Introduction to OOP

By: **Annamalai A**

- OOP: Object Oriented Programming
- About creating objects that contain both data and methods
- Contains two elements:
    - **Data** — Field or Attributes
    - **Code** — Procedures or Methods

Advantages of OOP

- Faster and easier to execute
- Provides a clear structure to programs
- Makes it possible to create full reusable applications with less code and devlopment time

Classes

- User defined data types that acts as blueprints to objects
- Holds all data members and methods that are common to all instances of the same object
- Can be accessed by creating and using its instances

Objects

- The basic unit of OOP representing real life entities
- An instance of a class
- When a class is defined, no memory is occupied. Memory gets occupied when an object has been instanciated

- Has an **identity**, **state**, and **behaviour**
- Interact without the knowledge of other object's data or code

Access Specifiers

Specifies restrictions for data members and methods.

- `private` — Members and methods that can be accessed only within the class
- `public` — Members and methods that can be accessed anywhere the object is visible
- `protected` — Members and methods are accessible to only the main class and its derived classes

Data Hiding and Data Abstraction

Data Hiding	Data Abstraction
Security	Complexity
Restricts access to data members	Hides the underlying implementation logic

Data hiding is also known as **Encapsulation**

Encapsulation

- The process of wrapping up data members and methods into a single unit
- The data and code is only visible to the class it is defined in and not any other class
- The data members can only be accessed through member functions
- Also called **data hiding**

Constructors and Destructors

Constructors

A special function with the **same name as the class** and **no return type**.

1. **Default Constructor:** Initializes data with default values (or zeros).
2. **Parameterized Constructor:** Allows you to pass values at the time of object creation.
3. **Copy Constructor:** Creates an object by copying an existing object.
    - *Signature:* `ClassName(const ClassName &obj)`

Destructors

- Used to clean up memory (especially heap memory allocated by `new`).
- Name is preceded by a tilde (`~`).
- Called automatically when the object goes out of scope.

Best Practices

It is always recommended to encapsulate data members in order to prevent unauthorized access to data.

So in order to work with them, we need to use member functions. There are two kinds of these:

- **Getter Method** — Returns the value stored by an encapsulated data member
- **Setter Method** — Sets the value of an encapsulated data member if possible

Example Code

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

using namespace std; // iostream usage

// 1. CLASS DEFINITION & DESIGN (UML equivalent)
class Student {
private:
    // DATA HIDING: These cannot be accessed directly from ma
in()
    string name;
    int* grades;    // Pointer for dynamic memory
    int count;

public:
    // 2. DEFAULT CONSTRUCTOR
    Student() {
        name = "Unknown";
        count = 0;
        grades = nullptr;
        cout << "Default Constructor Called" << endl;
    }

    // 3. PARAMETERIZED CONSTRUCTOR
    Student(string n, int c) {
        name = n;
        count = c;
        // DATA MEMBERS & NEW OPERATOR (Dynamic Memory)
        grades = new int[count];
        cout << "Parameterized Constructor Called for " << na
me << endl;
    }

    // 4. COPY CONSTRUCTOR (Pass by reference to avoid infini
te recursion)
    Student(const Student &obj) {
        name = obj.name + "_copy";
        count = obj.count;
        // Deep Copy: Allocating new memory for the copy
        grades = new int[count];
        for(int i = 0; i < count; i++) {
            grades[i] = obj.grades[i];
        }
        cout << "Copy Constructor Called" << endl;
    }

    // 5. DESTRUCTOR
    ~Student() {
        // DELETE OPERATOR: Freeing heap memory to prevent le
aks
        delete[] grades;
        cout << "Destructor Called for " << name << endl;
    }

    // 6. BEST PRACTICES: SETTER (Mutator) with Validation
    void setName(string n) {
        if (!n.empty()) name = n;
    }

    // 7. BEST PRACTICES: GETTER (Accessor)
    string getName() {
        return name;
    }

    // 8. MEMBER FUNCTION (Defining behavior)
    void display() {
        cout << "Student: " << name << " | Grades Count: " <<
count << endl;
    }
};

int main() {
    // 9. OBJECT CREATION
    cout << "--- Step 1: Default ---" << endl;
    Student s1;

    cout << "\n--- Step 2: Parameterized ---" << endl;
    Student s2("Alice", 5);

    cout << "\n--- Step 3: Setters/Getters ---" << endl;
    s2.setName("Alice Cooper");
    cout << "Updated name: " << s2.getName() << endl;

    cout << "\n--- Step 4: Copy Constructor ---" << endl;
    Student s3 = s2; // Calls Copy Constructor
    s3.display();

    cout << "\n--- Step 5: End of Program (Automatic Destruct
ors) ---" << endl;
    return 0;
}`
```