In Partial Fulfillment of the Requirements for the

CS 223 - Object-Oriented Programming

**“FOUR PRINCIPLES OF OBJECT–ORIENTED PROGRAMMING”**

**Presented to:** Dr. Unife O. Cagas Professor V

**Presented by:**

Leizl Capistrano

BSCS 2A2 Student

**"BOOKS"**

Project Title

**Project Description**

In the world of programming, abstraction allows us to encapsulate shared characteristics while enabling unique behavior among different classes. This code achieved by marking the attributes \_title, \_author, and \_genre as protected, denoted by the underscore prefix. This means they are intended to be accessed only within the class or its subclasses. Getter methods are provided to access these attributes, encapsulating their access. It also demonstrated through method overriding. Each subclass provides its own implementation of the display\_info() method, which is defined as abstract in the base class Book. Despite the method having the same name in each subclass, they behave differently based on the specific type of book, showcasing polymorphic behavior. Instances of different book types (e.g., ClassicBook, FantasyBook) are created and stored in a list named books. Then, polymorphism is utilized to call the display\_info() method on each book object, which displays information specific to the genre of that book.

**Objectives:**

* To ensure controlled access to attributes and maintain data integrity.
* To promote code reuse and a clear hierarchy.
* To enable flexible and dynamic behavior. By implementing an abstract method display\_info in the base class and overriding it in each subclass, the code demonstrates polymorphism.
* To organize code logically and improve readability. By structuring the code with a base class and specialized subclasses, the snippet reflects a clear organization that helps maintain a clean and understandable codebase.
* To facilitate the addition of new book genres or extensions without extensive code changes.
* To show how Object-Oriented Programming concepts can be applied to a real-world scenario.

**Importance and Contribution of the Project**

This code used to create a flexible and organized structure for representing different types of objects in this case, books , with encapsulation providing controlled access to attributes, inheritance enabling reuse and organization, and polymorphism allowing dynamic behavior through method overriding.

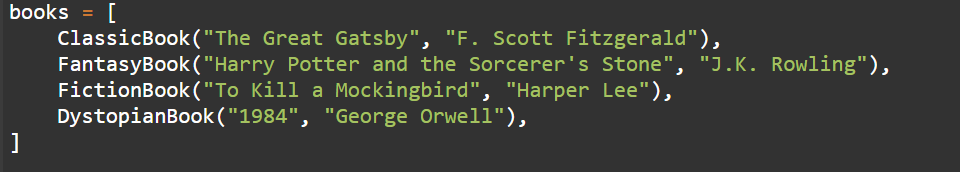
**Four Principles of Object-Oriented Programming with code**

**Class:**

The Book class is the base or parent class for all book-related objects. It encapsulates common attributes such as title, author, and genre. It also defines getter methods for encapsulation and an abstract method display\_info that subclasses must implement to provide their own version of book information.

**Object:**

Instantiated from these classes, with their specific attributes and behavior determined by the class they are created from.



**Inheritance:**

To promote code reuse and a clear hierarchy. The Book class serves as a base class with common attributes and methods that different genres of books can share. The subclasses (ClassicBook, FantasyBook, FictionBook, and DystopianBook) inherit from the Book class, allowing them to share common behavior while having specific unique traits.

**Encapsulation:**

the code demonstrates polymorphism by implementing the **display\_info()** method differently in each subclass (**ClassicBook**, **FantasyBook**, **FictionBook**, **DystopianBook**). Despite the method having the same name in each subclass, they behave differently based on the specific type of book, showcasing polymorphic behavior

**Polymorphism:**

To enable flexible and dynamic behavior. By implementing an abstract method display\_info in the base class and overriding it in each subclass, the code demonstrates polymorphism. This allows a common interface (display\_info) to behave differently based on the specific subclass, providing unique output for each book type.

**Abstraction**

The display\_info method in the Book base class is defined as an abstract method, raising a NotImplementedError if not implemented in a subclass. This enforces that subclasses must define their specific implementation, ensuring that each book type has its own way of displaying information.

**Inheritance:**

To promote code reuse and a clear hierarchy. The Book class serves as a base class with common attributes and methods that different genres of books can share. The subclasses (ClassicBook, FantasyBook, FictionBook, and DystopianBook) inherit from the Book class, allowing them to share common behavior while having specific unique traits.

**Encapsulation:**

the code demonstrates polymorphism by implementing the **display\_info()** method differently in each subclass (**ClassicBook**, **FantasyBook**, **FictionBook**, **DystopianBook**). Despite the method having the same name in each subclass, they behave differently based on the specific type of book, showcasing polymorphic behavior

**Polymorphism:**

To enable flexible and dynamic behavior. By implementing an abstract method display\_info in the base class and overriding it in each subclass, the code demonstrates polymorphism. This allows a common interface (display\_info) to behave differently based on the specific subclass, providing unique output for each book type.

**Abstraction**

The display\_info method in the Book base class is defined as an abstract method, raising a NotImplementedError if not implemented in a subclass. This enforces that subclasses must define their specific implementation, ensuring that each book type has its own way of displaying information.

**Hardware and Software Used**

**Hardware:**

• Laptop

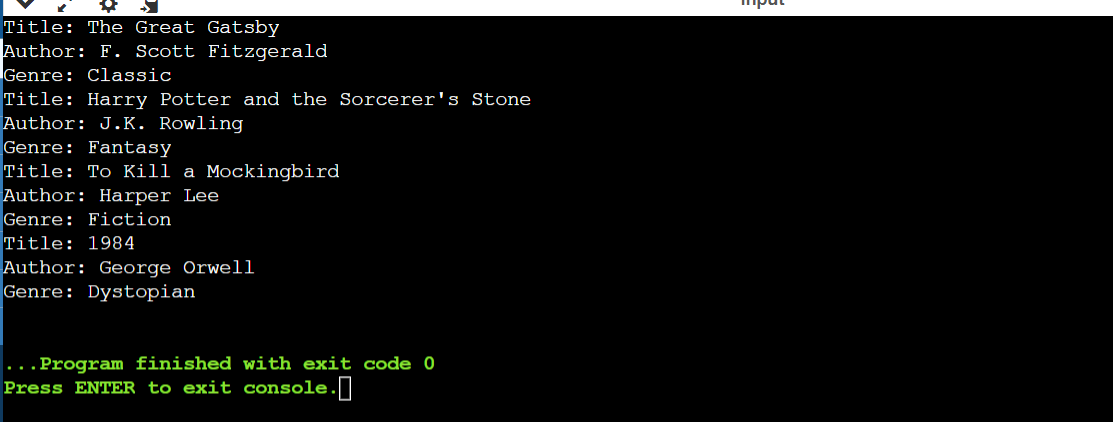
• Cellphone

**Software:**

• Visual Studio Code

• Online GDB

**Output:**

****

this code snippet demonstrates how encapsulation protects object data, inheritance enables code reuse and structure, and polymorphism allows for flexible behavior across different subclasses. The use of these OOP principles provides a structured approach to modeling different book genres with a consistent interface, while allowing for specific behaviors based on genre. This structure makes it easier to extend the system with new book genres or behaviors without extensive code changes.

**Code Documentation:**

# Base class for all books with encapsulation

class Book:

def \_\_init\_\_(self, title, author, genre):

self.\_title = title # Protected attribute

self.\_author = author # Protected attribute

self.\_genre = genre # Protected attribute

# Getters to encapsulate access to attributes

def get\_title(self):

return self.\_title

def get\_author(self):

return self.\_author

def get\_genre(self):

return self.\_genre

# Abstract method to be implemented by subclasses

def display\_info(self):

raise NotImplementedError("Subclasses must implement this method")

# Subclasses representing specific genres with polymorphism

class ClassicBook(Book):

def \_\_init\_\_(self, title, author):

super().\_\_init\_\_(title, author, "Classic")

# Implementing the abstract method (polymorphism)

def display\_info(self):

return f"Title: {self.get\_title()}\nAuthor: {self.get\_author()}\nGenre: {self.get\_genre()}"

class FantasyBook(Book):

def \_\_init\_\_(self, title, author):

super().\_\_init\_\_(title, author, "Fantasy")

def display\_info(self):

return f"Title: {self.get\_title()}\nAuthor: {self.get\_author()}\nGenre: {self.get\_genre()}"

class FictionBook(Book):

def \_\_init\_\_(self, title, author):

super().\_\_init\_\_(title, author, "Fiction")

def display\_info(self):

return f"Title: {self.get\_title()}\nAuthor: {self.get\_author()}\nGenre: {self.get\_genre()}"

class DystopianBook(Book):

def \_\_init\_\_(self, title, author):

super().\_\_init\_\_(title, author, "Dystopian")

def display\_info(self):

return f"Title: {self.get\_title()}\nAuthor: {self.get\_author()}\nGenre: {self.get\_genre()}"

# Create instances of specific book types (with encapsulation and polymorphism)

books = [

ClassicBook("The Great Gatsby", "F. Scott Fitzgerald"),

FantasyBook("Harry Potter and the Sorcerer's Stone", "J.K. Rowling"),

FictionBook("To Kill a Mockingbird", "Harper Lee"),

DystopianBook("1984", "George Orwell"),

]

# Use polymorphism to display information about each book

for book in books:

print(book.display\_info())

User Guide:

1. Creating Book Instances:

You can create instances of specific book types by instantiating the appropriate subclass (e.g., ClassicBook, FantasyBook) and providing the title and author as arguments.

2.Accessing Book Information:

Once a book instance is created, you can access its information (title, author, genre) using the getter methods (get\_title(), get\_author(), get\_genre()).

3.Displaying Book Information:

The display\_info() method can be used to display information about each book. Since it's an abstract method in the base class, it must be implemented by each subclass, providing a customized display for each genre.

4.Polymorphism:

Polymorphism is demonstrated by the fact that different subclasses (ClassicBook, FantasyBook, etc.) have their own implementations of the display\_info() method, allowing for different behavior while using a common interface.

5.terating Through Books:

The provided list book contains instances of different book types. You can iterate through this list and call display\_info() for each book to see its details. This utilizes polymorphism to display the appropriate information based on the genre of each book.

**References:**

"Python Tutorial" - W3Schools. [Online]. Available:

<https://www.w3schools.com/python/default.asp>

"Python Programming Language" - GeeksforGeeks. [Online]. Available:

<https://www.geeksforgeeks.org/python-programming-language/#python-oops-concepts>

"ChatGPT - Conversational AI Chatbots for Developers" - ChatGPT. [Online]. Available:

<https://chatgpt.com/c/ae2acb6c-efef-4e31-b0b6-20aeeb454f30>