|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1. | Consider using the following Card class.  public class Card  {  private String name;       public Card()      {          this(“john”); //calling the other constructor using  // the keyword ‘this’      }       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 | | Calling Card | Card number, PIN | | Driver License | Expiration year |   Write definitions for each of the subclasses. For each subclass, supply private instance variables. Leave the bodies of the constructors blank for now. |
|  |  |

**Calling the Superclass Constructor**

|  |  |
| --- | --- |
| 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;  } |

**Overriding Methods**

|  |  |
| --- | --- |
| 3. | Supply 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. |

|  |  |
| --- | --- |
| 4. | Devise another class, Billfold, which contains slots for two cards, card1 and card2, a method void addCard(Card) and a method String formatCards().  The addCard method checks if 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 concatenates the resulting strings.  What is your Billfold class? |

A place to store and manipulate groups of objects.

**package** inheritance;

**import** java.util.\*;

/\*\*

\* Stores an certain amount of cards.

\*

\* **@author** Katya Bezugla

\*/

**public** **class** Billfold {

/\*\*

\* Represents the cards stored.

\*/

ArrayList<Card> cards = **new** ArrayList<Card>();

/\*\*

\* Will construct the object, and set the amount of cards one can store as 2.

\*

\*/

**public** Billfold() {

**this**(2);

}

/\*\*

\* Will construct the object, and set the amount of cards one can store as the

\* inputed amount.

\* **@param** cardAmount the amount of cards one can store.

\*

\*/

**public** Billfold(**int** cardAmount) {

**for** (**int** i=0;i<cardAmount;i++) {

cards.add(**null**);

}

}

/\*\*

\* Adds a card to the first available slot.

\* **@param** card the card that will be added.

\*

\*/

**public** **void** addCard(Card card) {

**for** (**int** i=0;i<cards.size();i++) {

**if** (cards.get(i) == **null**) {

cards.set(i, card);

**break**;

}

}

}

/\*\*

\* Returns the formatted version of the information of all the cards stored.

\*

\* **@return** the formatted version of the cards' information.

\*/

**public** String formatCards() {

String card\_str = "";

**for** (**int** i=0;i<cards.size();i++) {

**if** (cards.get(i) != **null**) {

card\_str += cards.get(i).format() + "\n";

}

}

**return** card\_str;

}

}

|  |  |
| --- | --- |
| 5. | Write a class with a test program that adds two objects of different subclasses of Card to a Billfold object. Print the results of the formatCards methods.  What is the code for your test program?  **package** inheritance;  /\*\*  \* A playground for testing different methods of the card super and subclasses,  \* and the billfold class.  \*  \* **@author** Katya Bezugla  \*/  **public** **class** Tester {  **public** **static** **void** main(String[] args) {  Billfold billfold = **new** Billfold();  Card card1 = **new** Card("Bob");  Card card2 = **new** Card("Sue");    billfold.addCard(card1);  billfold.addCard(card2);    System.***out***.println(billfold.formatCards());  }  } |
| 6. | What is the output of your test program? |

Card holder: Bob

Card holder: Sue

|  |  |
| --- | --- |
| 7. | Explain why the output of your program demonstrates polymorphism. |

It demonstrated polymorphism as the program must decide which format method it will call, the one in the original Card class, or the one in the subclasses to the Card class. In my tester program, it ends up only having to call the format method in the superclass, but if a subclass is added to billfold, it will overwrite the format method in the superclass with the one in the subclass, displaying further polymorphism.

|  |  |
| --- | --- |
| 8. | The Card superclass defines a method isExpired, which always returns false. This method is not appropriate for the driver license. Supply a method DriverLicense.isExpired() that checks if the driver license is already expired (i.e., the expiration year is less than the current year).  To find out the current year, you can use the get method of the class Calendar. For example, if you create a Calendar as follows:  GregorianCalendar calendar = new GregorianCalendar();  Then, you can obtain the current year using  calendar.get(Calendar.YEAR)  What is the code for your isExpired method? |
|  |  |

**public** **boolean** isExpired() {

GregorianCalendar calendar = **new** GregorianCalendar();

**int** current = calendar.get(GregorianCalendar.***YEAR***);

**int** expire = Integer.*parseInt*(**this**.expire);

**if** (expire < current) {

**return** **true**;

} **else** {

**return** **false**;

}

}

|  |  |
| --- | --- |
| 9. | The ID card and the phone card don't expire. What should you do to reflect this fact in your implementation? |

Continue having them refer to the isExpired method in the Card superclass when a isExpired method is called for them, as it will always return false then.

|  |  |
| --- | --- |
| 10. | Add a method getExpiredCardCount, which counts the number of expired cards in the billfold, to the Billfold class. |
| 11. | Write a test class that populates a billfold with a phone card and an expired driver license. Then call the getExpiredCardCount method. Run your test program to verify that your method works correctly.  What is your test program? |

**package** inheritance;

/\*\*

\* A playground for testing different methods of the card super and subclasses, and

\* the billfold class.

\*

\* **@author** Katya Bezugla

\*/

**public** **class** Tester {

**public** **static** **void** main(String[] args) {

Billfold billfold = **new** Billfold();

Card card1 = **new** CallingCard("Bob","123","456");

Card card2 = **new** DriverLicense("Sue","2000","123456");

billfold.addCard(card1);

billfold.addCard(card2);

System.***out***.println(billfold.formatCards());

System.***out***.println("Amount of expired cards: " + billfold.getExpiredCardCount());

}

}

**The toString method**

|  |  |
| --- | --- |
| 12. | Define toString methods for the Card class and its three subclasses. The methods should print:   * the name of the class * the values of all instance fields (including inherited fields)   Typical formats are:  Card[name=Edsger W. Dijkstra]  CallingCard[name=Bjarne Stroustrup][number=4156646425,pin=2234]  Give the code for your toString methods.  Card Class:  **public** String toString()  {  **return** **this**.getClass().getSimpleName() + "[name=" + name + "]";  }  ID Card Class:  **public** String toString()  {  **return** **super**.toString() + "[idNum=" + idNum + "]";  }  Calling Card Class:  **public** String toString()  {  **return** **super**.toString() + "[cardNum=" + cardNum + ",pin=" + pin + "]";  }  Driver License Class:  **public** String toString()  {  **return** **super**.toString() + "[expire=" + expire + "]";  } |

**The equals method**

|  |  |
| --- | --- |
| 13. | Define 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.  We only need one equals method in the Card superclass which will work for all the subclasses.  **public** **boolean** equals(Card card1) {  **if** (**this**.toString().equals(card1.toString())) {  **return** **true**;  } **else** {  **return** **false**;  }  } |

**Protected access**

|  |  |
| --- | --- |
| 14. | Change the Card class and give protected access to name.  (a) Would that change simplify the toString method of the CallingCard class? How?  It would not simplify the toString method, as I do not call the getName function anywhere in that method. Instead I just call super.toString() to format the name part. If I did call getName anywhere in the toString card of the CallingCard class, it would simplify it a little bit as I could just call <object name>.name, but not by much.  (b) Is this change advisable? |

It is, as it will allow the subclasses to access the field methods of the superclass, which will simplify the code if you start adding more variables to the superclass that the subclasses still need to access. While the subclasses will be able to access the variables, nothing else will be able to access them, leaving them protected against change from outside classes and code.