2. Handout

Object-Oriented Programming (OOP) - Student Handout

Introduction to Object-Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects and classes. Objects represent real-world entities, and classes are blueprints for creating these objects.

Key Concepts of OOP

1. Classes and Objects

- **Class**: A blueprint or template that defines the structure and behavior of objects.
- **Object**: An instance of a class.

Examples:

1. **Python Example:**

```python
class Car:
    def __init__(self, color, model):
        self.color = color
        self.model = model

    def start(self):
        print(f"The {self.color} {self.model} is starting.")

my_car = Car("Red", "Honda Civic")
my_car.start()  # Output: The Red Honda Civic is starting.
```

2. **Java Example:**

```java
class Car {
    String color;
    String model;

    Car(String color, String model) {
        this.color = color;
        this.model = model;
    }

    void start() {
        System.out.println("The " + color + " " + model + " is 
starting.");
    }
}

Car myCar = new Car("Red", "Honda Civic");
myCar.start();  // Output: The Red Honda Civic is starting.
```

3. **C++ Example:**

```
class Car {
public:
    string color;
    string model;

    Car(string c, string m) : color(c), model(m) {}

    void start() {
        cout << "The " << color << " " << model << " is starting." << 
endl;
    }
};

Car myCar("Red", "Honda Civic");
myCar.start();  // Output: The Red Honda Civic is starting.
```

2. Encapsulation

- **Encapsulation**: Bundling data and methods that operate on the data into a single unit, or class, and restricting access to some components.

Examples:

1. **Python Example:**

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def deposit(self, amount):
```

```python
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

2. **Java Example:**

```java
class BankAccount {
    private double balance;

    BankAccount(double balance) {
        this.balance = balance;
    }

    void deposit(double amount) {
        balance += amount;
    }

    double getBalance() {
        return balance;
    }
}

BankAccount account = new BankAccount(1000);
account.deposit(500);
System.out.println(account.getBalance());  // Output: 1500
```

3. **C++ Example:**

```cpp
class BankAccount {
private:
    double balance;

public:
    BankAccount(double b) : balance(b) {}

    void deposit(double amount) {
        balance += amount;
    }

    double getBalance() {
        return balance;
    }
};
```

```
BankAccount account(1000);
account.deposit(500);
cout << account.getBalance();  // Output: 1500
```

3. Inheritance

- **Inheritance**: Allows one class to inherit the properties and methods of another class.

Examples:

1. **Python Example:**

```
class Animal:
    def speak(self):
        print("Animal is making a sound")

class Dog(Animal):
    def speak(self):
        print("Dog is barking")

my_dog = Dog()
my_dog.speak()  # Output: Dog is barking
```

2. **Java Example:**

```
class Animal {
    void speak() {
        System.out.println("Animal is making a sound");
    }
}

class Dog extends Animal {
    void speak() {
        System.out.println("Dog is barking");
    }
}

Dog myDog = new Dog();
myDog.speak();  // Output: Dog is barking
```

3. **C++ Example:**

```
class Animal {
public:
    virtual void speak() {
        cout << "Animal is making a sound" << endl;
    }
};

class Dog : public Animal {
public:
    void speak() override {
        cout << "Dog is barking" << endl;
    }
};

Dog myDog;
myDog.speak();  // Output: Dog is barking
```

4. Polymorphism

- **Polymorphism**: The ability of different objects to respond to the same method in different ways.

Examples:

1. **Python Example:**

```
class Bird:
    def fly(self):
        print("Bird is flying")

class Airplane:
    def fly(self):
        print("Airplane is flying")

def make_it_fly(flying_object):
    flying_object.fly()

sparrow = Bird()
boeing = Airplane()

make_it_fly(sparrow)  # Output: Bird is flying
make_it_fly(boeing)   # Output: Airplane is flying
```

2. **Java Example:**

```
interface Flyable {
    void fly();
}

class Bird implements Flyable {
    public void fly() {
        System.out.println("Bird is flying");
    }
}

class Airplane implements Flyable {
    public void fly() {
        System.out.println("Airplane is flying");
    }
}

void makeItFly(Flyable flyingObject) {
    flyingObject.fly();
}

Bird sparrow = new Bird();
Airplane boeing = new Airplane();

makeItFly(sparrow);  // Output: Bird is flying
makeItFly(boeing);   // Output: Airplane is flying
```

3. **C++ Example:**

```
class Flyable {
public:
    virtual void fly() = 0;
};

class Bird : public Flyable {
public:
    void fly() override {
        cout << "Bird is flying" << endl;
    }
};

class Airplane : public Flyable {
public:
    void fly() override {
        cout << "Airplane is flying" << endl;
    }
};

void makeItFly(Flyable* flyingObject) {
    flyingObject->fly();
}

Bird sparrow;
Airplane boeing;

makeItFly(&sparrow);  // Output: Bird is flying
makeItFly(&boeing);   // Output: Airplane is flying
```

Conclusion

Object-Oriented Programming (OOP) is a powerful paradigm that helps in organizing code by modeling real-world entities as objects. The four main pillars of OOP—**Classes and Objects**, **Encapsulation**, **Inheritance**, and **Polymorphism**—enable you to write organized, reusable, and flexible code. Understanding these concepts will help you think about your programs in terms of objects and their interactions, making your code more intuitive and easier to manage.