---

title: Introduction to smart contracts description: An overview of smart contracts, focussing on their unique characteristics and limitations. lang: en

---

What is a smart contract? {#what-is-a-smart-contract}

A "smart contract" is simply a program that runs on the Ethereum blockchain. It's a collection of code (its functions) and data (its state) that resides at a specific address on the Ethereum blockchain.

Smart contracts are a type of Ethereum account. This means they have a balance and can be the target of transactions. However they're not controlled by a user, instead they are deployed to the network and run as programmed. User accounts can then interact with a smart contract by submitting transactions that execute a function defined on the smart contract. Smart contracts can define rules, like a regular contract, and automatically enforce them via the code. Smart contracts cannot be deleted by default, and interactions with them are irreversible.

Prerequisites {#prerequisites}

If you're just getting started or looking for a less technical introduction, we recommend our introduction to smart contracts.

Make sure you've read up on accounts, transactions and the Ethereum virtual machine before jumping into the world of smart contracts.

A digital vending machine {#a-digital-vending-machine}

Perhaps the best metaphor for a smart contract is a vending machine, as described by Nick Szabo. With the right inputs, a certain output is guaranteed.

To get a snack from a vending machine:

```
money + snack selection = snack dispensed
```

This logic is programmed into the vending machine.

A smart contract, like a vending machine, has logic programmed into it. Here's a simple example of how this vending machine would look if it were a smart contract written in Solidity:

```solidity pragma solidity 0.8.7;

contract VendingMachine {

// Declare state variables of the contract
address public owner;
mapping (address => uint) public cupcakeBalances;

// When 'VendingMachine' contract is deployed:
// 1. set the deploying address as the owner of the contract
// 2. set the deployed smart contract's cupcake balance to 100
constructor() {
    owner = msg.sender;
    cupcakeBalances[address(this)] = 100;
}

// Allow the owner to increase the smart contract's cupcake balance
function refill(uint amount) public {
    require(msg.sender == owner, "Only the owner can refill.");
    cupcakeBalances[address(this)] += amount;
}

// Allow anyone to purchase cupcakes
function purchase(uint amount) public payable {
    require(msg.value >= amount * 1 ether, "You must pay at least 1 ETH per cupcake");
    require(cupcakeBalances[address(this)] >= amount, "Not enough cupcakes in stock to complete this purchase");
    cupcakeBalances[address(this)] -= amount;
    cupcakeBalances[msg.sender] += amount;
}
```

} ```

Like how a vending machine removes the need for a vendor employee, smart contracts can replace intermediaries in many industries.

Permissionless {#permissionless}

Anyone can write a smart contract and deploy it to the network. You just need to learn how to code in a smart contract language, and have enough ETH to deploy your contract. Deploying a smart contract is technically a transaction, so you need to pay gas in the same way you need to pay gas for a simple ETH transfer. However, gas costs for contract deployment are far higher.

Ethereum has developer-friendly languages for writing smart contracts:

- Solidity
- Vyper

More on languages

However, they must be compiled before they can be deployed so that Ethereum's virtual machine can interpret and store the contract. More on compilation

Composability {#composability}

Smart contracts are public on Ethereum and can be thought of as open APIs. This means you can call other smart contracts in your own smart contract to greatly extend what's possible. Contracts can even deploy other contracts.

Learn more about smart contract composability.

Limitations {#limitations}

Smart contracts alone cannot get information about "real-world" events because they can't retrieve data from off-chain sources. This means they can't respond to events in the real world. This is by design. Relying on external information could jeopardise consensus, which is important for security and decentralization.

However, it is important for blockchain applications to be able to use off-chain data. The solution is oracles which are tools that ingest off-chain data and make it available to smart contracts.

Another limitation of smart contracts is the maximum contract size. A smart contract can be a maximum of 24KB or it will run out of gas. This can be circumnavigated by using The Diamond Pattern.

Multisig contracts {#multisig}

Multisig (multiple-signature) contracts are smart contract accounts that require multiple valid signatures to execute a transaction. This is very useful for avoiding single points of failure for contracts holding substantial amounts of ether or other tokens. Multisigs also divide responsibility for contract execution and key management between multiple parties and prevent the loss of a single private key leading to irreversible loss of funds. For these reasons, multisig contracts can be used for simple DAO governance. Multisigs require N signatures out of M possible acceptable signatures (where $N \leq M$, and $M > 1$) in order to execute. `N = 3, M = 5` and `N = 4, M = 7` are commonly used. A 4/7 multisig requires four out of seven possible valid signatures. This means the funds are still retrievable even if three signatures are lost. In this case, it also means that the majority of key-holders must agree and sign in order for the contract to execute.

Smart contract resources {#smart-contract-resources}

OpenZeppelin Contracts - *Library for secure smart contract development.*

- openzeppelin.com/contracts/

- GitHub
- Community Forum

Further reading {#further-reading}

- Coinbase: What is a smart contract?
- Chainlink: What is a smart contract?
- Video: Simply Explained - Smart Contracts