A shared vault and a new asset

From many depositors to stablecoins · Sessions 10–11 · 5 October 2026

The course site and the achievements

cryptoassets2026.bruschi.dev: slides of every session, and an achievement board.

- Each address that joins an activity on the test network earns **badges**.
- ★ **In-Class**: done within a few hours of the session. ✓ **Completed**: done later.
- No names, no ranking: you are identified by your **address**.

At the end you will be able to **claim** an address by signing a message with its key.

Where we left off

The piggy bank is an **administrator with a memory**.

- It sleeps until someone calls it, then wakes up and reacts.
- Its memory: `depositTime`, `depositor`, `alreadyDeposited`.
- `receive()` records the deposit; `withdraw()` checks **who** and **when**.

It serves **one** deposit, once.

The piggy bank of session 9

```
contract PiggyBank {
    uint256 depositTime;
    address depositor;
    bool alreadyDeposited=false;

    receive() external payable {
        require(alreadyDeposited==false, "the vault is already taken");
        alreadyDeposited = true;
        depositTime = block.timestamp;
        depositor = msg.sender;
    }

    function withdraw() external {
        require(msg.sender == depositor, "you are not the depositor!");
        require(block.timestamp >= depositTime + 3 minutes, "not enough time elapsed yet!");
        payable(msg.sender).transfer((address(this).balance));
    }
}
```

Code that is meant to be read

Ordinary software: you trust its authors and never look at the code.

A smart contract is also a **contract**, somehow in the legal sense of the word:

- it is published so that anyone can **read it before** interacting;
- you may not read it every time, but you **can**, or ask someone who can.
- it is **binding**: the code compels the administrator to act as written, though the guarantee comes from the network rather than from courts.

One contract per deposit?

Each piggy bank serves one deposit. After the withdrawal it stays on the chain, **unusable**.

To offer the service to everyone, we would like **one contract**, deployed once, that handles **many depositors** at the same time.

One vault, many depositors

The contract holds all the ether in **a single balance**, and keeps a **table** of who is owed what.

Address	Deposited
Alice	1.0 ETH
Bob	2.0 ETH
Carol	4.0 ETH
Contract balance	**7.0 ETH**

When Carol asks for her money, the administrator looks up **her row** and sends exactly that.

Mappings

In Solidity a table with a unique key column is a **mapping**.

```
mapping(address => uint256) public balances;
```

- Give it an address, it returns a number.
- Think of it as a table with **every possible address**, all set to **0**.
- `balances[msg.sender]` is the row of whoever is calling.

`uint256`: an unsigned integer of 256 bits. `public` lets anyone read it from the explorer.

A first version

```
contract MultiVault {
    mapping(address => uint256) public balances;

    receive() external payable {
        // in the table, at the row belonging to msg.sender,
        // write the amount received
        balances[msg.sender] = msg.value;
    }
}
```

What happens to the ether sent to this contract?

Getting the money out

```
function withdraw() external {
    payable(msg.sender).transfer(balances[msg.sender]);
}
```

- Not `address(this).balance`: that is **everyone's** money.
- Not `msg.value`: a withdrawal sends no ether.
- The caller's own row: `balances[msg.sender]`.

A function that is not `payable` **refuses** any ether sent with the call.

Two deposits, one row

Step	`balances[Alice]`	Contract balance
Alice sends 1 ETH	1 ETH	1 ETH
Alice sends 1 ETH again	**1 ETH**	2 ETH

`=` **overwrites** the row. Deposits must be **added**:

```
balances[msg.sender] += msg.value;
```

A freezing time

A second mapping remembers **when** each address last deposited.

```
mapping(address => uint256) public depositTimes;
uint256 constant freezingTime = 5 minutes;

receive() external payable {
    balances[msg.sender] += msg.value;
    depositTimes[msg.sender] = block.timestamp;
}
```

`constant` : a name for a fixed value, written into the code at compilation. It takes no storage and costs nothing to read.

Withdrawing twice

```
function withdraw() external {
    require(block.timestamp > depositTimes[msg.sender] + freezingTime,
            "you're too early");
    payable(msg.sender).transfer(balances[msg.sender]);
}
```

After the freezing time, Alice can call `withdraw` **again and again**: her row is never reset.

The balance must be **set to zero** when it is paid out.

One clock per depositor

Alice deposits on **Monday**, then again on **Wednesday**. Freezing time: one week.

- When can she withdraw the Monday deposit?
- As written: **next Wednesday**. Each deposit resets her clock, and the whole balance waits.

Releasing each deposit separately is possible: the contract would need to record deposits one by one.

The version deployed in class

```
contract MultiVault {
    mapping(address=>uint256) public balances;
    mapping(address=>uint256) public depositTimes;
    uint256 constant freezingTime = 5 minutes;

    receive() external payable {
        balances[msg.sender] += msg.value;
        depositTimes[msg.sender] = block.timestamp;
    }

    function withdraw() external {
        require(block.timestamp > depositTimes[msg.sender] +
                        freezingTime, "you're too early");
        balances[msg.sender] = 0;
        payable(msg.sender).transfer(balances[msg.sender]);
    }
}
```

Nothing came back

The withdrawals **succeeded**, and returned **zero**.

Line	`balances[Alice]`	Sent
`balances[msg.sender] = 0;`	0	—
`transfer(balances[msg.sender])`	0	**0 ETH**

The deposits, about 0.26 test ETH, are locked in the contract **forever**: no function can move them!

The fix: read, reset, send

```
function withdraw() external {
    require(block.timestamp > depositTimes[msg.sender] + freezingTime,
            "you're too early");
    uint256 ethToBeGivenBack = balances[msg.sender];
    balances[msg.sender] = 0;
    payable(msg.sender).transfer(ethToBeGivenBack);
}
```

Reset **before** sending: if the payment called back into the contract, the row would already be empty.

A friend who reads the code

Asked "should I send ether to this contract?", a free LLM answered in seconds:

> No. You will lose your money permanently: the balance is zeroed before the transfer.

- An old objection: public code helps only those who can read it.
- Today anyone can ask a tool to read it, and **check** the answer against the code.

This is possible only because the code is **public** and **verified**.

An escape hatch, with a penalty

Let depositors leave **before** the freezing time, at a cost.

```
uint256 constant penaltyPercentage = 10;

function forcedWithdraw() external {
    uint256 ethToBeGivenBack =
        (balances[msg.sender] * (100 - penaltyPercentage)) / 100;
    balances[msg.sender] = 0;
    payable(msg.sender).transfer(ethToBeGivenBack);
}
```

The contract does not **forbid** leaving early. It **discourages** it.

Where does the penalty go?

Nowhere, for now: it stays **locked** in the contract.

Is that value destroyed?

- Burning banknotes leaves fewer claims on the same goods: everyone else's money is worth a little more.
- A company that buys back and cancels shares raises the value of the remaining ones.

Ethereum does the same: part of every fee, the **base fee**, is **burned**.

What to do with the penalties?

Proposal	Who decides	What users must trust
Keep them locked	nobody	nothing
Pay the operator of the service	the deployer	that the rule is fair
Share them among those who waited	the code	nothing
Give them to a charity	one appointed person	that person
Let depositors propose and **vote**	the depositors	the voting rules

The last option is a **decentralized autonomous organization** (DAO): funds governed by rules written in code.

A power, written in code

```
uint256 public penaltiesCollected;

function forcedWithdraw() external {
    uint256 ethToBeGivenBack =
        (balances[msg.sender] * (100 - penaltyPercentage)) / 100;
    penaltiesCollected += balances[msg.sender] - ethToBeGivenBack;
    balances[msg.sender] = 0;
    payable(msg.sender).transfer(ethToBeGivenBack);
}

function transferPenalties(address recipient) external {
    require(msg.sender == ADMIN, "You're not the admin!");
    uint256 amount = penaltiesCollected;
    penaltiesCollected = 0;
    payable(recipient).transfer(amount);
}
```

Checks and balances

The administrator gets **exactly** the power the code grants, and no more.

- Only the penalties, never the deposits.
- Possibly only to a **list of recipients** fixed in advance.
- The administrator could be **another contract**: a committee that votes, a majority of signatures, a way to vote the administrator out.

Every familiar arrangement of trust can be reproduced; others become possible.

A superior form of money?

What we can do with ether:

- open an account in seconds, with no one’s permission;
- send any amount, anywhere, for a small fee;
- build financial mechanisms like the vault in an afternoon.

So why don’t we pay for coffee with it?

Why not pay for coffee in ether?

Reason	Where it stands
Speed	blocks every 12 seconds; layer-2 chains confirm in about a second
Cost	layer-2 fees are small; card fees exist too, hidden in the price
Volatility	the café is exposed to the price of ether
Adoption	few accept it; but adding one more method costs little
Compliance	taxes, monetary policy, regulation
Technical risk	mistakes are yours, and they can be permanent
Irreversibility	no chargeback
Privacy	the café's address reveals its sales to anyone

First, volatility

A café needs money worth the same tomorrow, to pay rent and suppliers in euros.

Could we have an asset that:

- lives in MetaMask and moves like ether;
- can be used in contracts like the vault;
- is worth **one euro**, or **one dollar**?

One blockchain per asset?

Bitcoin has bitcoin; Ethereum has ether. Should a dollar asset get its own chain?

- A new chain needs its own **validators** and its own **security**.
- Its security would be split from Ethereum's, and much weaker.

Instead: create **new assets on Ethereum itself**, with a contract.

A token is a table of balances

Bitcoin’s administrator kept a table of who owns what. A contract can do the same for a **new asset**.

Address	Tokens
Alice	10
Bob	15
Carol	20 → **17**
Dave	0 → **3**

Carol asks the contract to move 3 of her tokens to Dave. It updates two rows.

The token written in class

```
contract AlisCoffeeToken {
    mapping(address => uint256) public balances;

    constructor() {
        balances[0x4a414ebE9384C36d10d2Ee6d4785C480322dCb19] = 1000;
    }

    function transfer(address _dest, uint256 _amount) external {
        require(balances[msg.sender] >= _amount, "not enough tokens");
        balances[msg.sender] -= _amount;
        balances[_dest] += _amount;
    }
}
```

Where do tokens come from?

Without the constructor, the contract would start with **no tokens at all**.

- The **constructor** runs once, when the contract is created.
- Here it gives **1,000 tokens** to one address. No function creates or destroys more.
- The supply is fixed: tokens can only change hands.

To obtain some, persuade a holder to transfer them: as a gift, or in exchange for something.

Checksummed addresses

```
0x4a414ebE9384C36d10d2Ee6d4785C480322dCb19
```

- The mix of upper and lower case is a **checksum**.
- A copying mistake in a digit almost certainly breaks the case pattern.
- The compiler rejects an address literal with a wrong checksum.

Moving tokens

On the contract's page in Blockscout:

1. **Read** `balances` for any address: anyone can check anyone's holdings.
2. **Write** `transfer(_dest, _amount)`, signed with MetaMask.
3. The fee is paid in **ether**; the tokens are a separate asset.

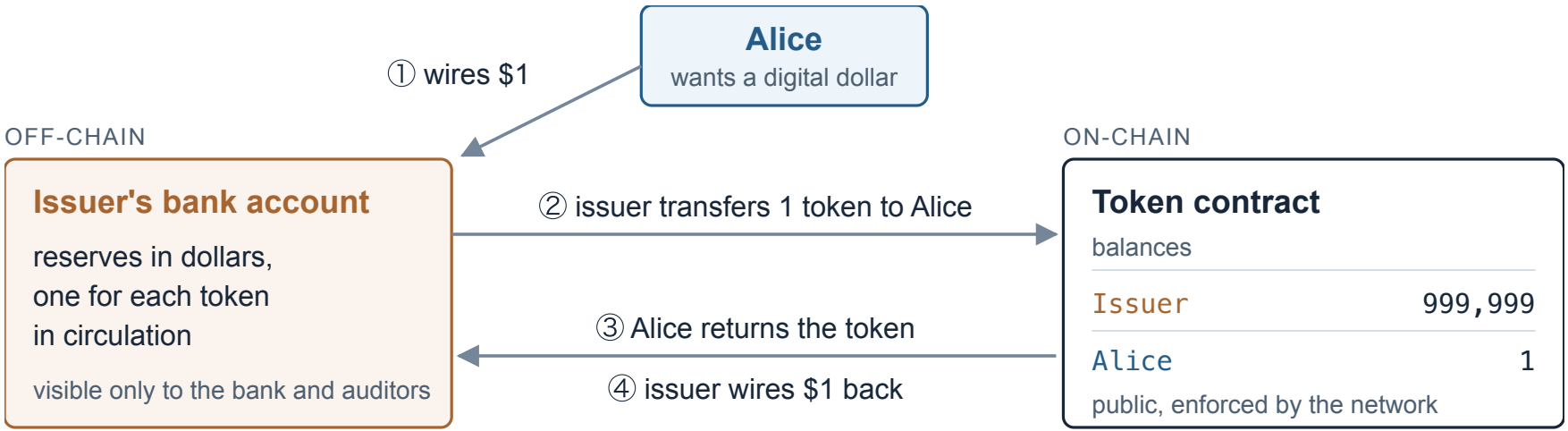
The same code, a different asset

Anyone can deploy the same code and create another 1,000 tokens.

They would be a **different** asset: a token is identified by its **contract address**.

The 1,000 tokens of this contract on Sepolia can never become 1,001.

From a token to a stablecoin



The contract cannot see the bank account: the link between token and dollar is the issuer's promise.

Why the price stays at one dollar

If everyone trusts the issuer to redeem at $1:

Market price	What a trader does	Effect
below $1, e.g. $0.50	buys tokens, redeems them for $1	price rises
above $1, e.g. $2	gets tokens from the issuer for $1, sells them	price falls

Arbitrage anchors the price to one dollar, **as long as** the promise is credible.

What a stablecoin changes

- A digital dollar that moves **like ether**, in any wallet.
- Usable in **contracts** like the ones we wrote.
- No exposure to the price of ether.

But the dollar is not on the chain. The token is worth what the **issuer's promise** is worth.

Next session

- Tokens shown in **MetaMask**, like ether.
- Why our token is not quite a standard one: **ERC-20**.
- Who can issue, freeze or destroy tokens, and how to check it in the code.