Blockchain Applications and Smart Contracts

Developing with Ethereum and Solidity

Jim Steele

Pearson

Audience

- Anyone interested in smart contracts
  - Software developers
  - Program managers, blockchain enthusiasts
- Basic understanding of computer programming
  - Linux command line interface
  - Familiar with a scripting language such as JavaScript (preferred)

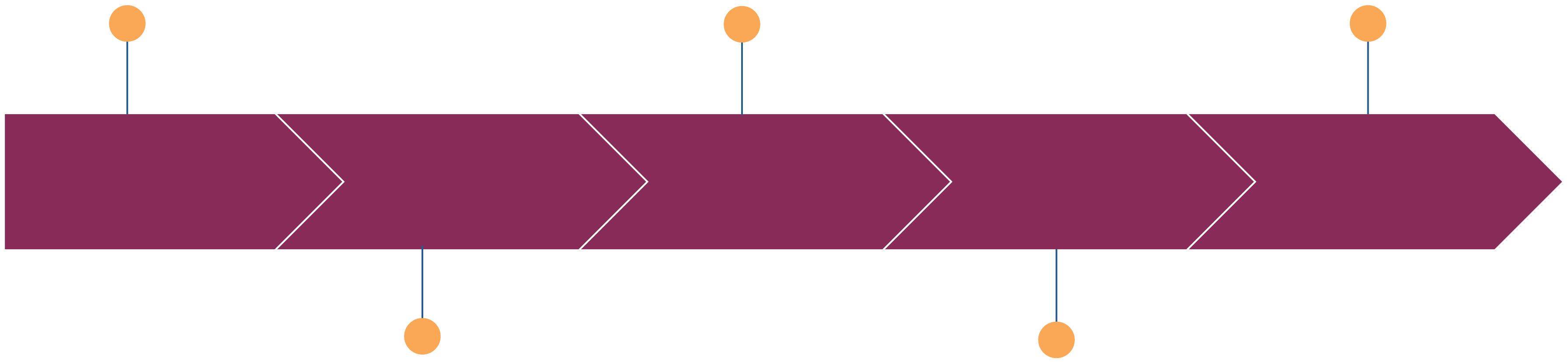
Objective

Learn how to create and run successful applications on the blockchain.

Requirements

- Computer with 2GB RAM and 16GB diskspace (e.g. EC2 instance >= t2.medium)
- Ubuntu 16.04 or similar
- Solidity >= v0.4.4 and Truffle >= v4.0.4
- Network connectivity for full test network
- Live account with Ethereum (if relevant)

Blockchain Applications and Smart Contracts: Course Outline



Understanding blockchain and relevant application use-cases

Ethereum details and the concept of gas

Details of the Solidity language and how to code a smart contract

Testing, debugging, and deploying smart contracts

Review of concepts in the course

Blockchain Applications and Smart Contracts: Course Outline

Understanding blockchain and relevant application use-cases

Ethereum details and the concept of gas

Details of the Solidity language and how to code a smart contract

Testing, debugging, and deploying smart contracts

Review of concepts in the course

Blockchain:

Immutable, unforgeable ledger of assets and transactions

Institutions lower uncertainty allowing two entities to transact without trust, e.g.

- Government issued ID
- Banks and escrows
- Ebay merchant and user reviews

However, these are fragmented with different databases / infrastructure and limited visibility into transactions. Difficult recourse if things go wrong.

Blockchain does not require institutions, instead it is a shared reality across non-trusting entities, and solves some problems of centralized systems:

- Controlled, portable identity
- Transparency
- Public registry, hard if not impossible to tamper with

Satoshi Nakamoto’s Innovation was really Blockchain

```
I've been working on a new electronic cash system that's fully
peer-to-peer, with no trusted third party.

The paper is available at:
http://www.bitcoin.org/bitcoin.pdf

The main properties:
Double-spending is prevented with a peer-to-peer network.
No mint or other trusted parties.
Participants can be anonymous.
New coins are made from Hashcash style proof-of-work.
The proof-of-work for new coin generation also powers the
network to prevent double-spending.
```

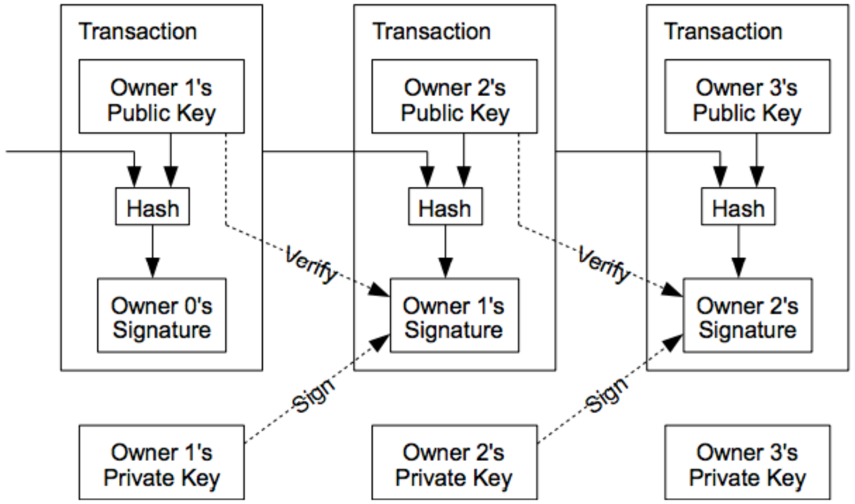
Transactions in a Blockchain

Each transaction digitally signed: more than just electronic signature, a mathematical way to demonstrate authenticity of digital content, e.g. using a public and private key (cryptography)

Electronic coin: chain of digital signatures

- Hash of previous transaction and public key of next owner
- Anyone can verify the chain of ownership

Order of transactions is determined by a collection of servers, or **nodes.**
"Mining" is an ordered selection mechanism



From Satoshi Nakamoto's original bitcoin whitepaper Oct, 2008

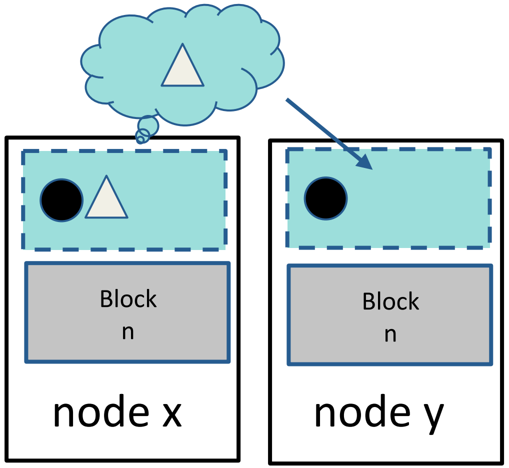
Blockchains

a decentralized way to prevent the Double-spending Problem (1 of 3)

Double-spending avoidance (without a central authority) motivates the need for a **blockchain**:

1. **Publicly announced spending transactions**
2. Each node keeps track of a chain of blocks of transactions (mining)
   a. competes for completed block of transactions (proof of work)
   b. broadcasts each completed block to all other nodes
   c. Accepts broadcast block only if all transactions are not already spent
   d. starts building the next block based on this
3. If any discrepancies exist between nodes, the longest chain wins, and invalid blocks are reverted, abandoning the later duplicate transactions

Overall, the underlying mechanism does not need to be understood, but provides a motivation for many aspects of the technology.



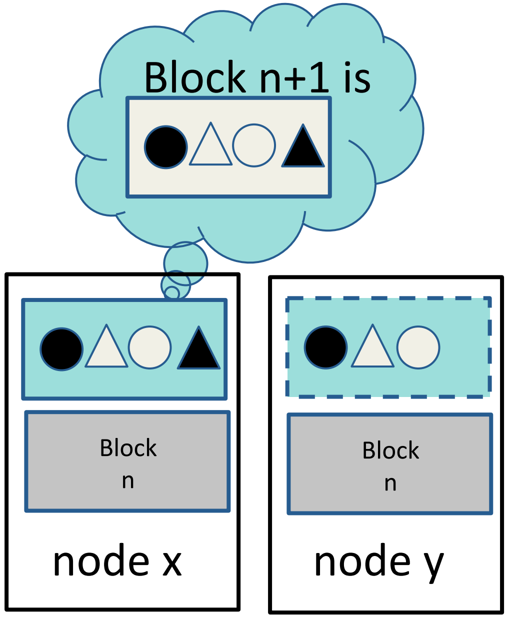
Blockchains

a decentralized way to prevent the Double-spending Problem (2 of 3)

Double-spending avoidance (without a central authority) motivates the need for a **blockchain**:

1. Publicly announced spending transactions
2. **Each node keeps track of a chain of blocks of transactions (mining)**
    a. **competes for completed block of transactions (proof of work)**
    b. **broadcasts each completed block to all other nodes**
    c. **Accepts broadcast block only if all transactions are not already spent**
    d. **starts building the next block based on this**
3. If any discrepancies exist between nodes, the longest chain wins, and invalid blocks are reverted, abandoning the later duplicate transactions

Overall, the underlying mechanism does not need to be understood, but provides a motivation for many aspects of the technology.



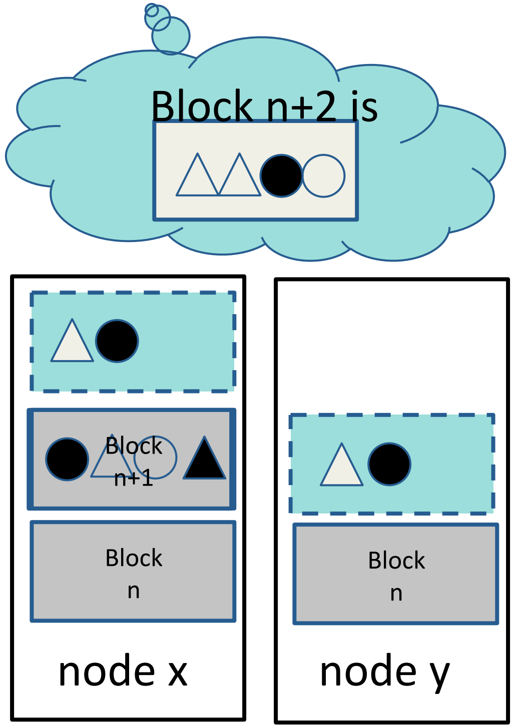
Blockchains

a decentralized way to prevent the Double-spending Problem (3 of 3)

Double-spending avoidance (without a central authority) motivates the need for a **blockchain**:

1. Publicly announced spending transactions
2. Each node keeps track of a chain of blocks of transactions (mining)
   a. competes for completed block of transactions (proof of work)
   b. broadcasts each completed block to all other nodes
   c. Accepts broadcast block only if all transactions are not already spent
   d. starts building the next block based on this
3. **If any discrepancies exist between nodes, the longest chain wins, and invalid blocks are reverted, abandoning the later duplicate transactions**

Overall, the underlying mechanism does not need to be understood, but provides a motivation for many aspects of the technology.



Multiple Blockchains with Different Objectives

Coin	Description	Details
Bitcoin (BTC)	Oldest application of blockchain, contracts not Turing-complete, uses Proof-of-Work to mine	“Digital gold”. Finite amount of coin (21 million by 2140), 10 minutes for 1MB block (6 tx/sec)
Ethereum Classic (ETC)	Added Turing-complete smart contracts	Exploited by hackers, but supporters still keep it alive
Ethereum (ETH)	Fork of Ethereum classic to remove exploitation by hackers, uses Proof-of-Work but migrating to Proof-of-Stake (maybe in 2018?)	“Digital dollar”. Unlimited Ethereum. 15 seconds for each block (size dictated by gas, approx 25 tx/sec), many altcoins based on Ethereum
Litecoin (LTC)	Faster transactions, built on BTC, developers test ideas here since does not alter BTC	“Digital silver”. 2.5 minutes for 1MB block (25 tx/sec), more will move here if any BTC turbulence
Bitcoin Cash (BCH)	Longer blocks allows more transactions per second	10 minutes for 8MB block size (48 tx/sec) and more room for extensions like Omni (altcoin on BTC)
Ripple (XRP)	Real-time gross settlement system backed by banks	Not a blockchain: Truly immutable once ledger closes, 3 second ledger update, 10,000 tx/sec
Nem (XEM)	Private/public blockchain. Proof-of-importance. Take what bitcoin had and apply to all technological infrastructure. Smart assets	Recognized by some Japanese banks, very scalable and low-cost, 1 min / block

Smart Contract Use Cases

Not good:

- Complex programs like machine learning, graphical output, etc. Only put business logic and data crucial for consensus
- Interacts with external service such as the Weather station: every node contacts at different times. Instead use oracle to enter data into the blockchain
- Relies on confidential information

Good:

- Tokenize all valuable assets, and trade these tokens for other tokens or fiat (refinance house without interest)
- Data store representing something which is useful to either other contracts or to the outside world (contract that records membership in an organization)
- Forwarding contract that resends incoming messages to some desired destination only if certain conditions are met (withdrawal limit that is over-rideable via some more complicated access procedure)
- Manage an ongoing contract or relationship between multiple users (escrow with some set of mediators)
- Open contract for any other party to engage with at any time (pay prize to first valid solution to some mathematical problem)

Pearson

Some Interesting Ethereum Projects

Augur, Gnosis: Decentralized prediction market

BoardRoom: Blockchain governance platform

Colony: Platform for autonomous blockchain organizations

BlockApps: Tools to build decentralized apps

Airlock: Keyless access protocol for smart property

Provenance: Gather and share information & stories behind products

Slock.it: Smart locking and billing for the sharing economy

DigixGlobal: Technology to own gold assets

WeiFund: Crowdfunding platform

Maker: Autonomous bank & market maker

HitFin: OTC derivatives settlement

Solidity: Online compiler

Etherparty: Smart contract deployment tools

DappLib: library of math functions

Quiz for Section 1

What is not a requirement of a conventional blockchain?

1. Immutable
2. Public
3. Trust

Blockchain Review

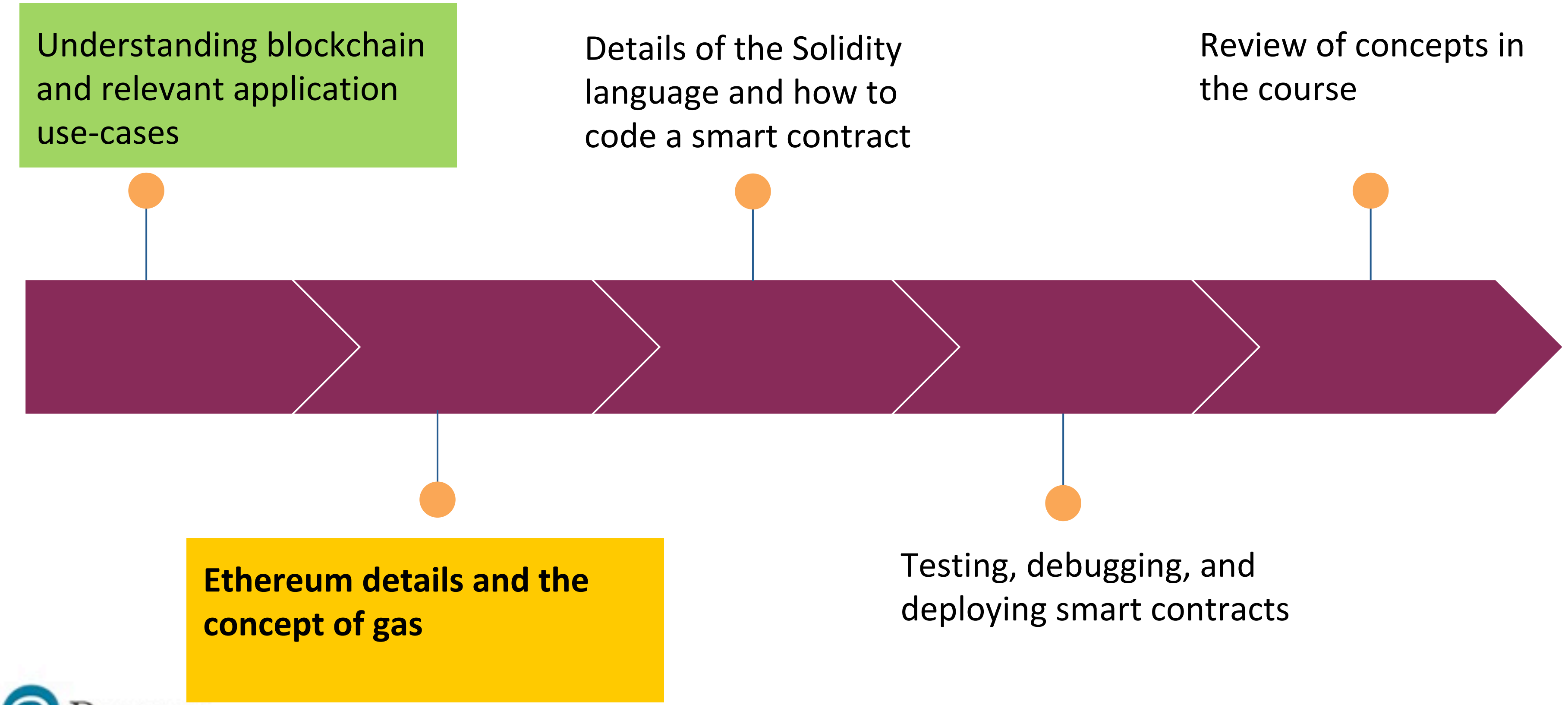
A conventional blockchain is:

Public…by default: a decentralized, peer-to-peer ledger without trust. But private blockchains can be set up in a similar manner

Immutable…over time: consensus is built through mining. Need 6 confirmations to be 99.9% sure of the transaction, so this takes an hour for Bitcoin and 1.5 minutes for Ethereum

Pearson

Blockchain Applications and Smart Contracts: Course Outline


Understanding blockchain and relevant application use-cases
Ethereum details and the concept of gas
Details of the Solidity language and how to code a smart contract
Testing, debugging, and deploying smart contracts
Review of concepts in the course

Ethereum in more Detail

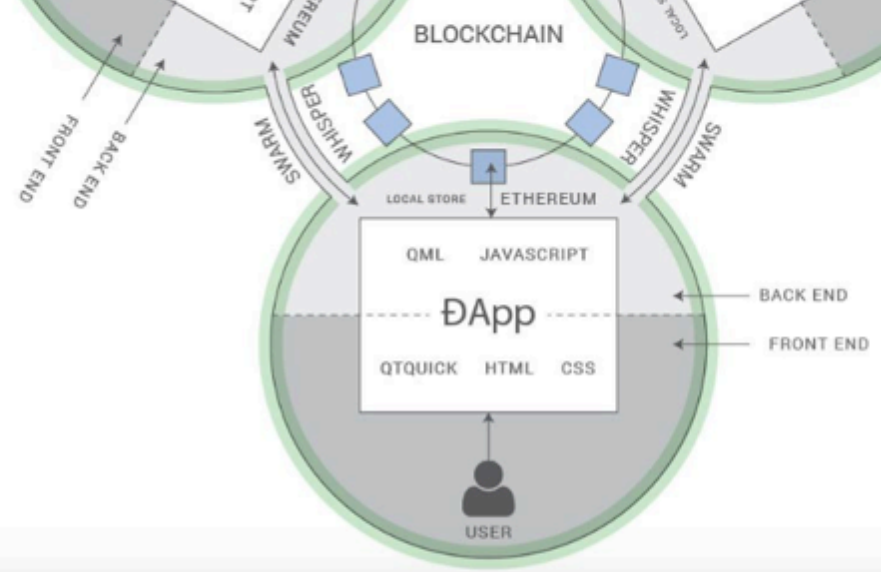
Called the "new web" or **web3.0** (no web servers or middleman).

A platform for smart contracts for decentralized apps (**ĐApps**). Large decentralized computer that knows each user. No logins needed!

- **Ethereum** = computer
- **IPFS** = storage (replaces Swarm)
- **Whisper** = message between ĐApps over time

Ethereum Virtual Machine (EVM):

- runtime environment handling the entire internal state and computation. It is sandboxed and completely isolated
- Every operation is executed on every node in the network.
- Not a register machine, but a stack machine. Stack maximum size of 1024 elements of 256-bits each



Ethereum Hard Fork	Time
Ice Age	September 2015
Homestead (1st production rel)	March 2016
Tangerine Whistle (DAO split)	October 2016
Spurious Dragon	November 2016
Byzantium	October 2017
Serenity (POS)	2018?

Ethereum Addresses

```
var util = require('ethereumjs-util');
var privateKey =
'0x123412341234123412341234123412341234123412341234
1234123412341234';
var publicKey =
util.bufferToHex(util.privateToPublic(privateKey));
var address = '0x' +
util.bufferToHex(util.sha3(publicKey)).slice(26);
```

Two types of accounts share the same address space:

- **Externally owned accounts** (EOAs) controlled by public-private key pairs (i.e. humans), have balance and storage

- **Contract accounts**: have code storage, controlled by that code

How an EOA address is created:

- Private key: 64 hex chars randomly generated
- Public key: 128 hex chars generated from private key using the Elliptic Curve Digital Signature Algorithm
- Public address: last 40 chars of 64 char hash of public key

```
var address = '0x' + util.bufferToHex(util.sha3(publicKey)).slice(26);
web3._extend.utils.isAddress(address)
```

How a Contract account address is created:

- Address generated from creator address and **nonce** (number of transactions sent from that address)

Ethereum Transactions

Message from one account to another
- Can include binary data (payload) and ether
- Can go between EOA and contract accounts

EOA -> EOA: transfer ether
EOA -> 0: create contract from payload
EOA -> contract: run code with data from payload
contract -> EOA: transfer ether
contract -> contract: run code with data from payload

To a contract account: smart contract activates and can
- Return data (like a function call)
- Call other contracts
- Only complete when all pieces complete (cannot move on until that happens)

- Message can contain: source, target, data payload, ether, gas, and return data

- Special type of message: A delegatecall is executed in the context of the calling function (like a library or subroutine)

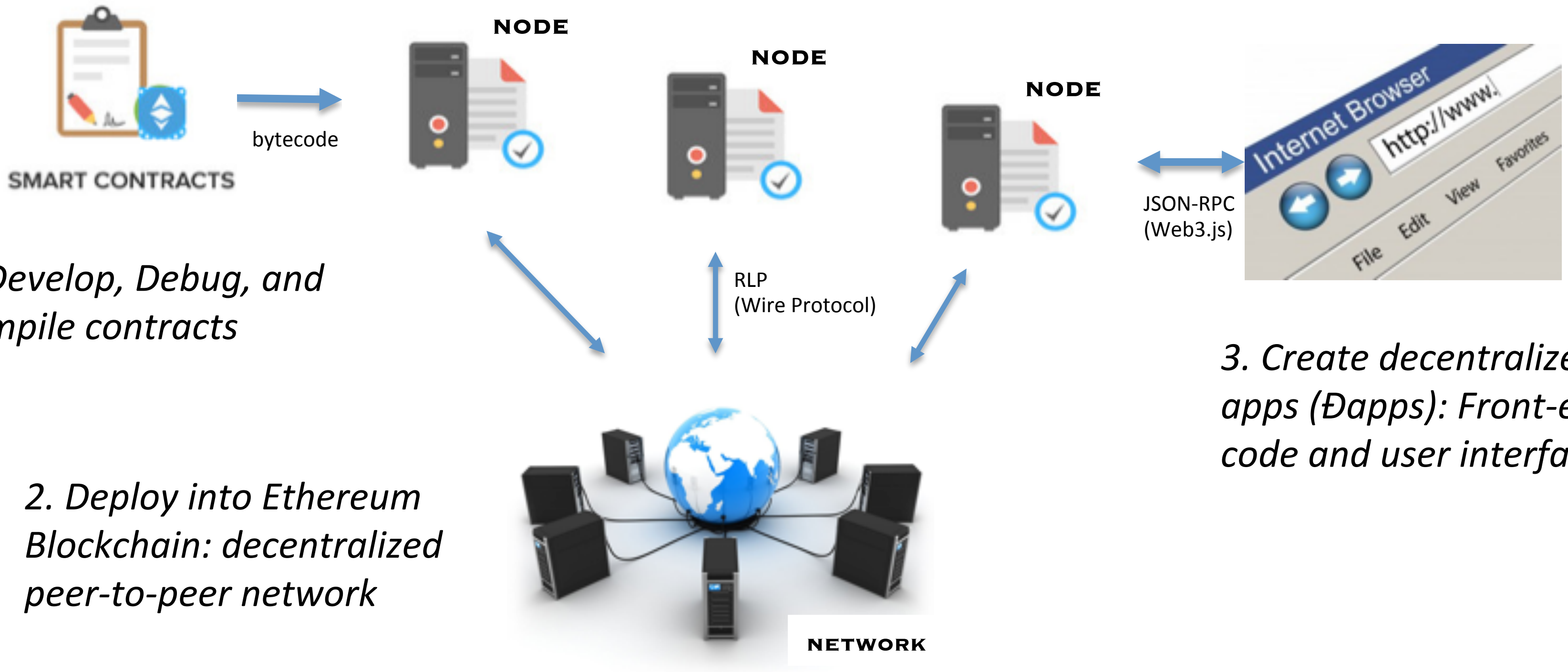
- Special operation: selfdestruct call removes contract from the blockchain

Storage and Memory

- **Storage:**
  - **permanent**
  - Persistent memory inside a contract
  - Costly to read or modify
  - Key/value store (each 32B) in sparse format

- **Memory:**
  - **temporary**
  - Fresh every contract transaction
  - Expanded by 32B chunks as it is accessed
  - Cost quadratically as access grows

Ethereum: How to Run a Decentralized Computer?



1. Develop, Debug, and Compile contracts

2. Deploy into Ethereum Blockchain: decentralized peer-to-peer network

3. Create decentralized apps (Đapps): Front-end code and user interfaces

How avoid malicious or poorly written code taking whole system down?

Ether and the Concept of Gas

Every account has a balance in **Ether (= 10^{18} Wei)**

Every transaction requires **Gas** to execute

> **maxTransactionCost = gasLimit * gasPrice**
>
> **gasLimit:** amount of gas purchased by sender
> **gasPrice:** specified by transactor, around 10^9 Wei

Transaction execution depletes gas according to specific rules:

- If gas runs out: all processing reverts, account still charged
- If any gas left: refunded to the sender's account

Operation Name	Gas Cost	Remark
STEP	1	default amount per execution cycle
STOP	0	free
SUICIDE	5000	permanently kill contract
SLOAD	200	load word from permanent storage
SSTORE	5000	put word into permanent storage
MLOAD	3	load word from memory
MSTORE	3	put word into memory
BALANCE	400	balance of given account
CREATE	53000	changed in homestead from 21000
LOG0	375+8*b	log b bytes of data
SHA3	30+6*w	encode w words of input

Quiz for Section 2

Which statement is false:

1. Ethereum externally owned accounts and contracts coexist in the same address space.
2. In Ethereum, it is more costly to use contract memory than permanent storage.
3. Gas is needed to execute any Ethereum transaction.

Blockchain Applications and Smart Contracts: Course Outline

Understanding blockchain and relevant application use-cases

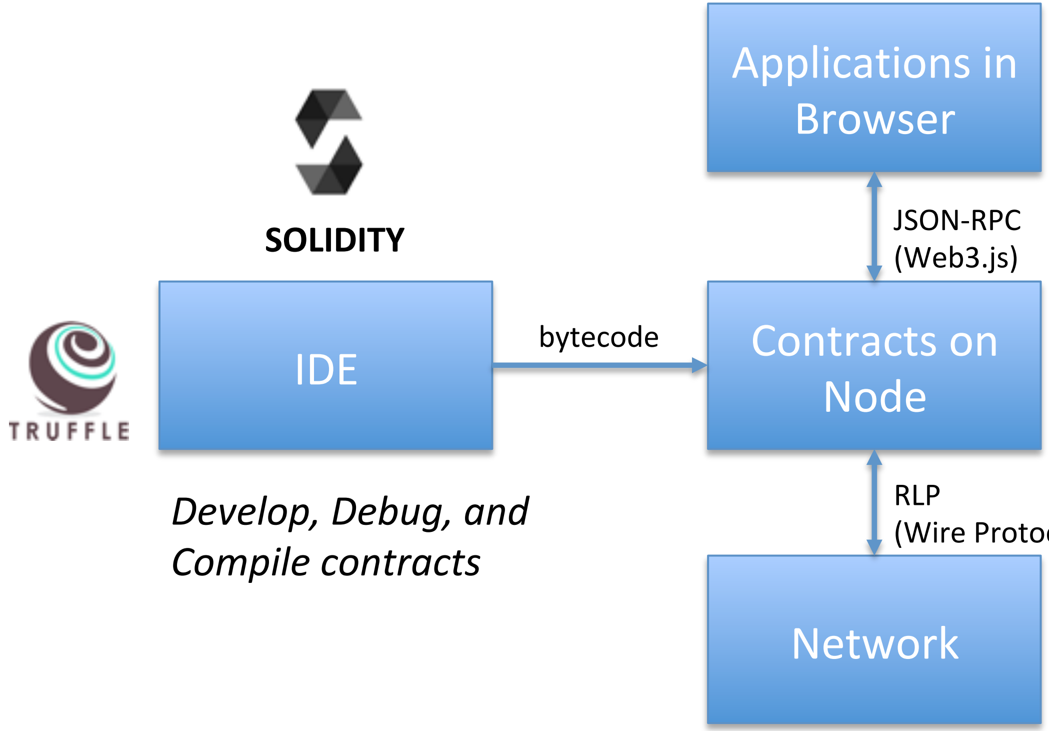
Ethereum details and the concept of gas

Details of the Solidity language and how to code a smart contract

Testing, debugging, and deploying smart contracts

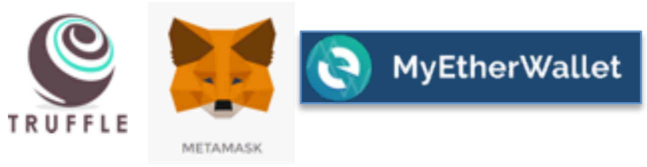
Review of concepts in the course

Ethereum High-Level Architecture



Develop, Debug, and Compile contracts

Đapps: Front-end code and user interfaces



Interface to an Ethereum node to mine ether, transfer funds, and create contracts

Go Ethereum (geth)

Ethereum Blockchain: decentralized peer-to-peer network

Ethereum Client and Network Types

- Personal blockchain: Ganache
  - Access through standalone client (e.g. ganache-cli previously testrpc)
  - Instantly processes transactions for quick development

Go Ethereum (geth)

- Public blockchain: geth, parity
  - Access through Ethereum client
  - Different networks to access (see right)
  - Mining needed to process transactions and deploy contracts

<u>Ethereum Networks</u>

- 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 Ethereum testnet
- 42: Kovan, the public Parity Ethereum testnet
- 77: Sokol, the public POA testnet
- 99: POA, the public Proof of Authority Ethereum network

Solidity in Detail

Solidity: probably the first "Contract-Oriented Language" and reworks the well-established Object-Oriented Languages (similar to Javascript or C++)

Some Similarities to Object-Oriented Languages:

- Contains persistent data in state variables and functions that can modify these variables
- Includes common Object-Orientated concepts such as Inheritance, Abstract Contracts, and Interfaces
- Calling a function on a different contract (instance) will perform an EVM function call and thus switch the context such that the calling contract state variables are inaccessible from the called contract

Also, Solidity contracts run solely on a blockchain and have important differences:

- Access blockchain data (such as this.balance and block.number)
- Execution depends on mining latency, use events to see results, utilizing logging
- Costs money, must be robust against reentrant issues
- Immutable, be sure no bugs or erratic behavior that uses gas. Utilize error checking and importance of unit testing
- Still evolving!

Example Solidity Contract

```
migrate --reset
var gns = GetAndSet.at(GetAndSet.address)
gns.getStoredData(1)
gns.setStoredData(1,2018)
gns.getStoredData(1)
```

```
pragma solidity ^0.4.4;

/// @title Simple example of setting and getting storage data
/// @author Dev Name
/// @dev Storage is costly. Only use for critical data.
contract GetAndSet {
   uint16[3] storedData;

   function setStoredData(uint8 n, uint16 x) {
      storedData[n] = x;
   }

   function getStoredData(uint8 n) public view returns (uint16) {
      return storedData[n];
   }
}
```

Note:

- Pragma
- Comments and Natspec
- Contract like a class
- State variables located in storage, function parameters located in memory
- Functions are executable units in a contract
- New keywords such as “view”

Solidity Declarations

Types:

- **bool:** Boolean can be either true or false
- **int / uint:** signed and unsigned integers from 8 to 256 bits (int and uint are aliases for int256 and uint256)
- **fixed / ufixed:** signed and unsigned fixed point numbers of various sizes (fixedMxN has M bits and N decimal points) (fixed is alias for fixed128x19)
- **address:** 160-bit non-arithmetic value. Has member functions “balance”, “transfer”, (“send”, “call”, “delegatecall”)
- **bytesN:** N bytes (byte is alias for bytes1).
- **bytes:** is a dynamically-sized array (more costly)
- **string:** dynamically-sized UTF-8 encoded string

Modifiers:

- **constant:** deprecated for functions, means variable is constant at compile time
- **pure:** does not read or change the storage state
- **view:** like “constant” only reads the storage state
- **payable:** can receive Ether through send or transfer, needs fallback function

Visibility:

- **external:** can be accessed from other contracts, efficient for large arrays of data
- **internal:** can only be accessed from inside the contract, default
- **public:** can be accessed outside the contract, default
- **private:** not in derived contracts, but information still visible to all external observers

Pearson

Public State Variables

```
<<set variable as public, try again without
getter>>
migrate --reset
var gns = GetAndSet.at(GetAndSet.address)
gns.storedData(1)
gns.setStoredData(1,2018)
gns.storedData(1)
```

```
pragma solidity ^0.4.4;

/// @title Simple example of setting and getting storage data
/// @author Dev Name
/// @dev Storage is costly. Only use for critical data.
contract GetAndSet {
   uint16[3] public storedData;

   function setStoredData(uint8 n, uint16 x) {
      storedData[n] = x;
   }

   ~~function getStoredData(uint8 n) public view returns (uint16) {~~
      ~~return storedData[n];~~
   ~~}~~
}
```

Declaring state variable as public basically adds a getter. For example:

```
uint256 public value1;
```

Makes an implicit function:

```
function value1() returns (uint256) {
        return value1;
}
```

Accessed internal manner: value1
Accessed external manner: this.value1()

Function Modifiers

- Function modifiers (like asserts or decorate patterns) are inherited and can be overridden
- Use "_;" to refer to main body of code in the method being modified
- Some examples of function modifier use on the right
- How would you write a modifier to protect a contract from reentrant calls?

```
enum State { Created, Locked, Inactive }

modifier isOwner() {
   if (msg.sender != owner) { throw; }
   _; // continue executing rest of method body
}

modifier isState(State _state) {
   require(state == _state);
   _;
}

modifier cleanUp() {
  _;  // finish running method body
  // clean up code
}

doSomething() isOwner isState(State.Created) cleanUp {
  // code
}
```

Function Modifiers Example: mutex

```
contract ExampleWithMutex {
	bool locked;
	modifier noReentrancy() {
		require(!locked);
		locked = true;
		_;
		locked = false;
	}
	/// This function is protected by a mutex, which means that
	/// reentrant calls from within `msg.sender.call` cannot call `f` again.
	/// The `return 7` statement assigns 7 to the return value but still
	/// executes the statement `locked = false` in the modifier.
	function f() public noReentrancy returns (uint) {
		require(msg.sender.call());
		return 7;
	}
}
```

Pearson

Error Checking

Handling of errors is very important in Solidity. The old way could sacrifice gas (used prior to Solidity 0.4.10):

```
if(msg.sender != owner) { throw; } //do not use
```

Three ways to check for errors:

1. **require**
   - Refunds remaining gas to caller
   - Use to validate inputs or state conditions
   - Can return a value
   - Most often used towards beginning of function

   require(msg.sender == owner);

2. **revert**
   - Refunds remaining gas to caller
   - Use same as require, but for more complex logic
   - Can return a value

   if(msg.sender != owner) { revert(); }

3. **assert**
   - Gas sacrificed to the asserted transaction
   - Use to check overflow/underflow
   - Should never be reached…an error in the code
   - Most often used towards end of function

   assert(this.balance > totalSupply);

Other Solidity Particulars

- Fallback function: **`function()`**
  - Cannot take arguments but can access msg.data
  - Used when just Ether is sent to the contract. Need this function to be payable to receive the Ether!
  - Also used when called function does not exist. Avoids the default behavior of throwing an exception

- Blockchain data accessible from the contract:
  - now
  - this.balance, <address>.balance
  - block.number, block.timestamp, etc.
  - msg.value, msg.sig, msg.data
  - tx.gasprice, tx.origin
  - gasleft()

- *Structs are allowed*

```
struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted; // if true, that person already voted
        address delegate; // person delegated to
        uint vote; // index of the voted proposal
}
```

- *Mappings are allowed with* mapping(_KeyType => _ValueType)

```
// This declares a state variable that
// stores a `Voter` struct for each possible address.
mapping(address => Voter) public voters;
```

- *Arrays are allowed*

```
// A dynamically-sized array of `Proposal` structs.
Proposal[] public proposals;
```

Printing and Events

- Print does not make sense in decentralized applications
- Low-level log messages log1, log2, … are not well supported by IDE’s
- Events provide help with debugging as well as useful in production code
  - Up to three indexed parameters (i.e. searchable)
  - Code in contract to emit event
    - event myLogMessage(address indexed d, uint val);
    - myLogMessage(msg.sender, quality);
  - Code in Client to watch for event
    - var event = myContract.myLogMessage ();
    - event.watch(function(error,result){/* some callback */});

Quiz for Section 3

- True or False: Contract anotherOne cannot retrieve the value of data1 since it is private.
- Which functions can anotherOne call? How about yetAnotherOne?

```
contract addOne {
  uint private data1;
  function getData() returns (uint) { return data1; }
  function f1(uint a) private returns (uint) { return a+1; }
  function f2(uint a) internal { data1=f1(a); }
}
contract anotherOne is addOne {
   function g(uint a) { addOne.f1(a); }
}
contract yetAnotherOne {
   function g(uint a) { addOne.f2(a); }
}
```

Blockchain Applications and Smart Contracts: Course Outline

- Understanding blockchain and relevant application use-cases
- Ethereum details and the concept of gas
- Details of the Solidity language and how to code a smart contract
- **Testing, debugging, and deploying smart contracts**
- Review of concepts in the course

Pearson

Creating and Testing a Contract

1. > truffle init
2. Update truffle.js with the development network
3. Create contracts
4. Update 2_deploy_contracts.js referencing the contracts
5. > ganache-cli &
6. > truffle migrate --reset
7. > truffle test

Make sure to test contracts!

Deploying a Contract on Live Testnet

1. Update truffle.js with the live test network
2. > geth --rpc --rpcapi "eth,net,web3" console
3. Wait for blockchain sync
4. geth> eth.accounts (if none, use "personal.newAccount()")
5. geth> personal.unlockAccount(web3.eth.coinbase, "password", 15000)
6. > truffle migrate --network live

Make sure to save account password information!

Blockchain Applications and Smart Contracts: Course Outline

- Understanding blockchain and relevant application use-cases
- Ethereum details and the concept of gas
- Details of the Solidity language and how to code a smart contract
- Testing, debugging, and deploying smart contracts
- **Review of concepts in the course**

Review of Concepts in the Course

- Blockchain: Immutable, unforgeable ledger of assets and transactions
- Ethereum: A platform for smart contracts for decentralized apps
- Solidity: “Contract-Oriented Language” and reworks the well-established Object-Oriented Languages
- Truffle IDE with Ganache client provide a simple development platform
- Truffle IDE with Geth client can access both testnet and live networks to deploy smart contracts

Thank you!

My page https://medium.com/@jim.steele

References

- What gas cost is per operation https://github.com/djrtwo/evm-opcode-gas-costs and What miners are accepting for gas price https://ethgasstation.info/index.php
- Self-documenting comments specification (natspec) https://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format
- Solidity best practices https://consensys.github.io/smart-contract-best-practices and https://github.com/ConsenSys/Ethereum-Development-Best-Practices