



Python Programming Bootcamp

Week 5: Object-Oriented Programming Fundamentals

Hilal Khan

Center for Artificial Intelligence & Emerging Technologies

By the end of this week, you will be able to:

- Understand the fundamentals of Object-Oriented Programming
- Create and use classes and objects in Python
- Differentiate between instance and class attributes/methods
- Apply the `self` keyword correctly
- Use static methods appropriately
- Model relationships between objects (Association)
- Read and create basic UML class diagrams
- Understand the four properties of OOP
- Build real-world applications using OOP principles

What We'll Cover Today:

- Introduction to Object-Oriented Programming
- Classes and Objects
- Understanding the `self` keyword
- Instance Attributes and Methods
- Class Attributes and Methods
- Static Methods
- Association: Modeling Relationships
- UML Class Diagrams
- Overview of OOP Properties
- Practical Examples and Best Practices

Quick Recap: Previous Weeks

Week 1:

- Python basics, variables, data types, operators
- Input/output, conditionals, loops

Week 2:

- Data structures: Lists, strings, tuples, dictionaries, sets

Week 3:

- Functions, parameters, return values
- Modules and imports

Week 4:

- Error handling with try-except
- File I/O operations
- Working with CSV and JSON files

Introduction to OOP

Why Learn OOP?

Real-World Programming Needs

- Organize and structure complex code
- Model real-world entities and relationships
- Enable code reusability and maintainability
- Facilitate team collaboration
- Industry standard for large-scale applications
- Create modular and scalable systems

Key Benefit

OOP helps you think about programs as collections of interacting objects

rather than just sequences of instructions

What is Object-Oriented Programming?

Core Concept

- **Programming paradigm** based on the concept of "objects"
- Objects contain both **data** and **behavior**
- Models real-world entities and their interactions
- Focuses on what objects are and what they can do

Key Terminology

- **Class**: Blueprint or template for creating objects
- **Object**: Specific instance of a class
- **Attribute**: Data/variables that belong to an object
- **Method**: Functions that belong to an object

Procedural vs Object-Oriented Programming

Procedural Approach (What We've Been Doing)

- Functions operate on data
- Data and functions are separate
- Focus on "what steps to perform"

Object-Oriented Approach (What We'll Learn)

- Objects combine data and functions
- Data and behavior bundled together
- Focus on "what objects exist and how they interact"

Example

Procedural: `calculate_area(width, height)`

OOP: `rectangle.calculate_area()`

Four Fundamental Properties of OOP

- **Encapsulation**

- Bundling data and methods together
- Controlling access to internal data
- Hiding implementation details

- **Inheritance**

- Creating new classes from existing ones
- Code reuse and hierarchy creation
- "is-a" relationship between classes

Four Fundamental Properties of OOP (continued)

- **Polymorphism**

- Same interface, different implementations
- Methods with same name but different behavior
- Flexibility in method execution

- **Abstraction**

- Hiding complex implementation details
- Showing only essential features
- Simplifying complex reality

Note

Week 6 will explore each property in detail with advanced examples

Classes and Objects

Classes: Blueprints for Objects i

What is a Class?

- A **template** or **blueprint** for creating objects
- Defines the structure and behavior of objects
- Contains attributes (data) and methods (functions)
- Created using the `class` keyword

Basic class syntax:

```
# Simplest class definition
class Car:
    pass

# Class with attributes and methods
class Student:
```

Classes: Blueprints for Objects ii

```
def __init__(self, name, age):  
    self.name = name  
    self.age = age  
  
def study(self):  
    return f"{self.name} is studying"
```

Real-world analogy:

- **Class = Blueprint:** A house blueprint
- **Object = Instance:** An actual house built from the blueprint
- Multiple houses can be built from the same blueprint
- Each house can have different colors, furniture, etc.

```
# Class is the blueprint
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        return "Woof!"
```

```
# Objects are actual instances
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Max", "German Shepherd")
```

What is an Object?

- A specific **instance** of a class
- Created from the class blueprint
- Has its own unique data (attribute values)
- Can perform actions defined by class methods

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        return "Woof!"
```

Objects: Instances of Classes ii

```
# Creating objects (instances)
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Max", "German Shepherd")

# Each object has its own data
print(dog1.name)  # "Buddy"
print(dog2.name)  # "Max"

# But same behavior
print(dog1.bark())  # "Woof!"
print(dog2.bark())  # "Woof!"
```

Creating and using objects:

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount
        return self.balance

    def get_info(self):
        return f"Account: {self.owner}, Balance: ${
self.balance}"

# Creating multiple objects
account1 = BankAccount("Alice", 1000)
```

Objects: Instances of Classes iv

```
account2 = BankAccount("Bob", 500)

# Each object maintains its own state
account1.deposit(200)  # Alice's account: 1200
account2.deposit(100)  # Bob's account: 600

print(account1.get_info())  # "Account: Alice, Balance
    : $1200"
print(account2.get_info())  # "Account: Bob, Balance:
    $600"
```

Understanding the self

Keyword

The self Keyword i

What is self?

- Reference to the **current instance** of the class
- First parameter of every instance method
- Used to access instance attributes and methods
- Python automatically passes it when calling methods

```
class Student:
    def __init__(self, name, age):
        # self refers to the current object being
        created
        self.name = name      # instance attribute
        self.age = age        # instance attribute

    def introduce(self):
```

The self Keyword ii

```
        # self refers to the current object calling
        this method
        return f"Hi, I'm {self.name} and I'm {self.age}
        } years old"
```

```
# Creating objects
```

```
student1 = Student("Alice", 20)
```

```
student2 = Student("Bob", 22)
```

```
# Python automatically passes the object as 'self'
```

```
print(student1.introduce()) # self = student1
```

```
print(student2.introduce()) # self = student2
```

The self Keyword iii

Why self is important:

Key Points About self

- **Access instance data:** Each object maintains its own state
- **Method binding:** Connects methods to specific objects
- **Object identity:** Distinguishes between different instances
- **Automatic passing:** Python handles it automatically

```
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder
        self.balance = balance

    def deposit(self, amount):
```

The self Keyword iv

```
        # self.balance refers to THIS account's
balance
        self.balance += amount
        return self.balance

def get_balance(self):
    # Returns THIS account's balance
    return self.balance

# Two different accounts
account1 = BankAccount("Alice", 1000)
account2 = BankAccount("Bob", 500)

account1.deposit(200)  # self = account1, balance
                       becomes 1200
```

The self Keyword v

```
account2.deposit(100)  # self = account2, balance
                        becomes 600

print(account1.get_balance())  # 1200
print(account2.get_balance())  # 600
```

Instance Attributes and Methods

What are Instance Attributes?

- **Unique to each object** (instance)
- Defined inside `__init__` method using `self`
- Each object has its own copy
- Store object-specific data

```
class Car:
    def __init__(self, brand, model, year):
        # Instance attributes - different for each car
        self.brand = brand
        self.model = model
        self.year = year
        self.mileage = 0 # Default value for all cars
```

Instance Attributes ii

```
def drive(self, distance):
    # Modifying instance attribute
    self.mileage += distance

# Different objects, different attribute values
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2023)

car1.drive(150)
car2.drive(75)

print(car1.mileage)    # 150 (car1's mileage)
print(car2.mileage)    # 75 (car2's mileage)
```

The `__init__` method:

Constructor Method

- Called automatically when creating a new object
- Used to initialize instance attributes
- First parameter is always `self`
- Can accept additional parameters

```
class Rectangle:
    def __init__(self, width, height):
        # Initialize instance attributes
        self.width = width
        self.height = height
```

```
def area(self):
    return self.width * self.height

def perimeter(self):
    return 2 * (self.width + self.height)

# __init__ is called automatically
rect1 = Rectangle(5, 10) # __init__(rect1, 5, 10)
rect2 = Rectangle(3, 7)  # __init__(rect2, 3, 7)

print(rect1.area())      # 50
print(rect2.area())      # 21
```

What are Instance Methods?

- Functions that **operate on instance data**
- First parameter is always `self`
- Can access and modify instance attributes
- Called on objects (instances)

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance

    # Instance method - operates on specific account
    def deposit(self, amount):
```

```
        self.balance += amount
        return f"New balance: ${self.balance}"

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            return f"Withdrew ${amount}. Balance: ${
self.balance}"
        return "Insufficient funds"

    def get_balance(self):
        return self.balance

# Using instance methods
account = BankAccount("Alice", 1000)
```

```
print(account.deposit(500))    # Operates on account's  
    balance  
print(account.withdraw(200))   # Operates on account's  
    balance
```

Class Attributes and Methods

What are Class Attributes?

- **Shared by all instances** of the class
- Defined directly in the class (outside methods)
- Same value for every object of that class
- Used for data common to all instances

```
class Car:
    # Class attribute - shared by all cars
    wheels = 4
    vehicle_type = "Automobile"

    def __init__(self, brand, model):
        # Instance attributes - unique to each car
        self.brand = brand
```

Class Attributes ii

```
        self.model = model

# Creating objects
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")

# Class attributes are the same for all instances
print(car1.wheels)          # 4
print(car2.wheels)          # 4
print(Car.wheels)           # 4 (accessed through class)

# Instance attributes are different
print(car1.brand)           # "Toyota"
print(car2.brand)           # "Honda"
```

When to use class attributes:

- Constants shared across all instances
- Counters (e.g., total number of objects created)
- Default values for all instances
- Configuration settings

```
class Employee:
    # Class attributes
    company_name = "Tech Corp"
    total_employees = 0
    min_salary = 30000

    def __init__(self, name, salary):
        self.name = name
```

```
        self.salary = salary
        # Increment class attribute
        Employee.total_employees += 1

    def get_info(self):
        return f"{self.name} works at {Employee.
company_name}"

emp1 = Employee("Alice", 50000)
emp2 = Employee("Bob", 60000)

print(Employee.total_employees)    # 2
print(emp1.get_info())             # "Alice works at
    Tech Corp"
```

What are Class Methods?

- Methods that **operate on the class itself**
- Use `@classmethod` decorator
- First parameter is `cls` (class, not instance)
- Can access and modify class attributes
- Called on the class or instances

```
class Student:
    # Class attribute
    school_name = "Python High School"
    total_students = 0

    def __init__(self, name):
```

Class Methods ii

```
        self.name = name
        Student.total_students += 1

# Class method - operates on class data
@classmethod
def get_school_info(cls):
    return f"School: {cls.school_name}, Total
Students: {cls.total_students}"

@classmethod
def change_school(cls, new_name):
    cls.school_name = new_name

# Using class methods
student1 = Student("Alice")
student2 = Student("Bob")
```

```
print(Student.get_school_info())  
# "School: Python High School, Total Students: 2"  
  
Student.change_school("Coding Academy")  
print(Student.get_school_info())  
# "School: Coding Academy, Total Students: 2"
```

Static Methods

What are Static Methods?

- Methods that **don't operate on instance or class**
- Use @staticmethod decorator
- No self or cls parameter
- Utility functions related to the class
- Can be called on class or instances

```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
```

Static Methods ii

```
def multiply(x, y):  
    return x * y  
  
@staticmethod  
def is_even(number):  
    return number % 2 == 0  
  
# Calling static methods  
print(MathOperations.add(5, 3))           # 8  
print(MathOperations.multiply(4, 7))      # 28  
print(MathOperations.is_even(10))         # True  
  
# Can also call on instance (but not common)  
math_obj = MathOperations()  
print(math_obj.add(2, 3))                 # 5
```

When to use static methods:

- Utility functions related to the class
- Functions that don't need instance or class data
- Helper functions for data validation
- Conversion or formatting functions

```
class DateValidator:
    @staticmethod
    def is_valid_year(year):
        return 1900 <= year <= 2100

    @staticmethod
    def is_leap_year(year):
```

```
        return (year % 4 == 0 and year % 100 != 0) or  
(year % 400 == 0)
```

```
@staticmethod
```

```
def days_in_month(month, year):
```

```
    if month in [1, 3, 5, 7, 8, 10, 12]:
```

```
        return 31
```

```
    elif month in [4, 6, 9, 11]:
```

```
        return 30
```

```
    else:
```

```
        return 29 if DateValidator.is_leap_year(  
year) else 28
```

```
# Use without creating instance
```

```
print(DateValidator.is_leap_year(2024))           # True
```

```
print(DateValidator.days_in_month(2, 2024))       # 29
```


Comparison: Instance vs Class vs Static Methods (Part 1)

Instance Methods

- First parameter: `self`
- Access: Instance and class attributes
- Use: Operate on specific object data
- Example: `account.deposit(100)`

Class Methods

- Decorator: `@classmethod`
- First parameter: `cls`
- Access: Class attributes (not instance)
- Use: Operate on class-level data
- Example: `Student.get_total_students()`

Comparison: Instance vs Class vs Static Methods (Part 2)

Static Methods

- Decorator: `@staticmethod`
- First parameter: None
- Access: No direct access to class or instance
- Use: Utility functions related to class
- Example: `MathOperations.add(5, 3)`

Quick Reference

- **Instance method:** Needs object data → use `self`
- **Class method:** Needs class data → use `@classmethod` and `cls`
- **Static method:** Needs neither → use `@staticmethod`

All Three Methods Together i

```
class Employee:
    company = "Tech Corp"
    total_employees = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.total_employees += 1

    # Instance method
    def give_raise(self, amount):
        self.salary += amount
        return f"{self.name}'s new salary: ${self.
salary}"

    # Class method
```

All Three Methods Together ii

```
@classmethod
def get_total_employees(cls):
    return f"{cls.company} has {cls.
total_employees} employees"

# Static method
@staticmethod
def is_valid_salary(salary):
    return salary >= 30000

# Usage
emp1 = Employee("Alice", 50000)
emp2 = Employee("Bob", 60000)

# Instance method - operates on emp1's data
```

All Three Methods Together iii

```
print(emp1.give_raise(5000))    # "Alice's new salary:
                                $55000"

# Class method - operates on class data
print(Employee.get_total_employees()) # "Tech Corp
    has 2 employees"

# Static method - utility function
print(Employee.is_valid_salary(25000)) # False
print(Employee.is_valid_salary(40000)) # True
```

Association: Modeling Relationships

What is Association?

- Relationship between two or more classes
- Objects of different classes working together
- Represents "has-a" or "uses-a" relationship
- One object uses or contains another object

Real-World Examples

- Student **has-a** Address
- Teacher **uses-a** Whiteboard
- Car **has-a** Engine
- Library **has-many** Books

Types of Association

Three Main Types

- **Simple Association:** Objects work together but are independent
 - Example: Teacher uses Whiteboard
 - Both exist independently
- **Aggregation:** "Has-a" relationship (weak ownership)
 - Example: Library has Books
 - Books can exist without Library
- **Composition:** "Part-of" relationship (strong ownership)
 - Example: Car has Engine
 - Engine cannot exist without Car

Note

We'll focus on Simple Association this week. Aggregation and Composition will be covered in Week 6.

Simple Association Example i

```
class Address:
    def __init__(self, street, city, zipcode):
        self.street = street
        self.city = city
        self.zipcode = zipcode

    def get_full_address(self):
        return f"{self.street}, {self.city} {self.zipcode}"

class Student:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address # Student HAS-A
                                Address
```

Simple Association Example ii

```
def get_info(self):
    return f"{self.name} lives at {self.address.get_full_address()}"

# Creating objects
addr1 = Address("123 Main St", "New York", "10001")
student1 = Student("Alice", 20, addr1)

print(student1.get_info())
# "Alice lives at 123 Main St, New York 10001"

# Address exists independently
print(addr1.get_full_address())
# "123 Main St, New York 10001"
```

Another example: Teacher and Course

```
class Course:
    def __init__(self, course_name, course_code):
        self.course_name = course_name
        self.course_code = course_code

    def get_info(self):
        return f"{self.course_code}: {self.course_name}"
    }"

class Teacher:
    def __init__(self, name, courses):
        self.name = name
        self.courses = courses    # Teacher HAS-A list
of Courses
```

Simple Association Example iv

```
def list_courses(self):
    result = f"{self.name} teaches:\n"
    for course in self.courses:
        result += f"    - {course.get_info()}\n"
    return result

# Creating objects
python_course = Course("Python Programming", "CS101")
oop_course = Course("OOP Fundamentals", "CS201")

teacher = Teacher("Dr. Smith", [python_course,
                                oop_course])
print(teacher.list_courses())
# Dr. Smith teaches:
#    - CS101: Python Programming
#    - CS201: OOP Fundamentals
```


One-to-Many Association i

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def __str__(self):
        return f"{self.title} by {self.author}"

class Library:
    def __init__(self, name):
        self.name = name
        self.books = [] # Library HAS-MANY Books

    def add_book(self, book):
        self.books.append(book)
        return f"Added: {book}"
```

One-to-Many Association ii

```
def list_books(self):
    if not self.books:
        return f"{self.name} has no books"
    result = f"{self.name} Books:\n"
    for i, book in enumerate(self.books, 1):
        result += f"{i}. {book}\n"
    return result

# Using association
library = Library("City Library")
library.add_book(Book("Python Crash Course", "Eric
    Matthes"))
library.add_book(Book("Clean Code", "Robert Martin"))

print(library.list_books())
```

```
# City Library Books:  
# 1. Python Crash Course by Eric Matthes  
# 2. Clean Code by Robert Martin
```

UML Class Diagrams

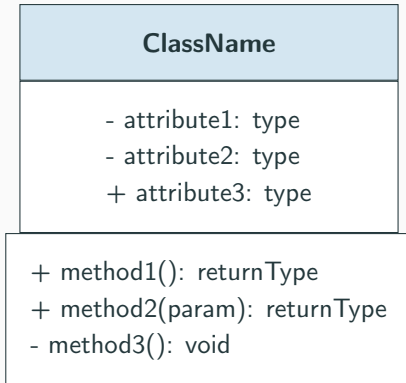
What is UML?

- **Unified Modeling Language**
- Visual representation of classes and relationships
- Industry standard for software design
- Helps plan before coding

UML Class Diagram Components

- **Class name** (top section)
- **Attributes** (middle section)
- **Methods** (bottom section)
- **Relationships** (arrows between classes)

UML Class Box Structure



Symbols

- + = Public (accessible from outside)
- - = Private (internal to class)
- # = Protected (for inheritance - Week 6)

Example: BankAccount Class

BankAccount

- owner: str
- balance: float

- + deposit(amount)
- + withdraw(amount)
- + get_balance()

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount

    def get_balance(self):
        return self.balance
```

UML: Showing Associations



Association Arrow

- Arrow shows relationship direction
- **has-a**: Student has an Address
- Student class contains Address object as attribute

Reading UML Diagrams

Steps to Read UML

1. Identify the class name
2. Look at attributes (data the class stores)
3. Look at methods (what the class can do)
4. Check relationships with other classes

Practice

Given this UML, what does the code look like?

Car
- brand: str - speed: int
+ accelerate() + brake()

Practical Examples

Practical Example 1: Library Management System i

```
class Book:
    total_books = 0    # Class attribute

    def __init__(self, title, author, isbn):
        self.title = title
        self.author = author
        self.isbn = isbn
        self.is_available = True
        Book.total_books += 1

    def borrow(self):
        if self.is_available:
            self.is_available = False
            return f"{self.title} borrowed successfully"
        return f"{self.title} is not available"
```

Practical Example 1: Library Management System ii

```
def return_book(self):
    self.is_available = True
    return f"{self.title} returned successfully"

@classmethod
def get_total_books(cls):
    return f"Total books in system: {cls.
total_books}"

@staticmethod
def validate_isbn(isbn):
    return len(isbn) == 13 and isbn.isdigit()
```

Practical Example 1: Library Management System iii

```
class Member:
    def __init__(self, name, member_id):
        self.name = name
        self.member_id = member_id
        self.borrowed_books = [] # Association with
Book

    def borrow_book(self, book):
        if book.is_available:
            result = book.borrow()
            self.borrowed_books.append(book)
            return result
        return f"{book.title} is not available"

    def return_book(self, book):
```

Practical Example 1: Library Management System iv

```
        if book in self.borrowed_books:
            result = book.return_book()
            self.borrowed_books.remove(book)
            return result
        return f"{book.title} was not borrowed by {
self.name}"

def list_borrowed_books(self):
    if not self.borrowed_books:
        return f"{self.name} has no borrowed books
"

    result = f"{self.name}'s borrowed books:\n"
    for book in self.borrowed_books:
        result += f"    - {book.title}\n"
    return result
```

Practical Example 1: Library Management System v

```
# Using the Library Management System
def main():
    # Create books
    book1 = Book("Python Crash Course", "Eric Matthes",
    "9781593279288")
    book2 = Book("Clean Code", "Robert Martin", "
    9780132350884")

    # Validate ISBN
    print(Book.validate_isbn("9781593279288")) # True

    # Create member
    member = Member("Alice", "M001")

    # Borrow books
```

Practical Example 1: Library Management System vi

```
print(member.borrow_book(book1))
# "Python Crash Course borrowed successfully"

print(member.borrow_book(book2))
# "Clean Code borrowed successfully"

# List borrowed books
print(member.list_borrowed_books())
# Alice's borrowed books:
#   - Python Crash Course
#   - Clean Code

# Return a book
print(member.return_book(book1))
# "Python Crash Course returned successfully"
```

Practical Example 1: Library Management System vii

```
# Check total books
print(Book.get_total_books()) # "Total books in
system: 2"

if __name__ == "__main__":
    main()
```

Best Practices

Naming Conventions

- **Class names:** PascalCase (BankAccount, StudentRecord)
- **Method names:** snake_case (calculate_salary, get_info)
- **Attribute names:** snake_case (account_balance, student_name)
- **Constants:** UPPER_CASE (MAX_SIZE, DEFAULT_VALUE)

Design Principles

- **Single Responsibility:** Each class should have one main purpose
- **Clear Naming:** Names should reveal intention
- **Consistent Structure:** Follow predictable patterns
- **Documentation:** Use docstrings to explain purpose

Common Mistakes to Avoid

Beginner Pitfalls

- **Forgetting self:** Methods without self parameter
- **Confusing class/instance attributes:** When to use each
- **Overusing class attributes:** When instance attributes are needed
- **Poor naming:** Unclear or misleading names
- **Too much in one class:** Violating single responsibility
- **Not using decorators:** Forgetting @classmethod or @staticmethod

Good Practice Examples

- `student.get_grade()` - Clear what it does
- `car.calculate_mileage()` - Specific purpose
- `L.s.process()` - Unclear purpose
- `L.obj.do_everything()` - Too broad responsibility

Practice Exercises

Exercise 1: Create a Rectangle Class

Build a complete Rectangle class:

Task

Create a Rectangle class with:

- Instance attributes: width, height
- Instance methods: area(), perimeter()
- Class attribute: shape_type = "Rectangle"
- Static method: is_square(width, height) - returns True if width == height

Example usage:

```
rect = Rectangle(5, 10)
print(rect.area())           # 50
print(rect.perimeter())      # 30
print(Rectangle.is_square(5, 5)) # True
print(Rectangle.is_square(5, 10)) # False
```

Exercise 2: Student and Course System

Model a student enrollment system:

Task

Create two classes with association:

- Course class:
 - Attributes: `course_name`, `course_code`, `credits`
 - Method: `get_info()`
- Student class:
 - Attributes: `name`, `student_id`, `enrolled_courses` (list)
 - Methods: `enroll(course)`, `list_courses()`, `total_credits()`

Bonus: Draw the UML diagram for these two classes showing the association.

Exercise 3: Temperature Converter

Create a utility class for temperature conversion:

Task

Create a `TemperatureConverter` class with static methods:

- `celsius_to_fahrenheit(celsius)`
- `fahrenheit_to_celsius(fahrenheit)`
- `celsius_to_kelvin(celsius)`
- `kelvin_to_celsius(kelvin)`

Example usage:

```
print(TemperatureConverter.celsius_to_fahrenheit(0))  
    # 32.0  
print(TemperatureConverter.fahrenheit_to_celsius(32))  
    # 0.0  
print(TemperatureConverter.celsius_to_kelvin(0))  
    # 273.15
```

Summary

What We Learned:

Core Concepts

- **OOP Basics:** Programming with objects and classes
- **Classes vs Objects:** Blueprints vs instances
- **self keyword:** Reference to current instance
- **Instance Members:** Unique data and methods for each object
- **Class Members:** Shared data and methods across instances
- **Static Methods:** Utility functions without instance/class access

Advanced Topics

- **Association:** Modeling relationships between classes
- **UML Diagrams:** Visual representation of classes
- **OOP Properties:** Overview of encapsulation, inheritance, polymorphism, abstraction

You should now be able to:

- Create classes with attributes and methods
- Instantiate objects from classes
- Use the `self` keyword correctly
- Differentiate instance, class, and static methods
- Apply appropriate decorators (`@classmethod`, `@staticmethod`)
- Model "has-a" relationships with association
- Read and create basic UML class diagrams
- Follow OOP naming conventions and best practices
- Design classes with single responsibility

Week 6: Advanced OOP Concepts

- **Encapsulation:** Private/protected members, getters/setters
- **Inheritance:** Parent-child relationships, `super()`
- **Polymorphism:** Method overriding, duck typing
- **Abstraction:** Abstract classes and methods
- **Magic Methods:** `__str__`, `__repr__`, `__eq__`, `__add__`
- **Advanced UML:** Inheritance hierarchies, aggregation, composition
- **Design Patterns:** Common solutions to recurring problems

Homework

- Complete all practice exercises
- Build a simple OOP project (e.g., Bank System, Student Manager)
- Practice drawing UML diagrams for your classes
- Review the four OOP properties

Thank You!

Thank you for your attention!

Keep Practicing!

Access Course Materials:

Download Course Materials



**Center for Artificial Intelligence & Emerging
Technologies**