. On the Main-Net you have no access to transaction logs and this is ultimately the information you need to debug your contracts.
c) Avoid regression bugs with contracts that are updated constantly on the main-net. Once you update a contract on the main-net, the address stays the same, but the code changes and this can have disastrous side-effects.

3. With truffle it's easy to write clean-room unit-tests
a) For Java, JavaScript, and C++
b) For JavaScript using Web3.js
c) For Solidity and JavaScript
d) For any language, as long as it adheres to the open Testing-Interface from Truffle.

4. With the truffle config file you can manage
a) The amount of gas your contract deployment and transactions, against your contract, will need. This way you can essentially lower the gas costs over traditional web3.js dApps.
b) Different Networks to deploy your contracts to. This way you can easily deploy to a local blockchain, the main-net or the Ropsten/Rinkeby Test-Net with only one parameter.
c) You can manage your secret API keys to the Ethereum Network. This way you can get access to several different Ethereum nodes at the same time without the need to switch your keyfiles.

5. Truffle boxes are a great way
a) To contribute to the box community which is the distributed file system for truffle
b) To start with a pre-configured environment for most web-development needs
c) To use tools that makes boxing of Dapps for different platforms very easy

6. Truffle has an integrated in-memory blockchain which makes unit-testing very easy
a) True, but it's still good to use Ganache, or even a real private network for testing.
b) False, it's necessary to use Ganache or even a real private network for testing.

7. Using truffle-contract over Web3.js

a) Is a must for every developer, because Web3.js changes so often

b) Is a convenient way because Web3.js is currently still in beta and truffle-contract can handle transactions with JavaScript-promises.

c) They are both completely different things. Truffle-Contract is a framework while Web3.js is a library.

Chapter 7 Quiz Solution

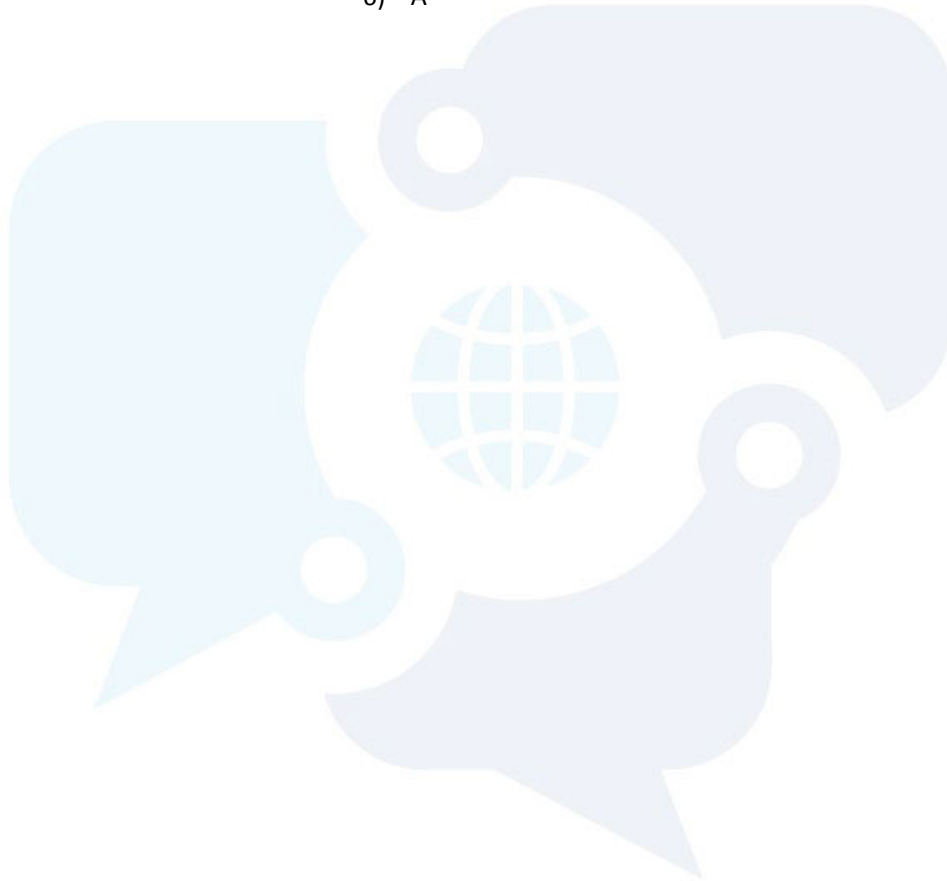|					
1)	A	4)	B	7)	B
2)	A	5)	B		
3)	C	6)	A		

Chapter 8: Contract Development Security

Security Best Practices

When it comes to development with Solidity and Blockchains, a lot of things change compared to traditional approaches. But not only the workflow changes, with solidity you have a brand-new language that isn't so mature yet. And tools that change on a regular basis.

First, always look for updates and new best practices. What's important right now may become old-news tomorrow. Things change rapidly and the best advice I can give here is stay up to date by reading news and being involved in the community.

Besides that, with Solidity, there are some things you want to follow.

1. Check-Effects-Interactions Pattern: If you access external smart contracts always make sure *not* to modify variables *after* the call, make it before. Check the return value and avoid Reentrancy situations where possible by avoiding low-level functions.
2. Privacy: "Private" Variables are not private on the blockchain
3. Loops & Gas Limits: Avoid Loops wherever possible. If you can't avoid loops, avoid loops with an "open end". Only use loops where you know the iterations at all times and don't underestimate how quickly you run out of gas.
4. Don't work only with the contract balance. It is possible to forcibly send Ether to a smart contract, so don't expect that your smart contract will have exactly the amount of Ether that you record through payable functions.
5. Use the "withdraw" pattern instead of "send": By withdrawing you have a dedicated function that let's a user "withdraw" money. The opposite would be a game for example, where the winner automatically gets the prize transferred once he won. Withdrawing would be a "pull" method, while sending money out is a "push" method.

There are more best practices and recommendations you might want to check out:

https://solidity.readthedocs.io/en/latest/security-considerations.html

https://consensys.github.io/smart-contract-best-practices/

Why Unit Testing?

Unit Testing can not only help you find regression bugs. Usually, when doing unit testing you get a better and deeper understanding of your code and all the different states it can take on. This, in turn, lets you find bugs early on and can also often reveal design flaws.

Unit testing doesn't have to be complicated. With frameworks like truffle it can be very easy and straight forward.

If you run smart contracts, then often money is involved. If you have bugs in your smart contracts then it is usually just a matter of time until hackers will steal your funds.

Losing money to hackers is not the only possibility where money can be lost. If your smart contract has bugs and, for example, costs too much gas or has functions that are not working properly, it might cost your users more money to run the smart contract than needed.

If you are working as a team or a single developer, it's important to test all aspects of your code. Best practices and workflows emerging, but the one thing that is similar to traditional software development is: testing, testing and more testing!

Contract Development Workflow

We usually do software development on traditional systems this way:

1. Prototyping phase
2. Alpha/Beta
3. Production
4. Then ship constantly updates until the product is satisfactory.

With development on the blockchain you can't really do step 4. So, things change a bit, but a real standard hasn't quite emerged yet.

What is currently state-of-the-art is something like this:

1) Prototyping/Architecture evaluation phase
2) Implementation in a Framework + Testing (in Memory Blockchain)
3) Testing on a Private Chain
4) Public Test-Net
5) Public Main-Net

With (1) it's usually just having a rough idea, but it's uncertain if the product can be programmed on the blockchain. Constraints like transaction speed and gas limits must be considered.

Also, it's important to note, to move only those parts to the blockchain that really benefit and need the blockchain technology. Many businesses try to map or migrate their entire systems to the blockchain and utterly fail.

At this stage it's already important to keep things simple. The simpler the smart contracts, the less prone to bugs they are.

After knowing that the products can be done on the blockchain, then it's usually re-integrated in (2) in a framework with a team. This can include test-driven development (TDD), but at least unit-testing all various aspects of the code. Unit testing can be done easily with in-memory blockchains like Ganache or the Truffle Developer Blockchain. Mining happens instantaneously, and you don't have to wait for any results. This also has side effects. On in-memory blockchains like Ganache you can set a timer for transactions, but it will never be the same situation as dealing with a real blockchain.

Therefore (3) is so important. Before deployment and inviting beta-testers, deploy your smart contracts to a private network. This can be done with Go-Ethereum for example by creating a new private chain and applying the same constraints as in the test-net or main-net environment.

If your program works, then it's time to invite real users. Before going into production there is usually a lengthy testing phase on one of the Ethereum Test-Networks. This way real users can interact with the smart contract and you can find scaling bugs for example. Or improve the smart contract. This way it's also possible to find upgrade mechanisms which suit your users. Upgrading can be informal, like telling users to use a new contract-address. Or it can be done via proxy-contracts or other mechanisms.

The very last step would be deployment on the Ethereum Main-Network. This is the place where transactions and interactions cost real money. So, users must be able to expect bug-free working smart contracts.

Upgrades and Bugs in Smart Contracts and the Ecosystem

The Ethereum Blockchain is in constant flux. Consensus just means that all nodes agree on *something*. But this "something" can be changed. For example, the gas limit. It can be increased or lowered if crypto-economics dictate. But this is not the only topic where updates are applied.

With the coming changes of Casper and the switch to Proof-of-Stake fundamental changes are applied to the Ethereum Blockchain. What was working before might not work after such a change. This means, that working smart contracts might suddenly have bugs because the underlaying system changed.

One option that can be considered early on is a "pause" function in case of unexpected upgrades of the platform. This way some time can be bought until a final solution is available.

This leads to an effective upgrade plan. Already during the prototyping/architectural planning phase an upgrade plan can be considered. While there is no one-fits-all upgrade mechanism, it's possible to upgrade smart contracts in a "non-traditional" way. This can be either by asking users to move up to proxy contracts using a set of contracts at different addresses. It's a balancing act between keeping immutability and trust and being able to react to bugs.

Interaction with Unknown Sources

At some point you are probably going to interact with unknown sources. This doesn't mean to use a library which isn't open source. Consider a simple withdraw method, where one withdraws funds to a specific address.

If you implement sending funds to an address with the .call.value()() method then you are inherently at risk because the called address receives all the gas. If the address is just an EOA then there is no problem. But if on the address is a smart contract that tries to re-call your smart contract and re-withdraw funds and you haven't taken care of this situation then you probably just lost all your funds.

So, if possible, you should avoid making any more changes to the contract state after doing the external call.

Also, mark untrusted sources in your contract code so they stand out.

And be aware of the difference between address.transfer(), address.send() and address.call.value()().

Favor pull over push methods for withdrawals, as external calls can fail accidentally or also deliberately leaving your smart contract in an unknown state.

Read more here: https://consensys.github.io/smart-contract-best-practices/recommendations/

Forcibly send money to smart contracts

One situation you might find yourself in is asserting the smart contracts balance against a counter-variable where deposited and withdrew funds are stored.

Let's assume you have a smart contract and it starts with 0 wei. Someone deposits 10 ether, so you increment a variable with the balance for the user and assert that all deposited ether equals the contract's balance. You don't have a fallback function in that smart contract, so you assume that nobody can deposit ether outside of the actual "deposit" function.

You couldn't be more wrong.

On the Ethereum blockchain it's possible to deposit Ether in two ways that jeopardizes these assertions.

Firstly, it's possible to send funds to a contract address before the contract is actually deployed. The contract address can be pre-calculated. It's a hashed value of the address that creates the smart contract plus the nonce. If it's known which address will be used to deploy the smart contract, then one can send a small amount of wei to the future contract address prior the contract being deployed. This way the contract starts already with a non-zero balance.

Read more here: https://github.com/ConsenSys/smart-contract-best-practices/issues/61

The other option is to send funds to a smart contract even though it doesn't have a fallback function or automatically throws an exception. If funds are stored in a smart contract, called ContractA and one wants to send those funds forcibly to ContractB then it's possible to use the "selfdestruct" function. With "selfdestruct(AddressOfContractB)" in ContratA, ContractB will automatically receive all stored funds in ContractA. This cannot be circumvented.

So, whatever you do, never code an invariant that only checks the balance of the contract. Don't assume that your "assert" statement which checks the balance of the contract is the same as all recorded wei-deposits at all times.

Randomness on the blockchain

There are two global variables that are mistakenly used for randomness on the blockchain.

One is the timestamp, stored in "block.timestamp" or "now". This is the most common mistake. This timestamp, while random to a certain degree, can be influenced by a miner. The timestamp ultimately gets set by a miner and is not set independently. So, if you are running a very successful smart contract then the chances are high that there is an incentive for miners to influence that timestamp.

The other variable is the block.number, which is not a good idea to use either. This variable is not so much used for randomness per se, it can be used to determine the end of an auction period for example. Let's consider the following code from the ConsenSys Website:

https://consensys.github.io/smart-contract-best-practices/recommendations/#caution-using-blocknumber-as-a-timestamp

```
modifier auction_complete {
    require(auctionEndBlock <= block.number      ||
        currentAuctionState == AuctionState.success ||
        currentAuctionState == AuctionState.cancel)
      _;}
```

It's not a good idea to use this either, as it cannot be guaranteed that the block mining time doesn't change. Especially with regards to platform upgrades.

Randomness is still a matter of active research on the blockchain, but two methods slowly emerge.

Method one is a smart contract that utilizes all participants as randomization. It's called a RANDAO smart contract that can be found here: https://github.com/randao/randao

The other method is trusting an external oracle or bind randomness to the bitcoin blockchain. You can read more about this here: https://consensys.github.io/smart-contract-best-practices/recommendations/#remember-that-on-chain-data-is-public

OpenZeppelin and Libraries

In contrast to other "seasoned" languages, such as Java or C++, there is not a plethora of frameworks and libraries available for Ethereum.

Especially here it's important to have reviewed and audited smart contracts when used as libraries in potentially thousands of projects.

OpenZeppelin set out to change this. They provide templates for Tokens, Crowdsales and many more applications. All their smart contracts are audited and peer reviewed and can be adopted in an uncomplicated way.

One thing to mention though, is that as soon as someone adapts those contracts they become un-audited. Nevertheless, it's often an excellent idea to start with the templates than from scratch.

Bug-Bounty Programs

So called "bug-bounty programs" are a great invention. They are not particularly new to the Blockchain world. In a bug-bounty program someone is rewarded to report a bug directly to a company instead of using (and probably abusing) it.

All major companies, such as Google, Facebook, Microsoft, etc. run bug-bounty programs. They categorize bugs in different severance categories and pay a remarkable amount of money, often in the tens of thousands of dollars, if someone finds and reports a bug.

For blockchain and smart contracts it can be used to incentivize the community to contribute to a smart contract security by using and literally brute forcing the smart contract during a testing phase.

Chapter 8 Quiz

1. Why is Unit-Testing so important?
a) It helps you find bugs, regression bugs and sometimes also helps you to understand your code from different angles.
b) It is a great way to spend time on something that you get paid for. But ultimately it will just slow down the development process.

2. If you are starting a new ERC20 token
a) It would be best to start from scratch, just looking at the required interface.
b) It is beneficial to copy and paste the already existing code from the Ethereum wiki and modify this until you like it.
c) Best is to start with an audited implementation, for example from OpenZeppelin, in order to re-use already existing code.

3. To generate a random number
a) It's good to use the block timestamp, as this is always different
b) It's good to use the block hash as this is clearly always very different
c) It's good to use the RANDAO smart contract
d) It's not possible to have a random number in a deterministic environment such as the Ethereum blockchain.

4. When you do external calls to other smart contracts
a) You should follow the checks-effects-interactions pattern and avoid state changes after the call.
b) You should follow the effects-checks-interactions pattern and avoid state changes before the call.
c) You should follow the checks-effects-interactions pattern, which is only necessary when you do calls to contracts where a direct contract call is not possible.

5. When you are programming a game like poker or battleships where you need to hide opponents' values is
a) with private state variables. This way nobody else other than the smart contract itself can see the information.
b) with external contracts holding those values. This way we can make sure that the information flow is following a clear logic and nobody else can access this information.
c) You can't hide anything on the blockchain, because the information is public, just the call is private which means only other smart contracts would be limited in accessing that information.

6. When considering smart contracts and the blockchain it's good
a) To move all existing logic to the blockchain, so everything runs on the same system. This way it might be more complex, but easier to maintain.
b) To move only those parts to the blockchain that really need the blockchain. This way smart contracts can be easier to read, easier to test and are not so complex.
c) To move those parts to the blockchain that deal with Ether transfers. All other parts can remain in traditional database systems. This way only the value-transfer is on the blockchain.

7. When a smart contract pays out money
a) It's good to use a push over a pull method.
b) It's good to use a push and a pull method to ensure that participants can get their money no matter the contract state. In addition to pushing it should contain a withdraw method.
c) It's good to use only pull and no push method.

8. To develop smart contracts:
a) It's good to start with a local in-memory blockchain with unit tests but then deploy to the main-net as rapidly as possible.
b) It's good to start with a local in-memory blockchain with unit-tests. Then, in the next step, debug and test the smart contract on a test-net like Ropsten or Rinkeby with beta customers to iron out last issues before deploying it to the main-net.
c) It's good to start with a test-net with beta-customers like on the Rinkeby or Ropsten testnet, before testing it locally on an in-memory blockchain simulation such as Ganache. Then deploy it to the main-net.

9. To avoid issues during Ethereum platform upgrades
a) It's good to inform users about the updates via a newsletter.
b) It's good to have the ability to pause a contract in order to manage the money at risk.
c) Ethereum doesn't upgrade the platform. It's fixed and final.

10. Integrating the community into your testing
a) Is great, because they often find bugs which weren't considered before.
b) Is not good, because you might give out secrets.

11. Having a bug-bounty program early on
a) Can help to engage the community in testing your smart contracts and therefore help find bugs early.
b) Might be a burden as it's mainly administrative overhead.
c) Is completely useless. Who wants to test beta-ware software. Better start with the bug-bounty program after the contract is released on the main-net.

12. Using selfdestruct(beneficiary) with the beneficiary being a contract without a payable fallback function:
a) Will throw an exception, because the fallback function is non-payable and thus cannot receive ether

b) It's impossible to secure a contract against receiving ether, because selfdestruct will always send ether to the address in the argument. This is a design decision of the Ethereum platform.

c) Selfdestruct doesn't send anything to a contract, it just re-assigns the owner of the contract to a new person. Sending ether must be done outside of selfdestruct.

Chapter 8 Quiz Solution

1) A	5) C	9) B
2) C	6) B	10) A
3) C	7) B	11) A
4) A	8) B	12) B