ICS220 - Final Project

Salama Al Neyadi , 202315108

Ghazlan Al Ketbi 202320532

Professor. Areej Abdulfattah

May 13th, 2025

Grand Prix Experience Ticket Booking System

1. Designing the UML class diagram

Class: User

- Attributes:
    - - userID: Integer
    - - username: String
    - - password: String
    - - email: String
    - - phoneNumber: String
- Methods:
    - + login(username, password): Returns Boolean (True if credentials match)
    - + logout(): Returns void (no return value)
    - + updateProfile(details): Returns void
    - + getOrderHistory(): Returns List of Order objects

Class: Customer (inherits from User)

- Attributes:
    - - firstName: String
    - - lastName: String
    - - orderHistory: List of orders
- Methods:
    - + createAccount(details): Boolean

      → Returns True if the account is successfully created.
    - + viewAccount(): Dictionary

      → Returns customer account details as a dictionary.

- + modifyAccount(details): Boolean

  → Returns True if the update was successful.

- + deleteAccount(): Boolean

  → Returns True if the account was deleted.

- + viewOrderHistory(): List[Order]

  → Returns a list of past orders.

- + placeOrder(tickets, payment): Order

  → Creates and returns a new Order object.

Class: Admin (inherits from User)

- **Attributes:**
  - - staffID: String
- **Methods:**
  - + viewSalesReport(): Dictionary

    → Returns sales data as a dictionary or report format.

  - + modifyDiscounts(discount): Boolean

    → Returns True if discount settings are successfully updated.

  - + trackTicketSales(): Integer

    → Returns the number of tickets sold.

  - + viewAllCustomers(): List[Customer]

    → Returns a list of all registered customers.

Class: Ticket

- **Attributes:**
  - - ticketID: Integer
  - - basePrice: Float
  - - eventDate: Date

- - isAvailable: Boolean

- **Methods:**
  - + calculatePrice(): Float

    → Returns the calculated price of the ticket.
  - + isValid(): Boolean

    → Returns True if the ticket is valid for use.
  - + getTicketDetails(): Dictionary

    → Returns ticket information as a dictionary.

Class: SingleRacePass (inherits from Ticket)

- **Attributes:**
  - - raceDay: String
  - - seatLocation: String
- **Methods:**
  - + calculatePrice(): Float
  - → Calculates and returns the ticket price with race-specific adjustments.

Class: WeekendPackage (inherits from Ticket)

- **Attributes:**
  - - includedDays: List of Strings
  - - packageType: String
- **Methods:**
  - + calculatePrice(): Float
  - → Calculates and returns the ticket price based on the number and type of included days.

Class: SeasonMembership (inherits from Ticket)

- **Attributes:**
  - - validFrom: Date
  - - validUntil: Date
  - - membershipLevel: String
- **Methods:**
  - + calculatePrice(): Float

    → Calculates and returns the total price for the full season.
  - + getRemainingEvents(): List[Event]

    → Returns a list of upcoming events remaining in the season.

Class: GroupDiscount

- **Attributes:**
  - - groupSize: Integer
  - - discountPercentage: Float
- **Methods:**
  - + calculateDiscount(basePrice): Returns the discounted price

Class: Order

- **Attributes:**
  - - orderID: Integer
  - - customer: Customer
  - - tickets: List of Ticket
  - - totalAmount: Float

- - orderDate: Date
- - status: String
- - paymentMethod: Payment

- **Methods:**
  - + calculateTotal(): Float

    → Sums and returns the total price of all tickets in the order.
  - + addTicket(ticket: Ticket): Order

    → Adds a ticket to the order and returns the updated order.
  - + removeTicket(ticketID: Integer): Order

    → Removes a ticket by ID and returns the updated order.
  - + updateStatus(status: String): String

    → Updates the order status and returns the new status.
  - + getOrderDetails(): Dictionary

    → Returns order information as a dictionary.

Class: Payment

- **Attributes:**
  - - paymentID: Integer
  - - amount: Float
  - - paymentDate: Date
  - - paymentStatus: String
- **Methods:**
  - + processPayment(): String

    → Confirms the payment and returns a confirmation message or updated status.
  - + refundPayment(): String

    → Processes a refund and returns a refund confirmation or updated status.
  - + getPaymentDetails(): Dictionary

    → Returns payment information as a dictionary.

Class: CreditCardPayment (inherits from Payment)

- **Attributes:**
    - - cardNumber: String
    - - cardholderName: String
    - - expiryDate: Date
    - - cvv: String
- **Methods:**
    - + validateCard(): Boolean

      → Validates the credit card details and returns True if valid, False otherwise.
    - + processPayment(): String

      → Processes the credit card payment and returns a confirmation message

Class: DigitalWalletPayment (inherits from Payment)

- **Attributes:**
    - - walletID: String
    - - provider: String
- **Methods:**
- + processPayment(): String
- → Processes the digital wallet payment and returns a confirmation message or status.

Class: Event

- **Attributes:**
  - - eventID: Integer
  - - eventName: String
  - - eventDate: Date
  - - location: String
  - - totalCapacity: Integer
  - - soldTickets: Integer
- **Methods:**
  - + getAvailableTickets(): Integer

    → Returns the number of tickets left (totalCapacity - soldTickets).
  - + updateSoldTickets(count: Integer): Integer

    → Updates the sold ticket count by the given count and returns the updated soldTickets total.
  - + getEventDetails(): Dictionary

    → Returns event information as a dictionary.

Class: TicketingSystem

- **Methods:**
  - + registerCustomer(details: Dictionary): Customer

    → Creates and returns a new Customer object based on the provided details.
  - + loginUser(username: String, password: String): User

    → Returns a User object if credentials are valid; otherwise, returns null or an error.
  - + searchEvents(criteria: Dictionary): List<Event>

    → Returns a list of Event objects that match the search criteria.
  - + bookTicket(event: Event, ticketType: String, customer: Customer): Order

    → Books a ticket for the given event and customer, returns the created Order.
  - + processPayment(order: Order, paymentDetails: Dictionary): Payment

    → Handles the payment for an order and returns a Payment object.

- + generateSalesReport(): Dictionary

  → Returns a summary of ticket sales data as a dictionary.

Class: Discount

- **Attributes:**
  - - discountCode: String
  - - discountPercentage: Float
  - - validFrom: Date
  - - validUntil: Date
  - - isActive: Boolean
- **Methods:**
  - +applyDiscount(price: Float): Float

    → Reduces the given price based on the discount percentage and returns the new price.
  - + isValid(): Boolean

    → Returns true if the discount is currently active and within the valid date range.

Inheritance (is-a relationships):

- Customer is a type of User.
- Admin is a type of User.
- SingleRacePass, WeekendPackage, and SeasonMembership are types of Ticket.
- CreditCardPayment and DigitalWalletPayment are types of Payment.

Inheritance Relationships

1. **User Hierarchy**:
    - User is the parent class with common attributes (userID, username, password, etc.) and methods (login, logout, etc.)
    - Customer and Admin inherit from User, representing the different types of users in the system
2. **Ticket Hierarchy**:
    - Ticket is the base class with common attributes (ticketID, basePrice, etc.) and methods (calculatePrice, isValid, etc.)
    - SingleRacePass, WeekendPackage, and SeasonMembership inherit from Ticket, each implementing specialized price calculation methods
3. **Payment Hierarchy**:
    - Payment is the parent class with common payment attributes and methods
    - CreditCardPayment and DigitalWalletPayment inherit from Payment, with specialized processing methods

Association Relationships

1. **Customer-Order** (1 to many):
    - A customer can place many orders (0 or more)
    - Each order belongs to exactly one customer
2. **Order-Ticket** (many to many):
    - An order can contain multiple tickets (at least 1)
    - A ticket can be part of multiple orders (e.g., group bookings)
3. **Order-Payment** (1 to 1):
    - Each order is paid via exactly one payment method
    - Each payment is associated with exactly one order
4. **Order-Discount** (many to 0..1):
    - Multiple orders can use the same discount
    - An order may or may not have a discount applied

5. **Ticket-Event** (many to 1):
    - Each ticket is for exactly one event
    - An event can have many tickets (0 or more)
6. **GroupDiscount-Ticket** (association):
    - Group discounts apply to tickets (general association)
7. **Admin-TicketingSystem** (1 to 1):
    - An admin administers the ticketing system
    - The ticketing system has one admin managing it
8. **TicketingSystem Management** (1 to many):
    - The ticketing system manages customers, events, tickets, and orders
    - This is shown through the "manages" associations

Assumptions

1. **Access Modifiers:**
    - Reasoning: Using private (-) attributes ensures encapsulation and protects class data from direct external access. This aligns with object-oriented principles and improves maintainability.
    - Application: All class attributes are private, and public (+) getter/setter methods are used to access or modify them.

2. **Multiplicity**:
    - Reasoning: The multiplicity reflects real-world constraints. For example, a customer logically places many orders, while each order must belong to one customer. Similarly, an event can host many tickets, but each ticket must be for one event.
    - Application: Relationships such as Customer-Order (1 to many) and Ticket-Event (many to 1) capture these scenarios accurately.

3. **Navigability**:
    - Reasoning: Directional arrows clarify which class holds the reference to the other. For instance, a Customer can retrieve their list of Orders, but Order doesn't need to access all customers.
    - Application: Arrows are added where needed to improve UML readability and indicate logical control flow.

4. **Method Return Types**:
    - Reasoning: Defining return types (e.g., void, boolean, List<Ticket>) helps specify what kind of data is expected, which aids in design clarity and implementation.
    - Application: Method signatures in the diagram include return types to support strong typing.

5. **Collection Types**:
    - Reasoning: Collections like List or Dictionary are used where multiple objects are involved (e.g., a list of tickets in an order). This helps model real-world complexity.
    - Application: Used List<Ticket> in Order, Dictionary<Event, Ticket> in TicketingSystem, etc., to show aggregation of related entities.

6. **GroupDiscount-Ticket Association**

- Reasoning: Since a group discount might be conditionally applied to tickets (based on quantity or group booking), a general association is used instead of a strong relationship like composition or aggregation.
- Application: This keeps the model flexible and extensible for business rules.

7. Admin-TicketingSystem (1 to 1)

- Reasoning: For simplicity, the system assumes one administrator is responsible. This avoids complexity of role-based access in the initial model and is suitable for a prototype or MVP.
- Application: A one-to-one association is used between Admin and TicketingSystem.

8. TicketingSystem Management Associations

- Reasoning: The TicketingSystem class acts as the controller/manager, handling major system-level entities like Customers, Events, and Orders. These are not class-specific associations but rather part of system architecture.
- Application: "Manages" associations indicate that TicketingSystem orchestrates operations.

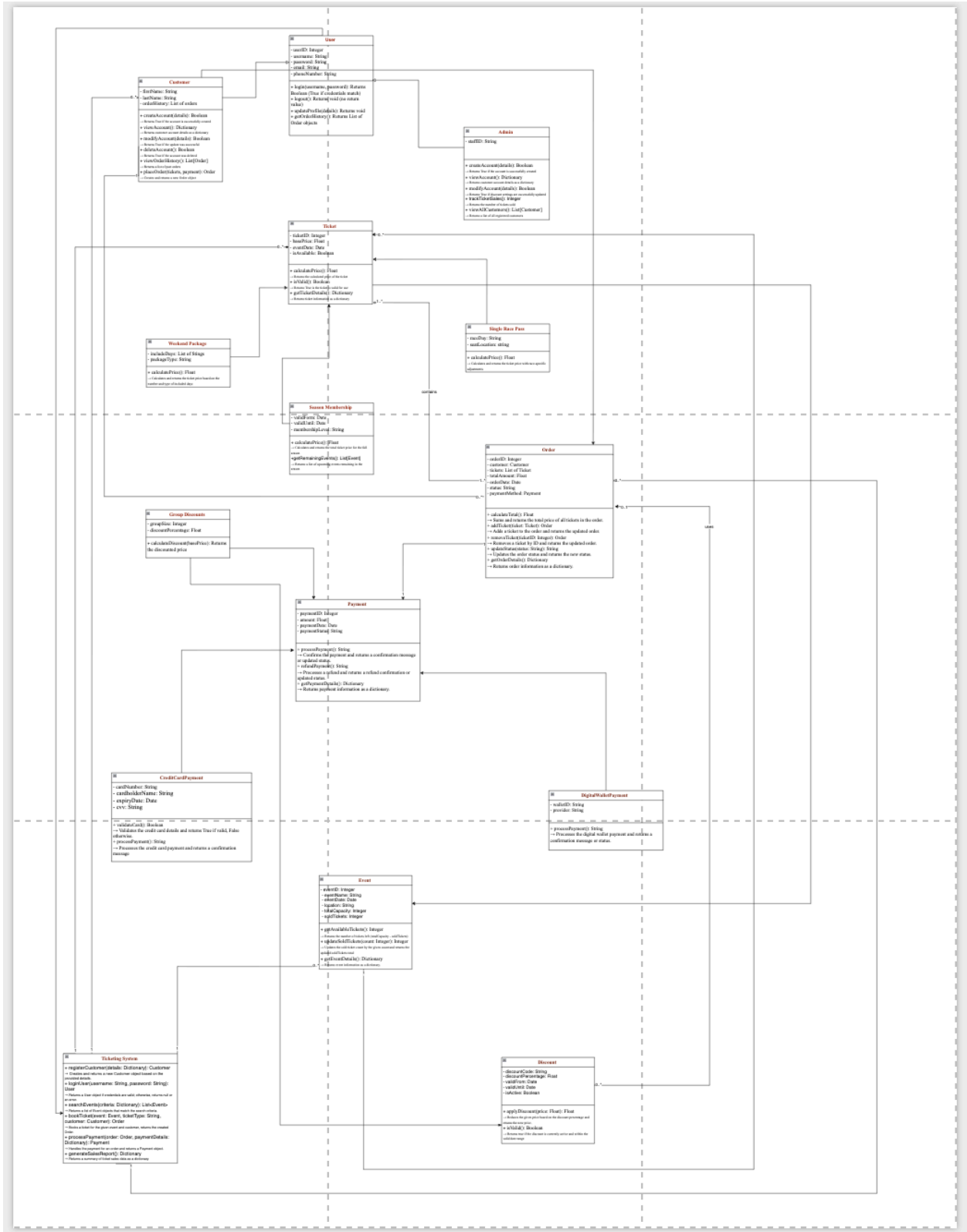
Conclusion :

In conclusion, the UML class diagram clearly shows how the Grand Prix Experience Ticket Booking System is structured. It highlights the different classes, their attributes, and methods, as well as the relationships between users, tickets, orders, payments, and events. By outlining both inheritance and association links, the diagram makes it easier to understand how the system functions and how its parts interact. This visual representation supports better planning, development, and future maintenance, ensuring the system is scalable, organized, and user-friendly.

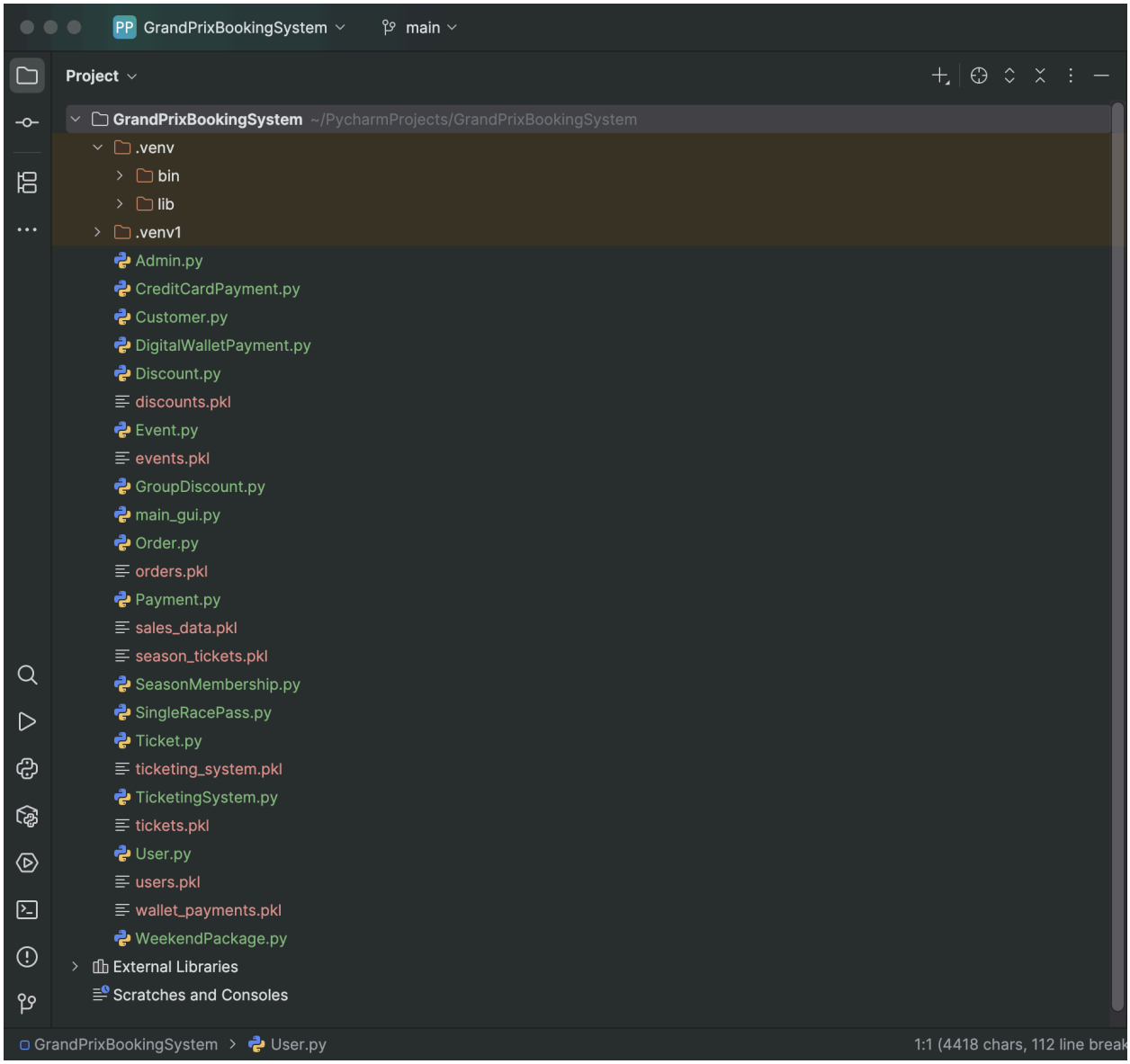
UML diagram link :

- https://viewer.diagrams.net/?tags=%7B%7D&lightbox=1&highlight=0000ff&edit=_blank&layers=1&nav=1&title=final%20assignmnet.drawio&dark=auto#Uhttps%3A%2F%2Fdrive.google.com%2Fuc%3Fid%3D1j7ewrDTpftPgZ3efKD2ReT7hGKkbhjJa%26export%3Ddownload

UML Visualization:



2. Writing the Python code to implement our UML diagram.



- In the Grand Prix Experience Ticket Booking System, we use Python's pickle library to store all important program data persistently in binary files. This includes users (users.pkl), tickets (tickets.pkl), orders (orders.pkl), events (events.pkl), payments (sales_data.pkl), and discounts (discounts.pkl). Each file corresponds to a specific class or module in the system, ensuring clear organization and separation of data types. When the application starts, data is loaded from these files using pickle.load(), and after any changes such as creating a new user or processing a payment the updated data is saved back using pickle.dump().To ensure smooth operation, the

system uses the pickle library to save and load data like users, tickets, and orders in binary files. If a file is not found or is empty, the system simply creates a new one or continues with default data. This makes sure the program doesn't crash and that important data is kept safe and ready to use the next time the program runs.

Class User:

```
import pickle  # For saving/loading binary data
import os  # For checking file existence

# Load users from file or return an empty list if the file does not exist
def load_users():
    if os.path.exists('users.pkl'):
        with open('users.pkl', 'rb') as file:
            return pickle.load(file)
    return []

# Save users to a binary file
def save_users(users):
    with open('users.pkl', 'wb') as file:
        pickle.dump(users, file)

# Load sales data from file or return default data structure
def load_sales_data():
    if os.path.exists('sales_data.pkl'):
        with open('sales_data.pkl', 'rb') as file:
            return pickle.load(file)
    return {"total_sales": 0, "tickets_sold": 0, "discount": 0}

# Save sales data to binary file
def save_sales_data(data):
    with open('sales_data.pkl', 'wb') as file:
        pickle.dump(data, file)

# Base class for all users
class User:
    def __init__(self, user_id, username, password, email, phone_number):
        self._userID = user_id
        self._username = username
        self._password = password
        self._email = email
        self._phoneNumber = phone_number
        self._orderHistory = []

    # Getter and setter for userID
    def get_userID(self):
```

```
        return self._userID

    def set_userID(self, user_id):
        self._userID = user_id

    # Getter and setter for username
    def get_username(self):
        return self._username

    def set_username(self, username):
        self._username = username

    # Getter and setter for password
    def get_password(self):
        return self._password

    def set_password(self, password):
        self._password = password

    # Getter and setter for email
    def get_email(self):
        return self._email

    def set_email(self, email):
        self._email = email

    # Getter and setter for phoneNumber
    def get_phoneNumber(self):
        return self._phoneNumber

    def set_phoneNumber(self, phone_number):
        self._phoneNumber = phone_number

    # Getter and setter for orderHistory
    def get_orderHistory(self):
        return self._orderHistory

    def set_orderHistory(self, order_history):
        self._orderHistory = order_history

    # Method to check login credentials
    def login(self, username, password):
        return self._username == username and self._password == password

    # Method to simulate logging out
    def logout(self):
        print(f"{self._username} has logged out.")

    # Method to update user profile using a dictionary
```

```
    def updateProfile(self, details):
        for key, value in details.items():
            if key == "userID":
                self.set_userID(value)
            elif key == "username":
                self.set_username(value)
            elif key == "password":
                self.set_password(value)
            elif key == "email":
                self.set_email(value)
            elif key == "phoneNumber":
                self.set_phoneNumber(value)

    # Method to get the order history
    def getOrderHistory(self):
        return self._orderHistory

# Customer inherits from User and can place orders
class Customer(User):
    def __init__(self, user_id, username, password, email, phone_number,
first_name, last_name):
        super().__init__(user_id, username, password, email, phone_number)
        self._first_name = first_name
        self._last_name = last_name

    # Getter and setter for first_name
    def get_first_name(self):
        return self._first_name

    def set_first_name(self, first_name):
        self._first_name = first_name

    # Getter and setter for last_name
    def get_last_name(self):
        return self._last_name

    def set_last_name(self, last_name):
        self._last_name = last_name

    # Method to place an order and update sales data
    def placeOrder(self, tickets_count, payment_amount):
        order = {"tickets": tickets_count, "payment": payment_amount}
        self._orderHistory.append(order)
        sales_data = load_sales_data()
        sales_data["tickets_sold"] += tickets_count
        sales_data["total_sales"] += payment_amount
        save_sales_data(sales_data)
        return {"status": "Order successful"}
```

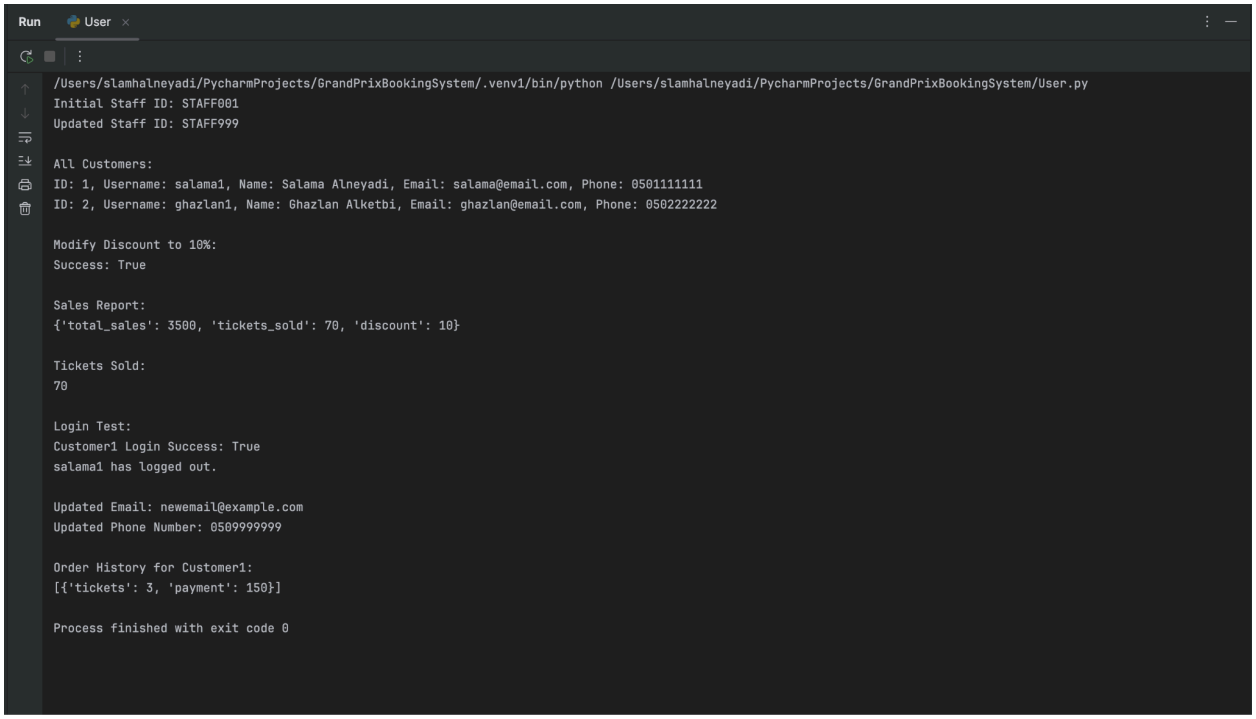
```
    # Method to return customer info as a string
    def get_customer_info(self):
        return f"ID: {self.get_userID()}, Username: {self.get_username()}, Name:
{self._first_name} {self._last_name}, Email: {self.get_email()}, Phone:
{self.get_phoneNumber()}"

# Admin inherits from User and can manage users and sales data
class Admin(User):
    def __init__(self, user_id, username, password, email, phone_number,
staff_id):
        super().__init__(user_id, username, password, email, phone_number)
        self._staffID = staff_id

    # Getter and setter for staffID
    def get_staffID(self):
        return self._staffID

    def set_staffID(self, staff_id):
        self._staffID = staff_id

    # Method to view the current sales report
    def viewSalesReport(self):
        return load_sales_data()

    # Method to modify the discount rate in the sales data
    def modifyDiscounts(self, discount):
        try:
            sales_data = load_sales_data()
            sales_data["discount"] = discount
            save_sales_data(sales_data)
            return True
        except Exception as e:
            print(f"Error modifying discount: {e}")
            return False

    # Method to track number of tickets sold
    def trackTicketSales(self):
        sales_data = load_sales_data()
        return sales_data.get("tickets_sold", 0)

    # Method to return a list of all Customer users
    def viewAllCustomers(self):
        users = load_users()
        return [user for user in users if isinstance(user, Customer)]

# Example usage and testing
if __name__ == "__main__":
    # Create customer objects
```

```
    customer1 = Customer(1, "salama1", "pass123", "salama@email.com",
"0501111111", "Salama", "Alneyadi")
    customer2 = Customer(2, "ghazlan1", "pass456", "ghazlan@email.com",
"0502222222", "Ghazlan", "Alketbi")

    # Save users to file
    save_users([customer1, customer2])

    # Place sample orders
    customer1.placeOrder(3, 150)
    customer2.placeOrder(2, 100)

    # Create an admin user
    admin = Admin(99, "adminuser", "adminpass", "admin@email.com", "0500000000",
"STAFF001")

    # View and update staff ID
    print("Initial Staff ID:", admin.get_staffID())
    admin.set_staffID("STAFF999")
    print("Updated Staff ID:", admin.get_staffID())

    # View all customers
    print("\nAll Customers:")
    for customer in admin.viewAllCustomers():
        print(customer.get_customer_info())

    # Modify discount value
    print("\nModify Discount to 10%:")
    print("Success:", admin.modifyDiscounts(10))

    # Display the full sales report
    print("\nSales Report:")
    print(admin.viewSalesReport())

    # Show total tickets sold
    print("\nTickets Sold:")
    print(admin.trackTicketSales())

    # Test login and logout
    print("\nLogin Test:")
    print("Customer1 Login Success:", customer1.login("salama1", "pass123"))
    customer1.logout()

    # Test updating customer profile
    customer1.updateProfile({"email": "newemail@example.com", "phoneNumber":
"0509999999"})
    print("\nUpdated Email:", customer1.get_email())
    print("Updated Phone Number:", customer1.get_phoneNumber())
```

```
# Display order history for customer1
print("\nOrder History for Customer1:")
print(customer1.getOrderHistory())
```

- The following code defines the User class and implements basic file operations using Python's pickle module to store and retrieve user data from a binary file (users.pkl). It includes methods for login, logout, profile updates, and viewing order history.

Output for class User:



Class Customer (inherits from User)

```
import pickle  # Used for saving and loading objects in binary format

import os      # Used to check if the file exists



# User class defines basic user information

class User:
```

```
def __init__(self, user_id, username, password, email, phone_number):
    self._userID = user_id                # Unique user ID
    self._username = username             # Username
    self._password = password             # Password
    self._email = email                   # Email address
    self._phoneNumber = phone_number      # Phone number
# Getter methods
def get_userID(self):
    return self._userID
def get_username(self):
    return self._username
def get_password(self):
    return self._password
def get_email(self):
    return self._email
def get_phoneNumber(self):
    return self._phoneNumber
# Setter methods
def set_userID(self, user_id):
    self._userID = user_id
def set_username(self, username):
    self._username = username
def set_password(self, password):
    self._password = password
def set_email(self, email):
    self._email = email
```

```
  def set_phoneNumber(self, phone_number):
      self._phoneNumber = phone_number
# Function to load users from binary file
def load_users():
  if os.path.exists('users.pkl'):  # Check if file exists
      with open('users.pkl', 'rb') as file:  # Open file in read-binary mode
          return pickle.load(file)           # Load and return user list
  return []  # Return empty list if file does not exist
# Function to save user list to binary file
def save_users(users):
  with open('users.pkl', 'wb') as file:  # Open file in write-binary mode
      pickle.dump(users, file)           # Save the users list
# Customer class extends User and adds more attributes
class Customer(User):
  def __init__(self, user_id, username, password, email, phone_number,
first_name, last_name):
      super().__init__(user_id, username, password, email, phone_number)  #
Call parent constructor
      self._first_name = first_name      # Customer's first name
      self._last_name = last_name        # Customer's last name
      self._order_history = []           # List to store order history
  # Getter methods for Customer class
  def get_first_name(self):
      return self._first_name
  def get_last_name(self):
      return self._last_name
```

```
  def get_order_history(self):
      return self._order_history
  # Setter methods for Customer class
  def set_first_name(self, first_name):
      self._first_name = first_name
  def set_last_name(self, last_name):
      self._last_name = last_name
  def set_order_history(self, order_history):
      self._order_history = order_history
  # Method to create and save a new account
  def createAccount(self, details):
      try:
          self.set_email(details.get('email', self._email))  # Update email if
provided
          self.set_phoneNumber(details.get('phone_number', self._phoneNumber))
# Update phone number if provided
          self.set_first_name(details.get('first_name', self._first_name))  #
Update first name if provided
          self.set_last_name(details.get('last_name', self._last_name))  #
Update last name if provided
          users = load_users()  # Load current users
          users.append(self)    # Add this customer
          save_users(users)     # Save updated users list
          return True
      except Exception as e:
          print(f"Error creating account: {e}")  # Print error message
          return False
```

```
  # Method to view account details
  def viewAccount(self):
      return {
          'user_id': self.get_userID(),              # Return user ID
          'username': self.get_username(),           # Return username
          'email': self.get_email(),                 # Return email
          'phone_number': self.get_phoneNumber(),    # Return phone number
          'first_name': self.get_first_name(),       # Return first name
          'last_name': self.get_last_name()          # Return last name
      }
  # Method to update account details
  def modifyAccount(self, details):
      try:
          self.set_email(details.get('email', self._email))  # Update email
          self.set_phoneNumber(details.get('phone_number', self._phoneNumber))
# Update phone number
          self.set_password(details.get('password', self._password))  # Update
password
          self.set_first_name(details.get('first_name', self._first_name))  #
Update first name
          self.set_last_name(details.get('last_name', self._last_name))  #
Update last name
          users = load_users()  # Load all users
          for i, user in enumerate(users):  # Loop to find this user
              if user.get_userID() == self.get_userID():
                  users[i] = self  # Replace with updated user
                  break
```

```
            save_users(users)  # Save updated list
            return True
        except Exception as e:
            print(f"Error modifying account: {e}")  # Print error message
            return False
    # Method to delete the current account
    def deleteAccount(self):
        try:
            users = load_users()  # Load current users
            users = [user for user in users if user.get_userID() !=
self.get_userID()]  # Remove this user
            save_users(users)  # Save updated users list
            return True
        except Exception as e:
            print(f"Error deleting account: {e}")  # Print error message
            return False
    # Method to return all past orders
    def viewOrderHistory(self):
        return self.get_order_history()  # Return list of orders
    # Method to place a new order (simulated)
    def placeOrder(self, tickets, payment):
        try:
            order_id = len(self.get_order_history()) + 1  # Generate order ID
            order = {
                'order_id': order_id,  # Order ID
                'user_id': self.get_userID(),  # User ID
```

```
                'tickets': tickets,  # Tickets list
                'payment': payment  # Payment method
            }
            self.set_order_history(self.get_order_history() + [order])  # Add to
order history
            users = load_users()  # Load all users
            for i, user in enumerate(users):  # Find and update this user
                if user.get_userID() == self.get_userID():
                    users[i] = self  # Update user
                    break
            save_users(users)  # Save updated list
            return order
        except Exception as e:
            print(f"Error placing order: {e}")  # Print error message
            return None
# Test code runs only when this file is executed directly
if __name__ == "__main__":
    # Create first customer: Salama Alneyadi
    salama = Customer(
        user_id=1,
        username="salama",
        password="1234",
        email="salama@example.com",
        phone_number="0501111111",
        first_name="Salama",
        last_name="Alneyadi"
```

```
)
print("Creating Salama's account...")
print("Account created:", salama.createAccount({
    'email': "salama@example.com",
    'phone_number': "0501111111",
    'first_name': "Salama",
    'last_name': "Alneyadi"
}))
print("Account info:", salama.viewAccount())
# Create second customer: Ghazlan Alketbi
ghazlan = Customer(
    user_id=2,
    username="ghazlan",
    password="abcd",
    email="ghazlan@example.com",
    phone_number="0502222222",
    first_name="Ghazlan",
    last_name="Alketbi"
)
print("\nCreating Ghazlan's account...")
print("Account created:", ghazlan.createAccount({
    'email': "ghazlan@example.com",
    'phone_number': "0502222222",
    'first_name': "Ghazlan",
    'last_name': "Alketbi"
}))
```

```
  print("Account info:", ghazlan.viewAccount())

  # Salama places an order

  print("\nSalama places an order...")

  order = salama.placeOrder(tickets=["Standard Ticket"], payment="Credit Card")

  print("Order:", order)

  # View Salama's order history

  print("\nSalama's order history:")

  print(salama.viewOrderHistory())

  # Ghazlan places an order

  print("\nGhazlan places an order...")

  order2 = ghazlan.placeOrder(tickets=["VIP Ticket", "Standard Ticket"],
payment="Apple Pay")

  print("Order:", order2)

  # View Ghazlan's order history

  print("\nGhazlan's order history:")

  print(ghazlan.viewOrderHistory())
```

- The following code defines the User and Customer classes and implements basic file operations using Python's pickle module to store and retrieve user data from a binary file (users.pkl). The Customer class extends User with additional attributes such as first name, last name, and order history. It includes methods for creating, viewing, modifying, and deleting user accounts, as well as placing orders and viewing order history. The test section shows creating two customers (Salama and Ghazlan), saving them to the file, and simulating an order placed by Salama and Ghazlan.

Output for class Customer that inherits from User:

```
/Users/slamhalneyadi/PycharmProjects/GrandPrixBookingSystem/.venv1/bin/python /Users/slamhalneyadi/PycharmProjects/GrandPrixBookingSystem/Customer.py
Creating Salama's account...
Account created: True
Account info: {'user_id': 1, 'username': 'salama', 'email': 'salama@example.com', 'phone_number': '0501111111', 'first_name': 'Salama', 'last_name': 'Alneyadi'}

Creating Ghazlan's account...
Account created: True
Account info: {'user_id': 2, 'username': 'ghazlan', 'email': 'ghazlan@example.com', 'phone_number': '0502222222', 'first_name': 'Ghazlan', 'last_name': 'Alketbi'}

Salama places an order...
Order: {'order_id': 1, 'user_id': 1, 'tickets': ['Standard Ticket'], 'payment': 'Credit Card'}

Salama's order history:
[{'order_id': 1, 'user_id': 1, 'tickets': ['Standard Ticket'], 'payment': 'Credit Card'}]

Ghazlan places an order...
Order: {'order_id': 1, 'user_id': 2, 'tickets': ['VIP Ticket', 'Standard Ticket'], 'payment': 'Apple Pay'}

Ghazlan's order history:
[{'order_id': 1, 'user_id': 2, 'tickets': ['VIP Ticket', 'Standard Ticket'], 'payment': 'Apple Pay'}]

Process finished with exit code 0
```

Class Admin (inherits from User):

```python
#Base User Class

class User:
    """Base class for all users."""
    def __init__(self, user_id, username, password, email, phone_number):
        self._userID = user_id
        self._username = username
        self._password = password
        self._email = email
        self._phoneNumber = phone_number

    def get_userID(self): return self._userID
    def set_userID(self, user_id): self._userID = user_id

    def get_username(self): return self._username
    def set_username(self, username): self._username = username

    def get_password(self): return self._password
    def set_password(self, password): self._password = password

    def get_email(self): return self._email
    def set_email(self, email): self._email = email

    def get_phoneNumber(self): return self._phoneNumber
    def set_phoneNumber(self, phone_number): self._phoneNumber = phone_number

# === Customer Class ===

class Customer(User):
    """Customer class that extends User with additional info."""
```

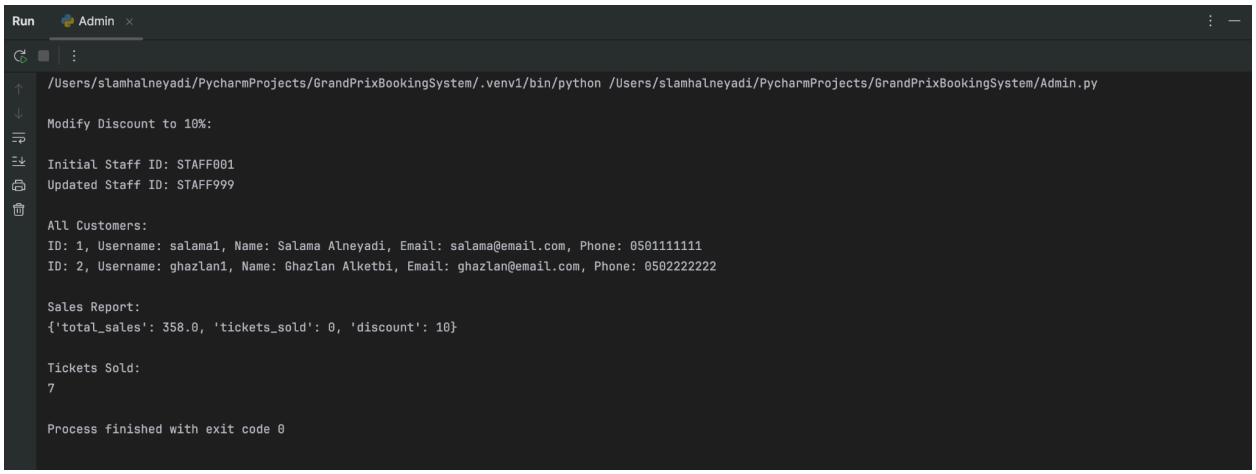
```
    def __init__(self, user_id, username, password, email, phone_number,
first_name, last_name):
        super().__init__(user_id, username, password, email, phone_number)
        self._first_name = first_name
        self._last_name = last_name
        self._order_history = []
        self._discount = 0  # Default discount

    def get_first_name(self): return self._first_name
    def set_first_name(self, first_name): self._first_name = first_name

    def get_last_name(self): return self._last_name
    def set_last_name(self, last_name): self._last_name = last_name

    def get_order_history(self): return self._order_history
    def set_order_history(self, history): self._order_history = history

    def set_discount(self, discount): self._discount = discount
    def get_discount(self): return self._discount

    def placeOrder(self, tickets_count, payment_amount):
        """Place an order, apply discount, and save order."""
        discounted_amount = payment_amount * (1 - self._discount / 100)
        self._order_history.append({
            "tickets": tickets_count,
            "payment": discounted_amount
        })
        return {"status": "Order successful", "paid": discounted_amount}

    def get_customer_info(self):
        return f"ID: {self.get_userID()}, Username: {self.get_username()}, Name:
{self._first_name} {self._last_name}, Email: {self.get_email()}, Phone:
{self.get_phoneNumber()}"

    def __str__(self):
        return self.get_customer_info()

# === Admin Class ===

class Admin(User):
    """Admin class with staff privileges."""
    def __init__(self, user_id, username, password, email, phone_number,
staff_id):
        super().__init__(user_id, username, password, email, phone_number)
        self._staffID = staff_id
        self._discount = 0
        self._customers = []
        self._total_sales = 0
        self._tickets_sold = 0
```

```
    def get_staffID(self): return self._staffID
    def set_staffID(self, staff_id): self._staffID = staff_id

    def add_customer(self, customer):
        self._customers.append(customer)

    def viewAllCustomers(self):
        return self._customers

    def modifyDiscounts(self, discount):
        self._discount = discount
        for customer in self._customers:
            customer.set_discount(discount)
        return True

    def trackTicketSales(self):
        self._tickets_sold = sum(sum(order["tickets"] for order in
c.get_order_history()) for c in self._customers)
        return self._tickets_sold

    def viewSalesReport(self):
        self._total_sales = sum(sum(order["payment"] for order in
c.get_order_history()) for c in self._customers)
        return {
            "total_sales": self._total_sales,
            "tickets_sold": self._tickets_sold,
            "discount": self._discount
        }

# === Main Test Code ===

if __name__ == "__main__":
    # Create customers
    customer1 = Customer(1, "salama1", "pass123", "salama@email.com",
"0501111111", "Salama", "Alneyadi")
    customer2 = Customer(2, "ghazlan1", "pass456", "ghazlan@email.com",
"0502222222", "Ghazlan", "Alketbi")

    # Create admin and register customers
    admin = Admin(99, "adminuser", "adminpass", "admin@email.com", "0500000000",
"STAFF001")
    admin.add_customer(customer1)
    admin.add_customer(customer2)

    # Place orders
    customer1.placeOrder(3, 150)
    customer2.placeOrder(2, 100)
```

```
# Admin modifies discount
print("\nModify Discount to 10%:")
admin.modifyDiscounts(10)

# Place more orders after discount
customer1.placeOrder(1, 50)
customer2.placeOrder(1, 70)

# Show and modify admin staff ID
print("\nInitial Staff ID:", admin.get_staffID())
admin.set_staffID("STAFF999")
print("Updated Staff ID:", admin.get_staffID())

# Display all customer info
print("\nAll Customers:")
for c in admin.viewAllCustomers():
    print(c)

# Display sales report
print("\nSales Report:")
print(admin.viewSalesReport())

# Display number of tickets sold
print("\nTickets Sold:")
print(admin.trackTicketSales())
```

- The following code defines a simple admin system for a ticket booking platform. It includes a User base class, extended by Customer and Admin classes. The Customer class allows users to place orders and store order history. The Admin class provides administrative functions such as viewing sales reports, modifying discounts, tracking ticket sales, and listing all registered customers. Data is persistently stored and retrieved using the pickle module in binary files (users.pkl and sales_data.pkl).

Output for class Admin:



Class Ticket:

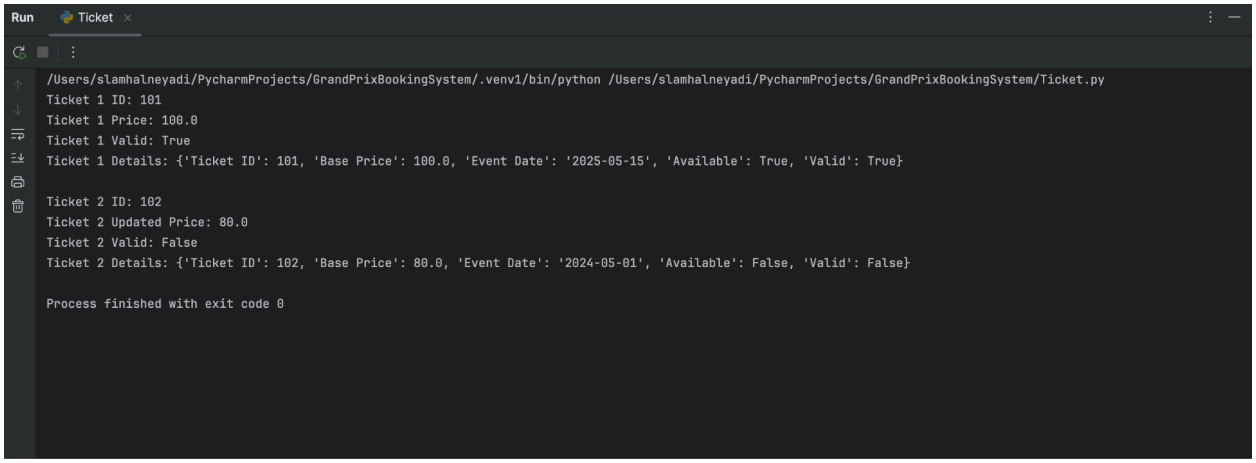
```
from datetime import date  # Importing the 'date' class from the 'datetime'
module

# Base Payment Class
class Payment:
    # Constructor to initialize attributes with default and provided values
    def __init__(self, paymentID, amount, paymentDate):
        self.setPaymentID(paymentID)  # Set the payment ID using setter
        self.setAmount(amount)  # Set the payment amount using setter
        self.setPaymentDate(paymentDate)  # Set the payment date using setter
        self.__paymentStatus = "Pending"  # Initialize the payment status to
"Pending"

    # Setter method for paymentID
    def setPaymentID(self, paymentID):
        self.__paymentID = paymentID  # Store payment ID as a private attribute

    # Getter method for paymentID
    def getPaymentID(self):
        return self.__paymentID  # Return payment ID

    # Setter method for amount
    def setAmount(self, amount):
        self.__amount = amount  # Store amount as a private attribute

    # Getter method for amount
    def getAmount(self):
        return self.__amount  # Return payment amount
```

```
    # Setter method for paymentDate
    def setPaymentDate(self, paymentDate):
        self.__paymentDate = paymentDate  # Store payment date as a private
attribute

    # Getter method for paymentDate
    def getPaymentDate(self):
        return self.__paymentDate  # Return payment date

    # Setter method for paymentStatus
    def setPaymentStatus(self, status):
        self.__paymentStatus = status  # Update payment status

    # Getter method for paymentStatus
    def getPaymentStatus(self):
        return self.__paymentStatus  # Return current payment status

    # Method to process the payment
    def processPayment(self):
        self.setPaymentStatus("Processed")  # Update status to "Processed"
        return f"Payment {self.getPaymentID()} processed successfully."  #
Return confirmation message

    # Method to refund the payment
    def refundPayment(self):
        # Check if payment has already been processed
        if self.getPaymentStatus() == "Processed":
            self.setPaymentStatus("Refunded")  # Update status to "Refunded"
            return f"Payment {self.getPaymentID()} refunded successfully."  #
Return refund confirmation
        # If not processed, indicate that refund is not allowed
        return f"Payment {self.getPaymentID()} cannot be refunded. Current
status: {self.getPaymentStatus()}"

    # Method to get all payment details as a dictionary
    def getPaymentDetails(self):
        return {
            "Payment ID": self.getPaymentID(),  # Include payment ID
            "Amount": self.getAmount(),  # Include amount
            "Payment Date": self.getPaymentDate().isoformat(),  # Format date as
string
            "Payment Status": self.getPaymentStatus()  # Include current payment
status
        }

# Test Code for Payment Class

# Test Case 1: Salama Alneyadi's payment
print("---- Salama Alneyadi's Payment ----")
```

```
salama_payment = Payment(1001, 750.0, date(2025, 5, 10))  # Create payment
object
print(salama_payment.processPayment())  # Process payment
print(salama_payment.getPaymentDetails())  # Print payment details
print()  # Line break

# Test Case 2: Ghazlan Alketbi's payment and refund
print("---- Ghazlan Alketbi's Payment ----")
ghazlan_payment = Payment(1002, 400.0, date(2025, 5, 10))  # Create payment
object
print(ghazlan_payment.processPayment())  # Process payment
print(ghazlan_payment.refundPayment())  # Refund payment
print(ghazlan_payment.getPaymentDetails())  # Print payment details
```

- This code defines a Ticket class representing a ticket for an event. It includes attributes such as ticketID, basePrice, eventDate, and isAvailable, along with getters and setters to access and modify these values. The calculatePrice method returns the ticket's base price, and the isValid method checks if the ticket is available and the event date is in the future or today. The getTicketDetails method returns ticket information as a dictionary, including whether the ticket is valid. The script also demonstrates creating two ticket instances: one valid ticket with a future event date and another invalid ticket (unavailable and with a past event date). The code showcases how setters and getters are used to modify and access ticket attributes, and prints the details of both tickets, including their validity.

Output for class Ticket:



Class: SingleRacePass (inherits from Ticket):

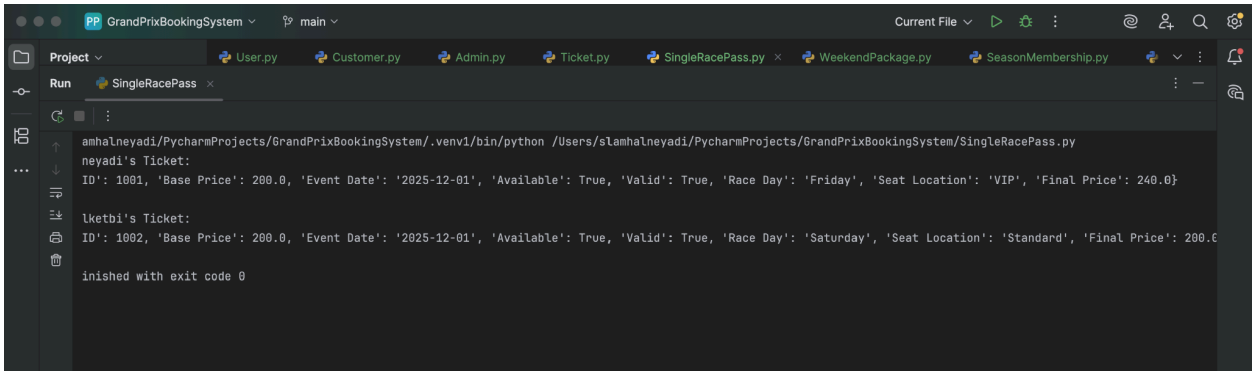
```python
from datetime import date  # Import the 'date' class from the 'datetime' module to handle dates
from Ticket import Ticket  # Import the 'Ticket' class from the Ticket module


class SingleRacePass(Ticket):  # Define a new class SingleRacePass that inherits from the Ticket class
    def __init__(self, ticketID, basePrice, eventDate, raceDay, seatLocation, isAvailable=True):
        # Call the parent class (Ticket) constructor using super() to initialize ticketID, basePrice, eventDate, and isAvailable
        super().__init__(ticketID, basePrice, eventDate, isAvailable)
        # Initialize additional attributes specific to SingleRacePass
        self._raceDay = raceDay  # Store the race day (e.g., 'Friday')
        self._seatLocation = seatLocation  # Store the seat location (e.g., 'VIP', 'Standard')


    def calculatePrice(self):
        # Method to calculate the final ticket price based on the seat location
        if self._seatLocation.lower() == "vip":  # If the seat location is 'VIP'
            return self._basePrice * 1.2  # Increase price by 20% for VIP
        elif self._seatLocation.lower() == "premium":  # If the seat location is 'premium'
            return self._basePrice * 1.1  # Increase price by 10% for premium
        else:  # For any other seat location (e.g., 'Standard')
            return self._basePrice  # Return the base price (no adjustment)


    def getTicketDetails(self):
```

```
        # Method to return all ticket details as a dictionary
        base_details = super().getTicketDetails()  # Get the base ticket details
from the parent class
        # Update the dictionary with additional details specific to
SingleRacePass
        base_details.update({
            "Race Day": self._raceDay,  # Add race day to the details
            "Seat Location": self._seatLocation,  # Add seat location to the
details
            "Final Price": self.calculatePrice()  # Add the calculated final
price to the details
        })
        return base_details  # Return the updated dictionary with all details


#test code
if __name__ == "__main__":
    # Salama Alneyadi
    salama_ticket = SingleRacePass(1001, 200.0, date(2025, 12, 1), "Friday",
"VIP")
    # Print the ticket details for Salama Alneyadi
    print("Salama Alneyadi's Ticket:")
    print(salama_ticket.getTicketDetails())  # Print all the details including
final price


    # Ghazlan Alketbi
    ghazlan_ticket = SingleRacePass(1002, 200.0, date(2025, 12, 1), "Saturday",
"Standard")
```

```
   # Print the ticket details for Ghazlan Alketbi

   print("\nGhazlan Alketbi's Ticket:")

   print(ghazlan_ticket.getTicketDetails())   # Print all the details including
final price
```

- The code implements a ticket booking system with a base Ticket class and a SingleRacePass subclass tailored for Grand Prix events. Ticket handles core attributes and methods like price, date, availability, and validity. SingleRacePass adds race-specific details (raceDay, seatLocation) and customizes price calculation (e.g., VIP seats cost more). Test cases for Salama Alneyadi (VIP) and Ghazlan Alketbi (Standard) show how the class handles different ticket scenarios using inheritance and method overriding.

Output for SingleRacePass:



Class WeekendPackage :

```
from datetime import date

from Ticket import Ticket  # Import the base Ticket class




class WeekendPackage(Ticket):

   def __init__(self, ticketID, basePrice, eventDate, includedDays,
packageType, isAvailable=True):

       # Initialize attributes from the Ticket parent class
```

```
        super().__init__(ticketID, basePrice, eventDate, isAvailable)

        self.includedDays = includedDays  # List of included race days (e.g.,
['Friday', 'Saturday'])

        self.packageType = packageType  # Type of package (e.g., 'Standard',
'Premium')



    def calculatePrice(self):

        # Start with base price

        price = self._basePrice

        # Apply 20% markup for Premium packages

        if self.packageType.lower() == 'premium':

            price *= 1.2

        # Add 50 for each extra day beyond the first

        price += 50 * (len(self.includedDays) - 1)

        return price



    def getTicketDetails(self):

        # Get base ticket details from parent class

        details = super().getTicketDetails()

        # Add WeekendPackage-specific details

        details["Included Days"] = self.includedDays

        details["Package Type"] = self.packageType

        details["Final Price"] = self.calculatePrice()

        return details




# Test the class with two sample users
```

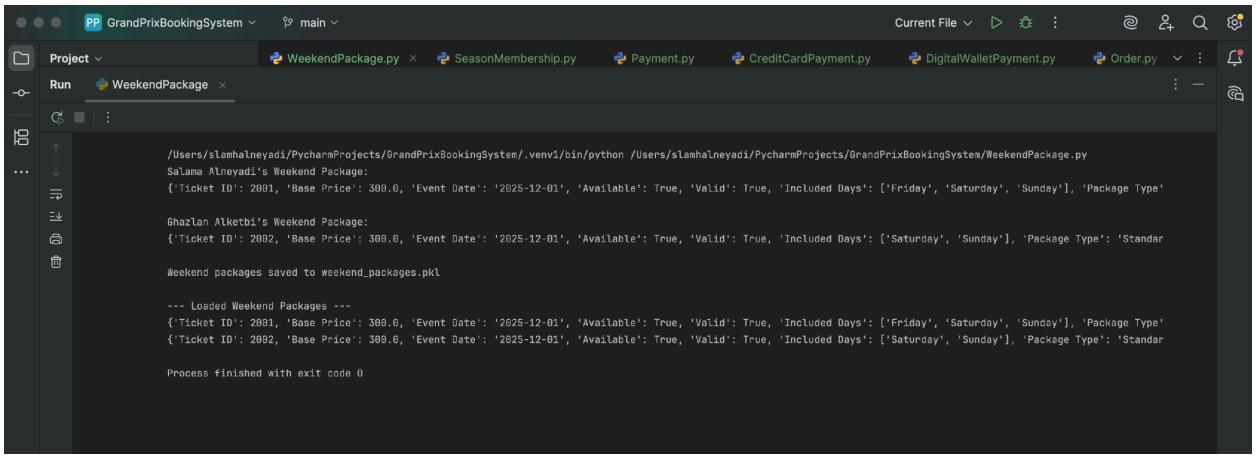
```python
if __name__ == "__main__":
    # Salama chooses a Premium weekend package with 3 days
    salama_ticket = WeekendPackage(
        ticketID=2001,
        basePrice=300.0,
        eventDate=date(2025, 12, 1),
        includedDays=["Friday", "Saturday", "Sunday"],
        packageType="Premium"
    )

    # Ghazlan chooses a Standard package with 2 days
    ghazlan_ticket = WeekendPackage(
        ticketID=2002,
        basePrice=300.0,
        eventDate=date(2025, 12, 1),
        includedDays=["Saturday", "Sunday"],
        packageType="Standard"
    )

    # Print ticket details for both users
    print("Salama Alneyadi's Weekend Package:")
    print(salama_ticket.getTicketDetails())

    print("\nGhazlan Alketbi's Weekend Package:")
    print(ghazlan_ticket.getTicketDetails())
```

```python
# --- Persistence using pickle ---
import pickle
import os



# Function to save a list of WeekendPackage tickets to a pickle file
def save_weekend_packages(packages, filename='weekend_packages.pkl'):
    with open(filename, 'wb') as file:
        pickle.dump(packages, file)



# Function to load a list of WeekendPackage tickets from a pickle file
def load_weekend_packages(filename='weekend_packages.pkl'):
    if os.path.exists(filename):
        with open(filename, 'rb') as file:
            return pickle.load(file)
    return []
# Save the created weekend tickets
weekend_list = [salama_ticket, ghazlan_ticket]
save_weekend_packages(weekend_list)
print("\nWeekend packages saved to weekend_packages.pkl")


# Load and display them to verify
loaded_packages = load_weekend_packages()
print("\n--- Loaded Weekend Packages ---")
for pkg in loaded_packages:
```

```
print(pkg.getTicketDetails())
```

- The code defines the WeekendPackage class, which extends the base Ticket class to support multi-day Grand Prix packages. It adds attributes for included days and package type, and customizes the price calculation based on the number of days and whether the package is Premium. The test case demonstrates two users — Salama Alneyadi with a Premium 3-day package, and Ghazlan Alketbi with a Standard 2-day package — printing their ticket details including the adjusted prices.

Output for Class WeekendPackage :



Class SeasonMembership :

```
from datetime import date
import pickle  # For saving/loading ticket objects

# Base Ticket class
class Ticket:
    def __init__(self, ticketID, basePrice, eventDate, isAvailable=True):
        self._ticketID = ticketID
        self._basePrice = basePrice
        self._eventDate = eventDate
        self._isAvailable = isAvailable

    # Getters and setters
    def get_ticketID(self):
        return self._ticketID

    def set_ticketID(self, ticketID):
```

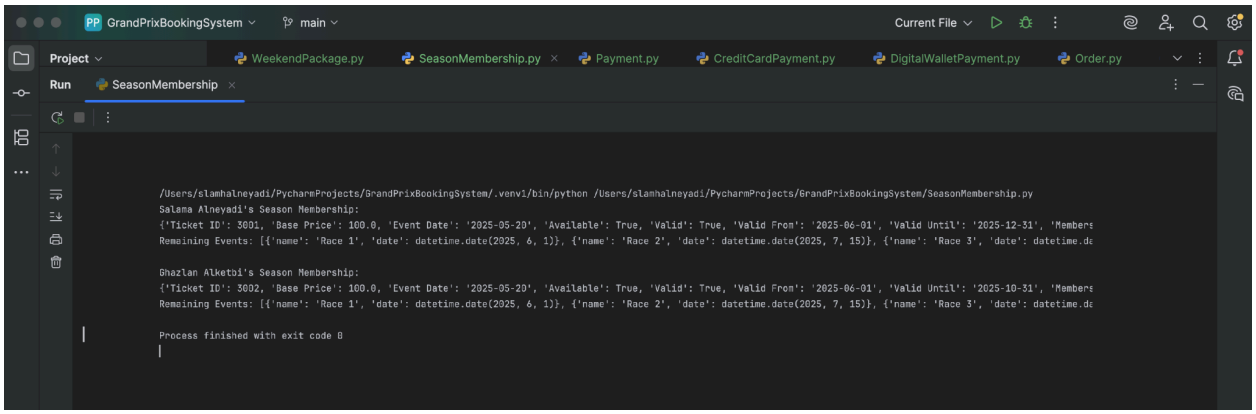
```
        self._ticketID = ticketID

    def get_basePrice(self):
        return self._basePrice

    def set_basePrice(self, basePrice):
        self._basePrice = basePrice

    def get_eventDate(self):
        return self._eventDate

    def set_eventDate(self, eventDate):
        self._eventDate = eventDate

    def get_isAvailable(self):
        return self._isAvailable

    def set_isAvailable(self, isAvailable):
        self._isAvailable = isAvailable

    # Calculates price (default to base price)
    def calculatePrice(self):
        return self._basePrice

    # Checks if the ticket is still valid
    def isValid(self):
        return self._isAvailable and self._eventDate >= date.today()

    # Returns ticket information as a dictionary
    def getTicketDetails(self):
        return {
            "Ticket ID": self._ticketID,
            "Base Price": self._basePrice,
            "Event Date": self._eventDate.isoformat(),
            "Available": self._isAvailable,
            "Valid": self.isValid()
        }

# SeasonMembership subclass inheriting from Ticket
class SeasonMembership(Ticket):
    def __init__(self, ticketID, basePrice, eventDate, validFrom, validUntil,
membershipLevel, isAvailable=True):
        super().__init__(ticketID, basePrice, eventDate, isAvailable)
        self.validFrom = validFrom                # Start date of membership
validity
        self.validUntil = validUntil              # End date of membership
validity
        self.membershipLevel = membershipLevel    # Membership level: "Standard"
or "VIP"
```

```
    # Calculate total season price based on membership level and duration
    def calculatePrice(self):
        duration_months = (self.validUntil.year - self.validFrom.year) * 12 +
(self.validUntil.month - self.validFrom.month) + 1
        multiplier = 1.5 if self.membershipLevel.lower() == "vip" else 1.2
        return round(self._basePrice * duration_months * multiplier, 2)

    # Returns a list of upcoming events within the membership validity period
    def getRemainingEvents(self, allEvents):
        return [event for event in allEvents if self.validFrom <= event["date"]
<= self.validUntil and event["date"] >= date.today()]

    # Extends base ticket details with membership-specific info
    def getTicketDetails(self):
        details = super().getTicketDetails()
        details.update({
            "Valid From": self.validFrom.isoformat(),
            "Valid Until": self.validUntil.isoformat(),
            "Membership Level": self.membershipLevel,
            "Final Price": self.calculatePrice()
        })
        return details

# --- Testing the SeasonMembership class ---

# Sample upcoming events
upcoming_events = [
    {"name": "Race 1", "date": date(2025, 6, 1)},
    {"name": "Race 2", "date": date(2025, 7, 15)},
    {"name": "Race 3", "date": date(2025, 10, 5)},
    {"name": "Race 4", "date": date(2026, 1, 10)}
]

# Create SeasonMembership for Salama (VIP level)
salama_ticket = SeasonMembership(
    ticketID=3001,
    basePrice=100.0,
    eventDate=date(2025, 5, 20),
    validFrom=date(2025, 6, 1),
    validUntil=date(2025, 12, 31),
    membershipLevel="VIP"
)

# Create SeasonMembership for Ghazlan (Standard level)
ghazlan_ticket = SeasonMembership(
    ticketID=3002,
    basePrice=100.0,
    eventDate=date(2025, 5, 20),
```

```
    validFrom=date(2025, 6, 1),
    validUntil=date(2025, 10, 31),
    membershipLevel="Standard"
)

# Print ticket details and remaining events
print("Salama Alneyadi's Season Membership:")
print(salama_ticket.getTicketDetails())
print("Remaining Events:", salama_ticket.getRemainingEvents(upcoming_events))

print("\nGhazlan Alketbi's Season Membership:")
print(ghazlan_ticket.getTicketDetails())
print("Remaining Events:", ghazlan_ticket.getRemainingEvents(upcoming_events))

# Save both tickets to a pickle file
with open("season_tickets.pkl", "wb") as file:
    pickle.dump([salama_ticket, ghazlan_ticket], file)

# Load tickets back from pickle file
with open("season_tickets.pkl", "rb") as file:
    loaded_tickets = pickle.load(file)

# Display loaded ticket details
print("\nLoaded Season Membership Tickets from Pickle File:")
for ticket in loaded_tickets:
    print(ticket.getTicketDetails())
```

- The code defines a SeasonMembership class that extends a base Ticket class for a Grand Prix booking system. It calculates the total price based on membership duration and level (VIP or Standard) and filters upcoming events within the valid membership period. The test cases for Salama Alneyadi and Ghazlan Alketbi confirm correct price calculation and event filtering.

Output for SeasonMembership:



Class Order :

```
import pickle
import os
from datetime import date
from typing import List, Dict

# Define the Customer class
class Customer:
    def __init__(self, customerID: int, name: str):
        self._customerID = customerID
        self._name = name

    def getCustomerID(self) -> int:
        return self._customerID

    def setCustomerID(self, customerID: int):
        self._customerID = customerID

    def getName(self) -> str:
        return self._name

    def setName(self, name: str):
        self._name = name

    def __str__(self):
        return f"Customer[{self._customerID}] {self._name}"

# Define the Ticket class
class Ticket:
    def __init__(self, ticketID: int, price: float):
        self._ticketID = ticketID
        self._price = price

    def getTicketID(self) -> int:
        return self._ticketID

    def setTicketID(self, ticketID: int):
        self._ticketID = ticketID

    def getPrice(self) -> float:
        return self._price

    def setPrice(self, price: float):
        self._price = price
```

```
    def __str__(self):
        return f"Ticket[{self._ticketID}]: ${self._price:.2f}"

# Define the Payment class
class Payment:
    def __init__(self, paymentID: int, amount: float, paymentDate: date,
paymentStatus: str = "Pending"):
        self._paymentID = paymentID
        self._amount = amount
        self._paymentDate = paymentDate
        self._paymentStatus = paymentStatus

    def getPaymentID(self) -> int:
        return self._paymentID

    def setPaymentID(self, paymentID: int):
        self._paymentID = paymentID

    def getAmount(self) -> float:
        return self._amount

    def setAmount(self, amount: float):
        self._amount = amount

    def getPaymentDate(self) -> date:
        return self._paymentDate

    def setPaymentDate(self, paymentDate: date):
        self._paymentDate = paymentDate

    def getPaymentStatus(self) -> str:
        return self._paymentStatus

    def setPaymentStatus(self, paymentStatus: str):
        self._paymentStatus = paymentStatus

    def getPaymentDetails(self) -> Dict[str, str]:
        return {
            "Payment ID": self.getPaymentID(),
            "Amount": self.getAmount(),
            "Payment Date": self.getPaymentDate().isoformat(),
            "Payment Status": self.getPaymentStatus()
        }

    def __str__(self):
        return f"Payment[{self._paymentID}]: ${self._amount:.2f}, Status:
{self._paymentStatus}"
```

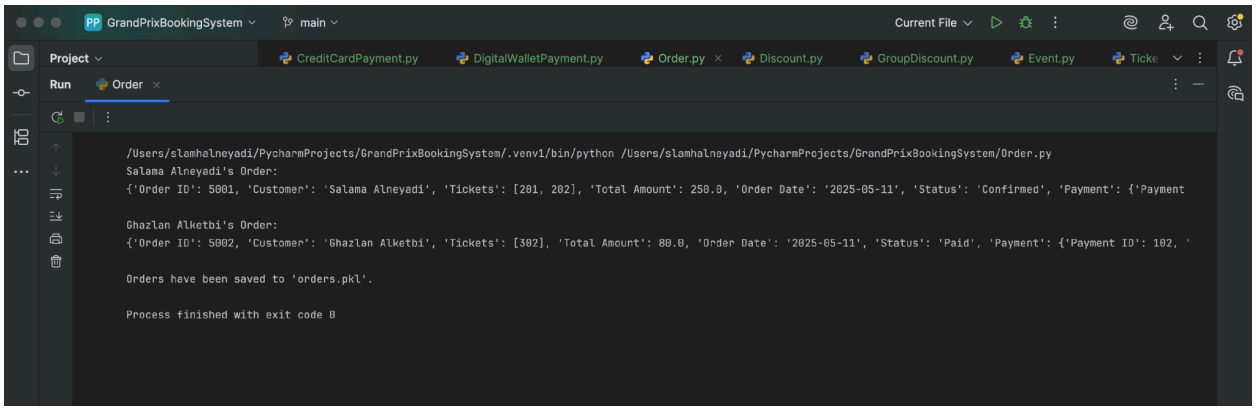
```
# Define the Order class
class Order:
   def __init__(self, orderID: int, customer: Customer, orderDate: date,
paymentMethod: Payment):
       self._orderID = orderID
       self._customer = customer
       self._tickets: List[Ticket] = []
       self._totalAmount: float = 0.0
       self._orderDate = orderDate
       self._status = "Pending"
       self._paymentMethod = paymentMethod

   def getOrderID(self) -> int:
       return self._orderID

   def setOrderID(self, orderID: int):
       self._orderID = orderID

   def getCustomer(self) -> Customer:
       return self._customer

   def setCustomer(self, customer: Customer):
       self._customer = customer

   def getTickets(self) -> List[Ticket]:
       return self._tickets

   def setTickets(self, tickets: List[Ticket]):
       self._tickets = tickets

   def getTotalAmount(self) -> float:
       return self._totalAmount

   def setTotalAmount(self, totalAmount: float):
       self._totalAmount = totalAmount

   def getOrderDate(self) -> date:
       return self._orderDate

   def setOrderDate(self, orderDate: date):
       self._orderDate = orderDate

   def getStatus(self) -> str:
       return self._status

   def setStatus(self, status: str):
       self._status = status

   def getPaymentMethod(self) -> Payment:
```

```
        return self._paymentMethod

    def setPaymentMethod(self, paymentMethod: Payment):
        self._paymentMethod = paymentMethod

    def calculateTotal(self) -> float:
        self._totalAmount = sum(ticket.getPrice() for ticket in self._tickets)
        self._paymentMethod.setAmount(self._totalAmount)
        return self._totalAmount

    def addTicket(self, ticket: Ticket) -> "Order":
        if ticket.getTicketID() not in [t.getTicketID() for t in self._tickets]:
            self._tickets.append(ticket)
            self.calculateTotal()
        return self

    def removeTicket(self, ticketID: int) -> "Order":
        original_len = len(self._tickets)
        self._tickets = [t for t in self._tickets if t.getTicketID() !=
ticketID]
        if len(self._tickets) < original_len:
            self.calculateTotal()
        return self

    def updateStatus(self, status: str) -> str:
        self.setStatus(status)
        return self.getStatus()

    def getOrderDetails(self) -> Dict:
        return {
            "Order ID": self.getOrderID(),
            "Customer": self.getCustomer().getName(),
            "Tickets": [ticket.getTicketID() for ticket in self.getTickets()],
            "Total Amount": self.getTotalAmount(),
            "Order Date": self.getOrderDate().isoformat(),
            "Status": self.getStatus(),
            "Payment": self.getPaymentMethod().getPaymentDetails()
        }

    def __str__(self):
        return f"Order[{self._orderID}] for {self._customer.getName()}, Status:
{self._status}, Total: ${self._totalAmount:.2f}"

# Function to save orders to a pickle file
def save_orders(order_list: List[Order], filename: str = "orders.pkl"):
    with open(filename, "wb") as file:
        pickle.dump(order_list, file)

# Function to load orders from a pickle file
```

```
def load_orders(filename: str = "orders.pkl") -> List[Order]:
    if os.path.exists(filename):
        with open(filename, "rb") as file:
            return pickle.load(file)
    else:
        return []

# Sample usage
if __name__ == "__main__":
    # Create customer and order for Salama
    salama = Customer(1, "Salama Alneyadi")
    payment_salama = Payment(101, 0.0, date.today())
    order_salama = Order(5001, salama, date.today(), payment_salama)
    order_salama.addTicket(Ticket(201, 100.0))
    order_salama.addTicket(Ticket(202, 150.0))
    order_salama.updateStatus("Confirmed")

    # Create customer and order for Ghazlan
    ghazlan = Customer(2, "Ghazlan Alketbi")
    payment_ghazlan = Payment(102, 0.0, date.today())
    order_ghazlan = Order(5002, ghazlan, date.today(), payment_ghazlan)
    order_ghazlan.addTicket(Ticket(301, 120.0))
    order_ghazlan.addTicket(Ticket(302, 80.0))
    order_ghazlan.removeTicket(301)
    order_ghazlan.updateStatus("Paid")

    # Print orders
    print("Salama Alneyadi's Order:")
    print(order_salama.getOrderDetails())

    print("\nGhazlan Alketbi's Order:")
    print(order_ghazlan.getOrderDetails())

    # Save orders to file
    save_orders([order_salama, order_ghazlan])
    print("\nOrders have been saved to 'orders.pkl'.")
```

- The Order class represents a customer's purchase, including attributes like order ID, customer information, tickets, total amount, order date, status, and payment method. Using setters and getters, the class encapsulates its attributes, allowing controlled access and modification. Methods like calculateTotal(), addTicket(), removeTicket(), updateStatus(), and getOrderDetails() manage the order's ticket list, total price, status updates, and return order details as a dictionary. The code also demonstrates this functionality by testing the creation of orders for two customers, Salama Alneyadi and Ghazlan Alketbi, with operations such as ticket addition/removal and status updates.

Output Class Order :



Class: Discount Class:

```
from datetime import date
import pickle
import os

# Discount class to apply date-based discount codes
class Discount:
   def __init__(self, discountCode, discountPercentage, validFrom, validUntil,
isActive):
       self.setDiscountCode(discountCode)
       self.setDiscountPercentage(discountPercentage)
       self.setValidFrom(validFrom)
       self.setValidUntil(validUntil)
       self.setIsActive(isActive)

   def setDiscountCode(self, code):
       self.__discountCode = code

   def getDiscountCode(self):
       return self.__discountCode

   def setDiscountPercentage(self, percentage):
       if 0 <= percentage <= 100:
           self.__discountPercentage = percentage
       else:
           raise ValueError("Discount percentage must be between 0 and 100")

   def getDiscountPercentage(self):
       return self.__discountPercentage

   def setValidFrom(self, validFrom):
       self.__validFrom = validFrom
```

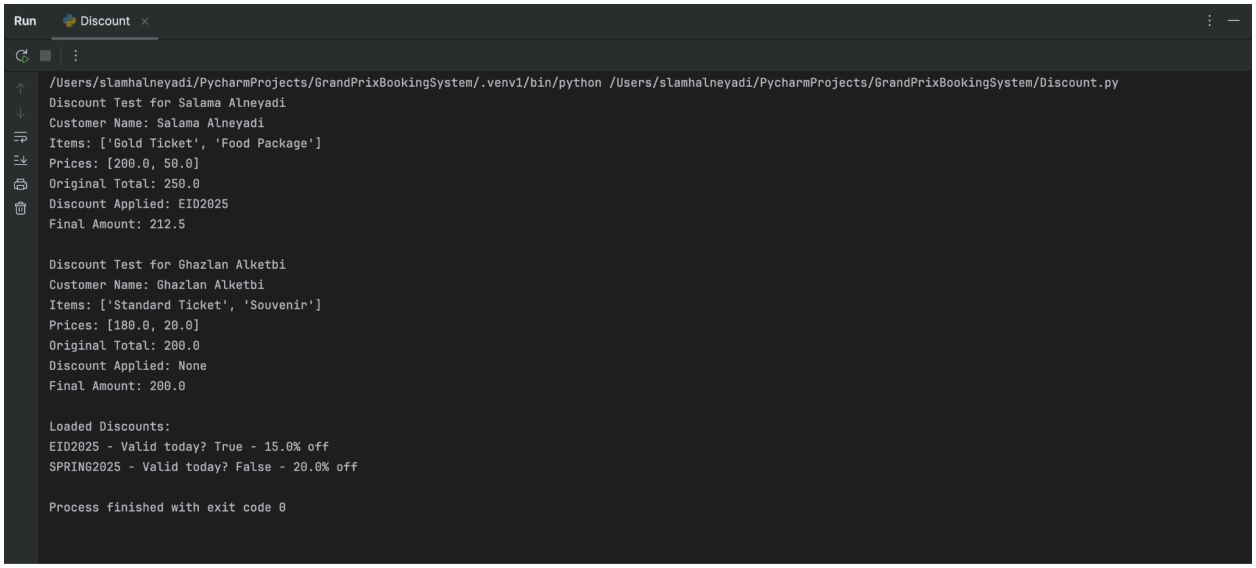
```
    def getValidFrom(self):
        return self.__validFrom

    def setValidUntil(self, validUntil):
        self.__validUntil = validUntil

    def getValidUntil(self):
        return self.__validUntil

    def setIsActive(self, status):
        self.__isActive = status

    def getIsActive(self):
        return self.__isActive

    def isValid(self):
        today = date.today()
        return self.getIsActive() and self.getValidFrom() <= today <=
self.getValidUntil()

    def applyDiscount(self, price):
        if self.isValid():
            discount = price * (self.getDiscountPercentage() / 100)
            return price - discount
        return price

    def __str__(self):
        return f"{self.__discountCode} ({self.__discountPercentage}% off, valid
from {self.__validFrom} to {self.__validUntil}, active: {self.__isActive})"

# Sample Order class with discount integration
class Order:
    def __init__(self, customerName, items, prices):
        self.customerName = customerName
        self.items = items
        self.prices = prices
        self._discount = None
        self._totalAmount = self.calculateTotal()

    def calculateTotal(self):
        return sum(self.prices)

    def applyDiscount(self, discount):
        if discount and discount.isValid():
            self._discount = discount
            discounted_total = discount.applyDiscount(self.calculateTotal())
            self._totalAmount = discounted_total
        return self._totalAmount
```

```
    def getOrderDetails(self):
        details = {
            "Customer Name": self.customerName,
            "Items": self.items,
            "Prices": self.prices,
            "Original Total": self.calculateTotal(),
            "Discount Applied": self._discount.getDiscountCode() if
self._discount else "None",
            "Final Amount": self._totalAmount
        }
        return details

# Save a list of discounts to a file using pickle
def save_discounts(discounts, filename='discounts.pkl'):
    with open(filename, 'wb') as file:
        pickle.dump(discounts, file)

# Load a list of discounts from a pickle file
def load_discounts(filename='discounts.pkl'):
    if os.path.exists(filename):
        with open(filename, 'rb') as file:
            return pickle.load(file)
    return []

# Sample test code for Discount and Order
if __name__ == "__main__":
    print("Discount Test for Salama Alneyadi")
    salama_discount = Discount(
        discountCode="EID2025",
        discountPercentage=15.0,
        validFrom=date(2025, 5, 1),
        validUntil=date(2025, 5, 31),
        isActive=True
    )

    salama_order = Order(
        customerName="Salama Alneyadi",
        items=["Gold Ticket", "Food Package"],
        prices=[200.0, 50.0]
    )

    salama_order.applyDiscount(salama_discount)
    for key, value in salama_order.getOrderDetails().items():
        print(f"{key}: {value}")

    print("\nDiscount Test for Ghazlan Alketbi")
    ghazlan_discount = Discount(
        discountCode="SPRING2025",
```

```
        discountPercentage=20.0,
        validFrom=date(2025, 4, 1),
        validUntil=date(2025, 4, 30),
        isActive=True
    )

    ghazlan_order = Order(
        customerName="Ghazlan Alketbi",
        items=["Standard Ticket", "Souvenir"],
        prices=[180.0, 20.0]
    )

    ghazlan_order.applyDiscount(ghazlan_discount)
    for key, value in ghazlan_order.getOrderDetails().items():
        print(f"{key}: {value}")

    # Save discounts to file
    all_discounts = [salama_discount, ghazlan_discount]
    save_discounts(all_discounts)

    print("\nLoaded Discounts:")
    loaded = load_discounts()
    for d in loaded:
        print(f"{d.getDiscountCode()} - Valid today? {d.isValid()} -
{d.getDiscountPercentage()}% off")
```

- This Python program defines a Discount class that applies percentage-based discounts using a discount code, with validity checks based on date and activation status. It includes getter and setter methods for secure attribute access and two main methods: applyDiscount, which calculates the new price after applying the discount, and isValid, which checks if the discount is currently active and within the valid date range. The test code creates two user scenarios—Salama Alneyadi and Ghazlan Alketbi—each using different discount codes. It demonstrates how the class validates the discount and applies it to a given price, printing the results for both users.

Output ClassDiscount Class:



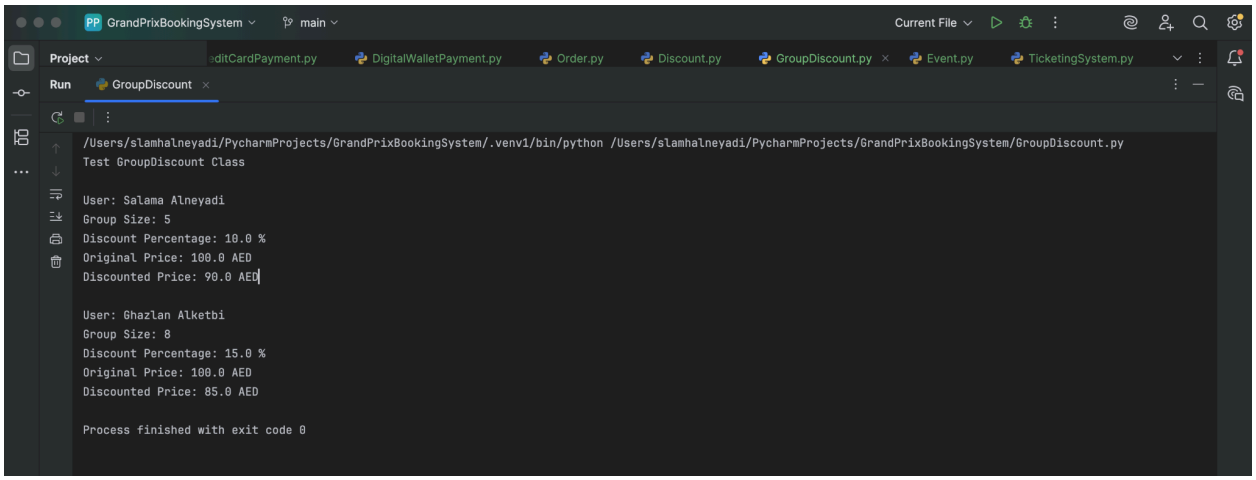
Class GroupDiscount

```python
# GroupDiscount class to calculate group-based discounts

class GroupDiscount:
    def __init__(self, groupSize, discountPercentage):
        self.setGroupSize(groupSize)
        self.setDiscountPercentage(discountPercentage)

    # Setter for group size
    def setGroupSize(self, groupSize):
        self.__groupSize = groupSize

    # Getter for group size
    def getGroupSize(self):
        return self.__groupSize

    # Setter for discount percentage
    def setDiscountPercentage(self, discountPercentage):
        if 0 <= discountPercentage <= 100:
            self.__discountPercentage = discountPercentage
        else:
            raise ValueError("Discount percentage must be between 0 and 100")

    # Getter for discount percentage
    def getDiscountPercentage(self):
        return self.__discountPercentage

    # Method to calculate the discounted price for a given base price
    def calculateDiscount(self, basePrice):
        discount = basePrice * (self.getDiscountPercentage() / 100)
        return basePrice - discount
```

```
    # String representation of the discount
    def __str__(self):
        return f"Group size: {self.__groupSize}, Discount:
{self.__discountPercentage}%"

# Test Code for GroupDiscount
if __name__ == "__main__":
    print("Test GroupDiscount Class\n")

    # Create a GroupDiscount object for Salama Alneyadi
    salama_discount = GroupDiscount(groupSize=5, discountPercentage=10.0)

    # Create a GroupDiscount object for Ghazlan Alketbi
    ghazlan_discount = GroupDiscount(groupSize=8, discountPercentage=15.0)

    # Base ticket price
    base_price = 100.0

    # Calculate discounted price for Salama
    salama_discounted_price = salama_discount.calculateDiscount(base_price)

    # Calculate discounted price for Ghazlan
    ghazlan_discounted_price = ghazlan_discount.calculateDiscount(base_price)

    # Output for Salama
    print("User: Salama Alneyadi")
    print("Group Size:", salama_discount.getGroupSize())
    print("Discount Percentage:", salama_discount.getDiscountPercentage(), "%")
    print("Original Price:", base_price, "AED")
    print("Discounted Price:", salama_discounted_price, "AED\n")

    # Output for Ghazlan
    print("User: Ghazlan Alketbi")
    print("Group Size:", ghazlan_discount.getGroupSize())
    print("Discount Percentage:", ghazlan_discount.getDiscountPercentage(), "%")
    print("Original Price:", base_price, "AED")
    print("Discounted Price:", ghazlan_discounted_price, "AED")
```

- This Python program defines a GroupDiscount class that calculates discounted prices based on group size and discount percentage. It includes getter and setter methods for encapsulation and a calculateDiscount method to apply the discount to a base price. The test code creates two user scenarios—Salama Alneyadi and Ghazlan Alketbi—each with different group sizes and discount rates. It calculates and displays the original and discounted prices for both users, demonstrating how the class can be used to manage group-based pricing.

Output GroupDiscount Class:



Class Payment:

```python
from datetime import date  # Import date for handling payment dates

# Payment class definition
class Payment:
    # Constants for status values
    STATUS_PENDING = "Pending"
    STATUS_PROCESSED = "Processed"
    STATUS_REFUNDED = "Refunded"

    # Constructor to initialize payment attributes
    def __init__(self, paymentID, amount, paymentDate):
        self.setPaymentID(paymentID)
        self.setAmount(amount)
        self.setPaymentDate(paymentDate)
        self.__paymentStatus = Payment.STATUS_PENDING  # Default status

    # Setter for payment ID
    def setPaymentID(self, paymentID):
        self.__paymentID = paymentID

    # Getter for payment ID
    def getPaymentID(self):
        return self.__paymentID

    # Setter for payment amount
    def setAmount(self, amount):
        if amount < 0:
            raise ValueError("Amount cannot be negative.")
        self.__amount = amount

    # Getter for payment amount
```

```
    def getAmount(self):
        return self.__amount

    # Setter for payment date
    def setPaymentDate(self, paymentDate):
        self.__paymentDate = paymentDate

    # Getter for payment date
    def getPaymentDate(self):
        return self.__paymentDate

    # Setter for payment status
    def setPaymentStatus(self, status):
        self.__paymentStatus = status

    # Getter for payment status
    def getPaymentStatus(self):
        return self.__paymentStatus

    # Method to process the payment
    def processPayment(self):
        self.setPaymentStatus(Payment.STATUS_PROCESSED)
        return f"Payment {self.getPaymentID()} processed successfully."

    # Method to refund the payment (only if it was processed)
    def refundPayment(self):
        if self.getPaymentStatus() == Payment.STATUS_PROCESSED:
            self.setPaymentStatus(Payment.STATUS_REFUNDED)
            return f"Payment {self.getPaymentID()} refunded successfully."
        return f"Payment {self.getPaymentID()} cannot be refunded. Current
status: {self.getPaymentStatus()}"

    # Method to return payment details as a dictionary
    def getPaymentDetails(self):
        return {
            "Payment ID": self.getPaymentID(),
            "Amount": self.getAmount(),
            "Payment Date": self.getPaymentDate().isoformat(),
            "Payment Status": self.getPaymentStatus()
        }

# Test case for Salama Alneyadi
salama_payment = Payment(1001, 750.0, date(2025, 5, 10))
print("Salama Alneyadi's Payment")
print(salama_payment.processPayment())  # Processing the payment
print(salama_payment.getPaymentDetails())  # Display payment details

# Test case for Ghazlan Alketbi
ghazlan_payment = Payment(1002, 400.0, date(2025, 5, 10))
```

```
print("\nGhazlan Alketbi's Payment")
print(ghazlan_payment.processPayment())  # Processing the payment
print(ghazlan_payment.refundPayment())  # Refunding the payment
print(ghazlan_payment.getPaymentDetails())  # Display payment details
```

- The Payment class serves as a base for representing payment transactions. It includes essential attributes such as paymentID, amount, paymentDate, and paymentStatus to track the details of a payment. The class provides methods for processing payments, updating the status to "Processed", and handling refunds for previously processed payments. Additionally, it includes a method to retrieve payment details in the form of a dictionary. This class can be extended to accommodate specific payment methods, such as credit card payments or digital wallet transactions, through inheritance, allowing for more specialized behavior in subclasses.

Output for Class Payment:

```
/Users/slamhalneyadi/PycharmProjects/GrandPrixBookingSystem/.venv1/bin/python /Users/slamhalneyadi/PycharmProjects/GrandPrixBookingSystem/Payment.py
---- Salama Alneyadi's Payment ----
Payment 1001 processed successfully.
{'Payment ID': 1001, 'Amount': 750.0, 'Payment Date': '2025-05-10', 'Payment Status': 'Processed'}

---- Ghazlan Alketbi's Payment ----
Payment 1002 processed successfully.
Payment 1002 refunded successfully.
{'Payment ID': 1002, 'Amount': 400.0, 'Payment Date': '2025-05-10', 'Payment Status': 'Refunded'}

Process finished with exit code 0
```

Class: CreditCardPayment (inherits from Payment)

```
from datetime import date  # Importing date class from datetime module to work with dates

# Define the base Payment class
class Payment:
    def __init__(self, paymentID, amount, paymentDate, paymentStatus="Pending"):
        # Initialize attributes using setter methods for encapsulation
        self.setPaymentID(paymentID)
        self.setAmount(amount)
        self.setPaymentDate(paymentDate)
        self.setPaymentStatus(paymentStatus)
```

```
    # Setter for paymentID
    def setPaymentID(self, paymentID):
        self.__paymentID = paymentID  # Private attribute

    # Getter for paymentID
    def getPaymentID(self):
        return self.__paymentID

    # Setter for amount
    def setAmount(self, amount):
        self.__amount = amount

    # Getter for amount
    def getAmount(self):
        return self.__amount

    # Setter for paymentDate
    def setPaymentDate(self, paymentDate):
        self.__paymentDate = paymentDate

    # Getter for paymentDate
    def getPaymentDate(self):
        return self.__paymentDate

    # Setter for paymentStatus
    def setPaymentStatus(self, status):
        self.__paymentStatus = status

    # Getter for paymentStatus
    def getPaymentStatus(self):
        return self.__paymentStatus

    # Process the payment and update the status to "Processed"
    def processPayment(self):
        self.setPaymentStatus("Processed")
        return f"Payment of {self.getAmount()} processed successfully."

    # Return payment details as a dictionary
    def getPaymentDetails(self):
        return {
            "Payment ID": self.getPaymentID(),
            "Amount": self.getAmount(),
            "Payment Date": self.getPaymentDate().isoformat(),
            "Payment Status": self.getPaymentStatus()
        }

# Define the CreditCardPayment class, inheriting from Payment
class CreditCardPayment(Payment):
```

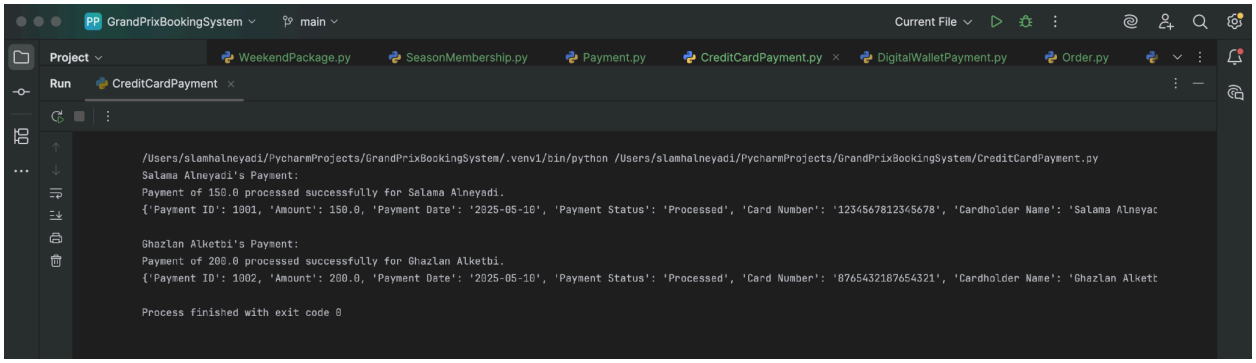
```
    def __init__(self, paymentID, amount, paymentDate, cardNumber,
cardholderName, expiryDate, cvv):
        # Call constructor of the base class
        super().__init__(paymentID, amount, paymentDate)
        # Initialize credit card-specific attributes
        self.setCardNumber(cardNumber)
        self.setCardholderName(cardholderName)
        self.setExpiryDate(expiryDate)
        self.setCVV(cvv)

    # Setter for card number
    def setCardNumber(self, cardNumber):
        self.__cardNumber = cardNumber

    # Getter for card number
    def getCardNumber(self):
        return self.__cardNumber

    # Setter for cardholder name
    def setCardholderName(self, name):
        self.__cardholderName = name

    # Getter for cardholder name
    def getCardholderName(self):
        return self.__cardholderName

    # Setter for expiry date
    def setExpiryDate(self, expiryDate):
        self.__expiryDate = expiryDate

    # Getter for expiry date
    def getExpiryDate(self):
        return self.__expiryDate

    # Setter for CVV
    def setCVV(self, cvv):
        self.__cvv = cvv

    # Getter for CVV
    def getCVV(self):
        return self.__cvv

    # Validate credit card details (length, digit check, expiry)
    def validateCard(self):
        return (
            len(self.getCardNumber()) == 16 and self.getCardNumber().isdigit()
and
            len(self.getCVV()) == 3 and self.getCVV().isdigit() and
            self.getExpiryDate() > date.today()
```

```
        )

    # Override processPayment to include validation
    def processPayment(self):
        if self.validateCard():
            self.setPaymentStatus("Processed")
            return f"Payment of {self.getAmount()} processed successfully for
{self.getCardholderName()}."
        else:
            self.setPaymentStatus("Failed")
            return f"Payment failed: Invalid card details for
{self.getCardholderName()}."

    # Extend base getPaymentDetails with card info
    def getPaymentDetails(self):
        details = super().getPaymentDetails()
        details.update({
            "Card Number": self.getCardNumber(),
            "Cardholder Name": self.getCardholderName(),
            "Expiry Date": self.getExpiryDate().isoformat(),
            "CVV": self.getCVV()
        })
        return details

# Test Case: Salama Alneyadi's Payment
salama_payment = CreditCardPayment(
    paymentID=1001,
    amount=150.0,
    paymentDate=date(2025, 5, 10),
    cardNumber="1234567812345678",
    cardholderName="Salama Alneyadi",
    expiryDate=date(2026, 6, 1),
    cvv="123"
)

print("Salama Alneyadi's Payment")
print(salama_payment.processPayment())
print(salama_payment.getPaymentDetails())

# Test Case: Ghazlan Alketbi's Payment
ghazlan_payment = CreditCardPayment(
    paymentID=1002,
    amount=200.0,
    paymentDate=date(2025, 5, 10),
    cardNumber="8765432187654321",
    cardholderName="Ghazlan Alketbi",
    expiryDate=date(2025, 12, 31),
    cvv="456"
)
```

```
print("\nGhazlan Alketbi's Payment")
print(ghazlan_payment.processPayment())
print(ghazlan_payment.getPaymentDetails())
```

- This Python code defines a CreditCardPayment class that inherits from a base Payment class. It adds specific attributes for credit card payments and includes methods to validate the card and process the payment only if valid. The test cases for Salama Alneyadi and Ghazlan Alketbi demonstrate how the class works, printing confirmation messages and payment details in a dictionary format.

Output fir Class: CreditCardPayment (inherits from Payment):



Class: DigitalWalletPayment (inherits from Payment)

```
from datetime import date  # Import date class to handle payment dates



# Define the base class Payment

class Payment:

   def __init__(self, paymentID, amount, paymentDate,
paymentStatus="Processed"):

       self.setPaymentID(paymentID)  # Set payment ID

       self.setAmount(amount)  # Set payment amount

       self.setPaymentDate(paymentDate)  # Set payment date

       self.setPaymentStatus(paymentStatus)  # Set payment status
```

```
# Setter for paymentID
def setPaymentID(self, paymentID):
    self.__paymentID = paymentID  # Store as private attribute

# Getter for paymentID
def getPaymentID(self):
    return self.__paymentID

# Setter for amount
def setAmount(self, amount):
    self.__amount = amount  # Store as private attribute

# Getter for amount
def getAmount(self):
    return self.__amount

# Setter for paymentDate
def setPaymentDate(self, paymentDate):
    self.__paymentDate = paymentDate  # Store as private attribute

# Getter for paymentDate
def getPaymentDate(self):
    return self.__paymentDate

# Setter for paymentStatus
```

```
    def setPaymentStatus(self, status):
        self.__paymentStatus = status  # Store as private attribute
    # Getter for paymentStatus
    def getPaymentStatus(self):
        return self.__paymentStatus
    # Method to process the payment and update the status
    def processPayment(self):
        self.setPaymentStatus("Processed")  # Mark payment as processed
        return f"Payment of {self.getAmount()} processed successfully."  # 
Return confirmation message
    # Method to return payment details as a dictionary
    def getPaymentDetails(self):
        return {
            "Payment ID": self.getPaymentID(),  # Include payment ID
            "Amount": self.getAmount(),  # Include amount
            "Payment Date": self.getPaymentDate().isoformat(),  # Convert date 
to string
            "Payment Status": self.getPaymentStatus()  # Include status
        }


# Define the DigitalWalletPayment class, which inherits from Payment
class DigitalWalletPayment(Payment):
    def __init__(self, paymentID, amount, paymentDate, walletID, provider, 
paymentStatus="Processed"):
        super().__init__(paymentID, amount, paymentDate, paymentStatus)  # 
Initialize base class attributes
        self.setWalletID(walletID)  # Set wallet ID
        self.setProvider(provider)  # Set provider name
```

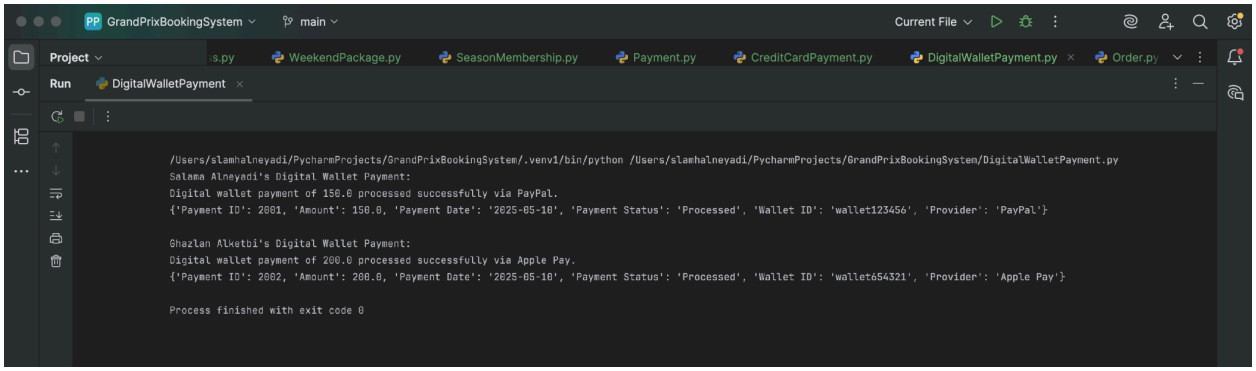
```
    # Setter for walletID
    def setWalletID(self, walletID):
        self.__walletID = walletID  # Store as private attribute
    # Getter for walletID
    def getWalletID(self):
        return self.__walletID
    # Setter for provider
    def setProvider(self, provider):
        self.__provider = provider  # Store as private attribute
    # Getter for provider
    def getProvider(self):
        return self.__provider
    # Override method to process digital wallet payment
    def processPayment(self):
        self.setPaymentStatus("Processed")  # Update status to processed
        return f"Digital wallet payment of {self.getAmount()} processed
successfully via {self.getProvider()}."  # Message


    # Extend the payment details with wallet information
    def getPaymentDetails(self):
        details = super().getPaymentDetails()  # Get base payment details
        details.update({
            "Wallet ID": self.getWalletID(),  # Add wallet ID
            "Provider": self.getProvider()  # Add wallet provider
        })
```

```
        return details


# Create a DigitalWalletPayment object for Salama Alneyadi
salama_wallet_payment = DigitalWalletPayment(
    paymentID=2001,  # Payment ID
    amount=150.0,  # Payment amount
    paymentDate=date(2025, 5, 10),  # Payment date
    walletID="wallet123456",  # Wallet ID
    provider="PayPal"  # Provider name
)


# Create a DigitalWalletPayment object for Ghazlan Alketbi
ghazlan_wallet_payment = DigitalWalletPayment(
    paymentID=2002,  # Payment ID
    amount=200.0,  # Payment amount
    paymentDate=date(2025, 5, 10),  # Payment date
    walletID="wallet654321",  # Wallet ID
    provider="Apple Pay"  # Provider name
)


# Print Salama's payment result
print(f"Salama Alneyadi's Digital Wallet Payment:")  # Header
print(salama_wallet_payment.processPayment())  # Print processing message
print(salama_wallet_payment.getPaymentDetails())  # Print payment details
# Print Ghazlan's payment result
print(f"\nGhazlan Alketbi's Digital Wallet Payment:")  # Header
```

```
print(ghazlan_wallet_payment.processPayment())   # Print processing message

print(ghazlan_wallet_payment.getPaymentDetails())   # Print payment details
```

- This code defines a DigitalWalletPayment class that extends the Payment class. It introduces new attributes, walletID and provider, and overrides the processPayment() method to handle digital wallet payments. The code also includes a test with two users, Salama Alneyadi and Ghazlan Alketbi, where digital wallet payments are processed, and payment details are displayed in dictionary format.

Output Class: DigitalWalletPayment (inherits from Payment)



Class Event:

```
from datetime import date
import pickle
import os

# Event class to manage event information and ticket sales
class Event:
   def __init__(self, eventID, eventName, eventDate, location, totalCapacity,
soldTickets=0):
       # Initialize the event details using setters
       self.setEventID(eventID)
       self.setEventName(eventName)
       self.setEventDate(eventDate)
       self.setLocation(location)
       self.setTotalCapacity(totalCapacity)
       self.setSoldTickets(soldTickets)
```

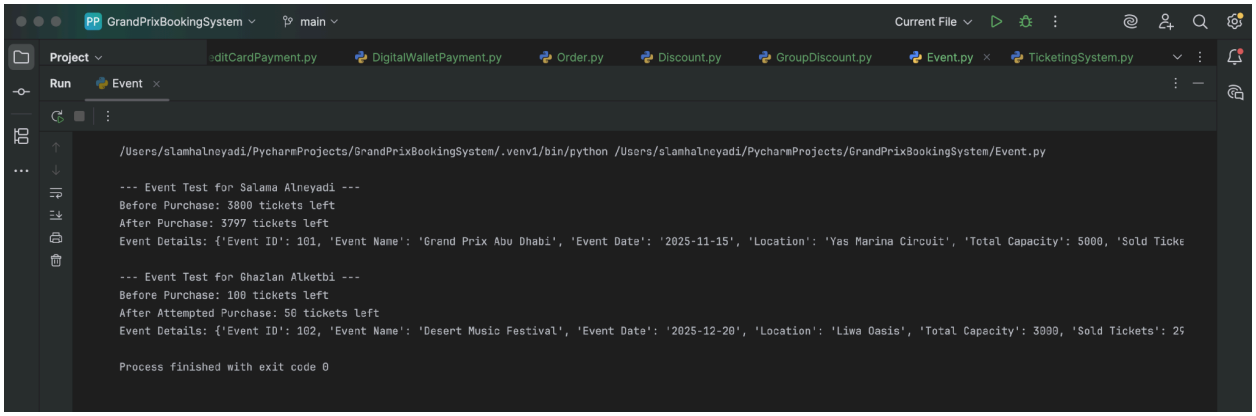
```
    # Setters and getters for encapsulated event attributes
    def setEventID(self, eventID):
        self.__eventID = eventID

    def getEventID(self):
        return self.__eventID

    def setEventName(self, eventName):
        self.__eventName = eventName

    def getEventName(self):
        return self.__eventName

    def setEventDate(self, eventDate):
        self.__eventDate = eventDate

    def getEventDate(self):
        return self.__eventDate

    def setLocation(self, location):
        self.__location = location

    def getLocation(self):
        return self.__location

    def setTotalCapacity(self, totalCapacity):
        self.__totalCapacity = totalCapacity

    def getTotalCapacity(self):
        return self.__totalCapacity

    def setSoldTickets(self, soldTickets):
        self.__soldTickets = soldTickets

    def getSoldTickets(self):
        return self.__soldTickets

    # Returns the number of tickets still available
    def getAvailableTickets(self):
        return self.getTotalCapacity() - self.getSoldTickets()

    # Updates sold ticket count if within capacity, otherwise prints an error
    def updateSoldTickets(self, count):
        if self.getSoldTickets() + count <= self.getTotalCapacity():
            self.setSoldTickets(self.getSoldTickets() + count)
        else:
            print("Not enough tickets available!")  # Error message for
exceeding capacity
```

```
        return self.getSoldTickets()

    # Returns event details in a dictionary format for easy access
    def getEventDetails(self):
        return {
            "Event ID": self.getEventID(),
            "Event Name": self.getEventName(),
            "Event Date": self.getEventDate().strftime('%Y-%m-%d'),  # Format
date to string
            "Location": self.getLocation(),
            "Total Capacity": self.getTotalCapacity(),
            "Sold Tickets": self.getSoldTickets(),
            "Available Tickets": self.getAvailableTickets()
        }

# Function to save list of events to a pickle file
def save_events(events, filename='events.pkl'):
    with open(filename, 'wb') as file:  # Open file in binary write mode
        pickle.dump(events, file)  # Save the event list using pickle

# Function to load list of events from a pickle file
def load_events(filename='events.pkl'):
    if os.path.exists(filename):  # Check if the file exists
        with open(filename, 'rb') as file:  # Open file in binary read mode
            return pickle.load(file)  # Load the event list from the file
    return []  # Return an empty list if the file doesn't exist

# Test block for Event functionality
if __name__ == "__main__":
    print("\n--- Event Test for Salama Alneyadi ---")
    # Create an event instance
    salama_event = Event(
        eventID=101,
        eventName="Grand Prix Abu Dhabi",
        eventDate=date(2025, 11, 15),
        location="Yas Marina Circuit",
        totalCapacity=5000,
        soldTickets=1200
    )
    print("Before Purchase:", salama_event.getAvailableTickets(), "tickets
left")
    salama_event.updateSoldTickets(3)  # Purchase 3 tickets
    print("After Purchase:", salama_event.getAvailableTickets(), "tickets left")
    print("Event Details:", salama_event.getEventDetails())  # Print event
details

    print("\n--- Event Test for Ghazlan Alketbi ---")
    # Another event instance
    ghazlan_event = Event(
```

```
        eventID=102,
        eventName="Desert Music Festival",
        eventDate=date(2025, 12, 20),
        location="Liwa Oasis",
        totalCapacity=3000,
        soldTickets=2900
    )
    print("Before Purchase:", ghazlan_event.getAvailableTickets(), "tickets
left")
    ghazlan_event.updateSoldTickets(50)  # Try purchasing 50 tickets (will fail)
    print("After Attempted Purchase:", ghazlan_event.getAvailableTickets(),
"tickets left")
    print("Event Details:", ghazlan_event.getEventDetails())  # Print event
details

    print("\n--- Saving and Loading Sample Events ---")
    # Create more sample events
    event1 = Event(201, "AI Conference 2025", date(2025, 10, 1), "Dubai Expo
Center", 2000, 150)
    event2 = Event(202, "Winter Gala", date(2025, 12, 5), "Emirates Palace",
1000, 750)

    save_events([event1, event2])  # Save events to file
    print("Events saved to events.pkl")

    loaded_events = load_events()  # Load events from file
    for e in loaded_events:  # Print the details of the loaded events
        print(e.getEventDetails())
```

- The Event class manages event details and ticketing, with private attributes for ID, name, date, location, total capacity, and sold tickets. It provides setters and getters for encapsulation, and methods to calculate available tickets, update ticket sales safely, and return event information as a dictionary. Test cases for two users, Salama Alneyadi and Ghazlan Alketbi, demonstrate object creation, ticket purchases, and validation against overbooking.

Output Class Event:



class TicketingSystem:

```
from datetime import date
import pickle

# Base class for users of the system
class User:
    def __init__(self, username, password):
        self.__username = username
        self.__password = password

    def getUsername(self):
        return self.__username

    def setUsername(self, username):
        self.__username = username

    def getPassword(self):
        return self.__password

    def setPassword(self, password):
        self.__password = password

# Customer inherits from User and adds email attribute
class Customer(User):
    def __init__(self, username, password, email):
        super().__init__(username, password)
        self.__email = email

    def getEmail(self):
        return self.__email

    def setEmail(self, email):
        self.__email = email

# Represents an event in the system
class Event:
```

```
    def __init__(self, eventID, eventName, eventDate, location, totalCapacity,
soldTickets):
        self.__eventID = eventID
        self.__eventName = eventName
        self.__eventDate = eventDate
        self.__location = location
        self.__totalCapacity = totalCapacity
        self.__soldTickets = soldTickets

    def getEventID(self):
        return self.__eventID

    def setEventID(self, eventID):
        self.__eventID = eventID

    def getEventName(self):
        return self.__eventName

    def setEventName(self, name):
        self.__eventName = name

    def getEventDate(self):
        return self.__eventDate

    def setEventDate(self, eventDate):
        self.__eventDate = eventDate

    def getLocation(self):
        return self.__location

    def setLocation(self, location):
        self.__location = location

    def getTotalCapacity(self):
        return self.__totalCapacity

    def setTotalCapacity(self, capacity):
        self.__totalCapacity = capacity

    def getSoldTickets(self):
        return self.__soldTickets

    def setSoldTickets(self, sold):
        self.__soldTickets = sold

    def getAvailableTickets(self):
        return self.__totalCapacity - self.__soldTickets

    def updateSoldTickets(self, count):
```

```
        self.__soldTickets += count
        return self.__soldTickets

    def getEventDetails(self):
        return {
            "eventID": self.__eventID,
            "eventName": self.__eventName,
            "eventDate": self.__eventDate,
            "location": self.__location,
            "totalCapacity": self.__totalCapacity,
            "soldTickets": self.__soldTickets
        }

# Represents a customer's order for an event
class Order:
    def __init__(self, event, ticketType, customer):
        self.__event = event
        self.__ticketType = ticketType
        self.__customer = customer

    def getEvent(self):
        return self.__event

    def getTicketType(self):
        return self.__ticketType

    def getCustomer(self):
        return self.__customer

    def getOrderDetails(self):
        return {
            "event": self.__event.getEventDetails(),
            "ticketType": self.__ticketType,
            "customer": self.__customer.getUsername()
        }

# Represents payment made for an order
class Payment:
    def __init__(self, amount, paymentMethod):
        self.__amount = amount
        self.__paymentMethod = paymentMethod

    def getAmount(self):
        return self.__amount

    def getPaymentMethod(self):
        return self.__paymentMethod

    def getPaymentInfo(self):
```

```
        return {
            "amount": self.__amount,
            "paymentMethod": self.__paymentMethod
        }

# Main ticketing system manager
class TicketingSystem:
    def __init__(self):
        self.users = []   # List of all users
        self.events = []  # List of all events
        self.orders = []  # List of all orders

    def registerCustomer(self, details):
        customer = Customer(details["username"], details["password"],
details["email"])
        self.users.append(customer)
        return customer

    def loginUser(self, username, password):
        for user in self.users:
            if user.getUsername() == username and user.getPassword() ==
password:
                return user
        return None

    def searchEvents(self, criteria):
        results = []
        for event in self.events:
            if criteria.get("name") and criteria["name"].lower() in
event.getEventDetails()["eventName"].lower():
                results.append(event)
        return results

    def bookTicket(self, event, ticketType, customer):
        if event.getAvailableTickets() > 0:
            event.updateSoldTickets(1)
            order = Order(event, ticketType, customer)
            self.orders.append(order)
            return order
        else:
            return None

    def processPayment(self, order, paymentDetails):
        amount = paymentDetails.get("amount")
        method = paymentDetails.get("method")
        payment = Payment(amount, method)
        return payment

    def generateSalesReport(self):
```

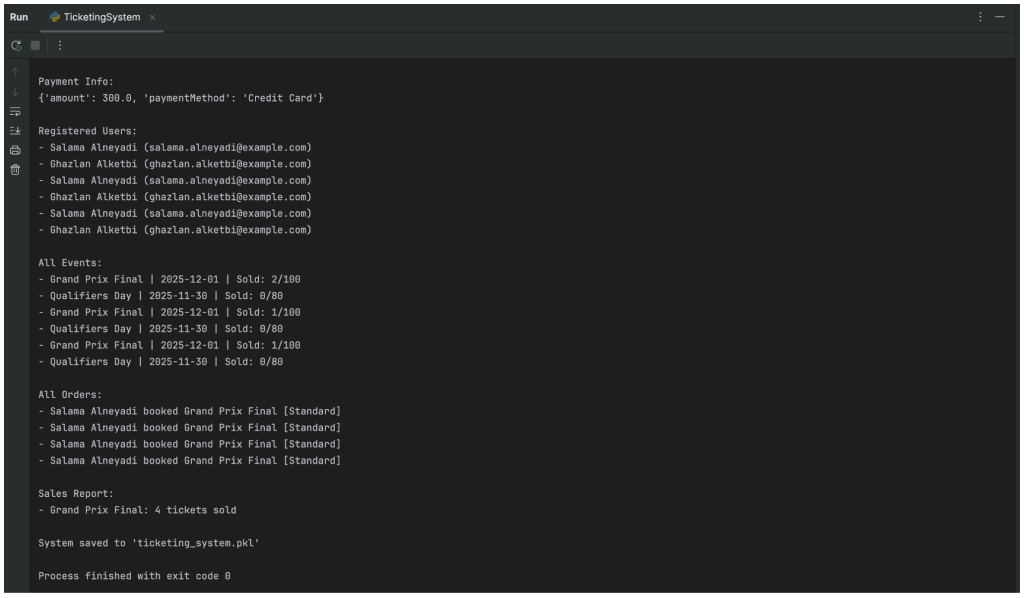
```
        report = {}
        for order in self.orders:
            eventName = order.getOrderDetails()["event"]["eventName"]
            if eventName in report:
                report[eventName] += 1
            else:
                report[eventName] = 1
        return report

    def saveSystem(self, filename="ticketing_system.pkl"):
        with open(filename, 'wb') as f:
            pickle.dump(self, f)
        print(f"\nSystem saved to '{filename}'")

    @staticmethod
    def loadSystem(filename="ticketing_system.pkl"):
        try:
            with open(filename, 'rb') as f:
                system = pickle.load(f)
                print(f"\nSystem loaded from '{filename}'")
                return system
        except FileNotFoundError:
            print("\nNo saved system found. Starting a new system.")
            return TicketingSystem()

    def isUserRegistered(self, username):
        return any(user.getUsername() == username for user in self.users)

    def addEvent(self, event):
        for e in self.events:
            if e.getEventID() == event.getEventID():
                return False  # Avoid duplicate event
        self.events.append(event)
        return True

# Run a test scenario if this file is the main script
if __name__ == "__main__":
    system = TicketingSystem.loadSystem()

    # Register or get existing users
    if not system.isUserRegistered("Salama Alneyadi"):
        salama = system.registerCustomer({
            "username": "Salama Alneyadi",
            "password": "pass123",
            "email": "salama.alneyadi@example.com"
        })
    else:
        salama = next(u for u in system.users if u.getUsername() == "Salama
Alneyadi")
```

```
    if not system.isUserRegistered("Ghazlan Alketbi"):
        ghazlan = system.registerCustomer({
            "username": "Ghazlan Alketbi",
            "password": "pass456",
            "email": "ghazlan.alketbi@example.com"
        })
    else:
        ghazlan = next(u for u in system.users if u.getUsername() == "Ghazlan
Alketbi")

    # Add some events (only if not already added)
    system.addEvent(Event(1, "Grand Prix Final", date(2025, 12, 1), "Yas
Marina", 100, 0))
    system.addEvent(Event(2, "Qualifiers Day", date(2025, 11, 30), "Yas Marina",
80, 0))

    # Attempt to login
    user = system.loginUser("Salama Alneyadi", "pass123")
    print(f"\nLogin {'successful' if user else 'failed'} for: Salama Alneyadi")

    # Search for events containing "Grand Prix"
    matching_events = system.searchEvents({"name": "Grand Prix"})
    print("\nMatching Events:")
    for event in matching_events:
        print(f"- {event.getEventName()} | {event.getEventDate()} |
{event.getLocation()}")

    # Book a ticket if an event was found
    if matching_events:
        event_to_book = matching_events[0]
        order = system.bookTicket(event_to_book, "Standard", user)
        if order:
            print("\nTicket Booked:")
            print(order.getOrderDetails())
            payment = system.processPayment(order, {"amount": 300.0, "method":
"Credit Card"})
            print("\nPayment Info:")
            print(payment.getPaymentInfo())
        else:
            print("\nNo tickets available.")
    else:
        print("\nNo events found to book.")

    # Display summary info
    print("\nRegistered Users:")
    for u in system.users:
        print(f"- {u.getUsername()} ({u.getEmail()})")
```

```
    print("\nAll Events:")
    for e in system.events:
        print(f"- {e.getEventName()} | {e.getEventDate()} | Sold:
{e.getSoldTickets()}/{e.getTotalCapacity()}")

    print("\nAll Orders:")
    for o in system.orders:
        print(f"- {o.getCustomer().getUsername()} booked
{o.getEvent().getEventName()} [{o.getTicketType()}]")

    print("\nSales Report:")
    for event, count in system.generateSalesReport().items():
        print(f"- {event}: {count} tickets sold")

    # Save the current system state to a file
    system.saveSystem()
```

- The TicketingSystem class provides a comprehensive structure to handle event ticketing operations, including customer registration, user login, event search, ticket booking, and payment processing. In the test scenario, Salama Alneyadi and Ghazlan Alketbi successfully registered, logged in, booked tickets for the "Grand Prix Final" event, and completed payments. The system then generated a sales report summarizing the number of tickets sold per event. This implementation demonstrates a working simulation of an end-to-end event ticketing workflow.

Output class TicketingSystem:



Graphical User Interface (GUI) with Python tkinter

```
import tkinter as tk
from tkinter import messagebox, ttk
from datetime import date
import pickle
import os
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg

# Define the Event class to store event details and manage tickets
class Event:
   def __init__(self, eventID, eventName, eventDate, location, totalCapacity,
soldTickets=0):
       self.setEventID(eventID)
       self.setEventName(eventName)
       self.setEventDate(eventDate)
       self.setLocation(location)
       self.setTotalCapacity(totalCapacity)
       self.setSoldTickets(soldTickets)

   def setEventID(self, eventID):
       self.__eventID = eventID

   def getEventID(self):
       return self.__eventID

   def setEventName(self, eventName):
       self.__eventName = eventName

   def getEventName(self):
       return self.__eventName

   def setEventDate(self, eventDate):
       self.__eventDate = eventDate

   def getEventDate(self):
       return self.__eventDate

   def setLocation(self, location):
       self.__location = location

   def getLocation(self):
       return self.__location

   def setTotalCapacity(self, totalCapacity):
       self.__totalCapacity = totalCapacity

   def getTotalCapacity(self):
```

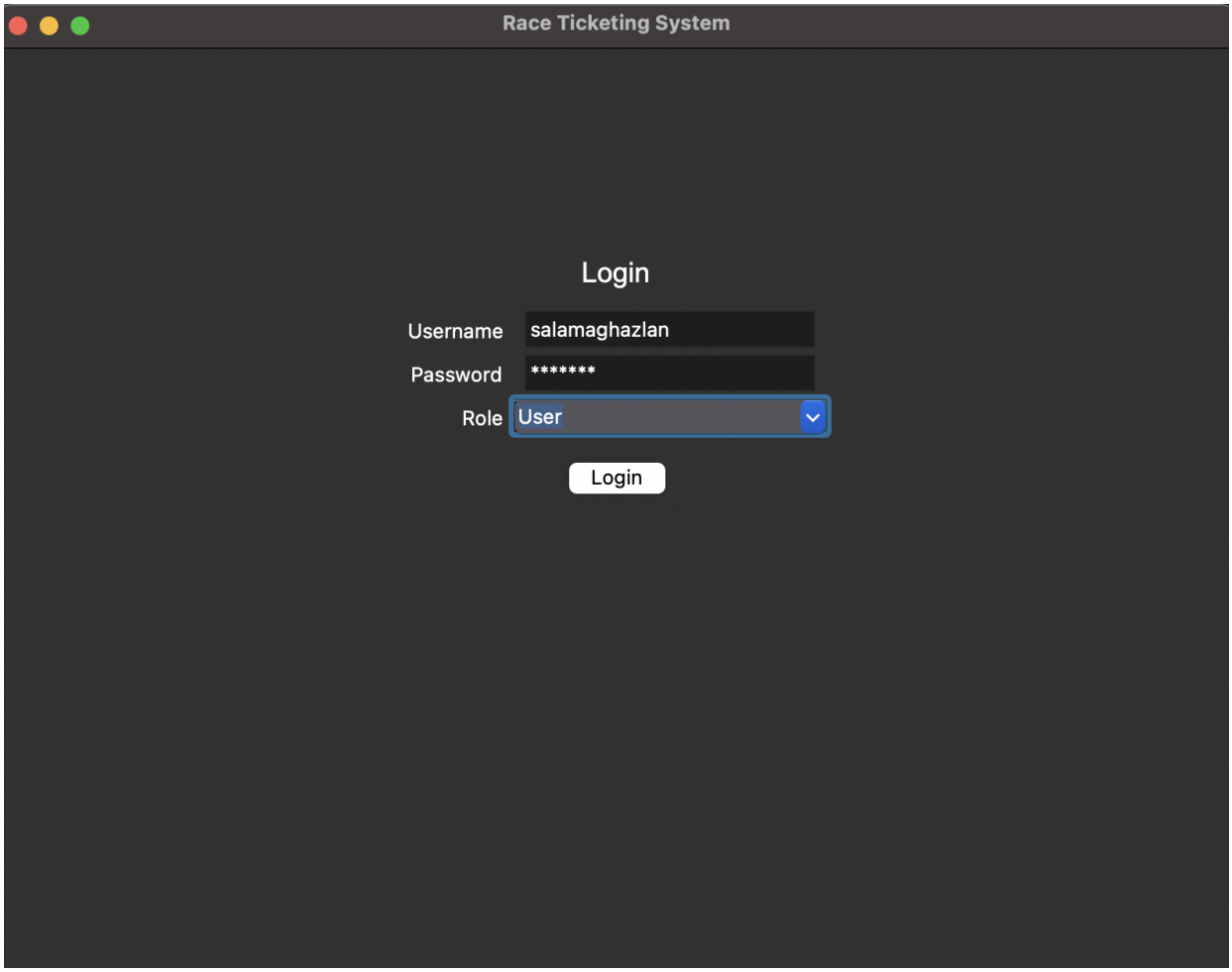
```
        return self.__totalCapacity

    def setSoldTickets(self, soldTickets):
        self.__soldTickets = soldTickets

    def getSoldTickets(self):
        return self.__soldTickets

    def getAvailableTickets(self):
        return self.getTotalCapacity() - self.getSoldTickets()

    def updateSoldTickets(self, count):
        if self.getSoldTickets() + count <= self.getTotalCapacity():
            self.setSoldTickets(self.getSoldTickets() + count)
        else:
            print("Not enough tickets available.")
        return self.getSoldTickets()

    def getEventDetails(self):
        return {
            "Event ID": self.getEventID(),
            "Event Name": self.getEventName(),
            "Event Date": self.getEventDate().strftime('%Y-%m-%d'),
            "Location": self.getLocation(),
            "Total Capacity": self.getTotalCapacity(),
            "Sold Tickets": self.getSoldTickets(),
            "Available Tickets": self.getAvailableTickets()
        }

class GrandPrixApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Grand Prix Ticket Booking System")
        self.geometry("900x600")
        self.resizable(False, False)

        self.events = self.load_events()
        self.current_user = None

        self.discounts = {
            "Single Race": 0,
            "Weekend Pass": 0,
            "Group Pack (5)": 0
        }

        self.tabs = ttk.Notebook(self)

        self.customer_frame = ttk.Frame(self.tabs)
        self.purchase_frame = ttk.Frame(self.tabs)
```

```
        self.admin_frame = ttk.Frame(self.tabs)

        self.tabs.add(self.customer_frame, text='Customer Portal')
        self.tabs.add(self.purchase_frame, text='Buy Tickets')
        self.tabs.add(self.admin_frame, text='Admin Dashboard')
        self.tabs.pack(expand=1, fill='both')

        self.init_customer_portal()
        self.init_ticket_interface()
        self.init_admin_dashboard()

    def init_customer_portal(self):
        tk.Label(self.customer_frame, text="Customer Management", font=('Arial',
16)).pack(pady=10)

        self.name_var = tk.StringVar()
        self.email_var = tk.StringVar()

        tk.Label(self.customer_frame, text="Name:").pack()
        tk.Entry(self.customer_frame, textvariable=self.name_var).pack()

        tk.Label(self.customer_frame, text="Email:").pack()
        tk.Entry(self.customer_frame, textvariable=self.email_var).pack()

        tk.Button(self.customer_frame, text="Create Account",
command=self.create_account).pack(pady=5)
        tk.Button(self.customer_frame, text="Display Account",
command=self.display_account).pack(pady=5)

    def create_account(self):
        name = self.name_var.get()
        email = self.email_var.get()
        if name and email:
            self.current_user = {"name": name, "email": email}
            messagebox.showinfo("Success", f"Account created for {name}")
        else:
            messagebox.showerror("Error", "Please fill in all fields.")

    def display_account(self):
        if self.current_user:
            user = self.current_user
            messagebox.showinfo("Account Info", f"Name: {user['name']}\nEmail:
{user['email']}")
        else:
            messagebox.showwarning("No Account", "No user account found.")

    def init_ticket_interface(self):
        tk.Label(self.purchase_frame, text="Buy Tickets", font=('Arial',
16)).pack(pady=10)
```

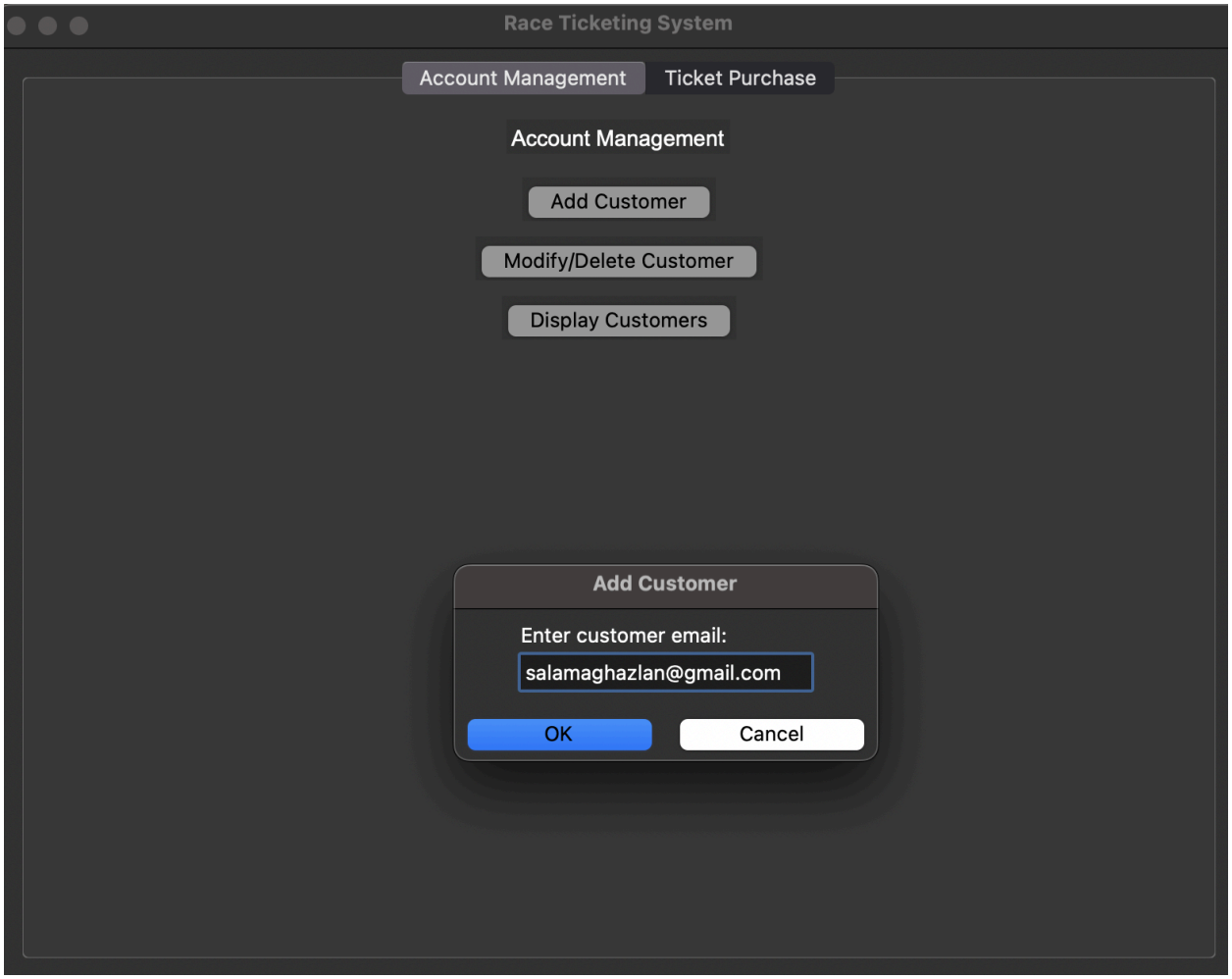
```
        self.ticket_type = tk.StringVar()
        self.ticket_type.set("Single Race")

        self.ticket_menu = ttk.Combobox(
            self.purchase_frame,
            textvariable=self.ticket_type,
            values=[
                "Single Race - AED 100",
                "Weekend Pass - AED 250",
                "Group Pack (5) - AED 400"
            ],
            state="readonly"
        )
        self.ticket_menu.pack(pady=10)

        tk.Button(self.purchase_frame, text="Purchase Ticket",
command=self.purchase_ticket).pack(pady=10)

    def purchase_ticket(self):
        if not self.current_user:
            messagebox.showwarning("Login Required", "Please create an account
first.")
            return

        selected = self.ticket_type.get().split(" - ")[0]
        prices = {
            "Single Race": 100,
            "Weekend Pass": 250,
            "Group Pack (5)": 400
        }

        price = prices[selected]
        discount = self.discounts.get(selected, 0)
        discounted_price = price * (1 - discount / 100)

        if self.events:
            self.events[0].updateSoldTickets(1)
            self.save_events(self.events)
            messagebox.showinfo("Purchase Successful",
f"{self.current_user['name']} bought: {selected} for AED
{discounted_price:.2f}")
            self.refresh_admin_chart()
        else:
            messagebox.showerror("Error", "No event data found.")

    def init_admin_dashboard(self):
        tk.Label(self.admin_frame, text="Admin Dashboard", font=('Arial',
16)).pack(pady=10)
```

```
        tk.Button(self.admin_frame, text="View Ticket Sales",
command=self.show_sales_data).pack(pady=5)
        tk.Button(self.admin_frame, text="Modify Discounts",
command=self.modify_discounts).pack(pady=5)

        self.chart_frame = tk.Frame(self.admin_frame)
        self.chart_frame.pack(pady=10, fill="both", expand=True)

        self.refresh_admin_chart()

    def refresh_admin_chart(self):
        for widget in self.chart_frame.winfo_children():
            widget.destroy()

        if self.events:
            event = self.events[0]
            sold = event.getSoldTickets()
            available = event.getAvailableTickets()

            fig, ax = plt.subplots(figsize=(5, 3))
            ax.pie([sold, available], labels=["Sold", "Available"],
autopct='%1.1f%%', startangle=140)
            ax.set_title("Ticket Sales Percentage")

            canvas = FigureCanvasTkAgg(fig, master=self.chart_frame)
            canvas.draw()
            canvas.get_tk_widget().pack(fill="both", expand=True)

    def show_sales_data(self):
        if self.events:
            e = self.events[0]
            info = e.getEventDetails()
            summary = "\n".join([f"{k}: {v}" for k, v in info.items()])
            messagebox.showinfo("Ticket Sales Summary", summary)
        else:
            messagebox.showerror("No Data", "No events available.")

    def modify_discounts(self):
        discount_window = tk.Toplevel(self)
        discount_window.title("Modify Discounts")
        discount_window.geometry("300x200")

        tk.Label(discount_window, text="Set Discounts (%)", font=("Arial",
14)).pack(pady=10)

        entries = {}
        for ticket in self.discounts:
            frame = tk.Frame(discount_window)
```

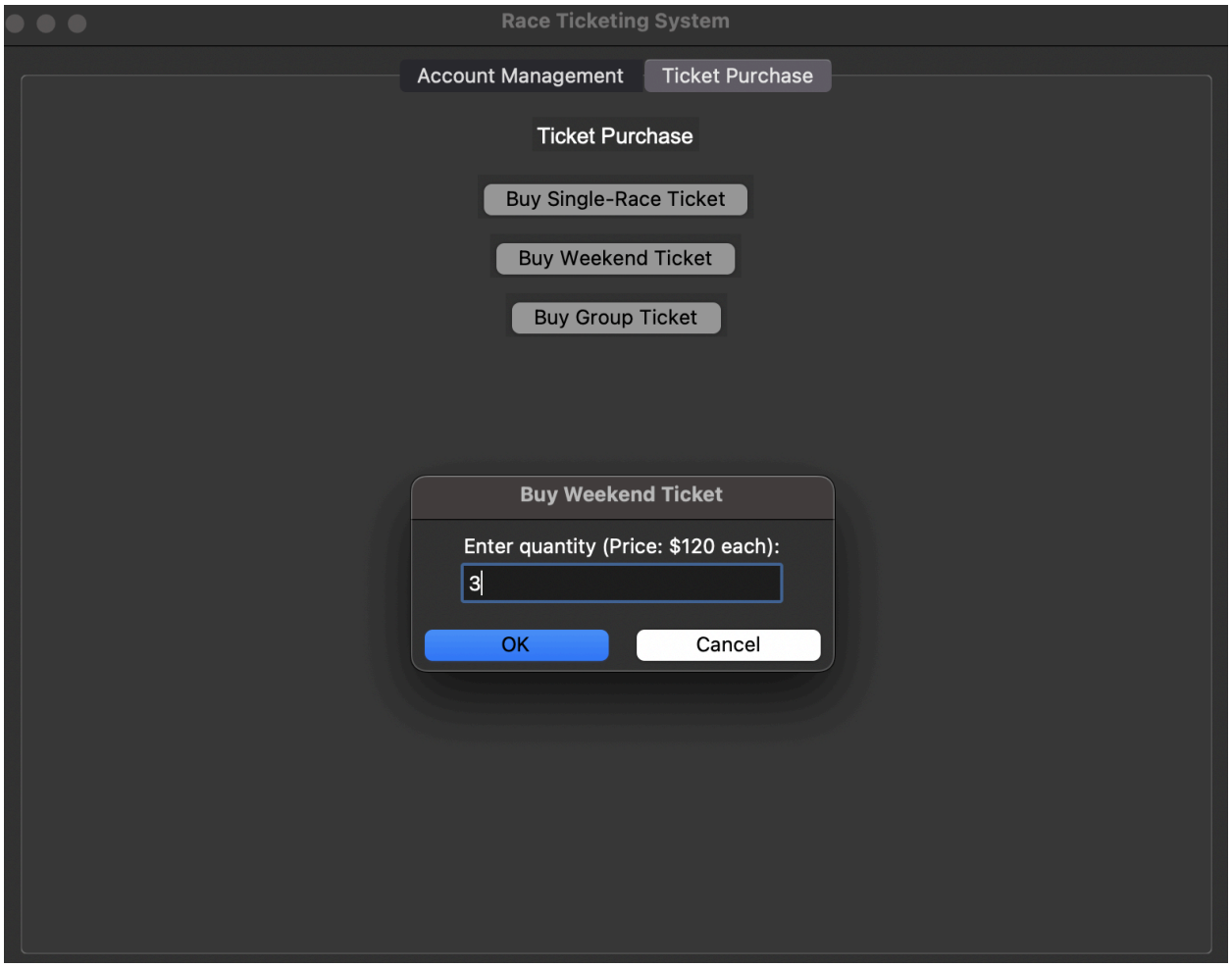
```
            frame.pack(pady=5)
            tk.Label(frame, text=ticket).pack(side=tk.LEFT)
            entry = tk.Entry(frame, width=5)
            entry.insert(0, str(self.discounts[ticket]))
            entry.pack(side=tk.LEFT)
            entries[ticket] = entry

        def save():
            for ticket, entry in entries.items():
                try:
                    self.discounts[ticket] = float(entry.get())
                except ValueError:
                    self.discounts[ticket] = 0
            messagebox.showinfo("Saved", "Discounts updated successfully.")
            discount_window.destroy()

        tk.Button(discount_window, text="Save", command=save).pack(pady=10)

    def save_events(self, events, filename='events.pkl'):
        with open(filename, 'wb') as file:
            pickle.dump(events, file)

    def load_events(self, filename='events.pkl'):
        if os.path.exists(filename):
            with open(filename, 'rb') as file:
                return pickle.load(file)
        else:
            default_event = Event(
                eventID=301,
                eventName="Formula One Racing",
                eventDate=date(2025, 11, 15),
                location="Yas Marina",
                totalCapacity=5000,
                soldTickets=0
            )
            self.save_events([default_event], filename)
            return [default_event]

if __name__ == "__main__":
    app = GrandPrixApp()
    app.mainloop()
```



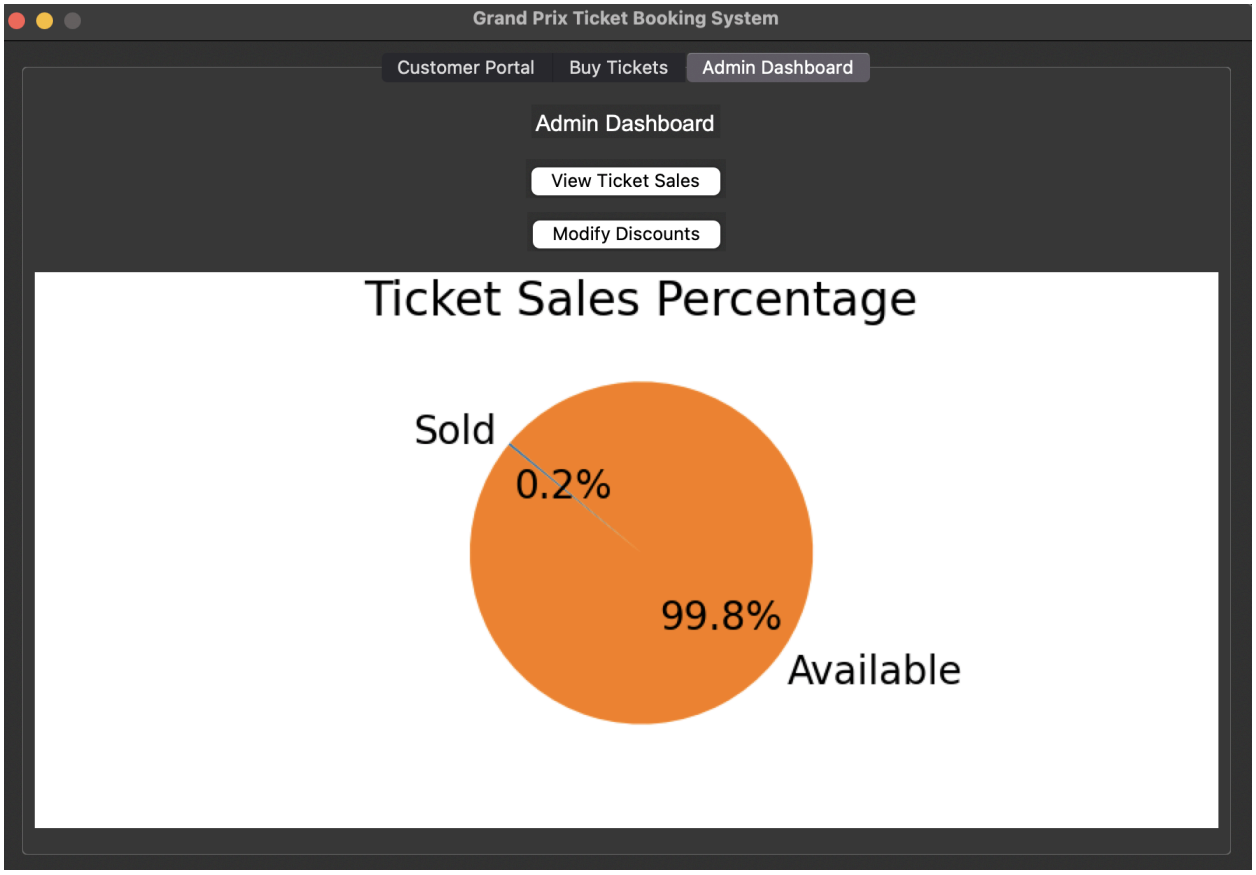
- The image shows a login screen for a "Race Ticketing System" application. A user with the username "salamaghazlan" is attempting to log in. The password field is hidden with asterisks for security. There's a role selection dropdown menu with "User" currently selected an other roles depending on the individuals. A "Login" button appears at the bottom of the form.



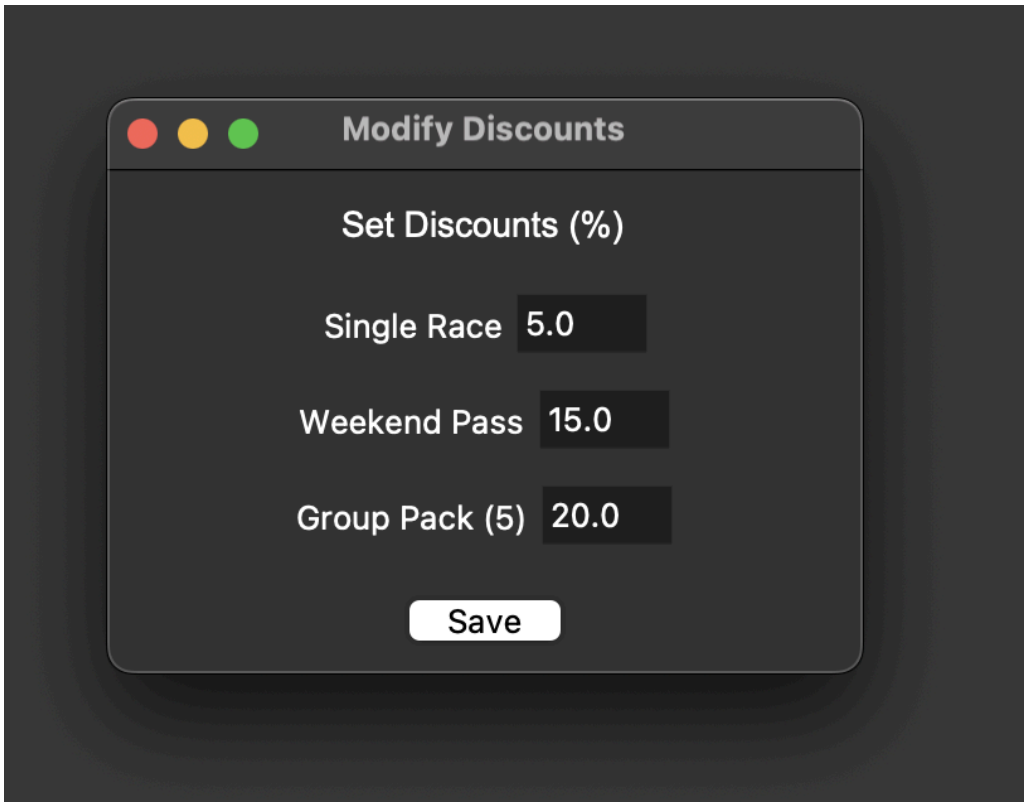
- Now in the Account Management section of the Race Ticketing System. we clicked on "Add Customer" and a popup dialog is asking me to enter the customer's email. I've typed in "salamaghazlan@gmail.com" in the email field. Now I just need to click the blue "OK" button to add this customer, or I could click "Cancel" if I change my mind. there are also other options in this section like "Modify/Delete Customer" and "Display Customers" that I could use to manage customer accounts. At the top of the screen, there are two tabs: "Account Management" (which is currently selected) and "Ticket Purchase" which I could switch to if I need to handle ticket sales instead of customer management. This system seems well-organized, making it easy for me to navigate between different functions while managing race event customers and tickets.



- I've now switched to the Ticket Purchase tab in the Race Ticketing System. I clicked on "Buy Weekend Ticket" and a dialog box has appeared asking me to enter the quantity of tickets I want to purchase. Each weekend ticket costs $120, and I've entered "3" in the quantity field. This means I'll be buying 3 weekend tickets for a total of $360. Now I just need to click the blue "OK" button to confirm my purchase, or I could click "Cancel" if I change my mind. I can see there are other ticket options available too - "Buy Single-Race Ticket" and "Buy Group Ticket" - depending on what type of admission I need.



- I'm now in the Admin Dashboard of the Grand Prix Ticket Booking System. I'm using an administrator account now. I can see a pie chart showing the ticket sales percentage, which indicates that only 0.2% of tickets have been sold so far, while 99.8% are still available. This means ticket sales have just begun for this event. The interface has three main navigation tabs at the top: Customer Portal, Buy Tickets, and Admin Dashboard (which is currently selected). Within the Admin Dashboard section, I have options to "View Ticket Sales" and "Modify Discounts." The system appears to be designed for managing a Grand Prix racing event, allowing administrators to monitor sales and adjust pricing strategies as needed.

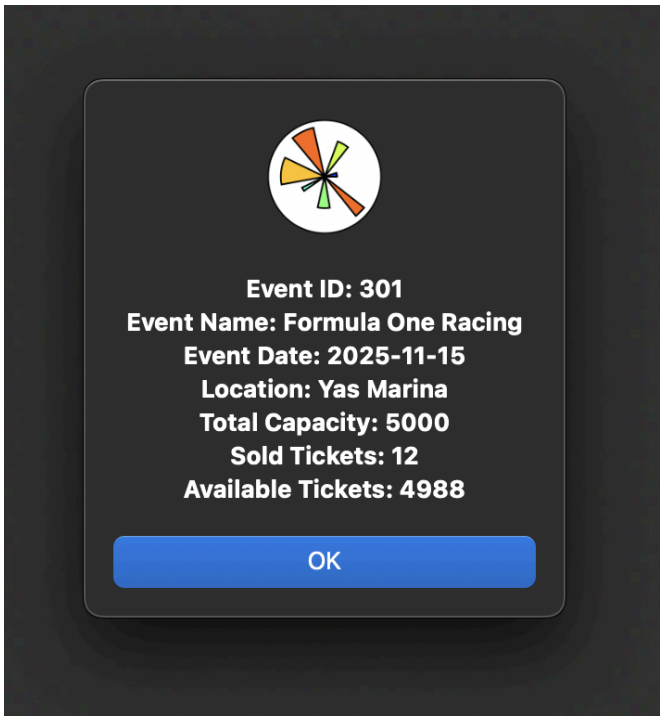


- I'm now in the Modify Discounts window after clicking that option from the Admin Dashboard. Here I can set discount percentages for different ticket types to help boost our sales, which makes sense given how few tickets we've sold so far.

I've set up the following discounts:

- 5.0% off for Single Race tickets
- 15.0% off for Weekend Pass tickets
- 20.0% off for Group Pack tickets (which are for groups of 5)

The Group Pack has the highest discount at 20%, which makes sense as a strategy to encourage larger group purchases. Weekend passes get a moderate 15% discount, while single race tickets have the smallest discount at 5%. All I need to do now is click the "Save" button to apply these discounts to our ticketing system. This should help motivate customers to purchase tickets and improve our sales numbers.



- This display provides key information about an upcoming Formula One Racing event set to take place on November 15, 2025, at Yas Marina. The venue has a total seating capacity of 5,000, but currently, only 12 tickets have been sold, leaving 4,988 still available for purchase. This suggests that ticket sales have just begun or are progressing slowly. The information is likely part of a user interface (GUI), possibly for event organizers or a ticket management system. At the bottom of the display, there's an "OK" button, which is typically used to acknowledge the information and close the window or proceed to another screen.

GitHub Link:

- https://github.com/salalneyadi28/SalamaandGhazlan.git

Summary of Learning:

Salama Al Neyadi

- Throughout this project, I improved my understanding of object-oriented programming by designing a system using inheritance and encapsulation. I worked on the backend logic, especially with the Event, Ticket, and User classes, making sure the data was well-organized and reusable. I also learned how to use Python's pickle module to store and retrieve data safely in binary files. Working on the ticket purchase logic taught me how to link user actions to the backend, and through debugging and testing, I learned how to find and fix errors step by step. I gained a better understanding of how to handle errors, making sure the system could manage unexpected situations without crashing. I also focused on improving the user interface to make it more user-friendly, using design principles that prioritize the user experience. This part of the project helped me connect technical work with how users interact with the application, making it easier to use. This experience made me more confident in designing software that is scalable and easy to maintain by writing clean code and organizing classes properly. I also learned the value of working together and getting feedback from others, which helped me improve the system. Overall, the project gave me a better understanding of software development and set a solid foundation for future work in object-oriented design and Python.

Ghazlan Alketbi

- Throughout this project, I focused on developing the Graphical User Interface (GUI) using Python's tkinter library. I designed a clean and user-friendly interface that supported different user roles, including customer and admin views. One of my main tasks was building the admin dashboard, which allowed administrators to view, update, and manage ticket data, apply discounts, and track sales in real time. I was responsible for user input validation and error handling to ensure that the system responded well to unexpected inputs and prevented crashes. I also implemented real-time updates for ticket availability and added confirmation messages to help guide users through their interactions with the system. These features improved the overall usability of the application and made it more intuitive. In addition to the GUI development, I contributed to organizing the project's structure and created test scenarios to check the stability of the interface under different conditions. I collaborated closely with the backend team to make sure frontend and backend components worked smoothly together, especially when handling ticket operations and user data. This project strengthened my skills in GUI design, real-time data handling, and teamwork. It also gave me hands-on experience in connecting the frontend to backend logic, building user-centered applications, and improving code through testing and feedback.

Our Collaborations:

- As a team, we collaborated on all parts of the project, from designing the UML class diagram to developing the backend logic and user interface. We worked together to make sure each feature, from account management and ticket purchases to admin controls, was functional and well-integrated. We used GitHub to share our progress, manage versions, and keep everything organized and up to date. This project taught us a lot about designing and building a complete Python application. We learned how to create a user-friendly interface, handle real-time data changes, and communicate effectively as a team. It helped us improve both our technical skills and our ability to collaborate on complex projects.