Carleton University
Department of Systems and Computer Engineering
SYSC 2004 A - Object-Oriented Software Development - Fall 2013

Midterm Exam – October 23, 2013 - Solutions

Please read these instructions before you answer any of the exam questions:

1. The midterm exam is closed book. Calculators are permitted.
2. Do not talk to any other students from the time the exam begins until your completed exam has been submitted **and you have left the exam room**.
3. **Do not ask your instructor or the TAs questions during the exam.** Exam questions will not be explained, and no hints will be given. **One of the things we are examining is your ability to understand and interpret requirements (problem statements), so we are not going to explain the questions to you.** If you think something is unclear or ambiguous, make a reasonable assumption (one that does not contradict the question) write it at the start of your solution, and answer the question.
4. You do not have to copy any of the Javadoc comments or Java code from the question paper to your answer booklet. Just write the code you are asked to write!
5. API documentation for `ArrayList` and `String` is provided at the end of this question paper. You probably won't need all the methods when writing your solutions; it's up to you to figure out which ones are useful and which ones aren't.

Overview

There are several Web-based stores that allow customers to purchase and download individual songs; for example, bandcamp.com or Amazon MP3. For many of these stores, a customer must first create an account. Account holders can then purchase any number of songs. Each song has a title and a price. From time to time, the store offers a discount on selected songs; for example, songs by a specific artist might be discounted by 10% this weekend.

Class `Song` models songs. Each instance of this class stores one song's title, price and discount.

Customers are modelled by class `Customer`. Each instance of this class stores the name and account number of the customer, along with a reference to the list of songs that this person wants to purchase; in other words, a "shopping cart".

Question 1 [15 marks]

Here is an incomplete implementation of class Song. The specifications for the methods you have to write are contained in the Javadoc comments. Read the Javadoc comments for the constructors and methods, and write the code in your answer booklet. Please write the constructor/method headers as well as their bodies. Do not copy the class header, the fields or the comments. You are not permitted to define additional fields in this class or change the signatures of the constructor and methods. Your methods can, of course, define any local variables that are required.

```
/**
 * A Song object represents a song's title, price and discount.
 */
public class Song
{
    private String title;
    private double price;
    private int discount;  // 10% is stored as 10, 30% as 30, etc.
    private static final int DEFAULT_DISCOUNT = 0;

    /**
     * Initialize this Song object with the specified title and price.
     * The discount is set to the default value of 0 (zero).
     * Assume that the price will always be positive.
     */
    public Song(String title, double price)
    {
        this.title = title;
        this.price = price;
        discount = DEFAULT_DISCOUNT;
    }
    /**
     * Initialize this Song object with the specified title, price
     * and discount. Assume that the price will always be positive.
     * If the discount is negative or greater than 100, set it to the
     * default value of 0 (zero).
     */
    public Song(String title, double price, int discount)
    {
        this.title = title;
        this.price = price;
        if (discount < 0 || discount > 100) {
            this.discount = DEFAULT_DISCOUNT;
        } else {
            this.discount = discount;
        }
    }
```

```
    /**
     * Accessor for the title.
     */
    public String getTitle()
    {
        return title;  // or, return this.title;
    }

    /**
     * Accessor for the price, before any discount is applied.
     */
    public double getPrice()
    {
        return price;     // or, return this.price;
    }

    /**
     * Accessor for the purchase price. This is the price that the
     * customer will pay, after any discount has been applied.
     */
    public double getPurchasePrice()
    {
        return price * (1 – discount / 100.0);
    }

    /**
     * Mutator for the discount.
     */
    public void setDiscount(int discount)
    {
        // The constructor ensures that the discount is always
        // between 0 and 100, inclusive, so this method should
        // do the same. Otherwise, we could call this method to
        // mutate the object into an invalid state.

        if (discount < 0 || discount > 100) {
            this.discount = DEFAULT_DISCOUNT;
        } else {
            this.discount = discount;
        }
    }

}
```

Alternate Solution:

```
/**
 * A Song object represents a song's title, price and discount.
 */
public class Song
{
    private String title;
    private double price;
    private int discount;  // 10% is stored as 10, 30% as 30, etc.
    private static final int DEFAULT_DISCOUNT = 0;

    /**
     * Initialize this Song object with the specified title and price.
     * The discount is set to the default value of 0 (zero).
     * Assume that the price will always be positive.
     */
    public Song(String title, double price)
    {
        this(title, price, DEFAULT_DISCOUNT);
    }

    /**
     * Initialize this Song object with the specified title, price
     * and discount. Assume that the price will always be positive.
     * If the discount is negative or greater than 100, set it to the
     * default value of 0 (zero).
     */
    public Song(String title, double price, int discount)
    {
        this.title = title;
        this.price = price;
        setDiscount(discount);
    }

    /**
     * Accessor for the title.
     */
    public String getTitle()
    {
        return title;  // or, return this.title;
    }

    /**
     * Accessor for the price, before any discount is applied.
     */
    public double getPrice()
```

```
    {
        return price;     // or, return this.price;
    }

    /**
     * Accessor for the purchase price. This is the price that the
     * customer will pay, after any discount has been applied.
     */
    public double getPurchasePrice()
    {
        return price * (1 – discount / 100.0);
    }

    /**
     * Mutator for the discount.
     */
    public void setDiscount(int discount)
    {
        // Ensure that the discount is always between 0 and 100,
        // inclusive.
        if (discount < 0 || discount > 100) {
            this.discount = DEFAULT_DISCOUNT;
        } else {
            this.discount = discount;
        }
    }

}
```

Question 2 [15 marks]

Here is an incomplete implementation of class `Customer`. The specifications for the methods you have to write are contained in the Javadoc comments. Read the Javadoc comments for the constructors and methods, and write the code in your answer booklet. Please write the constructor/method headers as well as their bodies. Do not copy the class header, the fields or the comments. You are not permitted to define additional fields in this class or change the signatures of the constructor and methods. Your methods can, of course, define any local variables that are required.

```
import java.util.*;

/**
 * A Customer has a name, an account number, and a list of the
 * songs that the person is going to purchase (a "shopping cart").
 */
public class Customer
{
```

```
private String name;
private int accountNumber;
private ArrayList<Song> shoppingCart;

/**
 * Initialize this Customer with the specified name and
 * account number, and create an empty list for the shopping cart.
 */
public Customer(String name, int acctNum)
{
    this.name = name;
    accountNumber = acctNum;
    shoppingCart = new ArrayList<Song>();
}

/**
 * Empty this Customer's shopping cart, discarding all songs
 * that were in the cart before this method was called.
 */
public void emptyShoppingCart()
{
    shoppingCart.clear();
}

// Alternate versions of emptyShoppingCart are presented after the
// complete class definition.
```

```
/**
 * Add the Song referred to by parameter song to the
 * shopping cart, according to these rules:
 *   If a song with the same title is already in the shopping
 *   cart, replace it with the one referred to by song (because
 *   it might have a different price or discount than the one in
 *   the shopping cart).
 *   If no song with the same title is currently in the
 *   shopping cart, add song to the cart.
 */
public void addSong(Song song)
{
    for (int i = 0; i < shoppingCart.size(); i += 1) {

        /* Check if the song at index i in the cart has the
         * same title as the Song referred to by parameter song.
         */
        if (shoppingCart.get(i).getTitle().equals(song.getTitle()))
        {
            shoppingCart.remove(i);
            // It doesn't matter where in the list we put song,
            // so append it to the end.
            shoppingCart.add(song);

            // or, instead of calling remove/add, we can
            // replace the Song object at index i with song:
            // shoppingCart.set(i, song);

            return;  // If we don't return here, the call to add()
                     // after the loop will put the song in the
                     // cart a second time.
        }
    }
    // We didn't find a song with the same title in the cart.
    shoppingCart.add(song);
}

/**
 * Return the purchase price of all the songs in the
 * shopping cart. This price includes all discounts.
 */
public double totalPurchasePrice()
{
    double price = 0;

    // sum the discounted prices of all the songs in the cart
    for (Song s : shoppingCart) {
        price += s.getPurchasePrice();
```

```
        }
        return price;
    }
}
```

Alternate Implementations of `emptyShoppingCart`

This method works:

```
    public void emptyShoppingCart()
    {
        int n = shoppingCart.size();
        for (int i = 0; i < n; i++) {

            // Remove the song at index 0, n times.
            // Each time a song is removed, all of the other
            // songs in the list "shift to the left" by one
            // position.

            shoppingCart.remove(0);
        }
    }
```

So does this one:

```
    public void emptyShoppingCart()
    {
        while (!shoppingCart.isEmpty()) {
            shoppingCart.remove(0);
        }
    }
```

This method doesn't work. It uses the `remove` method that is passed a reference to an object , instead of the `remove` method that is passed an integer index. For this method to do what we want, the object's class (in this case, `Song`) must define an `equals` method, which `remove` calls when it searches the `ArrayList` for the object to delete. Also, in general, you should not attempt to modify a list while iterating over it with a *for-each* loop. Doing so may cause a run-time error.

```
    public void emptyShoppingCart()
    {
        for (Song s: shoppingCart) {
            shoppingCart.remove(s);
        }
    }
```

Here's another method that doesn't work. Each time we remove a song, we decrease the size of the `ArrayList` by 1, which means `shoppingCart.size()` returns a different value each time it is called. As such, the loop won't iterate the correct number of times. Also, each time we call

`remove`, all remaining songs in the list shift to the left by 1 position. Because index `i` is incremented each time the loop body is executed, this loop skips over some songs.

```
public void emptyShoppingCart()
{
    for (int i = 0; i < shoppingCart.size(); i++) {

        // Each time a song is removed, all of the other
        // songs in the list “shift to the left” by one
        // position. The song that was at position i+1 is
        // now at position i. We don’t account for this.

        shoppingCart.remove(i);
    }
}
```

Question 3 [5 marks]

A preferred customer is a kind of customer who receives an extra 10% discount on all purchases if they purchase more than 20 songs at one time. Class `PreferredCustomer` models this type of customer.

When answering this question, you are not permitted to make any changes to the `Customer` class from Question 2; i.e., you cannot define additional fields or methods in `Customer`.

Write the complete class definition (header, fields, constructor and methods) for `PreferredCustomer`. This class must be a subclass of `Customer`.

- Define a constructor to initialize the customer's name and account number, and perform any other required initialization;

- Define a `totalPurchasePrice` method that returns the purchase price of all the songs in the shopping cart after the extra discount has been applied. Example: if class `Customer` calculates the total purchase price, including discounts, of the songs in the customer's shopping cart to be $5.00, this method will return 4.50 ($5.00 minus a 10% discount). This method's signature is:

```
public double totalPurchasePrice()
```

- Override any other inherited methods that you think should be overridden.

Solution

Things I looked for:

- Did the student declare that `PreferredCustomer` is a subclass of `Customer` (the `extends` clause)?
- Did the student copy the declarations of fields `name`, `accountNumber` and `shoppingCart` from `Customer` into `PreferredCustomer`? I've mentioned in several lectures that we don't do this. A `PreferredCustomer` object inherits these fields from `Customer`.
- Is the `PreferredCustomer` constructor passed values for the name and account number, and is `super()` called to pass these values to the superclass constructor?
- Do methods in `PreferredCustomer` access the `name`, `accountNumber` and `shoppingCart` fields that are inherited from `Customer`? This won't work, because the fields have `private` visibility.
- Does `totalPurchasePrice` call the `totalPurchasePrice` method in `Customer` (e.g., `price = super.totalPurchasePrice();`)?

```
public class PreferredCustomer extends Customer
{
    // titles keeps track of the titles of the Songs that are in the
    // customer's shopping cart. There are no duplicate titles, which
    // means that the size of this list will be the same as the size
    // of the preferred customer's shopping cart.

    private ArrayList<String> titles;

    // Minimum number of songs you have to buy to get the
    // additional discount.
    private static final int MIN_SONGS = 20;
    private static final int DISCOUNT = 10;  // 10%

    pubic PreferredCustomer(String name, int acctNum)
    {
        super(name, acctNum);
        titles = new ArrayList<String>();
    }

    public void addSong(Song song)
    {
        // Is a Song with the same title as song already in the
        // shopping cart?
        for (String title : titles) {
            if (title.equals(song.getTitle()) {
                // Found a match, so don't add the song to the cart.
                return;
            }
        }
        // No match found, put the song in the cart,
        // update the list of titles.
        super.addSong(song);
        titles.add(song.getTitle());
    }

    public double totalPurchasePrice()
    {
        double price = super.totalPurchasePrice();

        if (titles.size() > MIN_SONGS) {
            price = price * (1 – DISCOUNT / 100.0);
        }
        return price;
    }

    public void emptyShoppingCart()
```

```
    {
        titles.clear();
        super.emptyShoppingCart();
    }

}
```

Some students noted that the solution would be simpler (no need for the list of song titles, and no need to override `addSong` and `emptyShoppingCart`) if we could modify class `Customer`, defining a method that returns the number of songs in the shopping cart:

In `Customer`:

```
    public int sizeOfCart()
    {
        return shoppingCart.size();
   }
```

In `PreferredCustomer`:

```
    public double totalPurchasePrice()
    {
        double price = super.totalPurchasePrice();

        if (sizeOfCart() > MIN_SONGS) {
            price = price * (1 – DISCOUNT / 100.0);
        }
        return price;
    }
```

API Summary – Class `ArrayList<E>`

```
// Appends o (of type E) to the end of this list. Returns true if
// successful, otherwise returns false.
boolean add(E o);

// Returns the number of objects stored in this list.
int size();

// Returns the object (of type E) at the specified position (index)
// in the list. index must be >= 0 and < size().
// The object is not removed from the list.
E get(int index);

// Returns true if this list has no objects, otherwise returns false.
boolean isEmpty();

// Removes the object (of type E) at the specified position (index)
// in the list. index must be >= 0 and < size(). Shifts any subsequent
// elements to the left (subtracts one from their indices).
// Returns the object that was removed from the list.
E remove(int index);

// Replaces the object (of type E) at the specified position (index) in
// this list with the specified element. index must be >=0 and < size().
// Returns the object that was previously stored at the specified
// position.
E set(int index, E element);

// Removes all the objects from the list. The list will be empty when
// this method returns.
void clear();
```

API Summary – Class `String`

```
// Compares this string with the object provided as the argument
// Returns true if the two strings are the same length and are equal on a
// character-by-character basis.

boolean equals(Object anObject);
```