Smart contracts in practice

Ping and a piggy bank · Session 9 · 1 October 2026

Where we left off

- Bitcoin: simple **accounting rules** running on a **computer nobody controls**.
- Ethereum: the same kind of computer, open to **any program**.
- It has a native asset, **ether**, accounted for in a table of accounts.

Address	Balance	Nonce	Code
Alice	10 ETH	3	—
Bob	0 ETH	0	—

The life of a transfer

To send 3 ETH to Bob, Alice:

1. writes a transaction: recipient, value, **nonce**;
2. **signs** it with her private key;
3. sends it to the network, where it waits in the **mempool**;
4. a block producer includes it in a **block**;
5. once executed, Alice's balance decreases and Bob's increases.

Without a valid signature the transaction is rejected.

Hands-on: receiving test ether

- The lecturer’s new wallet was funded from another wallet.
- From it, 0.1–0.2 test ETH went to each submitted address.
- Recipients then passed some on to classmates.

MetaMask may take a while to show a new balance. A block explorer reads the chain directly.

Block explorers

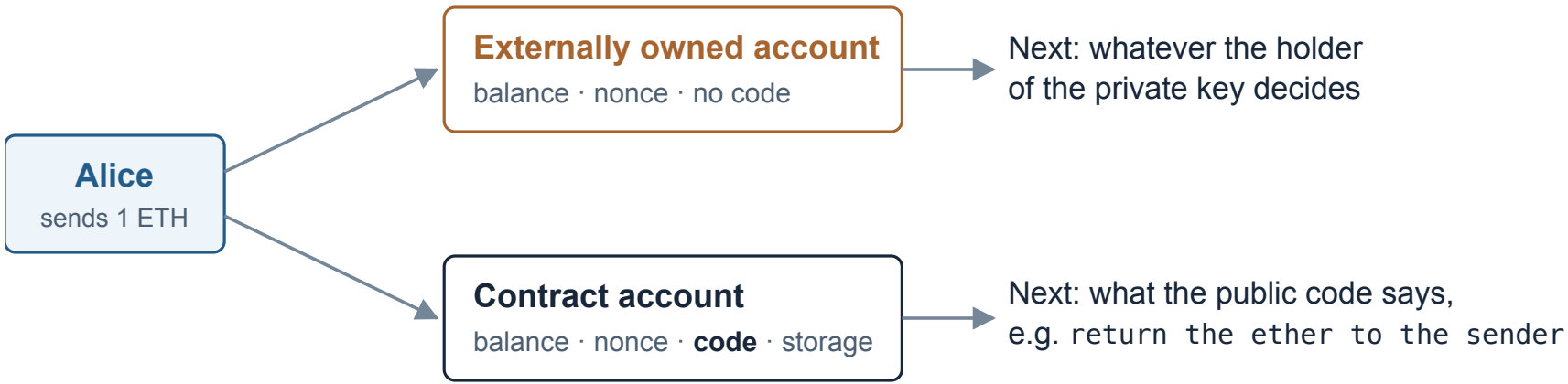
A **block explorer** keeps its own copy of the chain and presents it as web pages.

- Search any **address**: balance and full transaction history.
- Search any **transaction**: sender, recipient, value, fee, status.
- Follow links from one address to the next.

We use **Blockscout** (eth-sepolia.blockscout.com). Etherscan is another widely used explorer.

Everything is public: anyone can follow where anyone’s funds went.

Accounts with code



An address is the hash of a public key. A contract's address corresponds to **no key**: only its code acts on its behalf.

A financial ping

Imagine a bank account with a strange behavior:

whatever you send to it, it immediately sends back to you.

- Send it ten cents, receive ten cents.
- Send it a billion, receive a billion.

Useless, but a clean first example of a contract: the outcome depends only on its **code**.

Ping in Solidity

```
pragma solidity >=0.8.2 <0.9.0;

contract Ping {

    receive() external payable {
        // send all the ethers
        // back to the sender
        payable(msg.sender).transfer(msg.value);
    }

}
```

The contract as written and deployed in class, verified on Blockscout.

Reading the code

Code	Meaning
`pragma solidity >=0.8.2 <0.9.0;`	compiler versions the code is written for
`contract Ping { … }`	the program that will become an account
`receive() external payable`	runs when the contract receives plain ether
`msg.sender`	address that sent the current call
`msg.value`	amount of ether sent with it
`payable(…).transfer(x)`	send `x` wei to that address

Reading code like this does not require being a programmer.

Deploying a contract

Deployment transaction

to: (empty)

value: 0

data: `0x6080604052…`

compiled Ping bytecode



signed by the deployer, who pays the fee



State after execution

Address	Balance	Nonce	Code
Deployer	− fee	n + 1	—
Bob	2 ETH	6	—
`0x2768…3eCD`	0 ETH	1	**Ping code**

The new address is computed from the deployer's address and nonce.
No private key corresponds to it: only the code controls it.

Why deployment goes through a wallet

- Deployment is a **transaction**: someone must sign it and pay the **fee**.
- The deployer's address is recorded, and a contract **may** grant it special roles.

Ping grants none. After deployment its author has **no more power over it** than anyone else.

A contract on the block explorer

The new address looks like any other, with one difference: a **Contract** tab.

- It shows the **source code**, once the author has verified it.
- Verification proves that the source compiles to the bytecode stored on-chain.
- Before interacting, anyone can read what will happen to their ether.

Nobody can change this behavior, and no authority can intercept the funds in transit.

All in on Ping

1. Paste the Ping address in MetaMask's *Send*.
2. Press **Max**: MetaMask keeps back enough for the fee.
3. Confirm, then watch the balance on Blockscout.

   The full amount comes back in the same transaction. Only the fee is gone.

Could a bank offer this?

Possibly: with a banking license, or by persuading a bank to run it.

- It might **refuse**.
- Even if it agreed, users would have to **trust** it to keep sending money back.

On Ethereum:

- the guarantee that funds come back is the **same** that protects your keys;
- **anyone** can set it up, during a 9 a.m. lecture.

What if the code has a bug?

- The code at an address **cannot be changed**.
- A new contract can be deployed, at a **new address**.
- Contracts can be designed **modularly**: one contract routes calls to replaceable modules.

This is how **upgradeable** contracts are built. Users must then trust whoever can switch modules.

A piggy bank against Black Friday

Black Friday deals tempt us to spend everything. Commit in advance:

- **deposit** your money a few weeks before;
- **withdraw** it only after the date has passed.

As a traditional service this would need trust in the operator. Publishing a Python script would not help: nobody can see that it is the code actually running.

No waiting inside a contract

A contract cannot `sleep` for two weeks and then act.

- Code runs **only when called**, and runs to completion within one transaction.
- The contract is a sleeping administrator, woken by each call.

So the piggy bank needs **two entry points**:

- `receive()` when ether arrives;
- `withdraw()` when someone asks for it back.

First attempt

```
contract PiggyBank {

    receive() external payable {
    }

    function withdraw() external {
        payable(msg.sender).transfer(address(this).balance);
    }

}
```

- `msg.value` during `withdraw` is what that call sends, usually 0.
- `address(this).balance` is everything the contract holds.

Anyone can withdraw at any time. **What is missing?**

Memory and time

The administrator needs a notepad: **state variables**, kept between calls.

```
uint256 depositTime;

receive() external payable {
    depositTime = block.timestamp;
}

function withdraw() external {
    require(block.timestamp >= depositTime + 2 weeks);
    payable(msg.sender).transfer(address(this).balance);
}
```

`block.timestamp` is the time of the current block. `require` stops execution if its condition is false.

Who may withdraw?

As written, **anyone** can claim the money after two weeks.

```
address depositor;

receive() external payable {
    depositTime = block.timestamp;
    depositor = msg.sender;
}

function withdraw() external {
    require(msg.sender == depositor);
    require(block.timestamp >= depositTime + 2 weeks);
    payable(msg.sender).transfer(address(this).balance);
}
```

The bug found in class

Step	depositor	balance
Alice deposits 1 ETH	Alice	1 ETH
Mallory deposits 0.001 ETH	**Mallory**	1.001 ETH
Two weeks later, Mallory withdraws	Mallory	**0**

A second deposit **overwrites** the depositor, and with it the right to the whole balance.

The version deployed in class

```
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));
    }
}
```

When a requirement fails

If a `require` fails, the **whole transaction is reverted**.

- The state is left exactly as if the transaction had never run.
- Any ether sent with it stays with the sender.
- The sender still **pays the fee** for the computation performed.

A transaction either completes entirely or has no effect, apart from the fee.

Interacting through Blockscout

On the contract's page, *Contract → Read/Write contract*:

1. **Connect** the wallet.
2. **Simulate** a call: `withdraw` before the time limit shows *not enough time elapsed yet!*
3. Send it anyway: MetaMask warns that the transaction is likely to fail.
4. After three minutes, the simulation succeeds. Signing it returns the deposit.

A first example of decentralized finance

- A financial mechanism written in a few lines and deployed in minutes.
- Its behavior is **public**, **fixed**, and enforced by the network.
- **Anyone** can offer it, without asking permission.

Next sessions: more complex mechanisms, and where a yield could come from.