1.1. Consider using the following Card class.

```
public class Card
{
   private String name;

   public Card()
   {
      name = "";
   }

   public Card(String n)
   {
      name = n;
   }

   public String getName()
   {
      return name;
   }

   public boolean isExpired()
   {
      return false;
   }

   public String format()
   {
      return "Card holder: " + name;
   }
}
```

Use this class as a superclass to implement a hierarchy of related classes:

Class	Data
`IDCard`	ID number
`CallingCard`	Card number, PIN
`DriverLicense`	Expiration year

Write declarations for each of the subclasses. For each subclass, supply private instance variables. Leave the bodies of the constructors and the `format` methods blank for now.

Note that `DriverLicense` **extends** `IDCard`, not `Card`. Also note that this class will not compile until you complete the next sections.

1.2. Implement constructors for each of the three subclasses. Each constructor should call the superclass constructor to set the name. Here is one example:

```
public IDCard(String n, String id)
{
   super(n);
   idNumber = id;
```

```
}
```

1.3. Replace the implementation of the `format` method for the three subclasses. The methods should produce a formatted description of the card details. The subclass methods should call the superclass `format` method to get the formatted name of the cardholder.

1.4. Devise another class, `Billfold`, as below, which contains slots for two cards, `card1` and `card2`, a method `void addCard(Card)` and a method `String formatCards()`.

```
public class Billfold
{
   private Card card1;
   private Card card2;

   public void addCard(Card c)
   {
      if (card1 == null)
         card1 = c;
      else if (card2 == null)
         card2 = c;
   }

   public String formatCards()
   {
      return "[" + card1.format() + " | " + card2.format() + "]";
   }
}
```

The `addCard` method checks whether `card1` is `null`. If so, it sets `card1` to the new card. If not, it checks `card2`. If both cards are set already, the method has no effect.

Of course, `formatCards` invokes the `format` method on each non-`null` card and produces a string with the format

```
[card1|card2]
```

1.5. Write a tester program that adds two objects of different subclasses of `Card` to a `Billfold` object. Test the results of the `formatCards` methods.

What is the code for your `BillfoldTester` class?

1.6. Explain why the output of your `BillfoldTester` program demonstrates polymorphism.

1.7. The `Card` superclass defines a method `isExpired`, which always returns `false`. This method was overridden in `DriverLicense` with an identical body, but the method is not appropriate for the driver license. Supply a method body for `DriverLicense.isExpired()` that checks whether the driver license is already expired (i.e., the expiration year is less than the current year).

To work with dates, you can use the methods and constants supplied in abstract class `Calendar` which are inherited by the concrete class `GregorianCalendar`. You create a `Calendar` as follows:

```
GregorianCalendar calendar = new GregorianCalendar();
```

Then, you can obtain the current year using the constant `Calendar.YEAR` and method `get` in `GregorianCalendar`. The constant indicates that the method should return the current year. By comparing the returned value with the `expYear` field in `DriverLicense`, you can determine if the card is expired. The code below will retrieve the current year.

```
calendar.get(Calendar.YEAR)
```

1.8. Add a method `getExpiredCardCount`, which counts the number of expired cards in the billfold, to the `Billfold` class.

1.9. Write a `BillfoldTester` class that populates a billfold with a phone card and an expired driver license. Then call the `getExpiredCardCount` method. Run your tester to verify that your method works correctly.

What is your tester class?

1.10. Implement `toString` methods for the `Card` class and its three subclasses. The methods should print:

the name of the class
the values of all instance variables (including inherited instance variables)

Typical formats are:

```
Card[name=Edsger W. Dijkstra]
CallingCard[name=Bjarne Stroustrup][number=4156646425,pin=2234]
```

Write the code for your `toString` methods.

1.11. Implement `equals` methods for the `Card` class and its three subclasses. Cards are the same if the objects belong to the same class, and if the names and other information (such as the expiration year for driver licenses) match.

Give the code for your `equals` methods.

1.12. Change the `Card` class and give `protected` access to `name`. Would that change simplify the `toString` method of the `CallingCard` class? How?

Is this change advisable?

a. Answer the following questions in a separate sheet, use a "doc" or other text formats:

1) Have the exercises above required you to implement a software design specification for an appropriate development environment.

a) True

b) False

2) Have the lab exercises above demonstrated that you the need to acquire instruction on the proper documentation of source code?

a) True

b) False (If not, why and do n't you think that you need to provide documentation for any implementation & design for any software.)

3) Provide an example that you have acquired the knowledge to design and apply relevant software testing procedures, (Note: please provide step by step processes that you take).