#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 generalize digital cash to stateful applications. To motivate Solidity's design in the Ethereum setting, we introduce the account-based execution model and define the smart-contract abstraction that runs on the Ethereum Virtual Machine (EVM).

Ethereum. Ethereum is a single global state machine advanced by transactions. The world state maps addresses to accounts; each records a nonce (a per-account counter for replay protection), balance, code hash, and a storage root committing to contract storage. This account-based approach turns the chain into a stateful computer with persistent memory, enabling general-purpose smart contracts.

Smart contracts. A smart contract is code and data stored at a specific address. *Immutability* means the deployed bytecode at an address does not change; *transparency* means code and interactions are observable on chain. Many production systems add upgrade paths via proxies that forward (delegate) calls to new logic while each address's bytecode remains immutable. These properties shift trust to deterministic execution and put correctness, security, and clear state management at the center of Solidity's design.

2 SOLIDITY BUILDING BLOCKS

Solidity is a contract-oriented, **statically typed** language (types are known at compile time and must be declared) with a surface syntax similar to C++ and JavaScript. We begin with the smallest compilable file and persistent state, introduce data locations to pin down copying semantics, then discuss core types and structured state. Because many examples rely on who called and what value was attached, we surface the execution context here as well.

Conventions used in this chapter. A **transaction** executes on chain and can change state. A **call** to a `view`/`pure` function via a node's RPC interface is evaluated off chain and does not change state. **Gas** measures computational work during a transaction and is paid in Ether; off-chain reads through RPC do not spend gas because they do not create transactions.

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.

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

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. (Local variables live only during a function call and do not persist.)

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.

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

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 Data Locations: storage, memory, and calldata

Choosing the correct data location affects both semantics and cost. **storage** is persistent and lives on chain; **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 to external functions.

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.

```sol
contract Locations {
    struct Profile { string name; uint256 age; }
    mapping(address => Profile) private profiles;
    function getProfile(address user) public view returns (Profile memory p) {
        p = profiles[user]; // storage -> memory copy
    }
    function setAge(uint256 newAge) public {
        Profile storage ref = profiles[msg.sender];
        ref.age = newAge; // persistent write
    }
    function sum(uint256[] calldata xs) external pure returns (uint256 s) {
        for (uint256 i = 0; i < xs.length; i++) {
            s += xs[i];
        }
    }
}
```

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.

2.4 Core Types and Structured State

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.

Value types and sensible defaults. We will look at an example that explains why `uint256` is the common default and when smaller widths are chosen for domain clarity rather than gas micro-optimization.

```
contract ValueTypes {
    uint256 public a; // good default for arithmetic
    uint8 public b; // smaller width for constrained ranges
}
```

Smaller widths do not automatically save gas in arithmetic. Prefer `uint256` unless a narrower range better models inputs or ABI expectations.

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.

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

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. (A `mapping` is a key–value store optimized for lookups; it is not iterable by itself.)

2.5 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** for 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.

```
contract ContextDemo {
    uint256 public deadline;
    constructor(uint256 secondsFromNow) {
        deadline = block.timestamp + secondsFromNow;
    }
    function payBeforeDeadline() external payable {
        require(block.timestamp <= deadline, "past deadline");
        require(msg.value == 1 ether, "exactly 1 ether required");
    }
    function whoAmI() external view returns (address) {
        return msg.sender;
    }
}
```

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.

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 Visibility Specifiers

Public functions are callable inside and outside the contract. External functions are callable only from outside or via `this.f()`, which uses an *external call* (a new EVM call frame and calldata). Internal functions are callable in this contract and in derived contracts. Private functions are callable only in the defining contract. For state variables, the default visibility when omitted is `internal`.

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

```
contract VisibilityDemo {
    uint256 public counter;
    uint256 internal x;
    uint256 private secret;
    function inc() public {
        counter += 1;
    }
    function incExt(uint256 by) external {
        counter += by;
    }
    function _double(uint256 v) internal pure returns (uint256) {
        return v * 2;
    }
    function _bumpSecret() private {
        secret += 1;
    }
    function demoCalls() public {
        uint256 y = _double(counter);
        _bumpSecret();
        // incExt(5); // not allowed: cannot call external function internally
        this.incExt(5); // allowed: uses external call semantics and extra gas
        counter = y;
    }
}
```

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`.

3.2 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. `payable` indicates a function may receive Ether; the mechanics of value transfer are covered later.

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.

```
contract MutabilityDemo {
    uint256 private s;
    function write(uint256 v) public {
        s = v;
    }
    function read() public view returns (uint256) {
        return s;
    }
    function add(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }
}
```

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.

3.3 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.

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

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.

3.4 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; any remaining gas is returned to the caller, and state changes from the failed call are undone.

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.

```
contract Ownable {
    address public owner;
    constructor() {
        owner = msg.sender;
    }
    modifier onlyOwner() {
        require(msg.sender == owner, "caller is not the owner");
        _;
    }
    function changeOwner(address newOwner) public onlyOwner {
        owner = newOwner;
```

```
    }
}
```

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.

```
contract ErrorsDemo {
    uint256 private cap = 100;
    function setCap(uint256 newCap) public {
        require(newCap >= 10, "cap too small");
        cap = newCap;
    }
    function boundedAdd(uint256 x) public view returns (uint256) {
        require(x <= cap, "exceeds cap");
        return x + 1;
    }
}
```

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

4 VALUE TRANSFER

Value flow is central to many contracts. Receiving Ether requires explicit opt in with `payable`. Sending Ether should use the `call` pattern and check success.

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.

```
contract Bank {
    mapping(address => uint256) public balances;
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }
    function withdraw(uint256 amount) public {
        require(balances[msg.sender] >= amount, "insufficient balance");
        balances[msg.sender] -= amount; // effects first
        (bool ok, ) = payable(msg.sender).call{value: amount}("");
        require(ok, "eth transfer failed");
    }
}
```

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 gas stipend that may be insufficient for some recipients.

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.

```
contract PayableReceiver {
    event Paid(address indexed from, uint256 amount, bytes data);
    receive() external payable {
        emit Paid(msg.sender, msg.value, "");
    }
    fallback() external payable {
        emit Paid(msg.sender, msg.value, msg.data);
    }
}
```

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`.

5 A WORKING EXAMPLE: ETHERWALLET [1]

This example combines persistent state, execution context, value transfer, and function-level restrictions in a compact custodial wallet. The contract keeps Ether under its own address; anyone may send funds to it, and only the designated owner may withdraw.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;
contract EtherWallet {
    address payable public owner;
    constructor() {
        owner = payable(msg.sender);
    }
    receive() external payable {}
    function withdraw(uint256 _amount) external {
        require(msg.sender == owner, "caller is not owner");
        payable(msg.sender).transfer(_amount);
    }
    function getBalance() external view returns (uint256) {
        return address(this).balance;
    }
}
```

Structure and state. The sole piece of persistent state is `owner`. It is declared `address payable` because the owner may receive Ether from the contract. Initializing `owner` with `payable(msg.sender)` in the constructor records the deployer's address at creation time, illustrating how `msg.sender` anchors execution context during deployment.

Accepting incoming Ether. The empty `receive()` function makes the contract able to accept plain transfers of Ether. When calldata is empty and a value is sent, the EVM routes control to `receive`; no storage writes occur here, so the call is inexpensive beyond the value itself. This matches the common pattern of "anyone can deposit, the contract holds the funds."

Restricting withdrawals. The `withdraw` function is marked `external`. It first checks the caller with `require(msg.sender == owner, ...)`; only the recorded owner may move funds out. The

transfer then targets `msg.sender` cast as payable and attempts to move the requested amount of wei. If the contract balance is insufficient or the recipient cannot accept the stipend forwarded by `transfer`, the call reverts and state remains unchanged.[1]

Observing balance without writing. `getBalance` is a `view` function that returns `address(this).balance`. This reads the contract's native Ether balance directly from the EVM, contrasting with bookkeeping via a mapping as seen in a bank-style example. Because it is `view`, off-chain calls through RPC do not consume gas.

Concepts reinforced. This example reinforces (i) execution context via `msg.sender` in both constructor and function calls, (ii) value reception through `receive()` and explicit `payable`, (iii) visibility and mutability choices (`external`, `view`), and (iv) error handling with `require` to protect critical paths.

REFERENCES

[1] Solidity by Example [n. d.]. *Ether Wallet.* Solidity by Example. https://solidity-by-example.org/app/ether-wallet/

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

[1] In modern deployments, many teams prefer `(bool ok, ) = payable(to).call{value: amount}("")` and an explicit success check to avoid stipend-related failures and to better compose with contracts that need more gas in their fallback. This example uses `transfer` for compactness and to highlight its revert-on-failure behavior.