Object-Oriented Programming

CL-1004

Lab 05

Static Members/Functions, Constant Members/Functions, has-a Relationship, and Array of Objects

National University of Computer and Emerging Sciences–NUCES – Karachi

Contents

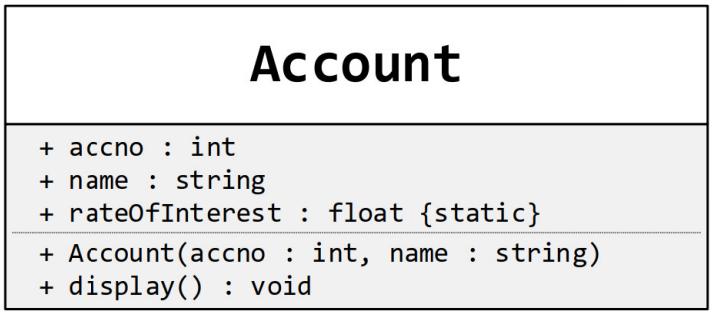
1. static Data Members & Member Functions

`static` in C++ OOP is used to declare class-level variables and member functions **that are shared by all instances of the class.** It means that the variable is initialized once and remains in memory throughout the execution of the program, and that the member function can be called directly on the class, rather than on an instance of the class.

Example 01:

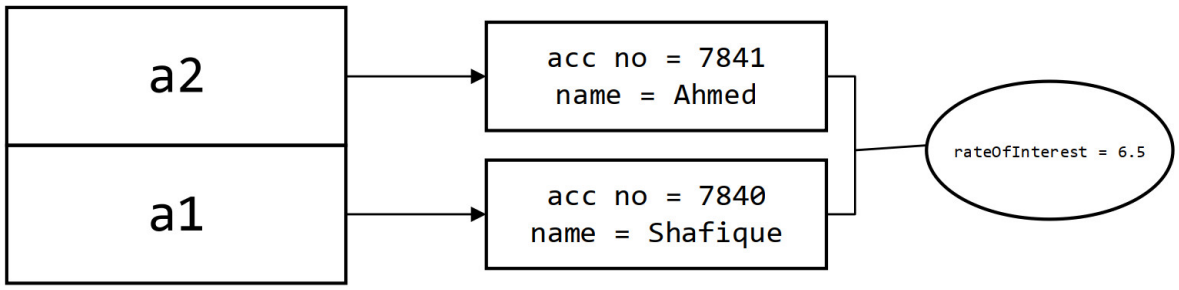
```
#include "iostream"
using namespace std;

class Account {
public:
    int accno;        //data member (also instance variable)
    string name;      // data member (also instance
                                variable)
    static float rateOfInterest; //static variable and will be shared in all
instances of objects


    Account (int accno, string name){

        this->accno = accno;
        this->name = name;
    }
    void display(){

        cout << accno << " " << name << " " << rateOfInterest << endl;
    }
};

float Account :: rateOfInterest = 6.5;

int main(){

    Account a1 = Account(7840, "Shafique");
    Account a2 = Account(7841, "Ahmed");

    a1.display();
    a2.display();

    return 0;
}
```

Output	7840 Shafique 6.5 7841 Ahmed 6.5



In above example, **rateOfInterest** is static which is shared between two instances(**a1** and **a2**) of the **Account** class. There is only one copy of static variable **rateOfInterest** created in the memory. It is shared to all the objects.

Memory mapping of the above code



A **static function** is a function that is associated with the class itself rather than any specific instance of the class. This means:

1. **Scope**: A static function belongs to the class, not to any particular object of the class.
2. **Access**: It can be called using the class name directly, without creating an instance of the class.
3. **Limitations**: A static function can only access static data members and other static functions of the class. It cannot access non-static members (instance variables or methods) because they belong to specific objects.

Example 02:

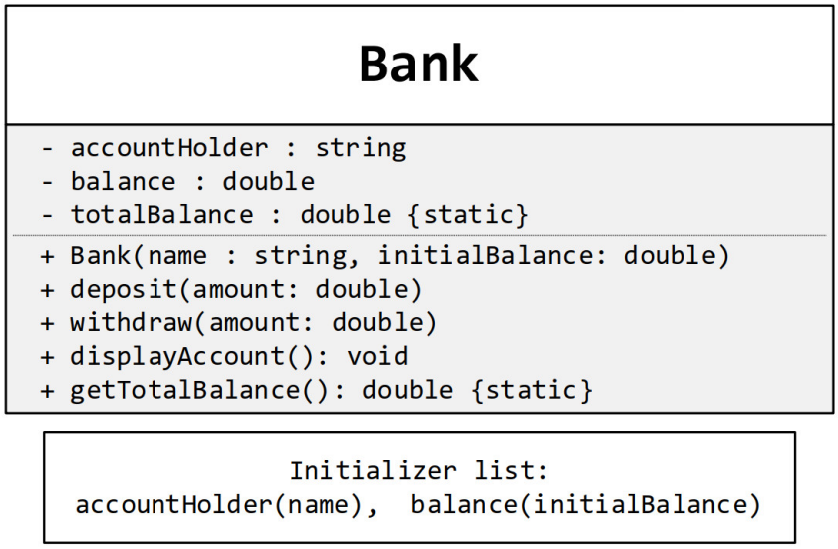
```
#include <iostream>
using namespace std;

class Bank {
private:
    string accountHolder;
    double balance;
    static double totalBalance;
```

```
public:

    Bank(string name, double initialBalance) : accountHolder(name),
balance(initialBalance) {
        totalBalance += initialBalance;
    }

    void deposit(double amount) {
        balance += amount;
        totalBalance += amount;
        cout << "Deposited " << amount << " into " << accountHolder << "'s
account." << endl;
    }

    void withdraw(double amount) {
        if (amount > balance) {
            cout << "Insufficient balance in " << accountHolder << "'s
account." << endl;
        } else {
            balance -= amount;
            totalBalance -= amount;
            cout << "Withdrawn " << amount << " from " << accountHolder <<
"'s account." << endl;
        }
    }

    static double getTotalBalance() {
        return totalBalance;
    }

    void displayAccount() {
        cout << "Account Holder: " << accountHolder << ", Balance: " <<
balance << endl;
    }
};


double Bank::totalBalance = 0;

int main() {
    // create accounts
    Bank account1("Shafique", 1000);
    Bank account2("Ahmed", 2000);

    // display initial total balance using the static function
    cout << "Initial Total Balance: " << Bank::getTotalBalance() << endl;

    // perform transactions
```

```
    account1.deposit(500);
    account2.withdraw(1000);

    // display account details
    cout << endl << "Account Details:" << endl;
    account1.displayAccount();
    account2.displayAccount();

    // display updated total balance using the static function
    cout << endl << "Updated Total Balance: " << Bank::getTotalBalance() <<
endl;

    return 0;
}
```

Output

```
Initial Total Balance: 3000
Deposited 500 into Shafique's account.
Withdrawn 1000 from Ahmed's account.

Account Details:
Account Holder: Shafique, Balance: 1500
Account Holder: Ahmed, Balance: 1000

Updated Total Balance: 2500
```



2. constant Data Members & Member Functions

In C++, **constant** data members and **constant** member functions are used to enforce immutability and ensure that certain values or behaviors cannot be modified after initialization or during execution.

Constant Data Member

- A **constant data member** is a data member of a class that cannot be modified after it is initialized.
- It must be initialized using an **initializer list** in the constructor.
- Once initialized, its value remains constant throughout the lifetime of the object.

Constant Member Function

- A **constant member function** is a function that guarantees it will not modify the state of the object on which it is called.
- It is declared by adding the const keyword at the end of the function signature.
- Constant member functions can only call other constant member functions and cannot modify non-static data members.

Example 01:

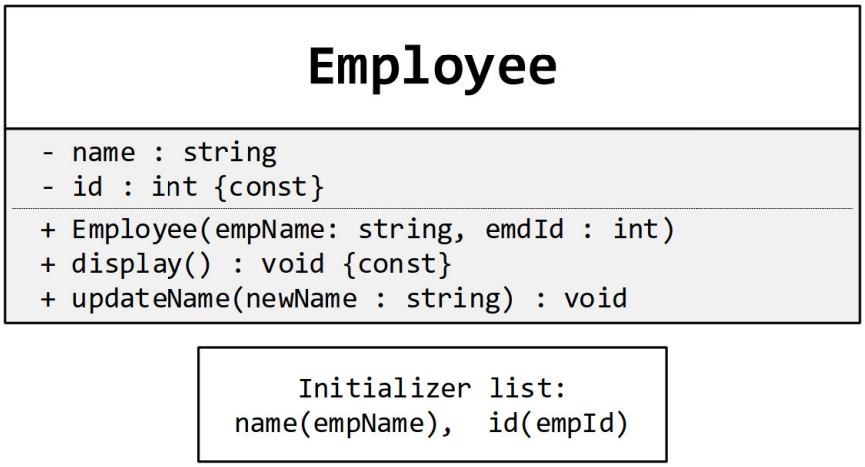
A class named **Employee** is created that stores an employee's **name** and **id**. The **id** is a **constant data member** because it should not change after initialization. A **constant member function** to display employee details without modifying the object is also implemented.

```
#include <iostream>
using namespace std;

class Employee {
private:
    string name;
    const int id;

public:
    // constructor to initialize constant and non-constant data members
    Employee(string empName, int empId) : name(empName), id(empId) {}

    // constant member function to display employee details
    void displayDetails() const {
        cout << "Employee Name: " << name << endl;
        cout << "Employee ID: " << id << endl;
    }

    // non-constant member function to update the employee's name
    void updateName(string newName) {
        name = newName;
        cout << "Name updated to: " << name << endl;
    }
};

int main() {
```

```
    Employee emp("Abdul", 101);

    // display employee details using a constant member function
    cout << "Initial Employee Details:" << endl;
    emp.displayDetails();

    // update the employee's name using a non-constant member function
    emp.updateName("Abdul Rehman");

    // display updated employee details
    cout << endl << "Updated Employee Details:" << endl;
    emp.displayDetails();

}
```

Output

```
Initial Employee Details:
Employee Name: Abdul
Employee ID: 101
Name updated to: Abdul Rehman

Updated Employee Details:
Employee Name: Abdul Rehman
Employee ID: 101
```



The constant data member (**id**) is declared as **const int id** and is initialized using the constructor's initializer list. Once initialized, the **id** cannot be modified, ensuring its value remains constant throughout the object's lifetime. The constant member function (**displayDetails**) is declared with the **const** keyword, meaning it guarantees that it will not modify any non-static data members of the class. This makes it safe to call **displayDetails** on a **const** object, as it only reads data without altering the object's state. On the other hand, the non-constant member function (**updateName**) modifies the **name** data member, which means it cannot be declared as **const**. This function is used to update the **name** of the employee, demonstrating how non-constant functions can change the

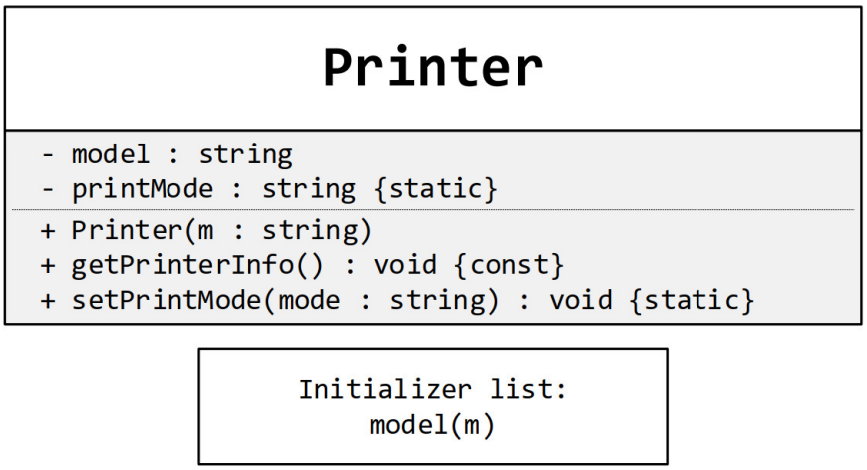
object's state. Together, these concepts illustrate how const enforces immutability for data members and ensures safe access to object data through constant member functions

Example 02:

Consider a **Printer** class where each printer object has a model name, but all printers share a **common printer mode** (e.g., **"Color"** or **"Black & White"**). The constant function **getPrinterInfo()** reads the printer details but also **modifies the static print mode**.

```
#include <iostream>
using namespace std;

class Printer {
private:
    string model;
    static string printMode; // shared mode for all printers

public:
    Printer(string m) : model(m) {}


    void getPrinterInfo() const {
        printMode = "Black & White"; // // constant function modifying a
static data member
        cout << "Printer Model: " << model << ", Mode: " << printMode <<
endl;
    }

    static void setPrintMode(string mode) {
        printMode = mode;
    }
};


string Printer::printMode = "Color";

int main() {
    Printer p1("HP LaserJet");
    Printer p2("Canon Pixma");

    p1.getPrinterInfo(); // modifies print mode to "Black & White"
    p2.getPrinterInfo(); // reflects the updated mode

}
```

Output	Printer Model: HP LaserJet, Mode: Black & White Printer Model: Canon Pixma, Mode: Black & White



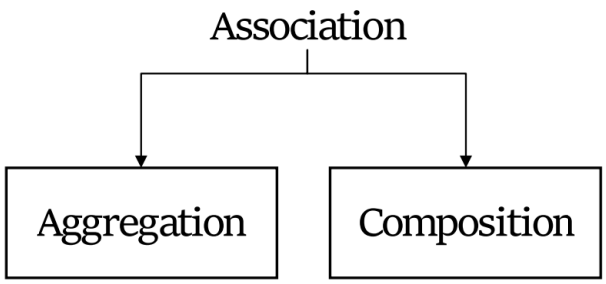
3. has-a Relationship

3.1. Association

A "**has-a**" relationship, also known as **association**, represents a connection between two classes where one class contains or owns another class as a member. In this relationship, one class has another class as a member, typically through a pointer or reference. The class that contains the other class is referred to as the owner or container, while the class that is contained is referred to as the member or component.

In simpler terms, a "**has-a**" relationship implies that one class possesses or is associated with another class. This association can be **one-to-one**, **one-to-many**, or **many-to-many**. The containing class provides access to the contained class's functionality and may manipulate or interact with it in various ways.

For example, a **car** "**has-a**" engine, a department "**has-a**" manager, or a library "**has-a**" collection of books. These relationships help model real-world scenarios in object-oriented programming and enable building more complex systems by composing simpler components.



3.1.1. Aggregation

Aggregation is a specialized form of association where objects have a "**whole-part**" relationship, but the parts can exist independently of the whole. The whole may contain parts, but the parts are not dependent on the existence of the whole. It represents a weaker form of ownership. **For example**, a **Department** class can have multiple **Employee** objects, but the employees can exist independently of the department.

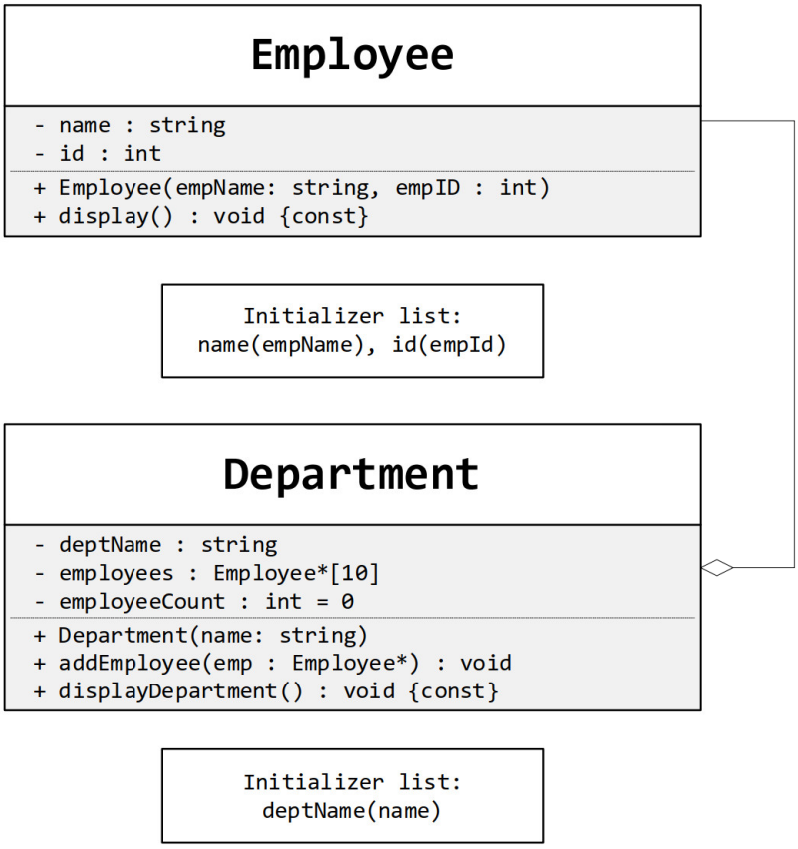
Example 01:

```
#include <iostream>
using namespace std;

class Employee {
public:
    string name;
    int id;

    Employee(string empName, int empId) : name(empName), id(empId) {}

    void display() const {
        cout << "Employee ID: " << id << ", Name: " << name << endl;
    }
};

class Department {
private:
    string deptName;
    Employee* employees[10];
    int employeeCount = 0;

public:
    Department(string name) : deptName(name) {}

    void addEmployee(Employee* emp) {
        employees[employeeCount++] = emp;
    }

    void displayDepartment() const {
        cout << "Department: " << deptName << endl << "Employees:" << endl;
        for (int i = 0; i < employeeCount; i++) employees[i]->display();
    }
};

int main() {
    Employee e1("Shaheen Shah Afridi", 10), e2("Muhammad Rizwan", 16),
e3("Babar Azam", 56);
    Department dept("Software Engineering");
```

```
    dept.addEmployee(&e1);
    dept.addEmployee(&e2);
    dept.displayDepartment();

    cout << endl << "Independent Employee:" << endl;
    e3.display();

    return 0;
}
```

Output

```
Department: Software Engineering
Employees:
Employee ID: 10, Name: Shaheen Shah Afridi
Employee ID: 16, Name: Muhammad Rizwan

Independent Employee:
Employee ID: 56, Name: Babar Azam
```



3.1.2. Composition

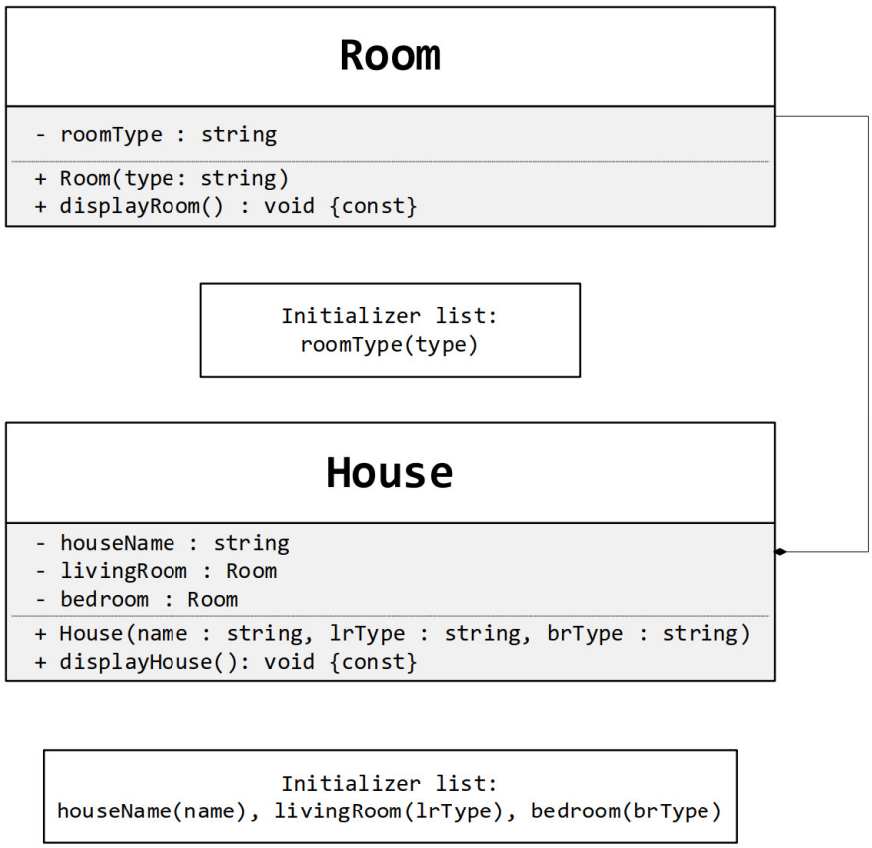
Composition is a stronger form of association where the parts are tightly coupled to the whole. The parts are created and destroyed along with the whole. If the whole is destroyed, all its parts are destroyed as well. It represents a stronger form of ownership.

For example, a **House** object might contain **Rooms**, and when the **House** is destroyed, all its **Rooms** are also destroyed.

Example 01:

```cpp
#include <iostream>
using namespace std;

class Room {
private:
    string roomType;

public:
    Room(string type) : roomType(type) {}

    void displayRoom() const {
        cout << "Room Type: " << roomType << endl;
    }
};

class House {
private:
    string houseName;
    Room livingRoom;
    Room bedroom;

public:
    House(string name, string lrType, string brType)
        : houseName(name), livingRoom(lrType), bedroom(brType) {}

    void displayHouse() const {
        cout << "House Name: " << houseName << endl;
        livingRoom.displayRoom();
        bedroom.displayRoom();
    }
};

int main() {

    House myHouse("Dream Villa", "Living Room", "Master Bedroom");
    myHouse.displayHouse();

}
```

Output	House Name: Dream Villa Room Type: Living Room Room Type: Master Bedroom



5. Array of Objects

In C++, you can create **arrays of objects** just like you create arrays of **primitive data types**. Arrays of objects allow you to store **multiple objects** of the **same class** in contiguous memory locations.

Example 01:

```
#include "iostream"
using namespace std;

class MyClass{
public:
    int num;

    MyClass(){

        num = 0;
    }
```

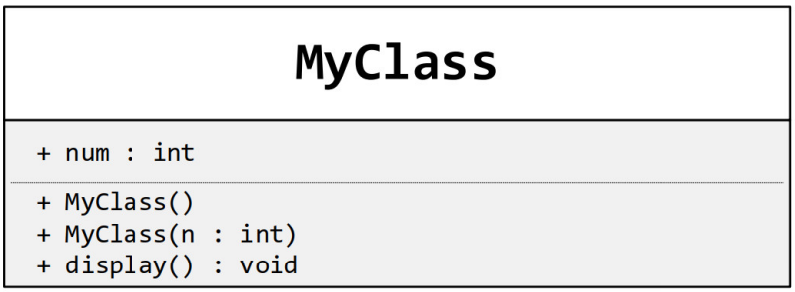
```
    MyClass(int n){

        num = n;
    }

    void display() {

        cout << "Number: " << num << endl;
    }
};

int main(){

    const int size = 5;
    MyClass arr[size]; // array of MyClass objects

    // initializing objects in the array
    for (int i = 0; i < size; i++){

        arr[i] = MyClass(i+1); // initializing with numbers 1 through 5
    }

    // displaying objects in the array
    for (int i = 0; i < size; i++){

        arr[i].display(); // displaying the number of each object
    }

}
```

Output	Number: 1 Number: 2 Number: 3 Number: 4 Number: 5

In this example:

- We define a class **MyClass** with a member variable **num**.
- There are two constructors: one default constructor which initializes **num** to **0**, and another constructor that takes an integer parameter and sets **num** to the value of the parameter.
- The **display()** method prints the value of **num** for an object.
- In the **main()** function, we create an array **arr** of **MyClass** objects with a size of **5** (size).

- We initialize each object in the array with a number from **1** to **5** using a for loop.
- Finally, we iterate through the array and call the **display()** method for each object to print its number.
- This demonstrates how you can use arrays of objects in C++ to manage multiple objects of the same class efficiently.



Example 02:

```
#include <iostream>
#include <cstring>
using namespace std;

class Employee{

    int id;
    char name[30];
public:

    void take_data(int id, char name[]){
        this->id = id;
        strcpy(this->name, name); // using strcpy() to copy name
    };

    void display_data(){
        cout << id << " " << name << " " << endl;
    };
};


int main(){

    Employee emp[30];
    int n, i;

    cout << "Enter number of employees: ";
    cin >> n;
```

```
    for (int i = 0; i < n; i++){

        int id;
        char name[30];

        cout << "Enter Id: "; cin >> id;
        cout << "Enter Name: "; cin >> name;

        emp[i].take_data(id, name);
    }

    cout << "Employee Data" << endl;

    for (int i = 0; i < n; i++){

        emp[i].display_data();
    }

}
```

Output

```
Enter number of employees: 5
Enter Id: 1
Enter Name: ABC
Enter Id: 2
Enter Name: EFG
Enter Id: 3
Enter Name: HIJ
Enter Id: 4
Enter Name: KLM
Enter Id: 5
Enter Name: NOP
Employee Data
1 ABC
2 EFG
3 HIJ
4 KLM
5 NOP
```