**Daen Antule CSE 210 - Programming with Classes**

Programming with Classes – Articulate

# **Fundamentals of the 4 Principles of Programming with Classes:**

## ABSTRACTION

Abstraction is the process of simplifying something by ignoring unnecessary details. It allows us to focus on the essential aspects of a program while disregarding the intricate inner workings. Similar to putting together a puzzle by starting with the big pieces, abstraction helps in making code more modular, easier to understand, and modify. It improves performance by allowing us to work at a higher level of abstraction, without getting overwhelmed by the underlying complexity.

## ENCAPSULATION

Encapsulation involves hiding the internal details and implementation of an object from the outside world. It's like having a locked box, where only certain parts of the program have access to the internal state and operations of an object. Encapsulation promotes a clear separation of concerns and protects the integrity of an object's data. By encapsulating data and methods within a class, we can modify the internal implementation without affecting other parts of the program. It enhances code maintainability and helps in identifying and resolving errors efficiently.

## INHERITANCE

Inheritance allows one class to inherit the attributes and behaviors of another class, establishing a parent-child relationship. It resembles building with Legos, where new objects can be created using existing pieces. In object-oriented programming, a derived class can inherit properties and methods from a base class. This promotes code reusability, as the derived class can leverage the functionality already defined in the base class, without the need to rewrite it. Inheritance facilitates the organization and classification of classes, enabling the creation of specialized classes that inherit and extend the characteristics of a more general class.

## POLYMORPHISM

Polymorphism refers to the ability to treat different objects as if they belong to the same type, even though they may have distinct attributes and behaviors. It allows us to write code that can interact with various objects interchangeably, based on their shared characteristics. Polymorphism is achieved through inheritance, where derived classes can be treated as instances of their parent class. This simplifies code maintenance and accelerates the process of adding new features or modifying existing ones. By treating diverse objects uniformly, polymorphism enhances code flexibility and adaptability.

# **Uses and Program Flexibility of the 4 Principles of Programming with Classes:**

## Program 1: Abstraction with YouTube Videos

Abstraction is being used in the program to hide the internal implementation details of the `Video` and `Comment` classes. The properties and methods of these classes provide an abstraction of the underlying data and operations. The `Video` class encapsulates the information related to a video, such as its title, author, length, and comments. The `Comment` class encapsulates the information related to a comment, such as the commenter's name and the comment text.

By using abstraction, the program achieves flexibility for changes because it separates the high-level concepts (videos and comments) from the low-level implementation details. The program can be easily modified or extended without affecting the main logic. For example, if the structure or properties of the `Video` or `Comment` class need to be changed, such as adding a new property or modifying the existing ones, the program can be updated in those specific classes without impacting other parts of the program. Or for instance, if new features need to be added, such as sorting the videos based on their length or filtering comments based on specific criteria, the necessary changes can be made within the relevant classes (`Video` or `Comment`) without requiring significant modifications in other parts of the program.

## Program 2: Encapsulation with Online Ordering

Encapsulation is being used to achieve data hiding and to control the access to the internal state of objects. The classes `Order`, `Product`, `Customer`, and `Address` encapsulate their data and provide public methods to interact with that data. Here's how encapsulation is being used in the program:

* The `Order` class encapsulates a list of `Product` objects and a `Customer` object. The internal state of the `Order` object is hidden from the outside world, and access to it is provided through public methods such as `AddProduct`, `GetTotalPrice`, `GetPackingLabel`, and `GetShippingLabel`.
* The `Product` class encapsulates the details of a product, including its name, ID, price, and quantity. The internal state of a `Product` object is hidden, and access to it is provided through public methods such as `GetPrice`, `GetName`, `GetProductId`, and `GetQuantity`.
* The `Customer` class encapsulates customer data, including the ID, name, and address. Access to the customer's data is controlled through properties (`Id`, `Name`, `Address`) that provide read and write access. Additionally, the class provides public methods such as `IsInUSA` and `GetShippingCost` to interact with the customer's data.
* The `Address` class encapsulates the details of a customer's address, including street address, city, state/province, and country. The internal state is hidden, and access to it is provided through public methods such as `IsInUSA` and the overridden `ToString` method.

Encapsulation makes the program more flexible for changes by providing a clear separation between the internal implementation details and the public interface of the classes. If any changes are required in the internal implementation of a class (e.g., modifying the data representation, adding validation logic), the external code that uses the class doesn't need to be modified as long as the public interface remains unchanged. This allows for easier maintenance and modification of the program without affecting the overall functionality. Encapsulation also helps in preventing unauthorized access to the internal state of objects and promotes better code organization and modularity.

## Program 3: Inheritance with Event Planning

Inheritance is used in the program to create a hierarchy of classes, where derived classes inherit properties and behaviors from a base class. In this program, we have a base class called `Event`, and three derived classes called `LectureEvent`, `ReceptionEvent`, and `OutdoorGatheringEvent`.

The base class `Event` defines common properties and behaviors for all types of events, such as event title, description, date, time, and address. It also provides methods to get the standard details, full details, short description, and event title. The derived classes inherit these properties and behaviors from the base class.

The derived classes (`LectureEvent`, `ReceptionEvent`, and `OutdoorGatheringEvent`) extend the base class by adding their specific properties and behaviors. For example, the `LectureEvent` class adds properties for the speaker and capacity, the `ReceptionEvent` class adds a property for RSVP email, and the `OutdoorGatheringEvent` class adds a property for weather forecast. Each derived class overrides the `GetFullDetails` method to include its specific details along with the base class details.

The use of inheritance makes the program more flexible for changes because it allows us to add new types of events easily without modifying the existing code significantly. For example, if we want to add a new type of event called "WorkshopEvent," we can simply create a new derived class from the `Event` base class and add the specific properties and behaviors for workshops. This way, we can reuse the existing code for handling events, such as printing event details, without having to modify it.

## Program 4: Polymorphism with Exercise Tracking

Polymorphism is being used in the provided program through method overriding and inheritance. The base class `Activity` defines virtual methods `GetDistance()`, `GetSpeed()`, `GetPace()`, and `GetSummary()`. These methods are overridden in the derived classes `Running`, `Cycling`, and `Swimming` to provide specific implementations based on the type of activity. This is an example of runtime polymorphism, where the appropriate method implementation is determined dynamically at runtime based on the actual type of the object.

The use of polymorphism in this program makes it more flexible for future changes because it allows for easy addition of new types of activities without modifying the existing code. For example, if a new type of activity such as "Walking" needs to be added, a new derived class can be created from the base class `Activity` and the necessary methods can be overridden to provide the specific behavior for walking. This can be done without modifying the existing code in the `Program` class or the base `Activity` class. The program can then seamlessly handle the new type of activity without requiring any changes to the main logic.

The program uses a dictionary to store exercise data for different users. The values in the dictionary are lists of `Activity` objects. Since the derived classes (`Running`, `Cycling`, and `Swimming`) inherit from the base class `Activity`, they can be stored in the same list and accessed through the base class reference. This allows for polymorphic behavior when retrieving and manipulating the exercise data. For example, the `GetSummary()` method is called on each `Activity` object in the list, and the appropriate implementation based on the actual type of the object is executed. This flexibility enables the program to handle various types of activities within the same data structure, making it easier to add, modify, or extend the functionality in the future.