4. Encapsulation

4.1 What is Encapsulation?

Encapsulation is the bundling of data (attributes) and methods that operate on that data within a single unit (class), while **restricting direct access** to some of the object's components.

Purpose:

- **Data Hiding**: Protect internal state from unauthorized access
- **Controlled Access**: Provide public methods to access/modify private data
- **Flexibility**: Change internal implementation without affecting external code
- **Validation**: Enforce rules when setting data

4.2 Access Specifiers

Three Access Levels:

Specifier	Access Within Class	Access in Derived Class	Access from Outside
`private`	✓	✗	✗
`protected`	✓	✓	✗
`public`	✓	✓	✓

Visual Representation:

```
┌──────────────────────────────────┐
│         Class: BankAccount       │
├──────────────────────────────────┤
│ private:                         │
│   - balance (hidden)             │ ← Cannot access from outside
│   - accountNumber (hidden)       │
├──────────────────────────────────┤
│ public:                          │
│   + deposit(amount)              │ ← Can access from outside
│   + withdraw(amount)             │
│   + getBalance()                 │
└──────────────────────────────────┘
```

4.3 Implementing Encapsulation

Bad Practice (No Encapsulation):

```
class BankAccount {
public:
    string accountNumber;
    double balance;  // Anyone can modify this directly!
};

int main() {
    BankAccount acc;
    acc.balance = 1000.0;

    // Problem: Direct modification allows invalid states
    acc.balance = -500.0;  // Negative balance! Bad!
    acc.balance = 999999999.99;  // Unrealistic amount

    return 0;
}
```

Good Practice (With Encapsulation):

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

class BankAccount {
private:
    string accountNumber;
    double balance;
    string ownerName;

public:
    // Constructor
    BankAccount(string accNum, string owner, double initialBalance = 0.0) {
        accountNumber = accNum;
        ownerName = owner;
        balance = (initialBalance >= 0) ? initialBalance : 0.0;
    }
```

```
// Getter methods (accessors)
double getBalance() const {
    return balance;
}

string getAccountNumber() const {
    return accountNumber;
}

string getOwnerName() const {
    return ownerName;
}

// Setter methods with validation (mutators)
bool deposit(double amount) {
    if (amount > 0) {
        balance += amount;
        cout << "Deposited: $" << amount << endl;
        return true;
    }
    cout << "Invalid deposit amount!" << endl;
    return false;
}

bool withdraw(double amount) {
    if (amount > 0 && amount <= balance) {
        balance -= amount;
        cout << "Withdrawn: $" << amount << endl;
        return true;
    }
    cout << "Invalid withdrawal or insufficient funds!" << endl;
    return false;
}

void displayInfo() const {
    cout << "Account: " << accountNumber << endl;
    cout << "Owner: " << ownerName << endl;
    cout << "Balance: $" << balance << endl;
}
```

```
};

int main() {
    BankAccount acc("ACC001", "Alice Johnson", 1000.0);

    acc.displayInfo();
    cout << endl;

    acc.deposit(500.0);
    acc.withdraw(200.0);
    acc.withdraw(2000.0);  // Will fail - insufficient funds

    cout << "\nFinal balance: $" << acc.getBalance() << endl;

    // acc.balance = -500;  // ERROR: Cannot access private member

    return 0;
}
```

Output:

```
Account: ACC001
Owner: Alice Johnson
Balance: $1000

Deposited: $500
Withdrawn: $200
Invalid withdrawal or insufficient funds!

Final balance: $1300
```

4.4 Benefits of Encapsulation

1. Data Validation:

```
class Temperature {
private:
    double celsius;

public:
```

```
    void setCelsius(double temp) {
        if (temp >= -273.15) {  // Absolute zero
            celsius = temp;
        } else {
            cout << "Invalid temperature!" << endl;
        }
    }

    double getCelsius() const { return celsius; }
    double getFahrenheit() const { return (celsius * 9.0/5.0) + 32; }
};
```

2. Read-Only Properties:

```
class Person {
private:
    string ssn;  // Social Security Number

public:
    Person(string socialSecNum) : ssn(socialSecNum) {}

    // Only getter, no setter - SSN is read-only
    string getSSN() const { return ssn; }
};
```

3. Internal Implementation Changes:

```
class Circle {
private:
    double radius;
    // We could change to store diameter instead later

public:
    void setRadius(double r) {
        if (r > 0) radius = r;
    }

    double getArea() const {
        return 3.14159 * radius * radius;
    }
```

```
    // External code doesn't need to change if we modify internal storage
};
```

4.5 Const Member Functions

Purpose: Indicate that a method does not modify object state

```
class Rectangle {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    // Const member functions - promise not to modify data
    double getWidth() const { return width; }
    double getHeight() const { return height; }
    double getArea() const { return width * height; }
    double getPerimeter() const { return 2 * (width + height); }

    // Non-const member functions - can modify data
    void setWidth(double w) { width = w; }
    void setHeight(double h) { height = h; }
};

int main() {
    const Rectangle rect(5.0, 3.0);  // Const object

    cout << rect.getArea();      // OK - const function
    // rect.setWidth(10);        // ERROR - can't call non-const function on const object

    return 0;
}
```

Revision #2
Created 2025-11-09 13:16:26 UTC by DS
Updated 2025-11-09 13:17:00 UTC by DS