


TAL TECH

 blockchain.taltech.ee

PROGRAMMING SMART CONTRACTS

INTRODUCTION INTO PROGRAMMING IN SOLIDITY

27.03.2019

AGENDA

- Introduction
- Bitcoin Script
- Basic of Ethereum and Solidity
- Smart contracts
- Programming languages
- Remix

SMART CONTRACT HISTORY

- Definition of Smart contracts (**According to Nick Szabo**):

“A canonical real-life example, which we might consider to be the primitive ancestor of smart contracts, is the humble vending machine. Within a limited amount of potential loss (the amount in the till should be less than the cost of breaching the mechanism), the machine takes in coins, and via a simple mechanism, which makes a freshman computer science problem in design with finite automata, dispense change and product according to the displayed price.

[...]

Smart contracts go beyond the vending machine in proposing to embed contracts in all sorts of property that is valuable and controlled by digital means.”

SMART CONTRACT HISTORY

- Definition of Smart contracts (**According to Nick Szabo**):
“Many kinds of contractual clauses (such as collateral, bonding, delineation of property rights, etc.) can be embedded in the hardware and software [...]”
- The inherent features of blockchains that have enabled its application in so many domains, has also made it possible to achieve Nick Szabo’s vision of smart contracts that he defines as “a set of promises, specified in digital form, including protocols within which the parties perform on these promises”

SMART CONTRACT INTERPRETATION

- **Legal contract:** "I promise to send you \$100 if my lecture is rated 1*"
- **Smart contract:** "I send \$100 into a computer program executed in a secure environment which sends \$100 to you if the rating of my lecture is 1*, otherwise it eventually sends \$100 back to me"
- **Example:**

if HAS_EVENT_X_HAPPENED() is true:

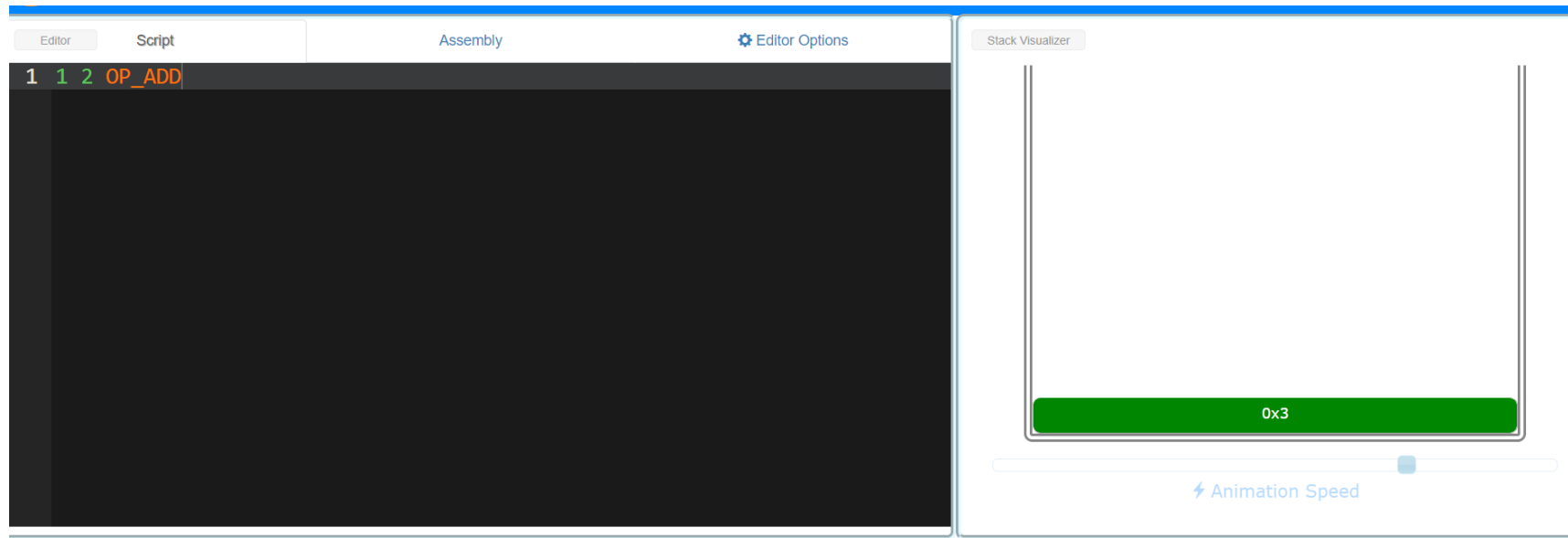
send(party_A, 1000)

else:

send(party_B, 1000)

INTRODUCTION INTO BITCOIN SCRIPT

- Transactions on Bitcoin are scripted. Script is forth-like, stack-based, and processed left to right.
- It is not Turing-complete and has no loops.



Source: <http://siminchen.github.io/bitcoinIDE/build/editor.html>

BITCOIN SCRIPT OPCODES

- This is a list of all Script words, also known as opcodes, commands, or functions.

Splice

If any opcode marked as disabled is present in a script, it must abort and fail.

Word	Opcodes	Hex	Input	Output	Description
OP_CAT	126	0x7e	x1 x2	out	Concatenates two strings. <i>disabled</i> .
OP_SUBSTR	127	0x7f	in begin size	out	Returns a section of a string. <i>disabled</i> .
OP_LEFT	128	0x80	in size	out	Keeps only characters left of the specified point in a string. <i>disabled</i> .
OP_RIGHT	129	0x81	in size	out	Keeps only characters right of the specified point in a string. <i>disabled</i> .
OP_SIZE	130	0x82	in	in size	Pushes the string length of the top element of the stack (without popping it).

Bitwise logic

If any opcode marked as disabled is present in a script, it must abort and fail.

Word	Opcodes	Hex	Input	Output	Description
OP_INVERT	131	0x83	in	out	Flips all of the bits in the input. <i>disabled</i> .
OP_AND	132	0x84	x1 x2	out	Boolean <i>and</i> between each bit in the inputs. <i>disabled</i> .
OP_OR	133	0x85	x1 x2	out	Boolean <i>or</i> between each bit in the inputs. <i>disabled</i> .
OP_XOR	134	0x86	x1 x2	out	Boolean <i>exclusive or</i> between each bit in the inputs. <i>disabled</i> .
OP_EQUAL	135	0x87	x1 x2	True / false	Returns 1 if the inputs are exactly equal, 0 otherwise.
OP_EQUALVERIFY	136	0x88	x1 x2	Nothing / <i>fail</i>	Same as OP_EQUAL, but runs OP_VERIFY afterward.

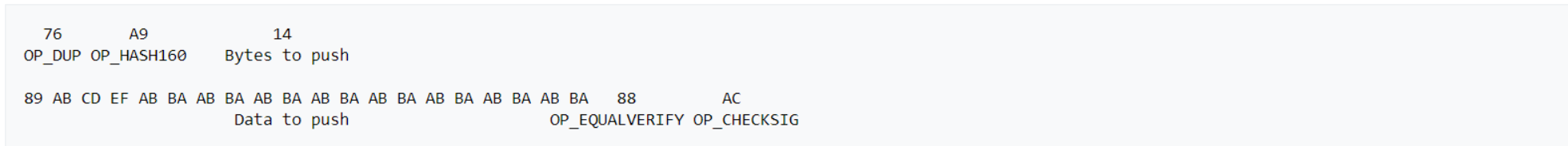
Source: <https://en.bitcoin.it/wiki/Script>

STANDARDIZED BITCOIN TRANSACTIONS AND SCRIPTS

- **Standard Transaction to Bitcoin address (pay-to-pubkey-hash):**

scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY
OP_CHECKSIG

scriptSig: <sig> <pubKey>



Note: scriptSig is in the input of the spending transaction and scriptPubKey is in the output of the previously unspent i.e. "available" transaction.

Here is how each word is processed:

Stack	Script	Description
Empty.	<sig> <pubKey> OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Constants are added to the stack.
<sig> <pubKey> <pubKey>	OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is duplicated.
<sig> <pubKey> <pubHashA>	<pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG	Top stack item is hashed.
<sig> <pubKey> <pubHashA> <pubKeyHash>	OP_EQUALVERIFY OP_CHECKSIG	Constant added.
<sig> <pubKey>	OP_CHECKSIG	Equality is checked between the top two stack items.
true	Empty.	Signature is checked for top two stack items.

STANDARDIZED BITCOIN TRANSACTIONS AND SCRIPTS

- **P2PK: "Pay To Public Key":**

OP_CHECKSIG is used directly without first hashing the public key. This was used by early versions of Bitcoin where people paid directly to IP addresses, before Bitcoin addresses were introduced. scriptPubKeys of this transaction form are still recognized as payments to user by Bitcoin Core.

```
scriptPubKey: <pubKey> OP_CHECKSIG  
scriptSig: <sig>
```

Checking process:

Stack	Script	Description
Empty.	<sig> <pubKey> OP_CHECKSIG	scriptSig and scriptPubKey are combined.
<sig> <pubKey>	OP_CHECKSIG	Constants are added to the stack.
true	Empty.	Signature is checked for top two stack items.

STANDARDIZED BITCOIN TRANSACTIONS AND SCRIPTS

- **P2SH: "Pay To Script Hash"**

A transaction's output is a boolean script that returns true or false. In this case, a miner will send money to your output if the script is run with the specified parameters. With P2SH, you can create multi-signature wallets that check for multiple signatures.

```
<recipient signature> [further signatures] <Redeem Script> OP_HASH160  
<Hash160(Redeem Script)>
```

```
OP_EQUAL
```

```
<OP2>
```

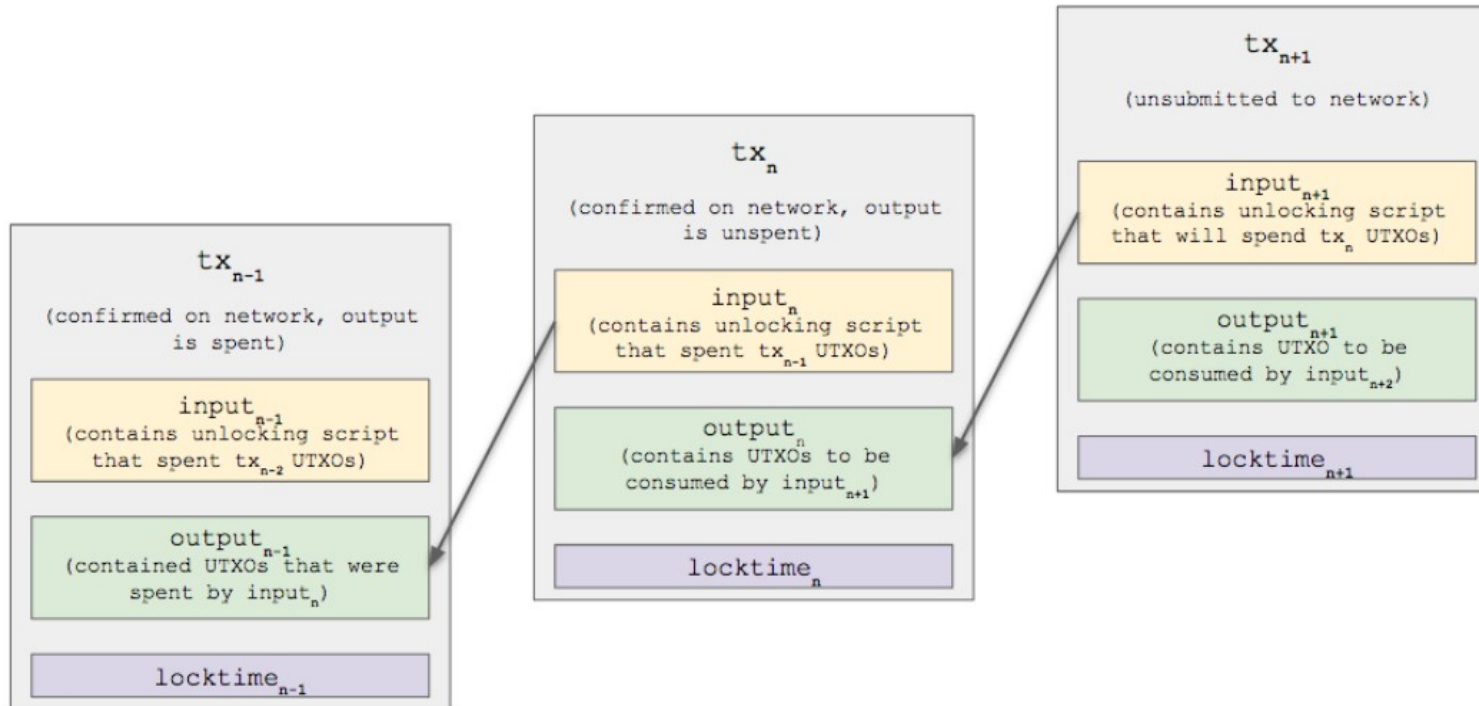
```
<pub key recipient A> <pub key recipient B> <pub key recipient C> <OP3>  
OP_CHECKMULTISIG
```

- **P2WPKH: "Pay To Witness Public Key Hash":**

This was a segwit feature (Segregated Witness). Instead of using scriptSig parameters to validate transactions, a new part of the transaction called witness is used.

BITCOIN TRANSACTION BLOCK-SCHEMA

- A relationship between confirmed transactions with unspent outputs and new unsubmitted transactions is shown in the figure below.



Transaction Relationship

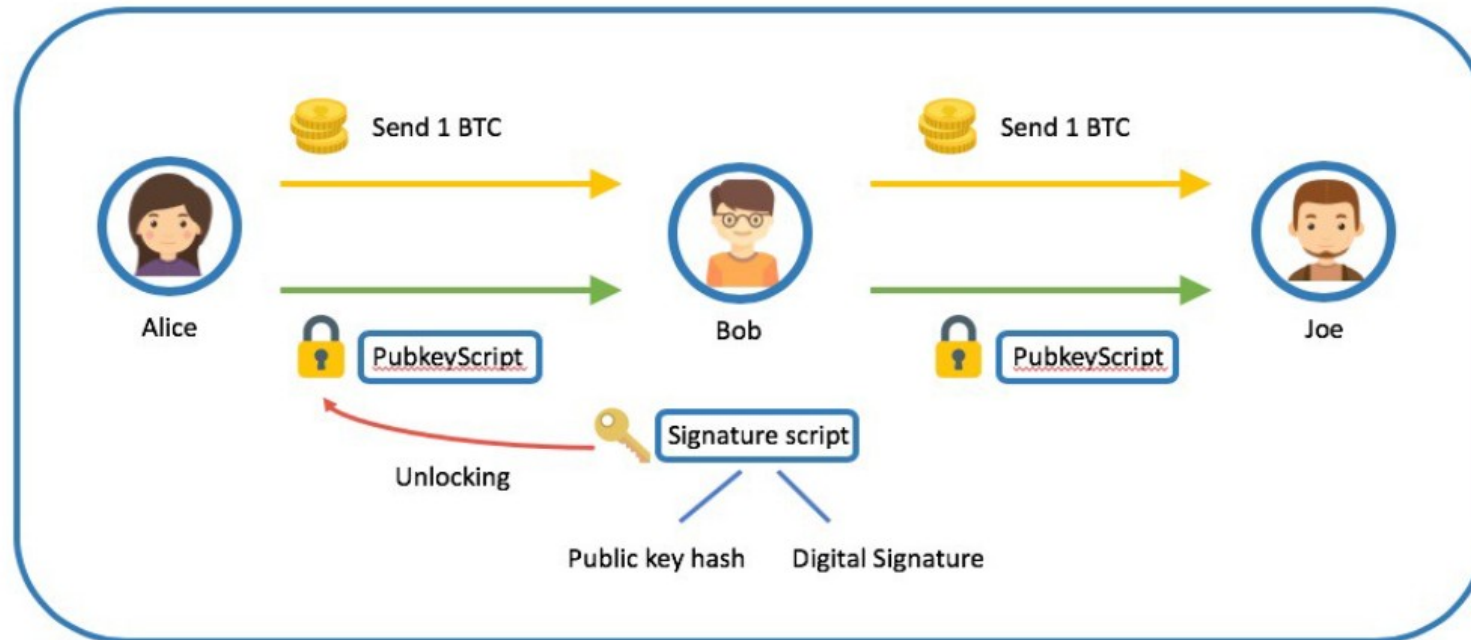
```
01000000019c2e0f24a03e72002a9
6acedb12a632e72b6b74c05dc3ce
ab1fe78237f886c48010000006a47
304402203da9d487be5302a6d69e
02a861acff1da472885e43d7528ed
9b1b537a8e2cac9022002d1bca03
a1e9715a99971bafef3b1852b7a4f0
168281cbd27a220380a01b330701
2102c9950c622494c2e9ff5a003e3
3b690fe4832477d32c2d256c67eab
8bf613b34effffffff02b6f505000000
0001976a914bdf63990d6dc33d705
b756e13dd135466c06b3b588ac84
5e02010000000001976a9145fb0e9
755a3424efd2ba0587d20b1e98ee
29814a88ac00000000
```

STEPS OF BUILDING A TRANSACTION

- **Step 1-** Identify the previous transaction that contains your UTXOs (Bitcoin).
- **Step 2-** Create input outputpoints for the new transaction to find previously spent UTXOs.
- **Step 3-** To spend/unlock the new UTXOs for the next transaction, you must build the outputs for the new transaction so the locking script contains the conditions.
- **Step 4-** Finally, write the unlocking script to meet the same criteria as the transaction locking script. The final step is the recipient's signature, which is also in the unlocking script, but is added in the middle of the transaction.

STEPS OF BUILDING A TRANSACTION

- A PubkeyScript is a set of recorded instructions that governs how the next person can receive and spend Bitcoin.
- Public key hash and private key signature are required to unlock the PubKey Script and spend received funds.



Source: <https://genesisblockhk.com/what-is-p2pkh/>

STEPS OF BUILDING A TRANSACTION

- To spend Bob's new bitcoin, he must prove he owns the Bitcoin address Alice sent it to.
- In P2PKH, only the owner of the bitcoin address provided to Alice can pass it on.
- Bob, generates the scriptSig to satisfy the scriptPubKey (generated by Alice) to send his Bitcoin to another person

REAL EXAMPLES OF BITCOIN TRANSACTION

- **Input**

Hex:

76a9144fd31c644c4b46c153601d
0e194ab689570f4ce488ac

ASM

OP_DUP

OP_HASH160

fd31c644c4b46c153601d0e194ab6895
70f4ce4

OP_EQUALVERIFY

OP_CHECKSIG

- Pksript is encoded in hexadecimal (Hex), and Bitcoin's scripting language op-codes are encoded in ASM, assembly.

- **Sigscript:**



483045022100c71b09c8161ca14b6ef96f155173bda080d72bb
77953122f268b9527ff806f9302200fac42255dbe0317bbbea75
837fe2e9e9bfdb3fc1161d5c45353dfdf4ecba074012102f4c4c2b
0b7c23b472cdc8c27d22a41677d7e69e220675647ccf231831e
5113cf

ASM:

- 3045022100c71b09c8161ca14b6ef96f155173bda080d72bb77
953122f268b9527ff806f9302200fac42255dbe0317bbbea7583
7fe2e9e9bfdb3fc1161d5c45353dfdf4ecba0740102f4c4c2b0b7c
23b472cdc8c27d22a41677d7e69e220675647ccf231831e5113
cf

REAL EXAMPLES OF BITCOIN TRANSACTION

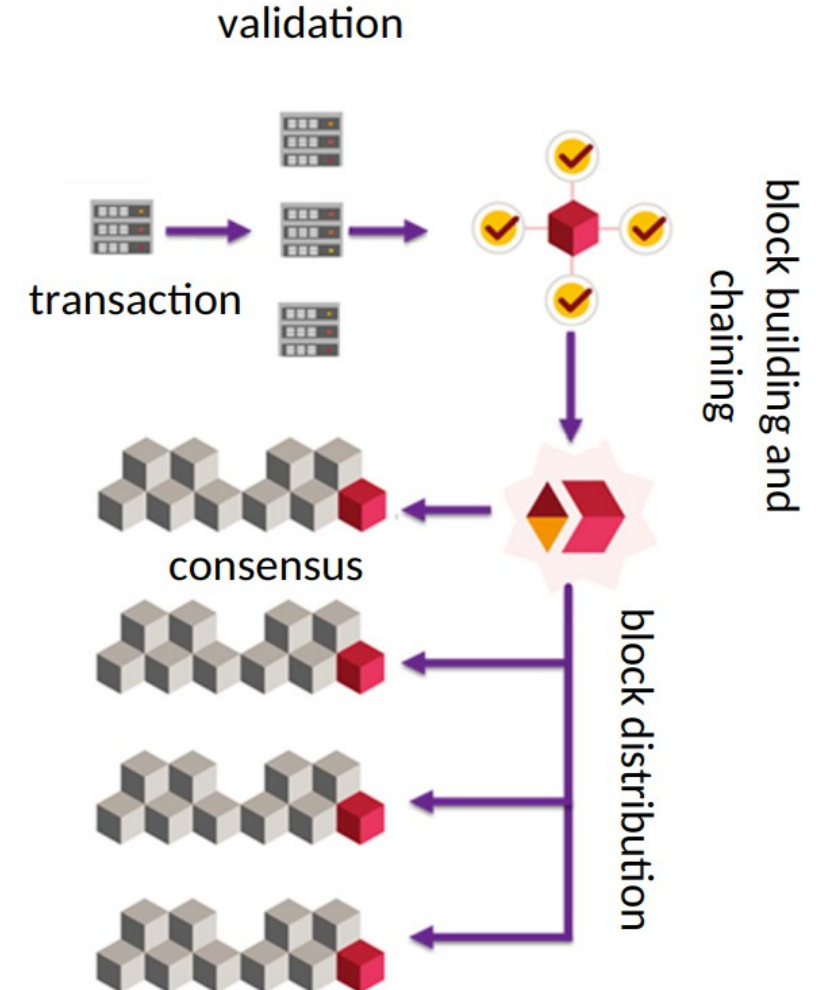
Outputs ⓘ

Index	0	Details	Spent
Address	18eszjuaEYUBgCnX4eVgErybziHYatjN4W 	Value	0.03847842 BTC
Pkscript	OP_DUP OP_HASH160 53f2d709258af543bbcfb3a271e3c26405c3935c OP_EQUALVERIFY OP_CHECKSIG		
<hr/>			
Index	1	Details	Spent
Address	1GnAy54jh5UC9ynAt1o4H7FNGfPzPgjSbj 	Value	1.04498774 BTC
Pkscript	OP_DUP OP_HASH160 ad14ecc25a9824657c0590e28f58c7c390a2fcf3 OP_EQUALVERIFY OP_CHECKSIG		

ETHEREUM AND ITS DATA PROCESSING

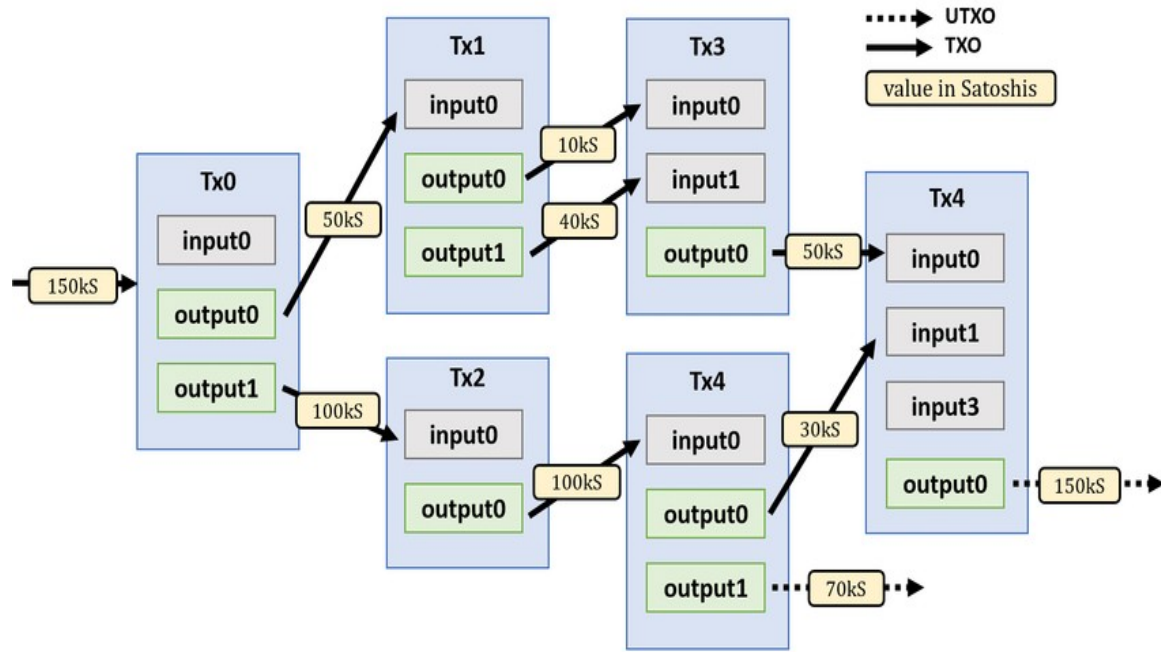
HOW BLOCKCHAIN ACCUMULATES BLOCKS

- **Transaction:** A node starts a transaction by first creating and then digitally signing it with its private key.
- **Validation:** A transaction is propagated to peers that validate the transaction based on preset criteria. Usually, more than one node are required to verify the transaction.
- **Block building and chaining:** Once the transaction is validated, it is included in a block, which is then propagated onto the network.
- **Block distribution:**
- **Indirect, algorithmic consensus**

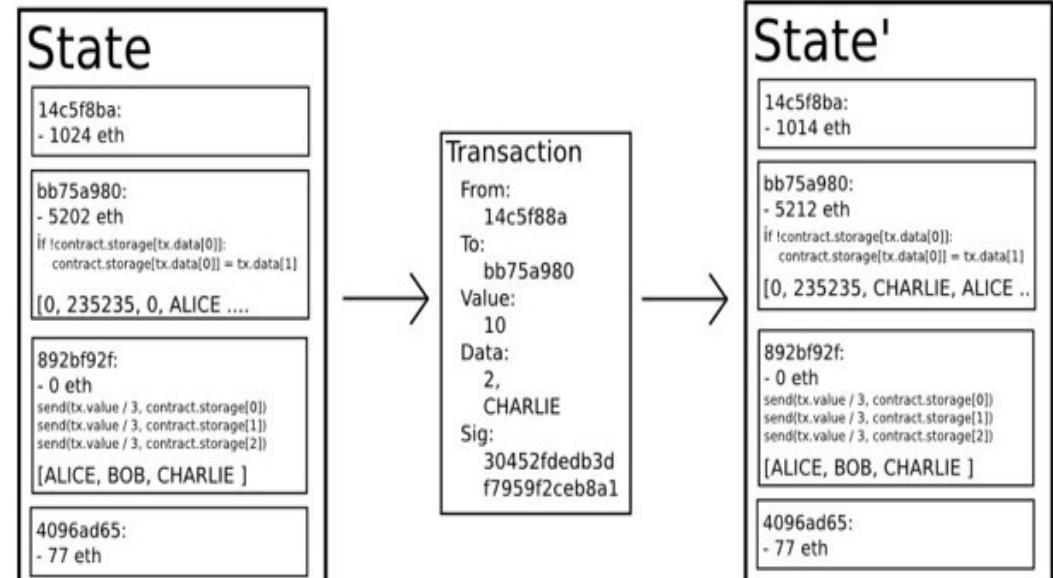


BITCOIN VS ETHEREUM

UTXO VS ACCOUNT



UTXO



Account State

PROGRAMMING LANGUAGES FOR BLOCKCHAINS

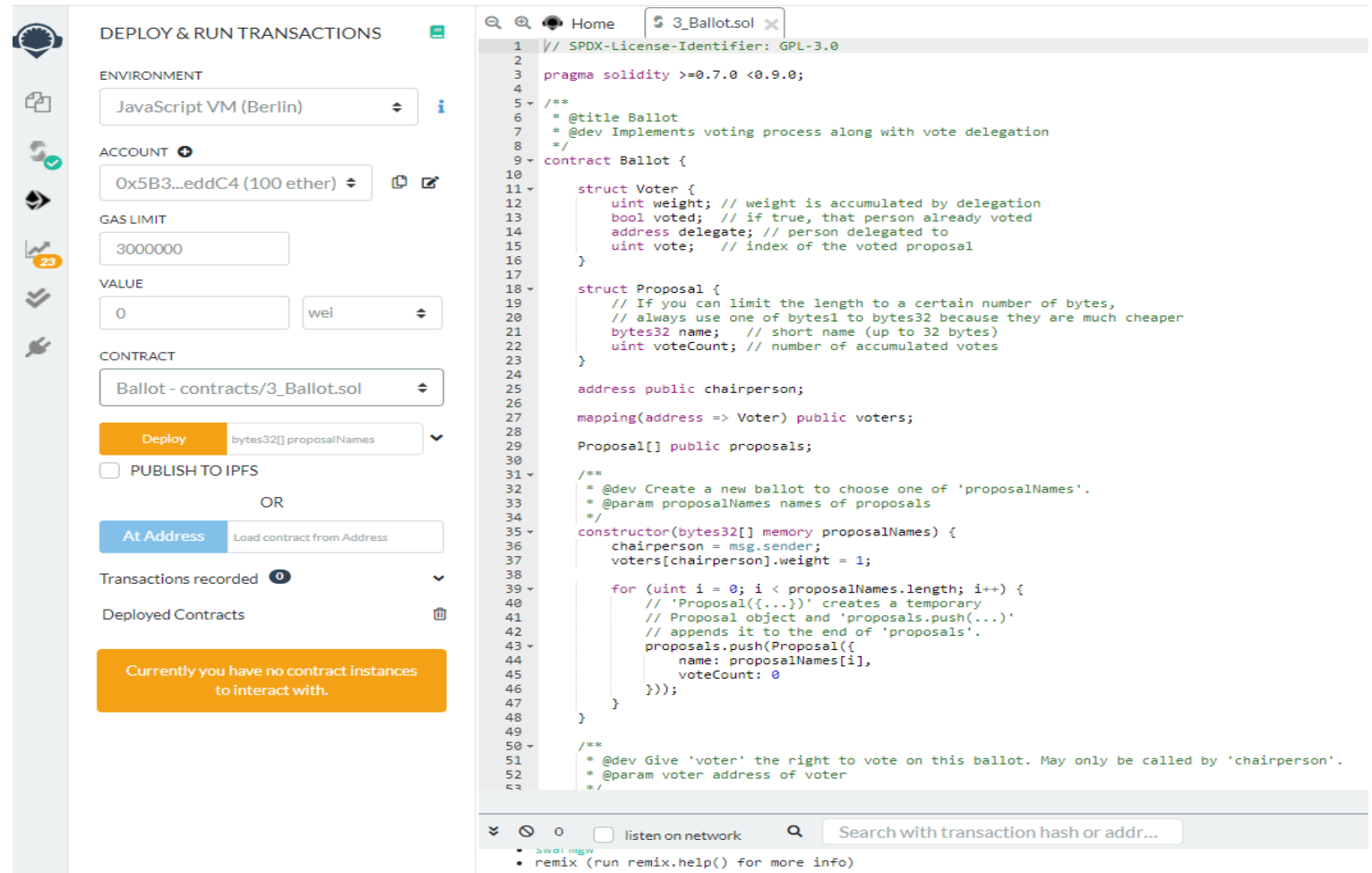
SCL	Study ID	Blockchain	Type-System	Paradigm	Purpose*	Focus ⁺
Reaction-Rule ML	SS1 [30]	-	Static	Declarative	SPEC	Legal Contracts
DSL4SC	SS2 [99]	Hyperledger	Dynamic	Declarative	SPEC	Natural Language
BitML	SS3 [11]	Bitcoin	Dynamic	Declarative	IMPL	Security
eSML	SS4 [77]	-	Dynamic	Declarative	SPEC	Business Process
Findel	SS5 [15]	<i>Independent</i>	Dynamic	Declarative	SPEC	Financial Contracts
BCRL	SS6 [7]	Hyperledger	Dynamic	Declarative	IMPL	Business Process
ADICO	SS7 [44]	Ethereum	Dynamic	Declarative	SPEC	Legal-Contracts
Commit-Rule ML	SS8 [31]	-	Static	Declarative	SPEC	Legal-Contracts
Scilla	SS9 [94]	Zilliqa	Static	Functional	IMPL	Security
SmaCoNat	SS10 [84]	-	Dynamic	Imperative	SPEC	Natural Language
SPESCS	SS11 [54]	-	Dynamic	Declarative	SPEC	Legal-Contracts
Simplicity	SS12 [102]	Bitcoin	Dynamic	Functional	IMPL	Security
Flint	SS13 [91]	Ethereum	Static	Imperative	IMPL	Security
Idris	SS14 [79]	Ethereum	Dependent	Declarative	IMPL	Security
FSolidM	SS15 [69]	<i>Independent</i>	Dynamic	Declarative	SPEC	Security
Marlowe	SS16 [61]	Cardano	Dynamic	Declarative	SPEC	Financial Contracts
Typecoin	SS17 [27]	Bitcoin	Type-Safety	Symbolic	IMPL	Crypto.
QSCL	GL1 [28]	Qtum	Static	Imperative	IMPL	Business Process

PROGRAMMING LANGUAGES FOR BLOCKCHAINS

Pact	GL2 [83]	Kadena	Dynamic	Declarative	IMPL	Security
BALZaC	GL3 [9]	Bitcoin	Dynamic	Imperative	SPEC	Verification
DAML	GL4 [34]	Hyperledger	Dynamic	Declarative	IMPL	Business Process
Ivy	GL5 [86]	Bitcoin	Static	Declarative	IMPL	Domain Specific
Bamboo	GL6 [110]	Ethereum	Type-Safety	Imperative	IMPL	Formal Verf.
Fi	GL7 [5]	Tezos	Type-Safety	Imperative	IMPL	Verification
Liquidity	GL8 [78]	Tezos	Dynamic	Functional	IMPL	Formal Verf.
Michelson	GL9 [101]	Tezos	Monomorphic	Low- Level	IMPL	Domain Specific
Plutus	GL10 [21]	Cardano	Dynamic	Declarative	IMPL	Financial Contracts
Rholang	GL11 [70]	Rchain	Dynamic	Declarative	IMPL	Domain Specific
Vyper	GL12 [20]	Ethereum	Dynamic	Imperative	IMPL	Security
Solidity	GL13 [43]	Ethereum	Static	Imperative	IMPL	Domain Specific
Formality	GL14 [68]	Ethereum	Static	Declarative	IMPL	Efficiency
Pyramid	GL15 [19]	Ethereum	Strongly Typed	Imperative	IMPL	Safety
LLL	GL16 [40]	Ethereum	Dynamic	Declarative	IMPL	User Frnd.
F-Sol	GL17 [85]	Ethereum	Static	Functional	IMPL	Verification
Babbage	GL18 [24]	Ethereum	Type-Safety	Symbolic	SPEC	Human Undr.
Mutan	GL19 [106]	Ethereum	Dynamic	Imperative	IMPL	Formal Verf.
ErgoScript	GL20 [33]	<i>Independent</i>	Type-Safety	Declarative	SPEC	Legal-

REMIK: OFFICIAL DEVELOPMENT ENVIRONMENT

- Remix is used for contract development, as well as learning and teaching Ethereum.
- Solidity contracts can be written using Remix IDE, a powerful open source tool.



The screenshot displays the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel is visible, showing the environment set to 'JavaScript VM (Berlin)', the account as '0x5B3...eddC4 (100 ether)', and the gas limit as '3000000'. The contract selected is 'Ballot - contracts/3_Ballot.sol'. Below this, there are buttons for 'Deploy' and 'At Address', along with a 'PUBLISH TO IPFS' checkbox. A message at the bottom of the panel states: 'Currently you have no contract instances to interact with.'

The right side of the interface shows the Solidity code editor with the following code:

```
1 // SPDX-License-Identifier: GPL-3.0
2
3 pragma solidity >=0.7.0 <0.9.0;
4
5 /**
6  * @title Ballot
7  * @dev Implements voting process along with vote delegation
8  */
9 contract Ballot {
10
11     struct Voter {
12         uint weight; // weight is accumulated by delegation
13         bool voted; // if true, that person already voted
14         address delegate; // person delegated to
15         uint vote; // index of the voted proposal
16     }
17
18     struct Proposal {
19         // If you can limit the length to a certain number of bytes,
20         // always use one of bytes1 to bytes32 because they are much cheaper
21         bytes32 name; // short name (up to 32 bytes)
22         uint voteCount; // number of accumulated votes
23     }
24
25     address public chairperson;
26
27     mapping(address => Voter) public voters;
28
29     Proposal[] public proposals;
30
31     /**
32     * @dev Create a new ballot to choose one of 'proposalNames'.
33     * @param proposalNames names of proposals
34     */
35     constructor(bytes32[] memory proposalNames) {
36         chairperson = msg.sender;
37         voters[chairperson].weight = 1;
38
39         for (uint i = 0; i < proposalNames.length; i++) {
40             // 'Proposal({...})' creates a temporary
41             // Proposal object and 'proposals.push(...)'
42             // appends it to the end of 'proposals'.
43             proposals.push(Proposal({
44                 name: proposalNames[i],
45                 voteCount: 0
46             }));
47         }
48     }
49
50     /**
51     * @dev Give 'voter' the right to vote on this ballot. May only be called by 'chairperson'.
52     * @param voter address of voter
53     */
```

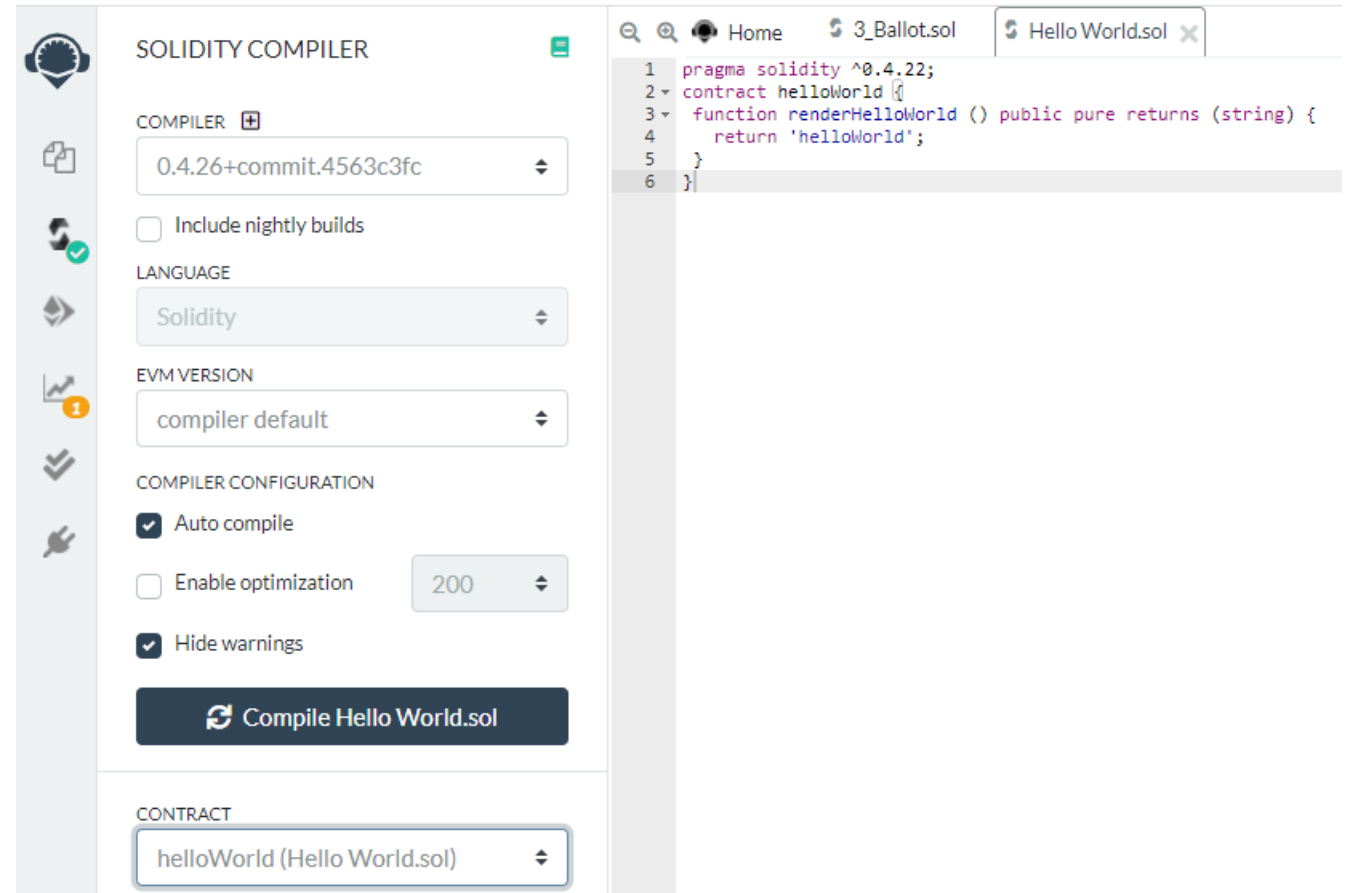
SOLIDITY SOURCE CODE

```
pragma solidity >=0.4.22 <0.6.0; //alt:^0.4.22 => version pragma: defines required compiler version
import "remix_tests.sol"; // this import is automatically injected by Remix. => single line comment
import "./ballot.sol"; library or contract imports (re-use)
/*
write something here multi line comment
*/
contract test3 { contract section
Ballot ballotToTest; object definition
function beforeAll () public { function definition without return value
ballotToTest = new Ballot(2);
}
function checkWinningProposal () public {
ballotToTest.vote(1);
Assert.equal(ballotToTest.winningProposal(), uint(1), "1 should be the winning proposal");
}
function checkWinninProposalWithReturnValue () public view returns (bool) { ...with return value
return ballotToTest.winningProposal() == 1;
}
}
```

EXERCISE 1

Task:

1. Enter Hello World smart contract into remix
2. Compile and debug it
3. Run it and see what happens
4. Switch the compiler version to some 0.4.x → compile + run
5. Change the pragma in code line 1 to 0.4.1 → c&r
6. Change line 4 (string) to (string memory) → c&r
7. Change the pragma + compiler version to higher than 0.5.x → c&r



The screenshot displays the Remix IDE interface. On the left, a vertical toolbar contains icons for Home, Compiler, Run, and other functions. The main panel is divided into two sections: the Solidity Compiler settings and the source code editor.

SOLIDITY COMPILER

- COMPILER**: 0.4.26+commit.4563c3fc
- Include nightly builds
- LANGUAGE**: Solidity
- EVM VERSION**: compiler default
- COMPILER CONFIGURATION**
 - Auto compile
 - Enable optimization (200)
 - Hide warnings
- CONTRACT**: helloWorld (Hello World.sol)

Source Code Editor

```
1 pragma solidity ^0.4.22;
2 contract helloWorld {
3   function renderHelloWorld () public pure returns (string) {
4     return 'helloWorld';
5   }
6 }
```

**TAL
TECH**

Thank you very much for your attention!

Q & A?

Reference: Arumaithurai M., Introduction to Blockchains, Tallinn, Estonia 2019, <https://tinyurl.com/n2y3k5pu>