Quiz 5

Solutions

Study Guide / Practice Problems

Topics:

- Comparators/Lambdas/Key Extractors
    - Resources:
        - Modules 18, 19, 20
        - Lab 6
    - **You should know:**
        - How to sort using comparators/comparable interface and know the difference
        - Know what is/how to make a functional interface with lambdas and method reference operators
- Inheritance (with Abstract Classes):
    - Resources:
        - Modules 15, 16, 17
    - **You should know:**
        - How to work with abstract classes (lab 5 and project 3)
        - How to pick whether to make a method abstract, overridden with super calls, pulled up with abstract helper methods
        - How to pick which abstract classes to make given base code
        - Difference between abstract classes / interfaces / concrete classes and when to use each

Study Material:

- Listed modules, this practice quiz, lab 5, lab 6, project 3

Practice problems

1. Functional Interfaces/Lambdas

1. What is the difference between the Comparable and Comparator interfaces?

Comparable has the compareTo() method and is implemented by the class that is to be sorted. Comparator has the compare method and is outside the class that is to be sorted, but can be passed separately into the sort method (following the strategy design pattern). Comparable gives the natural sort order.

2. What is a functional interface?

An interface with only one abstract method, so it can be implemented with a lambda.

3. Complete the following code given the fact that the Professor class is not Comparable and has the getters: `hasTenure()` and `getName()`

```
public class Test {
      public static void main(String[] args) {
      ArrayList profs = new ArrayList<>();
      profs.add(new Professor(500000, false, "Julie", 0, 2));
      profs.add(new Professor(300000, false, "Paul", 0, 2));
      profs.add(new Professor(250001, true, "Phil", 0, 2));
      profs.add(new Professor(250000, true, "Hugh", 0, 2));
      profs.add(new Professor(200000, false, "Kurt", 0, 2));

      // TODO:Write code to sort an ArrayList of Professor objects in:
      // descending alphabetical order,
      // with all of the tenured professors coming first.

      // Below: Several of many ways of creating the proper
      comparator.

      // 1. ------------ Using Lambdas -----------------------
      Comparator<Professor> comp1 = (p1, p2) -> {
                  if (p1.hasTenure() && !p2.hasTenure()) return -1;
                  if (!p1.hasTenure() && p2.hasTenure()) return 1;
                  return 0;
            };
```

```
// OR
Comparator<Professor> comp1 = (p1, p2) ->
((Boolean)p1.hasTenure().compareTo(p2.hasTenure());

Comparator<Professor> comp2 = (p1, p2) ->
                      p2.getName().compareTo(p1.getName());
Comparator<Professor> compFinal = comp1.thenComparing(comp2);

// 2. -------- Using key extractors ------------------------
Comparator<Professor> comp1 =
          Comparator.comparing(Professor::hasTenure).reversed()
     ;
Comparator<Professor> comp2 =
          Comparator.comparing(Professor::getName).reversed();
Comparator<Professor> compFinal = comp1.thenComparing(comp2);

 // 3. ------------ A single lambda
--------------------------- Comparator<Professor> compFinal =
(p1, p2) -> {
                if (p1.hasTenure() && !p2.hasTenure()) return
          -1;
                if (!p1.hasTenure() && p2.hasTenure()) return
          1;
                return p2.getName().compareTo(p1.getName());
          };

// To do the actual sorting...
Collections.sort(profs, compFinal);
for (Professor p : profs) { System.out.println(p.getName()); }
```

4. Create a **static** function `usePredicate` that takes in a **Predicate** (like sort takes in a Comparator) and a **Student s** (where the student class has getGpa() and getLastName()). The function should print "yay" if the predicate is true when applied to the passed in student and "no!" otherwise. All the function does is print - it doesn't return anything.

Look up Predicate in the javadocs if needed.

```
public static void usePredicate(Predicate<Student> pred, Student
s)
{
     if(pred.test(s))
```

```
            System.out.println("yay");
      else
            System.out.println("no!");
}
```

5. Assuming you are in the same class where your function from number 4 is, and assuming the variable `student1` exists and is of type Student:
Write the line to call your method, passing in a predicate that returns true if GPA is > 3.0 and passing in `student1`

```
usePredicate(s->s.getGpa() > 3.0, student1);
```

6. Be able to do problems like 4 / 5 with Function interface and Consumer interface as well.

2. Abstract Classes

Given the following code for **Food**, refactor it such that common functionality/data is lifted into abstract parent classes. Be sure to use data encapsulation principles (*do not use protected variables for this example - make your instance variables private with public getters if needed)*.

Do not assume that any private instance variable will need a getter method. *Only add getter/setter methods for a variable if needed.*

Show your answer as a UML diagram (but also be prepared to describe your code with words or with code). Explain your choices after.

```
public class Food {
      public FoodType kind;
      public int calories;
      public double flavor;
```

```
  public Color frosting;


  public Food(FoodType kind, int calories,
              double flavor) {
    this.kind = kind;
    this.calories = calories;
    this.flavor = flavor;
  }
public void consumed(){
    switch (kind) {
        case CAKE:
           consumedCake();
           break;
        case APPLE:
           consumedApple();
        break;
        case RICE:
           consumedRice();
        break;
  }


  public void consumedCake() {
              /* Assume flavor and calories variables used,
              but not changed.
              Code not shown - assume unique*/
   }


  public void consumedApple() {
              /* Assume flavor and calories variables used,
              but not changed.
              Code not shown - assume unique*/
   }

  public void consumedRice(){
              /* Assume flavor and calories variables used,
              but not changed.
              Code not shown - assume unique*/
   }
```

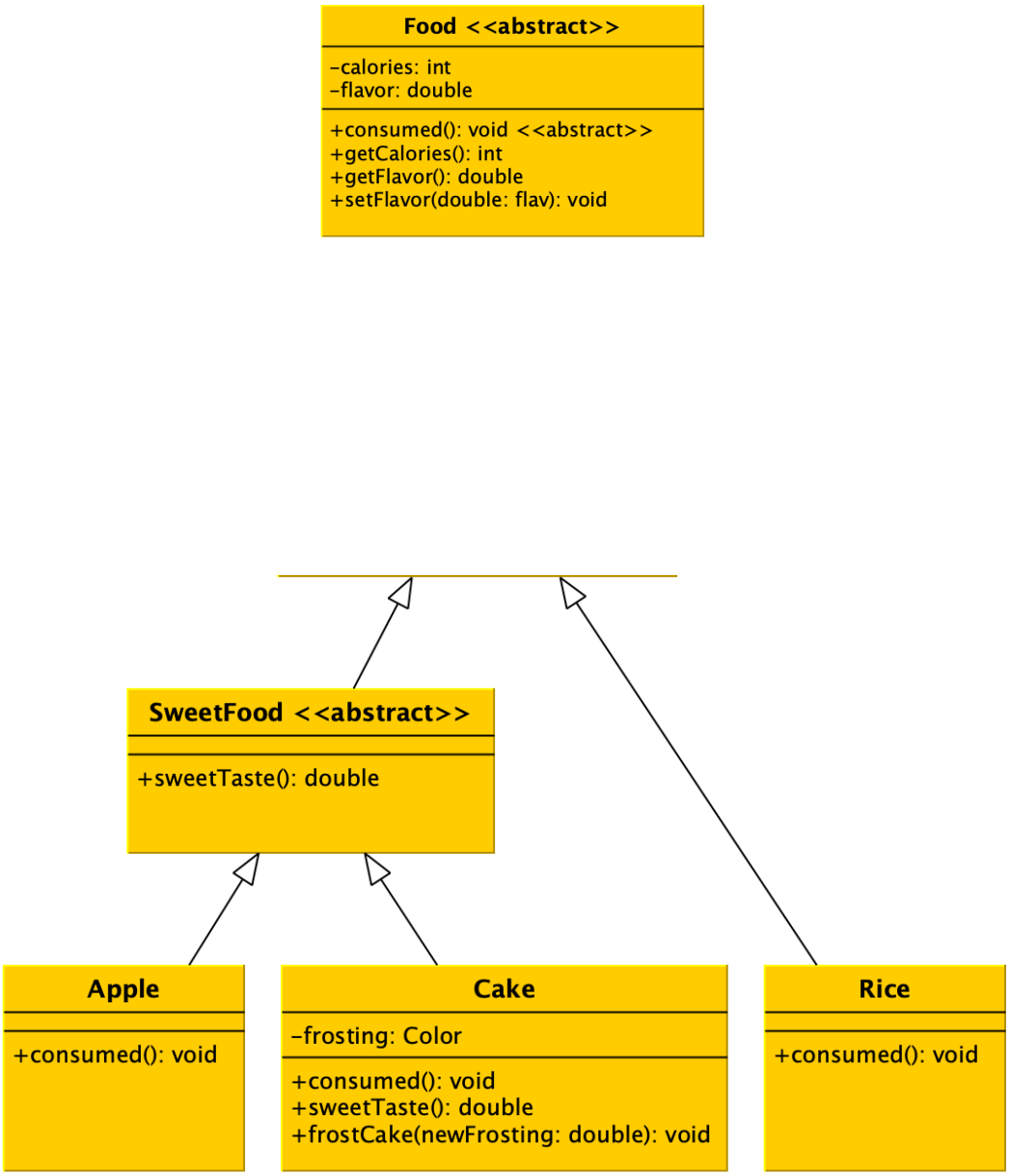
```
    public double sweetTaste() {
        switch(kind) {
          case RICE:
            throw new UnsupportedOperationException();

           case CAKE:
            frostCake(Color.pink);
        }
        flavor += 1;
        return flavor;
    }

    public void frostCake(Color newFrosting) {
        frosting = newFrosting;
        /* rest of code not shown */
    }

}
```

Solutions on next page



Your answer should match this exactly (aside from class names..you could have named SweetFood or Food something different. Also, you could have chosen to remove frostCake and placed that code in the overridden sweetTaste method. Everything else should be identical).

Food is abstract with the abstract method consumed. Each class needs this method, but each is unique so it is abstract. The two variables are used by

each subclass in their consumed method so they are shared, private, and need getters. Only flavor needs a setter since it is changed in sweetTaste().

SweetFood is abstract because we never seem to want to instantiate it. Also, consumed isn't implemented yet so it is still inherited and abstract. sweetTaste method is NOT abstract because there is shared code between all versions which can be pulled up.

Apple just needs to inherit sweetFood exactly as is and it also needs to implement it's consumed method.

Cake needs to also implement consumed, but needs to override sweetTaste (which is why it is included in its UML). It needs to add the cake specific code and then can call super.sweetTaste() for the common code. In this new code, the cake needs to access the frosting variable. frostCake can be left as its own method or the code can be moved in the overridden sweetTaste() method.

Rice has the same reasoning as Apple, except it doesn't need to inherit sweetTaste.