Object-Oriented Programming

Contents

Object-Oriented Programming

1. Approach

- Object-Oriented Programming (OOPs) is a programming approach that organizes code into **objects** and **classes** and makes it more structured and easier to manage.
- For example, people who have used **C programming language**, knows that we organize the code using **functions** and **procedures**.
    - **Procedures**: It is essentially a **function that performs a specific task** but does **not return a value.** It can take inputs **(parameters)** even though it does not return a value.

```c
void greetUser(char name[])
{
    printf("Hello, %s!\n", name);
}
```

    - **Functions**: A **function** is a **block of code** that can take input (parameters), **perform some operation**, and **optionally return a value**.

```c
int add(int a, int b)
{
    return a + b;
}
```

- Summary Table:

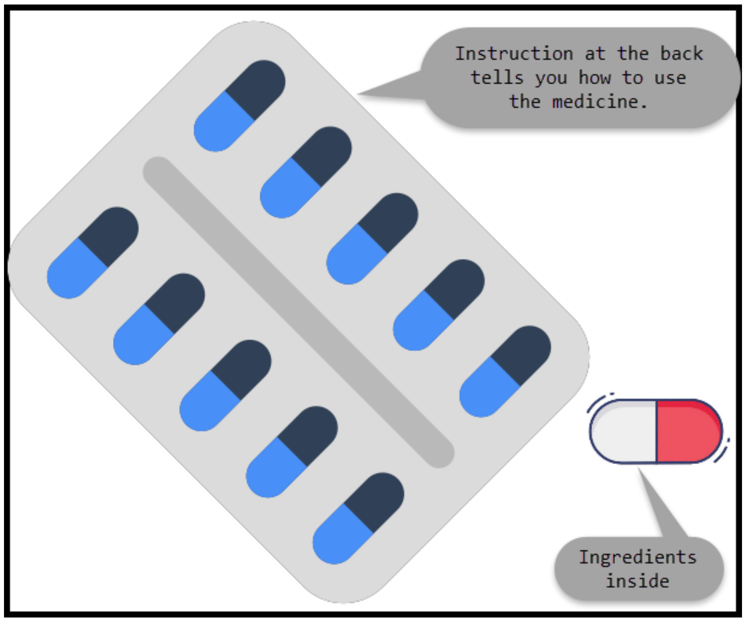
No.	Feature	C (Procedural)	OOP (e.g., C++, Java)
1	**Approach**	Function-based	Object-based
2	**Data Handling**	Globally accessible	Data hidden in objects
3	**Reusability**	Limited	High (via inheritance)
4	**Security**	Low	High
5	**Modularity**	Through functions	Through classes/objects
6	**Flexibility**	Less	More (supports overloading, etc.)

- We understood the 1 (Approach/Paradigm) in the above table.

2. Encapsulation

Note: When you are designing a class in an object-oriented language, following the principle of encapsulation is highly recommended and considered good practice.

- Imagine you have a medicine capsule. Inside the capsule are the actual ingredients (our data). You can't directly touch or change these **ingredients**. Instead, the capsule itself has instructions printed on it (our public interface or methods). These instructions tell you how to use the medicine – maybe "take one capsule with water" or "dissolve under the tongue."



- In programming, encapsulation is like that capsule:
  - **Private Data (The Ingredients Inside):**
    - This is the information or variables that an object holds.
    - We make this data "private" so that it can't be directly accessed or modified from outside the object.
    - This protects the data from accidental changes that could break things.
  - **Public Interface (The Instructions on the Capsule):**
    - These are the methods (functions that belong to the object) that allow you to interact with the object's data in a controlled way.
    - These methods define how you can access or modify the private data. They act as a gatekeeper.

- Why do we do this "data hiding" and "bundling"?
    - **Protection** (Like the **capsule protect**ing the **medicine**): By making data **private**, we prevent external code from directly messing with the internal workings of our object. This makes our code more robust and less prone to errors. If someone tries to change the data directly, the programming language will usually prevent it.
    - **Control** (Following the **instructions ensures correct usage**): The **public methods** provide a controlled way to interact with the data. We can define rules and logic within these methods to ensure that the data is modified in a valid way. For example, a method to set someone's **age** might include a check to make sure the **age** isn't **negative**.
    - **Flexibility** and **Maintainability** (If the ingredients change, the instructions might stay the same): If we need to change how the data is stored internally, as long as we keep the public methods the same (the "instructions" remain consistent), the code that uses our object won't break. The internal implementation can change without affecting the external interface.
- Example:

```
#include <iostream>
#include <string>

using namespace std;

class Car {
private:
   // Private member variables
   string engine = "V6";
   int fuelLevel = 50;

public:
   void accelerate()
   {
      cout << "Car is accelerating." << endl;
      fuelLevel -= 1; // Modifying private data within a public method
   }
```

```cpp
   int getFuelLevel()
   {
      return fuelLevel; // Providing controlled access to private data
   }

   // New public method to set the fuel level with validation
   void setFuelLevel(int newFuelLevel)
   {
      if (newFuelLevel >= 0 && newFuelLevel <= 100)
      {
         fuelLevel = newFuelLevel;
         cout << "Fuel level set to: " << fuelLevel << endl;
      }
      else
      {
         cout << "Invalid fuel level. "<<
            "Please enter a value between 0 and 100.\n";
      }
   }
};

int main() {
   Car myCar;

   cout << "Initial fuel level: " << myCar.getFuelLevel() << endl;

   // Attempting to set an invalid fuel level
   myCar.setFuelLevel(-100);
   cout << "Current fuel level after attempting "
      <<"to set invalid value : " << myCar.getFuelLevel() << endl;

   // Setting a valid fuel level
   myCar.setFuelLevel(80);
   cout << "Current fuel level after setting "
      <<"a valid value : " << myCar.getFuelLevel() << endl;

   myCar.accelerate();
   cout << "Current fuel level after acceleration: "
      << myCar.getFuelLevel() << endl;

   return 0;
}
```

- Some other examples:
    - `class`: `Student`
    - `private`: `name`, `rollNumber`, `marks`
    - `public`: `setName(), getName(), setMarks(), getMarks()`

```python
class Student:
    def __init__(self, name, roll_number):
        # Private attributes (With double underscore)
        self.__name = name
        self.__roll_number = roll_number
        self.__marks = {}  # Dictionary to store marks for different subjects

    # Public methods to set and get the name
    def set_name(self, new_name):
        if isinstance(new_name, str) and new_name.strip():
            self.__name = new_name
        else:
            print("Invalid name format.")

    def get_name(self):
        return self.__name

    # Public methods to set and get marks
    def set_marks(self, subject, mark):
        if isinstance(subject, str) and subject.strip() \
            and isinstance(mark, (int, float)):
            if 0 <= mark <= 100:  # Assuming marks are out of 100
                self.__marks[subject] = mark
            else:
                print(f"Invalid mark for {subject}. \
                      Marks should be between 0 and 100.")
        else:
            print("Invalid subject or mark format.")

    def get_marks(self, subject):
        if subject in self.__marks:
            return self.__marks[subject]
        else:
            return f"Marks for {subject} not found."

    def get_all_marks(self):
        return self.__marks
```

```python
# Creating a Student object
student1 = Student("Alice", "S101")

# Accessing and modifying data using public methods
student1.set_name("Alice Smith")
print(f"Name: {student1.get_name()}")

student1.set_marks("Math", 95)
student1.set_marks("Science", 88)
print(f"Marks in Math: {student1.get_marks('Math')}")
print(f"All marks: {student1.get_all_marks()}")

# Attempting to set invalid marks
student1.set_marks("History", 110)
student1.set_marks("English", -5)

# In Python, name mangling provides a degree of privacy, but it's not strict.
# You can still technically access 'private' attributes, but it's discouraged.
# Let's see how Python name mangles the attribute name:
# print(student1._Student__name) # This would work,
# but it's not the intended way
# The intended way is to use the public interface (the methods).
```

- class: BankAccount
- private: balance, accountNumber
- public: deposit(), withdraw(), checkBalance()

```csharp
using System;

public class BankAccount
{
    // Private member variables
    private decimal balance;
    private string accountNumber;

    // Constructor to initialize the BankAccount object
    public BankAccount(string accountNumber, decimal initialBalance)
    {
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
    }
    // Public method to deposit funds
    public void Deposit(decimal amount)
    {
        if (amount > 0)
        {
            balance += amount;
            Console.WriteLine($"Deposited ${amount}. New balance: ${balance}");
        }
        else
        {
            Console.WriteLine("Deposit amount must be positive.");
        }
    }

    // Public method to withdraw funds
    public void Withdraw(decimal amount)
    {
        if (amount > 0)
        {
            if (balance >= amount)
            {
                balance -= amount;
                Console.WriteLine($"Withdrew ${amount}. New balance: ${balance}");
            }
            else
            {
                Console.WriteLine("Insufficient balance.");
            }
        }
        else
        {
            Console.WriteLine("Withdrawal amount must be positive.");
        }
    }

    // Public method to check the current balance
    public decimal CheckBalance()
    {
        return balance;
    }

    // Public method to get the account number (read-only access)
    public string GetAccountNumber()
    {
        return accountNumber;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        // Create a BankAccount object
        BankAccount myAccount = new BankAccount("1234567890", 1000.00m);

        Console.WriteLine($"Account Number: {myAccount.GetAccountNumber()}");
        Console.WriteLine($"Initial Balance: ${myAccount.CheckBalance()}");

        // Perform some transactions using public methods
        myAccount.Deposit(500.00m);
        myAccount.Withdraw(200.00m);
        myAccount.Withdraw(1500.00m); // Attempt to withdraw more than the balance

        Console.WriteLine($"Current Balance: ${myAccount.CheckBalance()}");

        // Attempting to directly access private members (compile-time error)
        // myAccount.balance = -100.00m;
        // Console.WriteLine(myAccount.balance);
    }
}
```

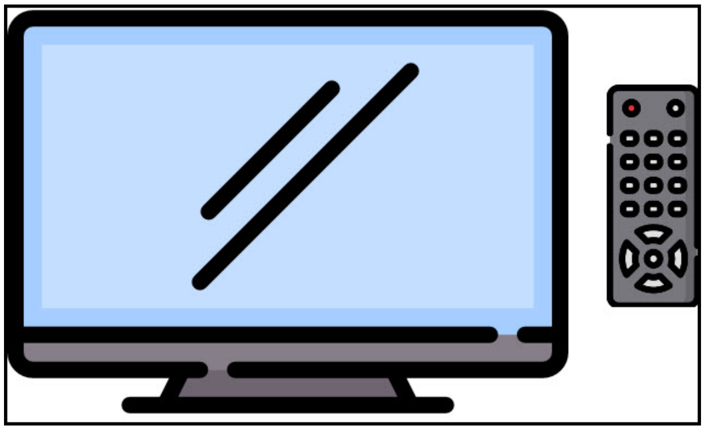
- Class: TemperatureSensor
- Private: celsius
- Public: setCelsius(), getFahrenheit()

```
public class TemperatureSensor {
    private double celsius; // Private member variable to store temperature

    // Public method to set the temperature in Celsius
    public void setCelsius(double celsius) {
        this.celsius = celsius;
    }

    // Public method to get the temperature in Fahrenheit
    public double getFahrenheit() {
        // Conversion formula: Fahrenheit = (Celsius * 9/5) + 32
        return (this.celsius * 9.0 / 5.0) + 32;
    }

    // Optional: Public method to get the temperature in Celsius (if needed)
    public double getCelsius() {
        return this.celsius;
    }

    public static void main(String[] args) {
        TemperatureSensor sensor = new TemperatureSensor();

        // Setting the temperature using the public method
        sensor.setCelsius(25.0);
        System.out.println("Temperature in Celsius: " + sensor.getCelsius());

        // Getting the temperature in Fahrenheit using the public method
        double fahrenheit = sensor.getFahrenheit();
        System.out.println("Temperature in Fahrenheit: " + fahrenheit);

        // Attempting to directly access the private variable (compile-time error)
        // sensor.celsius = 30.0; // This line would cause an error

        // You can only change the Celsius value through the
        public interface (setCelsius method)
        sensor.setCelsius(35.5);
        System.out.println("New Temperature in Celsius: " + sensor.getCelsius());
        System.out.println("New Temperature in Fahrenheit: " + sensor.getFahrenheit());
    }
}
```

3. Abstraction

Note: When you are designing a class in an object-oriented language, focus on what an object does rather than how it does.

- Imagine you're using a TV remote control.
    - **What you see** (**the abstraction**): Buttons for power, volume up/down, channel up/down, mute, etc. Each button has a clear function. You press the volume up button, and the volume goes up. You press the channel down button, and the channel changes.
    - **What's hidden** (**the internal details**): Inside the remote and the TV are complex electronic circuits, microprocessors, and wireless communication protocols working together. You don't need to understand how these intricate details function to use the remote effectively. You just care that pressing the "volume up" button achieves the desired outcome.



- In programming, abstraction is similar:
    - It's about showing only the essential information and functionality to the user (**to the developer using your class**) and hiding the complex implementation details. You focus on **what an object does** rather than **how it accomplishes it**.
- Why is Abstraction Important in Programming?
    - **Simplifies Complexity**: It allows developers to work with complex systems without getting bogged down in the low-level details. This makes the code easier to understand, use, and manage.

- **Reduces Code Duplication**: By abstracting common functionalities into reusable components (like methods or classes), you avoid writing the same code multiple times.
- **Increases Maintainability**: If the internal implementation of a feature changes, as long as the external interface (the methods and properties you expose) remains the same, the code that uses your abstraction won't break. You can modify the "**how**" without affecting the "**what.**"

- Example: Design and implement a software system to manage the shopping cart functionality for an e-commerce platform. This system should allow users to
    - Add items to their cart
    - Apply discount coupons
    - Generate a final bill reflecting the items, quantities, prices, and any applied discounts.

```
// Supports:
// 1. Adding items to cart
// 2. Applying coupons
// 3. Generating final bills.
class BillingSystem {
public:
   void addItem(const std::string& itemName, double price);
   void applyCoupon(const std::string& code);
   void generateFinalBill();

private:
   std::vector<std::pair<std::string, double>> items;
   std::string couponCode;
   void PrintBill(double total, double discount, double coupon,
      double tax, double finalAmount);
   double calculateSubtotal();
   double calculateDiscount(double total);
   double applyCouponDiscount(double amountAfterDiscount);
   double calculateTax(double afterCoupon);
};
```

- Focus on "What" the System Does, Not "How":
    - The `public` methods (`addItem`, `applyCoupon`, `generateFinalBill`) represent the **essential actions** that the user of my `class` can perform with

the billing system. They tell the user of my `class` *what* the system can do: add items, apply discounts, and produce a bill.

  - As a user of the `BillingSystem` `class` (like in the `main()` function which is given below), he/she calls these methods without needing to know the steps involved in storing the items, checking coupon validity, calculating discounts, or formatting the bill.

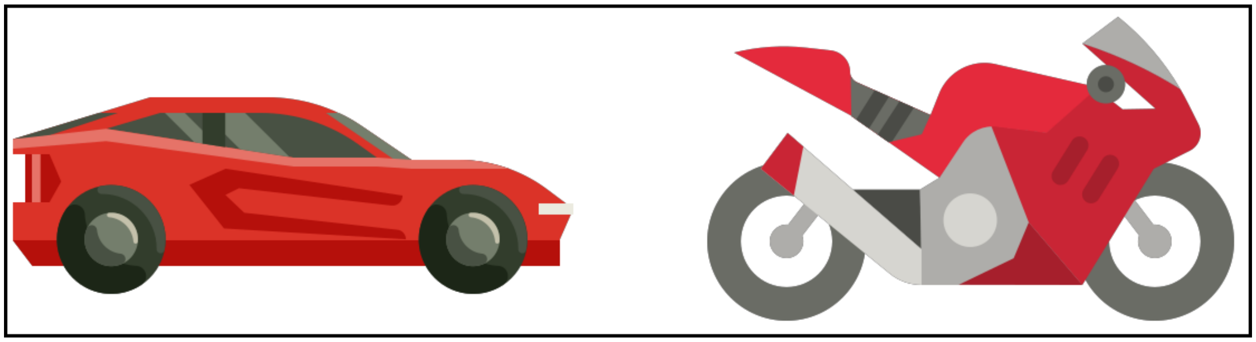
```
int main(void)
{
   BillingSystem bill;

   bill.addItem("Laptop", 3000);
   bill.addItem("Mouse", 50);
   bill.addItem("Headphones", 150);

   bill.applyCoupon("SAVE100");

   bill.generateFinalBill();

   return 0;
}
```

- Hiding Internal Complexity:
  - The `private` members (`items`, `couponCode`, `PrintBill`, `calculateSubtotal`, `calculateDiscount`, `applyCouponDiscount`, `calculateTax`) represent the internal mechanisms and data structures used by the `BillingSystem` to perform its tasks.
  - These `private` details are hidden from the outside. The `main` function doesn't directly interact with the `items` vector/list or the individual calculation methods. **This hiding of complexity is a key aspect of abstraction**. You don't need to know how the items are stored internally (e.g., using a vector of pairs) to add an item. You just use the `addItem` method.
- In essence, the `BillingSystem` `class` provides an abstraction over the complexities of calculating a bill. It exposes a clean and understandable interface (`addItem`, `applyCoupon`, `generateFinalBill`) while hiding the underlying implementation details (how items are stored, how discounts and taxes are calculated).

4. Inheritance

Note: When you are designing a class in an object-oriented language, establish "**is-a**" relationships **to reuse existing code** and build upon the common characteristics of more general concepts.

- Let's imagine you are developing a racing Game:
  - You have a blueprint for a general `Vehicle`. This blueprint includes common features that most vehicles have, like:
    - `startEngine()`
    - `stopEngine()`
    - `accelerate()`
    - `brake()`
- Now, you want to create blueprints for more specific types of vehicles, like a `Car` and a `Motorcycle`.



- Inheritance is like saying, "Hey, `Car` **and** `Motorcycle`**, you are both types of Vehicles**, so you **automatically get all the features from the Vehicle blueprint**!"
- Instead of writing the `startEngine()`, `stopEngine()`, `accelerate()`, and `brake()` functions again from scratch for both `Car` and `Motorcycle`, they inherit these functionalities from the `Vehicle` blueprint.
- Think of it as a family tree:
  - `Vehicle` is the parent class (or superclass, or base class).
  - `Car` and `Motorcycle` are the child classes (or subclasses, or derived classes).
- The children **inherit** characteristics (in our case, functionalities or methods) from their parent.

- But wait, Car and Motorcycle are also different!
    - A Car might have a openDoors() method and a turnOnRadio() method, which a Motorcycle doesn't have.
    - A Motorcycle might have a lean() method, which a Car doesn't have.
- Inheritance allows child classes to:
    - **Reuse code**: They automatically get the properties and methods of the parent class.
    - **Add new features**: Child classes can have their own unique properties and methods that are specific to their type. Car gets openDoors(), and Motorcycle gets lean().
    - **Override or modify existing features**: A child class can redefine a method it inherited from the parent class to behave in a way that's specific to it. For example, the accelerate() method in a SportsCar might behave differently (more aggressively) than the accelerate() method in a regular Car.
- "Is-a" Relationship:
    - Inheritance establishes an "**is-a**" relationship between the classes.
        - A Car **is a** Vehicle.
        - A Motorcycle **is a** Vehicle.
- Example:

```cpp
class Vehicle {
public:
   void startEngine() {
      std::cout << "Vehicle engine started.\n";
   }
   void stopEngine() {
      std::cout << "Vehicle engine stopped.\n";
   }
   void accelerate() {
      std::cout << "Vehicle is moving.\n";
   }
   void brake() {
      std::cout << "Vehicle is slowing down.\n";
   }
};
```

```
class Car : public Vehicle { // Car inherits from Vehicle
public:
   void openDoors() {
      std::cout << "Car doors opened.\n";
   }
   void turnOnRadio() {
      std::cout << "Radio turned on.\n";
   }
   // Override the accelerate method for Car
   void accelerate() {
      std::cout << "Car is accelerating quickly!\n";
   }
};
class Motorcycle : public Vehicle { // Motorcycle inherits from Vehicle
public:
   void lean() {
      std::cout << "Motorcycle is leaning into the turn.\n";
   }
};
int main() {
   Car myCar;
   Motorcycle myMotorcycle;

   // Using inherited methods
   myCar.startEngine();     // Inherited from Vehicle
   myMotorcycle.stopEngine(); // Inherited from Vehicle
   std::cout << std::endl;
   // Using methods specific to the subclass
   myCar.openDoors();
   myMotorcycle.lean();
   std::cout << std::endl;
   // Using an overridden method
   myCar.accelerate();        // Calls the Car's accelerate method
   myMotorcycle.accelerate(); // Calls the Vehicle's accelerate method

   return 0;
}
/*
Vehicle engine started.
Vehicle engine stopped.

Car doors opened.
Motorcycle is leaning into the turn.

Car is accelerating quickly!
Vehicle is moving.
*/
```

- Some more examples:
  - `class`: `Animal` → **Derived**: `Dog`, `Cat`, `Bird`
  - **Base**: `eat()`, `sleep()`
  - **Derived** adds: `bark()`, `meow()`, `fly()`

```
class Animal {
public:
   void eat() {
      std::cout << "Animal is eating.\n";
   }
   void sleep() {
      std::cout << "Animal is sleeping.\n";
   }
};

class Dog : public Animal {
public:
   void bark() {
      std::cout << "Dog is barking.\n";
   }
};

class Cat : public Animal {
public:
   void meow() {
      std::cout << "Cat is meowing.\n";
   }
};

class Bird : public Animal {
public:
   void fly() {
      std::cout << "Bird is flying.\n";
   }
};
```

5. Combine Concepts

- Here are scenarios combining all 3 pillars:
    - Game Example
        - **Base**: `Character` → **Derived**: `Knight`, `Archer`
        - **Encapsulated**: `health`, `stamina`
        - **Abstracted**: `attack()`, `move()`
    - Library System
        - **Base**: `LibraryItem` → **Derived**: `Book`, `DVD`, `Magazine`
        - **Encapsulated**: `title`, `id`, `availability`
        - **Abstracted**: `checkout()`, `returnItem()`
    - Smart Devices
        - **Base**: `SmartDevice` → **Derived**: `SmartSpeaker`, `SmartLight`
        - **Encapsulated**: `deviceID`, `status`
        - **Abstracted**: `turnOn()`, `turnOff()`