1. Student Data

The **Student Data Management project** is a Solidity-based smart contract designed to securely store and manage student records on the **Ethereum blockchain**. It demonstrates how structured data can be maintained transparently using decentralized technology.

The contract defines a **`Student` struct** containing three fields — `rollNo`, `name`, and `marks` — and stores multiple such entries in a dynamic array `students`. This enables addition and retrieval of multiple student records in a decentralized, tamper-proof environment.

Process Explanation

Struct Definition

A structure `Student` groups logically related attributes — roll number, name, and marks.

```
struct Student {
   uint256 rollNo;
   string name;
   uint256 marks;
}
```

1.

Data Storage

The contract maintains a dynamic array:

```
Student[] public students;
```

2. Being public, Solidity automatically generates a **getter function** for reading individual entries by index.

Adding a Student

The function `addStudent()` lets any user add a new student record:

```
function addStudent(uint256 _rollNo, string memory _name, uint256 _marks) public
```

3. It pushes a new `Student` struct into the array and emits a `StudentAdded` event to record this on the blockchain for transparency.

4. **Retrieving Student Data**
   The `getStudent(uint256 index)` function returns the roll number, name, and marks for a given index. It includes a **require() check** to ensure valid index access and avoid out-of-bounds errors.

5. **Counting Students**
   `getTotalStudents()` returns the total number of records stored — a quick way to know how many entries exist on-chain.

6. **Ether Handling**
   The contract includes a **fallback** and **receive** function:

   - `receive()` allows the contract to **accept Ether transfers**.

   - `fallback()` ensures any non-matching function calls or Ether transfers don't break the contract.
     These functions make the contract capable of receiving funds (though unused in this logic).

Time Complexity Analysis

Operation	Description	Time Complexity
addStudent()	Pushes one new record to dynamic array	**O(1)**
getStudent(index)	Reads struct from array by index	**O(1)**
getTotalStudents()	Returns `students.length`	**O(1)**
Event Emission	Logging to blockchain transaction logs	**O(1)** (but gas cost applies)

Fallback / Receive	No logic, just accept Ether	**O(1)**

✅ **Overall Smart Contract Complexity:**

- **Time:** Constant **O(1)** per operation
- **Space:** Grows linearly with number of students → **O(N)** for N records

Gas Usage Notes

- Each `addStudent()` transaction consumes gas proportional to **string size** and **storage cost** (writing to blockchain storage).
- Retrievals (`view` functions) are **gas-free** when called off-chain.

🗒️ **Summary**

Aspect	Details
Language / Platform	Solidity / Ethereum
Core Concept	On-chain student record storage using structs and dynamic arrays
Key Functions	`addStudent()`, `getStudent()`, `getTotalStudents()`
Complexity	O(1) per function, O(N) total storage

Blockchain Features Used	Events, fallback/receive functions, dynamic arrays
Purpose	Demonstrate secure, immutable record management on Ethereum

2. Bank Details

The `SimpleBank` smart contract is a minimal, secure, single-account banking system on Ethereum. It allows a single owner to deposit and withdraw Ether safely while anyone can send Ether to the contract. The design emphasizes security, transparency, and simplicity, integrating key patterns like reentrancy protection, ownership control, and event-driven transaction logs.

Whenever Ether is deposited—either directly, via a transaction, or through the deposit function—the contract updates the timestamp of the last deposit and emits a `Deposited` event for traceability. Withdrawals are limited strictly to the owner using a modifier `onlyOwner`, ensuring funds can't be accessed by unauthorized addresses. Reentrancy protection through a simple lock variable ensures the contract cannot be exploited during recursive calls, a common vulnerability in smart contracts.

All balance updates, ownership transfers, and withdrawals follow the Checks–Effects–Interactions pattern, maintaining strong defense against attack vectors. Because of its clean structure and O(1) operations, it is lightweight in both time complexity and gas usage, suitable for simple custodial wallets or learning examples for Solidity best practices.

🧱 Section-wise Code Explanation

Section	Description
State Variables	`owner` stores the contract owner's address (payable type for transfers). `lastDepositTimestamp` records when the last deposit was made. `locked` is a state variable used for the reentrancy guard mechanism.
Events	`Deposited`, `Withdrawn`, and `OwnershipTransferred` are emitted on key operations, ensuring transparency and enabling off-chain monitoring of activities.
Modifiers	`onlyOwner` ensures only the contract owner can perform critical actions like withdrawal or ownership transfer. `nonReentrant` prevents the same function from being re-entered before finishing execution (protection against reentrancy attacks).
Constructor	Automatically assigns the deployer as the contract owner and emits an ownership transfer event from the zero address.
Fallback & Receive Functions	Both handle Ether deposits — `receive()` is triggered when Ether is sent without calldata, and `fallback()` when data is included. Both redirect to `_deposit()` ensuring consistent behavior.
_deposit()	Core deposit logic — validates deposit amount, updates `lastDepositTimestamp`, and emits the `Deposited` event. Used by all entry points that accept Ether.
withdraw()	Allows owner to withdraw a specific amount. Uses both modifiers (`onlyOwner`, `nonReentrant`), checks balance, and transfers ETH safely using `call`.
withdrawAll()	Same as `withdraw()` but transfers the full balance to the owner. Useful for closing the account or clearing funds.
getBalance()	Returns the total Ether stored in the contract. It's a view-only function with no state modification.

transferOwnership()	Lets the current owner assign a new owner. Emits an `OwnershipTransferred` event and updates the `owner` variable.

⚙️ Time & Gas Complexity Table

Function	Purpose	Time Complexity	Gas Complexity	Remarks
constructor()	Set initial owner	O(1)	Constant (~25k)	One assignment + event
receive() / fallback()	Handle direct Ether	O(1)	Constant	Redirects to `_deposit()`
deposit()	Deposit Ether	O(1)	Constant	Updates timestamp, emits event
_deposit()	Core deposit logic	O(1)	Constant	No iteration, minimal storage write
withdraw(amount)	Partial withdrawal	O(1)	Constant	Validates & transfers ETH
withdrawAll()	Withdraw all funds	O(1)	Constant	Transfers full balance
getBalance()	Read-only check	O(1)	Constant (<200 gas)	No storage change
transferOwnership()	Change owner	O(1)	Constant	One storage write, one event

---