## 🔁 How Version1 Data is Called in Version2 Using Proxy

### 🎯 Goal:

We want to **upgrade from Version 1 (V1Marksheet)** to **Version 2 (V2Marksheet)** **without losing data**.

### 🧠 Basic Concept:

In the **Proxy Pattern**, the **data is stored in the Proxy Contract**, and the **logic comes from the implementation contract (V1, V2, V3, ...)**.

This is possible because we use:

solidity

CopyEdit

delegatecall

🔁 delegatecall means:

"Execute the code of the target contract **in the context of the calling contract's storage**."

## 🔗 What This Means for Your Project:

* 🏗 **Proxy contract** has all the **storage slots**.
* 🧠 V1, V2, V3 contracts have the **logic (functions)**.
* When you upgrade from V1 to V2, V2 functions will operate **on the same storage** used by V1.

### 🧪 Example:

#### Version 1 Storage

solidity

CopyEdit

mapping(uint => Student) public students;

This mapping gets stored in **slot X** in Proxy Contract.

#### Now you upgrade to Version 2:

solidity

CopyEdit

mapping(uint => Student) public students;

✅ **If the structure and storage order match**, V2 can read/write that same students mapping from Proxy’s storage.

## ✅ Key Rule:

To make it work, **V2 must use exactly the same storage layout as V1**, and only **add new variables at the end**.

### ✅ Do:

solidity

CopyEdit

// V1

struct Student {

string name;

uint english;

uint hindi;

uint math;

uint percentage;

}

// V2

struct Student {

string name;

uint english;

uint hindi;

uint math;

uint percentage;

uint science; // Added at end (SAFE)

}

### ❌ Don't:

solidity

CopyEdit

struct Student {

uint science; // ❌ Added at top (DANGEROUS — breaks storage order)

string name;

...

}

## 🔄 Behind the Scenes: What Happens on Network

1. You **deploy Proxy** with V1 implementation.
2. You **store data** using V1's addStudent() function.
3. Later, you call upgradeTo(V2) in Proxy.
4. Now, delegatecall sends calls to V2Marksheet.
5. Since storage layout is same, **data is still valid** and usable in V2.

**1. V1Marksheet Contract**

This contract is a simple system that stores student information, including their marks and percentage.

**Key Components:**

1. **Struct Student:**
   * A **struct** is a custom data type that can hold multiple pieces of data in one unit.
   * The Student struct here contains:
     + name: The student's name (string).
     + english: Marks in English (uint).
     + hindi: Marks in Hindi (uint).
     + math: Marks in Math (uint).
     + percentage: The student's percentage based on the sum of the English, Hindi, and Math marks.
2. **Mapping students:**
   * **Mapping** is a way of storing data in key-value pairs, similar to a dictionary in other programming languages.
   * In this contract, students maps a unique id (of type uint) to a Student struct. This allows us to store multiple students, each with a unique ID.
3. **addStudent Function:**
   * This function allows you to add a student with their marks and calculate their percentage.
   * It takes the following inputs:
     + id: Unique ID for the student.
     + name: Name of the student.
     + english, hindi, and math: Marks obtained in these subjects.
   * The percentage is calculated as (total marks / 300) \* 100, where total marks is the sum of English, Hindi, and Math.
   * **require** is used to make sure that the marks are not more than 100 for each subject. If they are, the transaction will fail.
4. **getStudent Function:**
   * This function allows you to fetch a student's details using their id.
   * It returns a Student struct, which contains the student's name, english, hindi, math, and percentage.

**Example:**

Let's say you want to add a student with the following details:

* ID = 1
* Name = "John Doe"
* English = 85
* Hindi = 90
* Math = 80

You would call addStudent as follows:

solidity

CopyEdit

addStudent(1, "John Doe", 85, 90, 80);

This would calculate the percentage for the student:

* Total marks = 85 + 90 + 80 = 255
* Percentage = (255 / 300) \* 100 = 85%

Now, you can fetch the student details:

solidity

CopyEdit

getStudent(1);

This would return the student's information:

* name: "John Doe"
* english: 85
* hindi: 90
* math: 80
* percentage: 85%

**2. Proxy Contract**

The Proxy contract is a way to upgrade a contract without changing the address of the deployed contract. This is useful when you want to update your contract logic (e.g., add new features or fix bugs) but still retain the same address.

**Key Components:**

1. **State Variables:**
   * implementation: This is the address of the contract that contains the actual business logic (in this case, it's the V1Marksheet contract). The Proxy contract delegates calls to this address.
   * admin: The address that has permission to upgrade the contract. In this case, it's the account that deployed the Proxy contract.
2. **Constructor:**
   * The constructor sets the initial implementation address (which is the address of V1Marksheet) and sets the admin as the contract creator (msg.sender).
3. **upgradeTo Function:**
   * This function allows the admin to upgrade the contract. It changes the implementation address to a new contract.
   * Only the admin can call this function to upgrade the contract.
4. **Fallback Function:**
   * The fallback function is a special function that is called when the contract doesn't have a matching function for the call being made.
   * In this case, it performs a **delegatecall** to the current implementation contract. A delegatecall means that the logic of the called contract is executed, but the state (storage) remains in the Proxy contract. This allows you to upgrade the logic (code) without affecting the data stored in the contract.
   * calldatacopy loads the data from the call, and delegatecall executes the code of the implementation contract with the same data.
5. **receive Function:**
   * This function is triggered when the contract receives Ether (value) without any data. In this case, it does nothing but allows the contract to accept Ether.

**How It Works:**

* **Initial Deployment:**
  + First, you deploy the V1Marksheet contract.
  + Then, you deploy the Proxy contract and set the implementation address to the address of the V1Marksheet contract.
* **Upgrading the Contract:**
  + When you want to upgrade the contract, you deploy a new version of the contract (let's say V2Marksheet).
  + You call the upgradeTo function to change the implementation address to the new contract address (V2Marksheet).
* **Using the Proxy Contract:**
  + Once the Proxy is deployed and the implementation is set, users can interact with the Proxy contract instead of the actual V1Marksheet contract.
  + When they call functions on the Proxy contract (like addStudent or getStudent), the call is delegated to the implementation contract (like V1Marksheet or V2Marksheet).

**Example:**

1. **Deploying V1Marksheet:**
   * Deploy V1Marksheet and get the address, say 0x123....
2. **Deploying Proxy:**
   * Deploy Proxy with the implementation address as the address of V1Marksheet (e.g., 0x123...).
   * Now, users interact with the Proxy contract.
3. **Upgrading to V2Marksheet:**
   * Deploy V2Marksheet.
   * The admin calls upgradeTo(address newImplementation) to change the implementation address to the address of V2Marksheet.
   * Now, the Proxy contract will delegate calls to V2Marksheet.
4. **Calling Functions:**
   * When a user calls the Proxy contract’s addStudent or getStudent, the call is forwarded to V1Marksheet (or V2Marksheet after the upgrade).
   * This makes the Proxy contract behave like the implementation contract, even though it’s technically a different contract.

**In Simple Terms:**

* **V1Marksheet**: The original contract where the student data and marks logic are stored.
* **Proxy**: A middleman contract that forwards the requests to the actual contract (like V1Marksheet or any upgraded version).
* **Upgrade Process**: The Proxy contract allows you to change the business logic (the contract’s code) without changing the storage or data that was already saved in the contract.

**V2Marksheet Contract Explanation (Deep Dive)**

This contract represents **Version 2 (V2)** of the Marksheet contract, where an additional subject, **Science**, is added to the student's marks.

Let's break it down thoroughly:

**Key Components of V2Marksheet Contract**

**1. Struct Student:**

* A **struct** is used to represent the data of a student.
* In **V1**, we had marks for English, Hindi, Math, and the percentage, which were calculated from these subjects.
* In **V2**, we added a new subject, **Science**. This adds a new field in the struct to store the Science marks.
  + name: The student's name (string).
  + english: Marks in English (uint).
  + hindi: Marks in Hindi (uint).
  + math: Marks in Math (uint).
  + percentage: The student's percentage (uint).
  + science: Marks in Science (uint).

In **V2**, we are using **the same storage layout** as in V1, but with an additional slot for science.

**2. Mapping students:**

* The students mapping stores all students using a unique id for each student.
* Each id maps to a Student struct that contains their details.
* The mapping(uint => Student) allows efficient storage and retrieval of student data.

**3. addOrUpdateStudent Function:**

This function adds or updates the student information based on their ID.

**Input Parameters:**

* id: The unique ID for the student (uint).
* name: The name of the student (string).
* english, hindi, math: The marks obtained by the student in the respective subjects (uint).
* science: The marks obtained in the new subject, Science (uint).

**Internal Logic:**

* **require**: It checks if the marks for all subjects are less than or equal to 100. If not, it throws an error with the message "Marks must be <= 100".
* **Total Marks Calculation**:
  + uint total = english + hindi + math + science;: The total marks are calculated by summing the individual subject marks.
* **Percentage Calculation**:
  + uint percentage = (total \* 100) / 400;: The percentage is calculated by dividing the total marks by the maximum possible marks (which is 400 in this case because now we have four subjects).
* **Store Data**:
  + The student data is saved in the mapping students[id] where id is the student's unique identifier. It stores the student's name, english, hindi, math, percentage, and science marks.

**Purpose of this function**:

* It allows the addition of new students or updating of existing students' information. If a student with the given ID already exists, their details will be updated with the new marks.

**4. getStudent Function:**

* This is a simple getter function that allows you to retrieve the details of a student by their id.
* It returns a Student struct, which includes:
  + name, english, hindi, math, percentage, and science.

This function is useful when you want to fetch the student's details after adding or updating them.

**What’s New in V2?**

The major difference between **V1Marksheet** and **V2Marksheet** is the **addition of the Science subject**. In V2, the marks for Science are stored in a new slot, and the total marks and percentage calculations are updated accordingly.

* **In V1**, we only had 3 subjects: English, Hindi, and Math, making the total out of 300 and the percentage out of 100.
* **In V2**, the total marks are now out of 400 (because there are 4 subjects: English, Hindi, Math, and Science). The percentage is calculated as (total / 400) \* 100.

**Rules to Follow While Making Version 2 Contracts (And Beyond)**

When you’re creating or upgrading to **Version 2 (V2)** of a contract (or any other version), you need to consider the following rules:

**1. Maintain Backward Compatibility (If Required):**

* When upgrading to a new version, ensure that the previous functionalities are not broken unless it's an intentional upgrade.
* In **V2**, we maintained backward compatibility because we kept the same storage layout and structure (only added a new slot for science).

**2. Incremental Updates:**

* Always aim to **incrementally update** the contract by adding new features without removing or modifying existing features.
  + In **V2**, we added a new subject (science) to the Student struct and updated the total marks and percentage calculations, but we didn’t change the existing structure significantly.

**3. Storage Layout:**

* **Storage is expensive in Ethereum**. Every time you add new variables or structs to a contract, they take up more storage space.
* When adding a new version of a contract, ensure the layout is compatible with previous versions.
  + In **V2**, we kept the same storage slot for existing fields (like english, hindi, math, percentage) but added the science field.
  + If you were to remove or rearrange the fields, you could **break the contract’s functionality** as data would be stored differently.

**4. Data Migration:**

* **Migration** of data from one version of a contract to another is a critical task when upgrading a contract.
  + For example, if you were upgrading from **V1** to **V2**, the older students’ data (from **V1**) wouldn’t have the science marks. You could write a migration function to set default values for the science field or leave it as 0.
  + This ensures that no data is lost when upgrading.

**5. Testing Before Deployment:**

* Before deploying a new version of the contract on the blockchain, **test** it thoroughly on a testnet. Ensure that:
  + All new functionalities work as expected.
  + The old functionalities still behave the same way.

**6. Gas Optimization:**

* Gas usage can become a concern when adding new features. In this case, adding science doesn’t cause a significant increase in gas costs, but adding more complex features may.
* **Optimize your functions** by reducing unnecessary state variables or computations. For example, you could optimize the percentage calculation to avoid repeating the same logic multiple times.

**7. Migration Path (Upgradeability):**

* If you're building a contract that may need to be upgraded in the future, consider using patterns like **Proxy contracts** or **delegatecalls** (as discussed earlier). These allow you to upgrade your contract logic without losing state or data.

**Example of How V2 Works**

Let’s say we have a student with the following details:

* id = 1
* name = "John Doe"
* english = 80, hindi = 70, math = 90, science = 85

When you call the function addOrUpdateStudent(1, "John Doe", 80, 70, 90, 85);, the contract will:

1. Calculate the total marks:
   * total = 80 + 70 + 90 + 85 = 325
2. Calculate the percentage:
   * percentage = (325 \* 100) / 400 = 81.25%
3. Store the student's details in the contract, including:
   * name = "John Doe"
   * english = 80
   * hindi = 70
   * math = 90
   * percentage = 81.25
   * science = 85

To fetch the details, you would call getStudent(1), and it would return:

* name = "John Doe"
* english = 80
* hindi = 70
* math = 90
* percentage = 81.25
* science = 85

**In Summary:**

* **V2Marksheet** is an upgrade to the V1 contract, adding the science subject and updating the total marks and percentage calculation.
* **Rules for Versioning**: When upgrading a contract:
  1. Maintain backward compatibility.
  2. Update data structures incrementally.
  3. Plan for data migration.
  4. Test extensively before deploying.
  5. Use gas optimizations where needed.
  6. Consider upgradeability patterns (like Proxy contracts) for future versions.

**V3Marksheet Contract Explanation (Deep Dive)**

The **V3Marksheet** contract represents an upgraded version of the Marksheet system that adds an additional field to track whether the student has passed or failed based on their percentage. This version also allows for the updating of the science marks independently and recalculating the percentage and pass/fail status.

Let’s break down **V3Marksheet** and understand how it works, as well as what improvements and considerations were made from **V1** and **V2**.

**Key Components of V3Marksheet Contract**

**1. Struct Student:**

* **New Field in V3**: The struct Student in V3 has an additional field:
  + isPass: A boolean that indicates whether the student has passed or failed based on their percentage.

The fields in the Student struct now include:

* name: Student's name.
* english, hindi, math, science: Marks for each of these subjects (uint).
* percentage: The overall percentage of marks (uint), calculated based on the total marks.
* isPass: Boolean indicating if the student has passed (true) or failed (false).

**2. Mapping students:**

* The students mapping stores each student's data using a unique id (uint) as the key.
* The mapping now stores an instance of the Student struct, which includes the student's marks in different subjects, the percentage, and their pass/fail status.

**3. addOrUpdateStudent Function:**

This function adds or updates a student's details, including:

* **Input Parameters**:
  + id: The unique identifier for the student (uint).
  + name: The name of the student (string).
  + english, hindi, math, science: The student's marks in the respective subjects (uint).
* **Logic**:
  + **Marks Validation**: It checks that the marks for all subjects are less than or equal to 100 (require statement).
  + **Total Marks Calculation**:
    - The total marks are calculated by adding the marks in English, Hindi, Math, and Science.
    - uint total = english + hindi + math + science;.
  + **Percentage Calculation**:
    - The percentage is calculated as (total \* 100) / 400, where 400 is the total possible marks (4 subjects with a max of 100 marks each).
  + **Pass/Fail Determination**:
    - The isPass boolean is set based on the condition that if the percentage is 60 or greater, the student is considered to have passed.
    - bool isPass = percentage >= 60;.
  + **Storing Student Data**: The student data, including name, english, hindi, math, percentage, science, and isPass, is stored in the students mapping for the given id.

**Purpose of this function**:

* This function is used to add a new student or update an existing student's details. It calculates the student's total marks, percentage, and pass/fail status before storing them.

**4. updateScienceMarks Function:**

This function allows updating only the science marks for a student after their initial data has been recorded.

* **Input Parameter**:
  + id: The unique student identifier (uint).
  + science: The new marks for the science subject (uint).

**Logic**:

* **Marks Validation**: The science marks are validated to ensure they are less than or equal to 100 (require statement).
* **Update Science Marks**: The science marks for the student are updated in the students mapping.
* **Recalculate Total Marks**: The total marks are recalculated by adding the updated science marks to the existing marks for other subjects.
* **Recalculate Percentage**: The percentage is recalculated based on the new total marks.
* **Recalculate Pass/Fail Status**: The isPass status is updated based on the new percentage.

**Purpose of this function**:

* This function allows modifying only the science marks for a student, recalculating their total marks, percentage, and pass/fail status.

**5. getStudent Function:**

* This function is a simple getter that allows you to retrieve the student's information by their id.
* It returns the full Student struct, which contains all the student's data: name, english, hindi, math, science, percentage, and isPass.

**What’s New in V3?**

The main new feature in **V3** compared to **V1** and **V2** is the **isPass field** and the ability to **update only the science marks** for an existing student.

* **In V3**:
  + **Pass/Fail Status**: We introduce the isPass boolean to track whether a student has passed (percentage >= 60) or failed (percentage < 60).
  + **Science Marks Update**: We allow updating just the science marks while preserving the other data (e.g., English, Hindi, Math) for the student. When the science marks are updated, the contract recalculates the total, percentage, and pass/fail status.
* **In V2**:
  + The contract did not have the ability to update individual subject marks. All subjects (including science) were updated in one go when using addOrUpdateStudent.
  + V2 did not track whether the student passed or failed. The pass/fail status could only be inferred from the percentage but was not explicitly stored.
* **In V1**:
  + V1 only had the basic functionality of adding a student with 3 subjects and calculating the percentage. It did not have any fields for tracking pass/fail status, and the data was more basic without any flexibility for updating specific subject marks.

**Important Things to Consider in V1 and V2 (in Context of V3)**

**1. Data Structure Considerations:**

* **V1** was quite simple, with only 3 subjects and no way to update or track pass/fail status.
* **V2** added more complexity by adding the science subject, but it still didn't allow updating just one subject's marks. It also didn't track pass/fail status explicitly.
* **V3** improves the structure by adding the ability to update only the science marks and by explicitly tracking the pass/fail status.

**2. Flexibility for Updates:**

* **V1** and **V2** required complete data updates for each student. In **V3**, students’ marks can be updated individually (i.e., only the science marks can be updated without affecting the rest of the subjects).
* **V3** offers more flexibility for future use cases, where only one subject might need to be updated.

**3. Pass/Fail Status:**

* **V1** and **V2** didn’t explicitly store pass/fail status. In **V3**, we make it explicit by storing it in the isPass field, which allows better tracking and querying of student status.

**4. Code Efficiency:**

* **V1** and **V2** had simpler calculations, but **V3** introduces extra logic for recalculating the percentage and pass/fail status whenever there is an update to the science marks.
* While V3 adds complexity, it is still highly flexible and modular.

**Example of How V3 Works**

Let's say we add a student with the following details:

* id = 1
* name = "Alice"
* english = 80, hindi = 70, math = 90, science = 85

Calling addOrUpdateStudent(1, "Alice", 80, 70, 90, 85) will:

1. Calculate total marks: total = 80 + 70 + 90 + 85 = 325
2. Calculate percentage: percentage = (325 \* 100) / 400 = 81.25%
3. Calculate pass status: isPass = true (because 81.25% >= 60)

The student's data will be stored in the students mapping.

Later, if you want to update just the science marks (e.g., to 95), you can call updateScienceMarks(1, 95), and the contract will:

1. Update the science marks for Alice.
2. Recalculate the total marks: total = 80 + 70 + 90 + 95 = 335
3. Recalculate the percentage: percentage = (335 \* 100) / 400 = 83.75%
4. Update the pass status to isPass = true (since 83.75% >= 60)

To retrieve Alice's updated data, you would call getStudent(1), and it will return her updated information.

## ONTRACT: V4Marksheet

solidity

CopyEdit

// SPDX-License-Identifier: MIT

* This line tells others that this contract uses the **MIT license**, which is open-source and permissive.

solidity

CopyEdit

pragma solidity ^0.8.20;

* This tells Solidity compiler to only work if version 0.8.20 or above is used. It avoids bugs and ensures compatibility.

## 📦 STRUCTS

### 1. MarksHistory

solidity

CopyEdit

struct MarksHistory {

uint english;

uint hindi;

uint math;

uint science;

uint percentage;

uint timestamp;

address transactionSender;

}

* This structure **stores a snapshot of a student’s marks** every time they’re updated.
* timestamp: When the update happened (block.timestamp gives this).
* transactionSender: Who updated the marks (tx.origin gives the user who triggered it).

### 2. Student

solidity

CopyEdit

struct Student {

string name;

uint currentEnglish;

uint currentHindi;

uint currentMath;

uint currentPercentage;

uint currentScience;

MarksHistory[] history;

}

* This keeps the **current marks** and a **list of previous marks (history)**.
* MarksHistory[] history: Stores every past change.

## 🗂 MAPPING

solidity

CopyEdit

mapping(uint => Student) public students;

* Each student has a **unique ID (uint)**.
* It maps that ID to a Student struct.

## ✍️ FUNCTION: addOrUpdateStudent

solidity

CopyEdit

function addOrUpdateStudent(...) public {

* Anyone (like a teacher or admin) can use this to **add or update** a student’s marks.

### ✅ Step-by-step Inside the Function:

solidity

CopyEdit

require(english <= 100 && hindi <= 100 && math <= 100 && science <= 100, "Marks must be <= 100");

* This ensures no marks are above 100. If so, the transaction is **reverted** (fails on the blockchain).

solidity

CopyEdit

uint total = english + hindi + math + science;

uint percentage = (total \* 100) / 400;

* It calculates the total and percentage of marks.

### 📜 Save Old Data to History

solidity

CopyEdit

if (bytes(students[id].name).length > 0) {

* Checks if a student already exists (name is not empty).

solidity

CopyEdit

MarksHistory memory prevHistory = MarksHistory({

english: students[id].currentEnglish,

hindi: students[id].currentHindi,

math: students[id].currentMath,

science: students[id].currentScience,

percentage: students[id].currentPercentage,

timestamp: block.timestamp,

transactionSender: tx.origin

});

students[id].history.push(prevHistory);

* Creates a record of **old marks**.
* Stores the **timestamp** and **address of the person** who made the change.
* Adds this history to the history array for that student.

### 🆕 Update to New Marks

solidity

CopyEdit

students[id].name = name;

students[id].currentEnglish = english;

students[id].currentHindi = hindi;

students[id].currentMath = math;

students[id].currentScience = science;

students[id].currentPercentage = percentage;

* Updates the student's data with new marks and percentage.

## 🔍 FUNCTION: getStudent

solidity

CopyEdit

function getStudent(uint id) public view returns (Student memory) {

return students[id];

}

* Anyone can view the **current details** of a student using their ID.

## 🕘 FUNCTION: getStudentHistory

solidity

CopyEdit

function getStudentHistory(uint id) public view returns (MarksHistory[] memory) {

return students[id].history;

}

* Returns the **entire history of marks** (all past versions with timestamps and addresses).

## 🔗 What Happens on the Blockchain Network?

1. ✅ **Immutability**:
   * Once a transaction is made (like updating marks), it's **permanently recorded** on the blockchain.
   * Even though the student’s current marks change, the **previous version is stored in history**.
2. ✅ **Transparency**:
   * You know **who** updated the data (transactionSender) and **when** (timestamp).
   * This increases **trust and accountability**.
3. ✅ **Gas Cost**:
   * More storage (history array) means **higher gas usage**.
   * But the benefit is that we get a full **audit trail**.

## 🔁 Difference from V1 → V4

| **Version** | **Features** |
| --- | --- |
| **V1** | Basic marks + percentage. |
| **V2** | Added science marks. |
| **V3** | Added isPass logic (>=60%), improved structure. |
| **V4** | ✅ **History of every update**,  ✅ Timestamp + sender address tracking,  ✅ Transparent versioning of marks. |

## ✅ What You Achieved with V4:

* Full **audit trail** of every change.
* More **secure and trustworthy** system.
* Can track **who made changes and when**.
* Useful for **academic proof**, student appeals, or transparency in institutions.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

contract Proxy {

    address public implementation;

    address public admin;

    constructor(address \_implementation) {

        implementation = \_implementation;

        admin = msg.sender;

    }

    function upgradeTo(address newImplementation) public {

        require(msg.sender == admin, "Only admin can upgrade");

        implementation = newImplementation;

    }

    fallback() external payable {

        address impl = implementation;

        require(impl != address(0), "Implementation not set");

        assembly {

            // Load calldata into memory

            calldatacopy(0, 0, calldatasize())

            // Delegate call to implementation

            let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)

            // Copy returned data

            returndatacopy(0, 0, returndatasize())

            // Return or revert

            switch result

            case 0 {

                revert(0, returndatasize())

            }

            default {

                return(0, returndatasize())

            }

        }

    }

    receive() external payable {}

}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

contract V1Marksheet {

    struct Student {

        string name;

        uint english;

        uint hindi;

        uint math;

        uint percentage; // out of 300

    }

    mapping(uint => Student) public students;

    function addStudent(uint id, string memory name, uint english, uint hindi, uint math) public {

        require(english <= 100 && hindi <= 100 && math <= 100, "Marks must be <= 100");

        uint total = english + hindi + math;

        uint percentage = (total \* 100) / 300;

        students[id] = Student(name, english, hindi, math, percentage);

    }

    function getStudent(uint id) public view returns (Student memory) {

        return students[id];

    }

}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

contract V2Marksheet {

    struct Student {

        string name;

        uint english;

        uint hindi;

        uint math;

        uint percentage; // same slot as V1

        uint science;    // new slot in V2

    }

    mapping(uint => Student) public students;

    function addOrUpdateStudent(

        uint id,

        string memory name,

        uint english,

        uint hindi,

        uint math,

        uint science

    ) public {

        require(english <= 100 && hindi <= 100 && math <= 100 && science <= 100, "Marks must be <= 100");

        uint total = english + hindi + math + science;

        uint percentage = (total \* 100) / 400;

        students[id] = Student(name, english, hindi, math, percentage, science);

    }

    function getStudent(uint id) public view returns (Student memory) {

        return students[id];

    }

}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

contract V3Marksheet {

    struct Student {

        string name;

        uint english;

        uint hindi;

        uint math;

        uint percentage; // same slot as V1

        uint science;    // same slot as V2

        bool isPass;     // new slot in V3

    }

    mapping(uint => Student) public students;

    function addOrUpdateStudent(

        uint id,

        string memory name,

        uint english,

        uint hindi,

        uint math,

        uint science

    ) public {

        require(english <= 100 && hindi <= 100 && math <= 100 && science <= 100, "Marks must be <= 100");

        uint total = english + hindi + math + science;

        uint percentage = (total \* 100) / 400;

        bool isPass = percentage >= 60;

        students[id] = Student(name, english, hindi, math, percentage, science, isPass);

    }

    function updateScienceMarks(uint id, uint science) public {

        require(science <= 100, "Science marks must be <= 100");

        Student storage s = students[id];

        s.science = science;

        uint total = s.english + s.hindi + s.math + s.science;

        s.percentage = (total \* 100) / 400;

        s.isPass = s.percentage >= 60;

    }

    function getStudent(uint id) public view returns (Student memory) {

        return students[id];

    }

}

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

contract V4Marksheet {

    struct MarksHistory {

        uint english;

        uint hindi;

        uint math;

        uint science;

        uint percentage;

        uint timestamp;

        address transactionSender; // Store address instead of string

    }

    struct Student {

        string name;

        uint currentEnglish;

        uint currentHindi;

        uint currentMath;

        uint currentPercentage;

        uint currentScience;

        MarksHistory[] history;

    }

    mapping(uint => Student) public students;

    // Add or Update student marks and track history

    function addOrUpdateStudent(

        uint id,

        string memory name,

        uint english,

        uint hindi,

        uint math,

        uint science

    ) public {

        require(english <= 100 && hindi <= 100 && math <= 100 && science <= 100, "Marks must be <= 100");

        uint total = english + hindi + math + science;

        uint percentage = (total \* 100) / 400;

        // Store previous marks in history

        if (bytes(students[id].name).length > 0) {

            MarksHistory memory prevHistory = MarksHistory({

                english: students[id].currentEnglish,

                hindi: students[id].currentHindi,

                math: students[id].currentMath,

                science: students[id].currentScience,

                percentage: students[id].currentPercentage,

                timestamp: block.timestamp,

                transactionSender: tx.origin // Track the transaction sender address

            });

            students[id].history.push(prevHistory);

        }

        // Update student data

        students[id].name = name;

        students[id].currentEnglish = english;

        students[id].currentHindi = hindi;

        students[id].currentMath = math;

        students[id].currentScience = science;

        students[id].currentPercentage = percentage;

    }

    // View student details with full history

    function getStudent(uint id) public view returns (Student memory) {

        return students[id];

    }

    // Get the history of a student with all previous versions of marks

    function getStudentHistory(uint id) public view returns (MarksHistory[] memory) {

        return students[id].history;

    }

}

**📘 Student Management**

1. Add student without marks (only name and ID).
2. Delete a student record.
3. Update only a specific subject mark.
4. Update student name only.
5. Freeze student data after final results (no edits allowed).
6. Mark student as "Graduated".
7. Add student photo hash (e.g., IPFS image).
8. Link Aadhar or unique student identifier hash.

**🧮 Marks & Grades**

1. Auto-assign grades based on percentage (A, B, C…).
2. Calculate class average.
3. Rank students by percentage.
4. Calculate topper (highest marks).
5. Add new subject (like Computer).
6. Remove a subject (if discontinued).
7. Track failed subjects separately.
8. Allow pass/fail per subject.
9. Weighted marks calculation (e.g., math = 1.5x).

**📅 Academic Year / Semesters**

1. Track marks per semester.
2. Add academic year to student data.
3. Switch between current and past semesters.
4. Retrieve semester-wise performance.
5. Calculate GPA/CGPA over semesters.

**⏳ History & Audit**

1. Track who updated what subject specifically.
2. Add reason/comment for mark change.
3. Rollback to previous marks version.
4. Count number of updates per student.
5. Export student data in batches.
6. Generate immutable report card hash.

**🛡 Access Control & Roles**

1. Only allow teachers to update marks.
2. Add Admin/Principal role.
3. Role-based access using msg.sender.
4. Add teacher’s signature (hash) for approval.

**🧾 Certificates & Verifications**

1. Generate a certificate hash on completion.
2. Verify student result authenticity via hash.
3. Integrate with external DApp for job/college verification.
4. NFT-based certificate/token for student success.

**📊 Analytics & Reporting**

1. Subject-wise performance trends.
2. District/Institute-wide analytics if used in multiple schools.
3. Comparison graph between two students.
4. Generate top 10 students list.