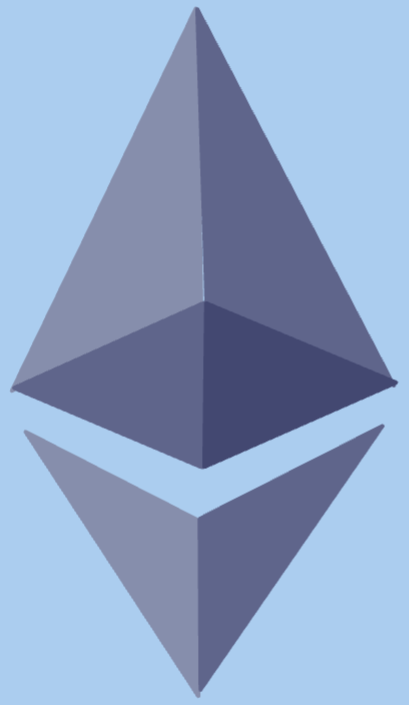
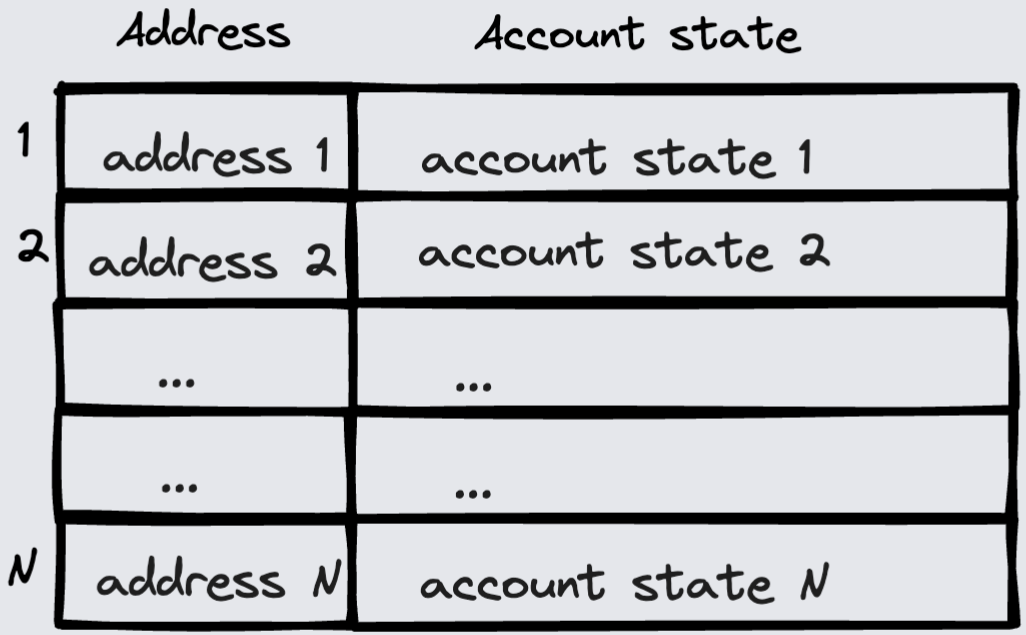
Ethereum & Smart Contracts

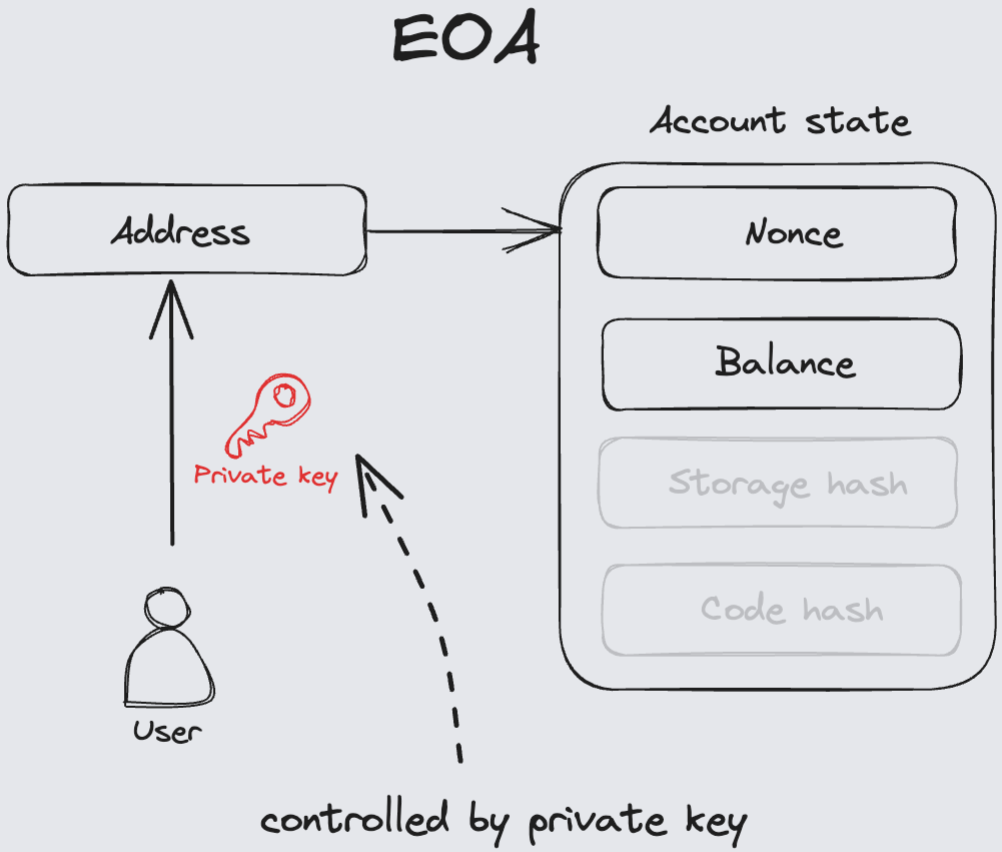
Prepared by Kirill Sizov

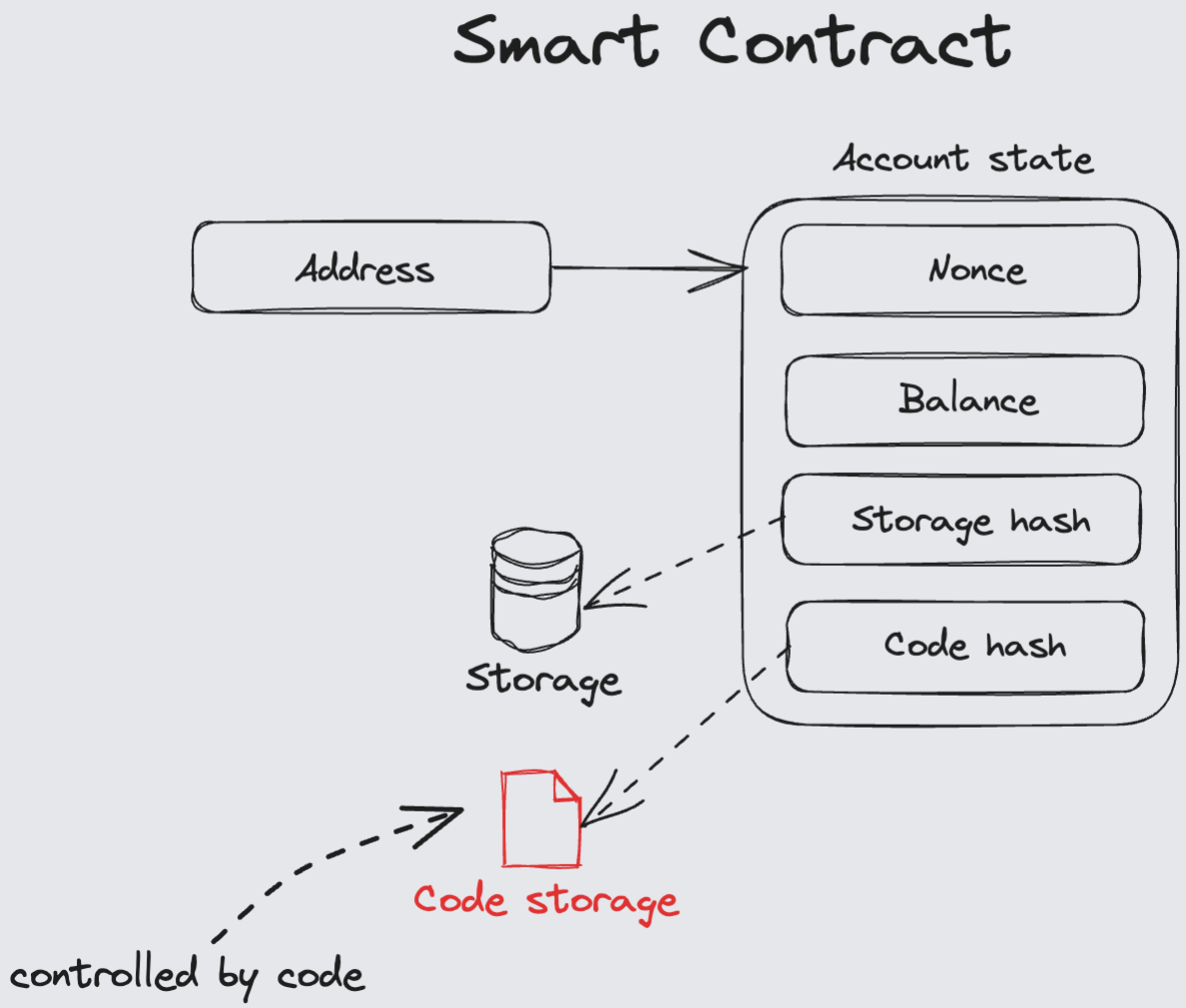


Blockchain state

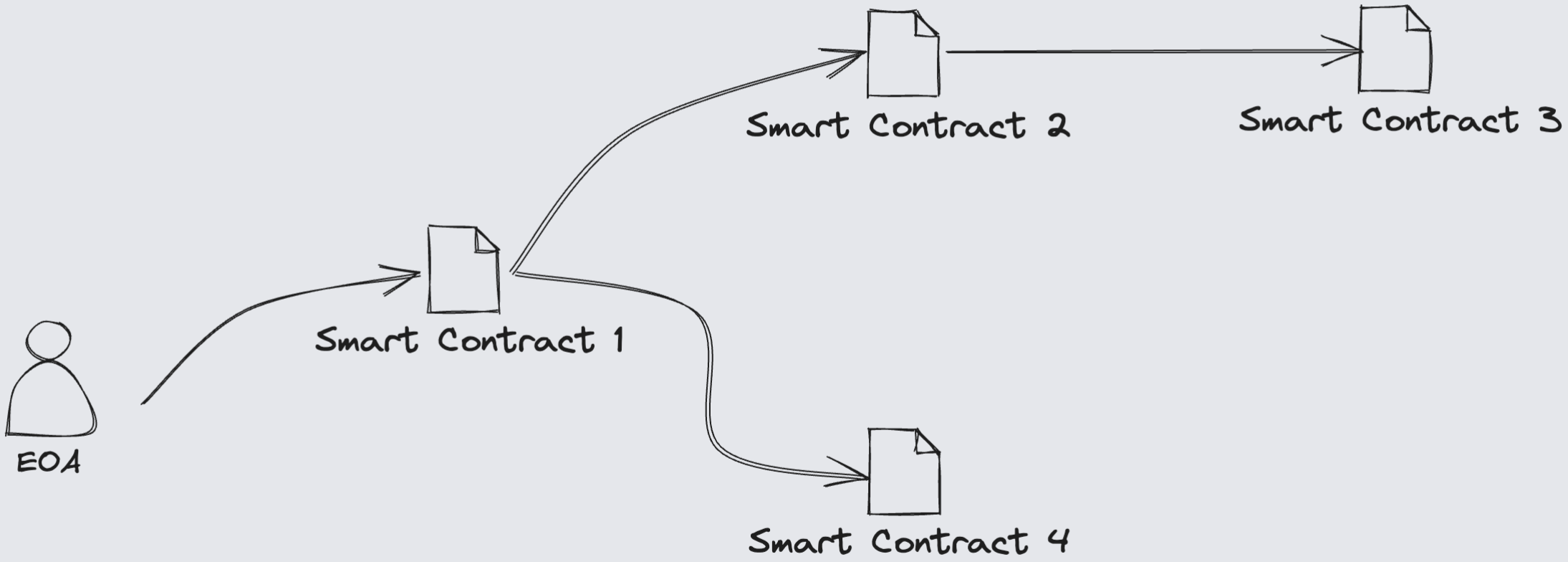


Account types





Transaction flow



Transaction flow: important considerations

- Atomicity of transaction execution.
- No overlapping between transactions.
- Transaction order is not guaranteed.

Transaction structure

Parameter	Description
Nonce	A sequence number, issued by the originating EOA, used to prevent message replay.
Gas price	The price of gas (in wei) the originator is willing to pay.
Gas limit	The maximum amount of gas the originator is willing to buy for this transaction.
Recipient	The destination address.
Value	The amount of ether to send to the destination.
Data	The variable-length binary data payload.

Ether and wei

Transactions are paid with ether. 1 `ether` = 10^{18} `wei`.

Ether is used for:

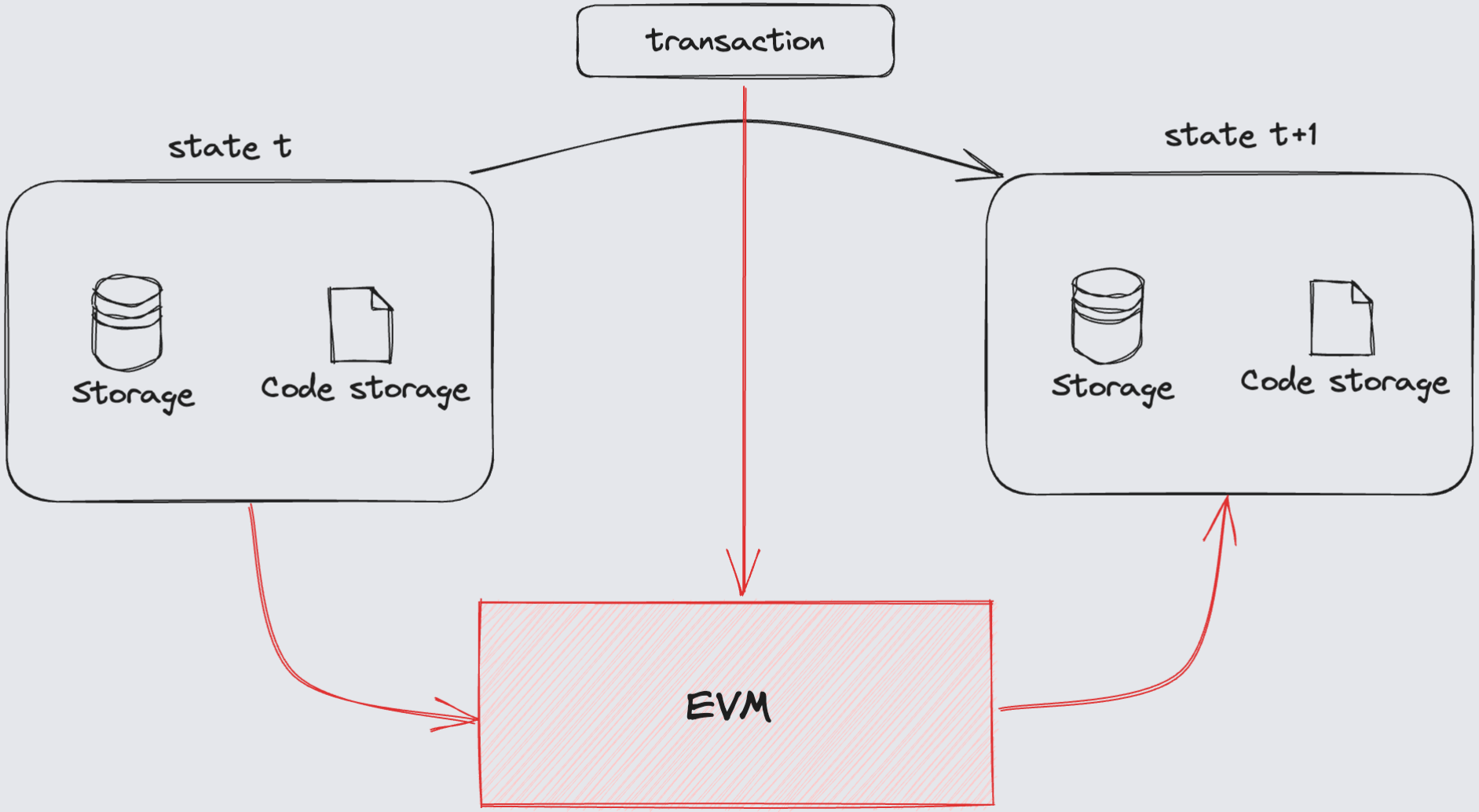
- Transferring funds.
- Paying transaction fee.
- Reward validators.

Gas

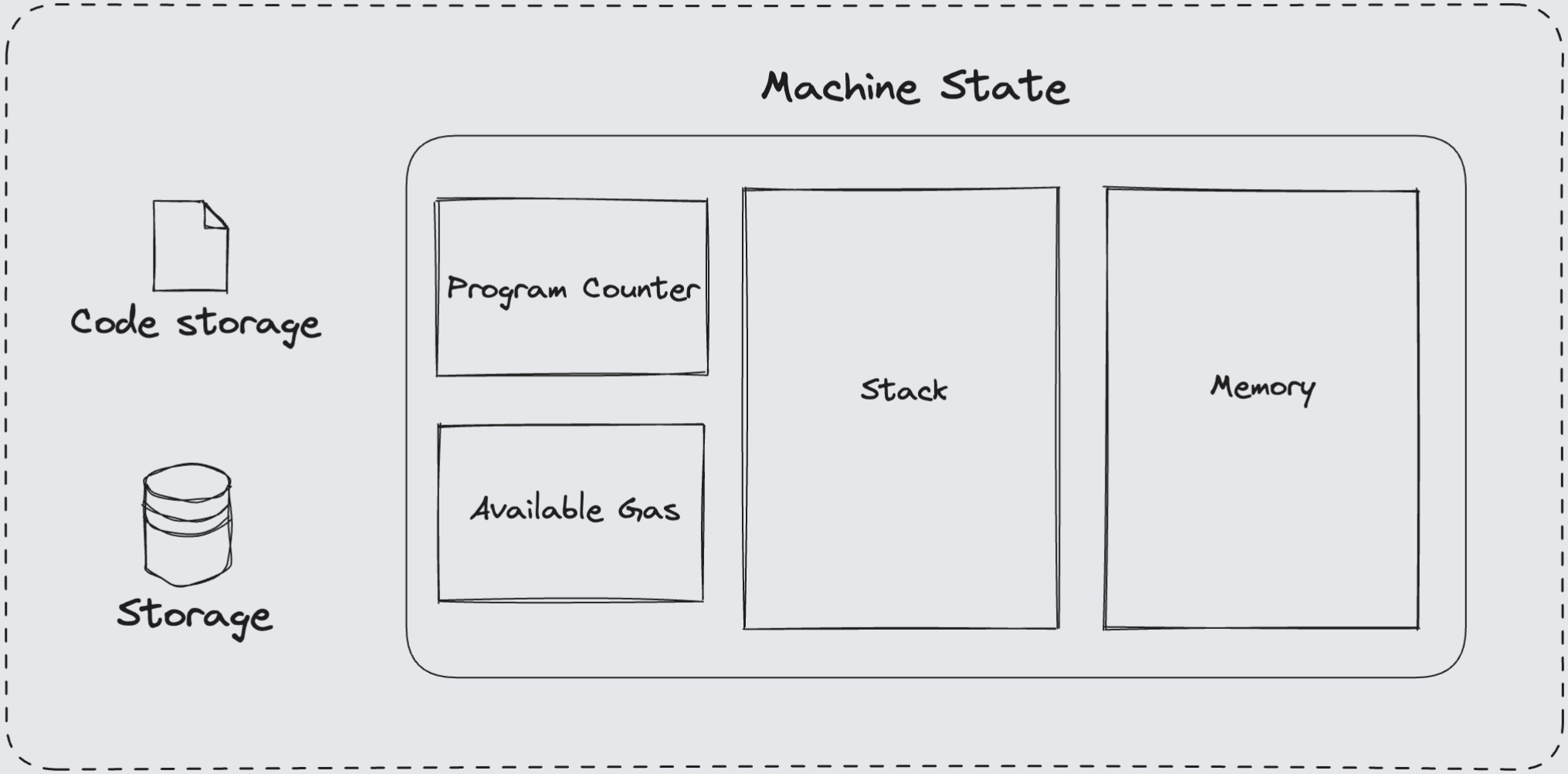
Gas is a unit of computation.

- `tx fee = gas spent * gas price`
- `gas spent` is the total amount of gas used in a transaction.
- `gas price` is how much ether transaction pay per gas.

Ethereum Virtual Machine (EVM)

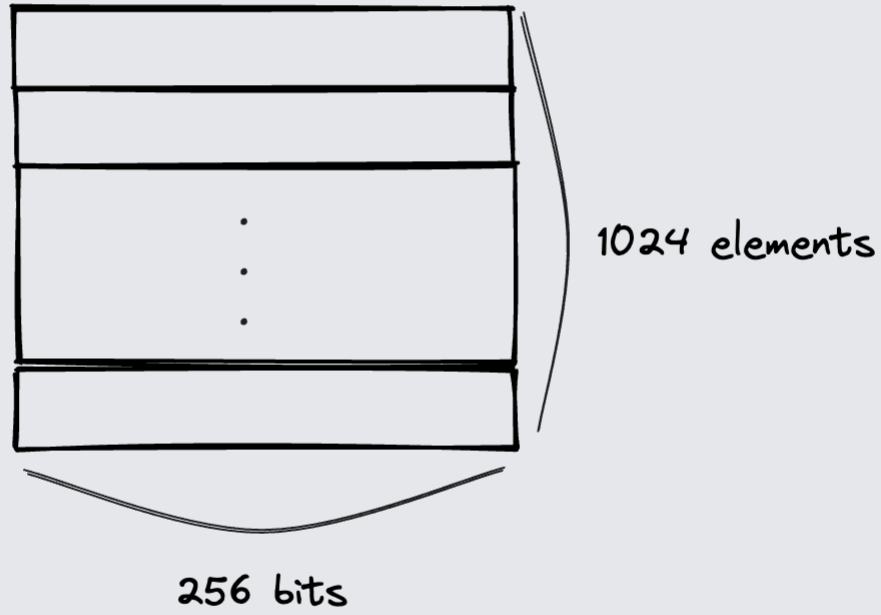


EVM architecture



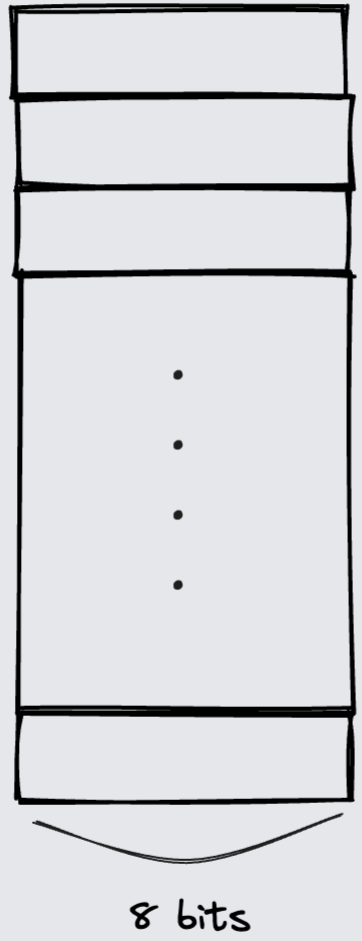
Stack

- Serves a large number of tasks,
  such as performing computations,
  storing temporary values, control flow, etc.
- Access with many instructions,
  e.g. PUSH/POP/COPY/SWAP.



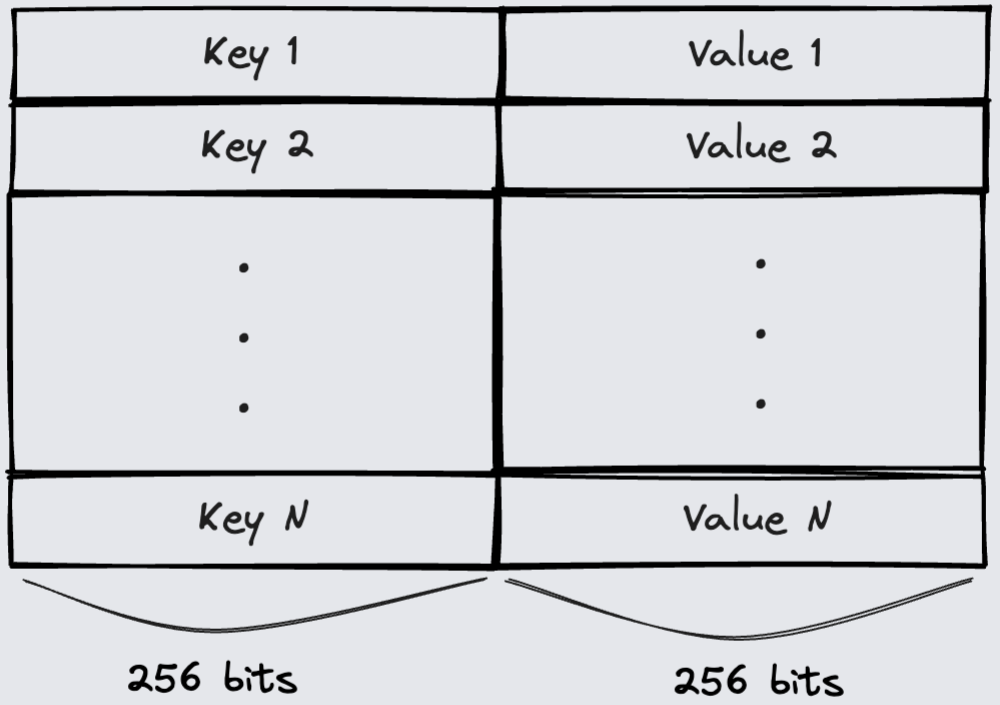
Memory

- Used to hold transient data while a contract is being executed.
- Structured as a linear byte array, and its size can be expanded as needed by a contract during execution.
- Access with MSTORE/MSTORE8/MLOAD.



Storage

- Used for persistent data.
- Structured as a mapping.
- All locations in storage are initially zero.
- Access with SSTORE/SLOAD.



Solidity

Solidity

- High-level language for EVM smart contracts.
- Object-oriented, statically typed.
- Influenced by C++, Python, JavaScript.

Hello world!

- `pragma` specifies the compiler version of Solidity.

```
// SPDX-License-Identifier: MIT
// compiler version must be greater than
// or equal to 0.8.20 and less than 0.9.0
pragma solidity ^0.8.20;

contract HelloWorld {
    string public greet = "Hello World!";
}
```

Data locations

- `storage` - variable is a state variable (store on blockchain).
- `memory` - variable is in memory and it exists while a function is being called.
- `calldata` - special data location that contains function arguments.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Variables {
    // State variables are stored in storage.
    string public text = "Hello";
    uint public num = 123;

    // Argument variables sometimes are stored in calldata
    function doSomething(uint[] calldata nums) public {

        // Local variables are stored in memory;
        uint x = 456;
        bool f = true;
    }
}
```

Data types

- **Value types** variables store their own data. These are the basic data types.
- **Reference types** variables point to the memory address of stored data.

Value types

- `bool`: `true` or `false`.
- `intX` / `uintX`: signed and unsigned integers of various sizes (from 8 to 256 bits).
- `address`: 20-byte EVM address
- `bytesX`: fixed-size array of bytes up to 32 bytes.
- `enum`: used to create user-defined data types.

Reference types

- **Fixed arrays** - pre-defined size at runtime, declared during initialization.
- **Dynamic arrays** - allocate size dynamically at runtime.
- **Structs** - custom type containing members of other types.
- **Mappings** - hash map.

Reference type variable may have an additional annotation, the “data location”, about where it is stored.

Constants and immutable

- Constants are variables that cannot be modified. Their value is hard coded and using constants can save gas cost.
- Immutable variables are like constants. Values of immutable variables can be set inside the constructor but cannot be modified afterwards.

Functions

Visibility

- `public` - any contract and account can call.
- `private` - only inside the contract that defines the function.
- `internal` - only inside contract that inherits an internal function.
- `external` - only other contracts and accounts can call.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Example {
    // Private function can only be called
    // - inside this contract
    // Contracts that inherit this contract cannot call this function.
    function privateFunc() private pure returns (string memory) {
        return "private function called";
    }

    // Internal function can be called
    // - inside this contract
    // - inside contracts that inherit this contract
    function internalFunc() internal pure returns (string memory) {
        return "internal function called";
    }

    // Public functions can be called
    // - inside this contract
    // - inside contracts that inherit this contract
    // - by other contracts and accounts
    function publicFunc() public pure returns (string memory) {
        return "public function called";
    }

    // External functions can only be called
    // - by other contracts and accounts
    function externalFunc() external pure returns (string memory) {
        return "external function called";
    }

    // State variables
    string private privateVar = "my private variable";
    string internal internalVar = "my internal variable";
    string public publicVar = "my public variable";
    // State variables cannot be external so this code won't compile.
    // string external externalVar = "my external variable";
}
```

State mutability

- `view` function declares that no state will be changed.
- `pure` function declares that no state variable will be changed or read.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract ViewAndPure {
    uint public x = 1;

    // Promise not to modify the state.
    function addToX(uint y) public view returns (uint) {
        return x + y;
    }

    // Promise not to modify or read from the state.
    function add(uint i, uint j) public pure returns (uint) {
        return i + j;
    }
}
```

Modifiers

Modifiers are code that can be run before and / or after a function call.

Modifiers can be used to:

- Restrict access.
- Validate inputs.
- Guard against reentrancy hack.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract FunctionModifier {
    // We will use these variables to demonstrate how to use
    // modifiers.
    address public owner;

    constructor() {
        // Set the transaction sender as the owner of the contract.
        owner = msg.sender;
    }

    // Modifier to check that the caller is the owner of
    // the contract.
    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        // Underscore is a special character only used inside
        // a function modifier and it tells Solidity to
        // execute the rest of the code.
        _;
    }

    // Modifiers can take inputs. This modifier checks that the
    // address passed in is not the zero address.
    modifier validAddress(address _addr) {
        require(_addr != address(0), "Not valid address");
        _;
    }

    function changeOwner(address _newOwner) public onlyOwner validAddress(_newOwner) {
        owner = _newOwner;
    }
}
```

Events

Events allow logging to the Ethereum blockchain.

Some use cases for events are:

- Listening for events and updating user interface.
- A cheap form of storage.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Event {
    // Event declaration
    // Up to 3 parameters can be indexed.
    // Indexed parameters helps you filter the logs by the indexed parameter
    event Log(address indexed sender, string message);
    event AnotherLog();

    function test() public {
        emit Log(msg.sender, "Hello World!");
        emit Log(msg.sender, "Hello EVM!");
        emit AnotherLog();
    }
}
```

Interface

Interact with other contracts by declaring an `interface`.

Interface

- Cannot have any functions implemented.
- Can inherit from other interfaces.
- All declared functions must be external.
- Cannot declare a constructor.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Counter {
    uint public count;

    function increment() external {
        count += 1;
    }
}

interface ICounter {
    function count() external view returns (uint);

    function increment() external;
}

contract MyContract {
    function incrementCounter(address _counter) external {
        ICounter(_counter).increment();
    }

    function getCount(address _counter) external view returns (uint) {
        return ICounter(_counter).count();
    }
}
```

Payable

Functions and addresses declared `payable` can receive `ether` into the contract.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Payable {
    // Payable address can send Ether via transfer or send
    address payable public owner;

    // Payable constructor can receive Ether
    constructor() payable {
        owner = payable(msg.sender);
    }

    // Function to deposit Ether into this contract.
    // Call this function along with some Ether.
    // The balance of this contract will be automatically updated.
    function deposit() public payable {}

    // Call this function along with some Ether.
    // The function will throw an error since this function is not payable.
    function notPayable() public {}

    // Function to withdraw all Ether from this contract.
    function withdraw() public {
        // get the amount of Ether stored in this contract
        uint amount = address(this).balance;

        // send all Ether to owner
        (bool success, ) = owner.call{value: amount}("");
        require(success, "Failed to send Ether");
    }

    // Function to transfer Ether from this contract to address from input
    function transfer(address payable _to, uint _amount) public {
        // Note that "to" is declared as payable
        (bool success, ) = _to.call{value: _amount}("");
        require(success, "Failed to send Ether");
    }
}
```

Send

You can send Ether to other contracts by

- `transfer` (2300 gas, throws error)
- `send` (2300 gas, returns bool)
- `call` (forward all gas or set gas, returns bool)

Fallback

`fallback` is a special function that is executed either when:

- A function that does not exist is called.
- Ether is sent directly to a contract but `receive()` does not exist or `msg.data` is not empty.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Fallback {
    event Log(string func, uint gas);

    // Fallback function must be declared as external.
    fallback() external payable {
        // send / transfer (forwards 2300 gas to this fallback function)
        // call (forwards all of the gas)
        emit Log("fallback", gasleft());
    }

    // Receive is a variant of fallback that is triggered when msg.data is empty
    receive() external payable {
        emit Log("receive", gasleft());
    }

    // Helper function to check the balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}

contract SendToFallback {
    function transferToFallback(address payable _to) public payable {
        _to.transfer(msg.value);
    }

    function callFallback(address payable _to) public payable {
        (bool sent, ) = _to.call{value: msg.value}("");
        require(sent, "Failed to send Ether");
    }
}
```

Call

`call` is a low level function to interact with other contracts.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

contract Receiver {
    event Received(address caller, uint amount, string message);

    fallback() external payable {
        emit Received(msg.sender, msg.value, "Fallback was called");
    }

    function foo(string memory _message, uint _x) public payable returns (uint) {
        emit Received(msg.sender, msg.value, _message);

        return _x + 1;
    }
}

contract Caller {
    event Response(bool success, bytes data);

    // Let's imagine that contract Caller does not have the source code for the
    // contract Receiver, but we do know the address of contract Receiver and the function to call.
    function testCallFoo(address payable _addr) public payable {
        // You can send ether and specify a custom gas amount
        (bool success, bytes memory data) = _addr.call{value: msg.value, gas: 5000}(
            abi.encodeWithSignature("foo(string,uint256)", "call foo", 123)
        );

        emit Response(success, data);
    }

    // Calling a function that does not exist triggers the fallback function.
    function testCallDoesNotExist(address payable _addr) public payable {
        (bool success, bytes memory data) = _addr.call{value: msg.value}(
            abi.encodeWithSignature("doesNotExist()")
        );

        emit Response(success, data);
    }
}
```

Delegatecall

`delegatecall` is a low level function similar to call.

When contract `A` executes `delegatecall` to contract `B`, `B`'s code is executed
with contract `A`'s storage, `msg.sender` and `msg.value`.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

// NOTE: Deploy this contract first
contract B {
    // NOTE: storage layout must be the same as contract A
    uint public num;
    address public sender;
    uint public value;

    function setVars(uint _num) public payable {
        num = _num;
        sender = msg.sender;
        value = msg.value;
    }
}

contract A {
    uint public num;
    address public sender;
    uint public value;

    function setVars(address _contract, uint _num) public payable {
        // A's storage is set, B is not modified.
        (bool success, bytes memory data) = _contract.delegatecall(
            abi.encodeWithSignature("setVars(uint256)", _num)
        );
    }
}
```

Take a break