Write a smart contract on a test network, for Bank account of a customer for following operations:

- **Deposit money**
- **Withdraw Money**
- **Show balance**

1. Open Remix IDE
2. In file explore create new file and paste the code
3. Open Solidity Compiler and set COMPILER as given in code and compile
4. Open Deploy & Run Transaction
i. Click on Deployed Contracts
ii. Deploy Contract
iii. Add deposite amount click on drop down and clik on trasact
iv. Add withdraw money click on drop down and click on transact
v. Now check balance and owner

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract BankAccount {
    address public owner;
    uint256 public balance;

    constructor() {
        owner = msg.sender;
        balance = 0;
    }

    modifier onlyOwner() {
        require(msg.sender == owner, "Only the owner can perform this operation");
        _;
    }

    function deposit(uint256 amount) public onlyOwner {
        require(amount > 0, "Amount to deposit must be greater than 0");
        balance += amount;
    }

    function withdraw(uint256 amount) public onlyOwner {
        require(amount > 0, "Amount to withdraw must be greater than 0");
        require(amount <= balance, "Insufficient balance");
        balance -= amount;
    }

    function getBalance() public view returns (uint256) {
        return balance;
    }
}
```

This smart contract, named "BankAccount," allows the owner (customer) to deposit and withdraw money, as well as check the account balance. Here's a breakdown of its features:

Constructor: The constructor initializes the contract by setting the owner to the address that deploys the contract (the customer) and sets the initial balance to 0.

onlyOwner Modifier: The onlyOwner modifier restricts certain functions to be callable only by the owner (customer). It checks if the sender of the transaction (msg.sender) is the owner before allowing the function to proceed.

deposit Function: The deposit function allows the owner to deposit a specified amount of money into the bank account. The deposited amount is added to the existing balance.

withdraw Function: The withdraw function permits the owner to withdraw a specified amount of money from the bank account, given that the withdrawal amount is less than or equal to the available balance.

getBalance Function: The getBalance function allows anyone, including the owner, to check the current account balance without modifying it.

Write a program in solidity to create Student data. Use the following constructs:
Structures
Arrays
Fallback
Deploy this as smart contract on Ethereum and Observe the transaction fee and Gas values.

1. Open Remix IDE
2. In file explore create new file and paste the code
3. Open Solidity Compiler and set COMPILER as given in code and compile
4. Open Deploy & Run Transaction
i. Click on Deployed Contracts
ii. Deploy Contract
iii. On addStudents Click on drop down and add rollnumber, name & age and click on transact
iv. On addStudents Click on drop down and write index(1) and call student

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract StudentRegistry {
    struct Student {
        uint rollNumber;
        string name;
        uint age;
    }
```

```
    Student[] public students;

    constructor() {
        // Initialize the contract with an empty array of students
    }

    function addStudent(uint _rollNumber, string memory _name, uint _age) public {
        students.push(Student(_rollNumber, _name, _age));
    }

    function getStudent(uint index) public view returns (uint, string memory, uint) {
        require(index < students.length, "Student not found");
        Student memory student = students[index];
        return (student.rollNumber, student.name, student.age);
    }

    fallback() external {
        // Fallback function can be used to handle unexpected calls
        revert("Invalid function call");
    }
}

 struct Student {
        uint rollNumber;
        string name;
        uint age;
    }
```

Explaination of code: -

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

- This line declares a public array named `students` of type `Student`. This array will be used to store instances of the `Student` structure, effectively creating a registry of student records. The `public` modifier makes the array publicly accessible, allowing external read access.

```
    constructor() {
        // Initialize the contract with an empty array of students
    }
```

- The constructor is a special function that runs only once when the contract is deployed. In this case, it initializes the contract with an empty array of students. This sets up the initial state of the contract.

```
  modifier onlyOwner() {
    require(msg.sender == owner, "Only the owner can perform this operation");
    _;
  }
```

- This is a custom modifier named `onlyOwner`. It checks if the sender of a transaction (msg.sender) is the owner (which will be set in the contract) and allows the function using this modifier to proceed. If the condition is not met, it reverts the transaction with the specified error message.

```
  function addStudent(uint _rollNumber, string memory _name, uint _age) public onlyOwner
{
    require(_rollNumber > 0, "Roll number must be greater than 0");
    students.push(Student(_rollNumber, _name, _age));
  }
```

- The `addStudent` function allows the owner (specified as `onlyOwner`) to add a new student to the `students` array. It takes three parameters: roll number, name, and age.
- It first checks that the roll number is greater than 0, and if not, it reverts the transaction with an error message.
- If the roll number is valid, a new instance of the `Student` structure is created with the provided data and added to the `students` array.

```
  function getStudent(uint index) public view returns (uint, string memory, uint) {
    require(index < students.length, "Student not found");
    Student memory student = students[index];
    return (student.rollNumber, student.name, student.age);
  }
```

- The `getStudent` function allows anyone to retrieve student information by providing an index. It returns the roll number, name, and age of the student at the specified index.
- It checks if the provided index is within the valid range of indices in the `students` array. If not, it reverts the transaction with an error message.

```
  fallback() external {
    // Fallback function can be used to handle unexpected calls
    revert("Invalid function call");
  }
```

- The fallback function is a special function that gets executed when the contract receives a call that doesn't match any valid functions. In this case, it simply reverts the transaction with an "Invalid function call" error message.