Experiment 09

AIM

To implement Smart contracts using Solidity/Python/ Java language.

THEORY

A **smart contract** is a program that runs on a blockchain (e.g., Ethereum). It encapsulates state and logic, ensuring deterministic execution and tamper resistance. Key concepts:

- **State variables**: Persisted on-chain storage.
- **Functions**: Public/external functions expose behavior; internal/private limit visibility.
- **Modifiers**: Precondition checks (e.g., onlyOwner).
- **Events**: Emitted logs that UIs and off-chain systems can subscribe to.
- **Access control**: Patterns like Ownable ensure only privileged accounts can call certain functions.
- **require/revert**: Runtime checks that abort state changes with error messages.
- **ABI**: Interface that off-chain clients (web3) use to call contract functions.

We'll build a small but realistic contract: **StudentRegistry**. It lets an owner (admin) add/update student records, while any user can read them. It demonstrates:

- Mappings & structs
- Access control
- Events & require checks
- Gas-safe patterns for updates and view calls

PROCEDURE / STEPS (Remix IDE)

A. Setup

1. Open **https://remix.ethereum.org** in your browser.
2. In the **File Explorer**, click **Create New File** → name it StudentRegistry.sol.
3. In the **Solidity Compiler** tab:
    - Select compiler version 0.8.25 (or ^0.8.20+).

- Enable **Auto compile**.

4. In the **Deploy & Run Transactions** tab:
   - Environment: **Remix VM (Cancún)** (in-memory local chain) for quick tests.
   - (Optional) To test with MetaMask: choose **Injected Provider - MetaMask** and switch to a testnet like **Sepolia**.

B. Paste Contract Code

Copy the Solidity code from the **CODE** section below into StudentRegistry.sol, save the file, and compile.

C. Deploy

1. In **Deploy & Run**, ensure the contract StudentRegistry is selected.
2. Click **Deploy**.
3. After deployment, the contract instance appears under **Deployed Contracts**.

D. Interact (Read/Write)

1. **addStudent**: Provide rollNo, name, course, and cgpa → click **transact**.
2. **getStudent**: Enter rollNo → click **call** to read values.
3. **updateCGPA**: Change CGPA for an existing student.
4. **setActive**: Toggle a student's active flag.
5. **owner**: View current owner.
6. **transferOwnership** (optional): Change owner to another address.

E. Observe Events

- Open the **Console/Logs** pane in Remix.
- Every successful write emits events (StudentAdded, StudentUpdated, StudentStatusChanged, OwnershipTransferred).

F. Troubleshooting Common Remix Errors

- **"Gas estimation errored... missing revert data"**:
  - Usually means your call would fail a require. Double-check inputs (e.g., student exists before update) and caller permissions (owner-only).
  - Try the **Remix VM** first; if using MetaMask/testnet, ensure you have test ETH and the correct network.

- **execution reverted: ...**: The revert reason string (e.g., "Not owner", "Student already exists") tells you exactly what to fix.
- **Compilation mismatch**: Use the compiler version specified in pragma (or compatible higher minor).

CODE (Solidity – Remix)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;

/**
 * @title StudentRegistry
 * @notice A simple registry to add, view, and update student
records.
 * @dev Demonstrates mappings, structs, events, access control, and
require checks.
 */
contract StudentRegistry {
    // --- Ownership (Ownable-lite) ---
    address public owner;

    modifier onlyOwner() {
        require(msg.sender == owner, "Not owner");
        _;
    }

    event OwnershipTransferred(address indexed previousOwner,
address indexed newOwner);

    // --- Student Model ---
    struct Student {
        string name;
        string course;
        uint8 cgpa;       // store CGPA in tenths (e.g., 85 = 8.5)
for uint efficiency
        bool active;
        bool exists;
    }

    mapping(uint256 => Student) private students;

    event StudentAdded(uint256 indexed rollNo, string name, string
course, uint8 cgpa, bool active);
    event StudentUpdated(uint256 indexed rollNo, string name, string
course, uint8 cgpa);
    event StudentStatusChanged(uint256 indexed rollNo, bool active);
```

```
    constructor() {
        owner = msg.sender;
        emit OwnershipTransferred(address(0), msg.sender);
    }

    // --- Admin Functions ---
    function addStudent(
        uint256 rollNo,
        string calldata name,
        string calldata course,
        uint8 cgpa,         // e.g., 85 means 8.5
        bool active
    ) external onlyOwner {
        require(rollNo != 0, "RollNo cannot be 0");
        require(!students[rollNo].exists, "Student already exists");
        require(bytes(name).length > 0 && bytes(course).length > 0,
"Empty fields");
        require(cgpa <= 100, "CGPA must be <= 10.0");

        students[rollNo] = Student({
            name: name,
            course: course,
            cgpa: cgpa,
            active: active,
            exists: true
        });

        emit StudentAdded(rollNo, name, course, cgpa, active);
    }

    function updateStudent(
        uint256 rollNo,
        string calldata name,
        string calldata course,
        uint8 cgpa
    ) external onlyOwner {
        require(students[rollNo].exists, "Student not found");
        require(bytes(name).length > 0 && bytes(course).length > 0,
"Empty fields");
        require(cgpa <= 100, "CGPA must be <= 10.0");

        Student storage s = students[rollNo];
        s.name = name;
        s.course = course;
        s.cgpa = cgpa;

        emit StudentUpdated(rollNo, name, course, cgpa);
```

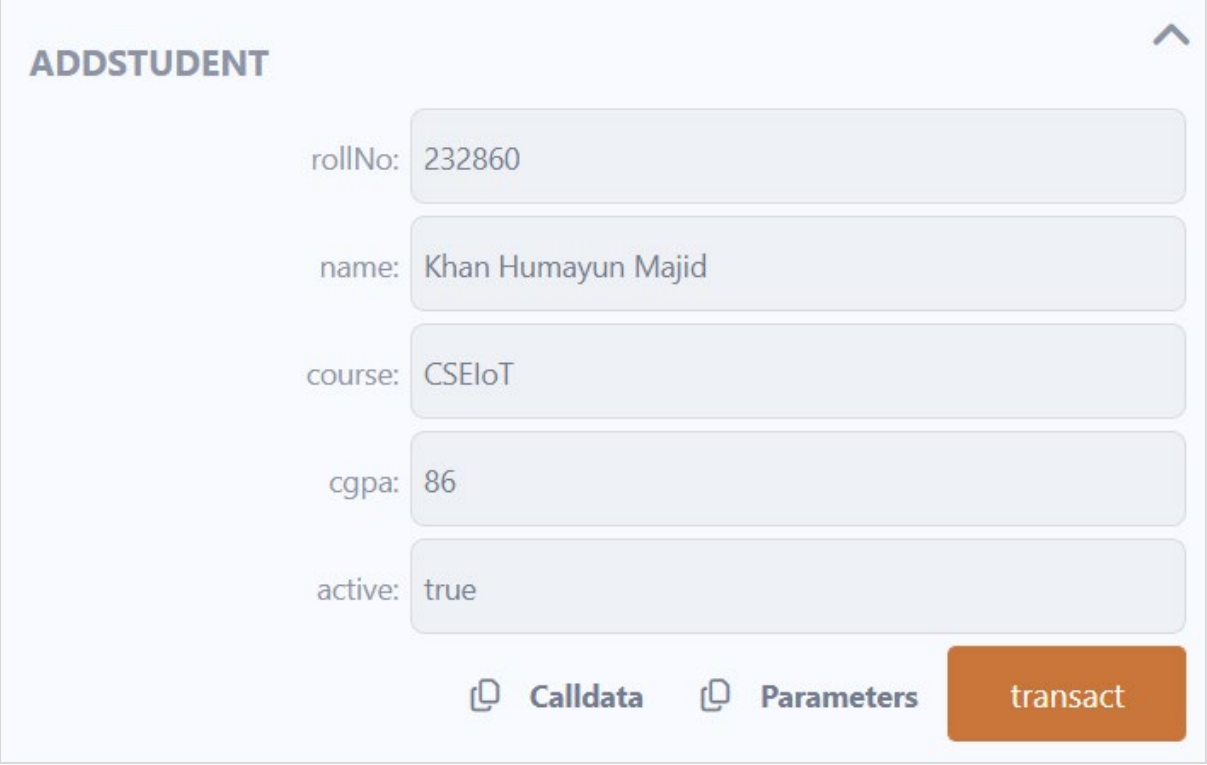
```
    }

    function updateCGPA(uint256 rollNo, uint8 cgpa) external
onlyOwner {
        require(students[rollNo].exists, "Student not found");
        require(cgpa <= 100, "CGPA must be <= 10.0");
        students[rollNo].cgpa = cgpa;
        emit StudentUpdated(rollNo, students[rollNo].name,
students[rollNo].course, cgpa);
    }

    function setActive(uint256 rollNo, bool active) external
onlyOwner {
        require(students[rollNo].exists, "Student not found");
        students[rollNo].active = active;
        emit StudentStatusChanged(rollNo, active);
    }

    function transferOwnership(address newOwner) external onlyOwner
{
        require(newOwner != address(0), "Zero address");
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }

    // --- View Functions ---
    function getStudent(uint256 rollNo)
        external
        view
        returns (string memory name, string memory course, uint8
cgpa, bool active, bool exists)
    {
        Student memory s = students[rollNo];
        return (s.name, s.course, s.cgpa, s.active, s.exists);
    }

    function studentExists(uint256 rollNo) external view returns
(bool) {
        return students[rollNo].exists;
    }
}
```

OUTPUT

Deployer/Owner: 0x5B3... (default VM account)

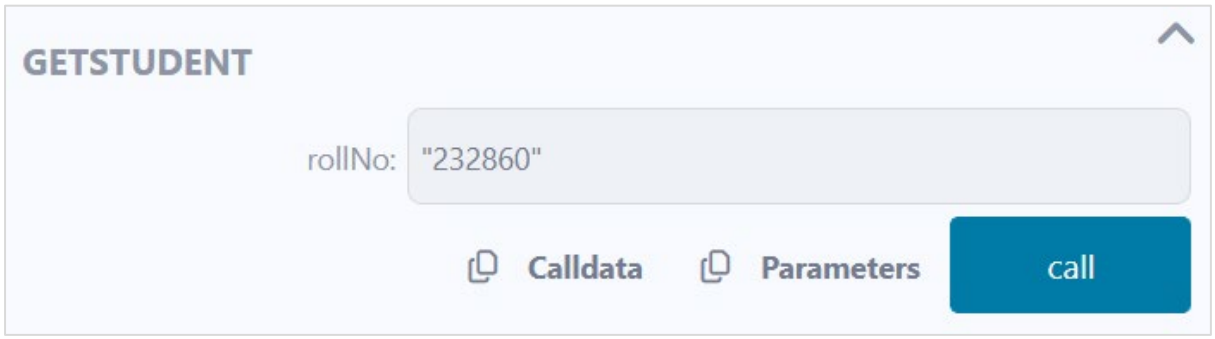
1. **Deploy** StudentRegistry → **Success**
   - Event: OwnershipTransferred(0x000..., 0x5B3...)

2. **addStudent**

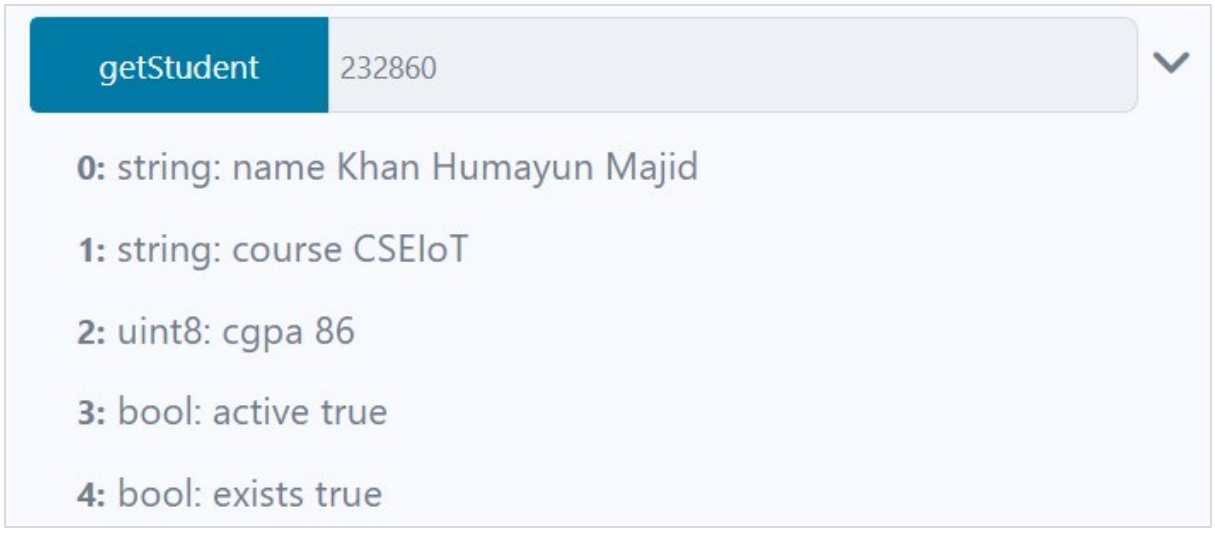


   - Transaction: **Success**
   - Event: StudentAdded(101, "Humayun Khan", "CSE (IoT & Cybersecurity)", 84, true)

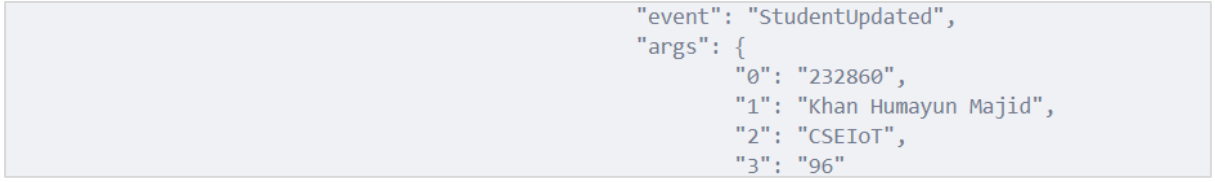
3. getStudent(232860)



- Returns:



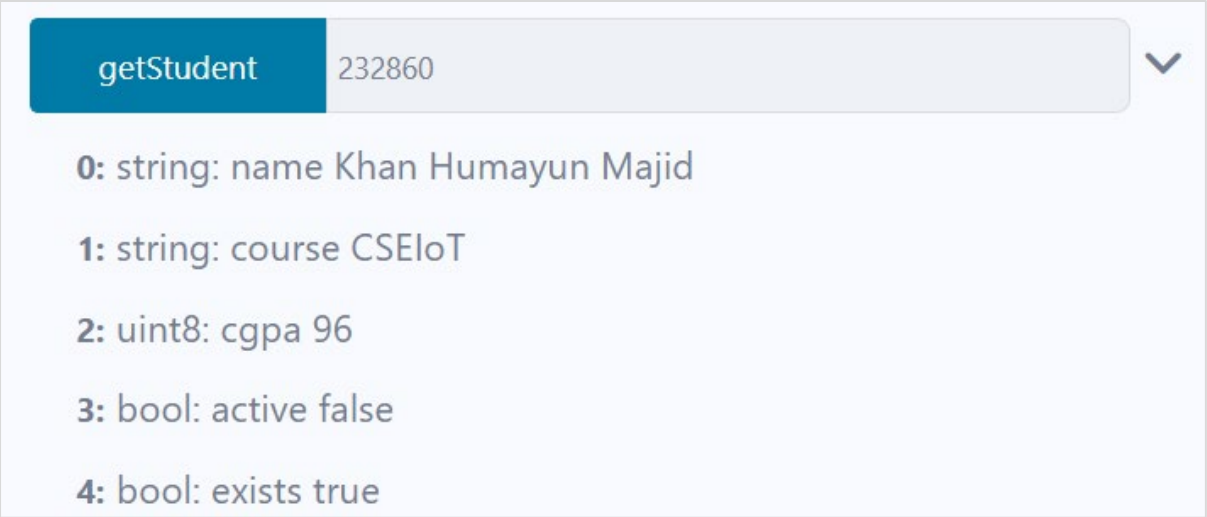
4. updateCGPA(86, 96) → Success



```
"event": "StudentUpdated",
"args": {
        "0": "232860",
        "1": "Khan Humayun Majid",
        "2": "CSEIoT",
        "3": "96"
```

5. **setActive(232860, false)**

6. **getStudent(232860)** → now shows cgpa = 90 and active = false.



CONCLUSION

We successfully implemented and tested a Solidity smart contract on Remix that manages student records with proper access control, validation, and event logging. The practical demonstrated:

- Contract structure (state, functions, events, modifiers)
- Safe update patterns and revert reasons
- Deployment & interaction via Remix VM
- How to interpret Remix logs and troubleshoot gas estimation errors

This fulfills the aim of implementing smart contracts using Solidity (primary), with awareness of cross-language interaction (optional snippets below).