

# Contracts

“A contract is a legally binding agreement which recognises and governs the rights and duties of the parties to the agreement.” - Wikipedia

A mutually beneficial relationship between equals (Jean-Jacques Rousseau)

# 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 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 the current transaction with ScriptPubKey of the referenced transaction
  - check if it successfully compiles with no errors

# Bitcoin Script

- Stack-based
- 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 Script example



Stack	Script	Description
Empty	<sig1> <pubKey1> OP_DUP OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	
<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> <pubKey1>	OP_HASH160 <pubKeyHash1> OP_EQUALVERIFY OP_CHECKSIG	Duplicate top stack item
<sig1> <pubKey1> <pub1Hash>	<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>	OP_CHECKSIG	Check if top two items are equal
Empty	TRUE	Verify the signature.

# 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.
- Fees depend on transaction's **length**.

# 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.
    - No SPV clients.
- 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.



# Alternative blockchain applications

- Namecoin:
  - Bitcoin fork: Currency NMC
  - Decentralized name registration database: DNS, identities etc
- Colored coins:
  - On top of Bitcoin
  - Allows people to create their own digital currencies
- OmniLayer (formerly Mastercoin)
  - On top of Bitcoin
  - Distributed exchange, smart property, distributed e-commerce, etc
- OpenBazaar
  - On top of Bitcoin
  - Decentralized marketplace

# Ethereum

# Same principles as Bitcoin

- **A peer-to-peer network:** connects the participants
- **A consensus algorithm:** Proof of Work (will move to PoS)
- **A digital currency:** ether
- **A global ledger:** the blockchain
  - Addresses: key pair
  - Wallets
  - Transactions: digital signatures
  - Blocks

# Ethereum: A universal Replicated State Machine

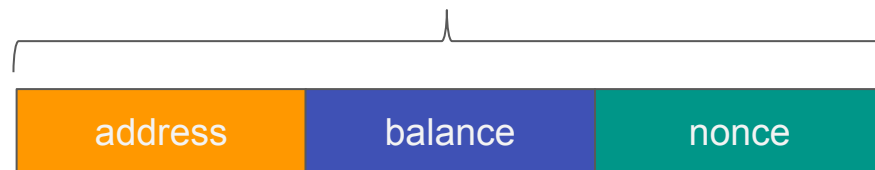
- Transaction-based deterministic state machine
  - Global singleton state
  - A virtual machine that applies changes to global state
- A global decentralized computing infrastructure
- Anyone can create their own state transition functions
- Stack-based bytecode language
- Turing-completeness
- Smart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

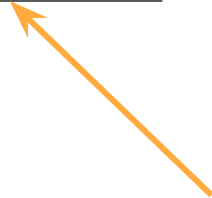
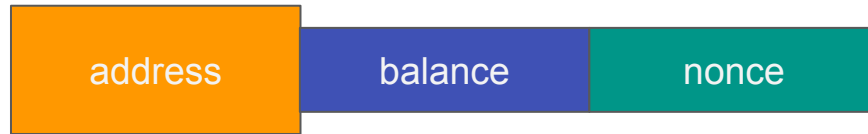
# Ethereum accounts

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

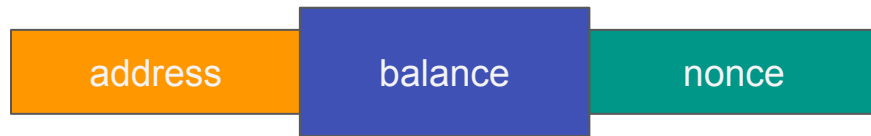


# Ethereum account





The **address** of the account



No UTXOs  
in Ethereum

The **balance** of the account





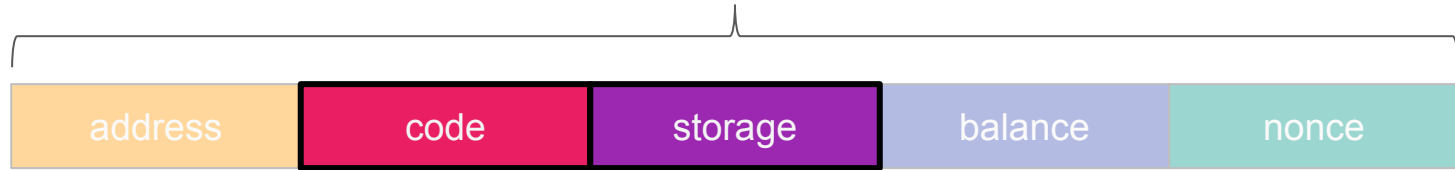
# UTXO vs Accounts

- UTXOs pros:
  - Higher degree of privacy
  - Scalability (parallelism, sharding)
- Accounts pros:
  - Space saving
  - Better fungibility
  - Simplicity
  - Efficiency

# Two types of accounts

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

# Ethereum contract account



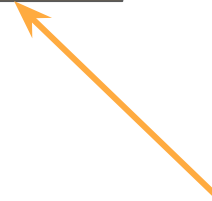
# Ethereum accounts

	Personal account	Contract account
address	$H(\text{pub\_key})$	$H(\text{creator, nonce})$
code	$\emptyset$	Code to be executed
storage	$\emptyset$	Data of the contract
balance	ETH balance (in Wei)	ETH balance (in Wei)
nonce	# transaction sent	# transaction sent

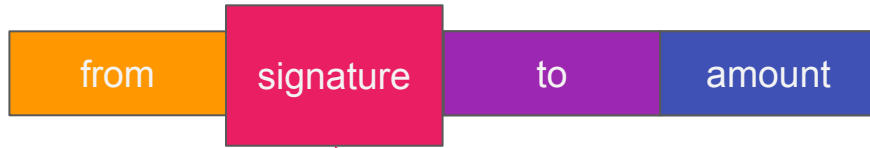


# Ethereum transaction



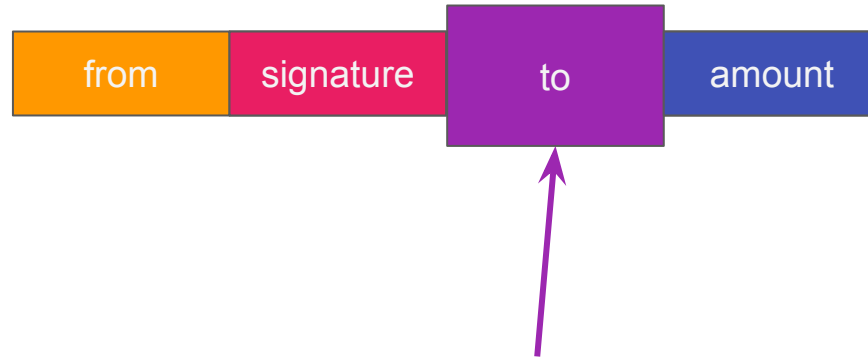


The **sender** of the transaction



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

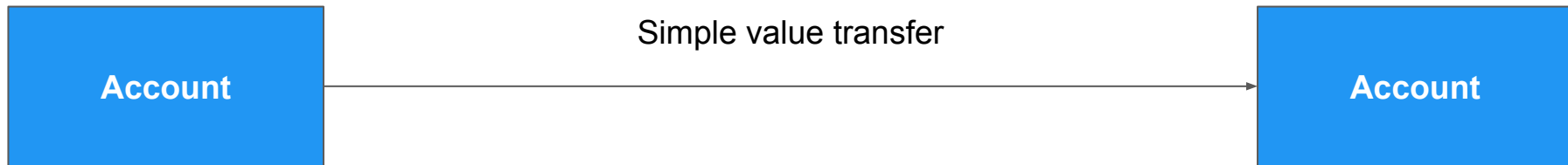




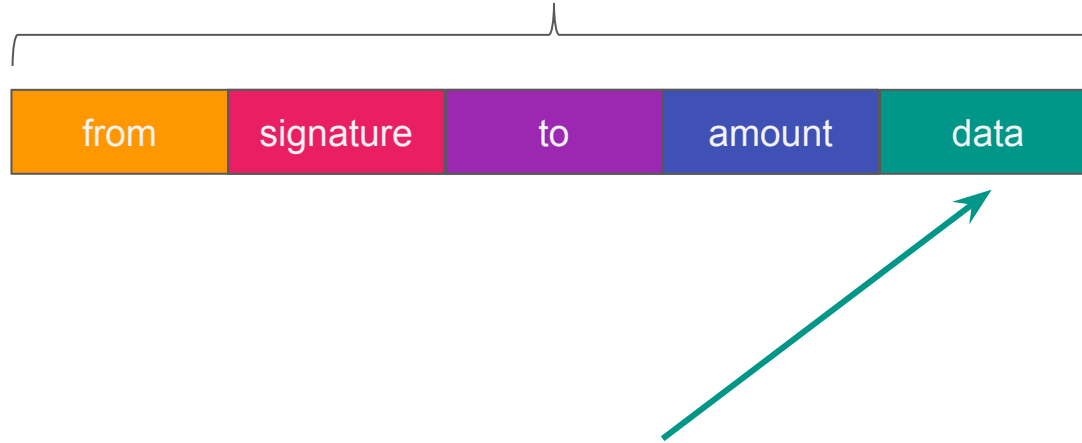
**Receiver** of the transaction



**Amount** transferred by transaction  
Given in Wei



# a transaction about a contract



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

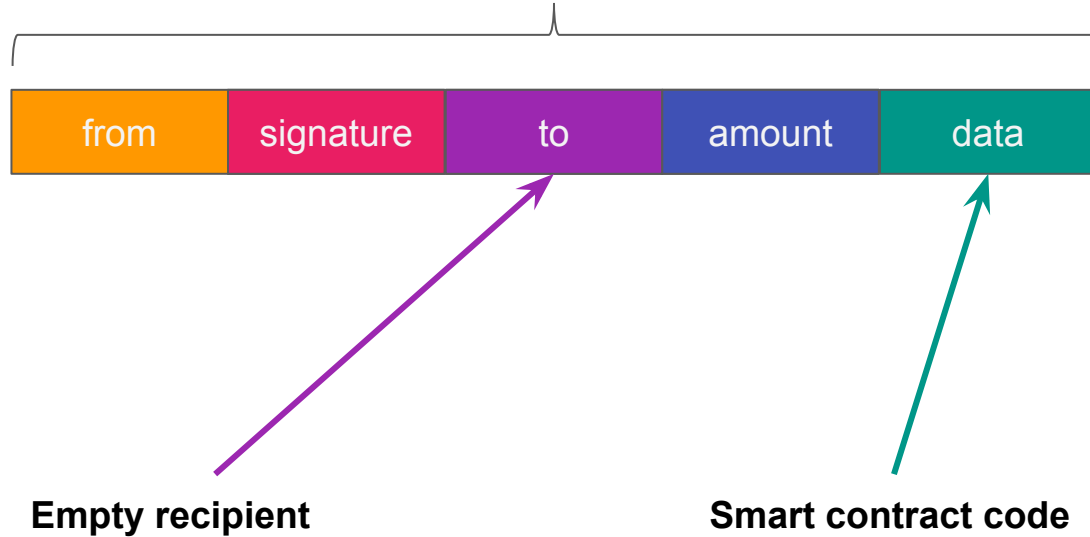
Transaction **about contracts**:  
Will contain **data about the contract**

# Smart contract lifecycle





# Transaction for contract creation





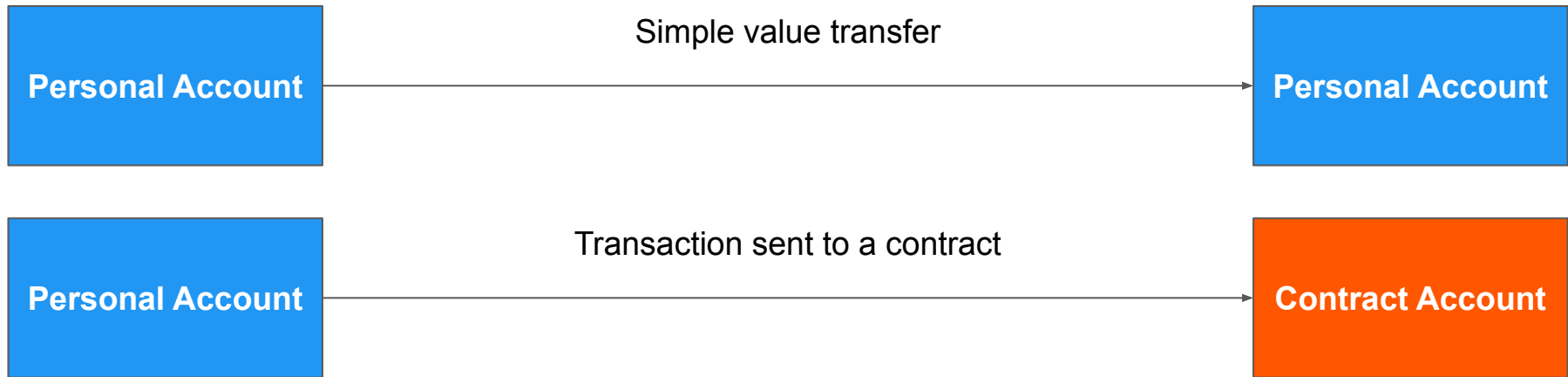


# Transaction for contract interaction



**Contract address**

**Which method to call + arguments**

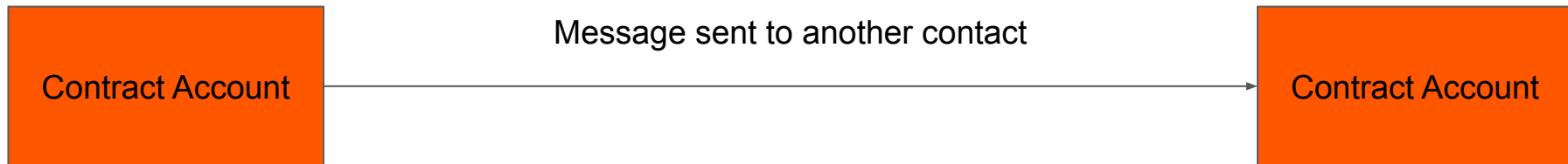


# Contract method call

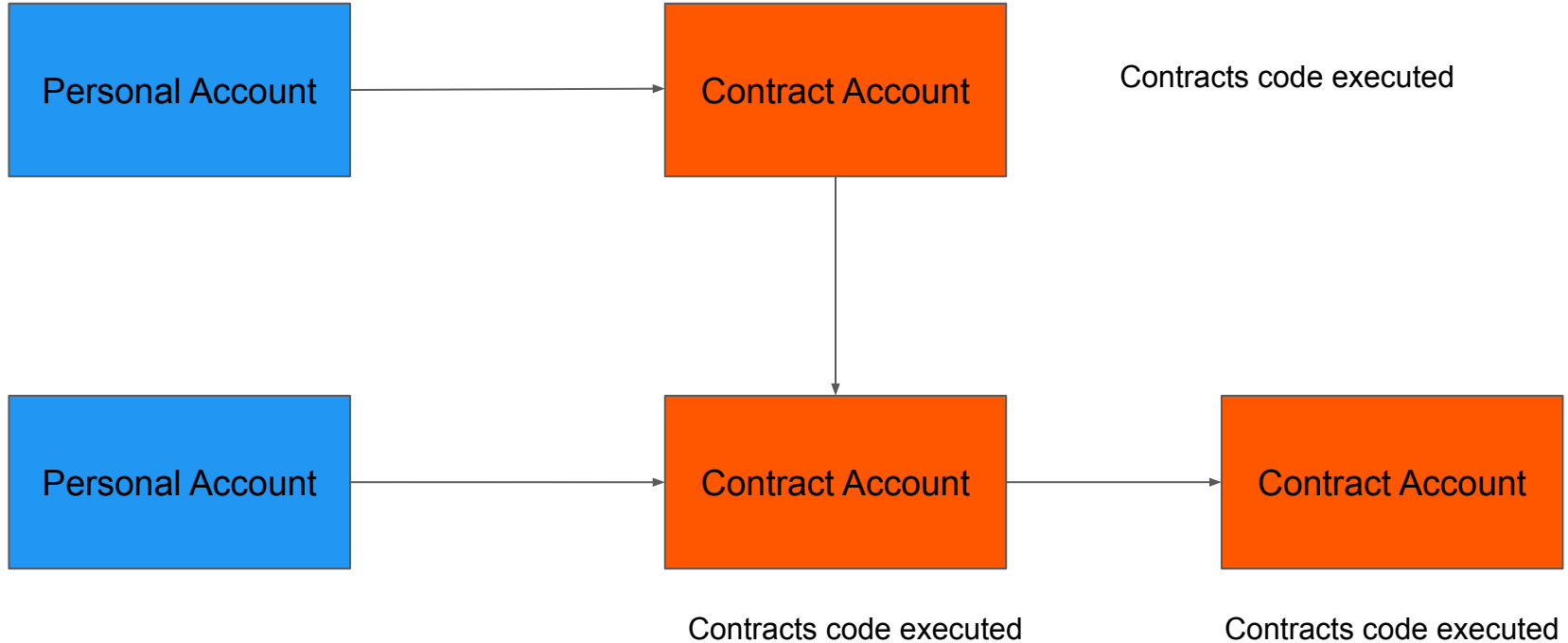
- When contract account is activated:
  - a. Contract **code** runs
  - b. It can read / write to **internal storage**
  - c. It can **send other transactions** or **call other contracts**
- Can't initiate new transactions on their own
- Can only fire transactions in response to other transactions received

# 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**



# Transactions & messages



# Types of transactions

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





# a transaction for contract destruction



**Contract address**

**The name of a method that calls the  
selfdestruct opcode**

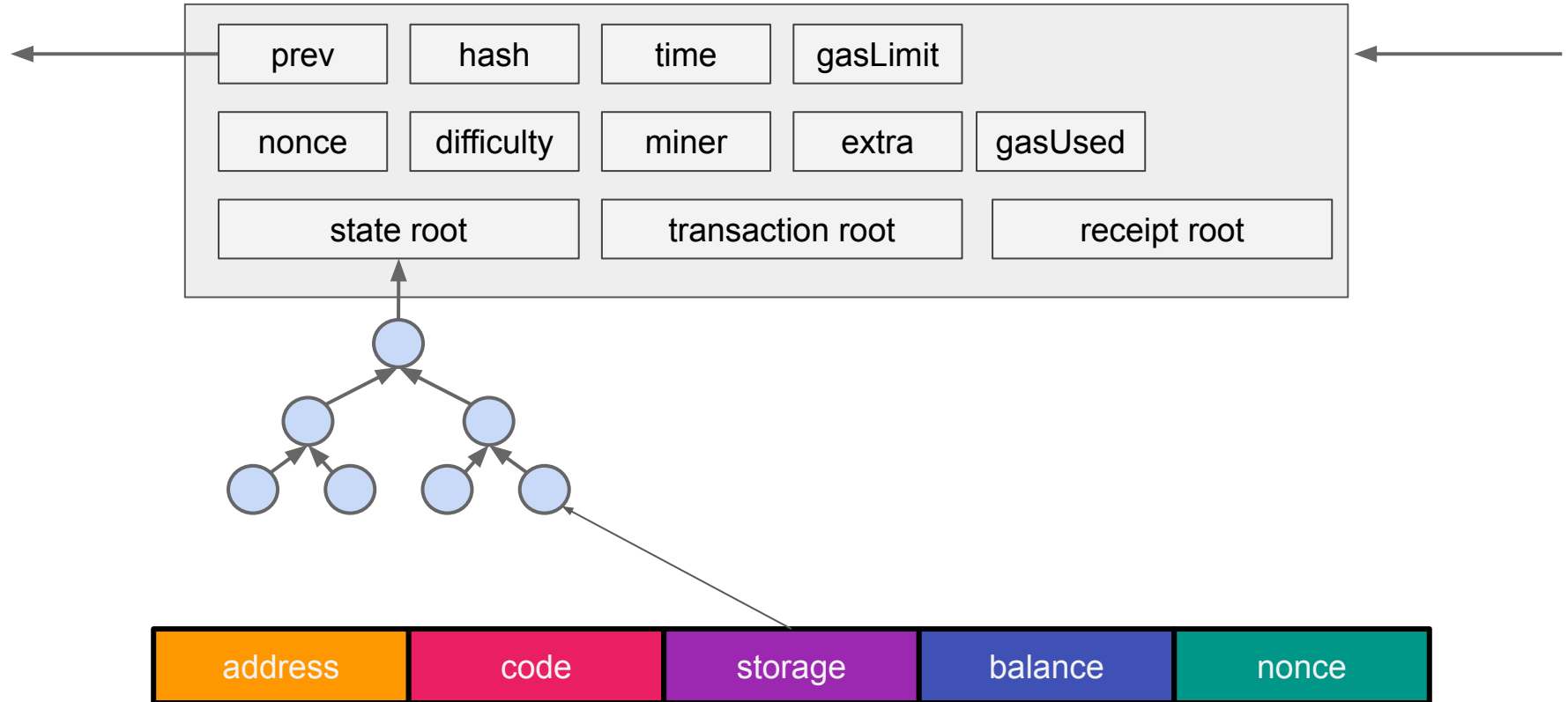
# Ethereum Virtual Machine

- 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

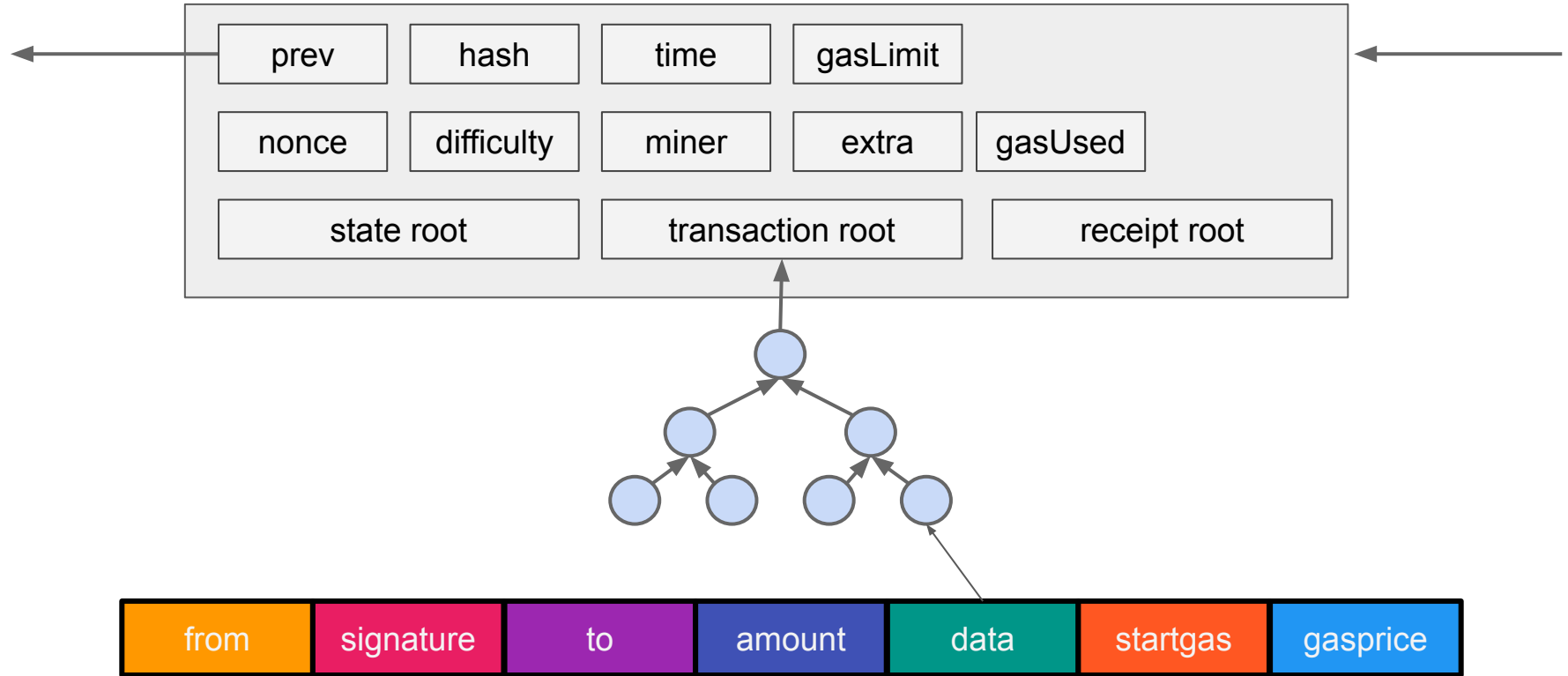
# EVM: contract execution

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

# Ethereum block



# Ethereum block



# Ethereum Mining

- **Similar to Bitcoin**
- **Blocks** contain: **transaction** list and most **recent state**
- Block **time**: ~12 - 15 **seconds**
- **Proof-of-work**: Ethash (designed to be **memory-hard**)
- Casper: Future transition to **proof-of-stake**
- **Winner** of the block: **3 ETH**

# Ethereum Mining

- Uses a variant of **GHOST** (Greedy Heaviest Observed Subtree) protocol to **reward stale blocks**
- The GHOST protocol rule **picks the chain** that has had the **most computation** done upon it
- Planned hard forks:
  - Frontier, Homestead, and Byzantium (Metropolis phase 1)
  - Next one at 2019: Constantinople (Metropolis phase 2)

# Ethereum fees: the phone booth model





# Gas: a necessary evil

- Every node on the network:
  - evaluate all **transactions**
  - store all **state**
- Halting problem

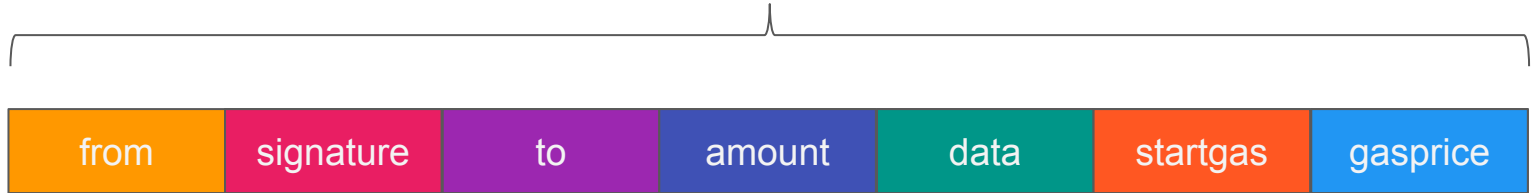


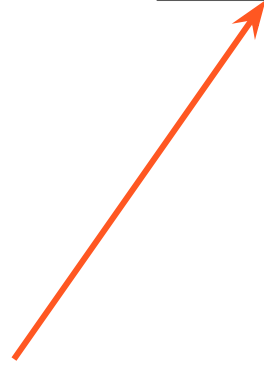
# Gas: a necessary evil

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



# Ethereum transaction



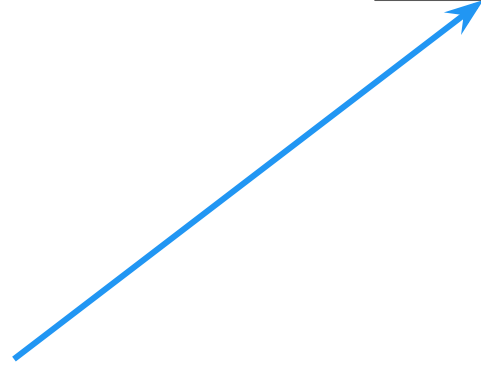


Maximum amount of gas willing to pay

# Gas Limit

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





Price to pay per gas unit

# Gas Price

- Measured in **gwei** ( $1 \times 10^9$  Wei)
- Determines how **quickly** a transaction will be **mined**



# Transaction Fees





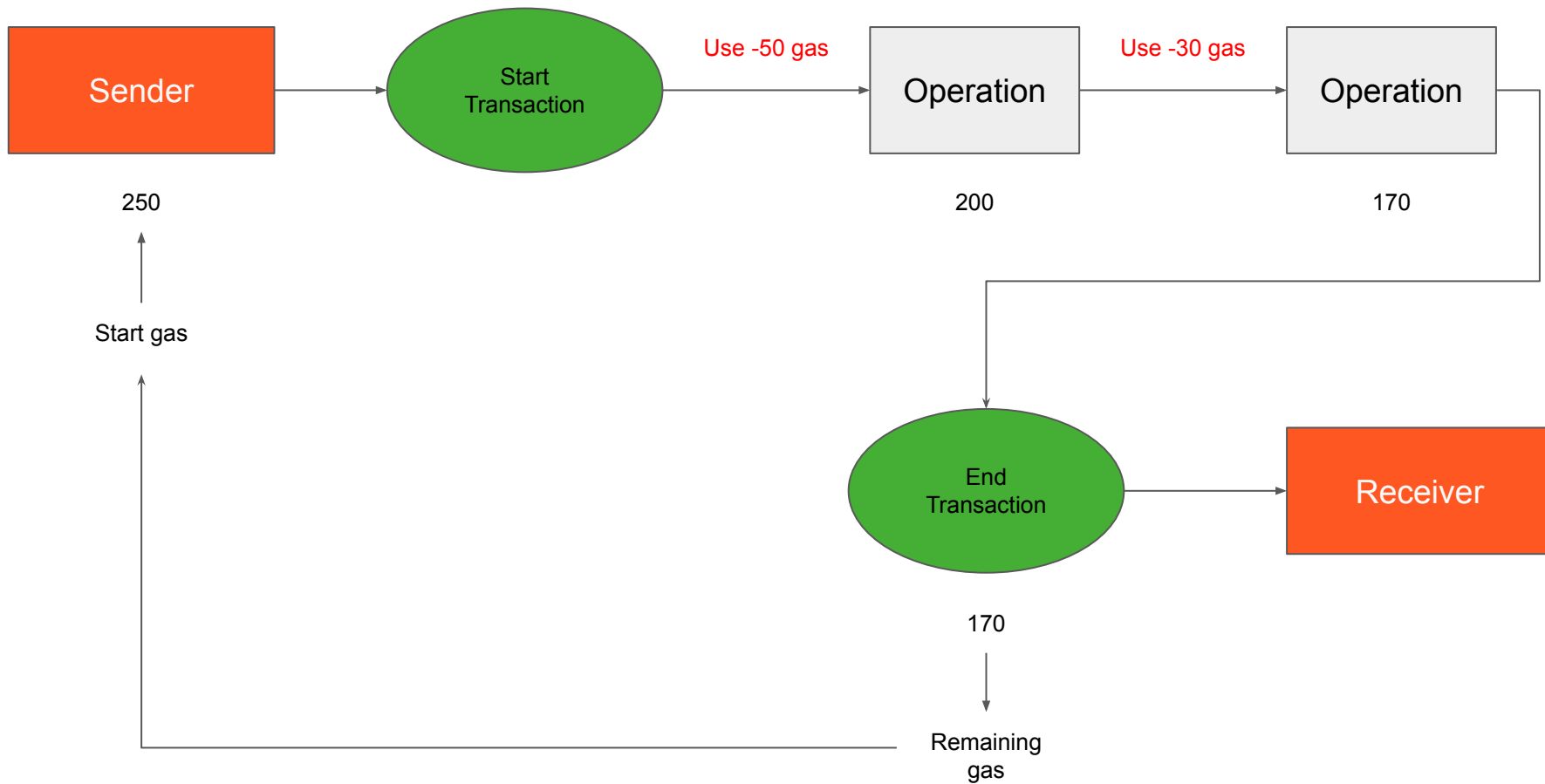
# Storage in Ethereum

ETH Price: \$166.41 (Apr 17, 2019) - Gas Price: 3 Gwei

Size	Gas	Cost (ETH)	Cost (\$)
32 bytes	21.000	0.000063	\$0.36088
1KB	724.664	0.002174	\$0.01046
1MB	~697.325.562	2.09198	\$347.268
10MB	~7.000.000.000	~21	\$3,486
100MB	~70.000.000.000	~210	\$34,860
1GB	~700.000.000.000	~2100	\$348,600

# 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**



# Out of gas exceptions

- State **reverts** to **previous state**
- $\text{gas\_limit} * \text{gas\_price}$  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** looks like **JavaScript**

# Solidity

- **Contracts** look like **classes / objects**
- **Static** Type (Most types can be cast, e.g `bool(x)`)
- Most of the control structures from **JavaScript** are available in **Solidity** **except** for **switch**

```
pragma solidity ^0.5.1;
```

```
contract HelloWorld {
```

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

```
        return 'Hello World!';
```

```
    }
```

```
}
```



# Pragmas

```
pragma solidity 0.5.0;
```

```
pragma solidity ^0.5.1;
```

```
pragma solidity >=0.5.0 < 0.6.0;
```

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

# Contract

```
contract <contract-name> { ... }
```

# Constructors

```
contract HelloWorld {  
    constructor () public { ... }  
}
```

```
contract HelloWorld {  
    constructor (uint x, string y) public { ... }  
}
```

# 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**: in **memory** by default

# Types

- The **type** of each variable **needs to be specified** (Solidity is a statically typed language)
- **Two** types:
  - **Value** types
  - **Reference** types
- **“undefined”** or **“null”** values **does not exist** in Solidity
- **Variables** without a value **always** have a **default value** (zero-state) dependent on its 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  $b \in \{8, 16, 24, 32, \dots, 256\}$
- 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).

Payable address is an address you can send Ether to (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 }  
}
```

# Reference types

# Types: arrays

```
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 a **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. You **cannot** resize a memory array.

# Types: Struct

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

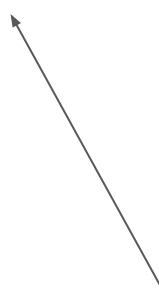
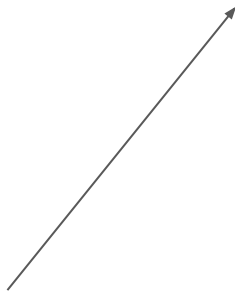
- A struct cannot contain a struct of its own type (the size of the struct has to be finite).
- A struct can contain mappings.

# Types: Mappings

```
contract Mappings {  
    mapping(address => uint256) balances;  
}
```

key

value



# Visibility



# Visibility

- **public:** Public functions can be called from other contracts, internally and personal accounts. For public state variables an automatic getter function is being created.
- **external:** External functions cannot be called internally. Variables cannot be declared as external.
- **Internal:** Internal function 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 in derived contract. **Warning:** private variables are visible to all observers external to the blockchain.

Data location

# Data location: areas

- Every complex type (arrays, structs, mappings) have a data location.
- Two types of location: storage and memory.
- As of Solidity version **5.0.0** you must **always declare** the data **location** of complex types inside functions' body, arguments and returned values.

# Data location: areas

- **Storage:**
  - Persistent.
  - All state variables are saved to storage.
  - Function's complex local values are saved to storage by default. (Solidity versions  $\geq 5.0.0$  force you to declare the data location).
- **Memory:**
  - Non-persistent.
  - Function's arguments and returned values are stored to memory by default. (Solidity versions  $\geq 5.0.0$  force you to declare the data location for complex types).

# Data location: assignment

- storage  $\leftrightarrow$  memory: copy
- state variable  $\leftarrow$  state variable, storage and memory: copy
- memory  $\leftrightarrow$  memory : reference
- local storage variable  $\leftarrow$  storage: reference

# Fallback function

# Fallback function

```
contract Fallback {  
    function () external {  
        ...  
    }  
}
```

Unnamed function



- No arguments (`msg.data` is accessible).
- No returned values.
- Mandatory visibility: `external`.
- Executed if no data (transaction field) is supplied or if the function that a user tries to call does not exist.
- Executed whenever the contract receives plain Ether (without data).
- To receive Ether the fallback function must be marked as `payable`.
- In the absence of fallback function a contract cannot receive Ether and an exception is thrown.

# 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 despite  
access declaration!!



```
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: events

- EVM logging mechanism
- Register listeners to events
- Arguments are stored in the transaction log
- An alternative to store data cheaply

# 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);  
    }  
}
```

```
var abi = /* abi as generated by the compiler */;  
var ClientReceipt = web3.eth.contract(abi);  
var clientInstance = ClientReceipt.at("0x1234...ab67" /*  
address */);  
  
var event = clientInstance.Deposit();  
  
// watch for changes  
event.watch(function(error, result){  
    if (!error)  
        console.log(result);  
});
```

# Solidity: 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;  
    }  
}
```

# Solidity: 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;  
    }  
}
```

# Solidity: Inheritance

- Multiple inheritance
- **One contract** is created on the **blockchain**: codes concatenates
- 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;
    }
}
```



# Solidity: Modifiers

```
pragma solidity ^0.4.24;
```

```
contract owned {
```

```
    address owner;
```

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

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

```
    }  
}
```

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

# 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`
- `msg.data`
- `msg.gas`
- `Msg.value`
- `msg.sender`
- `now`
- `tx.origin`

# Solidity: units and globally available variables

- Error Handling
  - 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
<code>transfer</code>	2300	throws on failure	<b>Safe</b> against re-entrancy
<code>send</code>	2300	<code>false</code> on failure	<b>Safe</b> against re-entrancy
<code>call</code>	all remaining gas	<code>false</code> on failure	<b>Not safe</b> against re-entrancy

Interacting with other  
contracts



# Interacting with other contracts

```
contract Planet {  
    string private name;  
    constructor (string memory _name) public { 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());  
    }  
}
```

# Thank you!

