Blockchains & Distributed Ledgers

Lecture 03

Dimitris Karakostas

Slide credits: DK, Aggelos Kiayias, Aydin Abadi, Christos Nasikas, Dionysis Zindros

Contracts

“A contract is a legally binding agreement that defines and governs the rights and duties between or among its parties.”

Contracts

“A contract is a legally binding agreement that defines and governs the rights and duties between or among its parties.”

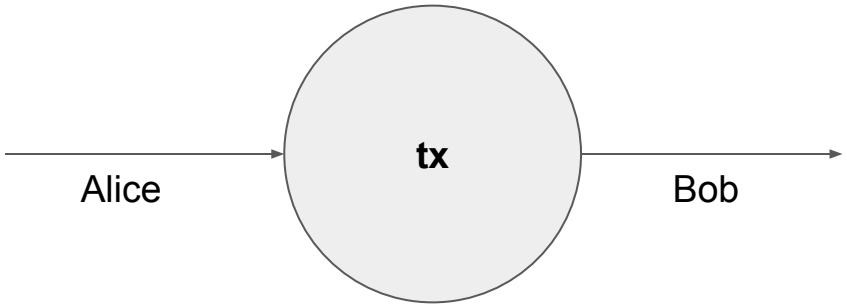
“smart contracts are neither smart nor contracts”

What is a smart contract?

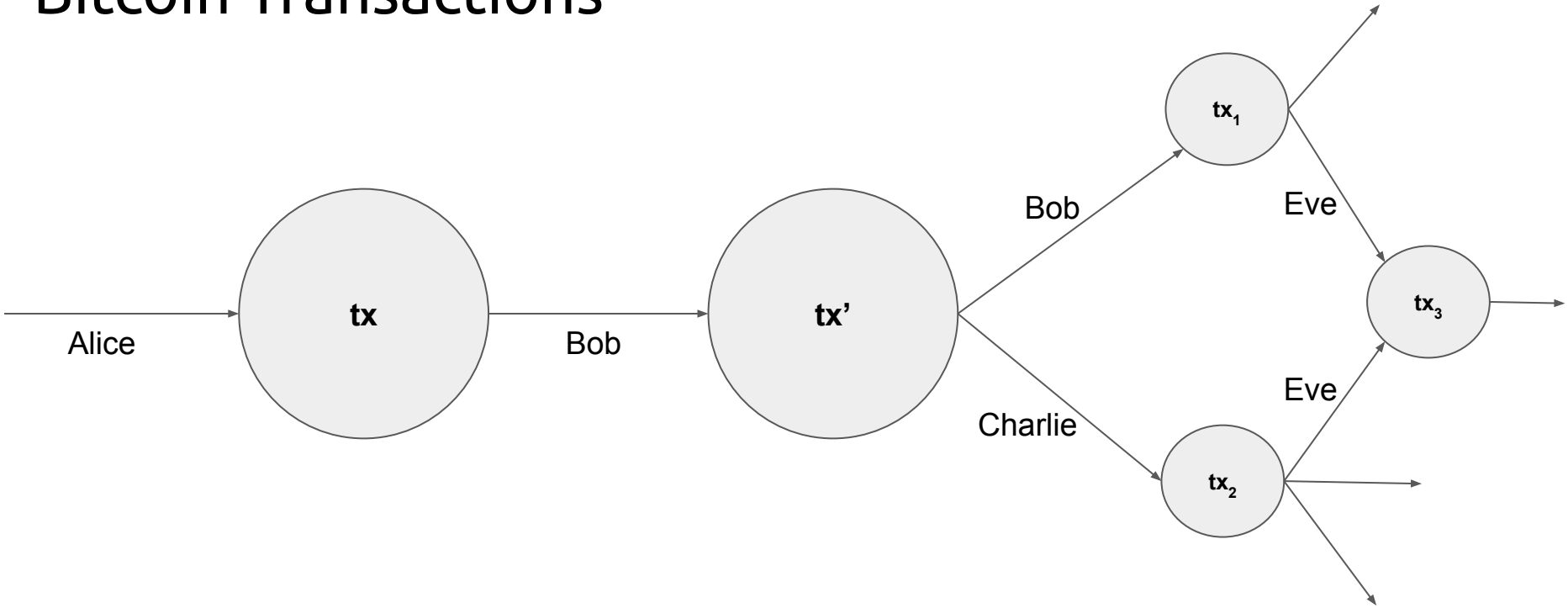
- Computer programs
- Contract code is executed by all full nodes
- The outcome of a smart contract is the same for everyone
- Context:
  - Internal storage
  - Transaction context
  - Most recent blocks
- The code of a smart contract cannot change

Bitcoin

Bitcoin Transactions



Bitcoin Transactions



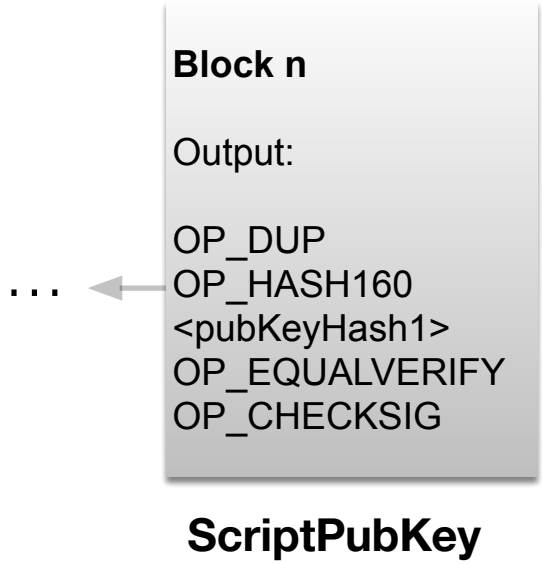
Bitcoin programs

- **Transaction:** a transfer of value in the Bitcoin network
- Each transaction consists of the following main fields:
  - **input**: a transaction output from which it spends bitcoins:
    i. previous transaction address
    ii. index
    iii. ScriptSig
  - **output**: instructions for spending the sent bitcoins:
    i. value: amount of bitcoins to send
    ii. ScriptPubKey: instructions on how to spend the sent bitcoins
- To validate a transaction:
  - concatenate ScriptSig of current tx with ScriptPubKey of referenced tx
  - check if it successfully compiles with no errors

Bitcoin Script

- Stack-based
- Notation: Data in the script is enclosed in <> (<sig>, <pubKey>, etc)
- Opcodes: commands or functions
  - Arithmetic, e.g. OP_ABS, OP_ADD
  - Stack, e.g. OP_DROP, OP_SWAP
  - Flow control, e.g. OP_IF, OP_ELSE
  - Bitwise logic, e.g. OP_EQUAL, OP_EQUALVERIFY
  - Hashing, e.g. OP_SHA1, OP_SHA256
  - (Multiple) Signature Verification, e.g. OP_CHECKSIG, OP_CHECKMULTISIG
  - Locktime, e.g. OP_CHECKLOCKTIMEVERIFY, OP_CHECKSEQUENCEVERIFY

Bitcoin Unspent Transaction Output (UTxO) example



Bitcoin Script example

Stack	Script	Description
Empty	<sig1> <pubKey1> OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Add constant values from left to right to the stack until we reach an opcode.
<sig1> <pubKey1>	OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Duplicate top stack item
<sig1> <pubKey1> <pubKey1>	OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Hash at the top of the stack
<sig1> <pubKey1> <pub1Hash>	<pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Push the hashvalue to the stack
<sig1><pubKey1> <pub1Hash><pubKeyHash1>	OP_EQUALVERIFY OP_CHECKSIG	Check if top two items are equal
<sig1> <pubKey1>	OP_CHECKSIG	Verify the signature.
Empty	TRUE	If stack empty return True, else return False.

Bitcoin’s scripting language limitations

- Lack of Turing-completeness: No loops
- Lack of state: Cannot keep internal state.
- Value-blindness: Cannot denominate the amount being sent
- Blockchain-blindness: Cannot access block header values such as nonce, timestamp and previous hash block.

Extending Bitcoin functionality: add new opcodes

- Building a protocol on top of Bitcoin:
  - Pros:
    - Take advantage of the underlying network and mining power.
    - Very low development cost
  - Cons:
    - No flexibility.
- Build an independent network:
  - Pros:
    - Easy to add and extend new opcodes.
    - Flexibility.
  - Cons:
    - Need to attract miners to sustain the network.
    - Difficult to implement.

Ethereum

Same principles as Bitcoin

- **A peer-to-peer network:** connects the participants
- **Sybil resistance**: Proof-of-Stake (former Proof-of-Work)
- **A digital currency:** ether
- **A global ledger**: the blockchain
  - Addresses: key pair
  - Wallets
  - Transactions: digital signatures
  - Blocks

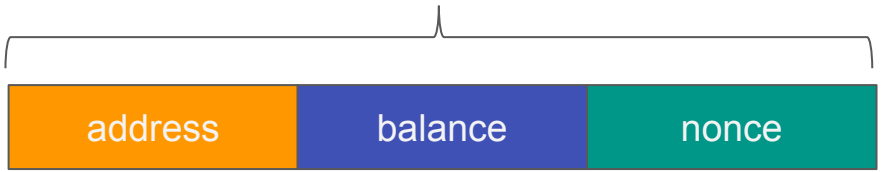
Ethereum: A universal Replicated State Machine

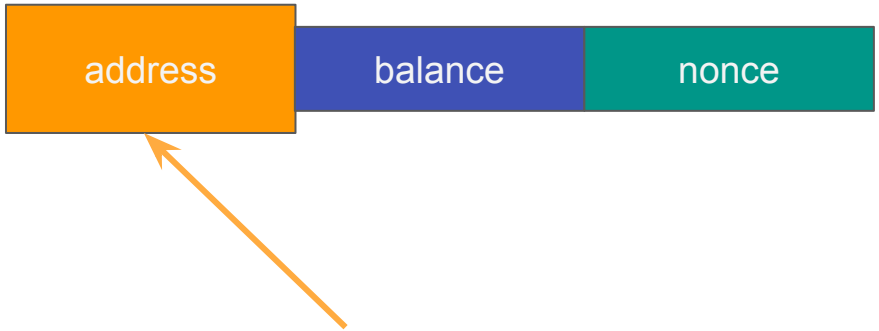
- Transaction-based deterministic state machine
  - Global state (singleton)
  - A virtual machine that applies changes to global state
- A global decentralized computing infrastructure
- Anyone can create their own state transition functions
- Stack-based bytecode language
- Turing-completeness
- Smart contracts
- Decentralized applications

Ethereum accounts

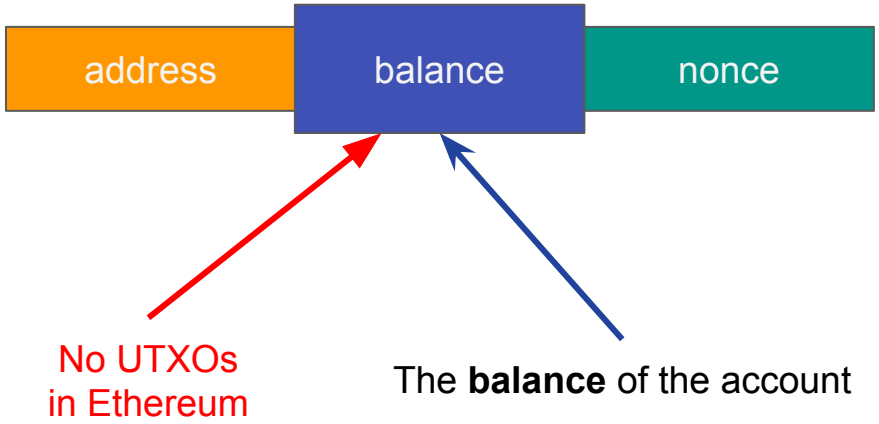
- Global state of Ethereum: **accounts**
- They **interact** to each other **through transactions** (or messages)
- A **state** and a 20-byte **address** (160-bit identifier) associated with each account

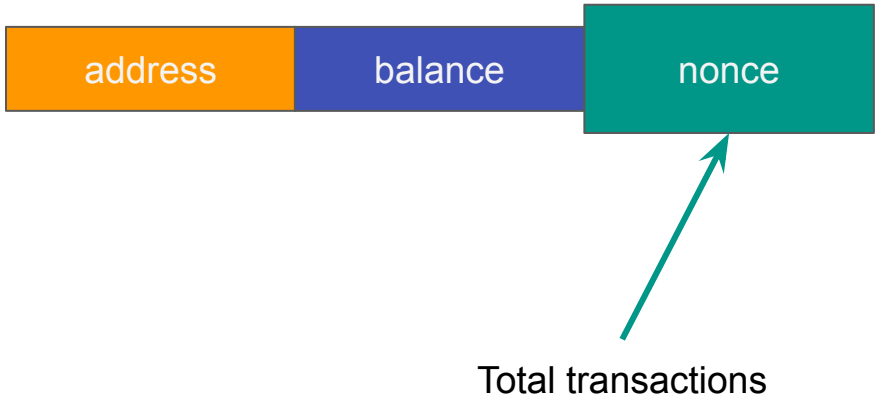
Ethereum account





The **address** of the account


address
balance
nonce
No UTXOs
in Ethereum
The **balance** of the account


address
balance
nonce
Total transactions

UTxO vs Accounts

- UTxO pros:
  - Unlinkability → Higher degree of privacy
  - Scalability (parallelism, sharding)
- Account pros:
  - Space saving
  - Conceptual simplicity

Two types of accounts

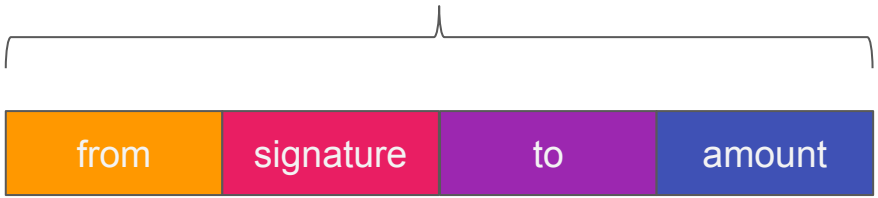
- Personal accounts (what we’ve seen)
- **Contract accounts**

Ethereum contract account

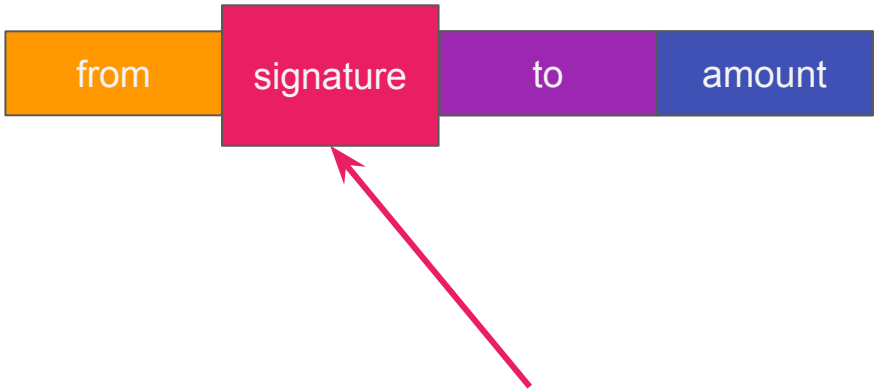
Ethereum accounts

	Personal account	Contract account
address	H(pub_key)	H(addr + nonce of creator)
code	∅	Code to be executed
storage	∅	Data of the contract
balance	ETH balance (in Wei)	ETH balance (in Wei)
nonce	# transaction sent	# transaction sent

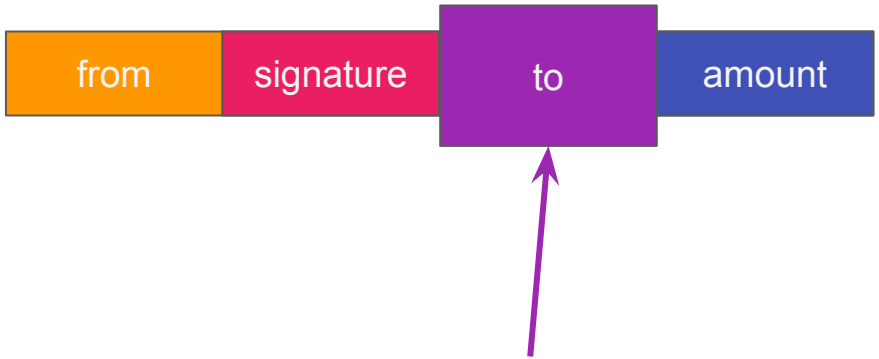
Ethereum transaction



The **sender** of the transaction

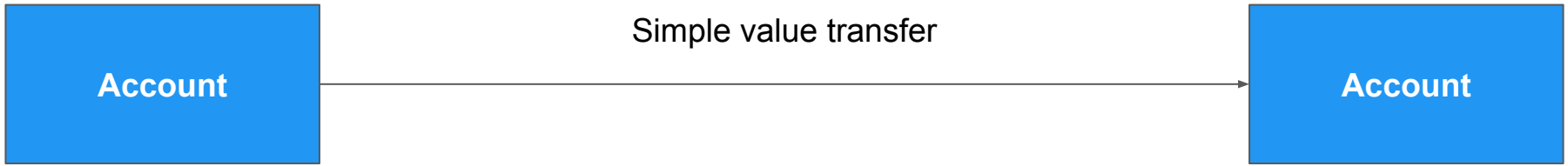


Digital signature on the **new transaction** created by **the sender's private key**

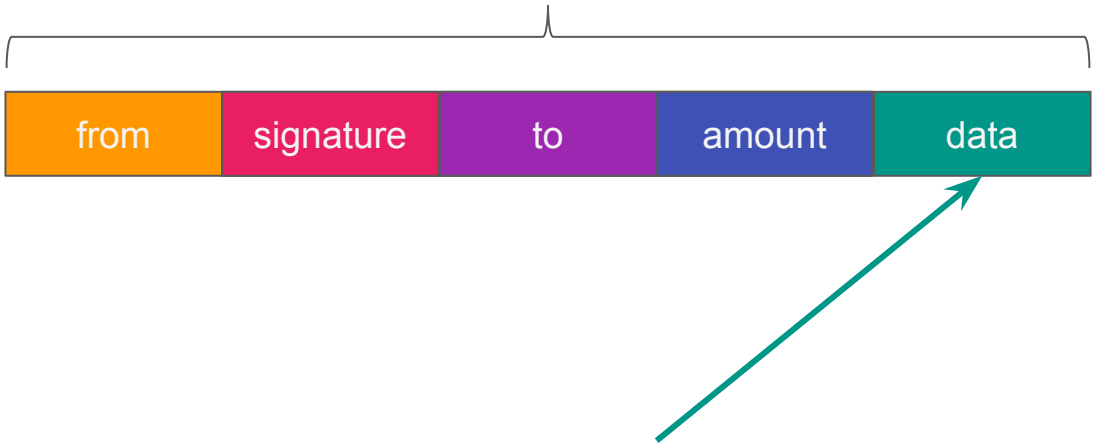


Receiver of the transaction

Amount transferred by transaction (in Wei)


Account
Simple value transfer
Account

a transaction about a contract



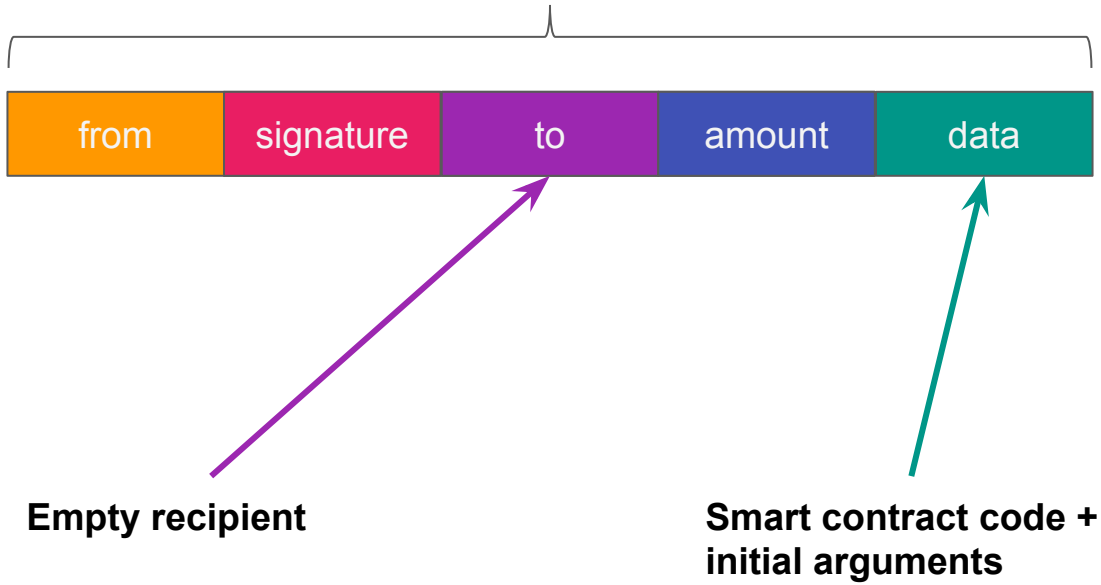
Transaction **about personal accounts**:
Field is unused

Transaction **about contracts**:
Will contain **data about the contract**

Smart contract lifecycle

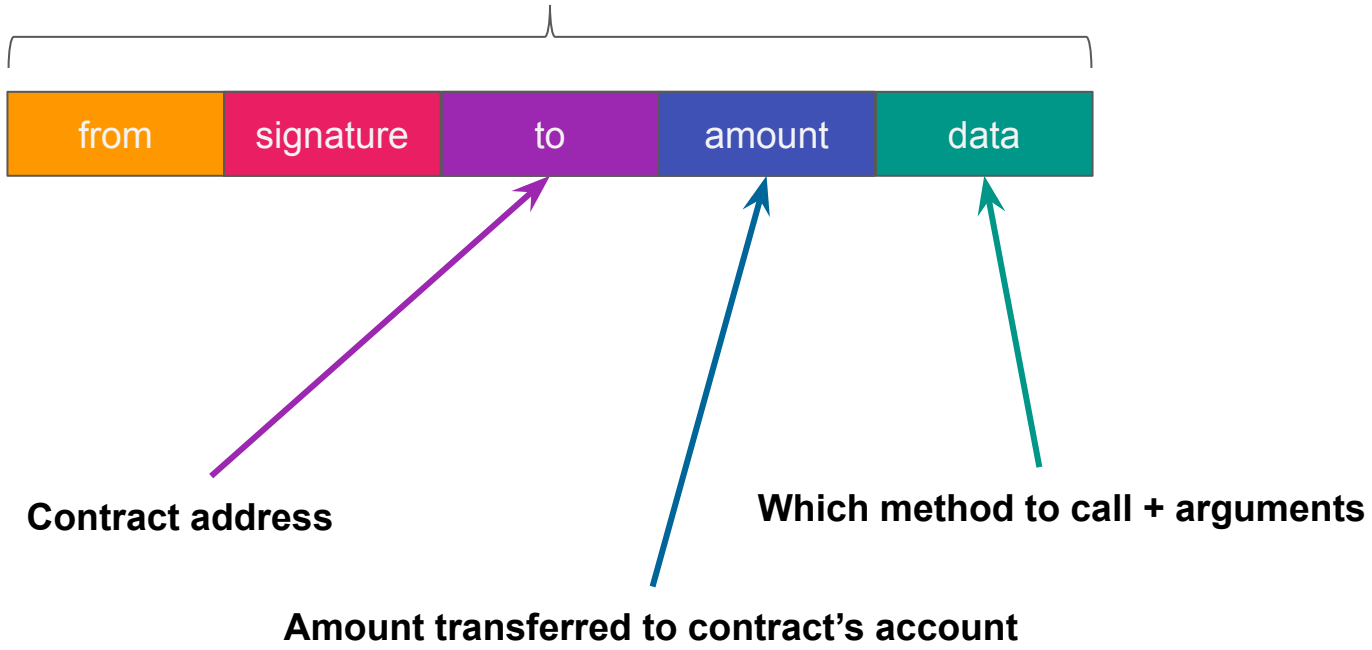

Create
Interact
Destroy

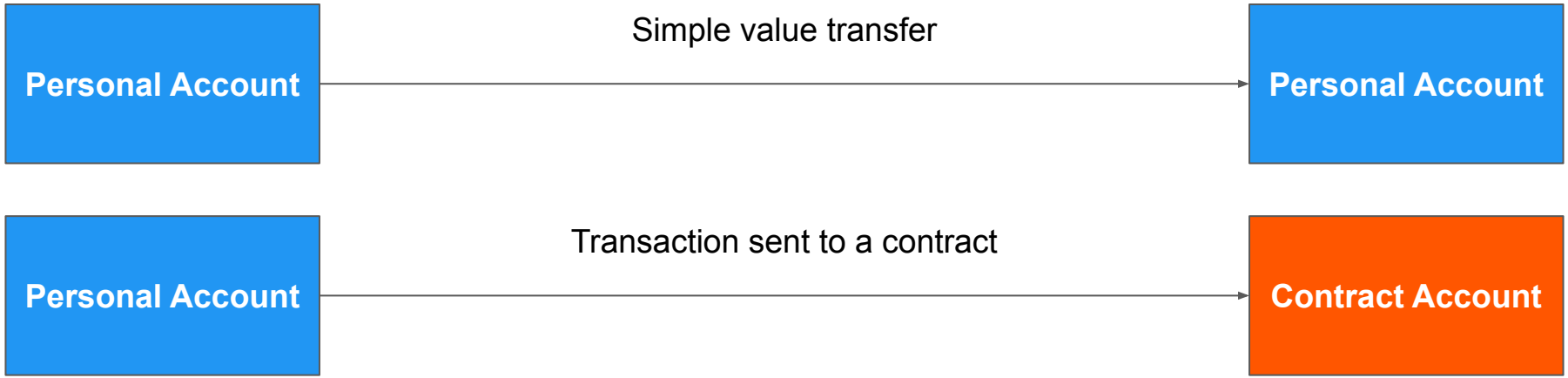
Transaction for contract creation




Create
Interact
Destroy

Transaction for contract interaction



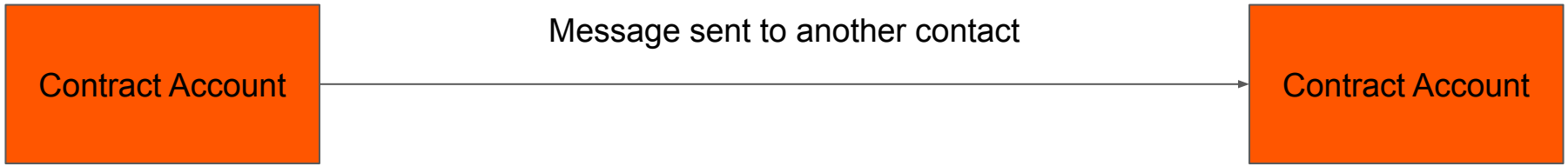

Personal Account
Simple value transfer
Personal Account
Personal Account
Transaction sent to a contract
Contract Account

Contract method call

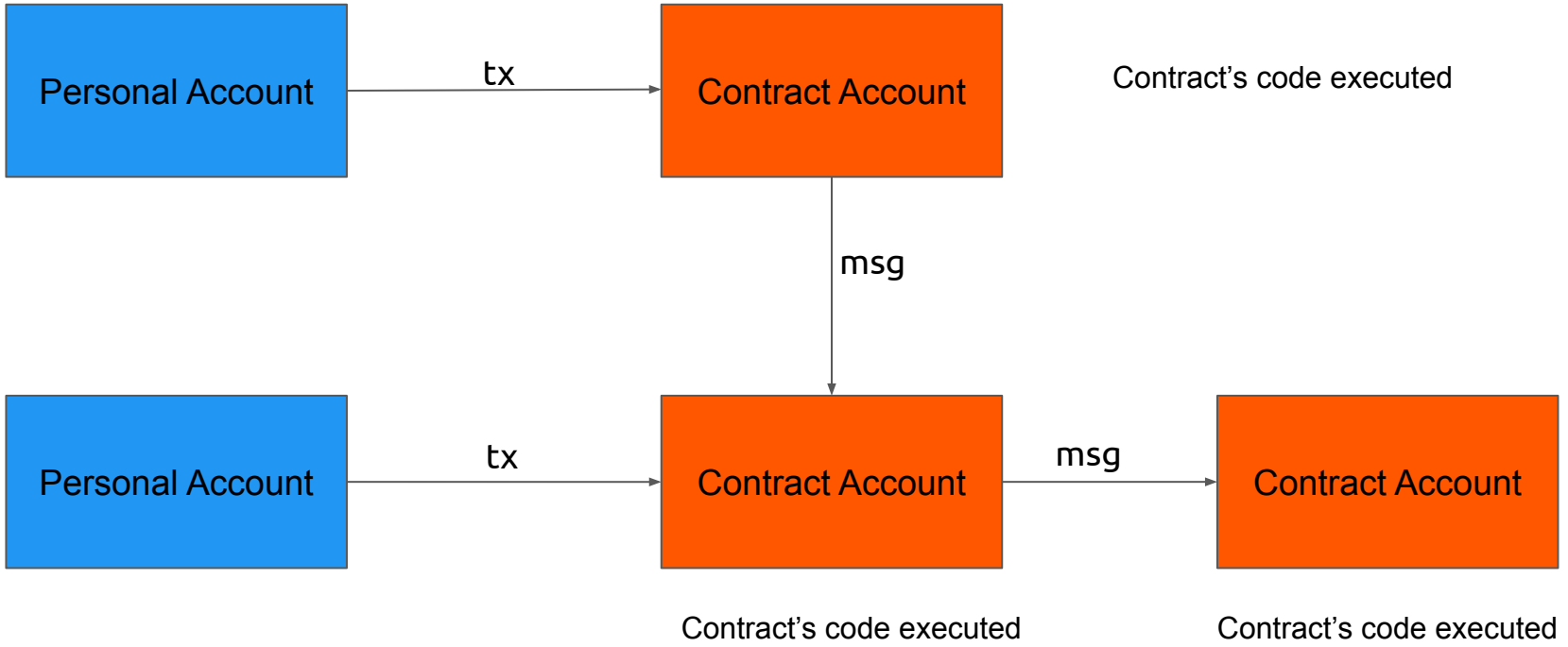
- When contract account is activated:
  a. Contract **code** runs
  b. It can read/write to **internal storage**
  c. It can **send other transactions** or **call other contracts**
- Can’t initiate new transactions on their own
- Can only fire transactions in response to other transactions received

Messages

- Like a **transaction** except it is **produced by a contract**
- Virtual objects
- Exist **only** in the **Ethereum execution environment**
- A message leads to the recipient account running its code
- **Contracts** can have **relationships** with **other contracts**


Contract Account
Message sent to another contact
Contract Account

Transactions & messages

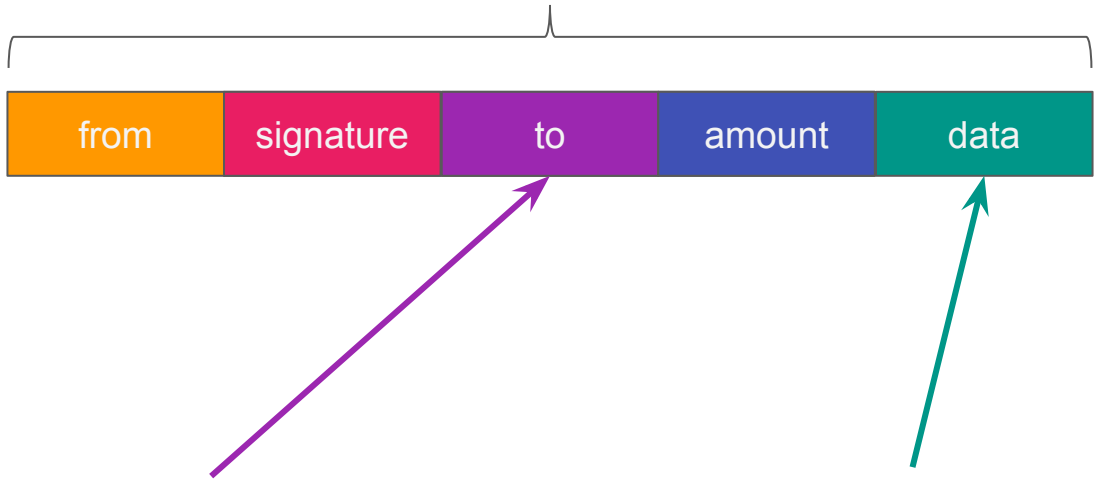


Types of transactions

	send	create	call
from	sender	creator	caller
signature	sig	sig	sig
to	receiver	∅	contract
amount	ETH	ETH	ETH
data	∅	code	f, args


Create
Interact
Destroy

a transaction for contract destruction



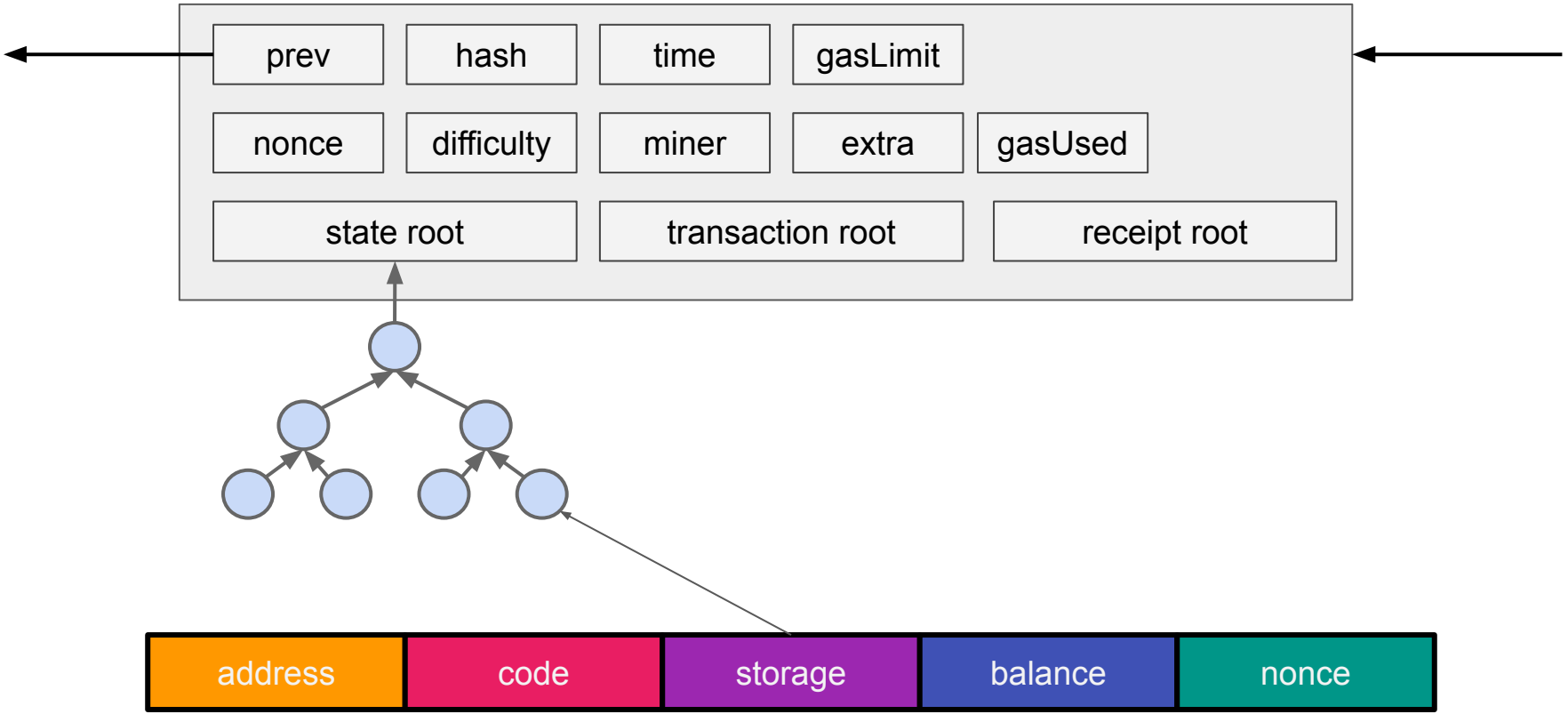
Ethereum Virtual Machine

- Series of **bytecode** instructions (EVM code)
- Each **bytecode** represents an **operation** (opcode)
- A quasi **Turing complete** machine
- **Stack-based** architecture (1024-depth)
- **32-byte** words (256-bit words)
- **Crypto** primitives

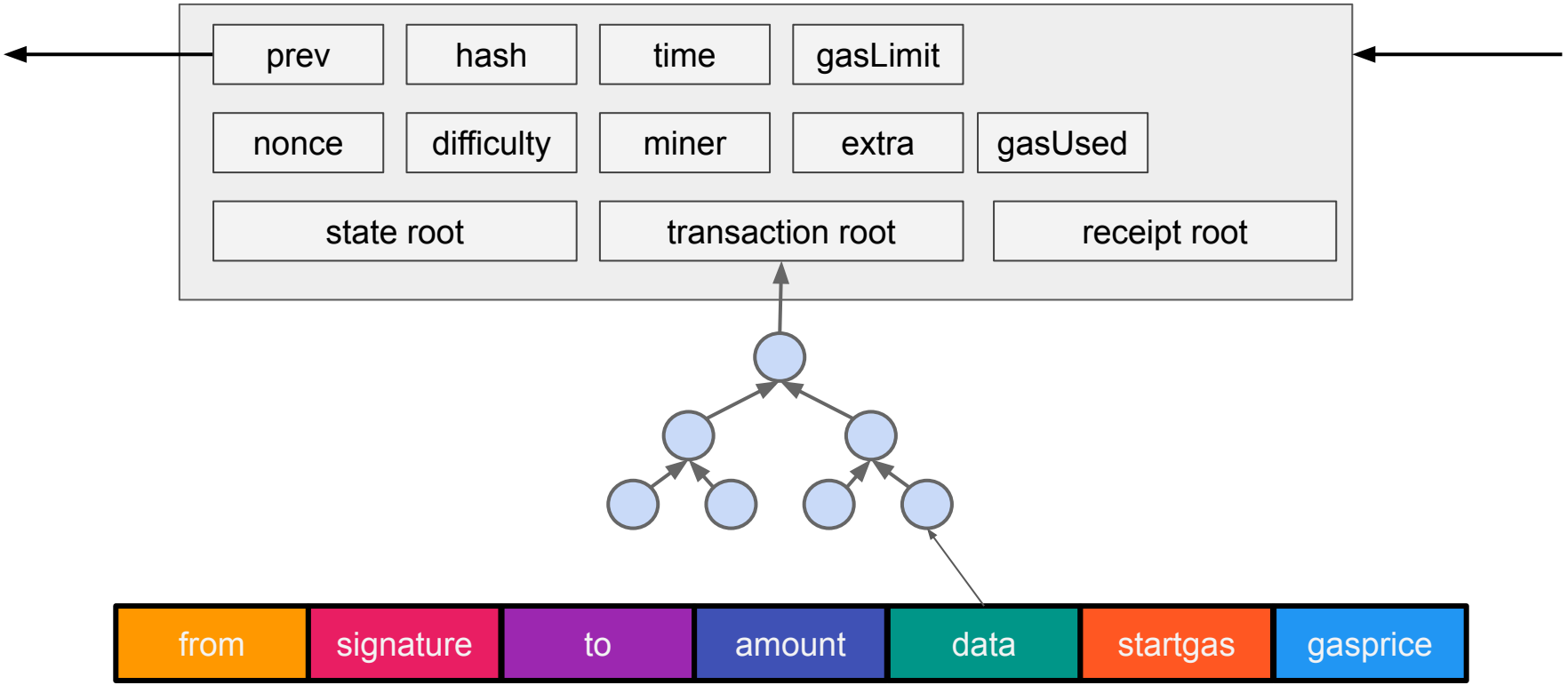
EVM: contract execution

- Three types of storage:
  - **Stack**
  - **Memory** (expandable byte array)
  - **Storage** (key/value store)
- All memory is **zero-initialized**
- Access:
  - **value**
  - **sender**
  - **data**
  - **gas** limit
  - **block header** data (depth, timestamp, miner, block id, ...)

Ethereum block



Ethereum block



Ethereum Mining

- **Blocks** contain: **transaction** list and most **recent state**
- Block **time**: ~12-15 **seconds**
- *(Since 2022)* Proof-of-stake (Gasper)
  - Previously **Proof-of-work**: Ethash (originally designed to be **memory-hard**)
- **Winner** of the block:
  - Previously: 2 ETH + tx fees
  - Now: a bit more complex

Transaction fees: the phone booth model

Gas: a necessary evil

- Every node on the network:
  - evaluates all **transactions**
  - stores all **state**

Gas: a necessary evil

- Every node on the network:
  - evaluates all **transactions**
  - stores all **state**
- The *halting problem*:
  - Miners cannot determine if a program can/will finish

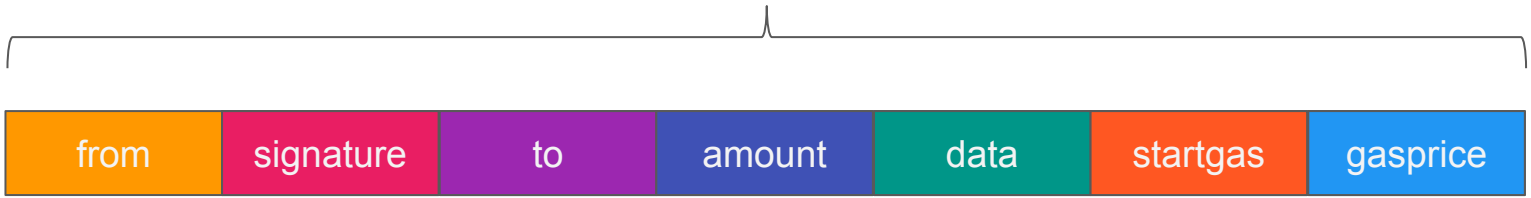
Gas: a necessary evil

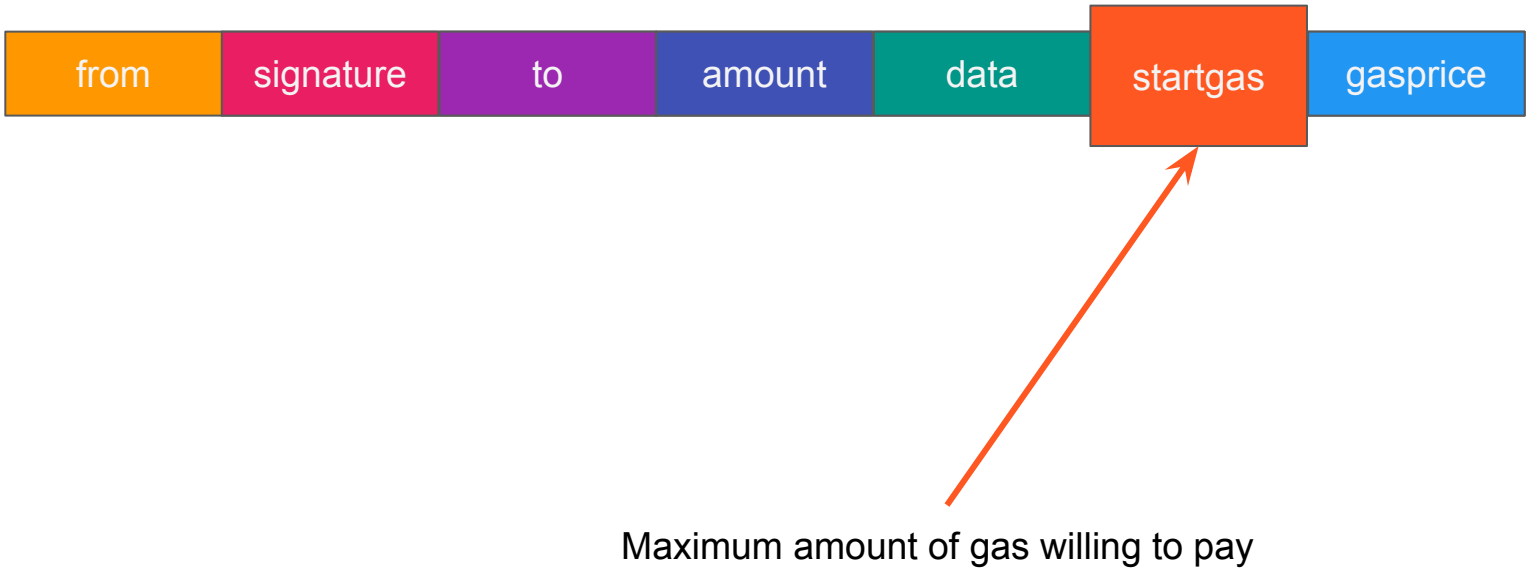
- Every node on the network:
  - evaluates all **transactions**
  - stores all **state**
- The *halting problem*:
  - Miners cannot determine if a program can/will finish

Solution

- Every **computation step** has a **fee**
- Fee is **paid** in **gas**
- **Gas** is the **unit** used to **measure computations**

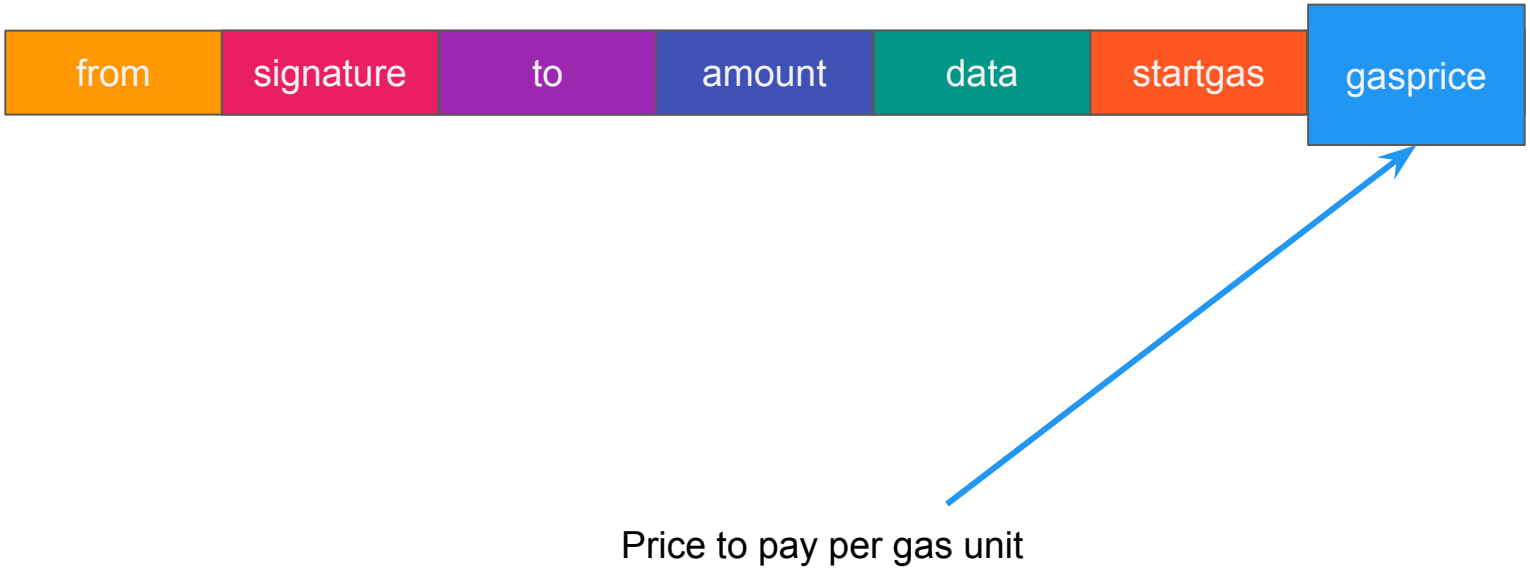
Ethereum transaction




from
signature
to
amount
data
startgas
gasprice
Maximum amount of gas willing to pay

Gas Limit

- Equals to startgas
- All **unused gas** is **refunded** at the end of a transaction
- **Out of gas** transactions are **not refundable**
- Blocks also have a **gas limit**


from
signature
to
amount
data
startgas
gasprice
Price to pay per gas unit

Gas Price

- Measured in **gwei** (10^9 Wei)
- Determines how **quickly** a transaction will be **mined**
  - Higher gas price makes transaction more appealing to miners

Transaction Fees

Confirmation vs. Gas price

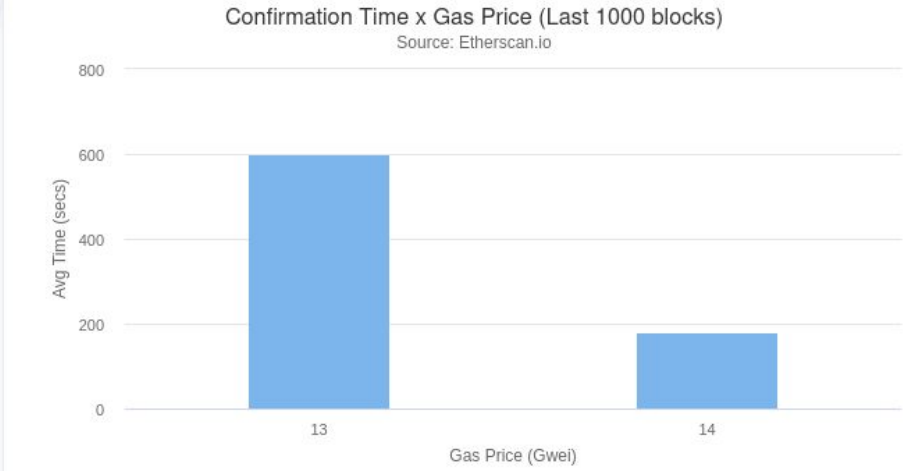
Next update in 10s

Wed, 28 Sep 2022 17:20:57 UTC

Low	Average	High
13 gwei	14 gwei	14 gwei
Base: 13 | Priority: 0	Base: 13 | Priority: 1	Base: 13 | Priority: 1
$0.39 | ~ 10 mins: 0 secs	$0.42 | ~ 3 mins: 0 secs	$0.48 | ~ 30 secs

Estimated Cost of Transaction Actions: View API

Action	Low	Average	High
OpenSea: Sale	$1.34	$1.43	$1.62
Uniswap V3: Swap	$3.45	$3.69	$4.18
USDT: Transfer	$1.01	$1.08	$1.23



https://etherscan.io/gastracker

Confirmation vs. Gas price

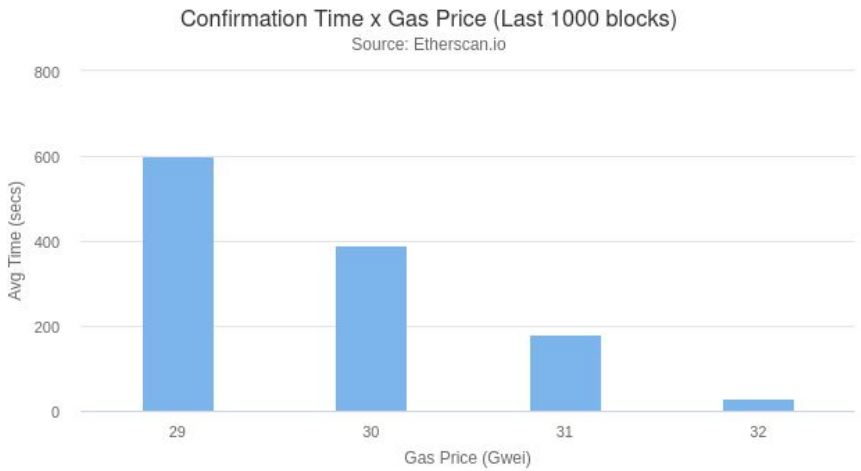
Next update in 2s

Wed, 28 Sep 2022 17:36:08 UTC

Low	Average	High
29 gwei	31 gwei	32 gwei
Base: 29 | Priority: 0	Base: 29 | Priority: 2	Base: 29 | Priority: 3
$0.81 | ~ 10 mins: 0 secs	$0.87 | ~ 3 mins: 0 secs	$0.90 | ~ 30 secs

Estimated Cost of Transaction Actions: View API

Action	Low	Average	High
OpenSea: Sale	$2.77	$2.96	$3.06
Uniswap V3: Swap	$7.14	$7.64	$7.88
USDT: Transfer	$2.10	$2.24	$2.31



https://etherscan.io/gastracker

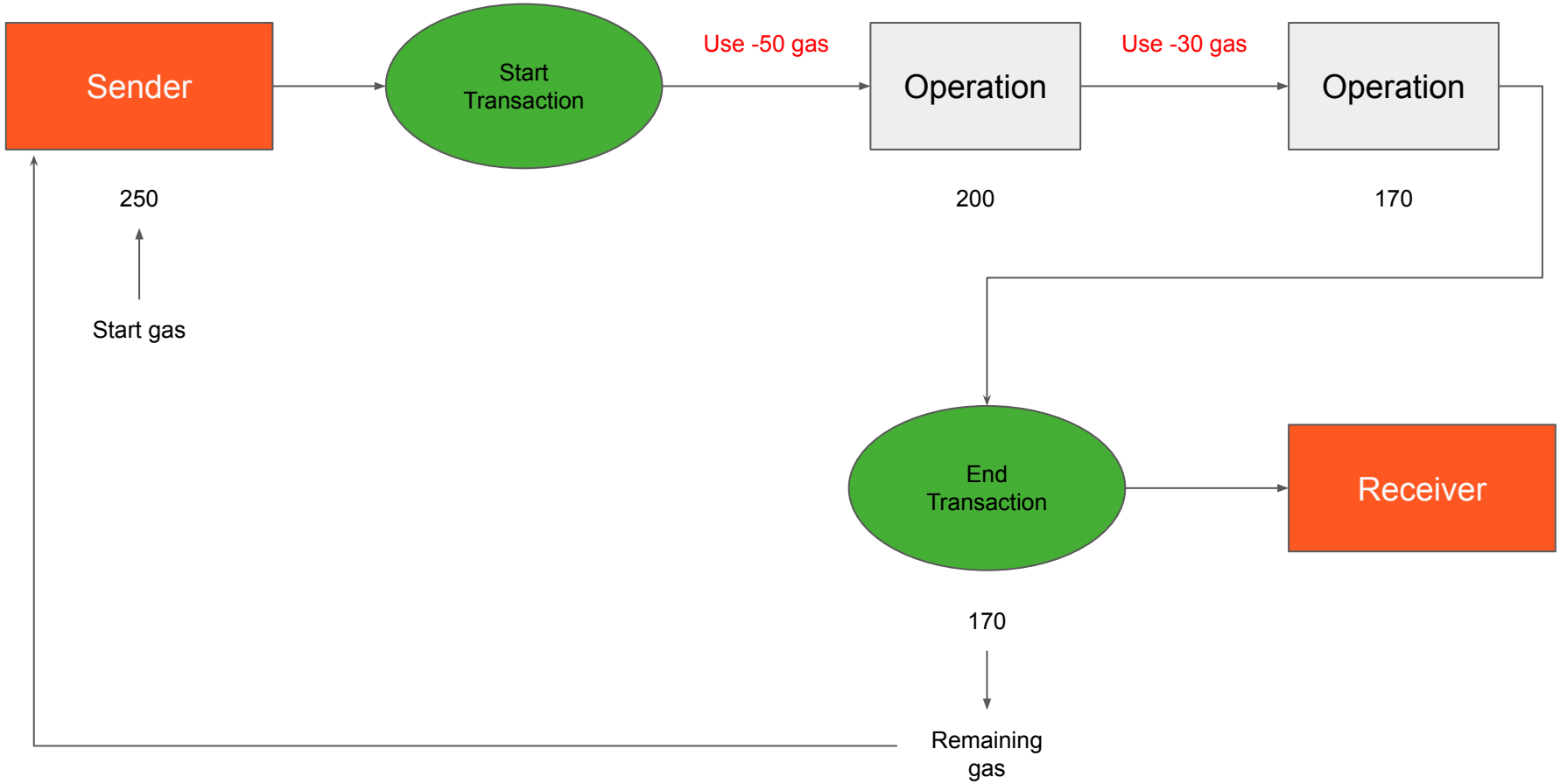
Storage in Ethereum

ETH Price: $1,300 (28 September, 2022) - Gas Price: 16 Gwei

Size	Gas	Cost (ETH)	Cost ($)
1KB	677,000	~0.021	$27
1MB	~693,000,000	21.33	$27,700
10MB	~7,000,000,000	~215	$280,000

Computation steps

1. If **gas_limit** * **gas_price** > **balance** then **halt**
2. **Deduct** gas_limit * gas_price from **balance**
3. Set gas = gas_limit
4. **Run code** deducting from gas
5. After termination **return remaining gas** to **balance**


Sender
Start Transaction
Use -50 gas
Operation
Use -30 gas
Operation
250
200
170
Start gas
End Transaction
Receiver
170
Remaining gas

Out of gas exceptions

- State **reverts** to **previous state**
- gas_limit * gas_price is **still deducted** from **balance**

Introduction to Solidity

Solidity

- A **high level programming** language for **writing** smart **contracts** on **Ethereum**
- **Compile** code for the **Ethereum Virtual Machine**
- **Syntax** similar to **JavaScript**

Documentation: docs.soliditylang.org

Solidity

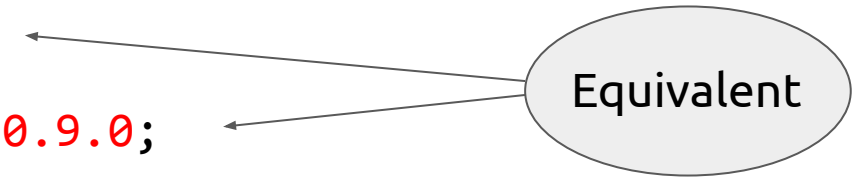
- **Contracts** look like **classes / objects**
- **Statically**-typed language (variable types must be set explicitly)
- Most of the control structures from **JavaScript** are available in **Solidity** (conditions, loops, exception handling, etc.)

HelloWorld contract

```
pragma solidity >=0.7.0 <0.9.0;

contract HelloWorld {

    function print () public pure returns (string memory) {

        return 'Hello World!';

    }

}
```

Pragmas

```
pragma solidity 0.8.0;

pragma solidity ^0.8.1;

pragma solidity >=0.8.1 < 0.9.0;
```



The pragma keyword is used to enable certain compiler (version) features or checks. Follows the same syntax used by npm.

Contract

```
contract <ContractName> { ... }
```

Constructors

```
contract HelloWorld1 {
    constructor () { … }
}


contract HelloWorld2 {
    constructor (uint x, string y) { … }
}
```

Solidity: Variables

- State variables:
  - Contract variables
  - **Permanently stored** in contract **storage**
  - **Must declare** at compilation time

- Local variables
  - Within a **function**: **cannot** be **accessed** outside
  - **Complex** types: at **storage** by default
  - **Value** types: in the **stack**
  - Function **arguments**

Types

- The **type** of each variable **needs to be specified** (Solidity is a statically typed language)
- **Two** categories:
  - **Value** types
  - **Reference** types
- “**undefined**” or “**null**” values **do not exist** in Solidity
- **Variables** without a value **always** have a **default value** (zero-state) dependent on their type.
- Solidity follows the scoping rules of C99 (variables are visible until the end of the smallest `{}`-block)

Value types

Types: booleans

```
contract Booleans {

    bool p = true;

    bool q = false;

}
```

Operators: `!`, `&&`, `||`, `!=`, `==`

Types: integers

```
contract Integers {

    uint256 x = 5;

    int8 y = -5;

}
```

- Two types:
  - `int` (signed)
  - `uint` (unsigned)
- Keywords: `uint8` / `int8` to `uint256` / `int256` in step of 8.
- `uint` / `int` are alias for `uint256` / `int256`.
- Operators as usual:
  - Comparisons: `<=, <, ==, !=, >=, >`
  - Arithmetic operators: `+, -, *, /, %, **`
  - Bitwise operators: `&, |, ^`
  - Shift operators: `>>, <<`
- Range: $2^b - 1$ where $b \in \{ 8, 16, 24, 32, \ldots, 256 \}$
- Division always results in an integer and round towards zero (5 / 2 = 2).
- No floats!

Types: address

```
contract Address {

    address owner;

    address payable anotherAddress;

}
```

Address type holds an Ethereum address (20 byte value).
The “payable” keyword enables to send Ether to the address (you cannot send to plain addresses).

Types: fixed-size byte arrays

```
contract ByteArrays {

    bytes32 y = 0xa5b9...;

    // y.length == 32

}
```

- `bytes1`, `bytes2`, `bytes3`, ..., `bytes32`
- `byte` is alias for `byte1`
- `length`: fixed length of the byte array. You cannot change the length of a fixed byte array.

Types: Enum

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

Example Enum

```
pragma solidity ^0.4.24;

contract Enum {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    function getChoice() public view returns (ActionChoices) {
        return choice;
    }
}
```

Reference types

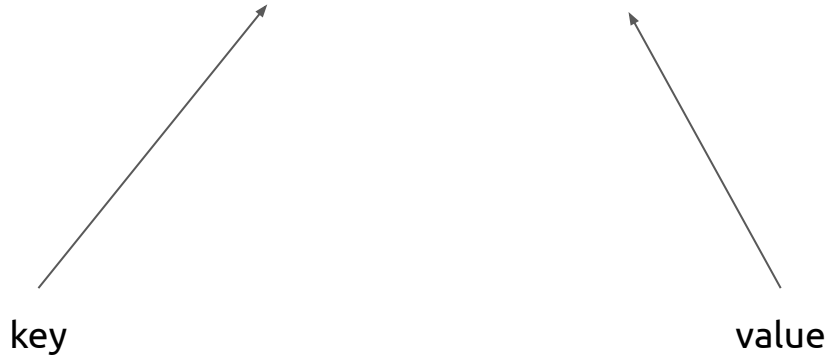
Types: arrays, static and dynamic

```
contract Arrays {
      uint256[2] x;
      uint8[] y;
      bytes z;
      string name;
      // 2D: dynamic rows, 2 columns!
      uint [2][] flags;

      function create () public {
            uint[] memory a = new uint[](7);
            flags.push([0, 1]);
      }
}
```

- The **notation** of declaring **2D** arrays is **reversed** when compared to **other languages**!
  - **Declaration**: `uint[columns][rows] z;`
  - **Access**: `z[row][column]`
- `bytes` and `string` are **special** arrays.
- `bytes` is similar to `byte`[] but is **cheaper** (gas).
- `string` is **UTF-8-encoded**.
- Members:
  - push: push an element at the end of array.
  - length: return or set the size of array.
- `string` does **not** have **length** member.
- **Allocate** memory **arrays** by using the **keyword** `new`. The size of memory arrays has to be known at compilation (in this case 7). You **cannot** resize a memory array.

Types: Mappings

```
contract Mappings {

    mapping(address => uint256) balances;

}
```



Types: Struct

```
contract Vote {
    struct Voter {
        bool voted;
        address voter;
        uint vote;
    }
}
```

- A struct cannot contain a struct of its own type (the size of the struct has to be finite).
- A struct can contain mappings.

Example Structs

```
pragma solidity ^0.4.24;

contract Ballot {
    struct Voter {
        uint weight;
        bool voted;
        address delegate;
        uint vote;
    }
}
```

```
contract CrowdFunding {
    struct Funder {
        address addr;
        uint amount;
    }

    struct Campaign {
        address beneficiary;
        uint fundingGoal;
        uint numFunders;
        uint amount;
        mapping (uint => Funder) funders;
    }
}
```

Visibility

Visibility

- **public**: Public functions can be called from other contracts, internally, and from personal accounts. For public state variables a getter function is automatically created.
- **external**: External functions cannot be called internally. Variables cannot be declared as external.
- **internal**: Internal functions and variables can be called only internally. Contracts that inherit another contract can access the parent’s internal variables and functions.
- **private**: Private functions and variables can be called only by the contract in which they are defined and not by a derived contract.

Solidity: Functions

- Can return multiple values
- Access
  - **Public**: Accessed by **anyone**
  - **Private**: Accessed **only** from the **contract**
  - **Internal**: Accessed **only internally**
  - **External**: Accessed **only externally**
- Declarations
  - **View**: They promise **not** to **modify** the **state**
  - **Pure**: They promise **not** to **read** from or **modify** the **state**.
  - **Payable**: Must be used to **accept Ether**

Remember that on-chain data is public regardless of access declaration!

Data location

Data location: areas

- Every reference type (array, struct, mapping) has a data location.
- Two main data locations: **storage** and **memory**.
- **Calldata**: special location for function's arguments.
- As of Solidity version **5.0.0** you must **always declare** the data **location** of reference types inside functions' body, arguments and returned values.

Data location: areas

- Storage:
  - Persistent
  - All state variables are saved to storage
- Memory:
  - Non-persistent
  - Can be used for function variables or arguments
- Calldata:
  - Non-modifiable (read-only)
  - Function arguments
  - Cheaper than memory
  - Used for dynamic params of an *external* function

Data location: assignment copy/reference rules

- Assignment of the form “variable <- variable”
- Assignment by copy
  - storage <-> memory
  - state (global storage) variable <- state variable, storage and memory
- Assignment by reference
  - memory <-> memory
  - local storage variable <- storage

https://docs.soliditylang.org/en/v0.8.17/types.html#data-location-and-assignment-behaviour

Fallback functions

```
contract Fallback {

    receive() external {

        …

    }

    fallback() external {

        …

    }

}
```

- No arguments (`msg.*` is accessible, contains all data about incoming transaction, incl. sender and value).
- No returned values.
- Mandatory visibility: external.
- `Receive` is executed if no data (transaction field) is supplied. It `is implicitly payable`.
- `Fallback` is executed if the function that a user tries to call does not exist. May or may not be `payable`.
- In the absence of a fallback function a contract cannot receive Ether and an exception is thrown.
- Should be simple - without consuming too much gas.

Solidity: events

- EVM logging mechanism
- Arguments are stored in the transaction log
- An alternative to store data cheaply
- Client software can create “listeners” to events (eg. in Python/JS)

Solidity: events

```
pragma solidity ^0.4.24;

contract ClientReceipt {
    event Deposit(
        address indexed _from,
        bytes32 indexed _id,
        uint _value
    );

    function deposit(bytes32 _id) public payable {
        emit Deposit(msg.sender, _id, msg.value);
    }
}
```

Contract - Solidity

```
var abi = /* abi as generated by the compiler */;
var web3 = /* http/ws connection to Eth full node */;
var contractObject = web3.eth.contract(abi);
var contractInstance = contractObject.at("0x1234...ab67");
/* address */

var event = contractInstance.Deposit();

// watch for changes
event.watch(function(error, result){
    if (!error)
        console.log(result);
    ....
    /* use result to access event data .. */
});
```

Client - Javascript

Solidity: Inheritance

- Multiple inheritance
- **One contract** is created on the **blockchain** for all derived contracts: codes concatenate
- The general **inheritance** system is very **similar** to **Python's**

Solidity: Inheritance

- Use ***is*** keyword to **extend** a contract
- **Derived** contracts: **access** all non-private members, internal functions and state variables
- **Abstract** contracts can be used as **interfaces**
- **Functions** can be **overridden**
- **Interfaces**: functions are not implemented

Solidity: Inheritance

```
pragma solidity ^0.4.24;

interface Regulator {
    function checkValue(uint amount)  external returns (bool);
    function loan() external returns (bool);
}

contract LocalBank is Bank(10) {
    string private name;
    uint private age;

    function setName(string newName) public {
        name = newName;
    }
    function getName() public view returns (string) {
        return name;
    }
    function setAge(uint newAge) public {
        age = newAge;
    }
    function getAge() public view returns (uint) {
        return age;
    }
}
```

```
contract Bank is Regulator {
    uint private value;
    constructor (uint amount) public  {
        value = amount;
    }
    function deposit(uint amount) public {
        value += amount;
    }
    function withdraw(uint amount) public {
        if (checkValue(amount)) {
            value -= amount;
        }
    }
    function balance() public view returns (uint) {
        return value;
    }
    function checkValue(uint amount) public view returns (bool) {
        return value >= amount;
    }
    function loan() public view returns (bool) {
        return value > 0;
    }
}
```

```
pragma solidity ^0.4.24;


contract Jedi {

   function computeForce() internal pure returns (uint){
         return 50;
   }

   function getExtraForce() private pure returns (uint) {
         return 100;
   }

}


contract Ewok {
   Jedi j = new Jedi();
   uint force =  j.computeForce(); // error private method
}
```

```
pragma solidity ^0.4.24;


contract Human is Jedi {
   uint age = 70;
   string name = “Luke”;
   string lastName = “Skywalker”;
   bool isMaster = false;
   uint force = 0;

   function setMaster(bool _master) external {
         isMaster = _master;
         force = computeForce(); // internal call
         force = force + getExtraForce(); // error private
method
   }

   function getJedi() public view returns (uint, string, string,
bool){
         return (age, name, lastName, isMaster) //
multi-values
   }

}
```

Solidity: Modifiers

```
pragma solidity ^0.4.24;

contract owned {

    address owner;


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


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

Declare modifier

```
contract mortal is owned {
    function close() public onlyOwner {
        selfdestruct(owner);
    }
}
```

Apply modifier

Solidity: units and globally available variables

- Ether Units
  - A literal number can take a suffix of wei, finney, szabo or ether (2 ether == 2000 finney evaluates to true)
- Time Units
  - Suffixes like seconds, minutes, hours, days, weeks and years (1 hours == 60 minutes)

Solidity: units and globally available variables

- Block and Transaction Properties
  - block.blockhash
  - Block.coinbase
  - block.timestamp
  - msg.data
  - msg.gas
  - msg.value
  - msg.sender
  - tx.origin

Solidity: units and globally available variables

- Error Handling
  - via error objects (see Solidity docs)
  - assert
  - require
  - revert
- Mathematical and Cryptographic Functions
  - addmod, mulmod
  - Keccak256 (SHA-3), sha256, ripemd160

Solidity: units and globally available variables

- Address Related
  - <address>.balance
  - <address>.transfer
  - <address>.send
  - <address>.call, <address>.callcode, <address>.delegatecall
- Contract Related
  - this, selfdestruct

Send ether

Send ether

Function	Gas forwarded	Error handling	Notes
`transfer`	2300	throws error on failure	• **Safe** against re-entrancy • **Fails** if recipient contract's fallback function consumes >2300 gas
`send`	2300	`false` on failure	• **Safe** against re-entrancy • **Fails** if recipient contract's fallback function consumes >2300 gas
`call`	all remaining gas	`false` on failure	• **Not safe** against re-entrancy

Interacting with other contracts

Interacting with other contracts

```
contract Planet {
     string private name;
     constructor (string memory _name){ name = _name; }
     function getName() public returns(string memory) { return name; }
}

contract Universe {
     address[] planets;
     event NewPlanet(address planet, string name);

     function createNewPlanet(string memory name) public {
          Planet p = new Planet(name);
          planets.push(address(p));
          emit NewPlanet(address(p), p.getName());
     }
}
```