قســــــم الامـــــــــــــــــن الـــــــــــــــــــســيبرانــــــــــــــــــي

DEPARTMENT OF CYBER SECURITY

SUBJECT:

OBJECT ORIENTED PROGRAMMING (OOP)

CLASS:

SECOND

LECTURER:

DR. ABDULKADHEM A. ABDULKADHEM

LECTURE: (3)

INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING (OOP)

1. General Introduction

Object-Oriented Programming, or OOP, is a programming paradigm that organizes software design around objects rather than actions. An object represents a real-world entity that combines both data, known as attributes, and behaviors, known as methods. For example, a car has attributes such as color and speed, and behaviors such as starting and driving. Similarly, a student has attributes such as name and age, and behaviors such as studying or attending a class. By modeling programs in this way, OOP makes code more natural to understand, closer to real life, and easier to maintain. Throughout this course, we will explore the four fundamental principles of OOP—abstraction, encapsulation, inheritance, and polymorphism—and learn how they work together to create software that is modular, reusable, and efficient.

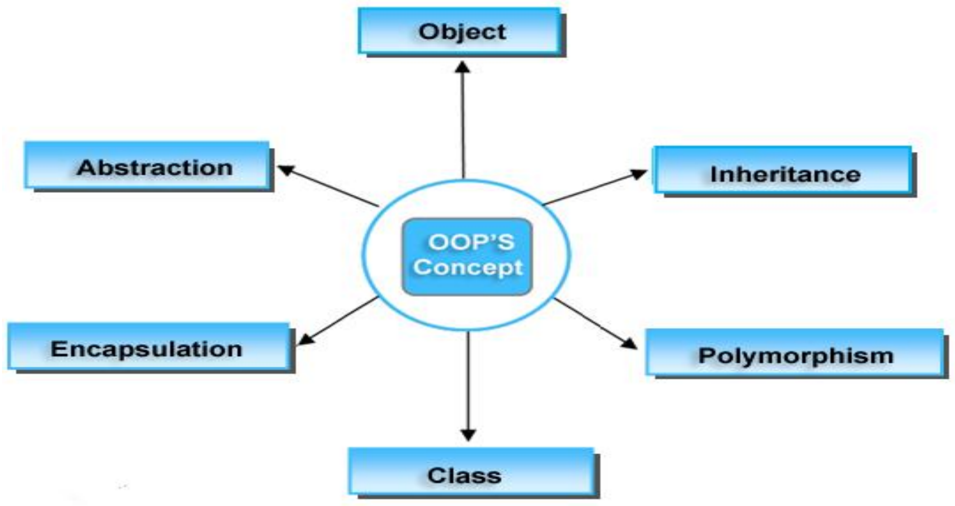
2. Object-Oriented Programming:

Object-Oriented Programming (OOP) is a programming paradigm that uses 'objects' to represent data and methods that operate on that data. This approach allows programmers to model real-world entities, making code more modular, reusable, and easier to maintain.

Core Concepts of OOP:

- **Object**: An object is an instance of a class that contains attributes (data) and methods (functions) that operate on the data. Think of it as a real-world entity, like a car or a person.
- **Class**: A class is a blueprint for creating objects. It defines the attributes and methods that its objects will have.

3. Key Concepts of OOP:



3.1. Abstraction:

Abstraction is the concept of hiding complex implementation details and exposing only the necessary parts. This allows users to interact with objects without worrying about their internal workings.

Example: In a car, you interact with the steering wheel without needing to know how the engine works.

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

class Car {
public:
    void startEngine() {
        cout << "Engine started!" << endl;
    }
    void drive() {
        cout << "Car is driving!" << endl;
    }
};
```

```
int main() {
   Car myCar;
   myCar.startEngine();
   myCar.drive();
   return 0;
}
```

Explanation:

- The Car class abstracts the complexity of starting and driving a car. The user doesn't need to know how the engine starts internally, just that they need to call the startEngine() and drive() methods.
- When we create an object myCar, we can use these functions to simulate starting and driving the car, without needing to understand the inner workings of the engine.

3.2. Encapsulation:

Encapsulation is the bundling (تجميع) of data (attributes) and methods (functions) that operate on the data into a single unit or class. It also **restricts** direct access to some components, which is essential for protecting data integrity.

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

class BankAccount {
private:
   int balance;

public:
   BankAccount(int initialBalance) {
      balance = initialBalance;
   }

   void deposit(int amount) {
      balance += amount;
```

```
    }

    void withdraw(int amount) {
        if (amount <= balance) {
            balance -= amount;
        } else {
            cout << "Insufficient funds!" << endl;
        }
    }

    int getBalance() {
        return balance;
    }
};

int main() {
    BankAccount account(500);
    account.deposit(200);
    account.withdraw(100);
    cout << "Current balance: " << account.getBalance() << endl;
    return 0;
}
```

Explanation:

- The BankAccount class encapsulates the balance attribute as a private variable. Users cannot modify it directly but can interact with it via public methods like deposit(), withdraw(), and getBalance().
- This protects the balance from being accidentally modified, ensuring that all changes occur through the proper methods.

3.3. Inheritance:

Inheritance allows one class to **inherit** (يرث) the attributes and methods of another class, promoting(ترويج) code reuse. The new class, known as the 'derived' or 'child' class, can also have its own additional attributes and methods.

```
#include <iostream>
using namespace std;
class Animal {
public:
   void eat() {
      cout << "This animal is eating!" << endl;
   }
};

class Dog : public Animal {
public:
   void bark() {
      cout << "The dog is barking!" << endl;
   }
};

int main() {
   Dog myDog;
   myDog.eat();  // Inherited method
   myDog.bark(); // Dog-specific method
   return 0;
}
```

Explanation:

- The Dog class inherits from the Animal class. This means that all Dog objects can use the eat() method defined in Animal.
- Inheritance allows us to reuse the functionality of the Animal class while adding more specific methods, like bark(), to the Dog class.

3.4. Polymorphism:

Polymorphism allows objects of different types to be treated as objects of a common superclass. It enables a single function or method to behave differently based on the object that calls it.

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

class Animal {
```

```
public:
    virtual void makeSound() {
        cout << "Some generic animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Woof!" << endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Meow!" << endl;
    }
};

int main() {
    Animal* myAnimal;
    Dog myDog;
    Cat myCat;

    myAnimal = &myDog;
    myAnimal->makeSound();  // Outputs: Woof!

    myAnimal = &myCat;
    myAnimal->makeSound();  // Outputs: Meow!

    return 0;
}
```

Explanation:

- Here, we use polymorphism to treat Dog and Cat objects as Animal objects. The method makeSound() behaves differently depending on whether it's called by a Dog or a Cat.

- Polymorphism enables flexibility and the ability to handle multiple object types through a common interface (in this case, Animal).

4. Conclusion:

Object-Oriented Programming makes software design more organized and intuitive by breaking down a problem into objects. Through abstraction, encapsulation, inheritance, and polymorphism, OOP encourages code reuse, modularity, and ease of maintenance.

Key Takeaways:

- **Abstraction** simplifies complex systems.
- **Encapsulation** protects data and ensures controlled access.
- **Inheritance** promotes code reuse.
- **Polymorphism** allows for flexibility in method behavior.

MCQ FOR LECTURE 3

Q1. Which of the following best defines a **Class** in OOP?
A) A real-world entity like a car or a person
B) An instance of an object
C) A blueprint for creating objects
D) A function that hides data
E) A variable that stores data

Q2. What is an **Object** in OOP?
A) A type of function
B) An instance of a class
C) A method inside a class
D) A way to protect data
E) A keyword in C++

Q3. Which OOP principle refers to **hiding internal details and showing only necessary features**?
A) Inheritance
B) Encapsulation
C) Abstraction
D) Polymorphism
E) Overloading

Q4. In the `BankAccount` example, why is `balance` declared as **private**?
A) To save memory
B) To restrict direct access and protect data integrity
C) To make the code faster
D) To allow all functions to access it directly
E) Because variables must be private in OOP

Q5. Which of the following demonstrates **Inheritance**?
A) A `Car` class with methods `start()` and `drive()`
B) A `Dog` class that extends an `Animal` class
C) A `BankAccount` class with `deposit()` and `withdraw()` methods
D) Using `private` to hide data
E) A function with the same name but different parameters

Q6. Which OOP concept allows the same function `makeSound()` to produce different outputs for `Dog` and `Cat`?
A) Abstraction
B) Encapsulation
C) Inheritance
D) Polymorphism
E) Aggregation

Q7. Which of the following is a key benefit of OOP?
A) Increases code duplication
B) Makes code less modular
C) Encourages code reuse and easier maintenance
D) Requires more memory than procedural programming
E) Prevents the use of functions