title: Anatomy of smart contracts description: An in-depth look into the anatomy of a smart contact – the functions, data, and variables. lang: en

A smart contract is a program that runs at an address on Ethereum. They're made up of data and functions that can execute upon receiving a transaction. Here's an overview of what makes up a smart contract.

Prerequisites {#prerequisites}

Make sure you've read about smart contracts first. This document assumes you're already familiar with programming languages such as JavaScript or Python.

Data {#data}

Any contract data must be assigned to a location: either to `storage` or `memory`. It's costly to modify storage in a smart contract so you need to consider where your data should live.

Storage {#storage}

Persistent data is referred to as storage and is represented by state variables. These values get stored permanently on the blockchain. You need to declare the type so that the contract can keep track of how much storage on the blockchain it needs when it compiles.

`solidity // Solidity example contract SimpleStorage { uint storedData; // State variable // ... }`

```python

# Vyper example

storedData: int128 ```

If you've already programmed object-oriented languages, you'll likely be familiar with most types. However `address` should be new to you if you're new to Ethereum development.

An `address` type can hold an Ethereum address which equates to 20 bytes or 160 bits. It returns in hexadecimal notation with a leading 0x.

Other types include:

- boolean
- integer
- fixed point numbers
- fixed-size byte arrays
- dynamically-sized byte arrays
- Rational and integer literals
- String literals
- Hexadecimal literals
- Enums

For more explanation, take a look at the docs:

- See Vyper types
- See Solidity types

### Memory {#memory}

Values that are only stored for the lifetime of a contract function's execution are called memory variables. Since these are not stored permanently on the blockchain, they are much cheaper to use.

Learn more about how the EVM stores data (Storage, Memory, and the Stack) in the Solidity docs.

### Environment variables {#environment-variables}

In addition to the variables you define on your contract, there are some special global variables. They are primarily used to provide information about the blockchain or current transaction.

Examples:

| **Prop** | **State variable** | **Description** |
|---|---|---|
| `block.timestamp` | uint256 | Current block epoch timestamp |
| `msg.sender` | address | Sender of the message (current call) |

## Functions {#functions}
```

In the most simplistic terms, functions can get information or set information in response to incoming transactions.

There are two types of function calls:

- `internal` – these don't create an EVM call
- Internal functions and state variables can only be accessed internally (i.e. from within the current contract or contracts deriving from it)
- `external` – these do create an EVM call
- External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works).

They can also be `public` or `private`

- `public` functions can be called internally from within the contract or externally via messages
- `private` functions are only visible for the contract they are defined in and not in derived contracts

Both functions and state variables can be made public or private

Here's a function for updating a state variable on a contract:

```
solidity // Solidity example function update_name(string value) public { dapp_name = value; }
```

- The parameter `value` of type `string` is passed into the function: `update_name`
- It's declared `public`, meaning anyone can access it
- It's not declared `view`, so it can modify the contract state

View functions {#view-functions}

These functions promise not to modify the state of the contract's data. Common examples are "getter" functions – you might use this to receive a user's balance for example.

```
solidity // Solidity example function balanceOf(address _owner) public view returns (uint256 _balance) { return ownerPizzaCount[_owner];
}
```

```python dappName: public(string)

@view @public def readName() -> string: return dappName ```

What is considered modifying state:

1. Writing to state variables.
2. Emitting events.
3. Creating other contracts.
4. Using `selfdestruct`.
5. Sending ether via calls.
6. Calling any function not marked `view` or `pure`.
7. Using low-level calls.
8. Using inline assembly that contains certain opcodes.

### Constructor functions {#constructor-functions}

`constructor` functions are only executed once when the contract is first deployed. Like`constructor` in many class-based programming languages, these functions often initialize state variables to their specified values.

```
solidity // Solidity example // Initializes the contract's data, setting the `owner` // to the address of the contract creator.
constructor() public { // All smart contracts rely on external transactions to trigger its functions. // `msg` is a global variable
that includes relevant data on the given transaction, // such as the address of the sender and the ETH value included in the
transaction. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/units-and-global-variables.html#block-and-transaction-properties
owner = msg.sender; }
~~~

```python

# Vyper example

@external def **init**(_beneficiary: address, _bidding_time: uint256): self.beneficiary = _beneficiary self.auctionStart = block.timestamp self.auctionEnd = self.auctionStart + _bidding_time ```

### Built-in functions {#built-in-functions}

In addition to the variables and functions you define on your contract, there are some special built-in functions. The most obvious example is:

- `address.send()` – Solidity
- `send(address)` – Vyper

These allow contracts to send ETH to other accounts.
~~~

## Writing functions {#writing-functions}

Your function needs:

- parameter variable and type (if it accepts parameters)
- declaration of internal/external
- declaration of pure/view/payable
- returns type (if it returns a value)

```solidity pragma solidity >=0.4.0 <=0.6.0;

contract ExampleDapp { string dapp_name; // state variable

```
// Called when the contract is deployed and initializes the value
constructor() public {
    dapp_name = "My Example dapp";
}

// Get Function
function read_name() public view returns(string) {
    return dapp_name;
}

// Set Function
function update_name(string value) public {
    dapp_name = value;
}
~~~

} ```

A complete contract might look something like this. Here the`constructor` function provides an initial value for the`dapp_name` variable.

Events and logs {#events-and-logs}

Events let you communicate with your smart contract from your frontend or other subscribing applications. When a transaction is mined, smart contracts can emit events and write logs to the blockchain that the frontend can then process.

Annotated examples {#annotated-examples}

These are some examples written in Solidity. If you'd like to play with the code, you can interact with them in Remix.

Hello world {#hello-world}

```solidity // Specifies the version of Solidity, using semantic versioning. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#pragma pragma solidity ^0.5.10;

// Defines a contract named `HelloWorld`. // A contract is a collection of functions and data (its state). // Once deployed, a contract resides at a specific address on the Ethereum blockchain. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html contract HelloWorld {

~~~
// Declares a state variable `message` of type `string`.
// State variables are variables whose values are permanently stored in contract storage.
// The keyword `public` makes variables accessible from outside a contract
// and creates a function that other contracts or clients can call to access the value.
string public message;

// Similar to many class-based object-oriented languages, a constructor is
// a special function that is only executed upon contract creation.
// Constructors are used to initialize the contract's data.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constructors
constructor(string memory initMessage) public {
    // Accepts a string argument `initMessage` and sets the value
    // into the contract's `message` storage variable).
    message = initMessage;
}

// A public function that accepts a string argument
// and updates the `message` storage variable.
function update(string memory newMessage) public {
    message = newMessage;
}
~~~

} ```

### Token {#token}

```solidity pragma solidity ^0.5.10;

contract Token { // An `address` is comparable to an email address - it's used to identify an account on Ethereum. // Addresses can represent a smart contract or an external (user) accounts. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/types.html#address address public owner;

~~~
// A `mapping` is essentially a hash table data structure.
```
```

```
// This `mapping` assigns an unsigned integer (the token balance) to an address (the token holder).
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/types.html#mapping-types
mapping (address => uint) public balances;

// Events allow for logging of activity on the blockchain.
// Ethereum clients can listen for events in order to react to contract state changes.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#events
event Transfer(address from, address to, uint amount);

// Initializes the contract's data, setting the `owner`
// to the address of the contract creator.
constructor() public {
    // All smart contracts rely on external transactions to trigger its functions.
    // `msg` is a global variable that includes relevant data on the given transaction,
    // such as the address of the sender and the ETH value included in the transaction.
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/units-and-global-variables.html#block-and-transaction-properties
    owner = msg.sender;
}

// Creates an amount of new tokens and sends them to an address.
function mint(address receiver, uint amount) public {
    // `require` is a control structure used to enforce certain conditions.
    // If a `require` statement evaluates to `false`, an exception is triggered,
    // which reverts all changes made to the state during the current call.
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/control-structures.html#error-handling-assert-require-revert-and-exceptions

    // Only the contract owner can call this function
    require(msg.sender == owner, "You are not the owner.");

    // Enforces a maximum amount of tokens
    require(amount < 1e60, "Maximum issuance exceeded");

    // Increases the balance of `receiver` by `amount`
    balances[receiver] += amount;
}

// Sends an amount of existing tokens from any caller to an address.
function transfer(address receiver, uint amount) public {
    // The sender must have enough tokens to send
    require(amount <= balances[msg.sender], "Insufficient balance.");

    // Adjusts token balances of the two addresses
    balances[msg.sender] -= amount;
    balances[receiver] += amount;

    // Emits the event defined earlier
    emit Transfer(msg.sender, receiver, amount);
}
```

} ```

Unique digital asset {#unique-digital-asset}

\```solidity pragma solidity ^0.5.10;

// Imports symbols from other files into the current contract. // In this case, a series of helper contracts from OpenZeppelin. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/layout-of-source-files.html#importing-other-source-files

import "../node_modules/@openzeppelin/contracts/token/ERC721/IERC721.sol"; import "../node_modules/@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol"; import "../node_modules/@openzeppelin/contracts/introspection/ERC165.sol"; import "../node_modules/@openzeppelin/contracts/math/SafeMath.sol";

// The `is` keyword is used to inherit functions and keywords from external contracts. // In this case,`CryptoPizza` inherits from the `IERC721` and `ERC165` contracts. // Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#inheritance contract CryptoPizza is IERC721, ERC165 { // Uses OpenZeppelin's SafeMath library to perform arithmetic operations safely. // Learn more: https://docs.openzeppelin.com/contracts/2.x/api/math#SafeMath using SafeMath for uint256;

```
// Constant state variables in Solidity are similar to other languages
// but you must assign from an expression which is constant at compile time.
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#constant-state-variables
uint256 constant dnaDigits = 10;
uint256 constant dnaModulus = 10 ** dnaDigits;
bytes4 private constant _ERC721_RECEIVED = 0x150b7a02;

// Struct types let you define your own type
// Learn more: https://solidity.readthedocs.io/en/v0.5.10/types.html#structs
struct Pizza {
    string name;
    uint256 dna;
}

// Creates an empty array of Pizza structs
Pizza[] public pizzas;

// Mapping from pizza ID to its owner's address
mapping(uint256 => address) public pizzaToOwner;

// Mapping from owner's address to number of owned token
mapping(address => uint256) public ownerPizzaCount;

// Mapping from token ID to approved address
```

```
mapping(uint256 => address) pizzaApprovals;

// You can nest mappings, this example maps owner to operator approvals
mapping(address => mapping(address => bool)) private operatorApprovals;

// Internal function to create a random Pizza from string (name) and DNA
function _createPizza(string memory _name, uint256 _dna)
    // The `internal` keyword means this function is only visible
    // within this contract and contracts that derive this contract
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#visibility-and-getters
    internal
    // `isUnique` is a function modifier that checks if the pizza already exists
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/structure-of-a-contract.html#function-modifiers
    isUnique(_name, _dna)
{
    // Adds Pizza to array of Pizzas and get id
    uint256 id = SafeMath.sub(pizzas.push(Pizza(_name, _dna)), 1);

    // Checks that Pizza owner is the same as current user
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/control-structures.html#error-handling-assert-require-revert-and-exceptions

    // note that address(0) is the zero address,
    // indicating that pizza[id] is not yet allocated to a particular user.

    assert(pizzaToOwner[id] == address(0));

    // Maps the Pizza to the owner
    pizzaToOwner[id] = msg.sender;
    ownerPizzaCount[msg.sender] = SafeMath.add(
        ownerPizzaCount[msg.sender],
        1
    );
}

// Creates a random Pizza from string (name)
function createRandomPizza(string memory _name) public {
    uint256 randDna = generateRandomDna(_name, msg.sender);
    _createPizza(_name, randDna);
}

// Generates random DNA from string (name) and address of the owner (creator)
function generateRandomDna(string memory _str, address _owner)
    public
    // Functions marked as `pure` promise not to read from or modify the state
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#pure-functions
    pure
    returns (uint256)
{
    // Generates random uint from string (name) + address (owner)
    uint256 rand = uint256(keccak256(abi.encodePacked(_str))) +
        uint256(_owner);
    rand = rand % dnaModulus;
    return rand;
}

// Returns array of Pizzas found by owner
function getPizzasByOwner(address _owner)
    public
    // Functions marked as `view` promise not to modify state
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/contracts.html#view-functions
    view
    returns (uint256[] memory)
{
    // Uses the `memory` storage location to store values only for the
    // lifecycle of this function call.
    // Learn more: https://solidity.readthedocs.io/en/v0.5.10/introduction-to-smart-contracts.html#storage-memory-and-the-stack
    uint256[] memory result = new uint256[](ownerPizzaCount[_owner]);
    uint256 counter = 0;
    for (uint256 i = 0; i < pizzas.length; i++) {
        if (pizzaToOwner[i] == _owner) {
            result[counter] = i;
            counter++;
        }
    }
    return result;
}

// Transfers Pizza and ownership to other address
function transferFrom(address _from, address _to, uint256 _pizzaId) public {
    require(_from != address(0) && _to != address(0), "Invalid address.");
    require(_exists(_pizzaId), "Pizza does not exist.");
    require(_from != _to, "Cannot transfer to the same address.");
    require(_isApprovedOrOwner(msg.sender, _pizzaId), "Address is not approved.");

    ownerPizzaCount[_to] = SafeMath.add(ownerPizzaCount[_to], 1);
    ownerPizzaCount[_from] = SafeMath.sub(ownerPizzaCount[_from], 1);
    pizzaToOwner[_pizzaId] = _to;

    // Emits event defined in the imported IERC721 contract
    emit Transfer(_from, _to, _pizzaId);
    _clearApproval(_to, _pizzaId);
}

/**
 * Safely transfers the ownership of a given token ID to another address
 * If the target address is a contract, it must implement `onERC721Received`,
 * which is called upon a safe transfer, and return the magic value
 * `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`;
```

```
     * otherwise, the transfer is reverted.
    */
    function safeTransferFrom(address from, address to, uint256 pizzaId)
        public
    {
        // solium-disable-next-line arg-overflow
        this.safeTransferFrom(from, to, pizzaId, "");
    }

    /**
     * Safely transfers the ownership of a given token ID to another address
     * If the target address is a contract, it must implement `onERC721Received`,
     * which is called upon a safe transfer, and return the magic value
     * `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`;
     * otherwise, the transfer is reverted.
     */
    function safeTransferFrom(
        address from,
        address to,
        uint256 pizzaId,
        bytes memory _data
    ) public {
        this.transferFrom(from, to, pizzaId);
        require(_checkOnERC721Received(from, to, pizzaId, _data), "Must implement onERC721Received.");
    }

    /**
     * Internal function to invoke `onERC721Received` on a target address
     * The call is not executed if the target address is not a contract
     */
    function _checkOnERC721Received(
        address from,
        address to,
        uint256 pizzaId,
        bytes memory _data
    ) internal returns (bool) {
        if (!isContract(to)) {
            return true;
        }

        bytes4 retval = IERC721Receiver(to).onERC721Received(
            msg.sender,
            from,
            pizzaId,
            _data
        );
        return (retval == _ERC721_RECEIVED);
    }

    // Burns a Pizza - destroys Token completely
    // The `external` function modifier means this function is
    // part of the contract interface and other contracts can call it
    function burn(uint256 _pizzaId) external {
        require(msg.sender != address(0), "Invalid address.");
        require(_exists(_pizzaId), "Pizza does not exist.");
        require(_isApprovedOrOwner(msg.sender, _pizzaId), "Address is not approved.");

        ownerPizzaCount[msg.sender] = SafeMath.sub(
            ownerPizzaCount[msg.sender],
            1
        );
        pizzaToOwner[_pizzaId] = address(0);
    }

    // Returns count of Pizzas by address
    function balanceOf(address _owner) public view returns (uint256 _balance) {
        return ownerPizzaCount[_owner];
    }

    // Returns owner of the Pizza found by id
    function ownerOf(uint256 _pizzaId) public view returns (address _owner) {
        address owner = pizzaToOwner[_pizzaId];
        require(owner != address(0), "Invalid Pizza ID.");
        return owner;
    }

    // Approves other address to transfer ownership of Pizza
    function approve(address _to, uint256 _pizzaId) public {
        require(msg.sender == pizzaToOwner[_pizzaId], "Must be the Pizza owner.");
        pizzaApprovals[_pizzaId] = _to;
        emit Approval(msg.sender, _to, _pizzaId);
    }

    // Returns approved address for specific Pizza
    function getApproved(uint256 _pizzaId)
        public
        view
        returns (address operator)
    {
        require(_exists(_pizzaId), "Pizza does not exist.");
        return pizzaApprovals[_pizzaId];
    }

    /**
     * Private function to clear current approval of a given token ID
     * Reverts if the given address is not indeed the owner of the token
     */
    function _clearApproval(address owner, uint256 _pizzaId) private {
        require(pizzaToOwner[_pizzaId] == owner, "Must be pizza owner.");
        require(_exists(_pizzaId), "Pizza does not exist.");
        if (pizzaApprovals[_pizzaId] != address(0)) {
            pizzaApprovals[_pizzaId] = address(0);
        }
    }

    /*
     * Sets or unsets the approval of a given operator
     * An operator is allowed to transfer all tokens of the sender on their behalf
     */
    function setApprovalForAll(address to, bool approved) public {
        require(to != msg.sender, "Cannot approve own address");
        operatorApprovals[msg.sender][to] = approved;
        emit ApprovalForAll(msg.sender, to, approved);
    }

    // Tells whether an operator is approved by a given owner
    function isApprovedForAll(address owner, address operator)
        public
        view
        returns (bool)
    {
        return operatorApprovals[owner][operator];
    }

    // Takes ownership of Pizza - only for approved users
    function takeOwnership(uint256 _pizzaId) public {
        require(_isApprovedOrOwner(msg.sender, _pizzaId), "Address is not approved.");
        address owner = this.ownerOf(_pizzaId);
        this.transferFrom(owner, msg.sender, _pizzaId);
    }

    // Checks if Pizza exists
    function _exists(uint256 pizzaId) internal view returns (bool) {
        address owner = pizzaToOwner[pizzaId];
        return owner != address(0);
    }

    // Checks if address is owner or is approved to transfer Pizza
    function _isApprovedOrOwner(address spender, uint256 pizzaId)
        internal
        view
        returns (bool)
    {
        address owner = pizzaToOwner[pizzaId];
        // Disable solium check because of
        // https://github.com/duaraghav8/Solium/issues/175
        // solium-disable-next-line operator-whitespace
        return (spender == owner ||
            this.getApproved(pizzaId) == spender ||
            this.isApprovedForAll(owner, spender));
    }

    // Check if Pizza is unique and doesn't exist yet
    modifier isUnique(string memory _name, uint256 _dna) {
        bool result = true;
        for (uint256 i = 0; i < pizzas.length; i++) {
            if (
                keccak256(abi.encodePacked(pizzas[i].name)) ==
                keccak256(abi.encodePacked(_name)) &&
                pizzas[i].dna == _dna
            ) {
                result = false;
            }
        }
        require(result, "Pizza with such name already exists.");
        _;
    }

    // Returns whether the target address is a contract
    function isContract(address account) internal view returns (bool) {
        uint256 size;
        // Currently there is no better way to check if there is a contract in an address
        // than to check the size of the code at that address.
        // See https://ethereum.stackexchange.com/a/14016/36603
        // for more details about how this works.
        // TODO Check this again before the Serenity release, because all addresses will be
        // contracts then.
        // solium-disable-next-line security/no-inline-assembly
        assembly {
            size := extcodesize(account)
        }
        return size > 0;
    }
```

} ```

Further reading {#further-reading}

Check out Solidity and Vyper's documentation for a more complete overview of smart contracts:

- Solidity
- Vyper

Related topics {#related-topics}

- Smart contracts
- Ethereum Virtual Machine

Related tutorials {#related-tutorials}

- Downsizing contracts to fight the contract size limit– *Some practical tips for reducing the size of your smart contract.*
- Logging data from smart contracts with events– *An introduction to smart contract events and how you can use them to log data.*
- Interact with other contracts from Solidity– *How to deploy a smart contract from an existing contract and interact with it.*