ABSTRACT CLASSES, INTERFACE & POLYMORPHISM

The topics for this week are the key concepts of OOP, abstract classes, interface and

polymorphism. Part A

1. True or false:
   a. An instance of an abstract class is an abstract object.
   b. A class that defines an abstract method, or that inherits an abstract method without overriding it, must be declared as an abstract class.
   c. A final class has no subclasses
   d. A final method can be overridden with an abstract method.
   f. An interface can be used to declare a reference variable, and then an object can be constructed from a class that implements the interface. Later, we can invoke the methods defined in the class from the interface reference.

Part B

Exercise 1:

We want to model dogs and cats. If we design them using inheritance, these two species would inherit from a parent `Animal` class. We also can use interface concept so that each class (`Cat` and `Dog`) implements an interface `Speakable` and has a method called `speak();` a Cat speaks "meow, my name is …" and a Dog speaks "woof, my name is …". Given the `Animal` class as below, define and implement other classes and interface for the task.

```
public abstract class Animal {

    private String name;

     public Animal(String name) {
            this.name = name;
    }

    public String getName( ) {
           return name;
    }
}
```

Exercise 2:

Background:

The TicketSales.Com website sells tickets to a range of entertainment events. The following are some examples:

- A single-use ticket to attend a sporting match
- A multi-use ticket to attend some fixed number of matches by a single team throughout an entire season of the sport.
- A single-use ticket to gain entry to the Zoo, or to a Theme Park, or other tourist attraction.
- A multi-use ticket to watch movies at cinemas

In this lab we will write a program to simulate the selling and usage of some tickets by the website. Read through *all* tasks before you begin, to get an idea of what is expected to be completed.

Task 1: Create Interface definitions

Create a new package for this week's activities called lab07.

First we need to define some *interfaces*, which we will later implement.

1. Create an Interface named *SingleUseTicketing*. It will need to define the following methods:
    - use( ) – this method will be called to signify that the ticket is to be used. The method should return true if the ticket is valid for use, and false if it is not valid for use because of having already been used. The effect of the method should be to change the ticket from useable (i.e. valid), to being no longer valid in the future.
2. Create an interface named *MultiUseTicketing*. It will need to define the following methods:
    - loadUp(*amount*) – this method will be called to add a specified amount of uses to the ticket (each is called a use-entitlement). For example if the parameter is 3, this means the ticket can be used 3 times more than it previously could. Thus if the ticket had been fully used-up, it would again be available to be used.
    - useOnce( ) – this method will be called to signify that the ticket is to be used one more time. The method should return true if the ticket is valid for use, and false if it is not valid for use because of having expired, i.e. having been fully used-up. The effect of the method should be to decrease the number of uses that remain for the ticket. For example, if 3 uses had been loaded onto the ticket, then the first call to useOnce will reduce the number to 2.
    - getLoadUpCost(amount) – returns the price required to increase the number of remaining uses of the ticket, for the *number* of uses specified.

Task 2: Create abstract class definition

3. Define an abstract class named *Ticket*. It should provide the following methods, possibly as abstract methods:
    - getPurpose( ) – returns a String describing the purpose for which the ticket may be used, e.g. to attend a sport match, to attend multiple sport matches, to enter the Zoo, etc.
    - getCost( ) – returns an amount signifying the cost of obtaining the ticket. Note that for single-use ticketing types, this is the same cost as obtaining the physical ticket; but for multi-use ticketing types, it is the cost of obtaining the physical ticket that can be used on multiple occasions and is separate from the cost of entry.
    - isValid( ) – returns a boolean value indicating if the ticket is valid for use, or not.

Task 3: Create other class definitions

4. Create a concrete subclass of *Ticket* named *TouristTicket* that implements the *SingleUseTicketing* interface.
    - One attribute should be the name of the tourist attraction (e.g. Zoo, Water World, etc.)

- Another attribute should be the price of entry to the attraction

These attributes should be set in the constructor, from parameters.
Ensure that the object has a suitable initial state.
Ensure that all required methods as specified by the interface or superclass are implemented (with bodies).
You may like to commence task 9 now, to check that this class is working as expected.

5. Create a subclass of *Ticket* named *SportTicket* that defines the following attributes:
   - Sport name
   - Main Team (the one who is the 'Home' team, in terms of marketing or seating arrangements)
   - The year in which the ticket may be used

   Ensure there is an appropriate constructor to set the values, and accessors and toString method, and implement any methods required by the superclass. Note that SportTicket will have two subclasses, one for a single-use ticket (task 6), and one for a multi-use ticket (task 7).

   Question: Will this class be abstract or concrete? Why? ______________________
   ____
   ____
   ____

6. Create a subclass of *SportTicket* named *SportOneOffTicket* that implements the *SingleUseTicketing* interface. It will need to ensure that the following information is stored in attributes:
    - The cost of the ticket
    - The venue at which it may be used (e.g. Melbourne Cricket Ground, Sydney Olympic Stadium, etc.)

7. Create a subclass of *SportTicket* named *SportSeasonTicket* that implements the *MultiUseTicketing* interface. It will need to ensure that the following information is stored in attributes:
    - The cost of the ticket itself, without any use-entitlements.
    - The cost for each use-entitlement (each entry/admittance to a match), i.e. to load 4 admittances onto the ticket may cost $17 each when pre-purchased, whereas the base cost of the ticket itself may be just $5 and is charged just once for the whole year.
    - How many times it may be used (the initial number of pre-paid admittances the person will be entitled to before needing to load more onto their smart ticket).

    Make sure that you implement the methods required by the MultiUseTicketing interface, so that you check how many uses remain and check the ticket's validity.

8. (Optional)
    Create a subclass of *Ticket* named *MovieTicket* that implements the *MultiUseTicketing* interface. It will need to ensure that the following information is stored in attributes:
    - The cost of the ticket itself, without any use-entitlements.
    - The name of the Cinema chain/company at which the ticket may be used.
    - The cost of each use-entitlement.

    Ensure there are accessors for the above, and that there is a suitable constructor to set initial values for the attributes.
    Provide a separate method named *setLoadUpCost* which allows you to set a new value for future purchases of use-entitlements to be added to the ticket. The initial value for this attribute should be set in the constructor.

Task 4: Write a driver class

You may prefer to do this part gradually as you work through the other tasks.

9. Write a driver which creates objects of the concrete class types, and calls methods on the objects.
    This driver could be a hard-coded one in early stages, but by the end should be a fully interactive, textual menu-based system that allows the user to select what type of ticket to create, specify the details of the ticket as depending on the type of tickets, and then set that ticket as the 'Currently Active' ticket for further actions to be done to it, such as to attempt to 'use' it, attempt to add more use-entitlements to it (if applicable), display ticket-specific information.

Extension Tasks:

- Consider writing classes for additional tickets such as for music concerts, transport ares, etc.