1. **Classes & Objects**
2. **Interfaces**
3. **Inheritance**
4. **Polymorphism**
5. **Abstraction**
6. **Encapsulation**
7. **Aggregation**

**Aggregation in Python**

**Introduction**

Aggregation is a key concept in object-oriented programming (OOP) that represents a "has-a" relationship between two classes, but with a crucial distinction: the lifecycle of the contained object is independent of the container object. This means that while one class contains another, the contained object can exist independently of the container.

Aggregation allows for better modularity, code reuse, and maintainability. It is different from composition, where the contained object cannot exist without the container.

**What is Aggregation?**

Aggregation is a form of association in OOP where an object of one class contains a reference to an object of another class. However, the contained object can exist independently of the container. This means that even if the container object is destroyed, the contained object can still be used elsewhere in the application.

**Key Characteristics of Aggregation:**

* Represents a **has-a** relationship.
* The contained object **can exist independently** of the container.
* Implemented using references to objects.
* Promotes **loose coupling** between objects.

**Example: A University and its Professors**

Consider a scenario where a University contains multiple Professor objects. However, a Professor can exist independently of any university. This is an example of aggregation.

class Professor:

def \_\_init\_\_(self, name, subject):

self.name = name

self.subject = subject

def teach(self):

print(f"{self.name} is teaching {self.subject}")

class University:

def \_\_init\_\_(self, university\_name):

self.university\_name = university\_name

self.professors = []

def add\_professor(self, professor):

self.professors.append(professor)

def show\_professors(self):

print(f"Professors at {self.university\_name}:")

for professor in self.professors:

print(f" - {professor.name}")

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

prof1 = Professor("Dr. Smith", "Computer Science")

prof2 = Professor("Dr. Johnson", "Mathematics")

university = University("Harvard University")

university.add\_professor(prof1)

university.add\_professor(prof2)

university.show\_professors()

# Professors can exist independently

prof1.teach()

prof2.teach()

**Output:**

Professors at Harvard University:

- Dr. Smith

- Dr. Johnson

Dr. Smith is teaching Computer Science

Dr. Johnson is teaching Mathematics

**Aggregation vs Composition**

| **Feature** | **Aggregation** | **Composition** |
| --- | --- | --- |
| Relationship | "Has-a" | "Has-a" |
| Ownership | Contained object **can exist independently** | Contained object **cannot exist without** the container |
| Lifetime | Contained object **outlives** the container | Contained object **is destroyed** with the container |
| Example | University and Professors | Car and Engine |

**Why Use Aggregation?**

**1. Promotes Code Reusability**

* Aggregated objects can be used in multiple places without being tightly coupled to a single container class.

**2. Encourages Loose Coupling**

* Aggregation allows objects to interact without being dependent on the lifecycle of each other.

**3. Better Maintainability**

* Changes in one class do not heavily impact the other, making the codebase easier to modify and extend.

**4. Real-World Applicability**

* Many real-world relationships, such as a school and its teachers, a company and its employees, naturally fit the aggregation model.

**Aggregation with Interfaces (Abstract Base Classes)**

Using abstract base classes, we can further enhance the flexibility of aggregation.

from abc import ABC, abstractmethod

class Teachable(ABC):

@abstractmethod

def teach(self):

pass

class Professor(Teachable):

def \_\_init\_\_(self, name, subject):

self.name = name

self.subject = subject

def teach(self):

print(f"{self.name} is teaching {self.subject}")

class University:

def \_\_init\_\_(self, university\_name):

self.university\_name = university\_name

self.professors = []

def add\_professor(self, professor):

self.professors.append(professor)

def show\_professors(self):

print(f"Professors at {self.university\_name}:")

for professor in self.professors:

professor.teach()

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

prof1 = Professor("Dr. Adams", "Physics")

prof2 = Professor("Dr. Lee", "Chemistry")

university = University("MIT")

university.add\_professor(prof1)

university.add\_professor(prof2)

university.show\_professors()

**Output:**

Professors at MIT:

Dr. Adams is teaching Physics

Dr. Lee is teaching Chemistry

**When to Use Aggregation?**

* When an object **can exist independently** from the container.
* When designing **loosely coupled** systems.
* When different objects need to be **shared** across multiple containers.
* When following **SOLID principles**, particularly the **Dependency Inversion Principle (DIP)**.

1. **Composition**

**Introduction**

Composition is one of the fundamental principles of object-oriented programming (OOP). It allows objects to be built using other objects, promoting code reuse, flexibility, and better maintainability. Unlike inheritance, which establishes an "is-a" relationship, composition represents a "has-a" relationship.

**What is Composition?**

Composition is a design principle in OOP where one class contains an instance (or instances) of another class as a field. The contained class is often called a component, and the containing class is referred to as a composite class. This helps in building complex systems by combining simpler objects.

**Example: A Car and its Components**

Consider a Car that consists of multiple components like an Engine, Wheel, and Transmission. Instead of inheriting from these components, a Car object will contain them as fields.

class Engine:

def \_\_init\_\_(self, horsepower):

self.horsepower = horsepower

def start(self):

print(f"Engine started with {self.horsepower} HP.")

class Wheel:

def \_\_init\_\_(self, type):

self.type = type

def rotate(self):

print(f"The {self.type} wheel is rotating.")

class Transmission:

def \_\_init\_\_(self, type):

self.type = type

def shift\_gear(self):

print(f"Transmission shifted: {self.type}")

class Car:

def \_\_init\_\_(self, engine, wheel, transmission):

self.engine = engine

self.wheel = wheel

self.transmission = transmission

def drive(self):

self.engine.start()

self.wheel.rotate()

self.transmission.shift\_gear()

print("Car is moving!")

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

engine = Engine(150)

wheel = Wheel("Alloy")

transmission = Transmission("Automatic")

car = Car(engine, wheel, transmission)

car.drive()

**Output:**

Engine started with 150 HP.

The Alloy wheel is rotating.

Transmission shifted: Automatic

Car is moving!

**Why Prefer Composition Over Inheritance?**

**1. Encapsulation and Flexibility**

* Composition allows us to change the behavior of an object dynamically by replacing components at runtime.
* Inheritance makes it difficult to modify an existing class hierarchy without breaking existing code.

**2. Better Code Reusability**

* Composition promotes reusable components. The Engine, Wheel, and Transmission classes can be used in multiple types of vehicles (Car, Bike, Truck) without modification.

**3. Avoids Inheritance Pitfalls**

* Inheritance can lead to deep class hierarchies, making maintenance difficult.
* It enforces strict parent-child relationships, which can be too rigid for some designs.

**4. Supports Interface-Based Design**

* Composition can be combined with interfaces (or abstract classes) to achieve powerful decoupling.

**Composition with Abstract Classes**

Using abstract base classes with composition allows for greater flexibility and loose coupling.

from abc import ABC, abstractmethod

class Engine(ABC):

@abstractmethod

def start(self):

pass

class PetrolEngine(Engine):

def start(self):

print("Petrol Engine started.")

class DieselEngine(Engine):

def start(self):

print("Diesel Engine started.")

class Car:

def \_\_init\_\_(self, engine: Engine):

self.engine = engine

def start\_car(self):

self.engine.start()

print("Car is ready to go!")

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

petrol\_car = Car(PetrolEngine())

petrol\_car.start\_car()

diesel\_car = Car(DieselEngine())

diesel\_car.start\_car()

**Output:**

Petrol Engine started.

Car is ready to go!

Diesel Engine started.

Car is ready to go!

**When to Use Composition?**

* When building complex objects that consist of multiple components.
* When you want to achieve **code reusability** without rigid inheritance hierarchies.
* When different behaviors need to be swapped dynamically (e.g., using different types of engines in a vehicle).
* When following the **favor composition over inheritance** principle.

1. **Association**

**Introduction**

Association is a fundamental concept in object-oriented programming (OOP) that defines a relationship between two or more objects. It represents how objects interact with each other while maintaining their independence.

Association is **not inheritance**—rather, it is a relationship between objects that allows communication while ensuring they remain loosely coupled.

**What is Association?**

Association defines a connection between two classes, where one class is linked to another. The association can be **one-to-one**, **one-to-many**, **many-to-one**, or **many-to-many**. Objects in an association can exist independently of each other.

**Key Characteristics of Association:**

* Represents a **uses-a** or **knows-a** relationship.
* Objects in an association **can exist independently**.
* Can be **unidirectional** or **bidirectional**.
* Promotes **modularity** and **code reusability**.

**Example: A Student and a Teacher**

A Student can be associated with multiple Teacher objects, and a Teacher can have multiple Student objects. This represents a **many-to-many** association.

class Teacher:

def \_\_init\_\_(self, name):

self.name = name

self.students = []

def add\_student(self, student):

self.students.append(student)

def show\_students(self):

print(f"{self.name} teaches:")

for student in self.students:

print(f" - {student.name}")

class Student:

def \_\_init\_\_(self, name):

self.name = name

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

teacher1 = Teacher("Mr. Smith")

teacher2 = Teacher("Mrs. Johnson")

student1 = Student("Alice")

student2 = Student("Bob")

teacher1.add\_student(student1)

teacher1.add\_student(student2)

teacher2.add\_student(student2)

teacher1.show\_students()

teacher2.show\_students()

**Output:**

Mr. Smith teaches:

- Alice

- Bob

Mrs. Johnson teaches:

- Bob

**Types of Association**

**1. One-to-One Association**

* Each object of class A is associated with one object of class B.
* Example: A Person has one Passport.

**2. One-to-Many Association**

* One object of class A can be associated with multiple objects of class B.
* Example: A Teacher teaches multiple Students.

**3. Many-to-One Association**

* Multiple objects of class A can be associated with one object of class B.
* Example: Multiple Students belong to one School.

**4. Many-to-Many Association**

* Multiple objects of class A can be associated with multiple objects of class B.
* Example: Teachers and Students.

**Why Use Association?**

* **Promotes Code Reusability**: Objects can be reused across multiple associations without duplication.
* **Encourages Loose Coupling**: Objects interact without depending on the internal implementation of each other.
* **Improves Maintainability**: Changing one object does not heavily impact others, making code easier to manage.
* **Better System Design**: Allows modeling of real-world relationships between entities effectively.

**Association vs Aggregation vs Composition**

| **Feature** | **Association** | **Aggregation** | **Composition** |
| --- | --- | --- | --- |
| Relationship | "Knows-a" | "Has-a" | "Has-a" |
| Object Independence | Objects are independent | Contained object **can exist independently** | Contained object **cannot exist without** the container |
| Lifetime | Objects exist separately | Contained object **outlives** the container | Contained object **is destroyed** with the container |
| Example | Teacher and Student | University and Professors | Car and Engine |

**Bidirectional Association**

Associations can be **unidirectional** (one object knows about another) or **bidirectional** (both objects know about each other).

**Example: A Library and Books (Bidirectional Association)**

class Library:

def \_\_init\_\_(self, name):

self.name = name

self.books = []

def add\_book(self, book):

self.books.append(book)

def show\_books(self):

print(f"Books in {self.name}:")

for book in self.books:

print(f" - {book.title}")

class Book:

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

self.title = title

self.library = library

def show\_library(self):

print(f"{self.title} is in {self.library.name}")

# Example Usage

if \_\_name\_\_ == "\_\_main\_\_":

library = Library("City Library")

book1 = Book("1984", library)

book2 = Book("Brave New World", library)

library.add\_book(book1)

library.add\_book(book2)

library.show\_books()

book1.show\_library()

book2.show\_library()

**Output:**

Books in City Library:

- 1984

- Brave New World

1984 is in City Library

Brave New World is in City Library