Introduction to Information & Communication Technologies

CL-1000

Lab 11

Introduction to Object-Oriented Programming | OOP

Part - 02

National University of Computer & Emerging Sciences – NUCES – Karachi

National University of Computer & Emerging Sciences – NUCES – Karachi

Course Code: CL-1000	Introduction to Information & Communication Technologies

Contents

1. Encapsulation

Encapsulation refers to bundling data and the methods that operate on that data within a single unit. It is the concept of combining data and the functions that manipulate it into one cohesive class. Encapsulation also facilitates data abstraction or hiding.

For example, consider an ATM machine. As a user, you can perform actions like withdrawing money or checking your balance, but you don't have direct access to the internal processes or data, such as the system managing your account or validating your PIN. These internal details are hidden from you and handled securely by the system, illustrating encapsulation.

Encapsulation is achieved through classes and access modifiers. Its two primary attributes are:

1. **Data Protection:** Safeguards the internal state of an object by keeping data members private.
2. **Information Hiding:** Conceals the internal implementation details of a class from external entities.

Example: Encapsulation

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

class ATM {
private:
    double accountBalance; // Data member for account balance

    // Private method to validate PIN
    bool validatePIN(int inputPIN) {
        const int correctPIN = 1234; // Example PIN
        return inputPIN == correctPIN;
    }

public:
    // Constructor to initialize account balance
    ATM(double initialBalance) : accountBalance(initialBalance) {}

    // Public method to withdraw money
    void withdrawMoney(int inputPIN, double amount) {
        if (validatePIN(inputPIN)) {
            if (amount <= accountBalance) {
                accountBalance -= amount;
                cout << "Withdrawal successful! Remaining balance: $" <<
accountBalance << endl;
            } else {
                cout << "Insufficient balance." << endl;
            }
        } else {
            cout << "Invalid PIN." << endl;
        }
    }
```

```
    // Public method to check account balance
    void checkBalance(int inputPIN) {
        if (validatePIN(inputPIN)) {
            cout << "Current balance: $" << accountBalance << endl;
        } else {
            cout << "Invalid PIN." << endl;
        }
    }
};

int main() {
    ATM myATM(1000.00); // Initialize ATM with $1000 balance

    int pin;
    cout << "Enter your PIN: ";
    cin >> pin;

    int choice;
    do {
        cout << endl << "ATM Menu:" << endl;
        cout << "1. Withdraw Money" << endl;
        cout << "2. Check Balance" << endl;
        cout << "3. Exit" << endl;
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice) {
        case 1: {
            double amount;
            cout << "Enter amount to withdraw: ";
            cin >> amount;
            myATM.withdrawMoney(pin, amount);
            break;
        }
        case 2:
            myATM.checkBalance(pin);
            break;
        case 3:
            cout << "Thank you for using the ATM. Goodbye!" << endl;
            break;
        default:
            cout << "Invalid choice. Please try again." << endl;
        }
    } while (choice != 3);

    return 0;
}
```

Output	**First Output:** Enter your PIN: 1234

```
ATM Menu:
1. Withdraw Money
2. Check Balance
3. Exit
Enter your choice: 1
Enter amount to withdraw: 500
Withdrawal successful! Remaining balance: $500

ATM Menu:
1. Withdraw Money
2. Check Balance
3. Exit
Enter your choice: 1
Enter amount to withdraw: 500
Withdrawal successful! Remaining balance: $0

ATM Menu:
1. Withdraw Money
2. Check Balance
3. Exit
Enter your choice: 1
Enter amount to withdraw: 500
Insufficient balance.

ATM Menu:
1. Withdraw Money
2. Check Balance
3. Exit
Enter your choice: 2
Current balance: $0

ATM Menu:
1. Withdraw Money
2. Check Balance
3. Exit
Enter your choice: 3
Thank you for using the ATM. Goodbye!
```

Second Output:

```
Enter your PIN: 123

ATM Menu:
1. Withdraw Money
2. Check Balance
3. Exit
Enter your choice: 1
Enter amount to withdraw: 500
```

	Invalid PIN. ATM Menu: 1. Withdraw Money 2. Check Balance 3. Exit

1.1 Key Concepts

1.1.1 Access Specifiers (Private and Public):

- **`private`**: Members (data and functions) declared under the **`private`** access specifier can only be accessed by other members (functions) of the **`class`**. This ensures **data protection** and hides the internal implementation.
    - Example: **`accountBalance`** and **`validatePIN`** are **`private`** in the **`ATM class`**, so they cannot be accessed directly from the **`main()`** function.
- **`Public:`** Members declared under the **`public`** access specifier can be accessed directly from outside the **`class`**.
    - Example: The **`withdrawMoney`** and **`checkBalance`** methods are **`public`**, so they can be called from the **`main()`** function.

1.1.2 Encapsulation:

- Encapsulation is achieved by wrapping **`accountBalance`** and its related functions (**`validatePIN, withdrawMoney,`** and **`checkBalance`**) into the **`ATM class`**.
- The **`ATM class`** hides its implementation details (like how **`PIN`** validation or account balance updates are performed) and provides a clean interface for external usage through **`public`** methods.

1.1.3 Constructor:

- The constructor **`ATM(double initialBalance)`** initializes the account balance when the object (**`myATM`**) is created.

1.1.4 Validation Logic:

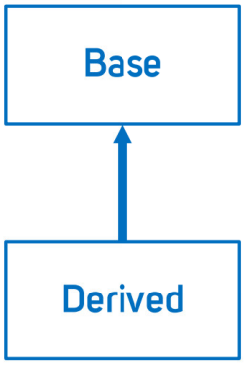
- The **`private`** method **`validatePIN`** ensures that only authorized users can access account operations (withdrawal or balance check).
- The **`PIN`** is validated in both the **`withdrawMoney`** and **`checkBalance`** methods.

1.1.5 Class Methods:

- **`withdrawMoney(int inputPIN, double amount):`**
    - First validates the **`PIN`**.
    - Checks if the withdrawal amount is less than or equal to the account balance. If valid, deducts the amount and displays the updated balance.
- **`checkBalance(int inputPIN):`**
    - Validates the **`PIN`** and displays the current balance if valid.

2. Inheritance

Inheritance is a mechanism in object-oriented programming where new classes are created based on an existing `class`. The newly created class is referred to as the "**`derived class`**," while the existing **`class`** is called the "**`base class`**." The **`derived class`** acquires all the properties and methods of the **`base class`** without modifying them and can also introduce additional features of its own.



2.1 Why use Inheritance?

Imagine a scenario involving a group of vehicles. You need to define classes for Bus, Car, and Truck. These vehicles share common methods like **`fuelAmount()`**, **`capacity()`**, and **`applyBrakes()`**. Writing these methods separately in each `class` would lead to code duplication, increased complexity, and a higher risk of errors. Inheritance helps solve this problem by allowing these shared methods to be defined once in a base class (e.g., **`Vehicle`**) and inherited by all derived classes, making the code more organized and efficient.

Example: Inheritance

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

// Base class
class Vehicle {
public:
    void fuelAmount() {
        cout << "Checking fuel amount..." << endl;
    }

    void capacity() {
        cout << "Checking vehicle capacity..." << endl;
    }

    void applyBrakes() {
        cout << "Applying brakes..." << endl;
    }
};

// Derived class for Bus
```

```cpp
class Bus : public Vehicle {
public:
    void busSpecificFunction() {
        cout << "This is a function specific to the Bus class." << endl;
    }
};

// Derived class for Car
class Car : public Vehicle {
public:
    void carSpecificFunction() {
        cout << "This is a function specific to the Car class." << endl;
    }
};

// Derived class for Truck
class Truck : public Vehicle {
public:
    void truckSpecificFunction() {
        cout << "This is a function specific to the Truck class." << endl;
    }
};

int main() {
    // Creating objects of derived classes
    Bus myBus;
    Car myCar;
    Truck myTruck;

    cout << "Bus actions:" << endl;
    myBus.fuelAmount();
    myBus.capacity();
    myBus.applyBrakes();
    myBus.busSpecificFunction();

    cout << "\nCar actions:" << endl;
    myCar.fuelAmount();
    myCar.capacity();
    myCar.applyBrakes();
    myCar.carSpecificFunction();

    cout << "\nTruck actions:" << endl;
    myTruck.fuelAmount();
    myTruck.capacity();
    myTruck.applyBrakes();
    myTruck.truckSpecificFunction();

    return 0;
}
```

Output	Truck actions:

	Checking fuel amount... Checking vehicle capacity... Applying brakes... This is a function specific to the Truck class.

2.1.1 Key Concepts

1. **Base Class (Vehicle)**:
   - Contains common methods like **fuelAmount()**, **capacity()**, and **applyBrakes()**.
   - These methods are shared by all derived classes.
2. **Derived Classes (Bus, Car, Truck):**
   - Each inherits the common methods from the **Vehicle class** using the **public** access specifier (here they are called mode of inheritance.
   - They can also have their own unique methods, such as **busSpecificFunction()**, **carSpecificFunction()**, and **truckSpecificFunction()**.
3. **Main Function:**
   - Creates objects for each derived class.
   - Demonstrates how the derived classes can access methods from the base class as well as their own specific methods.

2.3 Modes of Inheritance

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

class Parent{
public:
    int x;

private:
    int y;

protected:
    int z;
};

class child1 : public Parent{
    // x will remain public
    // y will remain protected
    // z will not be accessible
};

class child2 : private Parent{
    // x will private
    // y will private
    // z will be inaccessible
};

class child3 : protected Parent
{
    // x will remain protected
```

```
    // y will remain protected
    // z will be inaccessible
};
```

3. Polymorphism

Polymorphism refers to the ability of a single entity to take on multiple forms. For example, in real life, a man can simultaneously be a **father, husband**, and **son**, depending on the context. Similarly, an operation can behave differently based on the data types involved.

3.1 Types of Polymorphism

3.1.1 Compile-time Polymorphism

This type of polymorphism occurs during the compilation process and is implemented through **function overloading** or **operator overloading**.

1. **Function Overloading**

   Function overloading happens when multiple functions share the same name but have different parameters. This allows grouping multiple tasks under a single function name.

2. **Operator Overloading**

   Operator overloading allows operators to perform additional tasks specific to user-defined data types. For example, while the + operator is generally used for arithmetic addition, it can also be overloaded to concatenate strings.

Example: Function Overloading

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

class Calculator {
public:
    // method to add two integers
    int add(int a, int b) {
        return a + b;
    }

    // method to add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }

    // method to add two double numbers
    double add(double a, double b) {
        return a + b;
    }
```

```
};

int main() {
    Calculator calc;

    // calls the method with two integers
    cout << "Sum is: " << calc.add(3, 4) << endl;

    // calls the method with three integers
    cout << "Sum is: " << calc.add(1, 2, 3) << endl;

    // calls the method with two doubles
    cout << "Sum is: " << calc.add(2.5, 3.5) << endl;

    return 0;
}
```

Output	Sum is: 7 Sum is: 6 Sum is: 6

3.1.2 Runtime Polymorphism

Runtime polymorphism is determined during program execution, where the appropriate function to call is decided based on the object. It is also referred to as **late binding polymorphism** and is achieved through **function overriding**.

1. **Function Overriding**

   Function overriding occurs when a derived class provides its own implementation of a member function that is already defined in the base class. In this case, the derived class's version of the function replaces the base class's version.

Example: Function Overriding

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


class Animal {
public:
    // virtual method to describe the sound an animal makes
    virtual void makeSound() {
        cout << "This animal makes a generic sound." << endl;
    }
};


class Dog : public Animal {
public:
    // overriding the makeSound method
```

```
    void makeSound() {
        cout << "The dog barks!" << endl;
    }
};


class Cat : public Animal {
public:
    // overriding the makeSound method
    void makeSound() {
        cout << "The cat meows!" << endl;
    }
};

int main() {
    Animal* animalPtr; // pointer to base class
    Dog dog;
    Cat cat;

    // pointing to a Dog object
    animalPtr = &dog;
    animalPtr->makeSound(); // calls the overridden method in Dog

    // pointing to a Cat object
    animalPtr = &cat;
    animalPtr->makeSound(); // calls the overridden method in Cat

    return 0;
}
```

Output	The dog barks! The cat meows!

3.1.3 Key Concepts:

1. **`Base Class Animal`**:
    - The **`makeSound()`** function is declared as **`virtual`**, allowing function overriding in derived classes.
2. **`Derived Classes Dog and Cat`**:
    - Both classes redefine the **`makeSound()`** method from the **`base class`**.
3. **Polymorphism**:
    - The **`base class`** pointer (**`animalPtr`**) dynamically determines which **`makeSound()`** method to call based on the object it points to.

4. Abstraction

Abstraction focuses on presenting only the essential details to users while concealing the underlying complexities and processes running in the background that handle data manipulation.

For instance, consider using a mobile phone: we know how to operate it, but we don't understand the intricate workings of its internal components like RAM or the hard disk. Similarly, in programming, we use some already defined functions like the **`pow()`** function to calculate powers without knowing the algorithm that performs the calculation. This concept of hiding the implementation details is also referred to as abstraction in header files.

4.1 Types of Abstraction

1. **Data Abstraction**: Highlights only the necessary details about the data while hiding irrelevant information.
2. **Control Abstraction:** Displays the essential details of an implementation, keeping unnecessary complexities out of view.