Ethereum Transaction

From:[account]

To;[account](optional)

Value:[int] in wei – one eth is 1018 wei

Gas;max amount of gas

Gasprice:amount of wei per gas

Data : ABI Byte string

Nonce : Integer of a nonce

Ethereum Transaction Signature

<https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7#:~:text=ECDSA%20signatures%20consist%20of%20two,d%E2%82%90%20)%20to%20sign%20it%20with>.

Cryptographic Hashing

5 main property :

1.it is deterministic so the same msg always results in the same hash.

2. it is quick to compute the hash value for any given msg.

3. it is infeasible to generate a msg from its hash value expect y tring all possible msgs.

4. a small change to a msg should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value.

5.it is infeasible to find two different msgs with the same hash value.

Blockchain Hashing

Add 2 blocks with hashing blue + yellow = green

Green + orange = purple ( so we cant change green as it has info of previous 2) that’s why its so secure we cant just break into one sys. We have to break into all the sys it is connected to

Solidity

Language to write smart contracts

IDE : <https://remix.ethereum.org/>

Add Solidity Hello World Code

Now add the following code to the file:

// SPDX-License-Identifier: GPL-3.0

pragma solidity ^0.8.1;

contract MyContract {

string public myString = 'hello world';

}

Smart Contracts

A piece of code running on blockchain

-it’s a state machine

-needs transactions to change state

- can do logic operations

State change happens through mining + transactions.

Understanding Blockchain Nodes

Eth protocol

SOLIDITY TYPES CHEAT SHEET

## BOOLEANS

bool: The possible values are constants true and false. Operators:

! (logical negation)

&& (logical conjunction, “and”)

|| (logical disjunction, “or”)

== (equality)

!= (inequality)

The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y),

if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-eﬀects.

## EXPONENTIATION

Exponentiation is only available for unsigned types. Please take care that the types you are using are large enough to hold the result and prepare

for potential wrapping behaviour.

## DIVISION

Since the type of the result of an operation is always the type of one of the operands,

division on integers always results in an integer. In Solidity, division rounds towards zero.

This mean that int256(-5) / int256(2) == int256(-2).

Note that in contrast, division on literals results in fractional values of arbitrary precision.

## ADDITION, SUBTRACTION MULTIPLCATION

Addition, subtraction and multiplication have the usual semantics.

They wrap in two’s complement representation meaning that for example

uint256(0) - uint256(1) == 2\*\*256 - 1.

You have to take these overﬂows into account when designing safe smart contracts.

The expression -x is equivalent to (T(0) - x) where T is the type of x.

This means that -x will not be negative if the type of x is an unsigned integer type.

Also, -x can be positive if x is negative.

There is another caveat also resulting from two’s complement representation:

int x = -2\*\*255; assert(-x == x);

## INTEGERS

int / uint : Signed and unsigned integers of various sizes.

Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256.

uint and int are aliases for uint256 and int256, respectively.

Operators:

Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)

Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation) Shift operators: << (left shift), >> (right shift)

Arithmetic operators: +, -, unary -, \*, /, % (modulo), \*\* (exponentiation)

## MODULO

The modulo operation a % n yields the remainder r after the division of the operand a by the operand n,

where q = int(a / n) and r = a - (n \* q). This means that modulo results in the same

sign as its left operand (or zero) and a % n == -(-a % n) holds for negative a:

• int256(5) % int256(2) == int256(1) • int256(5) % int256(-2) == int256(1)

• int256(-5) % int256(2) == int256(-1) • int256(-5) % int256(-2) == int256(-1)

# FIXED-SIZE BYTE ARRAYS

**The value types bytes1, bytes2, bytes3, …, bytes32 hold a sequence of bytes from one to up to 32. byte is an alias for bytes1.**

**Operators:**

**Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)**

**Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation) Shift operators: << (left shift), >> (right shift)**

**Index access: If x is of type bytes**I**, then x[k] for** 0 **<= k <** I **returns the k th byte**

**(read-only).**

**The shifting operator works with any integer type as right operand (but returns the type of the left operand),**

**which denotes the number of bits to shift by. Shifting by a negative amount causes a runtime exception.**

**Members:**

**.length yields the ﬁxed length of the byte array (read-only).**

**DYNAMICALLY-SIZED BYTE ARRAY**

bytes:

Dynamically-sized byte array, see Arrays. Not a value-type!

string:

Dynamically-sized UTF-8-encoded string, see Arrays. Not a value-type!

**SHIFTS**

The result of a shift operation

has the type of the left operand, truncating the result to match the type.

For positive and negative x values, x << y is equivalent to x \* 2\*\*y. For positive x values, x >> y is equivalent to x / 2\*\*y.

For negative x values, x >> y is equivalent to (x + 1) / 2\*\*y - 1

(which is the same as dividing x by 2\*\*y while rounding down towards negative inﬁnity).

In all cases, shifting by a negative y throws a runtime exception.

## ADDRESS

The address type comes in two ﬂavours, which are largely identical: address: Holds a 20 byte value (size of an Ethereum address). address payable: Same as address, but with the additional members transfer and send.

The idea behind this distinction is that address payable is an address you can send Ether to, while a plain address cannot be sent Ether.

Type conversions:

Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable are not possible (the only way to perform such a conversion is by using an

intermediate conversion to uint160).

Address literals can be implicitly converted to address payable.

Explicit conversions to and from address are allowed for integers, integer literals, bytes20 and contract types with the following caveat:

Conversions of the form address payable(x) are not allowed. Instead the result of a conversion of the form address(x) has the type address payable, if x is of integer or ﬁxed bytes type, a literal or a contract with a payable fallback function. If x is a contract without payable fallback function,

then address(x) will be of type address. In external function signatures address is used for both the address and the address payable type.

Operators:

<=, <, ==, !=, >= and >

## MEMBERS OF ADDRESS

For a quick reference of all members of address, see Members of Address Types.

balance and transfer

it is possible to query the balance of an address using the property balance and to send Ether (in units of wei) to a payable address using the transfer function:

address payable x = address(0x123); address myAddress = address(this);

if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);

The transfer function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account.

The transfer function reverts on failure

UNITS AND GLOBALLY AVAILABLE VARIABLES

# ETHER UNITS

A literal number can take a suﬃx of wei, ﬁnney, szabo or ether to specify a subdenomination of Ether,

where Ether numbers without a postﬁvx are assumed to be Wei.

assert(1 wei == 1); assert(1 szabo == 1e12); assert(1 ﬁnney == 1e15); assert(1 ether == 1e18);

The only eﬀect of the subdenomination suﬃx is a multiplication by a power of ten.

# TIME UNITS

Suﬃxes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time

where seconds are the base unit and units are considered naively in the following way:

1 == 1 seconds

1 minutes == 60 seconds

1 hours == 60 minutes

1 days == 24 hours

1 weeks == 7 days

Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even

every day has 24 hours because of leap seconds.

Due to the fact that leap seconds cannot be predicted,

an exact calendar library has to be updated by an external oracle.

These suﬃxes cannot be applied to variables. For example, if you want to interpret a function parameter in days, you can in the following way:

function f(uint start, uint daysAfter) public { if (now >= start + daysAfter \* 1 days) {

// ...

}

}

# BLOCK AND TRANSACTION PROPERTIES

blockhash(uint blockNumber) returns (bytes32): hash of the

given block - only works for 256 most recent, excluding current, blocks block.coinbase (address payable): current block miner’s address block.diﬃculty (uint): current block diﬃculty

block.gaslimit (uint): current block gaslimit block.number (uint): current block number

block.timestamp (uint): current block timestamp as seconds since unix epoch gasleft() returns (uint256): remaining gas

msg.data (bytes calldata): complete calldata

msg.sender (address payable): sender of the message (current call) msg.sig (bytes4): ﬁrst four bytes of the calldata (i.e. function identiﬁer) msg.value (uint): number of wei sent with the message

now (uint): current block timestamp (alias for block.timestamp) tx.gasprice (uint): gas price of the transaction

tx.origin (address payable): sender of the transaction (full call chain)

# ABI ENCODING AND DECODING FUNCTIONS

abi.decode(bytes memory encodedData, (...)) returns (...): ABI-decodes the given data, while the types are given in parentheses as second argument.

Example: (uint a, uint[2] memory b, bytes memory c) = abi.decode(data, (uint, uint[2], bytes)) abi.encode(...) returns (bytes memory): ABI-encodes the given arguments abi.encodePacked(...) returns (bytes memory): Performs packed encoding of the given arguments. Note that packed encoding can be ambiguous!

abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory): ABI-encodes the given arguments starting from the second and prepends the given four-byte selector abi.encodeWithSignature(string memory signature, ...) returns (bytes memory):

Equivalent to abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), ...)`

# MEMBERS OF ADDRESS TYPES

<address>.balance (uint256):

balance of the Address in Wei

<address payable>.transfer(uint256 amount):

send given amount of Wei to Address, reverts on failure, forwards 2300 gas stipend, not adjustable

<address payable>.send(uint256 amount) returns (bool):

send given amount of Wei to Address, returns false on failure, forwards 2300 gas stipend, not adjustable

<address>.call(bytes memory) returns (bool, bytes memory):

issue low-level CALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

<address>.delegatecall(bytes memory) returns (bool, bytes memory):

issue low-level DELEGATECALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

<address>.staticcall(bytes memory) returns (bool, bytes memory):

issue low-level STATICCALL with the given payload, returns success condition and return data, forwards all available gas, adjustable