Blockchains: An Introduction to Blockchain Technology

Lecture 4, Ethereum & Smart Contracts:
The Account Model, the EVM, Solidity, and Reentrancy

Lecture Notes

Abstract

Lectures 1–3 built the stack from the bottom up: the two cryptographic primitives, Bitcoin's UTXO ledger as the simplest useful shared state, and the consensus protocols that let mutually distrusting parties agree on an ever-growing log. This lecture is about what that log lets you *build*. Ethereum turns "agree on a ledger" into "agree on the execution of arbitrary programs." We work through, in order: why Bitcoin Script cannot express a rule that spans many coins, and why that pushes us to a different model; Ethereum's **account model** and *world state*, with its two kinds of account and its per-contract storage array; the **Ethereum Virtual Machine** (EVM) as a metered stack machine, and **gas** as the meter, including the **EIP-1559** fee market and why part of every fee is burned; **Solidity**, the dominant contract language, covering types, visibility, the four data locations, ABI encoding, event logs, contract creation, and `delegatecall`-based proxies; and finally a security case study built around one bug, **reentrancy**: the vulnerable withdrawal pattern, the *checks–effects–interactions* fix, the reentrancy-guard mutex, the cross-function and read-only variants, and the 2016 **DAO hack** that made all of this famous. Consensus (Lecture 3) and the peer-to-peer network (Lecture 1) are assumed throughout: here we take for granted that roughly every twelve seconds one agreed block of transactions is appended and executed by every node identically.

Contents

1 From a ledger to a world computer

1.1 What Bitcoin Script cannot say

Recall the Bitcoin model (Lecture 2). Every coin is an unspent transaction output (UTXO) carrying a locking script, its *ScriptPK*, and that script checks the conditions under which *that one output* may be spent. The validator runs `ScriptSig | ScriptPK` on a single stack[1] and authorises the input if and only if it terminates with `true`. Crucially, the script sees *only* the input being spent and the transaction spending it. It has no view of the rest of the UTXO set.

Two things are therefore hard to express:

- **State that persists across a multi-stage interaction.** Each script execution is a fresh, isolated evaluation; there is no place to keep a running variable between transactions except by threading it manually through a chain of purpose-built outputs.
- **A rule that quantifies over many outputs at once.** "At most 2 BTC may leave this wallet per day, across *all* its coins" is a global constraint. A script that runs once per input, blind to the other inputs and to every other UTXO, cannot enforce it.

Example 1.1 (A wallet-wide spending limit)**.** Your wallet holds 100 UTXOs, all locked to your key. Desired policy: the wallet may spend at most 2 BTC in total per day. Bitcoin Script evaluates one UTXO at a time and cannot see or limit the others, so the policy is inexpressible

[1] Since Bitcoin 0.3.x the two scripts are actually evaluated *separately*: `ScriptSig` runs first, the resulting stack is copied into a fresh evaluation of `ScriptPK`, and only then is the top of the stack checked. This prevents `ScriptSig` from injecting control flow into `ScriptPK`. The pedagogical point—each script sees only its own input—is unaffected.

at the script level. Enforcing it requires logic *outside* the chain (in the wallet software), which any holder of the key can bypass.

1.2 A worked motivation: decentralized DNS

Consider putting the domain-name system on a blockchain: a public mapping `google.com` $\mapsto$ IP address that no registrar controls. We want three operations:

- `Name.new(OwnerAddr, DomainName)`, an intent to register a name;
- `Name.update(DomainName, newVal, newOwner, Sig)`, change the record or transfer ownership, authorised by the current owner's signature;
- `Name.lookup(DomainName)`, read the current value.

A broken attempt in the UTXO model. Represent each registration as a UTXO whose `ScriptPK` is the standard pay-to-public-key-hash lock

```
DUP HASH160 <OwnerPKHash> EQUALVERIFY CHECKSIG,
```

so that only the holder of the matching key can "spend" (update) it. Here `<OwnerPKHash>` is the 20-byte hash of the owner's public key, not the Base58Check address. Updating a record is then a transaction that spends the old registration UTXO and creates a new one.

Why it does not work. The contract we actually need is

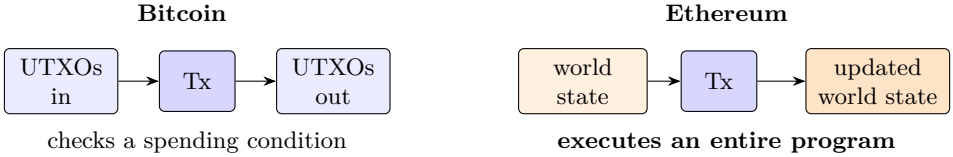
> *"if `google.com` is already registered, no one else may register it."*

That is a check *across* UTXOs, does any other output already claim this name?, not a check on the single output being spent. Bitcoin Script cannot ask the question, so nothing stops two independent registrations of the same name.

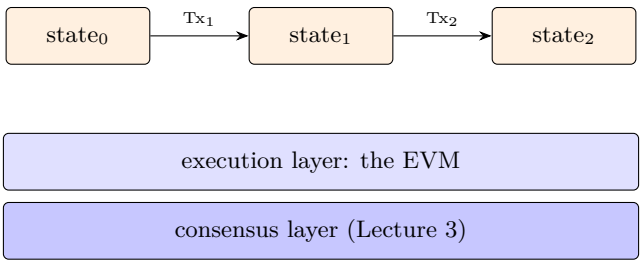
What people did, and the better idea. NameCoin (launched April 2011, with merged mining added that October) forked Bitcoin's *code* into a separate chain that hard-codes this one naming rule into its validation logic. That works, but it is a whole new network and consensus system for a single application. The more general answer is to build a blockchain that natively runs *arbitrary* contracts, so that "no duplicate names" is just a few lines of ordinary code that every node executes. That blockchain is **Ethereum**.

1.3 Two state-transition systems

Both Bitcoin and Ethereum are state-transition systems: there is a global state S, an input I (a transaction), and a deterministic function $F : S \times I \to S$ that every node applies identically (Lecture 1). They differ entirely in what F does.



In Bitcoin, F consumes some UTXOs and produces others, and the only computation is running each input's script to a boolean. In Ethereum, F takes the entire *world state*, runs a program that may read and write arbitrary parts of it, and returns the modified world state. A **decentralized application** (DAPP) is program code deployed on-chain; each transaction it processes is a state transition, recorded permanently.



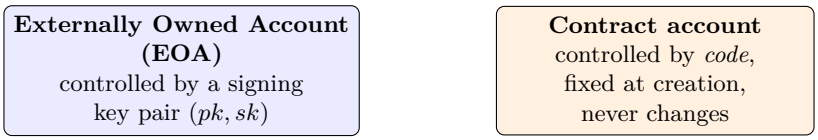
The consensus layer fixes *which* transactions happened and *in what order*; the EVM says what each one *does* to the state. Everything in the rest of this lecture lives in that execution layer.

Remark 1.1 (What Ethereum's programmability buys)**.** Once every account can run arbitrary code, a large design space opens up: fungible tokens through a shared interface (the ERC-20 standard, Section 4.6); non-fungible tokens (ERC-721); exchanges, lending markets, stablecoins and derivatives (Lecture 5); on-chain organisations (DAOs) that hold funds and act on recorded votes. The common thread is that the rules are code that every node re-executes, so no operator can quietly deviate from them.

2 Ethereum's state: accounts and storage

Where Bitcoin's state is a *set of UTXOs*, Ethereum's state is a map from 20-byte **addresses** to **accounts**. This is the "account model", and it is closer to how a bank ledger works: balances are stored explicitly and mutated in place, rather than represented as a pile of unspent outputs.

2.1 Two kinds of account



EOA.
What a user controls. It has no code; it acts only when its owner signs a transaction with sk. Every transaction on Ethereum originates from exactly one EOA.

Contract account.
Created by a transaction, it holds *code* that was set once at creation and can never be altered, plus its own private storage. It never initiates anything on its own: it runs only when *called*, by an EOA's transaction or by another contract during execution.

Optional, deeper detail (not examinable)

Since May 2025 (EIP-7702) an EOA may install a *persistent* pointer to contract code, blurring the line above. A type-`0x04` transaction writes a 23-byte delegation designator (`0xef0100` ‖ address) into the account's code field; that code then runs in the EOA's own storage and balance context. The delegation stays in place across later transactions until the owner replaces it or clears it (a fresh type-`0x04` transaction pointing at the zero address). This is used for "account abstraction" features (batching, sponsored gas, session keys). The two-account model still governs everything in these notes; treat EIP-7702 as an EOA opting in to contract-like behaviour.

2.2 What is stored per account

Every account, of either kind, has four fields; two of them are empty ($\perp$) for an EOA.

Field	EOA	Contract
address	last 20 B of keccak256(pk)	last 20 B of keccak256(rlp[CreatorAddr, nonce])
code	$\perp$ (or an EIP-7702 delegate address)	`CodeHash`
storage root	$\perp$	`StorageRoot`
balance	in Wei	in Wei
nonce	# transactions sent	1 + (# contracts created)

Notes:

- 1 Wei $= 10^{-18}$ ETH; balances are always integers of Wei.
- The **nonce** is a per-account counter. For an EOA it is the number of transactions it has already sent, and a transaction is valid only if its nonce equals the account's current nonce. This is the account model's anti-replay device: it plays the role that Bitcoin's (`TxID`, `index`) outpoint plays in the UTXO model (Lecture 2). It also forces a total order on each sender's transactions.
- A contract's **address** is derived by hashing the creator's address together with a nonce, so it is deterministic but, for ordinary `CREATE`, not known until the creating transaction executes (Section 4.10).
- In the `address` rows, keccak256 is Keccak-256 (the pre-standard Keccak, whose padding rule differs from finalised SHA-3), and "last 20 B" keeps the low-order 20 bytes of the 32-byte digest. For an EOA, pk is the 64-byte *uncompressed* public key—the curve point (x, y)—with its leading `0x04` tag byte stripped; hashing a 33-byte compressed key gives the wrong address.
- Since EIP-161 a new contract's nonce is initialised to 1, not 0, so a contract's nonce equals 1 plus the number of contracts it has itself created via `CREATE`/`CREATE2`.
- `CodeHash` and `StorageRoot` are *commitments*: the code and the storage contents are stored by nodes, and the account record holds only their hashes, so the whole world state collapses to a single root hash per block.

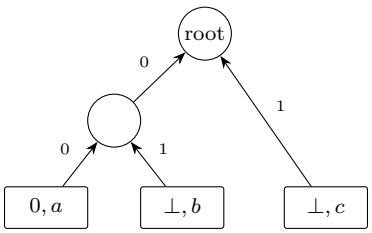
2.3 The contract storage array

Each contract owns a private array

$$S[0],\ S[1],\ \ldots,\ S[2^{256}-1],$$

of 2^{256} cells, each cell 32 bytes wide, every cell initialised to 0. A contract's state variables and mappings are laid out in this array by the compiler. The account's `StorageRoot` is the root of a *Merkle Patricia Trie* over $S[\,]$.

Obviously no node stores 2^{256} cells. Only the cells a contract has actually written to a non-zero value are materialised; the rest are implicitly 0 and contribute nothing to the trie. So the cost of recomputing `StorageRoot` is $\mathcal{O}(|S|)$ where $|S|$ is the number of non-zero cells, not 2^{256}.



Remark 2.1 (UTXO model vs. account model, revisited)**.** Lecture 2 previewed this comparison. The UTXO model makes transactions self-contained (an input names exactly the coin it spends) and easy to validate in parallel, and gives a natural notion of "coin history". The account model is more compact and far more natural for *stateful* programs: a contract just reads and writes named slots, with no need to thread state through a chain of outputs. Its cost is that validation is inherently sequential per sender (the nonce ordering) and that global state can only grow.

2.4 Where blocks come from

Consensus is Lecture 3's subject; here is only what the execution layer needs to assume. Ethereum runs proof of stake: time is divided into **slots** of 12 seconds, and for each slot one **block proposer** is drawn from the validator set. To be a validator, a party locks up a stake (at least 32 ETH per validator—since EIP-7251 in the 2025 "Pectra" upgrade a validator's effective balance may range up to 2048 ETH—or a share of a pooled provider); validators sign the blocks they see, and once enough signatures accumulate a checkpoint is *finalized* (Lecture 3). A proposer that produces a valid block earns transaction fees and newly issued ETH; a validator that signs conflicting or invalid data is *slashed*, losing part of its stake. A valid block is a batch of ordered transactions; every full node re-executes all of them through the EVM and checks that the resulting state root matches the one in the block header.

Check your understanding

Compared with an externally owned account, what is distinctive about a **contract** account?

(a) Its code is set at creation time and never changes.
(b) It has no nonce, because it never sends transactions.
(c) It is controlled by a private key, just like an EOA.
(d) Its balance is always zero.

Answer: (a). A contract does have a nonce (initialised to 1, then incremented once for each contract it creates), it is controlled by code rather than a key, and it can hold any balance.

3 The EVM and gas

3.1 The EVM as a metered stack machine

A contract's code is **EVM bytecode**. Developers write a higher-level language, almost always Solidity, and a compiler (`solc`) produces the bytecode; validators run the EVM to execute that bytecode in response to a transaction.

The EVM is a **stack machine**, like Bitcoin Script, but with two decisive differences:

- it has `JUMP` / `JUMPI`, hence loops and unbounded control flow;
- alongside the stack it has addressable **memory** (`MLOAD` / `MSTORE`, volatile, per call) and persistent per-contract **storage** (`SLOAD` / `SSTORE`, the array of Section 2.3).

The operand stack holds up to 1024 items of 32 bytes each. One contract can `CALL` another; the call creates a fresh execution frame with its own memory and its own stack, and that frame is discarded when the call returns. Because the machine can loop, "how long will this transaction take to validate?" is no longer answerable from the code length alone, the way it was for Bitcoin Script (Lecture 2). Something has to bound execution. That something is gas.

3.2 Gas: every instruction has a price

Each EVM operation costs a fixed, protocol-defined amount of **gas**. A transaction carries a `gasLimit`; execution proceeds instruction by instruction, subtracting each one's cost, and if the counter reaches zero the transaction *aborts* and all its state changes are rolled back. The gas

consumed up to that point is still charged: its base-fee portion is burned and its tip goes to the proposer.

Rough orders of magnitude (arithmetic and stack ops cost a few gas each):

- `SSTORE` (writing a storage slot) is the expensive one: setting a slot from zero to non-zero costs on the order of 2×10^4 gas, because every node must store that data forever.
- Zeroing a slot that was non-zero triggers a partial *gas refund*, because it *shrinks* the state everyone stores; since EIP-3529 the total refund is capped at 20% of the transaction's `gasUsed`, so zeroed slots cannot be hoarded for profit.
- `CALL` and `CREATE` cost thousands of gas plus a term proportional to the size of code or data involved.
- `MSTORE` to memory is cheap, but the cost of memory grows *quadratically* in the highest offset touched, so very large memory buffers become expensive.

Optional, deeper detail (not examinable)

Concrete post-2021 figures (they drift with hard forks, check a reference before quoting):

- `SSTORE` (since EIP-2929, "Berlin"): the first, *cold* access to a slot in a transaction adds 2 100 gas. Zero → non-zero: 20,000 + 2,100. Non-zero → non-zero: 5 000 cold / 2 900 warm. Non-zero → zero *refunds* 4 800 (cut from 15 000 by EIP-3529 in 2021, which also capped the total refund at 20% of the transaction's `gasUsed`, killing "gas token" schemes that hoarded refunds).
- `CREATE`: 32,000 + 200 × (deployed code size) + 2 × (init-code words); the per-word term is EIP-3860.
- `SELFDESTRUCT`: 5 000 gas, +25,000 if it sends balance to a previously empty account.
- Since "Fusaka" (EIP-7825) a single transaction is capped at $2^{24} \approx 16.78$M gas regardless of the block limit.

3.3 Why meter execution at all

Two reasons:

1. **Termination.** Without a per-step charge, a transaction containing `while (true) {}` would run forever, and every validating node with it. Gas guarantees every transaction halts: it runs out of gas if nothing else stops it.
2. **Rationing a scarce resource.** Block space (total gas per block) is limited. Under load, the proposer chooses which pending transactions to include so as to maximise its own fee income, so users must pay for the space they consume.

The pre-2021 fee market. Originally every transaction simply named a `gasPrice` it was willing to pay, and the proposer took the highest bidders. That is a first-price auction: users must guess what others will bid and routinely overpay, and the mechanism gives the proposer an incentive to *appear* congested. EIP-1559 replaced it.

3.4 EIP-1559: base fee plus tip

Every block carries a **base fee** per gas: the minimum `gasPrice` any transaction in that block must pay. The base fee is adjusted automatically from one block to the next, purely as a function of how full the previous block was:

- previous block *full* (used its entire gas limit) ⇒ base fee rises by 12.5%;
- previous block *empty* ⇒ base fee falls by 12.5%;
- previous block at the *target* (half the limit) ⇒ base fee unchanged.

A transaction now specifies three numbers: `gasLimit`, `maxFee` (the most it will pay per gas, all-in), and `maxPriorityFee` (a "tip" offered to the proposer on top of the base fee). The price

actually charged is

$$\texttt{gasPrice} \;=\; \min\bigl(\texttt{maxFee},\ \texttt{baseFee} + \texttt{maxPriorityFee}\bigr).$$

The design goals: let users bid their true value for inclusion without second-guessing the auction; remove the proposer's incentive to fake congestion; and discourage off-chain side deals (Section 3.6).

Optional, deeper detail (not examinable)

The gas *limit* per block is not fixed by the protocol; validators vote it up or down slowly, so it drifts over time (around 45M gas in late 2025, rising toward 60M through 2026). Quote a block explorer, not a lecture note, for its current value. The 12.5% step and the "target = limit/2" relationship are the stable parts.

3.5 Tracing a transaction's gas

When a transaction is executed, the node performs the following, in order:

1. If `maxFee` $<$ `baseFee`: reject (cannot afford the block).
2. If `gasLimit` $\times$ `maxFee` $+$ `value` $>$ sender's balance: reject (cannot cover the worst case plus the ETH being sent).
3. Deduct `gasLimit` $\times$ `maxFee` from the sender up front; set the internal counter `Gas` $\leftarrow$ `gasLimit`.
4. Execute the transaction, subtracting each instruction's cost from `Gas`. If `Gas` hits zero, abort and roll back all state changes; the entire `gasLimit` then counts as consumed, and an out-of-gas transaction gets no refund.
5. Let `gasUsed` $=$ `gasLimit` $-$ `Gas` (so `gasUsed` $=$ `gasLimit` on an out-of-gas abort). Return to the sender everything escrowed in step 3 above the true cost, namely `gasLimit` $\times$ `maxFee` $-$ `gasUsed` $\times$ `gasPrice`.
6. **Burn** `gasUsed` $\times$ `baseFee`; pay `gasUsed` $\times$ (`gasPrice` $-$ `baseFee`) to the proposer as the tip.

So the sender's true cost is `gasUsed` $\times$ `gasPrice`; the up-front charge in step 3 is only an escrow against the maximum, and step 5 returns the difference.

3.6 Why burn part of the fee

Suppose the base fee were paid to the proposer rather than destroyed. Then in a quiet period a proposer could privately promise to *rebate* the base fee to a large sender off-chain, effectively discounting inclusion and competing for volume through side deals, exactly the behaviour EIP-1559 set out to prevent.

Burning the base fee removes that lever: the base fee leaves circulation no matter what the proposer does, so there is nothing to rebate. The proposer's only competitive margin is the tip, which is transparent and on-chain. A side effect is monetary: when burnt base fees exceed newly issued ETH, total supply *decreases*.

Check your understanding

Under EIP-1559, if the previous block was completely full (it used its entire gas limit), what happens to the base fee for the next block?

(a) It stays exactly the same.
(b) It decreases by 12.5%.
(c) It increases by 12.5%.
(d) It is reset to zero.

Answer: (c). A full block signals excess demand, so the protocol raises the base fee by

12.5%; an empty block lowers it by 12.5%; a block at the target leaves it unchanged.

4 Solidity and smart contracts

Solidity is the dominant language for writing Ethereum contracts. It is statically typed and compiles down to the EVM bytecode of Section 3: every high-level construct below becomes `SSTORE`, `CALL`, `MSTORE`, and friends.

4.1 Contracts, interfaces, inheritance

A `contract` is a bit like a class: it bundles state variables with the functions that operate on them. An `interface` declares function signatures with no bodies; a contract can `is` another contract or interface to inherit its declarations and code.

```
interface IERC20 {
    function transfer(address _to, uint256 _value)
        external returns (bool);
}

contract ERC20 is IERC20 {              // inheritance
    address owner;
    constructor() { owner = msg.sender; }
    function transfer(address _to, uint256 _value)
        external returns (bool) { /* ... */ }
}
```

The `constructor` runs exactly once, at deployment (Section 4.10); `msg.sender` inside it is the deployer.

4.2 Value types and reference types

Solidity types split into two families by how they are passed around.

Value types (copied on assignment)	**Reference types** (a handle is passed)
`uint8` … `uint256`	`struct`, fixed and dynamic arrays
`address`, `address payable` (20 bytes); `.send()`/`.transfer()` require `payable`	`bytes`, `string`
`bytes32`, `bool`	`mapping(address => uint256)`

A reference type always carries a **data location** annotation (Section 4.8) that says which region of the EVM it lives in.

The `address` type and raw calls. `someAddress.call{value: v, gas: g}(data)` sends a raw, transaction-like message to another account: `data` is arbitrary calldata, and by default the call forwards essentially all remaining gas—strictly, EIP-150 withholds 1/64 of the gas left at each nesting level, so a deeply nested call chain eventually starves. The older `.send()` and `.transfer()` (methods of `address payable`, not plain `address`) forward only a fixed 2 300-gas stipend and return (or revert) on failure; they are discouraged because EIP-1884 ("Istanbul", December 2019) raised `SLOAD` and other opcode costs, so a recipient whose fallback does even one storage read now overruns the stipend. Keep the unmetered `.call` in mind: its "run whatever code you like, with all my gas" semantics is exactly what makes the reentrancy attack of Section 5 possible.

4.3 Visibility and state mutability

Two independent axes control a function.

Visibility	Who may call it
`external`	only from outside the contract
`public`	from outside and from inside
`internal`	this contract and contracts that inherit it
`private`	this contract only
State mutability	*(a separate axis)*
`view`	may read storage, must not write it
`pure`	must not touch storage at all

Two context values are easy to confuse: `msg.sender` is the *immediate* caller (which may be another contract), while `tx.origin` is the EOA that started the whole call chain. Authorisation checks should use `msg.sender`; relying on `tx.origin` is a known anti-pattern.

4.4 Modifiers and access control

A `modifier` wraps a function body with a reusable precondition. The body is spliced in where the modifier writes `_;`.

```
contract owned {
    address payable owner;
    constructor() { owner = payable(msg.sender); }
    modifier onlyOwner {
        require(msg.sender == owner);
        _;                      // function body runs here
    }
}

contract Closable is owned {
    function close() public onlyOwner {
        selfdestruct(owner);     // owner must be ‘address payable‘
    }
}
```

`Closable` inherits `owned`'s code, so the `require` in `onlyOwner` runs before `close`'s body on every call.

Optional, deeper detail (not examinable)

Since EIP-6780 ("Dencun", 2024) `selfdestruct` only erases a contract's code and storage if the contract was *created in the same transaction*; otherwise it merely forwards the contract's balance and the code lives on. Do not rely on `selfdestruct` to make a contract disappear.

4.5 Error handling: `require`, `revert`, custom errors

```
require(balances[msg.sender] >= _value, "insufficient balance");

error InsufficientBalance(uint256 requested, uint256 available);  // >=0.8.4
if (balances[msg.sender] < _value)
    revert InsufficientBalance(_value, balances[msg.sender]);

assert(totalSupply == sumOfAllBalances);   // internal invariant only
```

All of `require`, `revert`, and a failed `assert` abort the transaction and roll back *every* state change made so far. Unlike running out of gas, they *refund* the unused gas. `require` is for validating inputs and preconditions; `assert` is for invariants that should never be false if the code is correct. **Custom errors** are cheaper than string reasons: the string never has to be stored in the contract bytecode or returned in calldata, only a 4-byte selector and the arguments are.

4.6 ERC-20: a shared token interface

An ERC-20 token *is* a contract. It keeps balances in its own storage,

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

and exposes a fixed set of functions so that *any* application can handle *any* token with no token-specific code:

- `transfer`, `transferFrom`, `approve` (move tokens, and authorise a third party to move them on your behalf);
- `totalSupply`, `balanceOf`, `allowance` (read-only views).

A minimal `transfer`:

```
function transfer(address _to, uint256 _value)
    external returns (bool)
{
    require(balances[msg.sender] >= _value, "insufficient balance");
    balances[msg.sender] -= _value;
    balances[_to]        += _value;
    emit Transfer(msg.sender, _to, _value);   // write a log entry
    return true;
}
```

Note the order: the balances are updated *before* anything external happens (here nothing external happens at all, and `emit` is not a call out). That ordering is the whole lesson of Section 5.

4.7 Calling other contracts, and the ABI

To call another contract you cast its address to a contract type and invoke a method:

```
address _token = 0x2b34...791;
ERC20Token tokenContract = ERC20Token(_token);       // type cast, no code runs
bool ok = tokenContract.transfer(_to, _value);       // typed call: returns bool
```

A high-level *typed* call like this returns the declared type directly. The low-level form is `(bool success, bytes memory ret) = _token.call(data)`, which returns a success flag and raw return bytes and never reverts on its own. One sharp edge: a low-level `.call` to an address that holds *no code* still returns `success = true` (there is nothing to run, so nothing to fail), so the flag alone does not confirm that a contract was there. Either way, what actually travels to the callee is **calldata**:

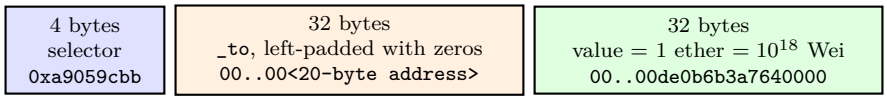
$$\texttt{calldata} \;=\; \underbrace{\text{4-byte selector}}_{\text{which function}} \;\|\; \underbrace{\text{ABI-encoded arguments}}_{\text{the arguments}},$$

where the *function selector* is

$$\texttt{selector} \;=\; \texttt{bytes4}(\text{keccak256}(\texttt{"transfer(address,uint256)"})).$$

The selector is computed from the function's *signature string* (name and parameter types, no spaces, no parameter names). The Application Binary Interface (ABI) is the fixed convention for how the argument bytes are laid out.

4.7.1 Worked example: encoding `transfer(_to, 1 ether)`

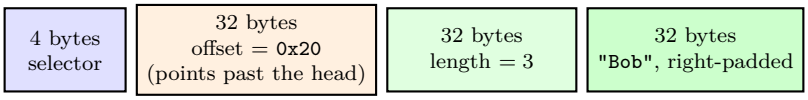


Every static argument occupies exactly 32 bytes, right-aligned for numbers and addresses. Total calldata: $4 + 32 + 32 = 68$ bytes. The same bytes can be built explicitly with

`abi.encodeWithSelector(tokenContract.transfer.selector, _to, 1 ether)`.

4.7.2 Dynamic types: offsets and a tail

A `string`, `bytes`, or dynamic array has no fixed width, so it cannot be inlined in the 32-byte slot for that argument. Instead the ABI splits the encoding into a fixed-size **head** and a variable-size **tail**: the head holds a 32-byte *offset* pointing into the tail, and the tail holds the length followed by the data, padded up to a multiple of 32 bytes.



For `setName("Bob")` the head is just one offset (`0x20`, meaning "32 bytes in"), and the tail is (length 3) ‖ (`"Bob"` padded). With several dynamic arguments, the head carries one offset each and the tail concatenates their data blocks in order.

4.8 Where variables live: the data locations

The EVM keeps data in several distinct regions, and in Solidity every non-trivial variable is tied to one of them. This is not cosmetic: the location determines gas cost, lifetime, and whether an assignment copies or aliases.

Location	Lifetime	Cost	Typically holds
Stack	one call frame	cheapest	simple values ≤ 32 bytes
Calldata	the transaction	cheap, read-only	`external` function arguments
Memory	one call frame	cheap, grows quadratically	structs, arrays, strings
Transient storage	one transaction	moderate ($\sim$100 gas)	reentrancy locks
Storage	permanent	most expensive	state variables, mappings

The `LOG0`–`LOG4` opcodes write to one further, *write-only* channel, the transaction log (Section 4.9).

Stack.
Cheapest of all. Holds any "simple" type of at most 32 bytes, represented internally as `bytes32`. The stack has 1024 slots, but `DUP`/`SWAP` only reach 16 deep, so the compiler can juggle at most about 16 live locals before it fails with "stack too deep" (compiling via the IR pipeline mostly removes this limit). `uint256 a = 123;` is a stack variable.

Calldata.
The raw, immutable bytes of the transaction (or of an inner call). Reading arguments straight from calldata avoids a copy; every byte still costs gas (16 for a non-zero byte, 4 for a zero byte). Marking a function `external` and *not* adding a `memory` qualifier keeps its arguments in calldata.

Memory.
A resizable byte array scoped to the current call frame, compiled to `MSTORE`/`MLOAD`. Complex values larger than 32 bytes (structs, arrays, strings) must live in memory, calldata, or storage—never on the stack. Cheap per access, but total cost grows *quadratically* in the highest offset touched. `string memory name = "Alice";` is explicit about the location.

Transient storage.
Behaves like storage but is wiped at the end of the transaction rather than persisted. Much cheaper than real storage. Its main use is exactly the reentrancy-guard flag of Section 5.6.

Storage.
The permanent per-contract array of Section 2.3, compiled to `SSTORE`/`SLOAD`. The most expensive location by a wide margin; writes cost more than reads; zeroing a slot earns a partial refund. `mapping`s and state variables are *always* in storage. A gas-saving trick: several variables each smaller than 32 bytes can be *packed* into a single slot.

Optional, deeper detail (not examinable)

Since EIP-7623 ("Pectra") a transaction that is heavy on calldata but light on computation pays a raised per-byte floor, to keep such transactions from being an underpriced way to bloat block size (relevant to rollups, which post data as calldata, Lecture 6).

Check your understanding

A Solidity state variable declared `mapping(address => uint256) balances;` is stored in which EVM location?
(a) Stack (b) Calldata (c) Memory (d) Storage
Answer: (d). Mappings and state variables always live in storage; they cannot be placed anywhere else.

4.9 Event logs and the receipts root

Sometimes a contract wants to *record* that something happened without any other *contract* needing to read it back, for example so that wallets and block explorers can display a token transfer. That is what **events** are for.

```
emit Transfer(msg.sender, _to, _value);
```

- Log entries are written to the transaction's **receipt**, not to contract storage, which makes them much cheaper than an `SSTORE`.
- Every block header commits to a **receipts root**: a Merkle root over all the transaction receipts in the block, and hence over all their logs.
- Off-chain software subscribes to logs and indexes them. On-chain code generally *cannot* read past logs, so events are a one-way channel from the chain to the outside world.

4.10 Contract creation: `CREATE` and `CREATE2`

Deploying a contract runs its constructor once and installs the returned runtime bytecode at a fresh address. There are two ways to compute that address.

- `CREATE` (ordinary deployment):

$$\text{address} = \text{last 20 B of } \text{keccak256}(\text{rlp}[\text{CreatorAddr}, \text{CreatorNonce}]).$$

  It depends on the creator's nonce, so you cannot know the address until the deploying transaction actually executes.

- `CREATE2`:

$$\text{address} = \text{last 20 B of } \text{keccak256}(\texttt{0xff} \,\|\, \text{CreatorAddr} \,\|\, \text{salt} \,\|\, \text{keccak256}(\text{initCode})).$$

  No nonce: given the creator, a chosen *salt*, and the init code, anyone can compute the address *before* deployment.

As in Section 2.2, the hash is Keccak-256 and only the low-order 20 bytes of each 32-byte digest are kept.

`CREATE2`'s predictability is what lets smart-contract wallets and L2 bridges use a *counterfactual* address: you can receive funds at an address whose contract has not been deployed yet, and deploy it later, on demand.

4.11 `delegatecall` and proxy contracts

Two ways for contract `A` to invoke code in contract `B`:

- `B.call(...)`: `B`'s code runs *in `B`'s context*. Storage reads and writes hit `B`'s storage, and inside `B`, `msg.sender` is `A`.
- `B.delegatecall(...)`: `B`'s code runs *in `A`'s context*. Storage reads and writes hit *`A`'s* storage, and `A`'s original `msg.sender` and `msg.value` are preserved as if the code were `A`'s own.

This is the mechanism behind upgradeable **proxy contracts**: a thin *proxy* holds all the state and storage, and `delegatecall`s every call into a separate *logic* contract that can be swapped for a new version.

Remark 4.1 (The storage-layout hazard)**.** Because `delegatecall` runs foreign code against *your* storage, the proxy's and the logic contract's storage layouts must line up slot for slot. If version 2 of the logic inserts a new state variable above an old one, every subsequent access reads or writes the wrong slot, silently. Mismatched storage layout across an upgrade is a recurring, high-severity class of bug.

5 Security: reentrancy

Everything in Section 4 was mechanism. This section is one vulnerability, studied closely, because it illustrates a whole family and because it caused the most consequential incident in Ethereum's history.

5.1 A vulnerable bank

The contract below looks reasonable: deposits add to a per-user balance, withdrawals send the balance back and then zero it.

```
contract Bank {
    mapping(address => uint256) public userBalances;

    function addToBalance() public payable {
        userBalances[msg.sender] += msg.value;
    }

    function withdrawBalance() public {
        uint256 amount = userBalances[msg.sender];
        (bool ok,) = msg.sender.call{value: amount}("");  // <-- sends ETH
        require(ok, "withdraw failed");
        userBalances[msg.sender] = 0;      // <-- cleared AFTER the call
    }
}
```

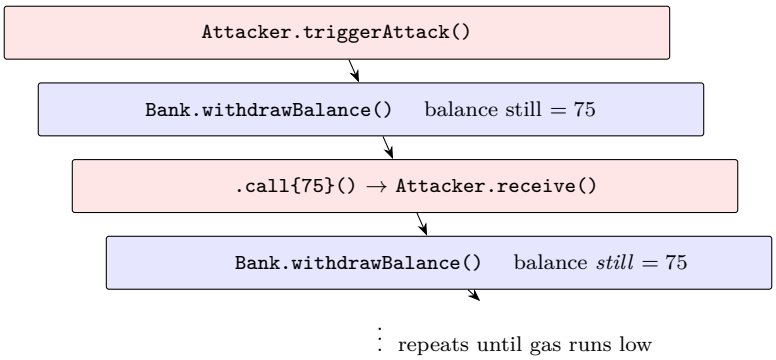
The flaw is the order of the last two statements: the external transfer happens *before* the balance is cleared.

5.2 The attacker

```
contract Attacker {
    Bank bank;

    constructor(address bankAddress) payable {
        bank = Bank(bankAddress);
        bank.addToBalance{value: 75}();      // genuine deposit of 75 Wei
    }

    function triggerAttack() public { bank.withdrawBalance(); }

    receive() external payable {
        if (gasleft() > 10000 && address(bank).balance >= 75) {
            bank.withdrawBalance();          // call straight back in
        }
    }
}
```

A contract's `receive()` function runs automatically whenever the contract is sent ETH, including *in the middle of* the Bank's `withdrawBalance`, right at the outward `.call`.

5.3 Tracing the drain



Each recursive entry reads `userBalances[msg.sender]`, still 75, sends another 75 Wei, and only *then* would have cleared the balance, except it never gets there, because the `.call` has not returned yet. The recursion unwinds only when `receive()`'s `if` finally fails (low gas, or the Bank drained below 75). The "low gas" outcome is guaranteed eventually: by EIP-150's 1/64 rule (Section 4.2) each nested `.call` is handed strictly less gas than its parent held, so the depth is bounded even without the explicit `gasleft()` check. By then the Bank has paid out many multiples of the attacker's real deposit.

5.4 Why it drains: state-update ordering

`userBalances[msg.sender] = 0;` runs only *after* `msg.sender.call{value: amount}("")` returns. But that call does not return until the callee's code, `Attacker.receive()`, finishes, and `receive()` immediately calls `withdrawBalance()` again, which reads the caller's `userBalances` entry *before it was ever cleared*.

The bug is not really "recursion". It is that an **external interaction happens before the effect that was meant to prevent reuse**. Anything the callee can do, including calling back in, sees stale state.

5.5 Fix 1: checks–effects–interactions

Reorder every function that touches state and then calls out, into three phases:

```
function withdrawBalance() public {
    uint256 amount = userBalances[msg.sender];

    require(amount > 0, "no balance");             // 1. checks
    userBalances[msg.sender] = 0;                  // 2. effects

    (bool ok,) = msg.sender.call{value: amount}(""); // 3. interactions
    require(ok);
}
```

Validate first, then update all state, then interact with the outside world. By the time `Attacker.receive()` re-enters, the balance is already zero and the second withdrawal transfers nothing.

5.6 Fix 2: a reentrancy guard

A mutex: set a flag on entry, reject any nested call that finds it set, clear it on exit.

```
bool transient locked;        // flag in transient storage (wiped each tx)

modifier nonReentrant {
    require(!locked, "reentrancy attempt");
    locked = true;
    _;
    locked = false;
}

function withdrawBalance() public nonReentrant { /* ... */ }
```

Placing the flag in *transient* storage (Section 4.8) makes the guard cheap; `transient` variables require `solc` $\geq$ 0.8.28 and an EVM target of Cancun or later, so on an older toolchain use a plain `bool` in ordinary storage instead. In practice production contracts apply *both* fixes: checks–effects–interactions as the discipline, plus `nonReentrant` as a backstop.

5.7 Cross-function and read-only reentrancy

Guarding `withdrawBalance` alone is not enough, because the callback need not re-enter the *same* function.

- **Cross-function reentrancy.** The callback enters a *different* function that reads the same not-yet-updated state, e.g. a `transfer` that reads `userBalances` while a withdrawal is mid-flight.
- **Read-only reentrancy.** The callback enters a `view` function. It writes nothing, but it returns *stale* data to some third party that trusts it, e.g. a price oracle reading a decentralized-exchange pool's reserves while a swap is only half-applied.

A `nonReentrant` modifier on one function does not stop either variant unless *every* function that touches the shared state, including the `view` functions other contracts rely on, is guarded consistently. (Real incidents: Cream Finance in 2021; several oracle integrations that read Curve pool state.)

Check your understanding

Most precisely, what category of bug is the `Bank` vulnerability?

(a) Integer overflow.
(b) Missing access control on `withdrawBalance`.

(c) State-update ordering: an external interaction runs before the effects.
(d) Signature replay.

Answer: (c). The withdrawal sends ETH before it clears the balance, so a re-entrant call reads a balance that should already be zero.

5.8 The DAO hack (2016)

"The DAO" was a crowdfunded venture fund on Ethereum, governed entirely by its Solidity code, which raised about 12.7M ETH (roughly $150M at the time). Its `splitDAO` withdrawal path had almost exactly the `Bank` bug: it sent ETH out through a low-level call and updated the internal balance *afterward*. An attacker re-entered `splitDAO` recursively and drained **3.6M ETH**, about a third of the fund, over several hours, in full public view on-chain. The drained ETH landed in a "child DAO" subject to a 27-day holding period, which is the only reason a response was still possible.

Ethereum's response was a contentious **hard fork** that effectively reversed the theft by moving the funds to a recovery contract. Most of the ecosystem followed the fork; the minority chain that kept the original, unreverted history still runs today as **Ethereum Classic** (ETC). The episode is why checks–effects–interactions is now taught as a rule rather than a style preference.

6 Summary

The lecture in five points:

1. **Bitcoin Script cannot express a rule across many coins**, because every evaluation sees only the one output being spent. Ethereum's answer is a blockchain that runs *arbitrary* contracts, so "no duplicate names", "at most 2 per day", and anything else are just code every node executes.
2. **The state is a map from addresses to accounts.** EOAs are key-controlled and start every transaction; contract accounts are code-controlled, run only when called, and own a 2^{256}-cell storage array committed to by a Merkle Patricia Trie root. The per-account *nonce* is the account model's anti-replay device.
3. **The EVM is a stack machine with loops, and gas is the meter.** Every instruction costs gas; a transaction that exhausts its `gasLimit` aborts and rolls back. **EIP-1559** sets a per-block *base fee* that moves ±12.5% with demand and is *burned*, plus a transparent *tip* to the proposer.
4. **Solidity compiles to that bytecode.** Key ideas: value vs. reference types; visibility and `view`/`pure`; the four data locations (stack, calldata, memory, storage) plus transient storage and the write-only log; ABI encoding (4-byte selector ‖ 32-byte-aligned args, with offsets for dynamic types); `CREATE2` for predictable addresses; and `delegatecall` for upgradeable proxies, with its storage-layout hazard.
5. **Reentrancy** is a state-update-ordering bug: interacting with the outside world before writing the state that was meant to prevent reuse. Fix it with *checks–effects–interactions* and a *reentrancy guard*; note the cross-function and read-only variants; and remember the DAO hack (3.6M ETH, 2016) that split the chain.

Lecture	Gave us	In service of
L1	hashing, Merkle trees, proof of work, signatures	the primitives
L2	UTXOs, Script, wallets	the simplest shared state: a ledger
L3	Byzantine agreement, Nakamoto, proof of stake	*how* strangers agree on a log
L4	the EVM, Solidity, gas, contracts	shared state becomes a computer

Looking ahead. Lecture 5 uses this machinery to build **decentralized finance**: stablecoins and over-collateralised lending, automated market makers and the constant-product formula behind Uniswap, and the maximal-extractable-value (MEV) that proposers and searchers pull out of ordinary transactions.