Generic Data Structures

1. Explain what an ArrayList is, how it differs from a normal array, and give some examples of where it may be useful.
    - An array has: finite length, requires manual resizing, requires effort to add/remove elements
    - An `ArrayList` handles resizing, provides insert/remove/get/modify methods, toString(), cannot be indexed
        - a class with an array as an instance variable
        - cannot store primitive data types
    - e.g. you need to maintain a list of user input, with unknown size in advance
    - e.g. you need to sort, maintain, update a list of elements
2. Explain what a HashMap is, and give some examples of where it may be useful.
    - key-value pair map
    - maintain a dictionary of items with particular values
    - map between an item ID and the item itself, facilitating lookup in efficient data structures

Design Patterns

1. What is a design pattern?
    - description to a recurring problem in software design
    - prevent reinventing the wheel by using a design pattern
2. Explain what the following design patterns are, and when they are useful.

(a) Singleton

- ensures a class has only one instance with a global point of access
- useful when there's a need to have a single instance of a class

(b) Strategy

- separate generic algorithm from detailed implementation via delegation

(c) Template

- separate generic algorithm from detailed implementation via inheritance

3. How does the Template pattern differ from the Strategy pattern?

- Template pattern uses inheritance, while Strategy pattern uses delegation, i.e. you delegate the algorithm to another object. The context class has an associative relationship with the object that will conduct the algorithm

Exceptions

1. What are the types of errors in Java?
   - syntax error
   - runtime error
     - divide by zero
     - out of bounds
2. What are some different ways to handle errors that may occur during a program's execution?
   - defensive programming, explicit guarding (e.g. that a number is not 0 to prevent divide by zero errors)
   - use exceptions to catch error states, then recover or gracefully exit
3. What are exceptions, and what are their advantages over other approaches?
   - exceptions are error states created by runtime error/object created by Java to represent such error states
   - improves code readability, we don't need to handle every possible error case. There may be actual failures that you can't guard against (code outside your control for example, or perhaps the network goes down)
4. When should we use exceptions?
   - when you foresee errors occurring e.g. accessing files (the file may have been deleted)

Design

1. Implement a method that processes an `ArrayList<String>`, and returns a single `String`, which is a comma separated list of all the items in the input that contain only a single word.

```java
public void processArrayList(ArrayList<String> list) {
    ArrayList<String> singleWords = new ArrayList<String>();
    for (String item : list) {
        if (item.split(" ").length == 1) {
            singleWords.add(item);
        }
    }
```

```
8
9       return singleWords.toString();
10  }
```

2. What are some error states that could happen when parsing a CSV file? Write some short Exception classes you might like to create if you were implementing a CSV reader.

    - FileNotFoundException
    - IOException
    - CSVParseException: missing/extra columns in a row

3. We are opening a bar to profit off the influx of people going out from June 1 onwards. People who attend the bar can either be friends or relatives of the owners, members of the bar club, or just a regular customer. Friends and relatives get a 99% discount, members get a 10% discount and regular customers pay full price. Using the strategy pattern, design a simple system to handle these discount variations. Once you've done this, implement a simple bar simulation in your code. Use a Map<String, Double> to store the drink names and their costs. Use two List's to store the names of family/friends, and members. When the simulation is run, the program should prompt for the customer's name, and then their drink. The appropriate pricing strategy would be applied and the adjust cost of the drink printed to the console. Bonus: Throw a custom exception DrinkNotFoundException if the user enters a drink name that isn't in your drinks list.

BarProgram.java

```
1  class BarProgram {
2      public static void main(String[] args) {
3
4          Bar bar = new Bar();
5          Scanner scanner = new Scanner(System.in);
6          while (scanner.hasNextLine()) {
7              System.out.print("Name: ");
8              String name = scanner.nextLne();
9              System.out.print("Order: ");
10             String drink = scanner.nextLine();
11             System.out.format("$%0.2f", bar.getPrice(name, drink));
12         }
13     }
14 }
```

Bar.java

```
1  public class Bar {
2      private final Map<String, Double> menu = new HashMap<>();
3      private final List<String> memberNames = new ArrayLIst<>();
4      private final List<String> familyNames = new ArrayLIst<>();
```

```
5
6       public Bar() {
7           menu.put("water", 0.00);
8           menu.put("soft drink", 6.00);
9
10          memberNames.add("alice");
11          memberNames.add("bob");
12          familyNames.add("shanika");
13
14      }
15
16      private DiscountStrategy getStrategy(Stringname) {
17          if (familyNames.contains(name.toLowerCase()) {
18              return new FamilyDiscountStrategy();
19          } else if (memberNames.contains(name.toLowerCase()) {
20              return new MemberDiscountStrategy();
21          } else {
22              return new RegularDiscountStrategy();
23          }
24      }
25
26      public double getPrice(String drink) {
27          if (!menu.containsKey(drink)) {
28              throw new DrinkNotFoundException(drink);
29          }
30          double price = menyu.get(Drink);
31      }
32
33  }
34
35  class DrinkNotFoudException extends Exception {
36      public DrinkNotFoundException(string drink) {
37          super("unknown drink:" + drink);
38      }
39  }
```

DiscountStrategy.java

```
1   public interface DiscountStategy {
2       public abstract double discount(double price);
3   }
4
5   class MemberDiscountStrategy implements DiscountStrategy {
6       @Override
7       public double discount(double price) {
8           return
9       }
10  }
```

Implementation

Continue working on Project 2.

Assessable Problem

Avnac is a new graphics design platform that allows users to create designs online. One part of their tech stack is a server to handle incoming requests from their clients (phone app, web app, third party integrations) and retrieve content from their database. Very recently, a pandemic has struck that has (indirectly) resulted in an increase in use of Zoom virtual backgrounds. More and more people are using Avnac to create designs that provide very short-lived and mild comedic relief in these 'strange and uncertain times'. In an attempt to make their request handling more efficient, Avnac's unpaid intern has developed (in their opinion) a state of the art caching mechanism1 to act as a middleman between the request-handling server and the design database. You, the senior engineer of Avnac, are nearly satisfied with this implementation of a cache but have noticed a small error in the intern's implementation: anyone can create multiple instances of the cache! We only want one cache to ever exist in our implementation, so this must be fixed.

Your Task:

Restructure the AvnacDesignCache class to follow the Singleton design pattern. You do not need to worry about caching and you do not need to add any additional classes to solve this problem. The singleton instance is to be accessed through a method called getInstance. Once you have succesfully restructured the class to follow the Singleton pattern, the sole unit test in the testing class should pass.

Package

The assessable problem package provides you a nearly-complete AvnacDesignCache class that does not follow the Singleton design pattern, an empty Design class (that must remain empty), and a testing class AvnacDesignCacheTest so that you can verify your solution.

Submission

You will submit your solution to your workshops repository. At the very least, you should maintain the following structure:

```
username-workshops
workshop-8
    src
        AvnacDesignCache.java
        Design.java
```

As always, you can use the example repository here as a reference. This assessable question is due on Friday the 29th of May at 11:59pm. This is a rather straightforward problem testing a very basic understanding of the Singleton pattern, it shouldn't take too long if you've learned the content.

1 This is a considerably bad cache implementation, as it doesn't perform cache invalidation or implement any eviction policies.