2. (a)
```
public String replaceAll(String line, String sub, String repl)
{
    int pos = line.indexOf(sub);
    while (pos >= 0)
    {
        line = line.substring(0, pos) + repl +
                line.substring(pos + sub.length());
        pos = line.indexOf(sub);
    }
    return line;
}
```

(b)
```
public void createPersonalizedLetters()
{
    for (int i = 0; i < customers.size(); i++)
    {
        List<String> tempLines = makeCopy();
        Customer c = customers.get(i);
        for (int j = 0; j < tempLines.size(); j++)
        {
            tempLines.set(j,
                replaceAll(tempLines.get(j), "@", c.getName()));
            tempLines.set(j,
                replaceAll(tempLines.get(j), "&", c.getCity()));
            tempLines.set(j,
                replaceAll(tempLines.get(j), "$", c.getState()));
        }
        writeLetter(tempLines);
    }
}
```

NOTE

- In part (a), each time you encounter `sub` in `line`, you simply create a new line that concatenates the "before" substring, the replacement, and the "after" substring. This guarantees termination of the loop: Eventually `sub` won't be found in `line` because all occurrences have been replaced, and `line.indexOf(sub)` will return `-1` (`sub` not found in `line`).
- In part (b), you need a nested loop: for each customer, loop through all the lines and do the replacements.
- In part (b), one of the tricky lines of code is

```
List<String> tempLines = makeCopy();
```

You need a fresh, unchanged copy of `lines` for each customer. If, by mistake, you use the line

```
List<String> tempLines = lines;
```

then `tempLines` and `lines` will be the same reference, so any changes to `tempLines` will also be made to `lines`, and the second (and all subsequent) customers won't have a fresh copy of `lines` with the tokens. Instead, `lines` will contain the first customer's information.

3.
```
public class Outfit implements ClothingItem
{
    private Shoes shoes;
    private Pants pants;
    private Top top;

    public Outfit (Shoes aShoes, Pants aPants, Top aTop)
    {
        shoes = aShoes;
        pants = aPants;
        top = aTop;
    }

    public String getDescription()
    {
        return shoes.getDescription() + "/" + pants.getDescription()
                + "/" + top.getDescription() + " outfit";
    }

    public double getPrice()
    {
        if (shoes.getPrice() + pants.getPrice() >= 100
                ||shoes.getPrice() + top.getPrice() >= 100
                ||top.getPrice() + pants.getPrice() >= 100)
            return 0.75 * (shoes.getPrice() + pants.getPrice() +
                    top.getPrice());
        else
            return 0.90 * (shoes.getPrice() + pants.getPrice() +
                    top.getPrice());
    }
}
```

NOTE

- To access the price and descriptions of items that make up an outfit, your class needs to have variables of type Shoes, Pants, and Top.

4. (a)
```
public void shuffle()
{
    for (int k = tiles.size() - 1; k > 0; k--)
    {
        int randIndex = (int) (Math.random() * (k + 1));
        Tile temp = tiles.get(k);
        tiles.set(k,  tiles.get(randIndex));
        tiles.set(randIndex, temp);
    }
    unusedSize = tiles.size();
}
```

(b)
```
public void replaceTiles(TileSet t)
{
    int numTiles = NUM_LETTERS - playerTiles.size();
    if (numTiles <= t.getUnusedSize())
    {
        for (int i = 1; i <= numTiles; i++)
            playerTiles.add(t.getNewTile());
    }
    else
    {
        for (int i = 1; i <= t.getUnusedSize(); i++)
            playerTiles.add(t.getNewTile());
    }
}
```

Alternatively

```
while (NUM_LETTERS > playerTiles.size() && !t.allUsed())
    playerTiles.add(t.getNewTile());
```

(c)
```
public int getWordScore(int[] indexes)
{
    if (indexes[0] == -1)
        return 0;
    int total = 0;
    if (indexes.length == NUM_LETTERS)
        total += 20;
    for (int i = 0; i < indexes.length; i++)
    {
        total += playerTiles.get(indexes[i]).getValue();
    }
    return total;
}
```

NOTE

- In part (a), the line

  ```
  int randIndex = (int) (Math.random() * (k + 1));
  ```

  returns a random integer in the range `0,1,2,...,k`.
- In part (b), there are two things to check:
  1. How many new tiles does the player need?
  2. Are there enough unused tiles available?
- In part (c), notice that indexes[i] are the positions of the tiles whose scores you need to access.