2. A company sends a form letter to all of its potential customers. In order to personalize each letter, various tokens (symbols) in the form letter are replaced by either the customer's name, city, or state, depending on the token. A customer can be represented by a Customer class, whose partial implementation is shown below.

```
public class Customer
{
    private String name;
    private String city;
    private String state;

    /** @return the name of this customer */
    public String getName()
    { return name; }

    /** @return the city of this customer */
    public String getCity()
    { return city; }

    /** @return the state of this customer */
    public String getState()
    { return state; }

    //Constructor and other methods are not shown.
}
```

A FormLetter object has a list of lines that make up the letter, and a list of customers who will receive the letter. In this question you will be asked to write two methods of the FormLetter class, whose partial implementation is shown below.

```
public class FormLetter
{
    /** The list of lines that make up this form letter */
    private List<String> lines;

    /** The list of customers */
    private List<Customer> customers;

    /** @return a copy of lines */
    public List<String> makeCopy()
    {
        List<String> newLines = new ArrayList<String>();
        for (String line: lines)
            newLines.add(line);
        return newLines;
    }
```

```
    /** Replace all occurrences of sub in line with replacement
     *  string, repl.
     *  @param line a String
     *  @param sub a substring to be replaced
     *  @param replacement the replacement string
     *  Precondition: sub is not a substring of repl,
     *                the replacement string.
     *  @return line with each occurrence of sub replaced by replacement
     */
    public String replaceAll(String line, String sub, String repl)
    { /* to be implemented in part (a) */ }

    /** Write letter for one customer, using personalized lines
     *  contained in customerLines.
     *  @param customerLines the personalized lines for one customer
     */
    public void writeLetter(List<String> customerLines)
    { /* implementation not shown */ }

    /** Creates and prints a personalized form letter for each
     *  customer in the customers list.
     *  Postcondition: In each customer letter:
     *    - every occurrence of "@" is replaced by the customer's name;
     *    - every occurrence of "&" is replaced by the customer's city;
     *    - every occurrence of "$" is replaced by the customer's state.
     *    - A letter with the replacements is printed for each customer.
     */
    public void createPersonalizedLetters()
    { /* to be implemented in part (b) */ }

  //Constructors and other methods are not shown.
}
```

(a) Write the `FormLetter` method `replaceAll`, which examines a given string and replaces all occurrences of a specified substring with a replacement string. In writing your solution, you may not use the `replace`, `replaceAll`, or `replaceFirst` methods in the Java `String` class.

Suppose `f` is a `FormLetter`. The following table shows the result of calling `f.replaceAll(line,substring,replacement)`.

line	substring	replacement	string returned
oh me oh my	oh	aah	aah me aah my
sing to me a sin	sin	brin	bring to me a brin
ooh la la	ah	oh	ooh la la

Practice Exam 3

Complete method replaceAll below.

```
/** Replace all occurrences of sub in line with replacement
 *  string, repl.
 *  @param line a String
 *  @param sub a substring to be replaced
 *  @param replacement the replacement string
 *  Precondition: sub is not a substring of repl,
 *                the replacement string.
 *  @return line with each occurrence of sub replaced by replacement
 */
public String replaceAll(String line, String sub, String repl)
```

(b) Write the FormLetter method createPersonalizedLetters. For each customer in the customers list, method createPersonalizedLetters should create then print a letter that

- replaces all occurrences of @ in lines, with the customer's name
- replaces all occurrences of & in lines, with the customer's city
- replaces all occurrences of $ in lines, with the customer's state

For example, suppose the first five lines in the form letter are:

```
Dear @,
If you buy a garden gnome you will
have the best-looking house in &,
heck, @, in the whole state of $!
@, @, @, don't delay.
```

The letter generated for a customer Joan from Glendale, California, should have these replacement lines:

```
Dear Joan,
If you buy a garden gnome you will
have the best-looking house in Glendale,
heck, Joan, in the whole state of California!
Joan, Joan, Joan, don't delay.
```

In writing method createPersonalizedLetters, you *must* use the method replaceAll that you wrote in part (a). Assume that replaceAll works as specified, regardless of what you wrote in part (a).

Complete method createPersonalizedLetters below.

```
/** Creates and prints a personalized form letter for each
 *  customer in the customers list.
 *  Postcondition: In each customer letter:
 *   - every occurrence of "@" is replaced by the customer's name;
 *   - every occurrence of "&" is replaced by the customer's city;
 *   - every occurrence of "$" is replaced by the customer's state.
 *   - A letter with the replacements is printed for each customer.
 */
public void createPersonalizedLetters()
```

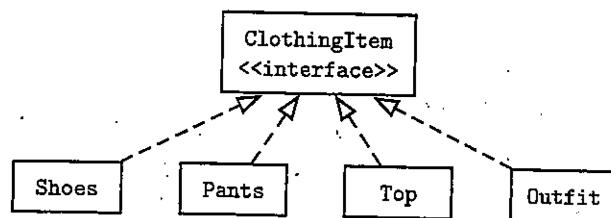
GO ON TO THE NEXT PAGE.

3. A clothing store sells shoes, pants, and tops. The store also allows a customer to buy an "outfit," which consists of three items: one pair of shoes, one pair of pants, and one top.
Each clothing item has a description and a price. The four types of clothing items are represented by the four classes `Shoes`, `Pants`, `Top`, and `Outfit`. All four classes implement the following `ClothingItem` interface.

```
public interface ClothingItem
{
    /** @return the description of the clothing item */
    String getDescription();

    /** @return the price of the clothing item */
    double getPrice();
}
```

The following diagram shows the relationship between the `ClothingItem` interface and the `Shoes`, `Pants`, `Top`, and `Outfit` classes.



The store allows customers to create `Outfit` clothing items each of which includes a pair of shoes, pants, and a top. The description of the outfit consists of the description of the shoes, pants, and top, in that order, separated by `"/"` and followed by a space and `"outfit"`. The price of an outfit is calculated as follows. If the sum of the prices of any two items equals or exceeds \$100, there is a 25% discount on the sum of the prices of all three items. Otherwise there is a 10% discount.
For example, an outfit consisting of sneakers (\$40), blue jeans (\$50), and a T-shirt (\$10), would have the name `"sneakers/blue jeans/T-shirt outfit"` and a price of $0.90(40 + 50 + 10) = \$90.00$. An outfit consisting of loafers (\$50), cutoffs (\$20), and dress-shirt (\$60), would have the description `"loafers/cutoffs/dress-shirt outfit"` and price $0.75(50+20+60) = \$97.50$.
Write the `Outfit` class that implements the `ClothingItem` interface. Your implementation must include a constructor that takes three parameters representing a pair of shoes, pants, and a top.
The code segment below should have the following behavior.

Practice Exam 3

```
Shoes shoes;
Pants pants;
Top top;
/* Code to initialize shoes, pants, and top */

ClothingItem outfit =
    new Outfit (shoes, pants, top); //Compiles without error
ClothingItem outfit =
    new Outfit (pants, shoes, top); //Compile-time error
ClothingItem outfit =
    new Outfit (shoes, top, pants); //Compile-time error
```

Write your solution below.

4. A word creation game uses letter tiles, each of which has a letter and numerical value printed on it. A partial implementation of the `Tile` class is shown below.

```
public class Tile
{
    private String letter;
    private int value;

    /** @return the value on this Tile */
    public int getValue()
    { return value; }

    /** @return the letter on this Tile */
    public String getLetter()
    { return letter; }

    //Constructor and other methods are not shown.
}
```

All tiles for the word game are called the tile set, which is represented by the `TileSet` class, whose partial implementation is shown below.

Practice Exam 3

```
public class TileSet
{
    /** tiles contains all the tiles in the word game,
     *  both used and not-yet-used.
     */
    private List<Tile> tiles;

    /** unusedSize is the number of tiles that are not yet used. */
    private int unusedSize;

    /** Determines if there are still unused tiles.
     *  @return true if all the tiles have been used; false otherwise
     */
    public boolean allUsed()
    { return unusedSize == 0; }

    /** @return the number of unused tiles in this tile set */
    public int getUnusedSize()
    { return unusedSize; }

    /** Shuffles the tiles in the tile set, and
     *  resets unusedSize to the total number of tiles in the set.
     */
    public void shuffle()
    { /* to be implemented in part (a) */ }

    /** Get an unused tile from this tile set.
     *  @return an unused tile, or null if all tiles have been used
     */
    public Tile getNewTile()
    { /* implementation not shown */ }

  //Constructors and other methods are not shown.
}
```

(a) Write the `shuffle` method for the `TileSet` class. Your method should use the following algorithm.

> for `k` starting at the end of the tiles list and going down to `1`:
>   pick a random index in `0,1,2,...,k`
>   swap the tiles at position `index` and position `k`
> Reset `unusedSize` to the number of tiles in the tile set.

Complete method `shuffle` below.

```
/** Shuffles the tiles in the tile set, and resets
 *  unusedSize to the total number of tiles in the set.
 */
public void shuffle()
```

Practice Exam 3

For parts (b) and (c) you will write methods from the `Player` class, whose partial implementation is shown below. A player in the word game has `NUM_LETTERS` tiles in front of her. After she makes a word, she helps herself to unused tiles to maintain `NUM_LETTERS` tiles, if possible.

```
public class Player
{
    /** NUM_LETTERS is the number of letter tiles a player should
     *  have (if tiles have not yet all been used) at the start of
     *  her turn. */
    public static final int NUM_LETTERS = < some integer >;

    /** playerTiles is the list of tiles for this player. */
    private List<Tile>playerTiles;

    /** Adds a sufficient number of unused tiles from tileSet t
     *  to playerTiles so that this player has NUM_LETTERS tiles.
     *  If there are insufficient unused tiles, the player should
     *  take all of the remaining available tiles.
     *  Precondition: playerTiles.size() < NUM_LETTERS.
     *  Postcondition: playerTiles.size() <= NUM_LETTERS.
     *  @param t the tile set for the word game
     */
    public void replaceTiles(TileSet t)
    { /* to be implemented in part (b) */ }

    /** Returns the score a player receives for using tiles from
     *  his playerTiles at his turn. The score is the sum of values
     *  on each tile used. Indexes of tiles used are contained in
     *  the indexes array. If index[0] is -1, the player
     *  has used no tiles at his turn and the method returns a
     *  score of 0. If the player uses all of the tiles in
     *  playerTiles, a bonus of 20 points is added to his score.
     *  @param indexes the array of positions of tiles in
     *     playerTiles that the player uses at his turn
     *  Precondition:
     *   - playerTiles contains NUM_LETTERS tiles.
     *   - indexes[0 .. n] is sorted in increasing order,
     *     n < NUM_LETTERS.
     */
    public int getWordScore(int[] indexes)
    { /* to be implemented in part (c) */ }
}
```

(b) Write the `Player` method `replaceTiles`. This method should, if possible, add unused tiles to the player's `playerTiles` list, until `playerTiles` contains `NUM_LETTERS` tiles. If there are insufficient unused tiles in the tile set, the player should take all of the remaining tiles.

Practice Exam 3

Complete method `replaceTiles` below.

```
/** Adds a sufficient number of unused tiles from tileSet t
 *  to playerTiles so that this player has NUM_LETTERS tiles.
 *  If there are insufficient unused tiles, the player should
 *  take all of the remaining available tiles.
 *  Precondition: playerTiles.size() < NUM_LETTERS.
 *  Postcondition: playerTiles.size() <= NUM_LETTERS.
 *  @param t the tile set for the word game
 */
public void replaceTiles(TileSet t)
```

(c) Write the `Player` method `getWordScore`. This method returns the total of the values of tiles in `playerTiles` whose positions are indicated in the `indexes` parameter. If `indexes` contains `{0,1,4}`, this means that the player will use the tiles at positions 0, 1, and 4 in his `playerTiles` list at his turn, and his score will be the sum of values of those tiles. If the player uses all of his tiles at his turn, a bonus of 20 points is added to his score. If the only value in the `indexes` array is -1, this means that the player passes at his turn, and `getWordScore` should return a value of 0.

For example, suppose `NUM_LETTERS` is 5, and `playerTiles` has the following state before the method call.

0	1	2	3	4
"O" 1	"C" 3	"V" 4	"E" 1	"N" 1

State of indexes array	Result of getWordScore(indexes)
{0,2,3,4}	7
{0,1,2,3,4}	30
{0,1,4}	5
{-1}	0

Complete method `getWordScore` below.

```
/** Returns the score a player receives for using tiles from
 *  his playerTiles at his turn. The score is the sum of values
 *  on each tile used. Indexes of tiles used are contained in
 *  the indexes array. If index[0] is -1, the player
 *  has used no tiles at his turn and the method returns a
 *  score of 0. If the player uses all of the tiles in
 *  playerTiles, a bonus of 20 points is added to his score.
 *  @param indexes the array of positions of tiles in
 *     playerTiles that the player uses at his turn
 *  Precondition:
 *   - playerTiles contains NUM_LETTERS tiles.
 *   - indexes[0 .. n] is sorted in increasing order,
 *     n < NUM_LETTERS.
 */
public int getWordScore(int[] indexes)
```

END OF EXAMINATION