Blockchain Course

# Ethereum


fausto.spoto@univr.it

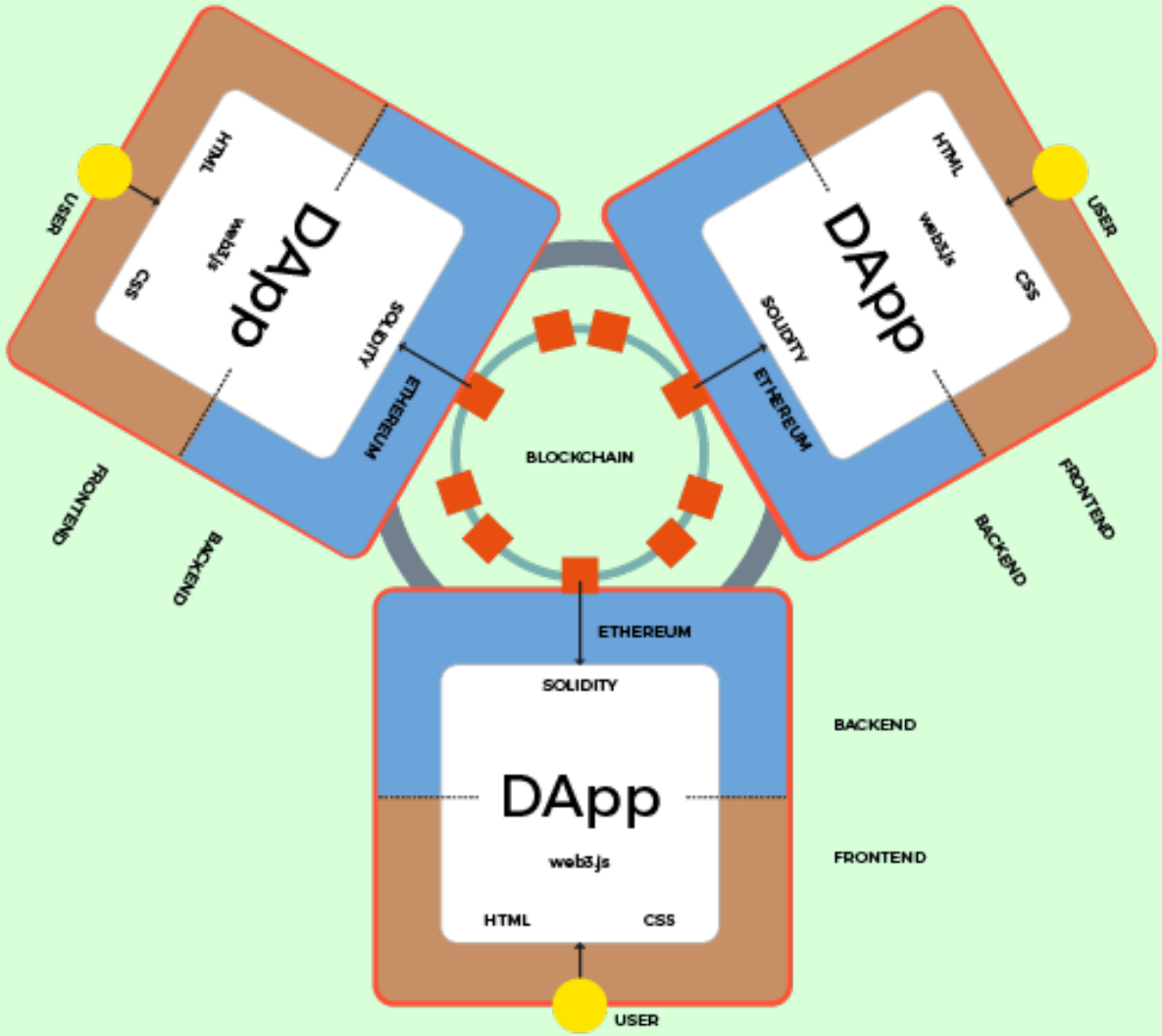
https://github.com/spoto/blockchain-course

# Thw world computer

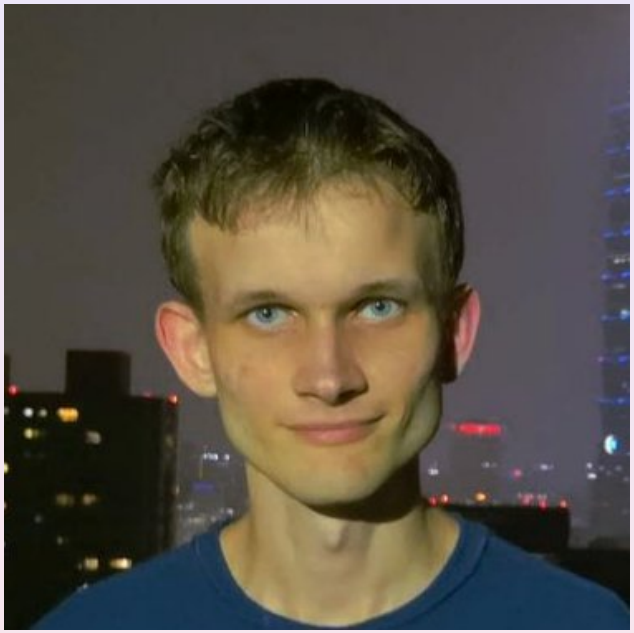
An open source, globally decentralized computing infrastructure that executes programs called smart contracts, written in a Turing-complete programming language, translated into bytecode and run on a virtual machine. It uses a blockchain to synchronize and store the system's singleton state changes (key/value tuples), along with a cryptocurrency called ether to meter and constrain execution resource costs. It enables developers to build decentralized applications with built-in economic functions

# DApps

## DApps = smart contracts (Solidity) + web3 frontend (JavaScript. . . )



# People behind Ethereum

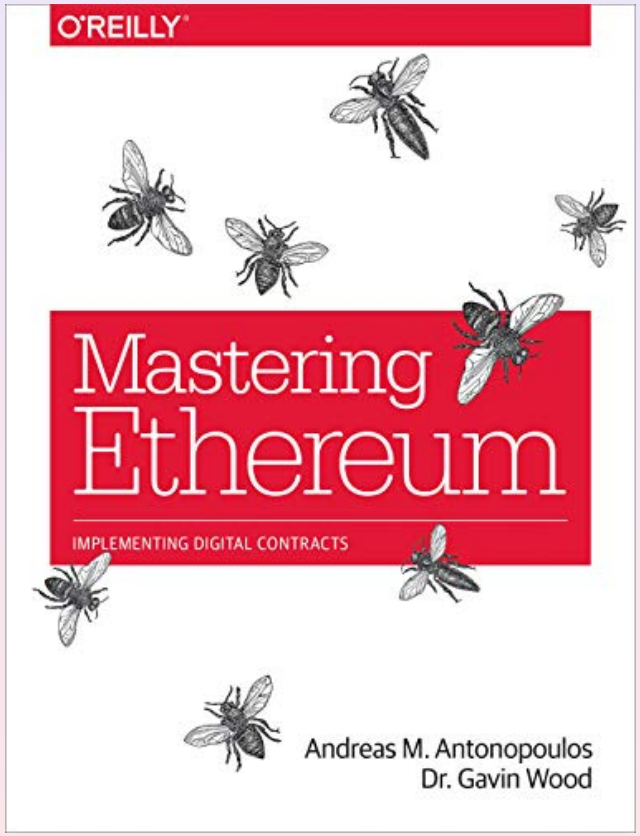


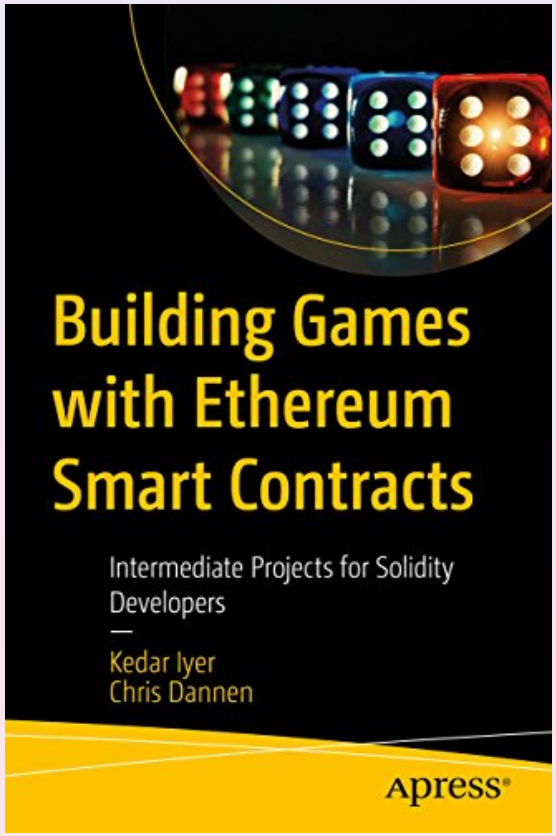
Vitalik Buterin

Gavin Wood

# References used in this course

Yellow Paper:
`https://ethereum.github.io/yellowpaper/paper.pdf`




The leftmost: `https://github.com/ethereumbook/ethereumbook`

# Deterministic (infinite) state machine

## A very abstract view of blockchain

A blockchain is a distributed ledger of transaction requests, aggregated in blocks

- Bitcoin: transaction requests require a change of the set of UTXOs
- Ethereum: transaction requests require a change of a map $key \rightarrow value$

The change must be deterministic otherwise consensus cannot be reached!

# Ether denominations and unit names

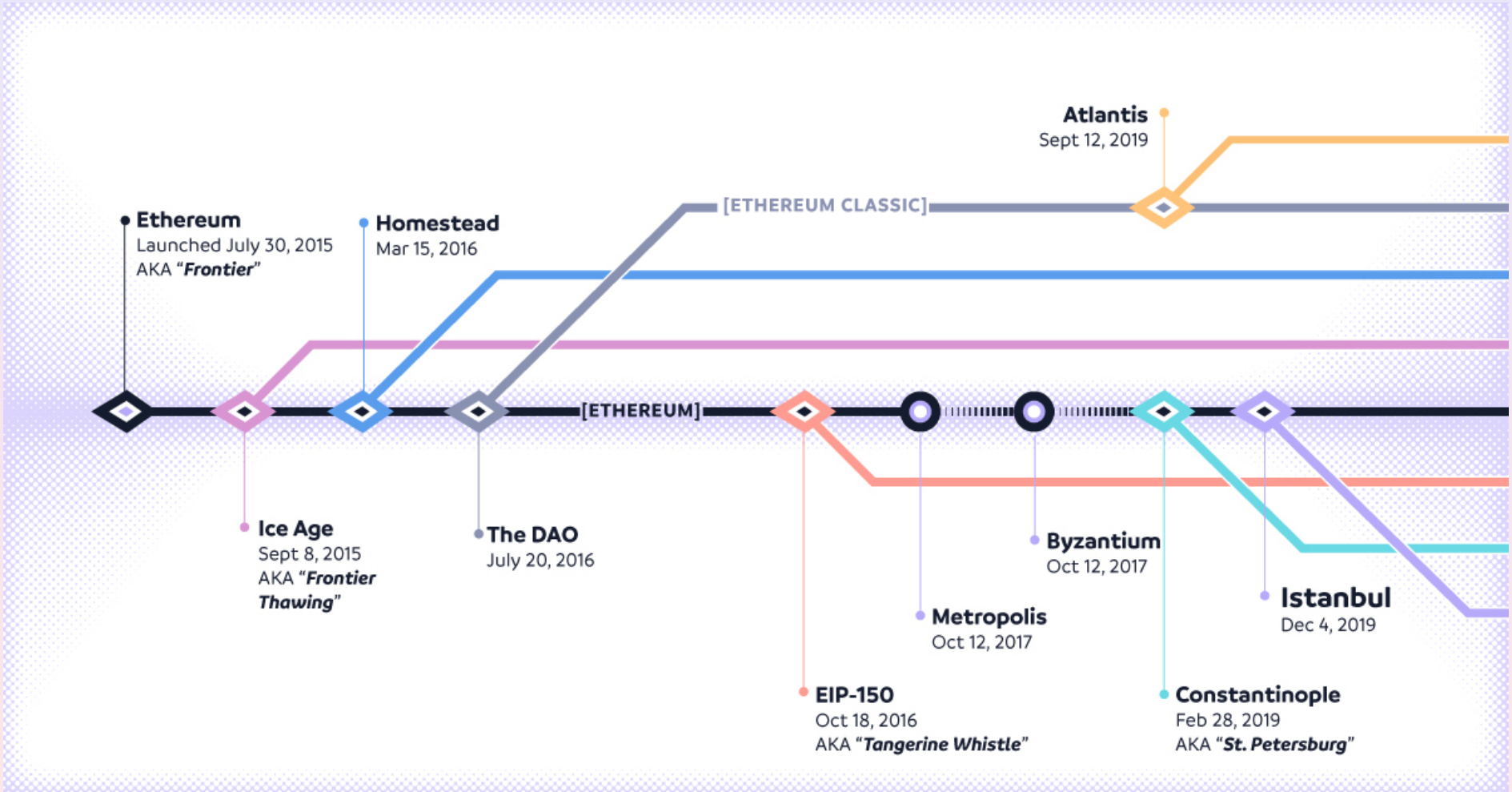
Table 1. Ether Denominations and Unit Names

| Value (in wei) | Exponent | Common Name | SI Name |
|---|---|---|---|
| 1 | 1 | wei | wei |
| 1,000 | $10^3$ | babbage | kilowei or femtoether |
| 1,000,000 | $10^6$ | lovelace | megawei or picoether |
| 1,000,000,000 | $10^9$ | shannon | gigawei or nanoether |
| 1,000,000,000,000 | $10^{12}$ | szabo | microether or micro |
| 1,000,000,000,000,000 | $10^{15}$ | finney | milliether or milli |
| *1,000,000,000,000,000,000* | *$10^{18}$* | *ether* | *ether* |
| 1,000,000,000,000,000,000,000 | $10^{21}$ | grand | kiloether |
| 1,000,000,000,000,000,000,000,000 | $10^{24}$ | | megaether |

# Ether (ETH) price over the years

# Ethereum Classic Ether (ETC) price over the years

# Ethereum forks



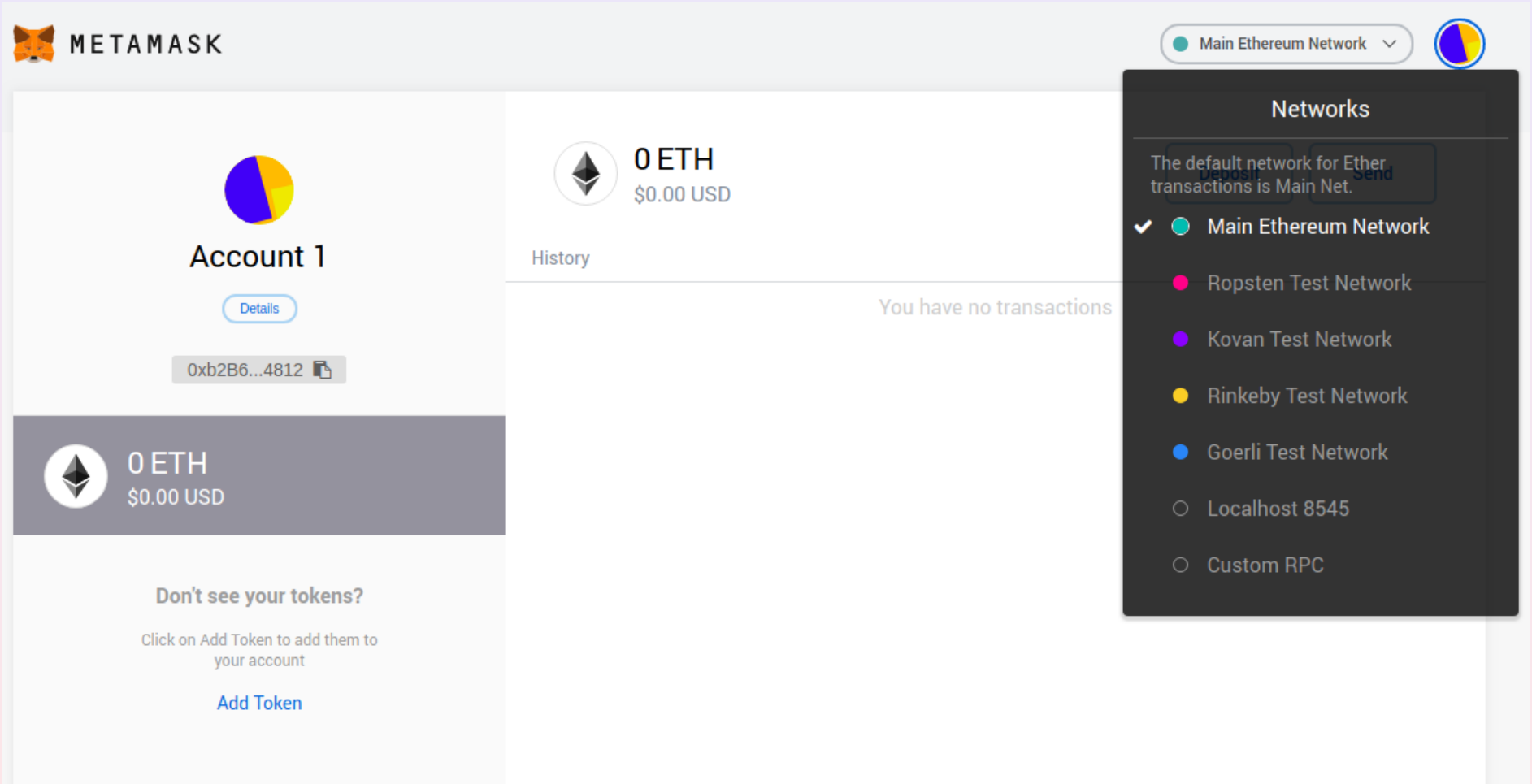
# Wallets

A wallet is a software application that helps you manage your Ethereum account(s), by keeping your keys, creating and broadcasting transactions:

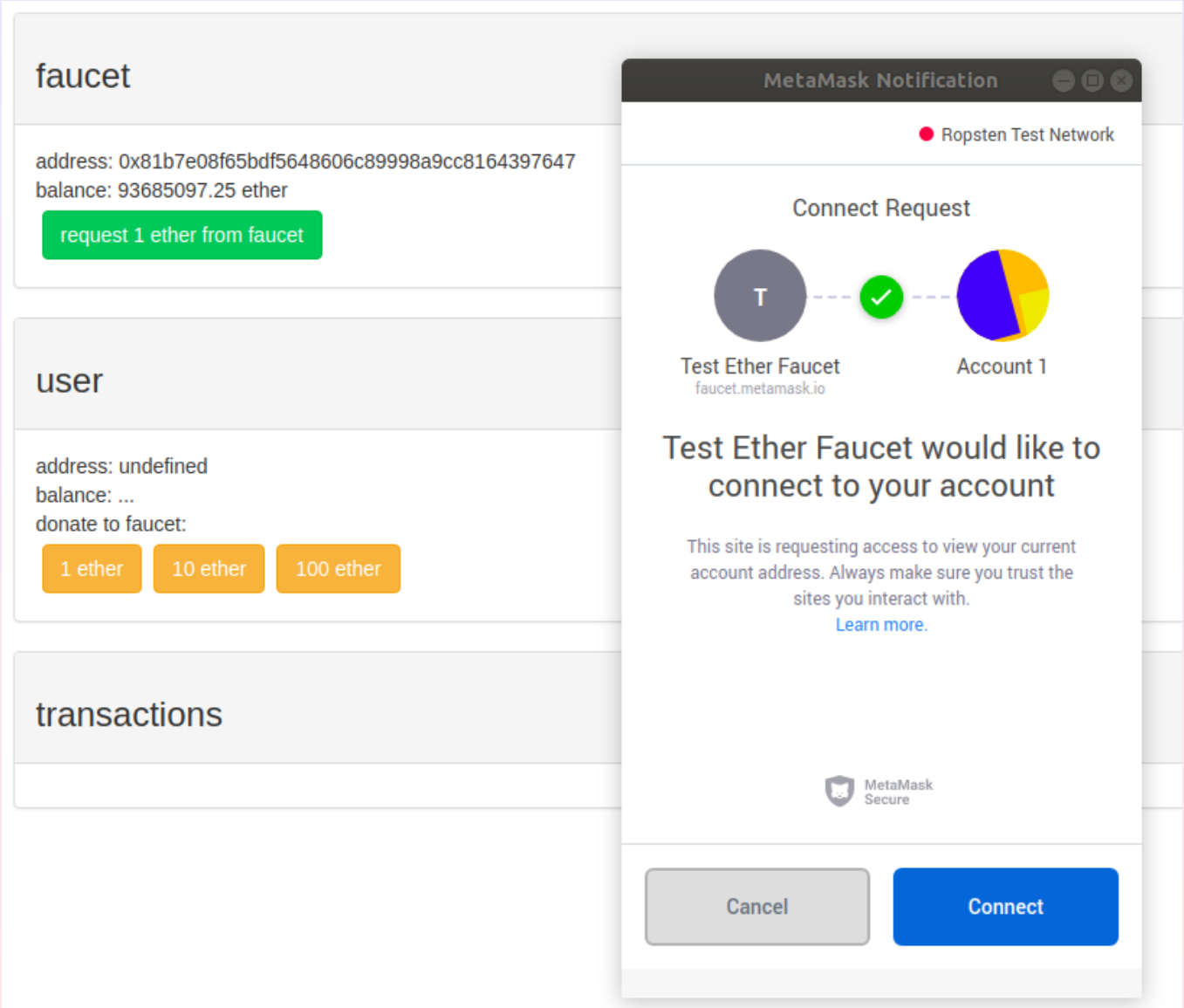
- mobile (Jaxx)
- desktop (Jaxx, Emerald Wallet)
- web-based (MetaMask, MyEtherWallet)

# MetaMask: create account

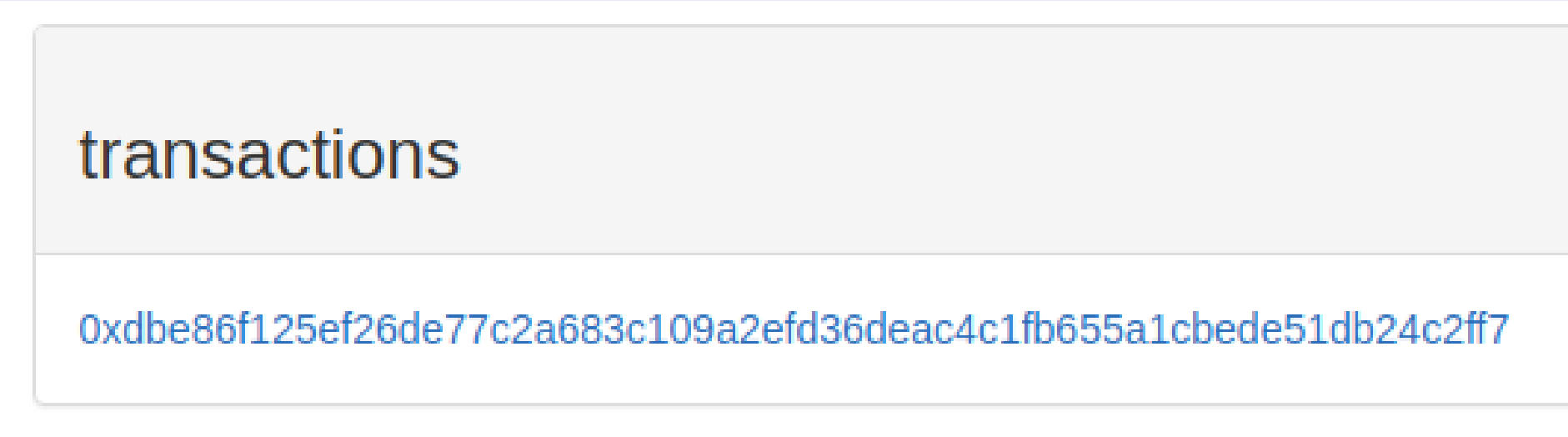
`https://metamask.io`



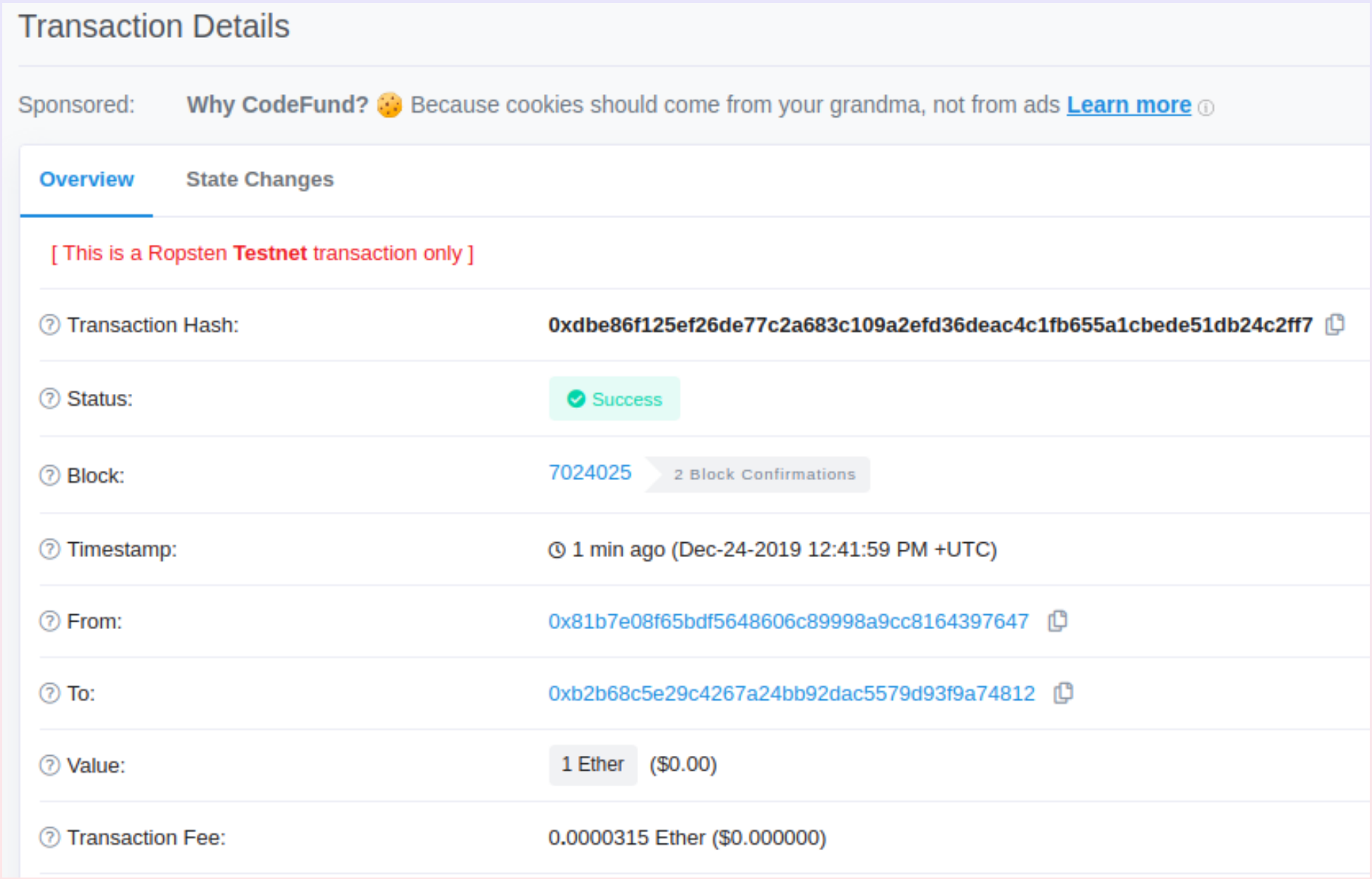
# MetaMask: connect to the faucet



# MetaMask: transaction for getting 1 ETH



# MetaMask: detail of the transaction



# MetaMask: transactions of our account

Transactions

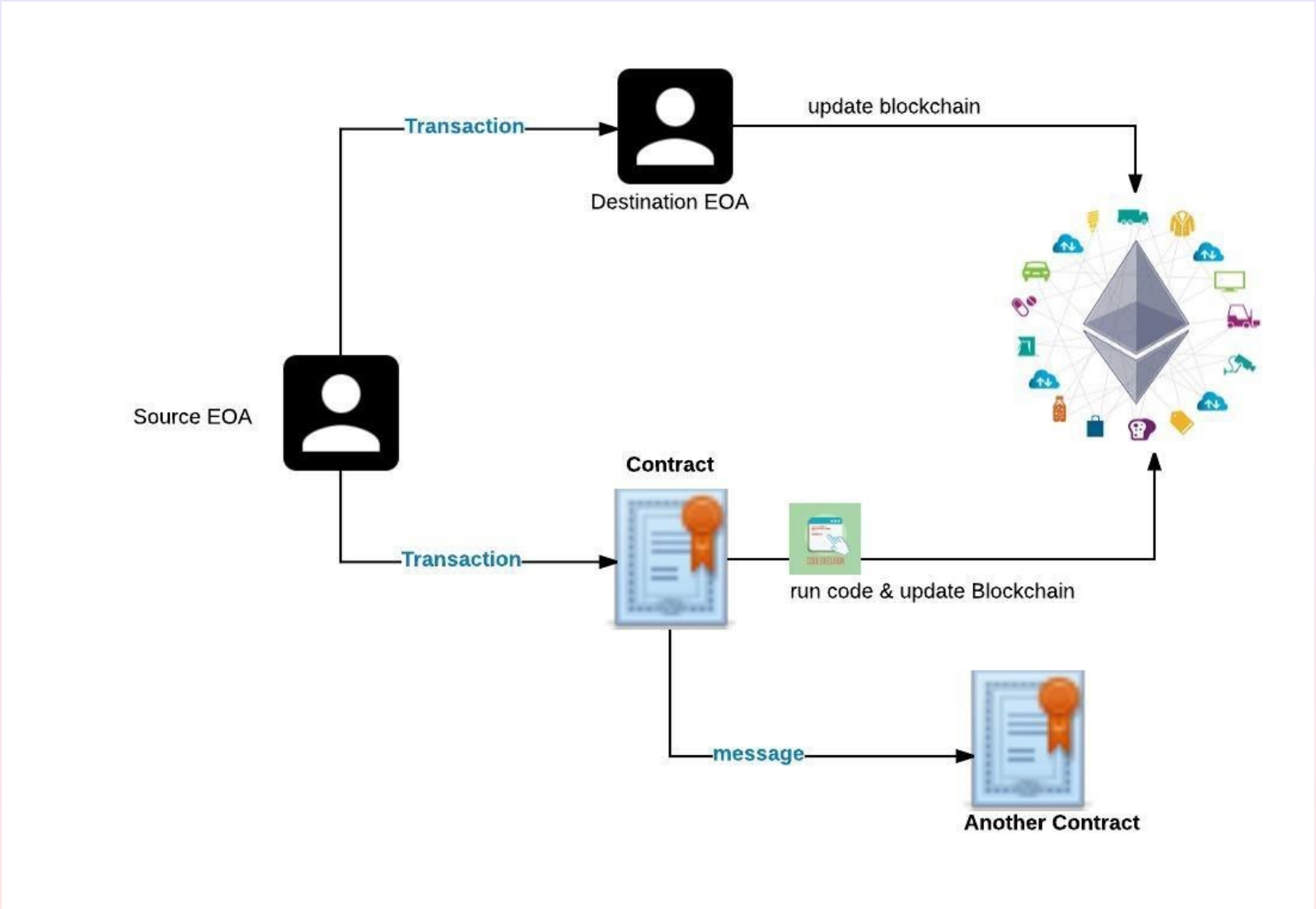
Latest 3 txns

| Txn Hash | Block | Age | From | | To | Value | [Txn Fee] |
|---|---|---|---|---|---|---|---|
| 0xebe816e6acd493... | 7024328 | 2 mins ago | 0xb2b68c5e29c426... | OUT | 0x81b7e08f65bdf56... | 1 Ether | 0.000084 |
| 0x5adec738dc2d85... | 7024325 | 3 mins ago | 0x81b7e08f65bdf56... | IN | 0xb2b68c5e29c426... | 1 Ether | 0.0000315 |
| 0xdbe86f125ef26de... | 7024025 | 1 hr 14 mins ago | 0x81b7e08f65bdf56... | IN | 0xb2b68c5e29c426... | 1 Ether | 0.0000315 |

[ Download CSV Export ]

# Externally owned accounts (EOA) and contracts

EOAs have keys, contracts have code, both have an address

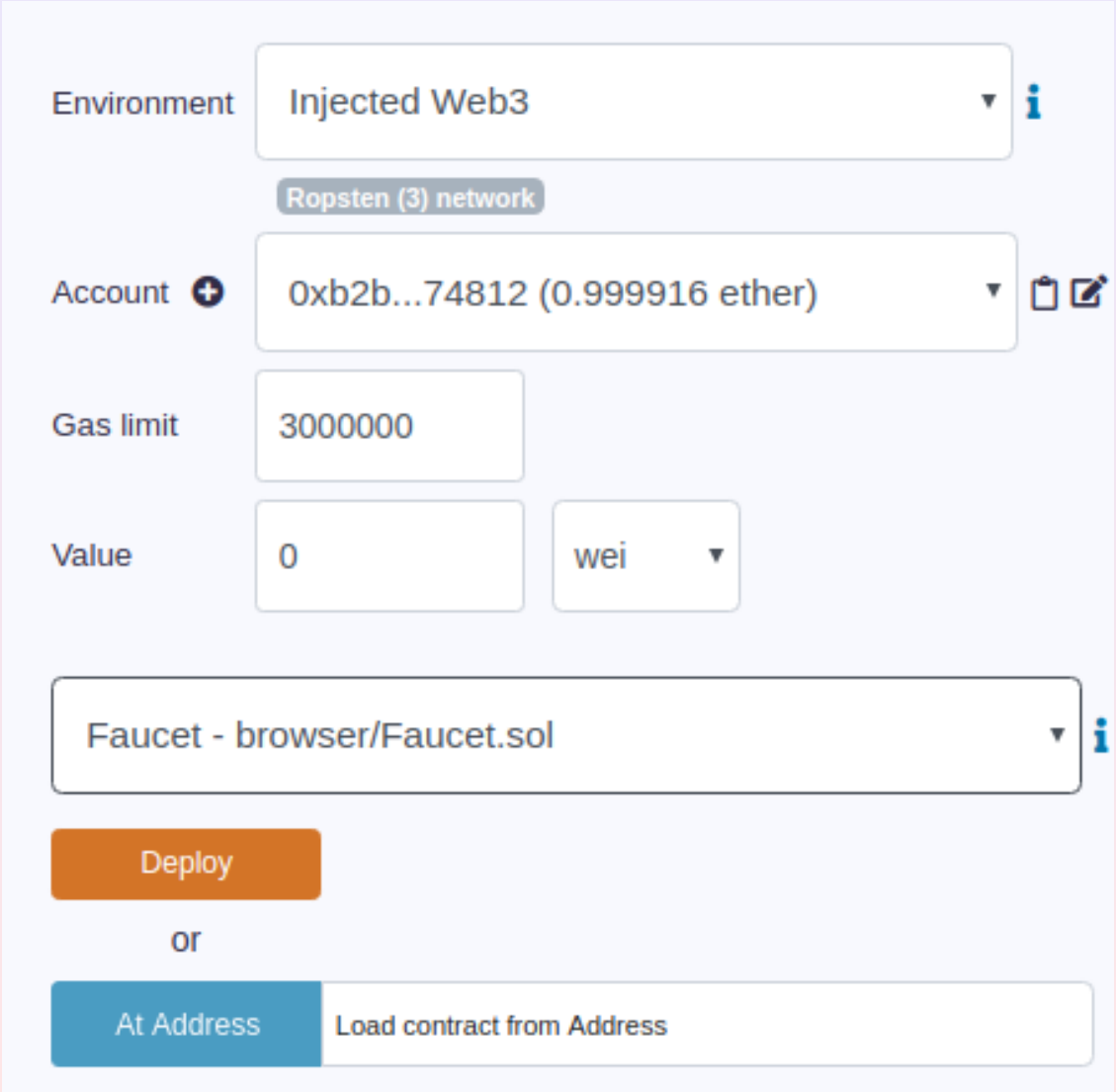


# A web IDE for Solidity: https://remix.ethereum.org

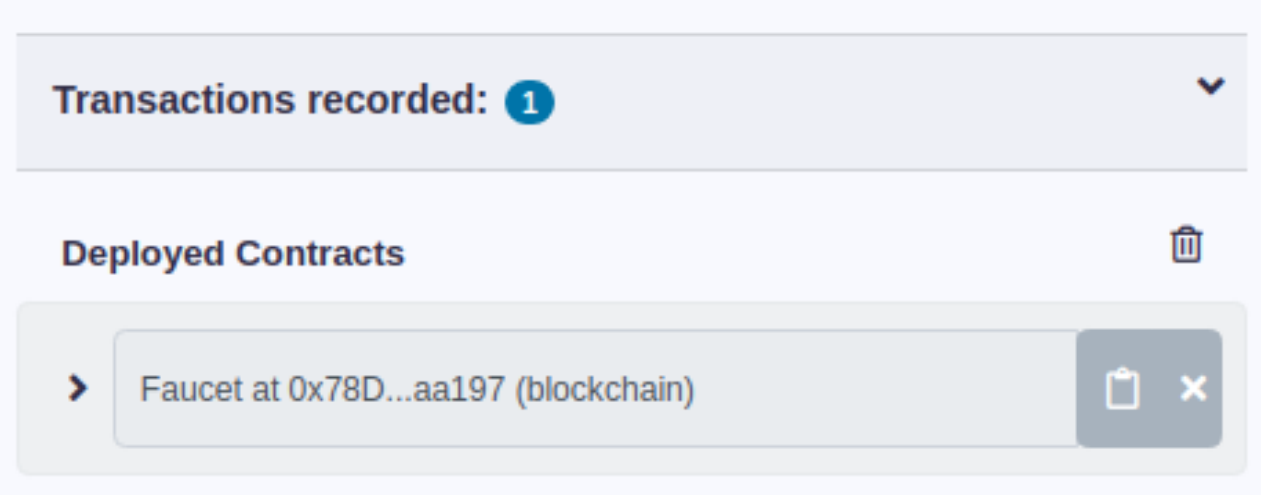
```
// Version of Solidity compiler this program was written for
pragma solidity ^0.6.0;

// Our first contract is a faucet!
contract Faucet {
    // Accept any incoming amount
    receive () external payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 10000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

# Remix: deploy an instance of the contract

Remix will create for us a transaction with the `0x0` address as destination.

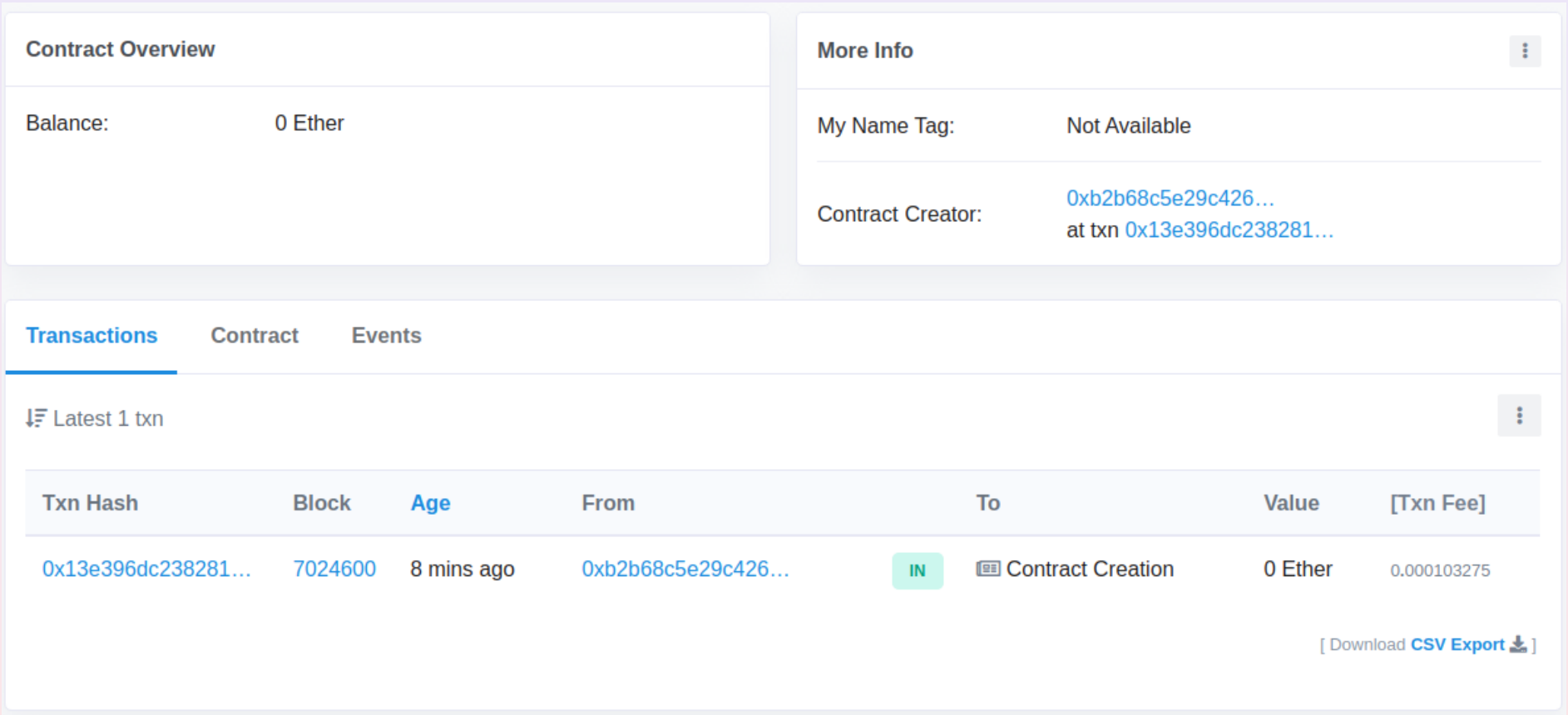


# Remix: an instance of `Faucet` has been deployed



# Check the outcome on Etherscan

`https://ropsten.etherscan.io`



**Contract Overview**

Balance: 0 Ether

**More Info**

My Name Tag: Not Available

Contract Creator: 0xb2b68c5e29c426… at txn 0x13e396dc238281…

**Transactions** | Contract | Events

Latest 1 txn

| Txn Hash | Block | Age | From | | To | Value | [Txn Fee] |
|---|---|---|---|---|---|---|---|
| 0x13e396dc238281… | 7024600 | 8 mins ago | 0xb2b68c5e29c426… | IN | Contract Creation | 0 Ether | 0.000103275 |

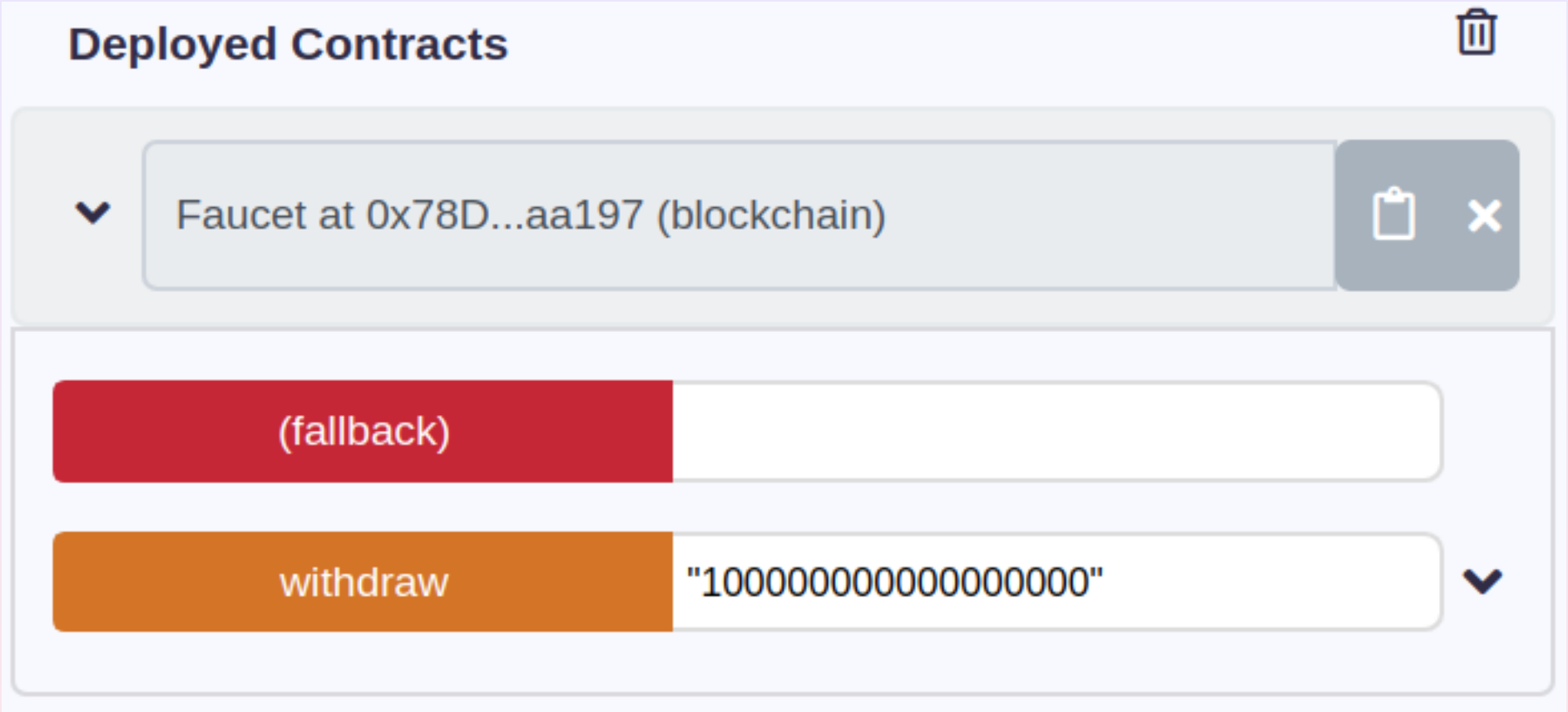
[ Download CSV Export ]

# Send one Ether to the `Faucet` through MetaMask

This corresponds to calling the default payable function.

| Txn Hash | Block | Age | From | | To | Value | [Txn Fee] |
|---|---|---|---|---|---|---|---|
| 0x424a1759911e2f4... | 7024670 | 6 mins ago | 0xb2b68c5e29c426... | IN | 0x78d632e695458b... | 1 Ether | 0.000069432 |
| 0x13e396dc238281... | 7024600 | 23 mins ago | 0xb2b68c5e29c426... | IN | Contract Creation | 0 Ether | 0.000103275 |

# Run the `withdraw` function of the `Faucet` in Remix




17 zero's

# Check again on Etherscan

The faucet worked properly!

**Contract Overview**

| | |
|---|---|
| Balance: | 0.9 Ether |

**More Info**

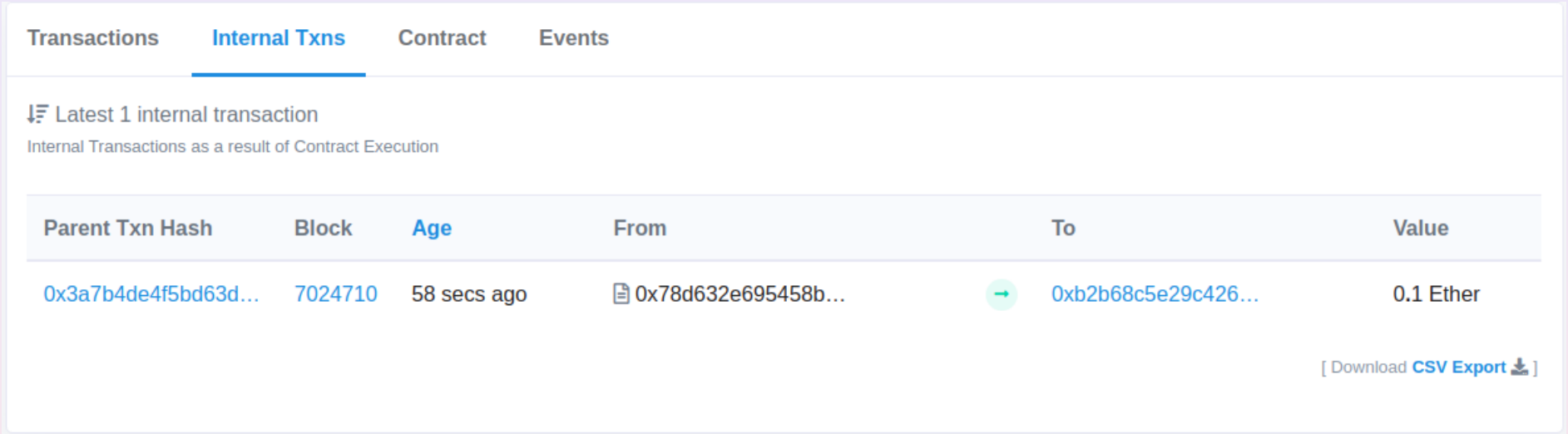
| | |
|---|---|
| My Name Tag: | Not Available |
| Contract Creator: | 0xb2b68c5e29c426…<br>at txn 0x13e396dc238281… |

Transactions | Internal Txns | Contract | Events

Latest 3 txns

| Txn Hash | Block | Age | From | | To | Value | [Txn Fee] |
|---|---|---|---|---|---|---|---|
| 0x3a7b4de4f5bd63d… | 7024710 | 58 secs ago | 0xb2b68c5e29c426… | IN | 0x78d632e695458b… | 0 Ether | 0.000028999 |
| 0x424a1759911e2f4… | 7024670 | 13 mins ago | 0xb2b68c5e29c426… | IN | 0x78d632e695458b… | 1 Ether | 0.000069432 |
| 0x13e396dc238281… | 7024600 | 30 mins ago | 0xb2b68c5e29c426… | IN | Contract Creation | 0 Ether | 0.000103275 |

# The `Faucet` has originated an internal transaction




```
msg.sender.transfer(withdraw_amount)
```

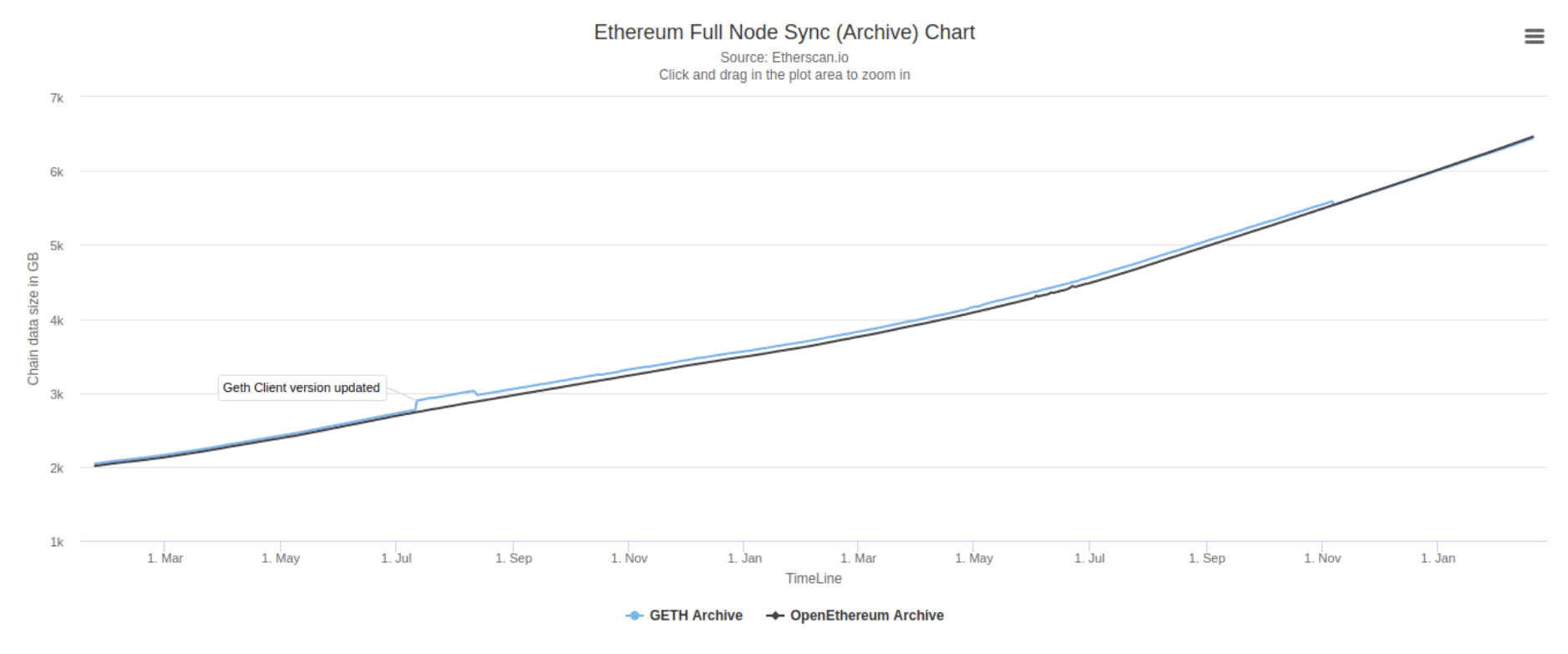
# Ethereum clients

An Ethereum client is a software application that implements the Ethereum specification (the *Yellow Paper*) and communicates over the p2p network with other Ethereum clients

- Parity (Rust)
- Geth (Go)
- `cpp-ethereum` (C++)
- `pyethereum` (Python)
- Mantis (Scala)
- Harmony (Java)

More clients means robustness against attacks

# Too big to fail?




An 8TB SSD unit costs 160€ (2021)

# Client types

## Full nodes

A full node stores the whole blockchain (∼6TB) and takes part in mining. Very slow to synchronize. It can query the blockchain offline and without letting a third party know the information that it's reading

## Remote clients

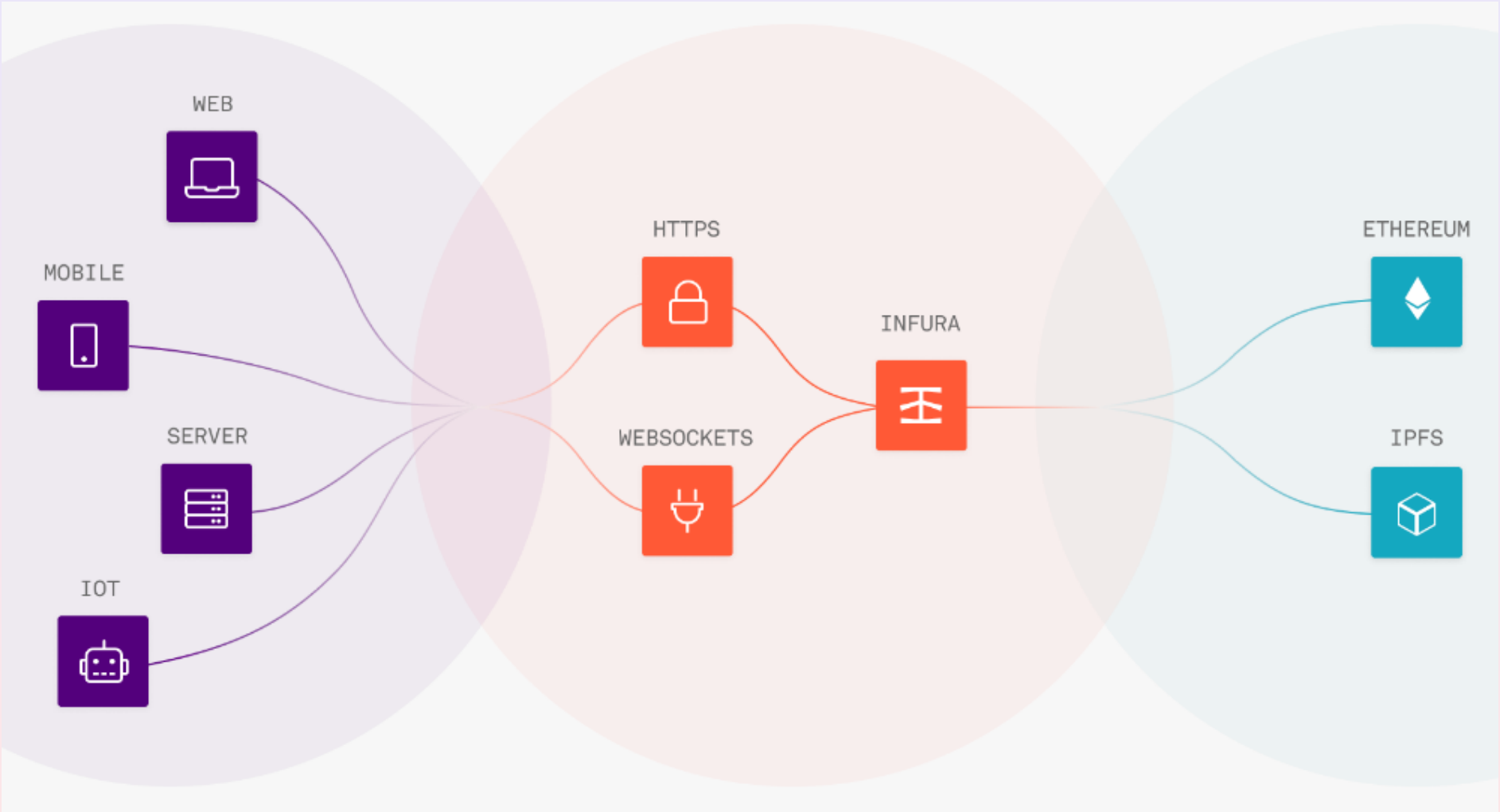
A remote client does not store the blockchain but depends on another full node for operation. They are wallets that create and broadcast transactions (for instance, MetaMask)

## Networks

The *mainnet*, or a *testnet*, or a local blockchain (Ganache)

# Infura: a cloud service to a full node

`https://infura.io`



# Infura: ask for client version

Register to Infura and create a project. Use the provided ID to build a query to Infura's JSON-RPC API:

```
$ curl https://mainnet.infura.io/v3/05550caa054f4fec80ff...
  -X POST
  -H "Content-Type: application/json"
  -d '{"jsonrpc":"2.0","method":"web3_clientVersion",
  "params": [],"id":1}'
{"jsonrpc":"2.0","id":1,
  "result":"Geth/v1.9.25-omnibus-331c5f09
            /linux-amd64/go1.15.6"}
```

# Infura: ask for gas price

Use your Infura ID to build a query to Infura's JSON-RPC API:

```
$ curl https://mainnet.infura.io/v3/05550caa054f4fec80ff...
  -X POST
  -H "Content-Type: application/json"
  -d '{"jsonrpc":"2.0","method":"eth_gasPrice",
       "params": [],"id":1}'
{"jsonrpc":"2.0","id":1,"result":"0x248dede201"}

$ echo $((0x248dede201))
157000000001
```

# Infura implements the Ethereum JSON-RPC API

ETHEREUM

JSON-RPC

eth_accounts

eth_blockNumber

eth_call

eth_chainId

eth_estimateGas

eth_gasPrice

eth_getBalance

eth_getBlockByHash

eth_getBlockByNumber

eth_getBlockTransactionCountByHash

eth_getBlockTransactionCountByNumber

eth_getCode

eth_getLogs

# A Java client that connects to Infura

## Import project `ethereum-client` as an existing Maven project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>it.univr</groupId>
  <artifactId>it.univr.ethereumclient</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Ethereum Client</name>
  <build>
    <plugins><plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration><release>11</release></configuration>
    </plugin></plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.web3j</groupId>
      <artifactId>core</artifactId>
      <version>5.0.0</version>
    </dependency>
  </dependencies>
</project>
```

# A Java client that connects to Infura

```
import java.io.IOException;

import org.web3j.protocol.Web3j;
import org.web3j.protocol.core.methods.response.Web3ClientVersion;
import org.web3j.protocol.http.HttpService;

public class ClientVersion {
    public static void main(String[] args) throws IOException {
        Web3j web3 = Web3j.build
            (new HttpService("https://ropsten.infura.io/v3/05..."));
        Web3ClientVersion web3ClientVersion = web3.web3ClientVersion().send();
        System.out.println(web3ClientVersion.getWeb3ClientVersion());
    }
}
```

https://javadoc.io/doc/org.web3j

# Ethereum's cryptographic hash function

## Keccak-256

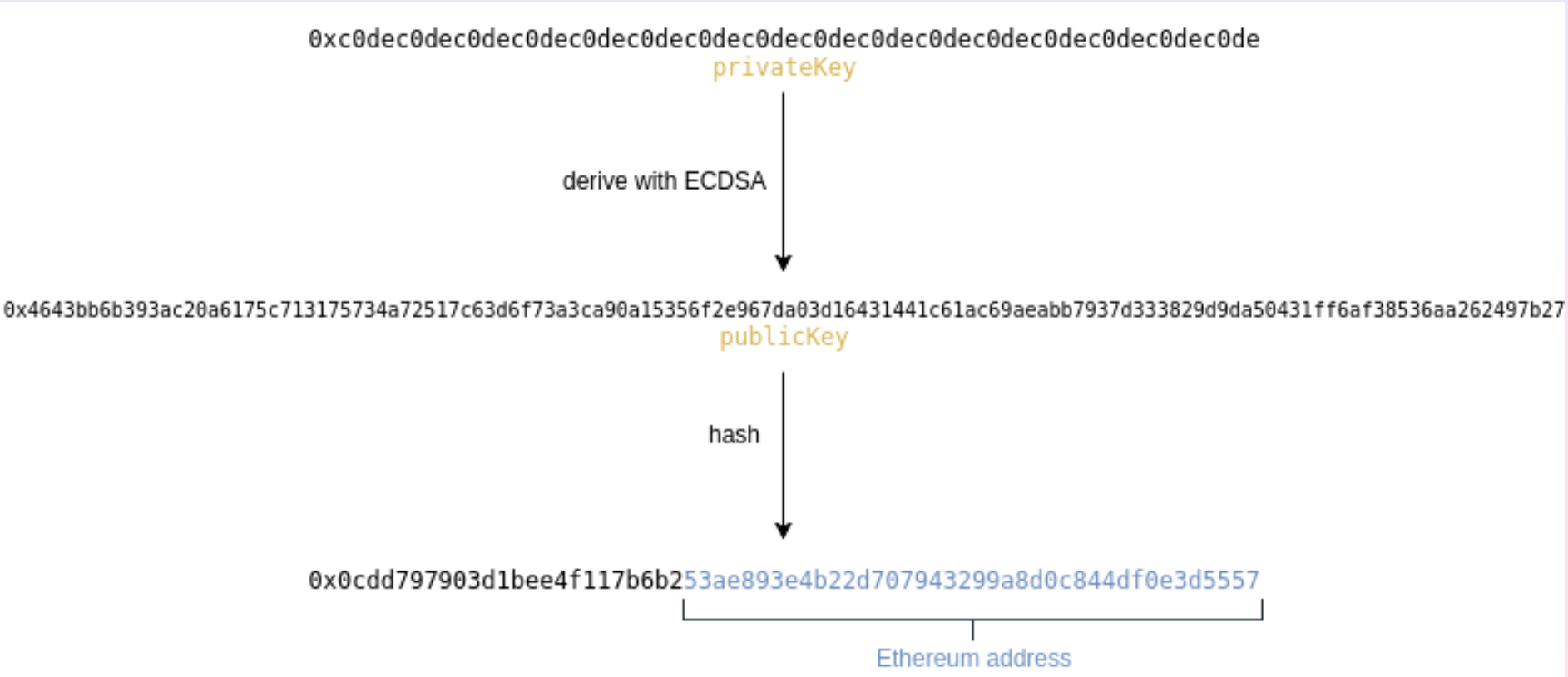
Ethereum uses the Keccak-256 cryptographic hash function. It is the original algorithm that won the SHA-3 Cryptographic Hash Function Competition of 2007. NIST standardized it in a slightly modified way as SHA-3. However, the Ethereum team never trusted such modification and used the original algorithm

## Keccak-256 or SHA-3?

You can still see blogs and even the source code of Ethereum refer to SHA-3. Despite of that, Etherem does not use SHA-3, but the original Keccak-256 algorithm!

# Derivation of an Ethereum address from the private key

Compute the public key, hash it with Keccak-256, keep only the last 20 bytes (*ie.*, 160 bits, that is, 40 hex digits). No checksum!




Bitcoin: private $\xrightarrow{ECDSA}$ public $\xrightarrow{sha256}$ hash $\xrightarrow{ripemd160}$ address

Ethereum: private $\xrightarrow{ECDSA}$ public $\xrightarrow{kekkak256}$ hash $\xrightarrow{last\ \ 20\ \ bytes}$

# Checksummed address format: ICAP

`XE` + 2 characters of checksum + base-36 integer of up to 30 digits

- the base-36 integer can represent up to 155 bits, hence ICAP can only be used for addresses starting with a zero byte
- ICAP is compatible with IBAN

**Example**

Ethereum address: `0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9`

ICAP: `XE60 HAMI CDXS V5QX VJA7 TJW4 7Q9C HWKJ D`

# Hex encoding with checksum in capitalization (EIP-55)

## A backward compatible checksum injection

1. use Keccak-256 to compute 64 hex digits from the uncapitalized 40 hex digits of the Ethereum address
2. capitalize each alphabetic address character if the corresponding hex digit of the hash is greater than or equal to `0x8`

## Example

```
Address:    001d3f1ef827552ae1114027bd3ecf1f086ba0f9
Keccak256:  23a69c1653e4ebbb619b0b2cb8a9bad49892a8b9...
Result:     001d3F1ef827552Ae1114027BD3ECF1f086bA0F9
```

## Checking procedure of an EIP-55 encoded address

- apply the above capitalization procedure to it
- the result must match the EIP-55 encoded address

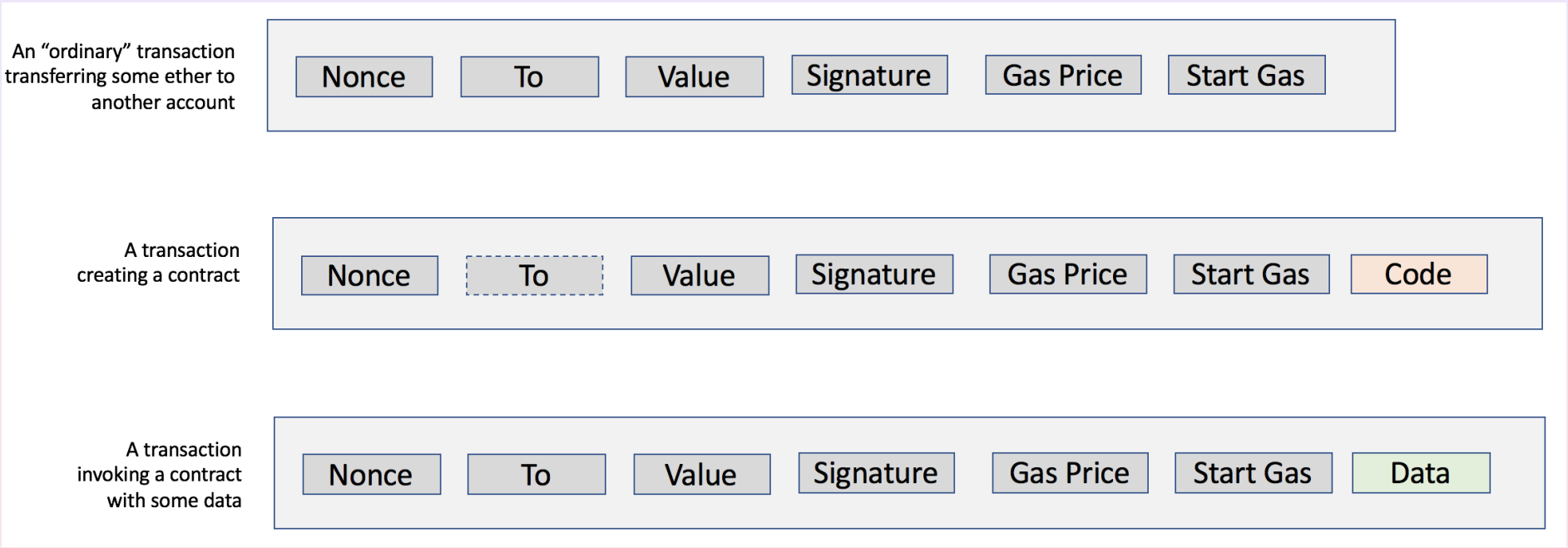
# Ethereum transactions

A transaction is a signed message originated by an EOA, transmitted by the Ethereum network, and recorded on the Ethereum blockchain:

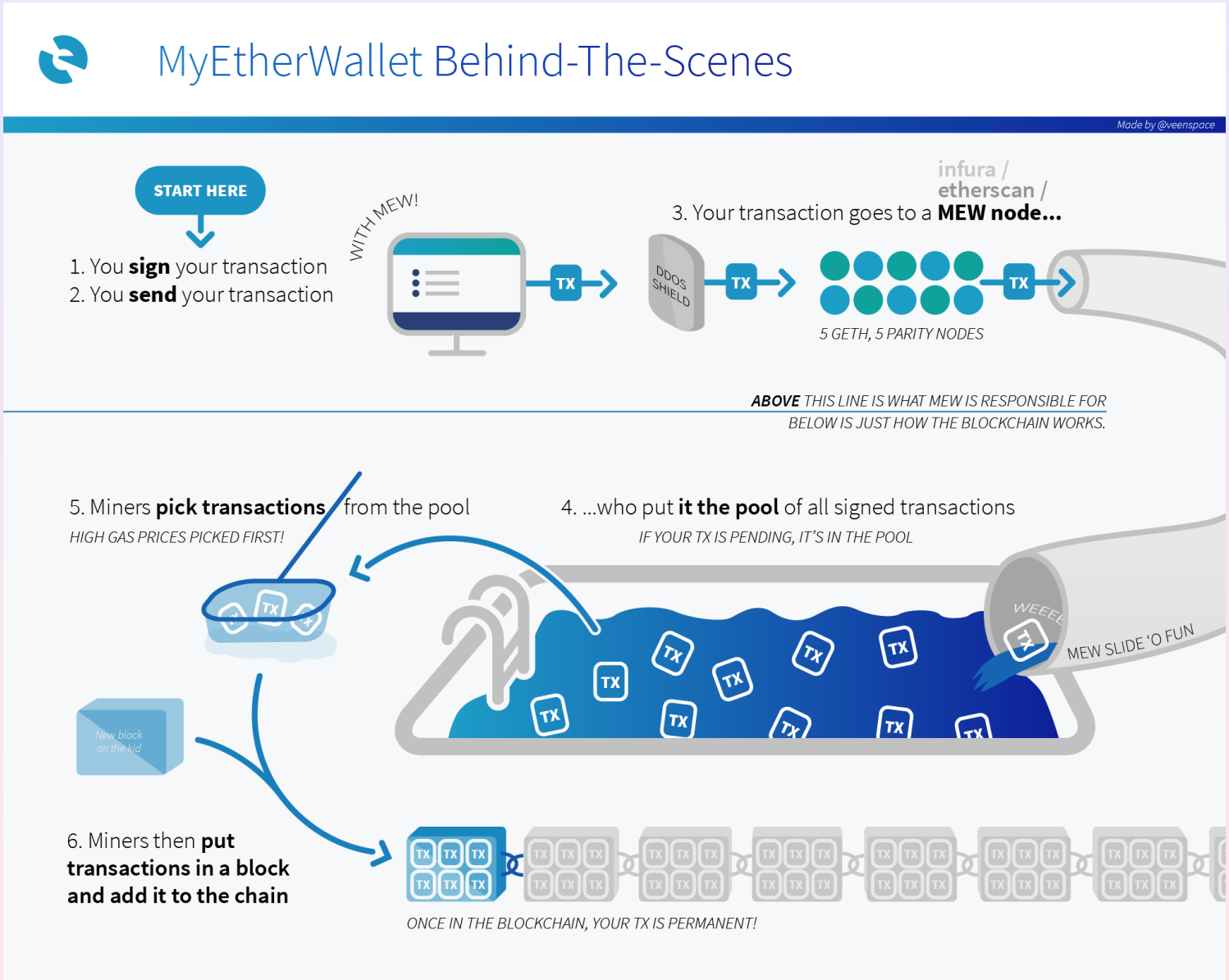
- nonce: sequence number per each originating EOA
- gas price: maximum willing to pay
- gas limit: maximum willing to consume
- to: recipient (destination address)
- value: ether sent to destination
- data: generic payload (method name, parameters, contract code...)
- v,r,s: ECDSA signature of the originating EOA

The address of the originating EOA is implied by v,r,s
The id of the transaction is the hash of its byte encoding

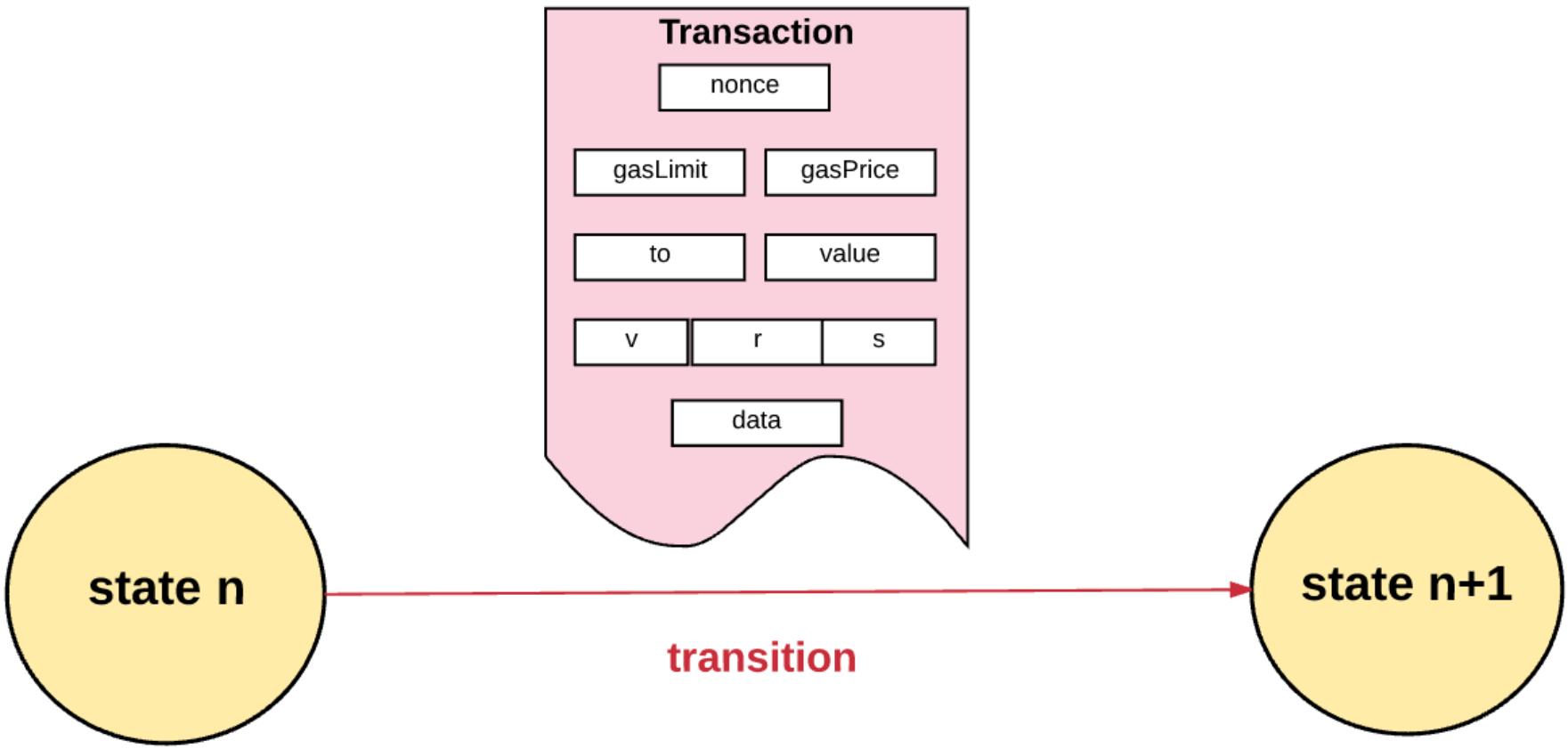
# Many kinds of transactions



# Transaction lifecycle



# Transactions induce state transitions



# The nonce

## The nonce of an EOA

A scalar value equal to the number of transactions sent from the EOA

## Wallets keep track of nonces

They increase it and attach to each transaction they create per originating EOA

## Nodes check nonces

They count the number $n$ of transactions originated by the EOA. If the nonce is smaller than $n + 1$, the transaction is rejected. If the nonce is greater than $n + 1$, the transaction is delayed and not yet executed:

- this guarantees transaction ordering
- and avoids transaction replaying

Keeping track of nonces is hard for concurrent clients!

# Why Bitcoin's transactions have no nonce?

## Bitcoin's transactions can only transform UTXOs into TXOs

It is not possible to spend a UTXO again, inside the same history: it would be against the consensus rules

$\Rightarrow$ Executing a valid transaction today makes it invalid tomorrow

## Ethereum's transactions can induce any state change or fail

A valid (syntactically correct) transaction can always be executed in Ethereum

$\Rightarrow$ Executing a valid transaction today doesn't make it invalid tomorrow (without a nonce)

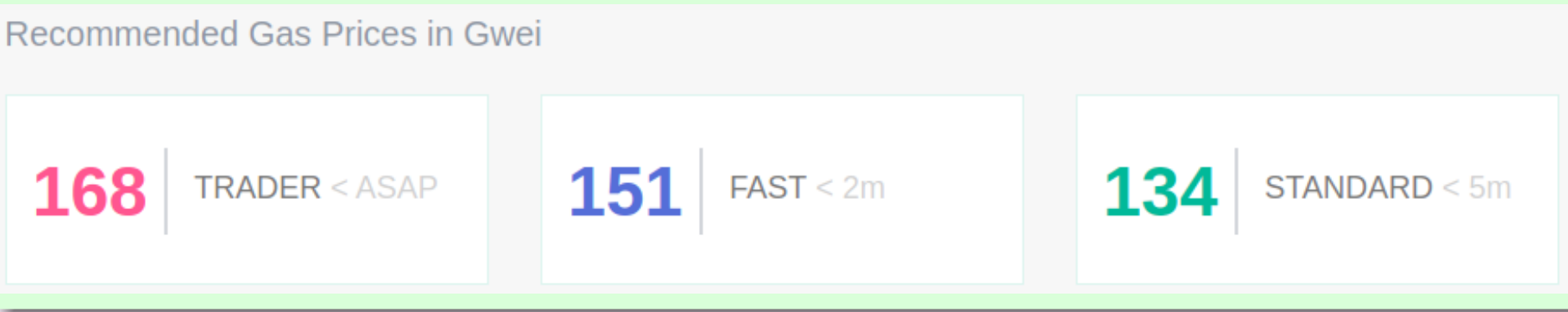
# Computing the next nonce in Java using Infura

```java
import java.math.BigInteger;
import java.util.concurrent.ExecutionException;

import org.web3j.protocol.Web3j;
import org.web3j.protocol.core.DefaultBlockParameterName;
import org.web3j.protocol.core.methods.response.EthGetTransactionCount;
import org.web3j.protocol.http.HttpService;

public class GetNonce {
    // the address that we created with MetaMask
    private final static String ME = "0xb2B68C5E29C4267A24BB92daC5579D93f9A74812";
    private final Web3j web3 = Web3j.build
        (new HttpService("https://ropsten.infura.io/v3/05..."));

    public BigInteger transactionCount() throws InterruptedException, ExecutionException {
        EthGetTransactionCount ethGetTransactionCount = web3.ethGetTransactionCount
            (ME, DefaultBlockParameterName.LATEST).sendAsync().get();
        return ethGetTransactionCount.getTransactionCount();
    }

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        var gn = new GetNonce();
        System.out.println("Next nonce is: " + gn.transactionCount());
    }
}
```

# Gas

The price of gas fluctuates, for protection against ETH spikes:

**https://ethgasstation.info/**




It is theoretically possible that transactions with 0 gas price will be eventually processed

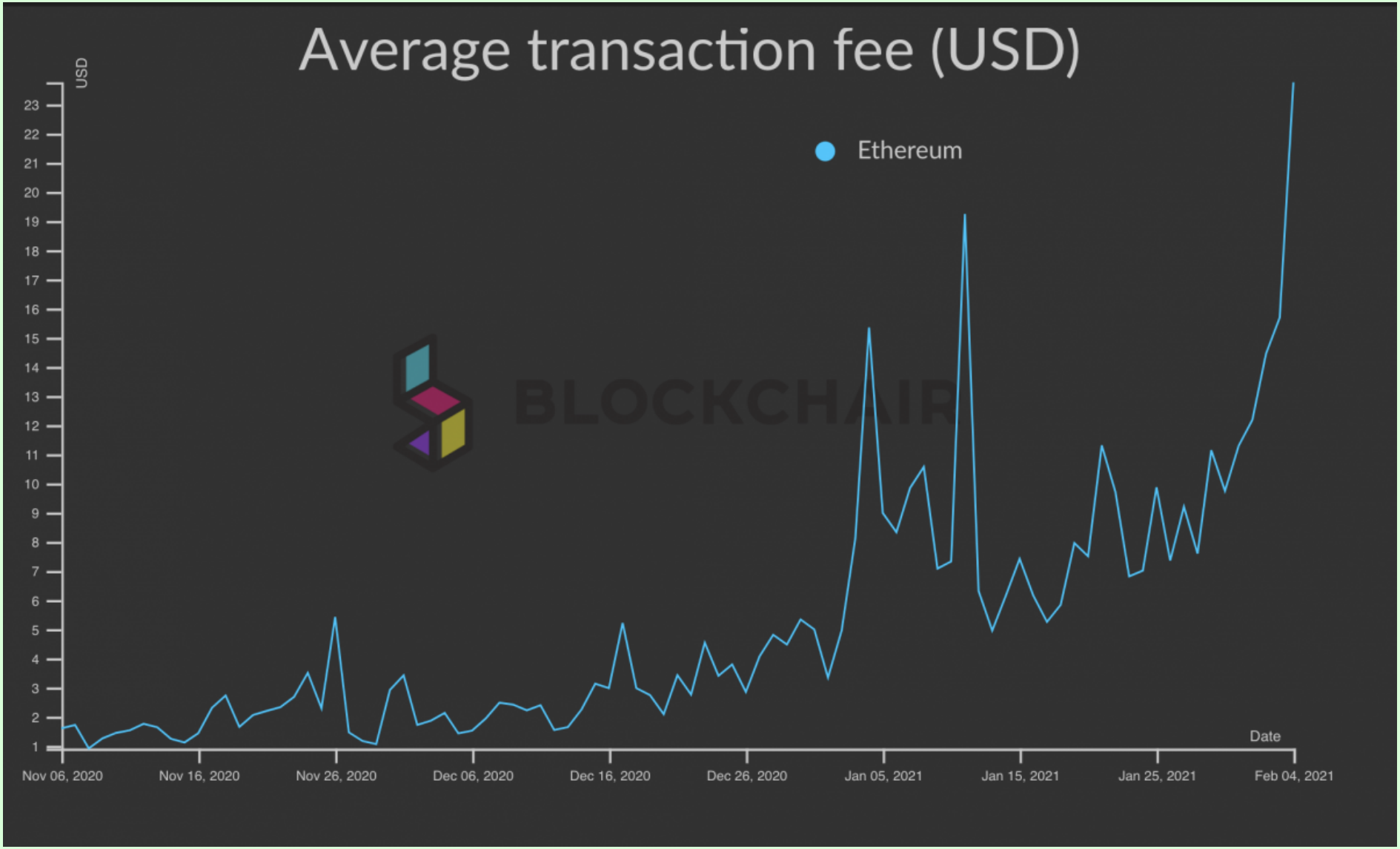
Maximal fees: *Gas price* $\times$ *Gas limit* $\times$ *Euros per Gwei*

**Example: a small transaction requires 21000 units of gas**

$$fees = 151 \times 21000 \times 0.00000162€ = 5.137€$$

# Average fees chart

https://blockchair.com/



# Computing the gas price in Java using Infura

```
import java.io.IOException;

import org.web3j.protocol.Web3j;
import org.web3j.protocol.core.methods.response.EthGasPrice;
import org.web3j.protocol.http.HttpService;

public class GetGasPrice {
    public static void main(String[] args) throws IOException {
        Web3j web3 = Web3j.build(new HttpService
            ("https://ropsten.infura.io/v3/05..."));
        EthGasPrice gasPrice =  web3.ethGasPrice().send();
        System.out.println(gasPrice.getGasPrice());
    }
}
```

# To

Any 20 bytes recipient address can be specified (EOA or contract), there is no control!

**Burning ETH**

Sending ether to a nonexistent address means *burning* that ether, since it's practically impossible to identify the private key that controls that address. It is standard practice to burn ether by sending it to the address

`0x000000000000000000000000000000000000dEaD`

Clients should check for correctness of addresses, for instance through EIP-55

# Signing and sending a transaction in Java

```
public abstract class SendTransaction {
    protected final static String ME = "0xb2B68C5E29C4267A24BB92daC5579D93f9A74812";
    private final static String ME_PRIVATE_KEY = "E2...";
    private final Credentials credentials = Credentials.create(ME_PRIVATE_KEY);
    protected final Web3j web3 =
            Web3j.build(new HttpService("https://ropsten.infura.io/v3/05..."));

    public final BigInteger transactionCount()
            throws InterruptedException, ExecutionException {
        EthGetTransactionCount ethGetTransactionCount = web3.ethGetTransactionCount
            (ME, DefaultBlockParameterName.LATEST).sendAsync().get();
        return ethGetTransactionCount.getTransactionCount();
    }

    public final String send(RawTransaction transaction)
            throws InterruptedException, ExecutionException {
        byte[] signedMessage = TransactionEncoder.signMessage(transaction, credentials);
        String hexValue = Numeric.toHexString(signedMessage);
        EthSendTransaction tr = web3.ethSendRawTransaction(hexValue).sendAsync().get();
        System.out.println("errors: " + tr.hasError());
        return tr.getTransactionHash();
    }
}
```

# Sending ETH from our account to the faucet

```
import java.math.BigInteger;
import java.util.concurrent.ExecutionException;

import org.web3j.crypto.RawTransaction;

public class SendToFaucet extends SendTransaction {
    // the Ropsten faucet
    private final static String FAUCET = "0x81b7E08F65Bdf5648606c89998A9CC8164397647";

    public String sendToFaucet() throws InterruptedException, ExecutionException {
        RawTransaction transaction = RawTransaction.createEtherTransaction
            (transactionCount(),              // nonce
            BigInteger.valueOf(100_000L),     // gas price,
            BigInteger.valueOf(21_000L),      // gas limit,
            FAUCET,                           // to
            BigInteger.valueOf(5_000_000L)); // value

        return send(transaction);
    }

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        var stf = new SendToFaucet();
        System.out.println("transaction hash: " + stf.sendToFaucet());
    }
}
```

# Transactions with value (*payments*)

- the recipient is an EOA (data will be kept in blockchain, but not used):
  - the EOA existed already: its balance gets increased by value
  - the EOA didn't exist before: its balance is set to value
- the recipient is a contract:
  - there is data:
    - there is a payable method specified by the data: it will be executed, after increasing the contract's balance by value
    - there is no such method or it is not payable: the transaction fails
  - there is no data:
    - there is a payable *fallback* function: it will be executed after increasing the contract's balance by value
    - there is no fallback function: the balance of the contract gets increased by value
    - there is a fallback function, but it is not payable: the transaction fails

# Transactions targetting contract methods

- the signing key specifies the caller
- the recipient is the address of the receiving contract of the call
- the data component encodes signature and actual arguments of the call

## How data is computed

1. take the callee's signature as a string: `withdraw(uint256)`
2. compute its Keccak256 hash
3. take the first 4 bytes: this is the *function selector*
4. append the actual arguments, serialized

# Calling `withdraw()` on a contract installed with Remix

```
public class CallWithdraw extends SendTransaction {
    // a Faucet.sol already deployed (for instance, with Remix)
    private final static String FAUCET = "0x78D632E695458B87ec7747234C2c9B306d3aa197";

    public String withdrawFromFaucetContract() throws InterruptedException, ExecutionException {
        Function sig = new Function("withdraw", // function name
            Arrays.asList(new Uint256(Convert.toWei("1", Unit.SZABO).toBigInteger())), // actuals
            Collections.emptyList()); // return types

        String data = FunctionEncoder.encode(sig);

        RawTransaction transaction = RawTransaction.createTransaction
            (transactionCount(),                                  // nonce
            Convert.toWei("10", Unit.GWEI).toBigInteger(), // gas price,
            BigInteger.valueOf(50_000L),                          // gas limit,
            FAUCET,                                               // to
            data);                                                // value

        return send(transaction);
    }

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        var cw = new CallWithdraw();
        System.out.println("transaction hash: " + cw.withdrawFromFaucetContract());
    }
}
```

# Transactions creating a new instance of a contract

- the signing key specifies the caller
- the recipient is set to the zero address
- the data component contains the compiled bytecode of the contract (ask Remix for that)
- the transaction receipt contains the address of the new contract

# Creating an instance of `Faucet.sol`

```
public class SendFaucetContract extends SendTransaction {
    // the bytecode can be obtained from Remix or by running
    // solc --bin Faucet.sol
    private final static String BYTECODE = "608060405234801561001057600080fd5b5060e78061001f...

    public String sendContract() throws InterruptedException, ExecutionException {
        RawTransaction transaction = RawTransaction.createContractTransaction
            (transactionCount(),                                  // nonce
            Convert.toWei("1", Unit.GWEI).toBigInteger(), // gas price,
            BigInteger.valueOf(200_000L),                         // gas limit,
            BigInteger.ZERO,                                      // value
            BYTECODE);                                            // data

        return send(transaction);
    }

    public static void main(String[] args) throws InterruptedException, ExecutionException, IOException {
        var sfc = new SendFaucetContract();
        String hash = sfc.sendContract();
        System.out.println("transaction hash: " + hash);
        TransactionReceipt receipt = sfc.waitForReceipt(hash);
        System.out.println("contract address: " + receipt.getContractAddress());
    }
}
```

# Waiting for the receipt

```
public abstract class SendTransaction {
    protected final static String ME = "0xb2B68C5E29C4267A24BB92daC5579D93f9A74812";
    private final static String ME_PRIVATE_KEY = "E2...";
    private final Credentials credentials = Credentials.create(ME_PRIVATE_KEY);
    protected final Web3j web3 =
            Web3j.build(new HttpService("https://ropsten.infura.io/v3/05..."));

    public final BigInteger transactionCount()

    public final String send(RawTransaction transaction)

    public final TransactionReceipt waitForReceipt(String transactionHash)
            throws IOException, InterruptedException {
        TransactionReceipt result;
        do {
            Thread.sleep(10_000);
            result = web3
                .ethGetTransactionReceipt(transactionHash)
                .send()
                .getResult();
        }
        while (result == null);

        return result;
    }
}
```

# Transaction signing (ECDSA)

Transactions are digitally signed before being broadcast to Ethereum:

```
Credentials credentials = Credentials.create(ME_PRIVATE_KEY);
byte[] signedMessage = TransactionEncoder.signMessage
  (transaction, credentials);
```

The signature serves three goals:

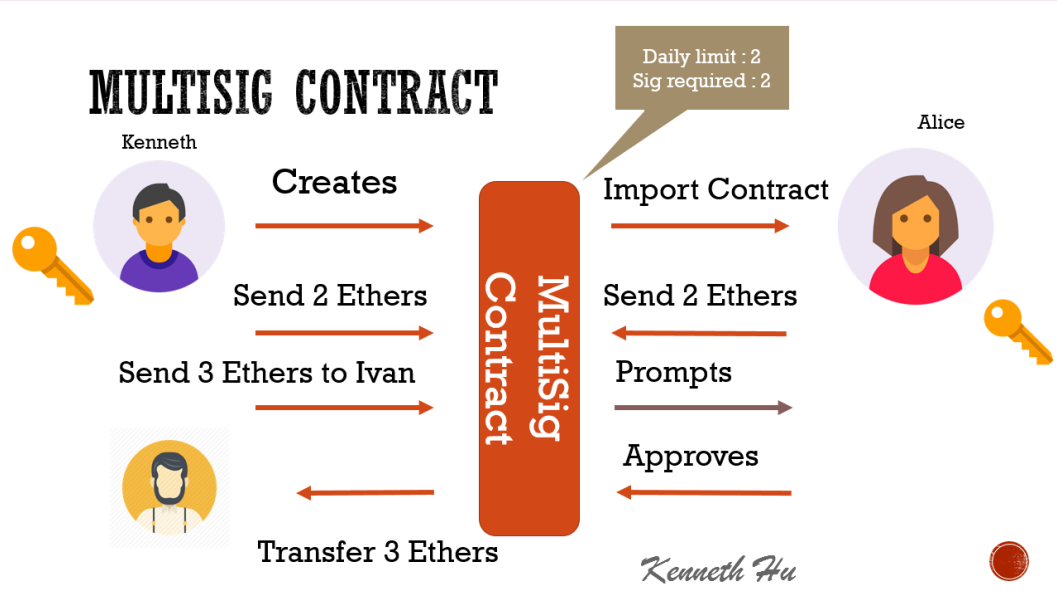
1. it guarantees that the transaction was created by the sender (authenticity)
2. it guarantees that the sender cannot deny having sent the transaction (non-repudiation)
3. it guarantees that the transaction was not altered in transit (integrity)

The public key of the signer can be recovered from the signed transaction. A numerical chain identifier is added to the transaction before signing, so that transactions cannot be replayed across networks (mainnet, testnets. . . )

# Multi-signature transactions

Differently from Bitcoin, Ethereum has no native support for multi-signature transactions (that is, transactions that are only valid if signed by more EOAs). However, multi-signature and other signature schemes can be programmed through smart contracts:

- extreme flexibility
- risk of bugs



# The state of Ethereum

The state of Ethereum is a global singleton map $\sigma : key \rightarrow value$

**The state is not in blockchain!**

Each node keeps and maintains its own copy in a private database

**API of the state**

1. `value=get(address)`
2. `put(address, value)`

# Encoding data into the state

## Store the balance of an EOA

```
put(address_of_EOA, balance)
```

## Read the balance of an EOA

```
get(address_of_EOA)
```

# Encoding data into the state

## EOA installs a smart contract

```
put(hash(address_of_EOA, nonce_of_EOA), bytecode_of_smart_contract)
```

*address of the new smart contract*

## A smart contract writes $v$ into its $n$th instance variable (field)

```
put(hash(address_of_smart_contract, n), v)
```

*address of the n−th field*

## A smart contract reads from its $n$th instance variable (field)

```
get(hash(address_of_smart_contract, n))
```

*address of the n−th field*

# The hash of the state

**In Bitcoin, the header of a block contains the hash of the transactions in the block**

That is, the head of the Merkle tree of transactions. Miners must execute the transactions to validate them, since invalid transactions would make the whole block invalid, which would make the miner lose money

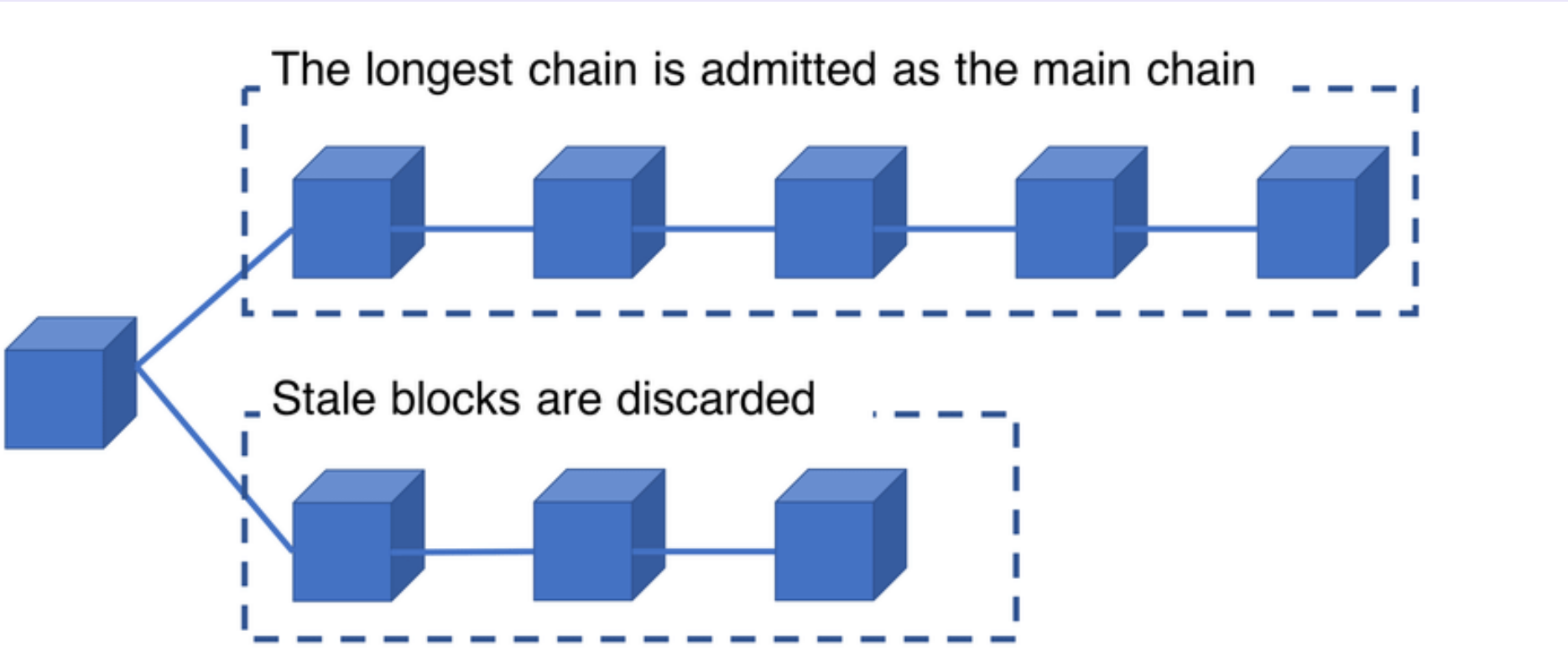
**In Ethereum, the header of a block contains the hash of the state at the end of the execution of the transactions in the block**

Since syntactically correct transactions are always valid, this obliges the miners to execute the transactions

**API of the state**

1. `value=get(address)`
2. `put(address, value)`
3. `h=get_hash()`

# The state of a node must change across different views




Bitcoin: (kind of) nothing to do

Ethereum: hard: method executions undo, others must be done

# Change view to the state at the end of a block

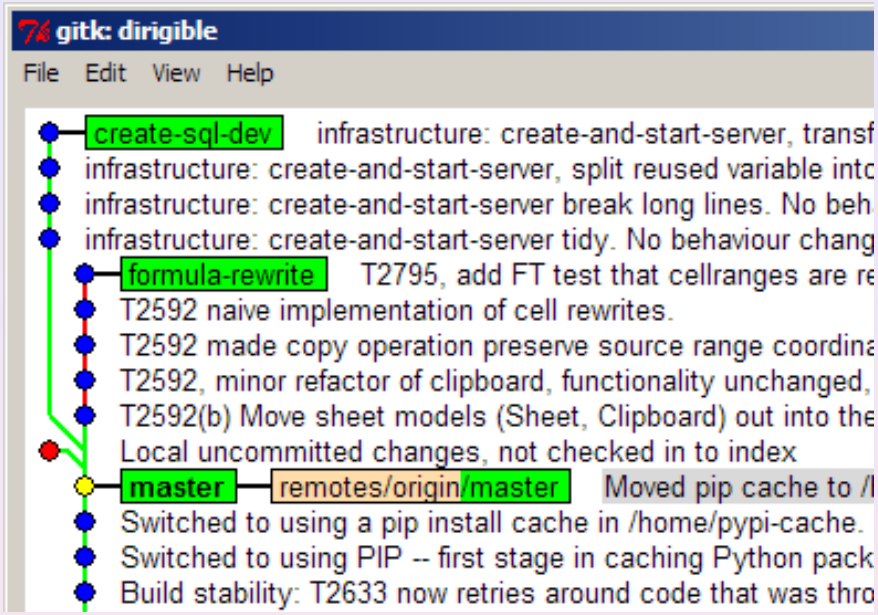
## Undo of state updates, up to the state at the end of an old block

```
checkout(old_block.header.state_hash)
```

## The final API of the state

1. `value=get(address)`
2. `put(address, value)`
3. `h=get_hash()`
4. `checkout(h)`

# Ethereum's state is on Git




```
git checkout create-sql-dev

git checkout master

git checkout formula-rewrite
```

# Merkle-Patricia tries

1. a compact representation by difference from a previous state
2. fast recomputation of its hash once the state is updated
3. fast access to the value bound to each given key
4. fast change of view

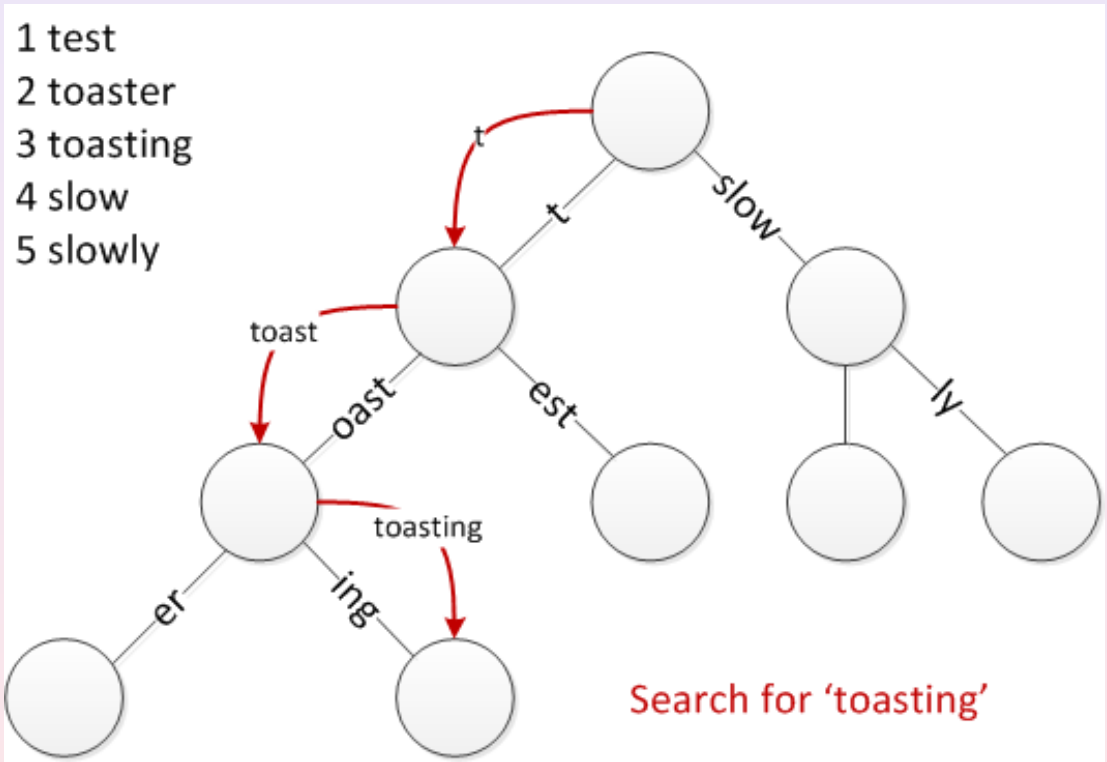
Traditional data structures such as hashmaps only satisfy 3 (and 2, but only for trivial hashing functions, which is not acceptable for blockchains)

## The final API of the state for keys of length $L$

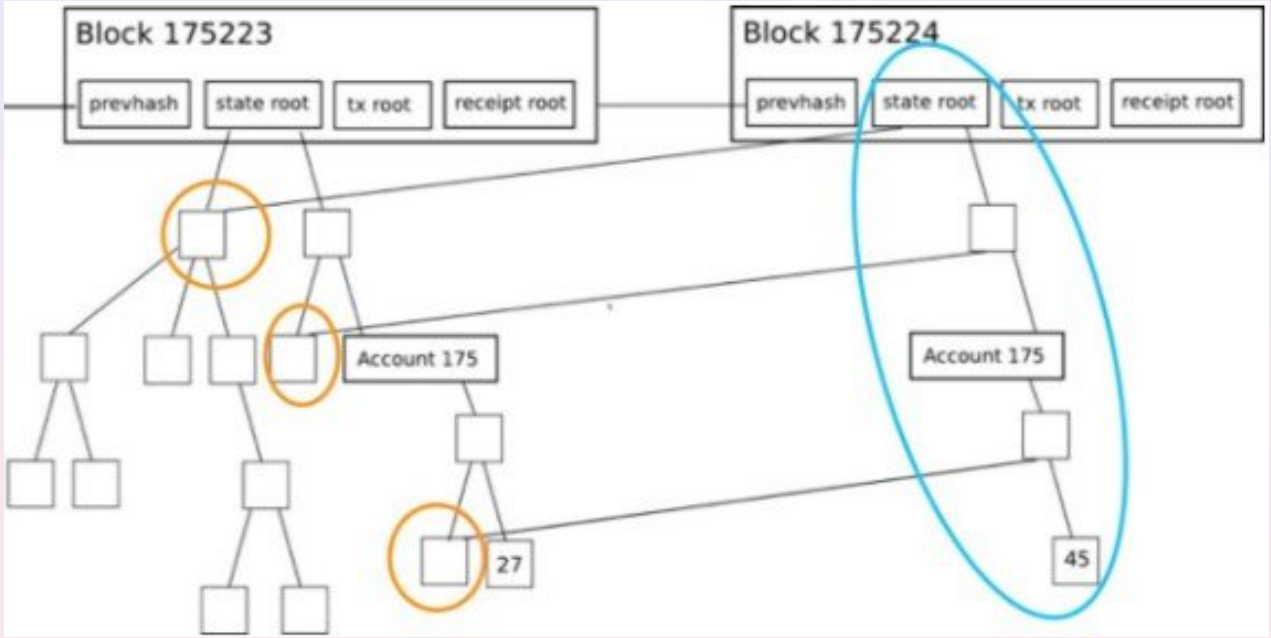
| | |
|---|---|
| 1. `value=get(address)` | $O(L)$ |
| 2. `put(address, value)` | $O(L + size(\texttt{value}))$ |
| 3. `h=get_hash()` | $O(1)$ |
| 4. `checkout(h)` | $O(1)$ |

# Patricia trie: fast access to the value bound to each key

**P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric

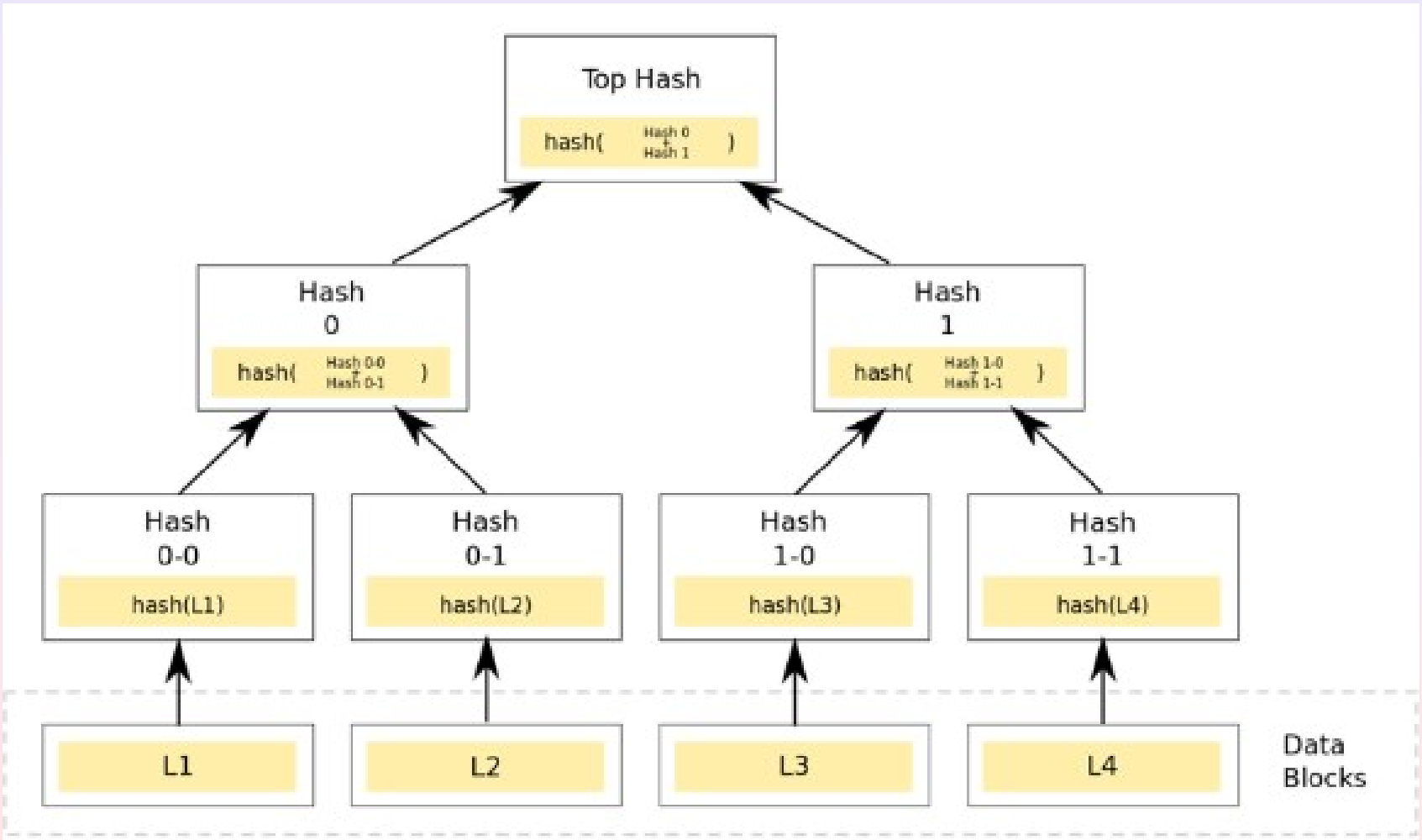


# Patricia trie: compact representation by difference

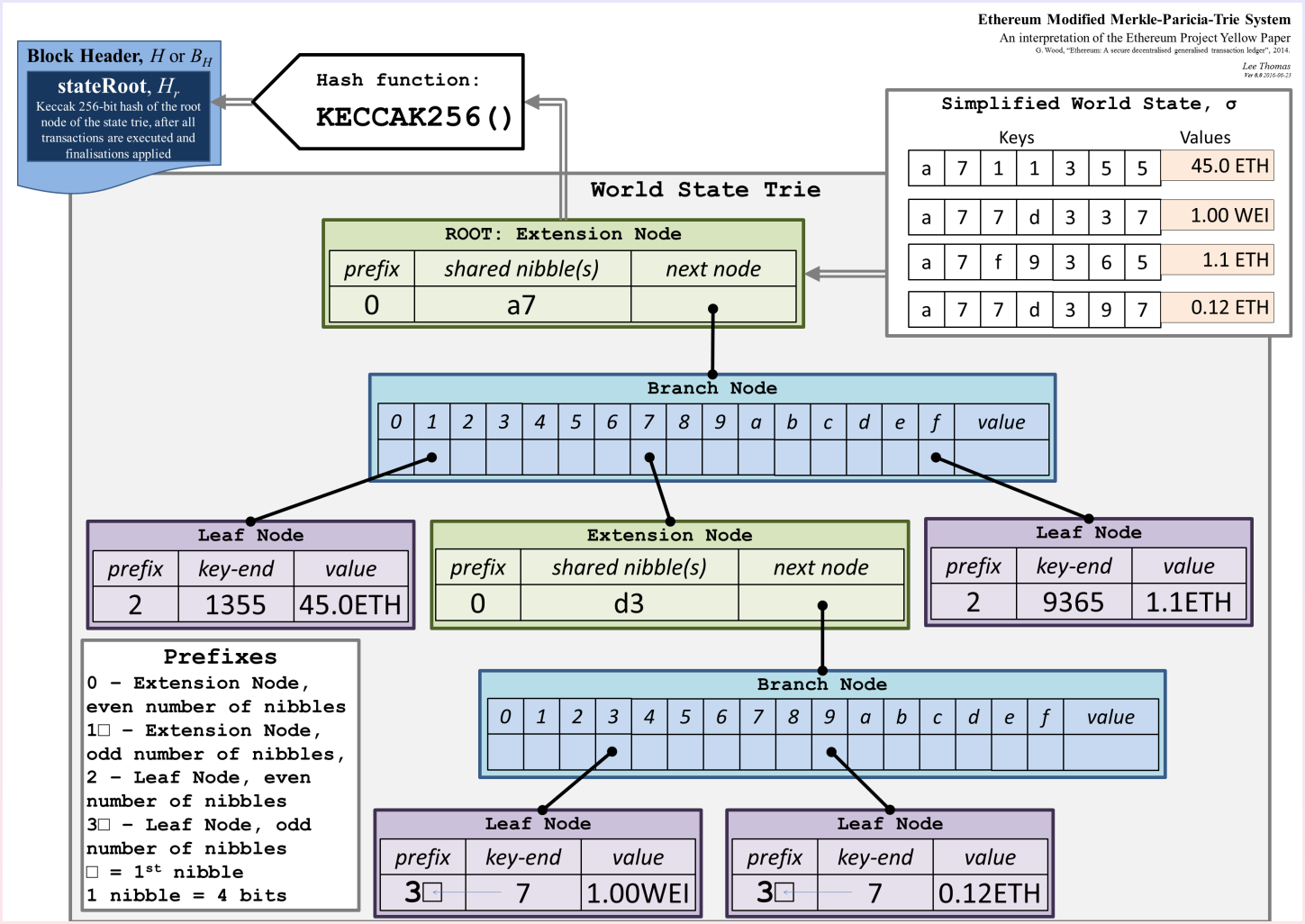


# Merkle trie: fast recomputation of its hash upon update

Proceed bottom-up:



# Merkle-Patricia tries




Links are not pointers, but the hash of the node they refer to

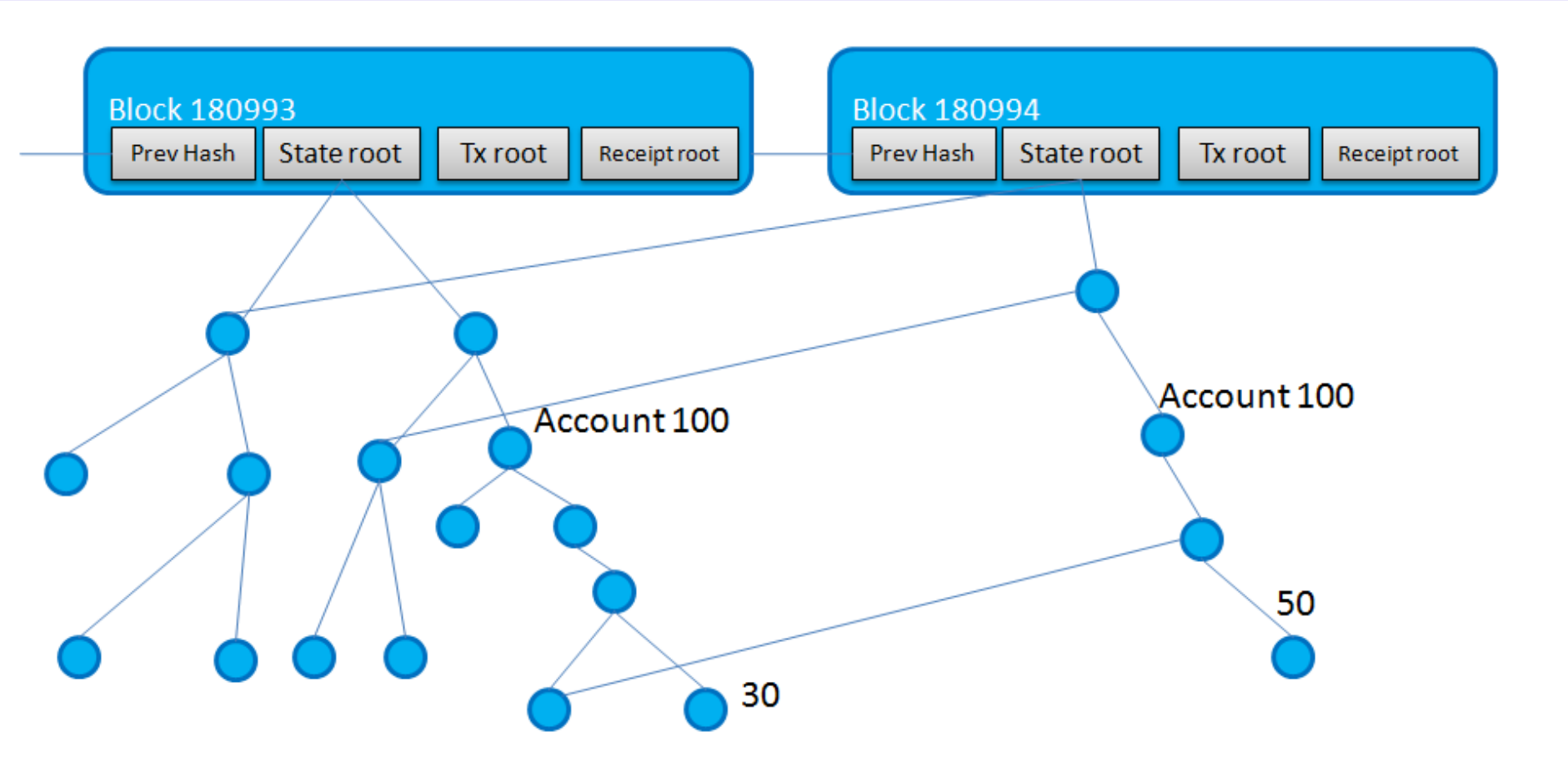
# A smart contract is. . .

- a computer program (not *smart* nor *a contract*)
- immutable
- deterministic
- operating on restricted data
- running on a decentralized world computer

In Ethereum:

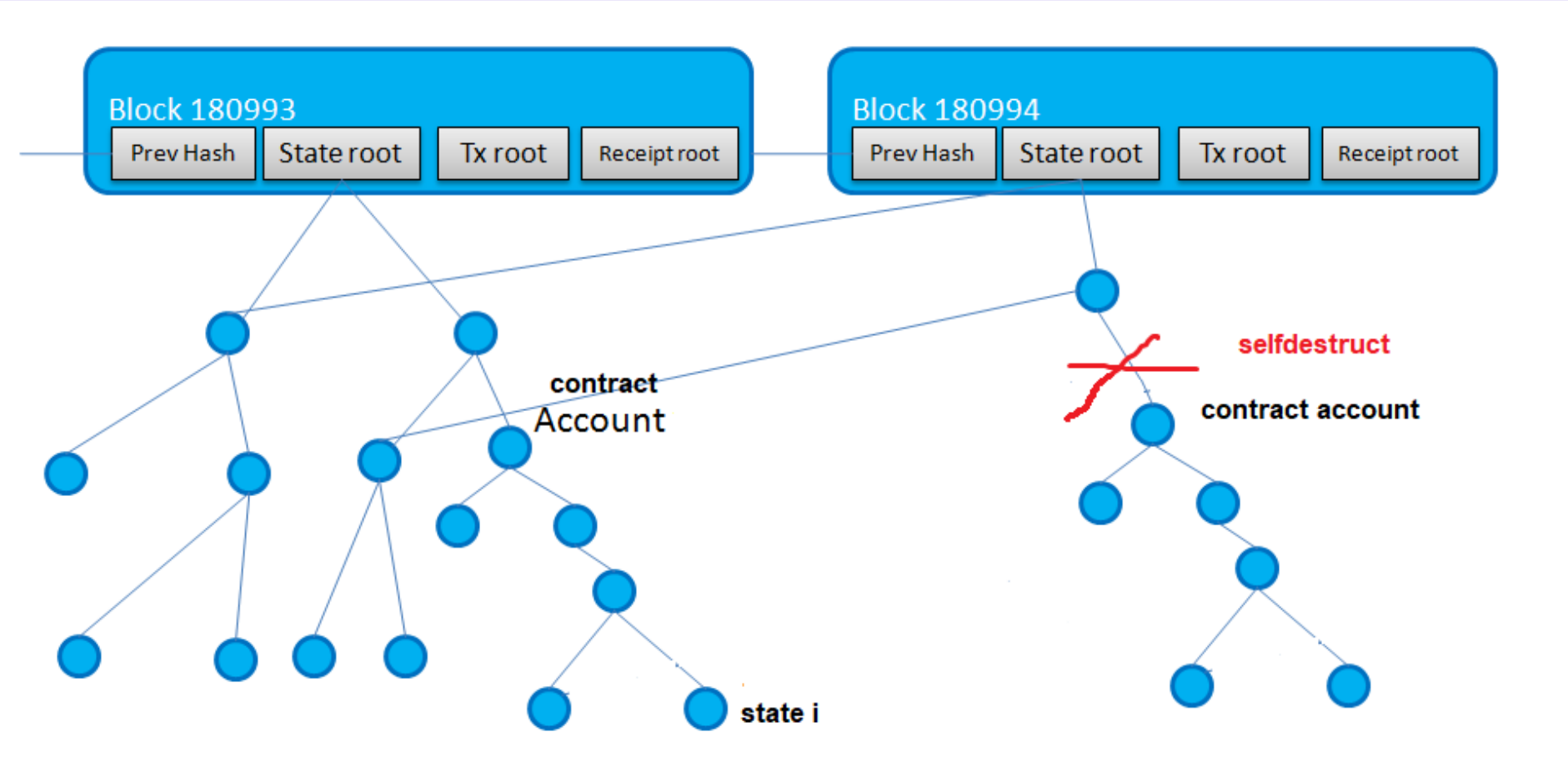
- compiled into EVM bytecode
- installed in blockchain by sending a special transaction to address `0x0`
- its code can be deleted if it executes a `SELFDESTRUCT` bytecode
- has no keys
- its installer gets no automatic privileges
- runs after a transaction initiated by an EOA
  - or a chain of transactions initiated by an EOA
  - no parallelism, no background processing
- transactions are atomic

# Blocks keep transactions not state




The state root is just a hash of the state tree, to prove that transactions have been executed by the miner

# SELFDESTRUCT has effect on the state



# Solidity

There are many programming languages for Ethereum smart contracts, but Solidity is the de facto standard:

- imperative
- vaguely object-oriented
- in continuous evolution
- non-strongly-typed
- unorthogonal features

- ✓ sequence
- ✓ conditional
- ✓ repetition

$\Rightarrow$ Turing complete (bug or feature?)

# Back to our first Solidity example

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.6.0;

// Our first contract is a faucet!
contract Faucet {
    // Accept any incoming amount
    receive () external payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 10000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

# ABI – from Remix or from `solc`

```
[
  {
    "payable": true,
    "stateMutability": "payable",
    "type": "fallback"
  },
  {
    "constant": false,
    "inputs": [
      {
        "internalType": "uint256",
        "name": "withdraw_amount",
        "type": "uint256"
      }
    ],
    "name": "withdraw",
    "outputs": [],
    "payable": false,
    "stateMutability": "nonpayable",
    "type": "function"
  }
]
```

# Select a compiler version

## Add a pragma directive

`pragma solidity ^0.4.19` requires to compile with a compiler for version $0.4.x$ with $x \geq 19$

# Basic Solidity types

**`bool`**

with constants `true` and `false` and usual operators

**`int`, `unint`**

signed or unsigned, with usual operators, in increments of 8 bit size: `uint8`, `uint16`, `int24`... Without specification, they stand for `int256` and `unint256`, respectively

**`fixed`$M \times N$, `ufixed`$M \times N$**

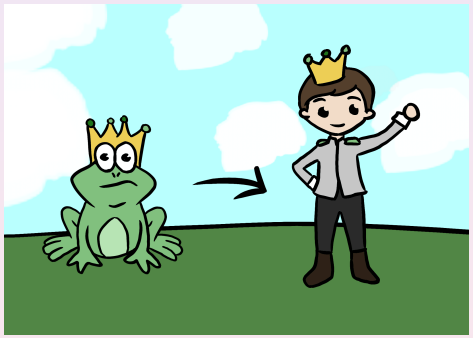
fixed point arithmetic, signed or unsigned, $M$ bits, $N$ decimals after the point: currently not implemented

# Back to C's `void *`

**`address`**

A 20-bytes Ethereum address

- it has a `balance` field
- there is no `instanceof` operator
- casts never fail



- you can dress a frog as a prince, but it will remain a frog
- you can dress a prince as a frog, but it will remain a prince
- you will understand the difference when you kiss it (segmentation fault)

# Basic Solidity types

## `bytes`*N*

fixed-size array of bytes, of length *N*

## `bytes` or `string`

variable-sized arrays of bytes

## Arrays

`uint32[][5]` is a fixed size array of five dynamic arrays of 32 bits unsigned integers

## Enumerations

```
enum NAME { A, B, ... }
```

# Basic Solidity types

## Structures

```
struct pair {
  int16 x;
  unint8 y;
}
```

## Mappings

```
mapping(address => unint256) balances;
```

A field of type mapping spreads its values into the state through hashing:
`balances[k]=v` executes `put(` $\underbrace{\texttt{hash(balances,k)}}_{\textit{address of } \texttt{balances[k]}}$ `, v)`

- $\Rightarrow$ mappings default to 0
- $\Rightarrow$ there is no `containsKey` (you need a sentinel value)
- $\Rightarrow$ it is not possible to compute the key set or value set of a mapping
- $\Rightarrow$ it is not possible to iterate on a mapping

# Unit multipliers

**Time units: `seconds`, `minutes`, `hours`, `days`**

```
uint delay = 3 hours;
```

**Ether units: `wei`, `finney`, `szabo`, `ether`**

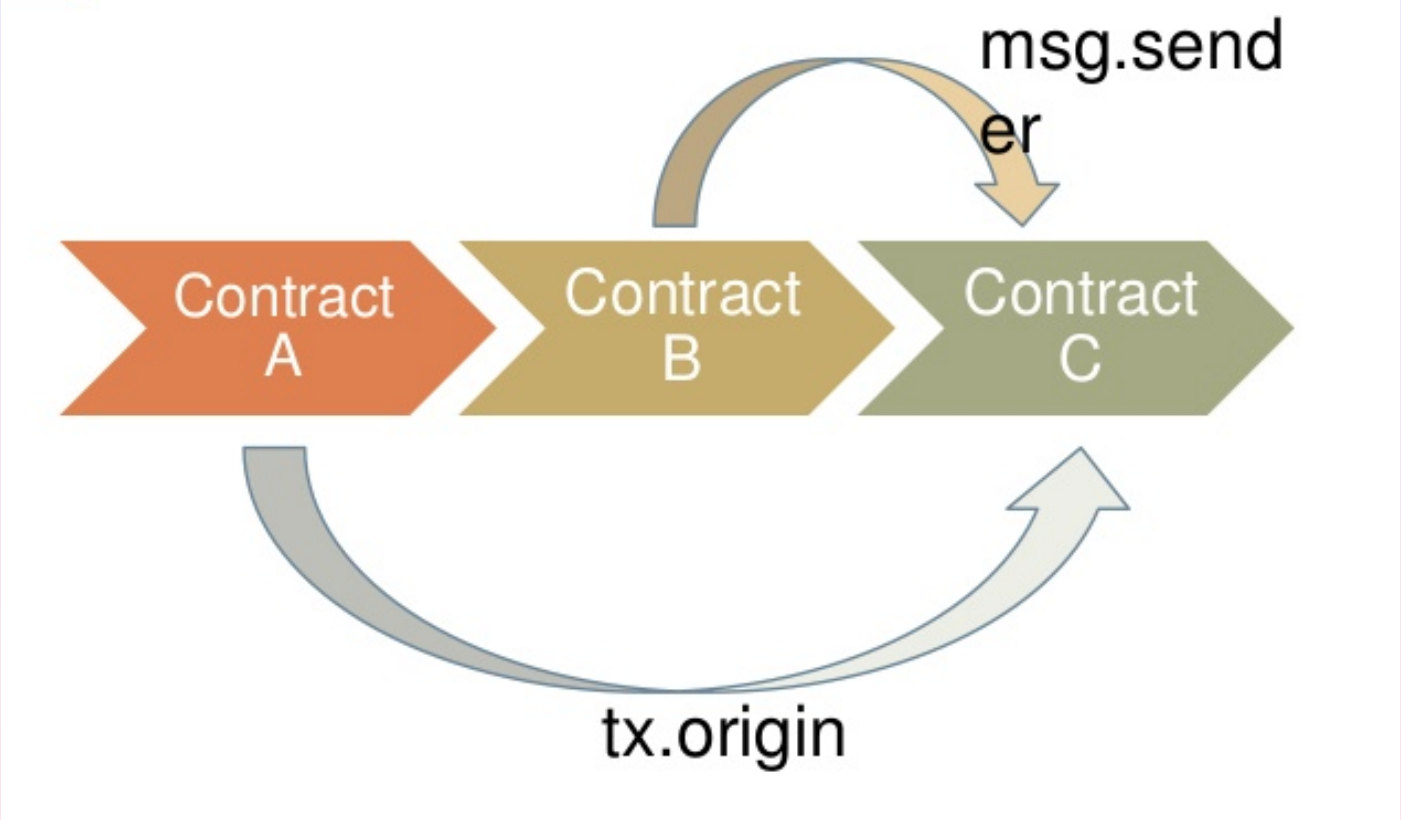
```
require(withdraw_amount <= 0.1 ether);
```

# Transaction information

**Structures `msg` and `tx` are derived from the transaction request**

| | |
|---|---|
| `msg.sender` | the address of the caller EOA or contract |
| `msg.value` | the ether sent along the transaction |
| `msg.gasleft` | what remains to consume of the gas limit |
| `msg.data` | the data payload of the transaction |
| `msg.sig` | the first four bytes of `msg.data` (method selector) |
| `tx.gasprice` | the gas price used for the transaction |
| `tx.origin` | the address of the originating EOA |

# msg.sender VS tx.origin



# Members of an address

`address.balance` the balance of the address (EOA or contract), such as `address(this).balance`

`address.transfer(x)` sends $x$ wei to the address; throws an exception in case of failure

`address.send(x)` as `transfer`, but returns `false` in case of failure

`address.call(p)` sends a method transaction to the address, with the given payload; returns `false` in case of failure

`address.callcode(p)` sends a method transaction to the address, with the given payload; runs in the scope of the caller; returns `false` in case of failure

`address.delegatecode(p)` sends a method transaction to the address, with the given payload, but keeps `msg.sender` and `msg.value` unchanged; runs in the scope of the caller; returns `false` in case of failure

# Block information

**The structure `block` is constructed by the miner**

`block.coinbase` the address of the miner itself

`block.difficulty` the proof of work difficulty

`block.gaslimit` the maximum gas that can be spent in the current block

`block.number` the current block height

`block.timestamp` the mining time

# Main types in Solidity

`contract` somehow similar to an object in OO-programming. It has a balance

`interface` functions have no code and must be implemented in subtypes

`library` a contract meant to be deployed once and used by other contracts through `delegatecall`. It has no state variables and no balance. It does not receive ether. It cannot execute `selfdestruct`.

```
pragma solidity ^0.4.24;

library Library {
    function addUint(uint a , uint b) public pure returns(uint) {
        uint c = a + b;
        require(c >= a);   // make sure the right computation was made
        return c;
    }
}
```

# Visibility modifiers

| | |
|---|---|
| `public` | can be called from everywhere |
| `external` | like `public` but, if called from the same contract, must be prefixed with `this.`, since it triggers a new transaction. It costs less gas than `public` |
| `internal` | can only be called from the contract where it is defined or from its subtypes |
| `private` | can only be called from the contract where it is defined |

**Privacy?**

Data is always publicly readable in blockchain. These modifiers only refer to who can call whom and are significant for the compiler only

# Side-effect modifiers

`view` or ~~`constant`~~ the function cannot modify the state

```
contract Test {
    uint i;

    function foo() view public {
        i++; // does not compile
    }
}
```

`pure` the function cannot modify nor read the state

```
contract Test {
    uint i;

    function foo() pure public {
        goo(i); // does not compile
    }

    function goo(uint j) pure public {

    }
}
```

`payable` the function can receive incoming payments

# Specifying return type(s) and value(s)

```
function min(int a, int b) pure public returns(int) {
    return a < b ? a : b;
}

function split(int n, int divisor) pure public returns(int, int) {
    return (n / divisor, n % divisor);
}

function split2(int n, int divisor) pure public returns(int quotient, int rest) {
    quotient = n / divisor;
    rest = n % divisor;
}
```

**`return` is not mandatory on every execution path**

When missing, the return value(s) defaults to 0

# Constructors

A contract can have up to one contructor, that gets called only once as part of each transaction that deploys an instance of the contract

```
pragma solidity ^0.6.0;

contract Test {
    uint internal i;

    constructor(uint _i) public {
        i = _i;
    }

    function get() public view returns(uint) {
        return i;
    }
}
```

# Self destruction example

Only needed to free state data in clients!

```
pragma solidity ^0.4.22;

contract Faucet {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function withdraw(uint withdraw_amount) public {
        require(withdraw_amount <= 0.1 ether);
        msg.sender.transfer(withdraw_amount);
    }

    function () external payable {}

    function destroy() public {
        require(msg.sender == owner);
        selfdestruct(owner); // send all balance to the owner
    }
}
```

# Self destruction example from 0.6.0

```
pragma solidity ^0.6.0;

contract Faucet {
    address payable owner;

    constructor() public {
        owner = msg.sender;
    }

    function withdraw(uint withdraw_amount) public {
        require(withdraw_amount <= 0.1 ether);
        msg.sender.transfer(withdraw_amount);
    }

    receive () external payable {}

    function destroy() public {
        require(msg.sender == owner);
        selfdestruct(owner); // send all balance to the owner
    }
}
```

# address payable

## Starting from 0.5.0, an `address` can be `payable`

- required for calling `send()`, `transfer()` and `call()`
- all of `msg.sender`, `tx.origin` and `block.coinbase` are `payable`
- you can cast between `address` and `address payable` and casts never fail

## Starting from 0.6.0, a contract can have at most a `receive` function

- it must be `external payable`
- called for transactions without data

## Starting from 0.6.0, a contract can have at most a `fallback` function

- it must be `external`
- called for transactions with data targetting no function in the contract or for transactions without data if there is no `receive` function

# Modifiers

## They can be used to weave aspects into code

```
pragma solidity ^0.6.0;

contract Faucet {
    address payable owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }

    function withdraw(uint withdraw_amount) public {
        require(withdraw_amount <= 0.1 ether);
        msg.sender.transfer(withdraw_amount);
    }

    receive () external payable {}

    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}
```

# Modifiers

## They can be used to do rather weird things

```
pragma solidity ^0.6.0;

contract Twice {
    uint i;

    modifier twice {
        _;_;
    }

    function increment() public twice {
        i++;
    }
}
```

# Contract extension and inheritance (also multiple)

```
pragma solidity ^0.6.0;

contract Owned {
    address payable owner;

    constructor() public {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}

contract Mortal is Owned {
    function destroy() public onlyOwner {
        selfdestruct(owner);
    }
}

contract Faucet is Mortal {
    function withdraw(uint withdraw_amount) public {
        require(withdraw_amount <= 0.1 ether);
        msg.sender.transfer(withdraw_amount);
    }

    receive () external payable {}
}
```

# Constructor chaining

```
pragma solidity ^0.6.0;

contract Base {
    uint x;
    constructor(uint _x) public { x = _x; }
}

// either directly in the inheritance list...
contract Derived1 is Base(7) {
    constructor() public {}
}

// or through a "modifier" of the derived constructor
contract Derived2 is Base {
    constructor(uint _y) Base(_y * _y) public {}
}
```

# Assertions

**`require(condition[,msg])`**

Evaluates the condition, that could reasonably not hold. If false, an exception is thrown and the transaction is reverted. Typically used for checking user input

**`assert(condition[,msg])`**

Evaluates the condition, that is expected to hold. If false, an exception is thrown and the transaction is reverted. Typically used to verify internal runtime conditions for debugging

**`revert([msg])` or ~~`throw`~~**

Throws an exception and reverts the current transaction

There is no way to catch exceptions

# Events

```
event Withdrawal(address indexed to, uint amount);
event Deposit(address indexed from, uint amount);

function withdraw(uint withdraw_amount) public {
    require(withdraw_amount <= 0.1 ether);
    msg.sender.transfer(withdraw_amount);
    emit Withdrawal(msg.sender, withdraw_amount);
}

receive () external payable {
    emit Deposit(msg.sender, msg.value);
}
```

DApps can listen and react to events

# Specifying gas and value for inner transactions

## `o.foo(pars)` can be decorated

- `o.foo.value(v)(pars)`
- `o.foo.gas(g)(pars)`
- `o.foo.gas(g).value(v)(pars)`

## `new C(pars)` can be decorated

- `(new C).value(v)(pars)`
- `(new C).gas(g)(pars)`
- `(new C).gas(g).value(v)(pars)`

# Contract creation and function calls

```
pragma solidity ^0.5.0;

import "Faucet.sol";
import "Mortal.sol";

contract Token is Mortal {
    Faucet faucet;

    constructor() public {
        faucet = new Faucet();
    }

    function destroy() public onlyOwner {
        faucet.destroy();
    }
}
```

In this case, we know what gets called with `faucet.destroy()`

# Contract creation with payable constructor

## If the constructor of `Faucet` were made `payable`

```
pragma solidity ^0.5.0;

import "Faucet.sol";
import "Mortal.sol";

contract Token is Mortal {
    Faucet faucet;

    constructor() public {
        faucet = (new Faucet).value(0.5 ether)();
    }

    function destroy() public onlyOwner {
        faucet.destroy();
    }
}
```

Also in this case, we know what gets called with `faucet.destroy()`

# Casts do not fail in Solidity

```
pragma solidity ^0.4.26;

contract D {
    constructor () payable public {
    }

    function callFoo(address e) public {
        E(e).foo.value(100)();
    }
}

contract E {
    function foo() public payable {
        // do something with the received value
    }
}

contract C {
    // accept any incoming amount
    function () external payable {}
}
```

Calling `d.callFoo(c)`, where `c` is a `C`, does not fail but calls `C`'s fallback function and transfers 100 wei to `c`!

# Parameter contract types are just casts

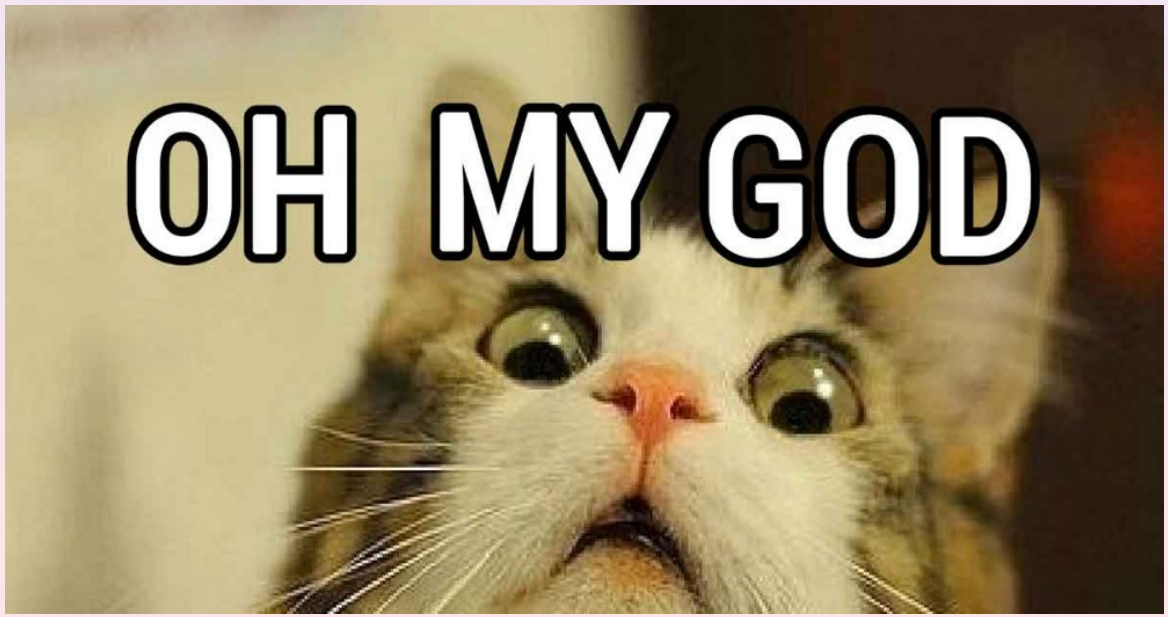
```
pragma solidity ^0.4.26;

contract D {
    constructor () payable public {
    }

    function callFoo(E e) public {
        e.foo.value(100)();
    }
}

contract E {
    function foo() public payable {
        // do something with the received value
    }
}

contract C {
    // Accept any incoming amount
    function () external payable {}
}
```

Calling `d.callFoo(c)`, where `c` is a `C`, does not fail but calls `C`'s fallback function and transfers 100 wei to `c`!

# Solidity is not strongly-typed

1. casts are not checked
2. parameter types are just Christmas decorations

Remember that a function declaring a formal parameter of type `address` or explicitly `c` can actually receive any contract, of any type, also completely unrelated to `c`. Callers can inject malicious code through such parameters!



# Members of an address for low-level calls

`address.call(p)` sends a method transaction to the address, with the given payload; returns `false` in case of failure. This is actually used by the compiler when we write normal function calls

`address.callcode(p)` sends a method transaction to the address, with the given payload; runs in the scope of the caller; returns `false` in case of failure

`address.delegatecode(p)` sends a method transaction to the address, with the given payload, but keeps `msg.sender` and `msg.value` unchanged; runs in the scope of the caller; returns `false` in case of failure

# Example of function call

```
pragma solidity ^0.4.26;

contract D {
    uint public n;
    address public sender;

    function callSetN(address _e, uint _n) public {
        E(_e).setN(_n);
    }
}

contract E {
    uint public n;
    address public sender;

    function setN(uint _n) public {
        n = _n;
        sender = msg.sender;
    }
}
```

Calling `d.callSetN(e, 42)` where `e` is a `E`:

- `e.n` will become 42
- `e.sender` will become `d`
- `d.n` and `d.sender` are not modified

# The previous example is actually compiled into this

```
pragma solidity ^0.4.26;

contract D {
    uint public n;
    address public sender;

    function callSetN(address _e, uint _n) public {
        _e.call(abi.encodeWithSignature("setN(uint256)", _n));
    }
}

contract E {
    uint public n;
    address public sender;

    function setN(uint _n) public {
        n = _n;
        sender = msg.sender;
    }
}
```

Calling `d.callSetN(e, 42)` where `e` is a `E`:

- `e.n` will become 42
- `e.sender` will become `d`
- `d.n` and `d.sender` are not modified

# Example of low-level `callcode`

```
pragma solidity ^0.4.26;

contract D {
    uint public n;
    address public sender;

    function callSetN(address _e, uint _n) public {
        _e.callcode(abi.encodeWithSignature("setN(uint256)", _n));
    }
}

contract E {
    uint public n;
    address public sender;

    function setN(uint _n) public {
        n = _n;
        sender = msg.sender;
    }
}
```

Calling `d.callSetN(e, 42)` where `e` is a `E`:

- `d.n` will become 42
- `d.sender` will become `d`
- `e.n` and `e.sender` are not modified

# Example of low-level `delegatecall`

```
contract D {
    uint public n;
    address public sender;

    function callSetN(address _e, uint _n) public {
        _e.delegatecall(abi.encodeWithSignature("setN(uint256)", _n));
    }
}

contract E {
    uint public n;
    address public sender;

    function setN(uint _n) public {
        n = _n;
        sender = msg.sender;
    }
}

contract C {
    function foo(D _d, E _e, uint _n) public {
        _d.callSetN(_e, _n);
    }
}
```

Calling `c.foo(d, e, 42)` where `d` is a `D` and `e` is a `E`:

- `d.n` will become 42
- `d.sender` will become `c`
- `e.n` and `e.sender` are not modified

# Gas consumption

## Each bytecode instruction and transaction type has a gas cost

- it is possible to compute in advance the gas cost of simple functions (use `estimateGas` of web3 for instance)
- the result is wrong in the presence of loops or recursion!
  - obvious, since gas cost computation is equivalent to complexity analysis which can be used to decide termination of programs
    - $\Rightarrow$ an algorithm for computing gas costs in advance cannot exist
- in general, it is important to know which operations (might) cost much gas, and avoid them
  - loops over unbounded dynamic arrays
  - calls to unknown contracts

# A simple Ponzi scheme

```
pragma solidity ^0.4.21;

contract SimplePonzi {
    address public currentInvestor;
    uint public currentInvestment = 0;

    function () payable external {
        uint minimumInvestment = currentInvestment * 11 / 10;
        require(msg.value > minimumInvestment);
        address previousInvestor = currentInvestor;
        currentInvestor = msg.sender;
        currentInvestment = msg.value;
        // for malicious investors it will return false but not fail
        previousInvestor.send(msg.value);
    }
}
```

The first investment will be burned to address `0x0`

# Exercise

1. create an account with MetaMask
2. charge it from the Ropsten faucet with 1 ETH
3. write the `SimplePonzi.sol` contract in Remix
4. compile the contract in Remix
5. deploy the contract in Ropsten with Remix
6. connect to the contract in Ropsten by somebody else
7. start calling the fallback function with increasing value (with Remix of with MetaMask; for the latter, remember to increase to gas limit for sending ETH)

You can check the current investment and investor from Remix

# A gradual Ponzi scheme

```
pragma solidity ^0.4.21;

contract GradualPonzi {
    address[] public investors; // dynamic array
    mapping (address => uint) public balances; // map
    uint public constant MINIMUM_INVESTMENT = 1e15;

    constructor () public {
        investors.push(msg.sender);
    }

    function () public payable {
        require(msg.value >= MINIMUM_INVESTMENT);
        uint eachInvestorGets = msg.value / investors.length;
        for (uint i = 0; i < investors.length; i++)
            balances[investors[i]] += eachInvestorGets;
        investors.push(msg.sender);
    }

    function withdraw() public {
        uint payout = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(payout);
    }
}
```

# A simple lottery (1)

```
pragma solidity ^0.4.21;

contract SimpleLottery {
    uint public constant TICKET_PRICE = 1e16; // 0.01 ETH
    address public owner;
    address[] public players;
    address public winner;
    uint public ticketingCloses;

    constructor (uint duration) public {
        owner = msg.sender;
        ticketingCloses = now + duration; // seconds
    }

    function buy() public payable {⟨⟩}

    function drawWinner() public {⟨⟩}

    function withdraw() public {⟨⟩}

    function () payable public {⟨⟩}
}
```

`now` is the timestamp of the block where the transaction is mined

# A simple lottery (2)

```
contract SimpleLottery {
    uint public constant TICKET_PRICE = 1e16; // 0.01 ETH
    address public owner;
    address[] public players;
    address public winner;
    uint public ticketingCloses;

    constructor (uint duration) public {}

    function buy() public payable {
        require(msg.value == TICKET_PRICE);
        require(now < ticketingCloses);
        players.push(msg.sender);
    }
```

# A simple lottery (3)

```
contract SimpleLottery {
    uint public constant TICKET_PRICE = 1e16; // 0.01 ETH
    address public owner;
    address[] public players;
    address public winner;
    uint public ticketingCloses;

    constructor (uint duration) public {⟺}

    function buy() public payable {⟺}

    function drawWinner() public {
        require(msg.sender == owner);
        require(players.length > 0);
        require(now > ticketingCloses + 5 minutes);
        require(winner == address(0)); // not set yet
        bytes32 rand = keccak256(blockhash(block.number - 1));
        winner = players[uint(rand) % players.length];
    }
```

# A simple lottery (4)

```
contract SimpleLottery {
    uint public constant TICKET_PRICE = 1e16; // 0.01 ETH
    address public owner;
    address[] public players;
    address public winner;
    uint public ticketingCloses;

    constructor (uint duration) public {⟺}

    function buy() public payable {⟺}

    function drawWinner() public {⟺}

    function withdraw() public {
        require(msg.sender == winner);
        msg.sender.transfer(this.balance);
    }

    function () payable public {
        buy(); // who sends money, gets a ticket
    }
}
```

# Exercise

1. create an account with MetaMask
2. charge it from the Ropsten faucet with 1 ETH
3. write the `SimpleLottery.sol` contract in Remix
4. compile the contract in Remix
5. deploy the contract in Ropsten with Remix, with 3 minutes for buying tickets
6. connect to the contract in Ropsten by somebody else
7. buy some tickets before the 3 minutes expire (with Remix or with MetaMask; for the latter, remember to increase to gas limit for sending ETH)
8. the owner of the lottery contract draws the winner
9. try to withdraw your price (if any!)

You can check the winner from Remix

# A simple auction (1)

```
pragma solidity ^0.5.0;

contract SimpleAuction {
    address payable public owner;
    uint public auctionEndTime;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    bool ended;

    constructor(uint biddingTime) public {
        owner = msg.sender;
        auctionEndTime = now + biddingTime;
    }

    function bid() public payable {}

    function withdraw() public returns (bool) {}

    function auctionEnd() public {}
}
```

# A simple auction (2)

```
contract SimpleAuction {
    address payable public owner;
    uint public auctionEndTime;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    bool ended;

    constructor(uint biddingTime) public {⏎}

    function bid() public payable {
        require(now <= auctionEndTime, "Auction already ended");
        require(msg.value > highestBid, "There is already a higher bid");

        if (highestBid != 0)
            pendingReturns[highestBidder] += highestBid;

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    function withdraw() public returns (bool) {⏎}

    function auctionEnd() public {⏎}
}
```

# A simple auction (3)

```
contract SimpleAuction {
    address payable public owner;
    uint public auctionEndTime;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    bool ended;

    constructor(uint biddingTime) public {⟺}

    function bid() public payable {⟺}

    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    function auctionEnd() public {⟺}
}
```

# A simple auction (4)

```
contract SimpleAuction {
    address payable public owner;
    uint public auctionEndTime;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;
    bool ended;

    constructor(uint biddingTime) public {⏎}

    function bid() public payable {⏎}

    function withdraw() public returns (bool) {⏎}

    function auctionEnd() public {
        require(now >= auctionEndTime, "Auction not yet ended");
        require(!ended, "auctionEnd has already been called");
        ended = true;
        owner.transfer(highestBid);
    }
}
```

# Why this complicated withdraw pattern?

```
function bid() public payable {
    require(now <= auctionEndTime, "Auction already ended");
    require(msg.value > highestBid, "There is already a higher bid");

    if (highestBid != 0)
        // highestBidder.transfer(highestBid); DON'T DO THIS!!!
        pendingReturns[highestBidder] += highestBid;

    highestBidder = msg.sender;
    highestBid = msg.value;
}

function withdraw() public returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        pendingReturns[msg.sender] = 0;

        if (!msg.sender.send(amount)) {
            pendingReturns[msg.sender] = amount;
            return false;
        }
    }
    return true;
}
```

# Why this complicated withdraw pattern?

`highestBidder.transfer(highestBid); DON'T DO THIS!!!`

The problem with this call (inside function `bid()`) is that a broken or malicious bidder might redefine its own fallback function in such a way to consume all gas (for instance):

- all subsequent bidders will get a failure when calling `bid()`
- the broken or malicious contract will never be replaced by another bidder

# The reentrancy nightmare

```
function withdraw() public returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        pendingReturns[msg.sender] = 0;

        if (!msg.sender.send(amount)) {
            pendingReturns[msg.sender] = amount;
            return false;
        }
    }
    return true;
}
```

Why this complication of setting `pendingReturns[msg.sender]=0` so early?

# The reentrancy nightmare

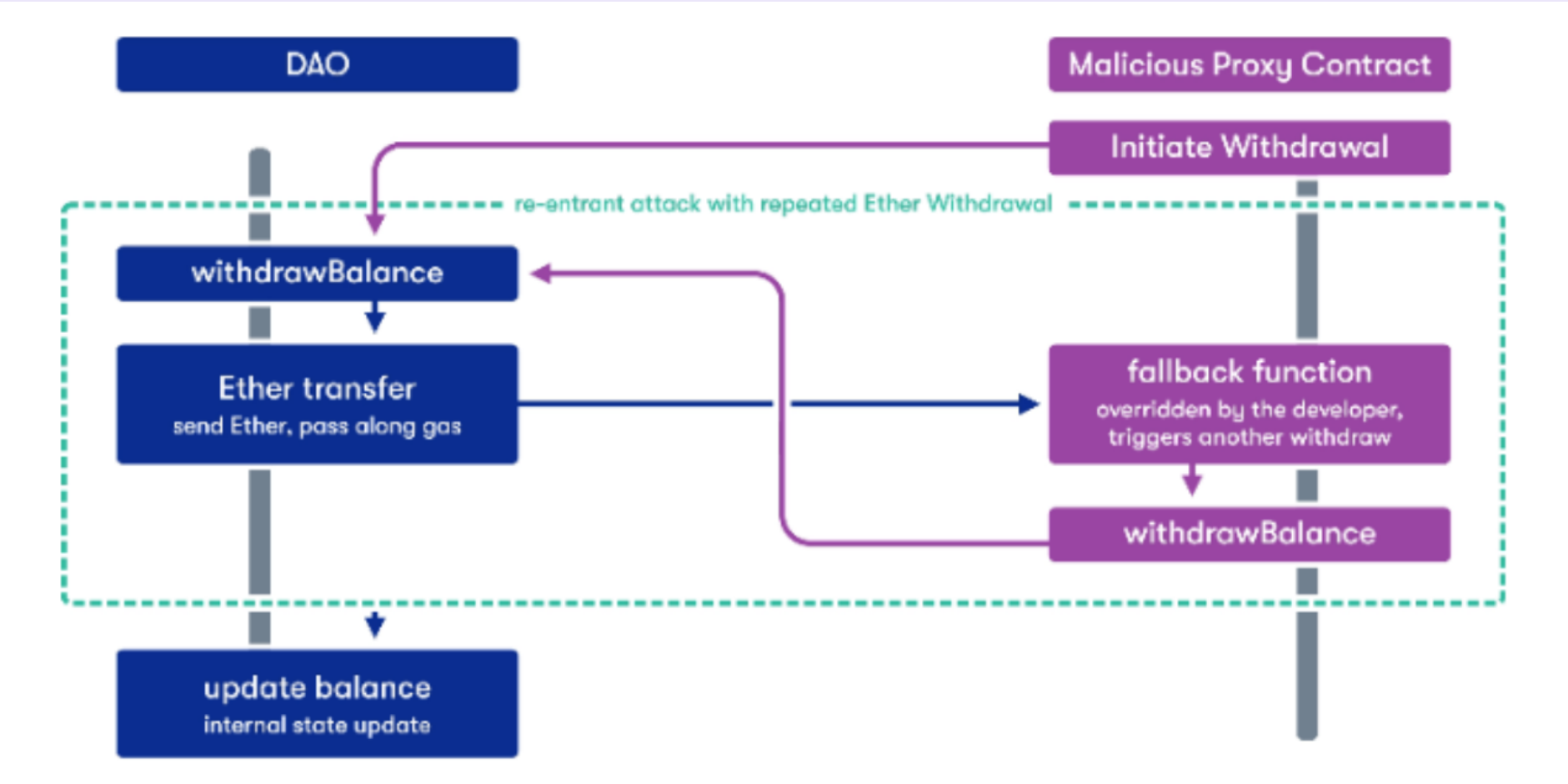
Beware: code vulnerable to reentrancy:

```
function withdraw() public returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        if (!msg.sender.send(amount))
            return false;

        pendingReturns[msg.sender] = 0;
    }
    return true;
}
```

## Order does matter

- a malicious bidder might redefine its own fallback function in order to call `withdraw()` back again, as many times as it likes
- it will withdraw its `amount` as many times as it wants, until the auction contract is depleted
- basically, it runs away with the highest bid and with the bids of all bidders that have not been withdrawn yet

# The reentrancy nightmare



# The DAO attack (2016)

## The most famous reentrancy exploit

- the DAO was a contract for autonomous decentralized organizations
- the attacker used reentrancy to steal 50M$ equivalent of ETH
- the Ethereum team decided to make the consensus rules more restrictive in order to make such transactions illegal and get some of that money back
- some node maintainers didn't accept the change and continued operating with the old rules and another chain id, leading to a network hard fork known as Ethereum Classic

# Are `send()` and `transfer()` safe from reentrancy?

## Currently yes

- they only forward 2300 units of gas, not enough for reentrancy
- differently from the low-level `msg.sender.call.value(amount)("")`, that forwards all gas and definitely allows reentrancy
- however:
  - gas costs might change in the future, allowing reentrancy also with 2300 units of gas
  - the programmer might decide to forward more gas explicitly in order to deal with contracts with complex fallback functions
  - `send()` and `transfer()` risk to be deprecated soon...
- in conclusion, it's good practice to set `pendingReturns[msg.sender] = 0` before sending the money to `msg.sender`, however that last operation is performed

# A blind auction (1)

## In our simple auction, everything happens in clear

We want a blind auction now, where:

- **start** an EOA starts the auction and becomes its owner
- **bidding phase** bidders place zero, one or more bids in closed envelopes; some envelopes might contain bids marked as fake, which allows one to bluff
- **revealing phase** bidders provide the secret needed to open the envelopes
- **end** the best non-fake bid is forwarded to the auction's owner

# A blind auction (2)

```
pragma solidity ^0.4.21;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address owner;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;
    mapping(address => Bid[]) public bids;
    address public highestBidder;
    uint public highestBid;
    mapping(address => uint) pendingReturns;

    constructor(uint biddingTime, uint revealTime) public {
        owner = msg.sender;
        biddingEnd = now + biddingTime;
        revealEnd = biddingEnd + revealTime;
    }
```

# A blind auction (3)

```
modifier onlyBefore(uint _time) { require(now < _time); _; }
modifier onlyAfter(uint _time) { require(now > _time); _; }

/// Place a blinded bid with `_blindedBid` =
/// keccak256(abi.encodePacked(value, fake, secret)).
/// The sent ether is only refunded if the bid is correctly
/// revealed in the revealing phase. The bid is valid if the
/// ether sent together with the bid is at least "value" and
/// "fake" is not true. Setting "fake" to true and sending
/// not the exact amount are ways to hide the real bid but
/// still make the required deposit. The same address can
/// place multiple bids.
function bid(bytes32 _blindedBid) public payable onlyBefore(biddingEnd) {
    bids[msg.sender].push(Bid({ blindedBid: _blindedBid, deposit: msg.value }));
}
```

# A blind auction (4)

```
/// Reveal your blinded bids. You will get a refund for all correctly blinded
/// invalid bids and for all bids except for the totally highest.
function reveal(uint[] _values, bool[] _fake, bytes32[] _secret)
        public onlyAfter(biddingEnd) onlyBefore(revealEnd) {
    uint length = bids[msg.sender].length;
    require(_values.length == length && _fake.length == length && _secret.length == length);

    uint refund = 0;
    for (uint i = 0; i < length; i++) {
        Bid storage bidToCheck = bids[msg.sender][i];
        (uint value, bool fake, bytes32 secret) = (_values[i], _fake[i], _secret[i]);
        if (bidToCheck.blindedBid != keccak256(abi.encodePacked(value, fake, secret)))
            continue; // bid was not actually revealed: do not refund the deposit

        refund += bidToCheck.deposit;
        if (!fake && bidToCheck.deposit >= value && value > highestBid) {
            updateHighestBid(msg.sender, value);
            refund -= value;
        }
        bidToCheck.blindedBid = bytes32(0); // cannot claim it back again
    }
    msg.sender.transfer(refund);
}

function updateHighestBid(address bidder, uint value) internal {
    if (highestBidder != address(0))
        pendingReturns[highestBidder] += highestBid;
    highestBid = value;
    highestBidder = bidder;
}
```

# A blind auction (5)

```
function withdraw() public {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        pendingReturns[msg.sender] = 0;
        msg.sender.transfer(amount);
    }
}

function auctionEnd() public onlyAfter(revealEnd) {
    require(!ended);
    ended = true;
    owner.transfer(highestBid);
}
```

# Exercise

## Write a `TicTacToe.sol` smart contract

- the first EOA that calls the payable function `play(x,y)` becomes player 1: she must pay at least 0.01 ETH
- the second EOA that calls the payable function `play(x,y)` becomes player 2: he must be different from player 1; he must pay at least what player 1 payed
- subsequent calls to `play` do not require any payed amount
- if one player wins, the balance of the game goes
  - 90% to the winner
  - 10% to the creator of the game
- if the game is a draw, 10% of the balance of the game goes to the creator of the game
- after victory or draw, the game resets and is ready to accept new players

# Exercise

**Write a user interface that connects to the `TicTacToe.sol` smart contract**

- as a Java textual desktop application using Infura
- as a Java graphical desktop application using Infura
- as an Android mobile application using Infura
- as a web application (Javascript) using Infura
- the same as above, but connecting directly to an Ethereum node...

# Security best practice

- Minimalism: the simpler, the better
- Code reuse: DRY, use well-known libraries
- Study: be aware of well-known issues and solutions
- Readability: simpler audit
- Test: try corner cases
- Analysis: static or dynamic, still in infancy

Considering the importance of security for smart contracts, it is questionable to have invented Solidity (hard, new, with complex semantics) for writing such delicate pieces of software

Remember the DAO

# Issue #1: Reentrancy

Our contract calls an unknown contract. The latter, unexpectedly, calls back into our contract again. Particularly dangerous if this cycle passes through a call sending ETH, that would be repeated as many times as the unknown contract decides

## Three ways of sending ETH

`c.send(amount)` forwards 2300 units of gas, currently not enough for reentrancy; possibly deprecated in the future

`c.transfer(amount)` forwards 2300 units of gas, currently not enough for reentrancy; possibly deprecated in the future

`c.call.value(amount)("")` forwards all remaining units of gas, typically enough for reentrancy; programmers like it since it allows one to activate the complex fallback function of any contract, not just the default fallback function

# Reentrancy: prevention

## Use the checks/effects/interactions pattern

```
function withdraw() public returns (bool) {
    uint amount = pendingReturns[msg.sender];
    if (amount > 0) {
        pendingReturns[msg.sender] = 0;

        if (!msg.sender.send(amount)) {
            pendingReturns[msg.sender] = amount;
            return false;
        }
    }
    return true;
}
```

## Use a mutex to ban reentrancy

```
pragma solidity ^0.4.21;

contract Mutex {
    bool internal mutex = false;

    function withdraw(uint amount) external {
        require(!mutex);
        mutex = true;
        msg.sender.call.value(amount)();
        mutex = false;
    }
}
```

# Issue #2: Arithmetic over/underflows

An operation tries to compute a fixed-size value that is outside the bounds for its type

What if an account with 0 balance tries to send 1 ETH to another account?

```
pragma solidity ^0.5.0;

contract Token {
    mapping (address => uint) balances;

    constructor (uint totalSupply) public {
        balances[msg.sender] = totalSupply;
    }

    function transfer(address to, uint value) public {
        require(balances[msg.sender] - value >= 0);
        balances[msg.sender] -= value;
        balances[to] += value;
    }
}
```

# Over/underflows: prevention

## Test numerical operations

```
pragma solidity ^0.5.0;

contract Token {
    mapping (address => uint) balances;

    constructor (uint totalSupply) public {
        balances[msg.sender] = totalSupply;
    }

    function transfer(address to, uint value) public {
        require(balances[msg.sender] >= value);
        balances[msg.sender] = balances[msg.sender] - value;
        require(balances[to] + value >= balances[to]);
        balances[to] = balances[to] + value;
    }
}
```

# Over/underflows: prevention

## Replace numerical operations with a library for safe arithmetic

```
pragma solidity ^0.5.0;

import "SafeMath.sol";

contract Token {
    mapping (address => uint) balances;

    constructor (uint totalSupply) public {
        balances[msg.sender] = totalSupply;
    }

    function transfer(address to, uint value) public {
        require(balances[msg.sender] >= value);
        balances[msg.sender] = SafeMath.sub(balances[msg.sender], value);
        balances[to] = SafeMath.add(balances[to], value);
    }
}
```

# Over/underflows: prevention

## A library for safe arithmetic

```
pragma solidity ^0.5.0;

library SafeMath {

    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        assert(b <= a);
        return a - b;
    }

    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        assert(c >= a);
        return c;
    }
}
```

https://docs.openzeppelin.com/contracts/3.x/api/math

# Issue #3: Unexpected ether

Code correctness depends on invariants about `this.balance`, that actually do not hold

This code assumes `this.balance % 0.5 ether == 0`. But this is false since it is possible to send 0.1 ether to this contract...

```
pragma solidity ^0.5.0;

contract EtherGame {
    uint public constant milestone = 10 ether;
    uint public constant step = 0.5 ether;

    function play() external payable {
        require(msg.value == step);
        uint currentBalance = address(this).balance + step;
        require(currentBalance <= milestone);
        if (currentBalance == milestone)
            msg.sender.transfer(milestone);
    }
}
```

# Issue # 3: Unexpected ether

It is possible to send ether to a contract, without calling any `payable` function nor its fallback function (if any)!

## How?

- write a suicidal contract that executes `selfdestruct(beneficiary)` and forwards its balance to `beneficiary`: it does not even call the fallback function of `beneficiary`!
- mine a block and put the contract as destination coinbase address (expensive, but it depends on the final goal...)
- let ether be there from the very beginning, by sending ether to the address of the contract before its same creation! Remember that the address of a contract is computed deterministically from the address of the creator and from its current nonce

# Unexpected ether: prevention

**Do not rely on `this.balance`**

Use your own counter instead

```
pragma solidity ^0.5.0;

contract EtherGame {
    uint public constant milestone = 10 ether;
    uint public constant step = 0.5 ether;
    uint internal deposit;

    function play() external payable {
        require(msg.value == step);
        deposit += step;
        require(deposit <= milestone);
        if (deposit == milestone) {
            deposit = 0;
            msg.sender.transfer(milestone);
        }
    }
}
```

# Issue #4: `delegatecall`

It is a low-level call that allows one to call any method, as done in Java by reflection (actually much worse than reflection)

**It keeps the calling context**

If the callee reads/modifies its *n*th state variable, it actually reads/modifies the *n*th state variable of the caller, regardless of names and types

# Example: storage slot clash

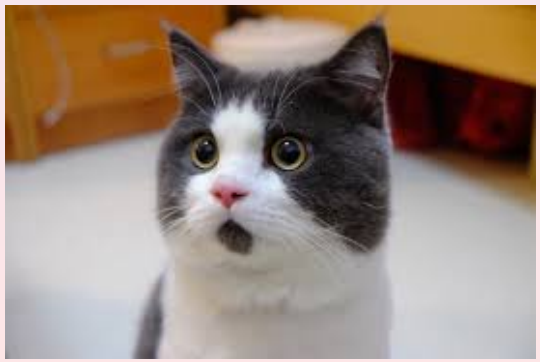
```
pragma solidity ^0.4.21;

contract Callee {
    uint public i;

    function set(uint _i) public {
        i = _i;
    }
}

contract Caller {
    address private lib;

    constructor(address _lib) public {
        lib = _lib;
        lib.delegatecall(abi.encodeWithSignature("set(uint256)", 42));
    }
}
```

By invoking the constructor of `Caller` with the address of a `Callee` . . .

# Example: storage slot clash

```
pragma solidity ^0.4.21;

contract Callee {
    uint public i;

    function set(uint _i) public {
        i = _i;
    }
}

contract Caller {
    address private lib;

    constructor(address _lib) public {
        lib = _lib;
        lib.delegatecall(abi.encodeWithSignature("set(uint256)", 42));
    }
}
```

By invoking the constructor of `Caller` with the address of a `Callee` one actually sets the `private` field `lib` of the `Caller` to 42



# Example: call injection

```
function () public {
    lib.delegatecall(msg.data);
}
```

The caller can forge proper data so that `delegatecall` ends up calling everything!

The same can occur with all low-level calls

# `delegatecall`: prevention

- use it only on `library`s: they cannot have state, hence they do not read/modify fields
  - double-check that the receiver cannot be injected
- use it with prepared call data only, to avoid call injections
- let the compiler, not you, use it, to compile calls to `library`s

# Issue #5: Weak encapsulation

Programmers tend to be too permissive about visibility modifiers, often because they get along with the default `public` visibility

```
pragma solidity ^0.4.21;

contract HashForEther {
    function withdrawWinnings() {
        require(uint32(msg.sender) == 0);
        sendWinnings();
    }

    function sendWinnings() {
        msg.sender.transfer(this.balance);
    }
}
```

Everybody can call `sendWinnings()`, it ~~is~~ was `public` by default!

Too stupid? Do not overestimate the intelligence of programmers: somebody stole $31M from a slightly more complex contract than this

# Weak encapsulation: prevention

- be the stricter you can about visibility
  - even `internal` can be dangerous, because it is callable from subcontracts
- beware of default visibility (*ie.*, `public`)
  - latest compilers forbid default visibility
- think seriously about the API of your contracts

What, in traditional software development, is *just* a question of style, order and good engineering, becomes here a question of MONEY

# Issue #6: Entropy illusion

**Nothing is really random in blockchain**

Do not rely on `block.number`, `block.gaslimit`, `block.timestamp` or `now` as a source of randomness: they are (partially) controlled by the miner

```
pragma solidity ^0.4.21;

contract Roulette {
    uint public pastBlockTime;

    constructor () public payable {}

    function () public payable {
        require(msg.value == 10 ether);
        require(now != pastBlockTime);
        pastBlockTime = now;
        if (now % 15 == 0)
            msg.sender.transfer(address(this).balance);
    }
}
```

The miner might choose `now` so that the condition holds (or does not hold)

# Entropy illusion: prevention

- use an external trusted oracle as a source of randomness:

  Oracle A device that regularly sends transactions that store data in blockchain. For instance, it sends and stores a random number

- if `now` or `block.timestamp` is used to wait for a period of time, it might be safer to wait for a specific blockchain height instead, since it cannot really be controlled by miners

# Issue #7: External contract referencing

Casts and contract types are unchecked. Do not let clients inject contract types through the public API

## By injecting a malicious `_tail`, function `length` might run arbitrary code

```
pragma solidity ^0.4.21;

contract List {
    uint private head;
    List private tail;

    constructor (uint _head, List _tail) public {
        head = _head;
        tail = _tail;
    }

    function length() public returns(uint) {
        if (address(tail) == 0x0000000000000000000000000000000000000000)
            return 1;
        else
            return 1 + tail.length();
    }
}
```

This even compiles if `length` is defined as `view`, although it can have any possible side-effect

# External contract referencing: prevention

- avoid it!
- do not call functions on externally referenced contracts
- only talk to your friends!
  - if `field = new MyContract()` is the only way to initialize `field`, then you can be sure about the run-time type of `field`

# Issue #8: Short address/parameter attack

Function parameters are encoded as transaction data. If shorter than expected, the encoding is silently padded with 0's

## How to buy a Porsche with 1431 euros

1. generate private keys until the derived address `A` ends in `0x00`
2. communicate the first 38 digits `A'` of `A` to an exchange (without the two final `0x00`), asking to buy 1 ETH (as of today, 1431 euros)
3. pay the exchange the 1431 euros (plus commission)
4. its wallet will prepare a transaction to some `transfer(address to, uint tokens)`, with data `selector::A'::0x000000....00001::0x00`, where the final `0x00` is padding to match the expected data size for `transfer`
5. the EVM will interpret the address as `A'::0x00`, that is, `A` (ours!)
6. the EVM will interpret the `tokens` parameter as `0x0000...000100`, that is, 256 ETH (as of today, 366336 euros)
7. you will earn $336336 - 1431 = 364905$ euros (minus commission)

# Prevention: Short address/parameter attack

## Spoiler: exchanges check consistency of data nowadays


Let your applications check parameters passed into blockchain

# Issue #9: Unchecked return value

The return value of some functions (typically, `send` and `call`) should be checked, since it is a boolean that informs about their outcome

```
pragma solidity ^0.4.21;

contract Lotto {
    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // missing code not shown

    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount); // return value not checked!
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(address(this).balance);
    }
}
```

`payedOut` is set to true even if `winner.send(winAmount)` fails

Prevention: check the result of `send` and `call`. Use `transfer` instead of `send`

# Issue #10: Race conditions/front running

Miners process transactions from the mempool in the order they like (if ever); in general, they tend to proceed in decreasing `gasPrice` order

## Scenario #1 (transaction ban, unlikely)

A miner could be instructed to never include a transaction that is detrimental to its owner. Unlikely to work for long, since it is practically impossible to be the fastest to mine the next block, always

## Scenario #2 (transaction front running, likely)

A network sniffer could scan transactions, looking for those containing solutions to priced puzzles, and immediately place its own transaction with the same solution but a much higher `gasPrice`, so that all miners will prefer to mine its transaction rather than the original one. The onwer of the sniffer will cash the price

# Front running example

```
pragma solidity ^0.4.21;

contract FindThisHash {
    bytes32 constant public hash =
        0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;

    constructor () public payable {
        require(msg.value == 1000 ether);
    }

    function solve(string solution) public {
        require(hash == keccak256(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

1. clients try to guess the `solution` whose hash is the given one
2. a network sniffer scans all transactions, looking for calls to `solve`
3. for each call to `solve`, the sniffer verifies if the solution is correct
4. when, finally, a user submits the winning solution `Ethereum!`, the sniffer realizes the user is going to win and immediately places its own transaction with the same solution but a very high `gasPrice`
5. miners will likely prefer its transaction to the original one

# Prevention: race conditions/front running

- at the level of protocol, the consensus rules might impose a maximal `gasPrice`
- at the application level, solutions could be sent through the commit-reveal approach:
  1. first a hashed solution is sent
  2. later, when no new solutions are allowed, the original image of the hash is revealed

  - a big coding complication
  - payable amounts should be hidden, or otherwise the sniffer might suspect something when seeing high amounts
  - see the blind auction example

# Issue #11: Denial of service

If the cost of running transactions exceeds the block's gas limit, the transaction will never be executed, which might block a contract

```
pragma solidity ^0.4.21;

contract DistributeTokens {
    address public owner;
    address[] investors; // dynamic array
    uint[] tokens; // dynamic array

    constructor () public {
        owner = msg.sender;
    }

    function invest() public payable {
        investors.push(msg.sender);
        tokens.push(msg.value * 5);
    }

    function distribute() public {
        require(msg.sender == owner);
        for (uint i = 0; i < investors.length; i++) {
            // do something that distributes the investment here:
            // the gas cost of this loop is unbounded
        }
    }
}
```

# Prevention: denial of service

- use the withdrawal pattern
- perform very cheap operations inside loops (no inner transactions, such as `send`)
- impose an upper bound to the size of dynamic structures

# Issue #12: Floating point imprecision

Solidity currently does not implement fixed nor floating-point arithmetic, hence integers are used instead

Beware of calculations that might lose precision! In particular: integer division

```
pragma solidity ^0.4.21;

contract FunWithNumbers {
    uint constant public tokensPerEth = 10;
    uint constant public weiPerEth = 1e18;
    mapping (address => uint) public balances;

    function buyTokens() public payable {
        uint tokens = msg.value / weiPerEth * tokensPerEth; // !!
        balances[msg.sender] += tokens;
    }

    function sellTokens(uint tokens) public {
        require(balances[msg.sender] >= tokens);
        uint eth = tokens / tokensPerEth; // !!
        balances[msg.sender] -= tokens;
        msg.sender.transfer(eth * weiPerEth);
    }
}
```

# Prevention: floating point imprecision

- divide at the end: `msg.value * tokensPerEth / weiPerEth` rather than ~~`msg.value / weiPerEth * tokensPerEth`~~
- use libraries that simulate floating-point arithmetic

# Issue #13: `tx.origin` authentication

`tx.origin` is the sender of the first transaction in a chain of transactions, not the sender of the current transaction (*ie.*, it is not `msg.sender`)

```
contract Phishable {
    address public owner;
    constructor () public { owner = msg.sender; }
    function () public payable {}
    function withdrawAll(address recipient) public {
        require (tx.origin == owner); // !!
        recipient.transfer(address(this).balance);
    }
}

contract Phisher {
    Phishable phishable;
    address attacker;
    constructor (Phishable _phishable, address _attacker) public {
        phishable = _phishable; attacker = _attacker;
    }
    function () public payable { phishable.withdrawAll(attacker); }
}
```

1. an `attacker` identifies a `phishable` in blockchain
2. creates `phisher = new Phisher(phishable, attacker)`
3. convinces `phishable.owner` to send ETH to `phisher` and caches all the money of the `phishable`!

# Prevention: `tx.origin` authentication

- it is almost always correct to use `msg.sender` instead of `tx.origin`
- nevertheless, `tx.origin` is legitimate sometimes: `require(tx.origin == msg.sender)` constrains functions to be called from EOAs only