iPizza Manager CSS 162
Course Project
CSS 162: Programming Methodology
Due Finals Week
Rob Nash, Senior Lecturer

Level of Challenge: Hard.

Number of classes/files to submit: 10-30

Number of inheritance Hierarchies: 3 complete hierarchies involving Shapes, Ingredients, & Pizzas.

Files to Submit:

- The Pizza Class
- The PizzaManager Subclass (named "MyPizzaManger")
- Ingredient Class Hierarchy (13 Classes)
- The PizzaException Class
- The ArrayList Class used to hold Pizzas
- Support Classes including Fraction, Shape, Money, Color, ….

Summary

In this assignment, we'll tie together multiple software techniques we've learned throughout the quarter. We'll reuse our Fractions, Shapes, and other classes as well as our data structures to build a fully functioning pizza sales simulator program that would power a vending machine similar to what's pictured. Be sure to build each method using the name, arguments, and return value as indicated in each class. For example, do not rename the Fraction Class to MyFraction or the "public void eatSomePizza(Fraction f)" to "private boolean reducePizza(Object o)". For a complete execution, see the sample output at the end of this document.

Getting Started: Gathering Classes To Reuse

These classes we will assume you have standing by to use in this final project. They are taken from your previous homeworks and labs, and can be recreated from scratch if you have lost them.

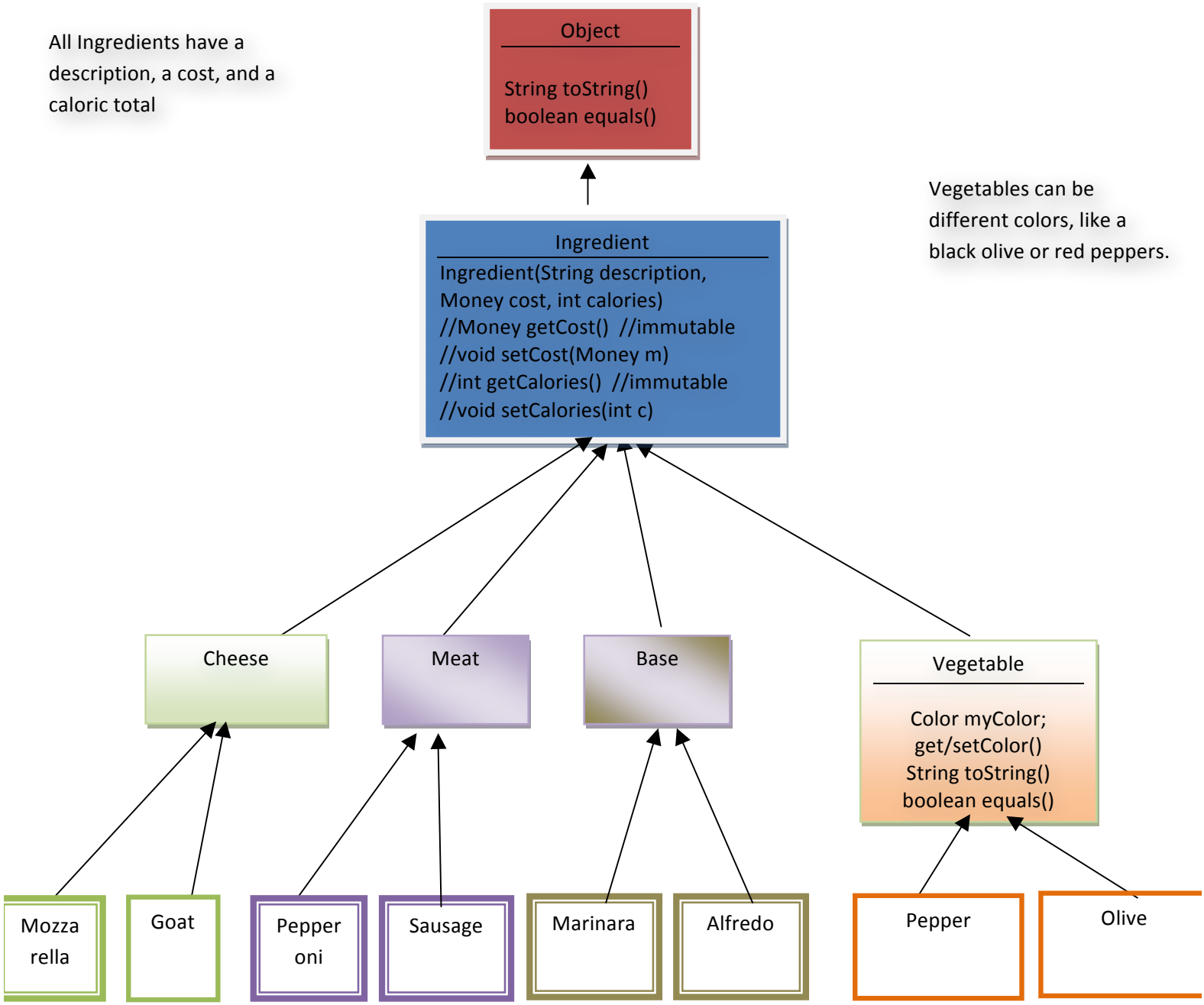
- Make a new project in Eclipse and put the files mentioned below in this new project
    - Also download the files for this HW and put them in this same project
- Find or create a **Fraction** class that manages a numerator and denominator with reduction.
    - Make your numerator and denominator final.
    - See the previous assignments for the Fraction's interface (ie, constructors, copy constructors, etc)
- Find or create your **Money** class, to be used to represent costs for **Ingredients** and **Pizzas**
- Find or create the **Shape** hierarchy from a previous lab or homework.
    - Your **Shape** superclass should manage x,y, colors, etc. just as in the labs
        - See the Shape and PolyDemo lab or Savitch text for more information.
    - You should have two subclasses for managing **Pizza** Shapes: **Circle** and **Square**
- Find or use Java's **Color24** class (called java.util.Color) for use in tracking **Vegetable** colors.
    - Note that this is the only class you use one of Java's classes as a substitute for; all others must be a class of your own design.
- Find or create your **ArrayList** that holds Comparable Objects and is dynamically resizable
    - Modify this **ArrayList** so it uses generics so that we can build an **ArrayList**<**Pizza**>.
- Find your **LinkedListException** and copy it over to a new class called **PizzaException**.
    - Throw these in place of standard **RuntimeExceptions** throughout your software.
- Find or build a **Time** and **Date** class for use when building a **TimeStamp** Class.
- Find your code for sorting (QuickSort) & binary search, as you will need those algorithms for use here.

Invariants

1. When a Pizza is made, it starts with 100% of it's area left (so a Fraction of 1/1)
2. A Pizza's remaining ratio must be between (0,1] excluding 0 and including 1.
3. When a Pizza's fractionRemaining reaches 0, that Pizza is removed from the list of Pizzas
    a. If a Pizza's remaining fraction is less than 0, throw an exception as this is an error case
    b. If a Pizza's remaining fraction is equal to 0, throw an exception that is caught by your Manager and remove that eaten pizza from the list.
4. A Pizza's price is the sum of its ingredients' costs.
    a. Thus, calling addIngredient() on a Pizza changes the cost of the pizza.
5. A Pizza's calorie count is the sum of its ingredients' calories.
    a. So, calling addIngredient() on a Pizza changes the calorie total for the pizza.
    b. This is a serving size, and so the pizza's calories will never change as the pizza is being eaten – this is the per slice "serving size".
6. Ingredient is Immutable (all data items are final)
7. Fraction is Immutable, just as we have done before.
8. Fraction implements Comparable
9. Shape will implement Cloneable but define the method as abstract
10. Adding ingredients will add the price of a Pizza
11. Reducing the remaining Fraction of pizza left will reduce the price of the Pizza accordingly
    a. Square and Circle will implement the clone methods to override the abstract superclass clone
12. All Pizzas have a TimeStamp that indicates when they were created.
    a. You can use the current date and time when a Pizza is built, or use random values for the time and date the pizza was made.

The Ingredient Hierarchy

All Ingredients have a description, a cost, and a caloric total



Vegetables can be different colors, like a black olive or red peppers.

All of the bottommost subclasses (leaves) will contain constructors to provide descriptions, costs, and calories to the superclass.

All of the interior subclasses will contain constructors to provide descriptions, costs ,and calories to the superclass.

The Vegetable class requires additional methods as indicated in the UML diagram.

Top-Down Design & Stepwise Refinement

In each section, we'll build components of our Pizza Simulator one piece at a time, building each component in steps. By refining each component in stages, we can incrementally build logic and test it in isolation as we progress, before combining the components into our complete Pizza solution. We'll often start by examining an inheritance hierarchy and realizing that hierarch in Java code by building each class one at a time, starting with parent classes. This approach essentially captures the "big picture" first (the top levels or classes in our inheritance hierarchy), and fills in the specific details as you flesh out classes farther and farther down the hierarchy. We can also build each class using a "Top-Down" approach where we first define each method we are to have as empty stubs and verify this compiles with the driver, and then fill in the missing details in each function as you progress. In each of the examples, you're starting with a broad picture that lacks details just like the first few iterations of your Top-Down Stepwise Refinement exercises using pseudocode. As build each method and class, you move your overall solution ahead one or more steps while practicing software design techniques called **Top-Down Design** and **Stepwise Refinement.**

1. Start by downloading the **PizzaManager** class and **PizzaComparable** interface to get started; read the code and comments.

The PizzaManager Superclass & MyPizzaManager Subclass

Start in the MyPizzaManager.java subclass, provided for you. A **PizzaManager** keeps track of pizzas over the course if it's existence. When a Pizza is first made, it has 100% (1/1) of it's surface area available for eating. As a pizza is eaten, fractions are removed from the ratio of pizza remaining, until the pizza is gone and the ratio reaches 0. **PizzaManagers** manage all things Pizza; so if you want to build new pizzas in bulk, display pizzas, sort or rearrange pizzas, and eat pizzas, then your code to test those features goes here. The **PizzaManager** interface is completely defined for you both in the class itself and also indicated below. In other classes, you'll have to define your own methods accordingly, but here, we'll want to be very specific so as to exactly override the methods you need to complete. You'll know you've overridden the method when it no longer throws the RuntimeException: "method not yet implemented!".

PizzaManager Interface

- All of the methods in the PizzaManager have been defined for you but don't work – they simply throw exceptions; your job is to simply to fill in those methods by overriding them in your **MyPizzaManager** subclass.
    - You can add additional helper methods ***but should not remove or rename*** any of the functions provided for you in this class.

MyPizzaManager Subclass

This is where all of your code will go to build, eat, sort and just generally manage pizzas. You will need to override superclass methods that currently just throw exceptions. When you are done, your software should output results that match the sample execution provided in this document. Here is what it looks like when you're just getting started and haven't overridden any methods yet:

Sample Execution

```
BAD P!ZZAS
----------------------
Welcome to PizzaManager
----------------------
(A)dd a random pizza
Add a (H)undred random pizzas
(E)at a fraction of a pizza
QuickSort pizzas by (P)rice
QuickSort pizzas by (S)ize
QuickSort pizzas by (C)alories
(B)inary Search pizzas by calories
(L)inear search pizzas by day
(R)everse order of pizzas with a stack
Remove (D)ay-old pizzas
(Q)uit

Adding one hundred random pizzas to the ArrayList<Pizza>.
Exception in thread "main" java.lang.RuntimeException:
addRandomPizza() not yet implemented in MyPizzaManager subclass!
      At…
```

PizzaManager Methods to Override

You will know you've finished your MyPizzaManager work when you can try out each option in the console menu and none of them throw exceptions but instead invoke your overridden subclass code for each method. Start with the following methods to add a single pizza to your ArrayList of pizzas, and to display your list of pizzas

- protected void addOnePizza()
    - Build a pizza that randomly chooses between pie shape (circular or deep dish square) and adds some random ingredients.
- protected void displayAllPizzas()

- This should display the set of pizzas you have. At first, this will be empty, but after you call addOnePizza(), you should see the output from your ArrayList's toString() reflect this new addition.

STEPwise Approach for MyPizzaManager Subcass

1. Start in the empty class called **MyPizzaManager,** *which is provided for you*. Override the following methods to stop them from throwing exceptions and crashing your code in the superclass.
    a. In the data section, define an **ArrayList**<**Pizza**> to hold your pizzas.
    b. In the methods section, define and implement the following methods:
        i. Write a function to list all **Pizzas** in their current order in your **ArrayList**
            1. protected void displayAllPizzas() {
        ii. Write a function that builds a Scanner and returns a single character that was typed.
            1. protected char getNextChar()
                a. Currently, the superclass function just returns a random char
        iii. Write a function that builds a Scanner and returns the next integer typed at the console
            1. protected int getNextInt()
                a. Currently, the superclass version just returns a random int.
        iv. Write a function to subtract a fractional amount from a Pizza
            1. protected void eatSomePizza()
                a. This function subtracts the amount from the remaining pizza.
                    i. To accomplish this, ask the user for the fractional amount to eat and the index of the pizza to eat
                    ii. If the ratio reaches zero, throw a PizzaException in the Pizza class and catch it in the MyPizzaManager class so this pizza is removed
                    iii. If the ratio is negative, throw a PizzaException as this is an error case.
        v. Build a function that sorts **Pizzas** in order of greatest calories first.
            1. Notice you're sorting primitives here.
            2. protected void quickSortByCalories() {
                a. To finish this method, you'll need to add the following methods to **ArrayList:**
                    i. public int size()
                    ii. public void swap(int idx1, int idx2)
                        1. Swaps the two elements in the arraylist using the specified indices.
                    iii. public TBA get(int idx)
                        1. Returns the item at the specified index.

vi. Write a function to sort all **Pizzas** based on price using the QuickSort.
    1. protected void quickSortByPrice() {
    2. Notice you're sorting Objects here, so:
        a. Be sure to call "pizzaOne.compareTo(pizzaTwo)" in this section of your code, rather than using "<" or ">" like in the sorting of calories
vii. Build a function to display pizzas with the largest remaining areas first
    1. protected void quickSortBySize() {
    2. This sorts **Pizzas** based on the remaining area left, so be sure to scale the area returned by getArea() by the **Fraction** representing the remaining pie.
    3. Notice you're sorting Objects here, so:
        a. Be sure to call "pizzaOne.compareToBySize(pizzaTwo)" in this section of your code, rather than using "<" or ">" like in the sorting of calories
viii. Build a function that searches over pizzas using their calorie count.
    1. protected int binarySearchByCalories(int targetCal) {
        a. Be sure to call quickSortByCalories() first so the data is sorted!
ix. Build a function that removes all pizzas that don't have the same day as it is currently
    1. protected void removeDayOldPizzas();
    2. That is, you are removing all day old (or older) pizzas.
x. Build a function that reverses the list of pizzas you have using a stack
xi. Build a function that linearly searches over the pizzas using the day the pizza was made as the search target.
    1. protected int linearSearchByDay()

The PizzaComparable Interface (Provided)

This interface implements more than the Comparable Interface, and is provided for you. No additions or changes need to be made to this class.

The PizzaException Class

This is similar to any Exception class we've ever built; this should avoid Java's Catch-or-Declare.

The Pizza Class Implements PizzaComparable

Much of the work for managing Pizzas goes into this class. A Pizza has a list of ingredients, a shape, a cost, a calorie total, and a few other details. Pizzas can be eaten, and once a pizza is completely gone (i.e., the **Fraction** is 0) they should be removed from our PizzaManager's list by way of a thrown exception. Let's declare an empty class called Pizza that implements comparable; consider using Stepwise Refinement here, on any of the following steps, to flesh out more detail and make implementation more straightforward.

The Pizza Class Methods – A Stepwise Approach

1. Build the Pizza class hierarchy, starting with the superclass Pizza.
    a. Add "implements PizzaComparable" to your class definition.
        i. The compareTo function should compare Pizzas based on price
            1. Thus, the Money class must also implement Comparable
    b. Add data items to your class to track the following
        i. A list of ingredients
        ii. An integer calorie count
            1. This is the sum of all ingredients' calories
                a. Adding an ingredient changes this
        iii. A Money object to track cost of ingredients
            1. This is the sum of all ingredients' costs
                a. Adding an ingredient changes this
        iv. A Shape object to manage the shape of this pie
            1. Either Square or Circle in this simulation
        v. A Fraction object to manage the remaining pizza (and associated cost)
    c. Add getters and setters as follows:
        i. public Fraction getRemainingFraction()
            1. This gets the amount of pizza left using a Fraction
        ii. public void setRemainingFraction(Fraction f)
            1. This sets the amount of pizza left using a Fraction
        iii. public double getRemainingArea();
            1. This returns the area of the Shape scaled by the fraction of remaining pizza
        iv. public int getCalories(); //no setCalories, since this is defined by addIngredient()
        v. public int getCost(); //no setCost for the same reason
        vi. Add the following functions to get and set Shapes in your **Pizza** class. Notice the new syntax, ***which you should not change***.
            1. public void setShape(Shape s) { myShape = (Shape)s.clone();}
            2. public Shape getShape() { return (Shape) myShape.clone();}
    d. Build a method called "void addIngredient(Ingredient a)"

        i. This will add the ingredient to our list of ingredients
            1. And adjust the calories and cost accordingly
    e. Build a method called “void eatSomePizza(Fraction amt)”
        i. This will subtract amt from our Fraction of remaining pizza
            1. And throw an exception at 0 to be caught and handled.
            2. And throw a *different* type of exception to indicate an error if the remaining amount minus the provided fraction is negative, as this is an error case.
    f. Add a toString() and a compareTo()
    g. Add a default, no-arg constructor that generates a Pizza with a random shape (Square or Circle) and a random set of ingredients.
        i. This will be useful for testing your software quickly

The Ingredient Hierarchy

This set of classes will be used to decorate **Pizza** Objects. They’ll manage their cost and calorie count per serving, as well as define some custom characteristics (read member variables) like **Colors** for the **Vegetable** subclass. Before starting on the code, be sure to check out the inheritance hierarchy at the beginning of this assignment

1. Create the **Ingredient** class hierarchy from the **Top Down** (also see **Stepwise Refinement**)
    a. Build the **Ingredient** superclass which “implements Comparable” based on cost
        i. Note that this method can simply redirect to the Money’s compareTo function, and so is a façade or adapter.
    b. Add data items to track the Ingredient’s String description, cost, and calorie count.
        i. You can make these public and final, or private with getters/setters
    c. Add a constructor to **Ingredient** that requires a String description, a Money object, and a calorie count.
        i. This will, in turn, require all subclasses provide such a constructor, too, which calls the superclass constructor.
    d. Build three interior subclasses for **Base** (**Marinara** and **Olive** oil are bases), **Cheese**, **Vegetable**, and **Meat**.
        i. For the **Vegetable** Class, add the following data items:
            1. A variable to manage the **Vegetable’s Color**
        ii. Getters and setters for the Color
        iii. Two constructors:
            1. One that takes a String description, a Money object, and a calorie count
            2. The other that takes those and a default, starting color.

iv. A toString() that also prints the color in addition to the output for the superclass toString().
v. An equals() that compares colors and invokes the superclass equals() function.

e. Build the final level of leaf subclasses that inherit from **Base**, **Cheese**, **Vegetable**, or **Meat**.
   i. Provide at least the two subclasses per interior class, as specified in the diagram
      1. Use their names exactly as indicated for the class name.
      2. Define a String description and a calorie count for each of these subclasses.

Notes

- Don't wait until the last minute to get help from your instructor, classmates, and tutor – this project is the longest we'll see in 162.
- Build your software in pieces
  - First, build the PizzaException.
  - Next, build the slightly larger Ingredient superclass
    - Then the vegetable subclass
      - And the Olive and Pepper (sub)subclasses
  - Test your vegetables before proceeding
  - Build the other Ingredients (short classes)
  - Then build the Pizza or the PizzaManager Class.
  - Test each piece as you progress, rather than "waiting until the end"
    - i.e., put a main in each class and test them

Sample Output

```
----------------------
Welcome to PizzaManager
----------------------
(A)dd a random pizza
Add a (H)undred random pizzas
(E)at a fraction of a pizza
QuickSort pizzas by (P)rice
QuickSort pizzas by (S)ize
QuickSort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
A
Adding a random pizza to the ArrayList<Pizza>.
Pizza has a price:$10.22 and calories:315, Fraction remaining:1 and area left:804.25 and shape:Circular
----------------------
```

Welcome to PizzaManager

(A)dd a random pizza
Add a (H)undred random pizzas
(E)at a fraction of a pizza
QuickSort pizzas by (P)rice
QuickSort pizzas by (S)ize
QuickSort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
E
Eating a fraction of a pizza. Which index & how much?**(index numerator/denominator)**
0 1/5
Pizza has a price:$8.17 and calories:315, Fraction remaining:4/5 and area left:643.40 and shape:Circular
---
Welcome to PizzaManager

(A)dd a random pizza
Add a (H)undred random pizzas
(E)at a fraction of a pizza
QuickSort pizzas by (P)rice
QuickSort pizzas by (S)ize
QuickSort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
h
Adding one hundred random pizzas to the ArrayList<Pizza>.
Pizza has a price:$8.17 and calories:315, Fraction remaining:4/5 and area left:643.40 and shape:Circular
Pizza has a price:$10.22 and calories:315, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$11.47 and calories:390, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$2.99 and calories:80, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.70 and calories:625, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$10.98 and calories:460, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.70 and calories:625, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.21 and calories:695, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$8.48 and calories:310, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$5.49 and calories:230, Fraction remaining:1 and area left:804.25 and shape:Circular
---
Welcome to PizzaManager

(A)dd a random pizza
Add a (H)undred random pizzas

(E)at a fraction of a pizza
QuickSort pizzas by (P)rice
QuickSort pizzas by (S)ize
QuickSort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
p
QuickSorting pizzas by (P)rice
Pizza has a price:$18.70 and calories:625, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.70 and calories:625, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.21 and calories:695, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$11.47 and calories:390, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$10.98 and calories:460, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$10.22 and calories:315, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$8.48 and calories:310, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$8.17 and calories:315, Fraction remaining:4/5 and area left:643.40 and shape:Circular
Pizza has a price:$5.49 and calories:230, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$2.99 and calories:80, Fraction remaining:1 and area left:804.25 and shape:Circular
---
Welcome to PizzaManager

(A)dd a random pizza
Add a (H)undred random pizzas
(E)at a fraction of a pizza
QuickSort pizzas by (P)rice
QuickSort pizzas by (S)ize
QuickSort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
c
QuickSorting pizzas by (C)alories
Pizza has a price:$2.99 and calories:80, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$5.49 and calories:230, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$8.48 and calories:310, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$10.22 and calories:315, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$8.17 and calories:315, Fraction remaining:4/5 and area left:643.40 and shape:Circular
Pizza has a price:$11.47 and calories:390, Fraction remaining:1 and area left:804.25 and shape:Circular

```
Pizza has a price:$10.98 and calories:460, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.70 and calories:625, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.70 and calories:625, Fraction remaining:1 and area left:804.25 and shape:Circular
Pizza has a price:$18.21 and calories:695, Fraction remaining:1 and area left:804.25 and shape:Circular
----------------------
Welcome to PizzaManager
----------------------
(A)dd a random pizza
Add a (H)undred random pizzas
(E)at a fraction of a pizza
QuickSort pizzas by (P)rice
QuickSort pizzas by (S)ize
QuickSort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
```

Notes & Hints

- Name classes ***exactly*** as they appear above.
    - Don't pluralize "Olives", for example.
    - Don't change the name of a method from "eatSomePizza()" to "eatPizza"
    - Don't change the input of a method. So, eatSomePizza(Fraction amt) cannot instead be eatSomePizza(int numSlices).