```
#     **Day 8 of Training at Ansh Info Tech**
##    ***Topics Covered***
###   **Object Oriented Programming in Python**

* **Abstraction in Python**
* **Encapsulation in Python**
* **Access Specifiers in Python**
* **Class Diagram**
* **Questions on Access Specifiers**
* **10+ Practice Questions on these Topics**
```

Day 8 of Training at Ansh Info Tech

Topics Covered

Object Oriented Programming in Python

- **Abstraction in Python**
- **Encapsulation in Python**
- **Access Specifiers in Python**
- **Class Diagram**
- **Questions on Access Specifiers**
- **10+ Practice Questions on these Topics**

```python
# ability to use something without having to know the details of how it is working is abstraction
# we can put a lock on that datta by adding a double underscore in front of it.
# Adding a double underscore makes the attribute private and are accessible only inside the class.
# This method of restricting access is Encapsulation
# When we put a double underscore in front of an attribute, python changes its name to _className__attributeName
class Student:
  def __init__(self, name, age):
    self.__name = name
    self.__age = age
  def get_name(self):
    return self.__name
  def get_age(self):
    return self.__age
  def set_name(self, name):
    self.__name = name
  def set_age(self, age):
    self.__age = age
s1 = Student("John", 20)
print(s1.get_name())
print(s1.get_age())
s1.set_name("Jane")
s1.set_age(21)
print(s1.get_name())
print(s1.get_age())
```

```
John
20
Jane
21
```

Exercise on Access Specifiers - Level 1

In the Athlete class given below,

- make all the attributes private and
- add the necessary accessor and mutator methods

Represent Maria, the runner and make her run.

```python
class Athlete:
    def __init__(self,name,gender):
        self.name=name
        self.gender=gender

    def running(self):
        if(self.gender=="girl"):
            print("150mtr running")
        else:
            print("200mtr running")
```

```
class Athlete:
  def __init__(self, name, gender):
    self.__name = name
    self.__gender = gender
  def get_name(self):
    return self.__name
  def get_gender(self):
    return self.__gender
  def set_name(self, name):
    self.__name = name
  def set_gender(self, gender):
    self.__gender = gender
  def running(self):
    if self.__gender == "girl":
      print("150 mtr running")
    else:
      print("200 mtr running")
a1 = Athlete("Maria", "girl")
a1.running()

a2 = Athlete("John", "boy")
a2.running()
a2.__name = "Hero"
print(a2._Athlete__name)
```

```
150 mtr running
200 mtr running
John
```

Practice Questions:

Question 1: Library Management System

Scenario: You are tasked with developing a library management system where users can borrow and return books.

Requirements:

Create a Book class that has the following private attributes: title, author, and isbn. Implement methods to get and set the values of these attributes (getters and setters). Create a Library class that has a list of Book objects. Implement methods to add a new book, remove a book by its ISBN, and list all available books. Use encapsulation to ensure that the books can only be manipulated through the Library class. Task:

Implement the Book and Library classes with proper encapsulation. Add methods to add, remove, and list books in the library.

```
class Book:
  def __init__(self, title, author, isbn):
    self.__title = title
    self.__author = author
    self.__isbn = isbn
  def get_title(self):
    return self.__title
  def get_author(self):
    return self.__author
  def get_isbn(self):
    return self.__isbn
  def set_title(self, title):
    self.__title = title
  def set_author(self, author):
    self.__author = author
  def set_isbn(self, isbn):
    self.__isbn = isbn

class Library:
  def __init__(self):
    self.__books = []
  def add_book(self, book):
    self.__books.append(book)
  def remove_book(self, isbn):
    for book in self.__books:
      if book.get_isbn() == isbn:
        self.__books.remove(book)
        break
    else:
      print("Book not found")
  def list_books(self):
    for book in self.__books:
      print(f"Title: {book.get_title()}, Author: {book.get_author()}, ISBN: {book.get_isbn()}")

Library = Library()
Library.add_book(Book("Book 1", "Author 1", "ISBN1"))
Library.add_book(Book("Book 2", "Author 2", "ISBN2"))
Library.remove_book("ISBN2")
Library.list_books()
```

```
Title: Book 1, Author: Author 1, ISBN: ISBN1
```

Question 2: Bank Account Management Scenario: Develop a bank account management system where users can create accounts, deposit, and withdraw money.

Requirements:

Create a BankAccount class with private attributes: account_number, account_holder, and balance. Implement methods to get the account details and balance (getters). Implement methods to deposit and withdraw money, ensuring that the balance cannot go negative. Use encapsulation to ensure that the balance cannot be directly modified. Task:

Implement the BankAccount class with proper encapsulation. Add methods to deposit and withdraw money, and to get account details.

```python
class BankAccount:
  def __init__(self, account_number, account_holder, balance):
    self.__account_number = account_number
    self.__account_holder = account_holder
    self.__balance = balance
  def get_account_number(self):
    return self.__account_number
  def get_account_holder(self):
    return self.__account_holder
  def get_balance(self):
    return self.__balance
  def deposit(self, amount):
    if amount > 0:
      self.__balance += amount
  def withdraw(self, amount):
    if amount > 0 and self.__balance - amount >= 0:
      self.__balance -= amount
    else:
      print("Insufficient Balance, cannot Withdraw Money")
  def displayAccountDetails(self):
    print("Account Number:", self.__account_number)
    print("Account Holder:", self.__account_holder)
    print("Balance:", self.__balance)
Account1 = BankAccount("123456789", "John Doe", 1000)
Account1.deposit(500)
Account1.withdraw(200)
Account1.displayAccountDetails()
Account2 = BankAccount("987654321", "Jane Smith", 2000)
Account2.deposit(1000)
Account2.withdraw(5500)
Account2.displayAccountDetails()
```

```
Account Number: 123456789
Account Holder: John Doe
Balance: 1300
Insufficient Balance, cannot Withdraw Money
Account Number: 987654321
Account Holder: Jane Smith
Balance: 3000
```

Question 3: Employee Management System Scenario: You are developing an employee management system for a company.

Requirements:

Create an Employee class with private attributes: employee_id, name, position, and salary. Implement methods to get and set the values of these attributes, with validation on the salary (e.g., salary cannot be negative). Create a Department class that has a list of Employee objects. Implement methods to add a new employee, remove an employee by their ID, and list all employees in the department. Use encapsulation to ensure that employees can only be manipulated through the Department class. Task:

Implement the Employee and Department classes with proper encapsulation. Add methods to add, remove, and list employees in the department.

```
class Employee:
  def __init__(self, employee_id, name, position, salary):
    self.__employee_id = employee_id
    self.__name = name
    self.__position = position
    if(salary > 0):
      self.__salary = salary
    else:
      raise ValueError("Salary cannot be negative")
  def get_id(self):
    return self.__employee_id
  def get_name(self):
    return self.__name
  def get_position(self):
    return self.__position
  def get_salary(self):
    return self.__salary
  def set_id(self, employee_id):
    self.__employee_id = employee_id
  def set_name(self, name):
    self.__name = name
  def set_position(self, position):
    self.__position = position
class Department:
  def __init__(self):
    self.__employees = []
  def add_employee(self, employee):
    self.__employees.append(employee)
  def remove_employee(self, employee_id):
    for employee in self.__employees:
      if employee.get_id() == employee_id:
        self.__employees.remove(employee)
        return
    else:
      print("Employee Not Found")
  def list_employees(self):
    for employee in self.__employees:
      print("Employee ID:", employee.get_id())
      print("Name:", employee.get_name())
      print("Position:", employee.get_position())
      print("Salary:", employee.get_salary())

dept = Department()
dept.add_employee(Employee("1", "John Doe", "Manager", 50000))
dept.add_employee(Employee("2", "Jane Smith", "Developer", 60000))
dept.remove_employee("2")
dept.list_employees()
```

```
Employee not found
Employee ID: 1
Name: John Doe
Position: Manager
Salary: 50000
```

Question 4: Online Shopping System Scenario: Develop an online shopping system where users can add products to a shopping cart and view the total cost.

Requirements:

Create a Product class with private attributes: product_id, name, and price. Implement methods to get the product details (getters). Create a ShoppingCart class that has a list of Product objects. Implement methods to add a product to the cart, remove a product by its ID, and calculate the total cost of all products in the cart. Use encapsulation to ensure that products in the cart can only be manipulated through the ShoppingCart class. Task:

Implement the Product and ShoppingCart classes with proper encapsulation. Add methods to add, remove, and list products in the cart, and to calculate the total cost.

```
class Product:
  def __init__(self, product_id, name, price):
    self.__product_id = product_id
    self.__name = name
    self.__price = price
  def get_product_id(self):
    return self.__product_id
  def get_name(self):
    return self.__name
  def get_price(self):
    return self.__price
  def set_product_id(self, product_id):
    self.__product_id = product_id
  def set_name(self, name):
    self.__name = name
  def set_price(self, price):
    self.__price = price
  def displayProductDetails(self):
    print("Product ID:", self.__product_id)
    print("Name:", self.__name)
    print("Price:", self.__price)
class ShoppingCart:
  def __init__(self):
    self.__products = []
  def add_product(self, product):
    self.__products.append(product)
  def remove_product(self, product_id):
    for product in self.__products:
      if product.get_product_id() == product_id:
        self.__products.remove(product)
        return
    else:
      print("Product Not Found")
  def calculate_total_cost(self):
    total_cost = 0
    for product in self.__products:
      total_cost += product.get_price()
    return total_cost
  def list_products(self):
    for product in self.__products:
      product.displayProductDetails()

ShoppingCart = ShoppingCart()
ShoppingCart.add_product(Product("1", "Product 1", 10))
ShoppingCart.add_product(Product("2", "Product 2", 20))
ShoppingCart.add_product(Product("3", "Product 3", 60))
ShoppingCart.add_product(Product("4", "Product 4", 50))
ShoppingCart.remove_product("2")
print("Total Cost:", ShoppingCart.calculate_total_cost())
ShoppingCart.list_products()
```

```
Total Cost: 120
Product ID: 1
Name: Product 1
Price: 10
Product ID: 3
Name: Product 3
Price: 60
Product ID: 4
Name: Product 4
Price: 50
```

Question 5: School Management System Scenario: Develop a school management system where you can manage students and their courses.

Requirements:

Create a Student class with private attributes: student_id, name, and courses (a list of course names). Implement methods to get and set the student's details and courses (getters and setters). Create a School class that has a list of Student objects. Implement methods to add a new student, remove a student by their ID, and list all students and their courses. Use encapsulation to ensure that students can only be manipulated through the School class. Task:

Implement the Student and School classes with proper encapsulation. Add methods to add, remove, and list students and their courses.

```
class Student:
  def __init__(self, student_id, name, courses):
    self.__student_id = student_id
    self.__name = name
    self.__courses = courses
  def get_student_id(self):
    return self.__student_id
  def get_name(self):
    return self.__name
  def get_courses(self):
    return self.__courses
  def set_student_id(self, student_id):
    self.__student_id = student_id
  def set_name(self, name):
    self.__name = name
  def set_courses(self, courses):
    self.__courses = courses
class School:
  def __init__(self):
    self.__students = []
  def add_student(self, student):
    self.__students.append(student)
  def remove_student(self, student_id):
    for student in self.__students:
      if student.get_student_id() == student_id:
        self.__students.remove(student)
        return
    else:
      print("Student Not Found")
  def list_students(self):
    for student in self.__students:
      print("Student ID:", student.get_student_id())
      print("Name:", student.get_name())
      print("Courses:", student.get_courses())

School = School()
School.add_student(Student("1", "John Doe", ["Math", "Science"]))
School.add_student(Student("2", "Jane Smith", ["English", "History"]))
School.add_student(Student("3", "Michael Johnson", ["Computer Science", "Art"]))
School.remove_student("2")
School.list_students()
```

```
Student ID: 1
Name: John Doe
Courses: ['Math', 'Science']
Student ID: 3
Name: Michael Johnson
Courses: ['Computer Science', 'Art']
```

Question 6: Car Rental System Scenario: Develop a car rental system where users can rent and return cars.

Requirements:

Create a Car class with private attributes: car_id, make, model, and availability (boolean). Implement methods to get and set the car's details and availability (getters and setters). Create a CarRental class that has a list of Car objects. Implement methods to add a new car, remove a car by its ID, and list all available cars. Implement methods to rent a car (set its availability to False) and return a car (set its availability to True). Use encapsulation to ensure that cars can only be manipulated through the CarRental class. Task:

Implement the Car and CarRental classes with proper encapsulation. Add methods to add, remove, list, rent, and return cars.

```
class Car:
  def __init__(self, car_id, make, model, availability):
    self.__car_id = car_id
    self.__make = make
    self.__model = model
    self.__availability = availability
  def get_car_id(self):
    return self.__car_id
  def get_make(self):
    return self.__make
  def get_model(self):
    return self.__model
  def get_availability(self):
    return self.__availability
  def set_car_id(self, car_id):
    self.__car_id = car_id
  def set_make(self, make):
    self.__make = make
  def set_model(self, model):
    self.__model = model
  def set_availability(self, availability):
    self.__availability = availability
  def displayCarDetails(self):
    print("Car ID:", self.__car_id)
    print("Make:", self.__make)
    print("Model:", self.__model)
    print("Availability:", self.__availability)
class carRental:
  def __init__(self):
    self.__cars = []
  def add_car(self, car):
    self.__cars.append(car)
  def remove_car(self, car_id):
    for car in self.__cars:
      if car.get_car_id() == car_id:
        self.__cars.remove(car)
        return
    else:
      print("Car Not Found")
  def list_cars(self):
    for car in self.__cars:
      car.displayCarDetails()

  def rent_car(self, car_id):
    for car in self.__cars:
      if car.get_car_id() == car_id and car.get_availability()==True:
        car.set_availability(False)
        return
    else:
      print("Car Not Found or Car Not Available")
  def return_car(self, car_id):
    for car in self.__cars:
      if car.get_car_id() == car_id and not car.get_availability():
        car.set_availability(True)
        return
    else:
      print("Car Not Found or Car Already Available")

carRental = carRental()
carRental.add_car(Car("1", "Toyota", "Corolla", True))
carRental.add_car(Car("2", "Honda", "Civic", True))
carRental.add_car(Car("3", "Ford", "Mustang", True))
carRental.remove_car("2")
carRental.list_cars()
carRental.rent_car("1")
carRental.return_car("3")
carRental.rent_car("3")
carRental.return_car("3")
carRental.list_cars()
```

```
Car ID: 1
Make: Toyota
Model: Corolla
Availability: True
Car ID: 3
Make: Ford
Model: Mustang
Availability: True
```

```
Car Not Found or Car Already Available
Car ID: 1
Make: Toyota
Model: Corolla
Availability: False
Car ID: 3
Make: Ford
Model: Mustang
Availability: True
```

Class Diagram

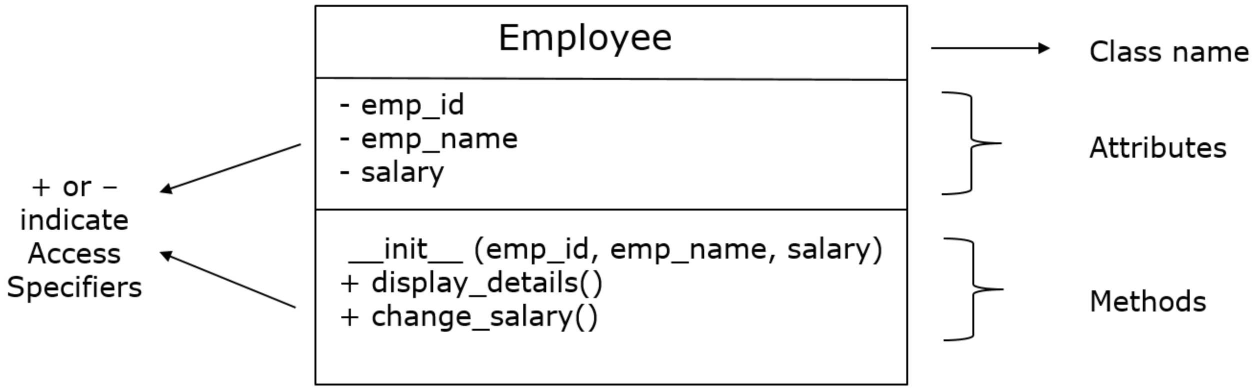
A lot of things can go wrong in communication.

To ensure that programmers all over understand each other properly, we need a common way of representing a class. This is called as a class diagram. This is similar to a circuit diagram or a plan or machine diagram which allows engineers to understand each others' ideas clearly.

Visibility Notation:

Visibility notations indicate the access level of attributes and methods. Common visibility notations include:

- **+** for public (visible to all classes)
- **-** for private (visible only within the class)
- **#** for protected (visible to subclasses)
- **~** for package or default visibility (visible to classes in the same package)



Note: We can create private methods by adding a double underscore in front of it, just like private variables. Also, if a method has both leading and trailing double underscores (like __ init __ , __ str __, etc) it indicates that it is a special built-in method. As per coding convention, we are not supposed to create our own methods with both leading and trailing underscores.

- If there are both leading and trailing underscores for a method, it is a special built in method

- Setter methods are called as mutator methods (as they change or mutate the value) and the getter methods are called accessor methods (as they access the values)

Abstraction & Encapsulation - Summary

- Encapsulation is preventing access to a data outside the class
- Adding a __ in front of a attribute makes it private
- In python, adding a __ changes the name of the attribute to _Classname__attribute

1. Bank Account Management System Create a BankAccount class with private attributes account_number, account_holder, and balance. Implement getter and setter methods for account_number and account_holder. Implement a setter method for balance that checks if the balance being set is non-negative.

```
class bankAccount:
  def __init__(self, account_number, account_holder, balance):
    self.__account_number = account_number
    self.__account_holder = account_holder
    if(balance>=0):
      self.__balance = balance
    else:
      print("Balance Cannot be Negative")
      self.__balance = 0

  def getAccountNumber(self):
    return self.__account_number

  def getAccountHolder(self):
    return self.__account_holder

  def getBalance(self):
    return self.__balance

  def setAccountNumber(self, AccNum):
    self.__account_number = AccNum

  def setAccountHolder(self, AccHolder):
    self.__account_holder = AccHolder

  def setBalance(self, newBalance):
    if(newBalance>=0):
      self.__balance = newBalance
    else:
      print("Balance Cannot be Negative")

Account1 = bankAccount("A101", "Hero", 1000)
print(Account1.getAccountNumber())
print(Account1.getAccountHolder())
print(Account1.getBalance())
Account1.setAccountNumber("S101")
Account1.setAccountHolder("Zero")
Account1.setBalance(-100)
print(Account1.getAccountNumber())
print(Account1.getAccountHolder())
print(Account1.getBalance())
Account1.setBalance(500)
print(Account1.getBalance())
```

```
A101
Hero
1000
Balance Cannot be Negative
S101
Zero
1000
500
```

2. Student Record System Create a Student class with private attributes student_id, name, age, and grades (a list of integers). Implement getter and setter methods for name and age. Implement a setter method for grades that ensures all grades are within a valid range (e.g., 0-100).

```
class Student:
    def __init__(self, student_id, name, age, grades):
        self.__student_id = student_id
        self.__name = name
        self.__age = age
        self.__grades = grades

    # Getter for name
    def get_name(self):
        return self.__name

    # Setter for name
    def set_name(self, name):
        self.__name = name

    # Getter for age
    def get_age(self):
        return self.__age

    # Setter for age
```

3. Employee Management System Create an Employee class with private attributes employee_id, name, position, and salary. Implement getter and setter methods for position. Implement a setter method for salary that ensures the salary being set is positive.

```
    def get_grades(self):
```

```
class Employee:
    def __init__(self, employee_id, name, position, salary):
        self.__employee_id = employee_id
        self.__name = name
        self.__position = position
        self.__salary = salary

    # Getter for position
    def get_position(self):
        return self.__position

    # Setter for position
    def set_position(self, position):
        self.__position = position

    # Getter for salary
    def get_salary(self):
        return self.__salary

    # Setter for salary with validation
    def set_salary(self, salary):
        if salary > 0:
```