Question 1.1

The comparison to string constants is worrisome --- a typo in this or other code could be hard to detect, and what if the `student discount'' is renamed an` academic discount?” Refactor the co
de so that typos are a non-issue.

Answer

```
public static final String CUSTOMER_TYPE_STUDENT = "student";
public static final String CUSTOMER_TYPE_EMPLOYEE = "employee";
```

Question 1.2

`customerTypeDiscount` and `dayOfWeekDiscount` can contain arbitrary strings, even though they can only take a restricted set of values. What happens if the calling code passes in `Weekend` as a day-of-week discount, or if the programmer adds a `veteran discount` but forgets to update this code? Refactor the day-of-week and customer-type discounts so that they can only contain valid days of week or customer types.

Answer

```
public class Customer {
  public CustomerType getCustomerType() { ... }
}
public enum CustomerType { STUDENT("student"), EMPLOYEE("employee") }

public enum DayOfWeek { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
 ↪  FRIDAY, SATURDAY }
// (Or use java.time.DayOfWeek)
```

Question 1.3

This API allows you to apply a discount to an item that shouldn’t get it. How would you modify the API to prevent that?

Answer

Option 1: applyDiscount now only conditionally applies discount; does its own discount checking.

```
  private boolean doesDiscountApply(Customer c, Item item){
    // same as before
  }
  public double applyDiscount(Customer c, Item item, double price){
    return doesDiscountApply(c, item) ?
        price * (1 - discountPercent):
        price;
  }
```

Option 2: Use Tokens. doesDiscountApply returns an optional "DiscountApplier" object. Either make applyDiscount require a DiscountApplier as an argument, or move applyDiscount to be a method of DiscountApplier

```
public class Discount {
    public Optional<DiscountApplier> doesDiscountApply(Customer c,
     ↪  Item item);
    public double applyDiscount(DiscountApplier applier, double
     ↪  price)
}
public class DiscountApplier {}
```

Question 1.4

It's intended that a discount can only be one of the three types. How would you redesign this code so that `doesDiscountApply` contains no conditionals?

Answer

Option 1:

```
public class CustomerTypeDiscount extends Discount { ... }
public class ItemNameDiscount extends Discount { ... }
```

Option 2:

```
public class Discount {
  private interface DiscountChecker {
    public boolean doesDiscountApply(Customer c, Item item);
  }

  private static class CustomerTypeDiscount implements
   ↪  DiscountChecker { ... }
  private static class ItemNameDiscount implements DiscountChecker {
   ↪  ... }
  private static class DayOfWeekDiscount implements DiscountChecker {
   ↪  ... }

  public static Discount createCustomerTypeDiscount(
   ↪  discountPercentage: double, customer: Customer) { ... }
  public static Discount createItemNameDiscount( discountPercentage:
   ↪  double, item: Item) { ... }
  public static Discount createDayOfWeekDiscount( discountPercentage:
   ↪  double, day: DayOfWeek) { ... }

  DiscountChecker discountChecker;

  public boolean doesDiscountApply(Customer c, Item item) {
      return discountChecker.doesDiscountApply(c, item);
  }

}
```

Question 1.5

Bonus: With the current implementation, a day-of-week discount can't be tested without waiting until that day. How would you modify this program to make day-of-week discounts unit-testable?

Answer

Dependency-injection.

Question 2.1

The Facebook codebase of 2010 is roughly organized into layers. The website layer displays user information in webpages, and provides end-user functionality like the Like button. The middle layer organizes data and provides operations on it, like like finding a user's top posts. The data layer organizes requests to the database. In the design above, all privacy checks happen in the web layer. How would you redesign the code so that all privacy checks happen at the data layer?

Answer

The core idea to solve this privacy issue is to make the data layer only return information that has already been checked for privacy concerns. By moving all the privacy checks to the data layer the rest of the program doesn't have to concern itself with the privacy policy, and thus the likelihood of privacy leaks is reduced. This is the same reasoning that's needed to get the last round of the Todo exercise of week 2 right.

Another part of the solution is to get the necessary info into the data layer to make the request. Facebook did it by threading a ViewerContext parameter through all methods. So a rough draft of the solution after the refactor would look something along these lines:

```
// no need to check since getPhotos only returns photos viewable for
 ↪  the context.
def listPhotos(user, viewerContext):
  for photo is viewerContext.getPhotos(user, db):
    displayPhoto(photo)
```