1. Vehicle Management System

Scenario:

Imagine you are building a vehicle management system for a large fleet of vehicles. You need to ensure that each type of vehicle can start its engine in its own way, but all vehicles should be able to share or override certain common behaviors like fuel type. How would you design this system using an abstract class and subclasses?

Abstract Class: `Vehicle`

- **Abstract Method**: `startEngine()`
  - **Purpose**: Different vehicles have different ways of starting their engines. For example, a car might use key ignition, a bike might use a kick-start or self-start, and a truck might need a heavy-load engine warm-up. This method should be abstract because the logic varies for each vehicle type.
- **Concrete Method**: `fuelType()`
  - **Purpose**: Most vehicles use petrol by default, but some might use diesel or electric power. This method provides a default implementation that can be overridden by subclasses if needed.
- **Shared Properties**: `modelName`, `vehicleNumber`, `company`
  - **Purpose**: These properties are common to all vehicles and can be shared across all subclasses.

Subclasses:

- `Car`
  - **Instance Variables**: `numberOfDoors`, `hasSunroof`
  - **Implementation**: Implement `startEngine()` using key ignition logic.
  - **Purpose**: Cars typically start using a key ignition system, so this subclass provides the specific logic for starting a car's engine.
  - **Expected Input**: `Car("Toyota Camry", "ABC123", "Toyota", 4, true)`
  - **Expected Output**:
    - `startEngine()`: "Starting engine with key ignition."
    - `fuelType()`: "Petrol"

- `Bike`
  - **Instance Variables**: `hasSidecar`
  - **Implementation**: Implement `startEngine()` using kick or self-start logic.
  - **Purpose**: Bikes can start using a kick-start or self-start mechanism. This subclass provides the specific logic for starting a bike's engine.
  - **Expected Input**: `Bike("Yamaha R1", "XYZ789", "Yamaha", false)`
  - **Expected Output**:
    - `startEngine()`: "Starting engine with kick-start."
    - `fuelType()`: "Petrol"

- `Truck`
  - **Instance Variables**: `cargoCapacity`
  - **Implementation**: Implement `startEngine()` with heavy-load engine warm-up logic.
  - **Purpose**: Trucks often require a warm-up period before the engine can start due to their heavy-load nature. This subclass provides the specific logic for starting a truck's engine.
  - **Expected Input**: `Truck("Volvo FH", "DEF456", "Volvo", 20000)`

    - **Expected Output**:
        - `startEngine()` : "Starting engine with heavy-load warm-up."
        - `fuelType()` : "Diesel"

Objective:

Ensure all vehicle types start differently, but all can share or override `fuelType()` if needed. This design allows for flexibility while maintaining a common interface for all vehicles.

---

2. Payment Processing System

Scenario:

You are tasked with creating a payment processing system that supports multiple payment methods. Each payment method has its own validation logic, but all payments must follow a consistent process and generate unique transaction IDs. How would you design this system using an abstract class and subclasses?

Abstract Class: `Payment`

- **Abstract Methods**:
    - `processPayment()`
        - **Purpose**: The process of making a payment varies depending on the payment method. For example, a credit card payment requires validation of the card number, expiry date, and CVV, while a UPI payment requires validation of the UPI ID and phone number. This method should be abstract because the implementation varies for each payment type.
    - `validateTransaction()`
        - **Purpose**: Each payment method has specific fields that need to be validated. This method should be abstract to allow each subclass to implement its own validation logic.
- **Concrete Method**:
    - `generateTransactionId()`
        - **Purpose**: All payments need a unique transaction ID. This method provides a common implementation to generate unique IDs, avoiding duplication of code across subclasses.

Subclasses:

- `CreditCardPayment`
    - **Instance Variables**: `cardNumber` , `expiryDate` , `cvv`
    - **Implementation**: Validate card number, expiry, CVV.
    - **Purpose**: Credit card payments require specific validation of the card details. This subclass implements the necessary logic for validating credit card transactions.
    - **Expected Input**: `CreditCardPayment("1234567890123456", "12/25", "123")`
    - **Expected Output**:
        - `processPayment()` : "Processing credit card payment."
        - `validateTransaction()` : "Validating card number, expiry, and CVV."
        - `generateTransactionId()` : "Generated transaction ID: 123456789"
- `UPIPayment`

    - **Instance Variables**: `upiId`, `phoneNumber`
    - **Implementation**: Validate UPI ID and phone number.
    - **Purpose**: UPI payments require validation of the UPI ID and phone number. This subclass implements the necessary logic for validating UPI transactions.
    - **Expected Input**: `UPIPayment("user@upi", "1234567890")`
    - **Expected Output**:
        - `processPayment()`: "Processing UPI payment."
        - `validateTransaction()`: "Validating UPI ID and phone number."
        - `generateTransactionId()`: "Generated transaction ID: 987654321"

- `PayPalPayment`
    - **Instance Variables**: `email`, `authToken`
    - **Implementation**: Validate email and authentication token.
    - **Purpose**: PayPal payments require validation of the email and authentication token. This subclass implements the necessary logic for validating PayPal transactions.
    - **Expected Input**: `PayPalPayment("user@example.com", "abc123")`
    - **Expected Output**:
        - `processPayment()`: "Processing PayPal payment."
        - `validateTransaction()`: "Validating email and authentication token."
        - `generateTransactionId()`: "Generated transaction ID: 543216789"

Objective:

All payment types must follow the same contract, but implementation varies per platform. Shared ID generation avoids duplication. This design ensures consistency and flexibility in the payment processing system.

---

3. Employee Payroll System

Scenario:

You are developing an employee payroll system for a company with different types of employees, each with its own salary calculation logic. How would you design this system using an abstract class and subclasses to ensure all employees can be managed uniformly while having custom salary calculation logic?

Abstract Class: `Employee`

- **Abstract Method**: `calculateSalary()`
    - **Purpose**: Salary calculation varies depending on the type of employee. For example, a full-time employee might have a base pay plus benefits, a part-time employee might have a salary based on hours worked and a rate, and a freelancer might have a salary based on project payments. This method should be abstract because the implementation varies for each employee type.
- **Concrete Methods**:
    - `applyLeave()`
        - **Purpose**: All employees can apply for leave. This method provides a common implementation for applying leave.
    - `getDetails()`

        - **Purpose**: All employees have details that can be retrieved. This method provides a common implementation for getting employee details.

Subclasses:

- `FullTimeEmployee`
  - **Instance Variables**: `basePay`, `benefits`
  - **Implementation**: Implements salary with base pay + benefits.
  - **Purpose**: Full-time employees have a fixed base pay and additional benefits. This subclass provides the specific logic for calculating the salary of full-time employees.
  - **Expected Input**: `FullTimeEmployee("John Doe", "12345", 50000, 10000)`
  - **Expected Output**:
    - `calculateSalary()`: "Calculating salary: Base pay + benefits = $60000"
    - `applyLeave()`: "Leave applied successfully."
    - `getDetails()`: "Employee Details: John Doe, ID: 12345"

- `PartTimeEmployee`
  - **Instance Variables**: `hourlyRate`, `hoursWorked`
  - **Implementation**: Salary based on hours × rate.
  - **Purpose**: Part-time employees are paid based on the number of hours worked and a fixed rate. This subclass provides the specific logic for calculating the salary of part-time employees.
  - **Expected Input**: `PartTimeEmployee("Jane Smith", "67890", 20, 15)`
  - **Expected Output**:
    - `calculateSalary()`: "Calculating salary: Hours worked × rate = $300"
    - `applyLeave()`: "Leave applied successfully."
    - `getDetails()`: "Employee Details: Jane Smith, ID: 67890"

- `Freelancer`
  - **Instance Variables**: `projectPayment`
  - **Implementation**: Salary based on project payment.
  - **Purpose**: Freelancers are paid based on completed projects. This subclass provides the specific logic for calculating the salary of freelancers.
  - **Expected Input**: `Freelancer("Alice Johnson", "54321", 5000)`
  - **Expected Output**:
    - `calculateSalary()`: "Calculating salary: Project payment = $5000"
    - `applyLeave()`: "Leave applied successfully."
    - `getDetails()`: "Employee Details: Alice Johnson, ID: 54321"

Objective:

Let all employees behave similarly at the system level but have custom salary calculation logic. This design ensures uniform management of employees while allowing for flexibility in salary calculations.

---

4. User System for E-Learning Platform

Scenario:

You are building a user system for an e-learning platform that supports different types of users, each with its own dashboard view. How would you design this system using an abstract class and subclasses to ensure a consistent user flow while allowing role-based UI and logic?

Abstract Class: `User`

- **Abstract Method**: `accessDashboard()`
  - **Purpose**: Each user role has a different dashboard view. For example, a student might see enrolled courses and grades, an instructor might see created courses and student submissions, and an admin might see site analytics and user management tools. This method should be abstract because the implementation varies for each user role.
- **Concrete Methods**:
  - `login()`
    - **Purpose**: All users can log in. This method provides a common implementation for logging in.
  - `logout()`
    - **Purpose**: All users can log out. This method provides a common implementation for logging out.
  - `updateProfile()`
    - **Purpose**: All users can update their profiles. This method provides a common implementation for updating user profiles.
- **Common Properties**: `username`, `email`, `role`
  - **Purpose**: These properties are common to all users and can be shared across all subclasses.

Subclasses:

- `Student`
  - **Instance Variables**: `enrolledCourses`, `grades`
  - **Implementation**: Dashboard shows enrolled courses and grades.
  - **Purpose**: Students need to see their enrolled courses and grades. This subclass provides the specific logic for displaying the student dashboard.
  - **Expected Input**: `Student("student1", "student1@example.com", ["Math", "Science"], [90, 85])`
  - **Expected Output**:
    - `accessDashboard()`: "Accessing student dashboard: Enrolled Courses: Math, Science; Grades: 90, 85"
    - `login()`: "Login successful."
    - `logout()`: "Logout successful."
    - `updateProfile()`: "Profile updated successfully."
- `Instructor`
  - **Instance Variables**: `createdCourses`, `studentSubmissions`
  - **Implementation**: Dashboard shows created courses and student submissions.
  - **Purpose**: Instructors need to see the courses they have created and the submissions from students. This subclass provides the specific logic for displaying the instructor dashboard.
  - **Expected Input**: `Instructor("instructor1", "instructor1@example.com", ["Math 101"], {"Math 101": ["Submission 1", "Submission 2"]})`
  - **Expected Output**:

        - `accessDashboard()`: "Accessing instructor dashboard: Created Courses: Math 101; Student Submissions: Math 101 - Submission 1, Submission 2"
        - `login()`: "Login successful."
        - `logout()`: "Logout successful."
        - `updateProfile()`: "Profile updated successfully."

- `Admin`
    - **Instance Variables**: `siteAnalytics`, `userManagementTools`
    - **Implementation**: Dashboard shows site analytics and user management tools.
    - **Purpose**: Admins need to see site analytics and manage users. This subclass provides the specific logic for displaying the admin dashboard.
    - **Expected Input**: `Admin("admin1", "admin1@example.com", {"visitors": 1000, "courses": 50}, ["Add User", "Remove User"])`
    - **Expected Output**:
        - `accessDashboard()`: "Accessing admin dashboard: Site Analytics - Visitors: 1000, Courses: 50; User Management Tools: Add User, Remove User"
        - `login()`: "Login successful."
        - `logout()`: "Logout successful."
        - `updateProfile()`: "Profile updated successfully."

Objective:

Enforce a consistent user flow but allow role-based UI and logic with clean abstraction. This design ensures a uniform user experience while allowing for role-specific functionality.

---

5. Report Generator with Template Pattern

Scenario:

You are tasked with creating a report generator that supports different formats (PDF, Excel, HTML) while following a consistent process. How would you design this system using an abstract class and subclasses to ensure the template pattern is followed?

Abstract Class: `ReportGenerator`

- **Abstract Method**: `generateContent()`
    - **Purpose**: The content of the report varies depending on the format. For example, a PDF report might use block text format, an Excel report might use tabular data formatting, and an HTML report might use markup to format the content. This method should be abstract because the implementation varies for each report format.
- **Concrete Methods**:
    - `openFile()`
        - **Purpose**: All reports need to open a file. This method provides a common implementation for opening the file.
    - `writeContent()`
        - **Purpose**: All reports need to write content. This method provides a common implementation for writing the content.
    - `saveFile()`

        - **Purpose**: All reports need to save the file. This method provides a common implementation for saving the file.

- **Template Method**: Combines all steps into a single flow.
    - **Purpose**: The template method defines the skeleton of the algorithm, calling the abstract and concrete methods in a fixed sequence. This ensures that all report generators follow the same process.

Subclasses:

- `PDFReport`
    - **Instance Variables**: `blockText`
    - **Implementation**: Implements content generation with block text format.
    - **Purpose**: PDF reports require specific formatting using block text. This subclass provides the specific logic for generating PDF content.
    - **Expected Input**: `PDFReport("Sample Report", "This is a sample block text.")`
    - **Expected Output**:
        - `generateContent()`: "Generating PDF content with block text."
        - `openFile()`: "Opening PDF file."
        - `writeContent()`: "Writing content to PDF file."
        - `saveFile()`: "Saving PDF file."

- `ExcelReport`
    - **Instance Variables**: `tabularData`
    - **Implementation**: Implements content generation with tabular data formatting.
    - **Purpose**: Excel reports require specific formatting using tables. This subclass provides the specific logic for generating Excel content.
    - **Expected Input**: `ExcelReport("Sample Report", [["Name", "Age"], ["John", 30], ["Jane", 25]])`
    - **Expected Output**:
        - `generateContent()`: "Generating Excel content with tabular data."
        - `openFile()`: "Opening Excel file."
        - `writeContent()`: "Writing content to Excel file."
        - `saveFile()`: "Saving Excel file."

- `HTMLReport`
    - **Instance Variables**: `markupContent`
    - **Implementation**: Implements content generation with markup to format report content.
    - **Purpose**: HTML reports require specific formatting using HTML markup. This subclass provides the specific logic for generating HTML content.
    - **Expected Input**: `HTMLReport("Sample Report", "<h1>Sample Report</h1><p>This is a sample report.</p>")`
    - **Expected Output**:
        - `generateContent()`: "Generating HTML content with markup."
        - `openFile()`: "Opening HTML file."
        - `writeContent()`: "Writing content to HTML file."
        - `saveFile()`: "Saving HTML file."

Objective:

Follow the Template Design Pattern. Subclasses only deal with the content part while the parent class manages the overall flow. This design ensures consistency and flexibility in report generation.

---

6. Game Character Actions

Scenario:

You are developing a game with different types of characters, each with its own attack logic. How would you design this system using an abstract class and subclasses to ensure all characters can perform common actions while having custom attack logic?

Abstract Class: `GameCharacter`

- **Abstract Method**: `attack()`
  - **Purpose**: The attack logic varies depending on the character type. For example, a warrior might use melee attacks, a mage might use magic spells, and an archer might use long-distance arrow attacks. This method should be abstract because the implementation varies for each character type.
- **Concrete Methods**:
  - `chooseTarget()`
    - **Purpose**: All characters need to choose a target. This method provides a common implementation for selecting a target.
  - `animateAttack()`
    - **Purpose**: All characters need to animate their attacks. This method provides a common implementation for animating attacks.
- **Common Properties**: `name`, `level`, `healthPoints`
  - **Purpose**: These properties are common to all characters and can be shared across all subclasses.

Subclasses:

- `Warrior`
  - **Instance Variables**: `weaponType`
  - **Implementation**: Implements melee attack logic.
  - **Purpose**: Warriors use melee attacks. This subclass provides the specific logic for warrior attacks.
  - **Expected Input**: `Warrior("Conan", 5, 100, "Sword")`
  - **Expected Output**:
    - `attack()`: "Performing melee attack with Sword."
    - `chooseTarget()`: "Target selected."
    - `animateAttack()`: "Attack animation played."
- `Mage`
  - **Instance Variables**: `spellType`
  - **Implementation**: Implements magic spell attack.
  - **Purpose**: Mages use magic spells. This subclass provides the specific logic for mage attacks.
  - **Expected Input**: `Mage("Gandalf", 7, 120, "Fireball")`
  - **Expected Output**:
    - `attack()`: "Casting spell: Fireball."
    - `chooseTarget()`: "Target selected."
    - `animateAttack()`: "Attack animation played."
- `Archer`

    - **Instance Variables**: `arrowType`
    - **Implementation**: Implements long-distance arrow attack.
    - **Purpose**: Archers use long-distance arrow attacks. This subclass provides the specific logic for archer attacks.
    - **Expected Input**: `Archer("Legolas", 8, 110, "Elven Arrow")`
    - **Expected Output**:
        - `attack()`: "Shooting arrow: Elven Arrow."
        - `chooseTarget()`: "Target selected."
        - `animateAttack()`: "Attack animation played."

Objective:

Ensure all characters can perform common actions while having custom attack logic. This design allows for flexibility in character behavior while maintaining a common interface for all characters.