

## #4 Solidity I: Basics

Lecture Notes for CS190N: Blockchain Technologies and Security

October 8, 2025

In this lecture, we develop a clear working model of Solidity and the contracts it enables on Ethereum. We begin with the motivation from Bitcoin's limits and introduce Ethereum's account based global state, then build the anatomy of a Solidity source file, covering state variables, core types, and how storage layout affects persistence and cost. We make data flow concrete by following values between calldata, memory, and storage, and we study function design through visibility and mutability together with precise use of data locations. We handle value transfer with payable functions and the call pattern, add reusable access control with modifiers, and use execution context variables such as `msg.sender`, `msg.value`, and `block.timestamp`. By the end, you will be able to read and write small contracts, reason about which operations consume gas and why, and apply safe patterns to implement and audit basic on chain logic.

### 1 FOUNDATIONS AND MOTIVATION

Programmable blockchains extend the idea of digital cash to general applications. To understand why Solidity looks the way it does, we start with Bitcoin's design constraints, then the Ethereum state model, and finally the smart contract abstraction that runs on the EVM.

#### 1.1 Bitcoin's Design and Its Limits

Bitcoin uses a simple stack based language named Script. Script is intentionally not Turing complete, which keeps validation bounded and lowers denial of service risk. Bitcoin tracks value with the **UTXO** model. Transactions consume existing unspent outputs and create new ones. A user's balance is the sum of UTXOs that their keys control. This model is efficient for payments. It is inconvenient for applications that need persistent shared state, for example enforcing a per user daily quota, because there is no central account object that can be updated in place.

#### 1.2 Ethereum as a Transaction Based State Machine

Ethereum models the blockchain as a single global transaction based state machine. The world state maps addresses to **accounts**. Each account object contains a nonce, a balance, a code hash, and a storage root that commits to contract storage. This account based approach turns the chain into a stateful computer with persistent memory and enables general purpose smart contracts.

#### 1.3 What a Smart Contract Is

A smart contract is code and data stored at a specific address on the chain. Immutability means the deployed bytecode at an address does not change. Transparency means code and interactions are publicly observable on chain. Many production systems add upgrade paths with proxies. Immutability still applies to the bytecode at each address, while a proxy can forward calls to new logic if governance updates the pointer. This shifts trust to deterministic execution, which makes code correctness and security central.

### 2 SOLIDITY BUILDING BLOCKS

Solidity is a contract oriented, statically typed language with a surface syntax similar to C++ and JavaScript. We first cover the source file skeleton, then persistent state, core types and storage layout, and finally data locations so later examples are self contained.

## 2.1 Source File Skeleton

Controlling the compiler range avoids accidental semantic drift, and a license tag makes tooling happy.

*Selecting a compiler range and adding a license.* We will look at an example that shows the smallest compilable skeleton and explains the purpose of each line.

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.26;
3 contract SimpleStorage {
4     // state variables and functions will be added below
5 }
```

The license is machine readable and the pragma selects compilers greater or equal to 0.8.26 and less than 0.9.0. Keeping the skeleton minimal helps isolate compilation errors while you iterate.

## 2.2 State Variables and Persistent Storage

State variables are declared at contract scope. They persist in the contract's storage. Writing storage changes global state and consumes gas. Reading storage is cheaper than writing, yet it still runs inside the EVM and costs gas when executed within a transaction.

*Declaring a persistent number with an auto getter.* We will look at an example that declares a persistent integer and uses the auto generated getter for read access.

```
1 contract SimpleStorage {
2     uint256 public storedData; // public generates a getter named storedData()
3 }
```

After deployment, calling `storedData()` returns the current value. Reading through RPC can be done off chain without spending gas because it does not create a transaction.

## 2.3 Core Types and Storage Layout

Solidity is statically typed. Value types such as `uint256` are the practical default for arithmetic because they match the EVM word size. Reference types such as `mapping` and `struct` let you model richer state. Storage layout influences cost when you write multiple fields.

*Value types and when smaller widths help.* We will look at an example that explains why `uint256` is the common default and when smaller widths are useful.

```
1 contract ValueTypes {
2     uint256 public a; // good default for arithmetic
3     uint8 public b; // smaller width helps only when packing fields
4 }
```

Smaller widths do not automatically save gas in arithmetic. Savings appear when several small fields are stored together in one 256 bit slot.

*Reference types for structured state.* We will look at an example that combines `mapping` and `struct` to create a persistent and auditable shape of state.

```
1 contract UserDirectory {
2     struct User { string name; bool isRegistered; }
3     mapping(address => User) public users; // address -> profile
4     mapping(address => uint256) public balances; // address -> balance
5     function register(string memory name) public {
6         users[msg.sender] = User({name: name, isRegistered: true}); // storage write
7     }
8 }
```

```

7   }
8 }

```

This maps each address to a profile and to a balance. The assignment to `users[msg.sender]` is a storage write that persists on chain and has a gas cost.

*How slot packing reduces writes.* We will look at an example that contrasts packed and unpacked layouts to show the difference in the number of storage writes.

```

1 contract Packing {
2     struct Packed { uint128 a; uint128 b; } // can fit into one slot
3     struct Unpacked { uint256 a; uint256 b; } // uses two slots
4     Packed public p;
5     Unpacked public u;
6     function setPacked(uint128 x, uint128 y) external {
7         p = Packed(x, y); // one slot write when stored together
8     }
9     function setUnpacked(uint256 x, uint256 y) external {
10         u = Unpacked(x, y); // two slot writes
11     }
12 }

```

When fields fit in a single 256 bit slot and you write them together, the EVM can perform fewer `SSTORE` operations, which lowers cost on write heavy paths.

## 2.4 Data Locations: storage, memory, and calldata

Choosing the correct data location affects both semantics and gas. Storage is persistent. Memory is temporary for the duration of a call. Calldata is a read only view of external call input and is efficient for large parameters.

*Copying versus referencing and why it matters.* We will look at an example that first copies a struct from storage to memory, then edits through a storage reference, and finally consumes a large array from calldata.

```

1 contract Locations {
2     struct Profile { string name; uint256 age; }
3     mapping(address => Profile) private profiles;
4     function getProfile(address user) public view returns (Profile memory p) {
5         p = profiles[user]; // storage -> memory copy
6     }
7     function setAge(uint256 newAge) public {
8         Profile storage ref = profiles[msg.sender];
9         ref.age = newAge; // persistent write
10    }
11    function sum(uint256[] calldata xs) external pure returns (uint256 s) {
12        for (uint256 i = 0; i < xs.length; i++) {
13            s += xs[i];
14        }
15    }
16 }

```

Returning a struct as memory copies it out of storage. Editing through a storage reference changes the on chain value. Marking external parameters as calldata avoids copying them into memory and can reduce cost for large inputs.

### 3 FUNCTIONS AND CONTRACT BEHAVIOR

Functions define behavior and the public interface. Solidity requires explicit visibility for functions. Mutability annotations explain how a function interacts with state and the execution environment. Modifiers and clear error handling complete the picture.

#### 3.1 Function Definition and Multiple Returns

Named return variables can improve readability by avoiding repetition when you return multiple values.

*Declaring parameters and named returns.* We will look at an example that uses named return values and expands the body for clarity.

```
1 function analyze(uint256 a, uint256 b) public pure returns (uint256 sum, bool isEven) {
2     sum = a + b;
3     isEven = (sum % 2 == 0);
4 }
```

When named returns are present you can assign to them directly and then return; without restating the tuple. Many teams still prefer an explicit final return (sum, isEven); for readability in longer functions.

#### 3.2 Visibility Specifiers

Public functions are callable inside and outside the contract. External functions are callable only from outside or via this.f(). Internal functions are callable in this contract and in derived contracts. Private functions are callable only in the defining contract. Modern compilers require explicit visibility for functions. For state variables the default is internal when omitted.

*How each visibility behaves in practice.* We will look at an example that highlights calling differences and a common pitfall with external functions.

```
1 contract VisibilityDemo {
2     uint256 public counter;
3     uint256 internal x;
4     uint256 private secret;
5     function inc() public {
6         counter += 1;
7     }
8     function incExt(uint256 by) external {
9         counter += by;
10    }
11    function _double(uint256 v) internal pure returns (uint256) {
12        return v * 2;
13    }
14    function _bumpSecret() private {
15        secret += 1;
16    }
17    function demoCalls() public {
18        uint256 y = _double(counter);
19        _bumpSecret();
20        // incExt(5); // not allowed: cannot call external function internally
21        this.incExt(5); // allowed: uses external call semantics and extra gas
22        counter = y;
23    }
```

```
24 }
```

If a function is intended only for external callers and it accepts large arrays or strings, marking it external can reduce copying because parameters are read directly from calldata.

### 3.3 State Mutability

A function without a mutability keyword may read and write state. A view function may read state but does not write it. A pure function does not read or write state and does not depend on `msg.*` or `block.*`. Off chain calls to view and pure through RPC do not spend gas because they do not create transactions.

*Reading, writing, and pure computation side by side.* We will look at an example that places the three categories together to make the differences concrete.

```
1 contract MutabilityDemo {
2     uint256 private s;
3     function write(uint256 v) public {
4         s = v;
5     }
6     function read() public view returns (uint256) {
7         return s;
8     }
9     function add(uint256 a, uint256 b) public pure returns (uint256) {
10        return a + b;
11    }
12 }
```

The write changes persistent state. The read observes persistent state. The pure function uses only its inputs and can be evaluated off chain without a transaction.

### 3.4 Modifiers and Errors

Modifiers let you factor out preconditions and reuse them across functions. Clear errors improve debuggability and user feedback. On revert the caller pays for gas consumed up to the revert point. Unused gas is not charged.

*Restricting writes to an owner address with a modifier.* We will look at an example that implements the common `onlyOwner` pattern and we will trace the execution order.

```
1 contract Ownable {
2     address public owner;
3     constructor() {
4         owner = msg.sender;
5     }
6     modifier onlyOwner() {
7         require(msg.sender == owner, "caller is not the owner");
8         _;
9     }
10    function changeOwner(address newOwner) public onlyOwner {
11        owner = newOwner;
12    }
13 }
```

When a function annotated with `onlyOwner` is called, the modifier code runs first. If the check passes, control reaches the function body. Centralizing access control keeps call sites readable and allows policy changes in one place.

*Validating inputs and explaining failures clearly.* We will look at an example that shows boundary checks and clear revert messages that improve testability and user feedback.

```

1 contract ErrorsDemo {
2   uint256 private cap = 100;
3   function setCap(uint256 newCap) public {
4     require(newCap >= 10, "cap too small");
5     cap = newCap;
6   }
7   function boundedAdd(uint256 x) public view returns (uint256) {
8     require(x <= cap, "exceeds cap");
9     return x + 1;
10  }
11 }

```

Concise and specific messages help users and developers understand why a call failed and they make tests easier to read.

## 4 VALUE TRANSFER AND EXECUTION CONTEXT

Value flow and context are central to many contracts. Receiving Ether requires explicit opt in with payable. Sending Ether should use the call pattern and check success. Global variables expose the caller, attached value, and coarse time.

### 4.1 Receiving and Sending Ether

Only functions marked payable can receive Ether. Sending value to a nonpayable function reverts. All arithmetic is done in wei. Convenience units are available, for example 1 gwei =  $10^9$  wei and 1 ether =  $10^{18}$  wei.

*Deposit and withdraw with checks effects interactions.* We will look at an example that implements deposit and withdraw and explains how the order of operations reduces reentrancy risk.

```

1 contract Bank {
2   mapping(address => uint256) public balances;
3   function deposit() public payable {
4     balances[msg.sender] += msg.value;
5   }
6   function withdraw(uint256 amount) public {
7     require(balances[msg.sender] >= amount, "insufficient balance");
8     balances[msg.sender] -= amount; // effects first
9     (bool ok, ) = payable(msg.sender).call{value: amount}("");
10    require(ok, "eth transfer failed");
11  }
12 }

```

The withdrawal first reduces the balance, then performs the external call. This order lowers reentrancy risk. The code uses `call` and checks the boolean because transfer and send forward a fixed stipend that can be insufficient under modern gas costs.

*Receiving plain ether transfers.* We will look at an example that shows when receive and fallback run, which is important for contracts that accept plain transfers.

```

1 contract PayableReceiver {
2     event Paid(address indexed from, uint256 amount, bytes data);
3     receive() external payable {
4         emit Paid(msg.sender, msg.value, "");
5     }
6     fallback() external payable {
7         emit Paid(msg.sender, msg.value, msg.data);
8     }
9 }

```

If calldata is empty and receive exists, the EVM calls receive. If no function selector matches, the EVM calls fallback. If receive does not exist but fallback is payable, plain transfers also reach fallback.

## 4.2 Execution Context

Contracts often need to know who is calling, how much value was attached, and what the current time is. Solidity exposes read only global variables that cover these needs.

*Using msg.sender, msg.value, and block.timestamp together.* We will look at an example that combines these three globals into a small task that validates a timed payment.

```

1 contract ContextDemo {
2     uint256 public deadline;
3     constructor(uint256 secondsFromNow) {
4         deadline = block.timestamp + secondsFromNow;
5     }
6     function payBeforeDeadline() external payable {
7         require(block.timestamp <= deadline, "past deadline");
8         require(msg.value == 1 ether, "exactly 1 ether required");
9     }
10    function whoAmI() external view returns (address) {
11        return msg.sender;
12    }
13 }

```

The timestamp is suitable for deadlines and grace periods. It should not be used for randomness or fine precision because block producers have limited flexibility.

## 5 WORKED EXAMPLE: SIMPLESTORAGE

We now combine state, visibility, modifiers, and context. The owner sets an integer value and anyone can read it. This captures the typical write path and the common read path in a minimal form.

*A compact but realistic storage contract.* We will look at an example that presents the full code, then we will explain its execution trace step by step.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.26;
3 contract SimpleStorage {
4     uint256 private storedData; // not directly readable from outside
5     address public owner; // public generates a getter
6     constructor() {
7         owner = msg.sender;
8     }

```

```
9  modifier onlyOwner() {  
10     require(msg.sender == owner, "caller is not the owner");  
11     _;  
12 }  
13 function set(uint256 newValue) public onlyOwner {  
14     storedData = newValue;  
15 }  
16 function get() public view returns (uint256) {  
17     return storedData;  
18 }  
19 }
```

At deployment the constructor runs once and records the deployer as the owner. A read through `get` can be performed off chain without gas because it does not create a transaction. A write through `set` changes storage on chain after the `onlyOwner` check passes. A failing write from a non owner reverts, the state remains unchanged, and the caller pays for the work performed up to the failure point.

## REFERENCES

### A USEFUL RESOURCES

*A.0.1 Official Solidity Documentation.* The authoritative source for information on language features, syntax, compiler details, and advanced concepts. <https://docs.soliditylang.org/>

*A.0.2 Remix Project.* A powerful, browser-based Integrated Development Environment (IDE) for rapid prototyping, compiling, deploying, testing, and debugging smart contracts without local setup. <https://remix-project.org/>

*A.0.3 OpenZeppelin Contracts.* A community-vetted and professionally audited library of smart contracts that provides implementations of common standards (e.g., ERC20, ERC721) and security utilities (e.g., Ownable, ReentrancyGuard), representing industry best practices. <https://github.com/OpenZeppelin/openzeppelin-contracts>