

Description

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 later on in this document.

This is a big project: Do not wait till the last minute to start it!:



- Level of Challenge: Hard.
- Number of classes/files to submit: 10-20
- Number of inheritance Hierarchies: 3 complete hierarchies involving Shapes, Ingredients, & Pizzas.

What to Submit

- The Pizza Class
- The PizzaManager Class (driver & test code)
- Ingredient Class Hierarchy (13 classes)
- The PizzaException Class
- Any other dependencies needed for the program to work

Please submit a zip archive with all your files inside called **PizzaSimulator.zip**.

Diagram and Code Files

- Ingredient Hierarchy diagram: [IngredientHierarchy.pdf](#)
- Driver file: [PizzaManager.java](#) 
- PizzaComparable file: [PizzaComparable.java](#) 

Top-Down Design & Stepwise Refinement

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 hierarchy 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**.

In the sections below, we start by describing the components of the project that utilize classes we've built/used before, revised and reused. Next, we describe those components that are new classes. Finally, we outline the class invariants for the project and finish with notes to be aware of. As you read the assignment description, you'll want to also have the **PizzaManager** class and **PizzaComparable** interface code and comments alongside.

Classes to Reuse

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

- Find or create a **Fraction** class that manages a numerator and denominator with reduction.
 - Make your numerator and denominator final and set them in the constructor.
 - See the previous assignments for the Fraction's interface (i.e., 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 **Color** class (called java.awt.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.

New Class: PizzaComparable

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

New Class: PizzaException

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

New Classes: 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 diagram found in IngredientHierarchy.pdf.

1. Create the **Ingredient** class hierarchy from the **Top Down** (also see **Stepwise Refinement**):
 - Build the **Ingredient** superclass which "implements Comparable" based on cost. Note that this method can simply redirect to the Money's compareTo function, and so is a façade or adapter.
 - Add data items to track the Ingredient's String description, cost, and calorie count. You can make these immutable by making them public and final or private with getters/setters.

- Add a constructor to **Ingredient** that requires a String description, a Money object, and a calorie count. This will, in turn, require all subclasses provide such a constructor, too, which calls the superclass constructor.
 - Don't forget to define toString() and equals() for this class.
- 2. Build four interior subclasses: **Base** (**Marinara** and **Alfredo** are bases), **Cheese**, **Vegetable**, and **Meat**.
- 3. For the **Vegetable** Class, add the following data items:
 - A variable to manage the **Vegetable's Color** (this is not a String; use the Color class)
 - Getters and setters for the Color
 - Two constructors: One that takes a String description, a Money object, and a calorie count. The other that takes those and a starting color.
 - A toString() that also prints the color in addition to the output for the superclass toString().
 - An equals() that compares colors and invokes the superclass equals() function.
- 4. Build the final level of leaf subclasses that inherit from **Base**, **Cheese**, **Vegetable**, or **Meat**. Provide at least the two subclasses per interior class, as specified in the diagram (use their names exactly as indicated for the class name).

New Class: Pizza

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. (Note that to obtain an area, you'll have to set a radius/lengths for your pizza shape. You can choose how you'd like to do that as long as you adequately document your choice.) 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. As we write Pizza, start with an empty class definition and then use Stepwise Refinement to add the detail:

1. Pizza will implement PizzaComparable, so add "implements PizzaComparable" to your class definition:
 - The compareTo function should compare Pizzas based on price
 - Thus, the Money class must also implement Comparable
2. Add data items to your class to track the following:
 - A list of ingredients
 - An integer calorie count. This is the sum of all ingredients' calories. Adding an ingredient changes this
 - A Money object to track cost of ingredients. This is the sum of all ingredients' costs. Adding an ingredient changes this
 - A Shape object to manage the shape of this pie. Either Square or Circle in this simulation.
 - A Fraction object to manage the remaining pizza (and associated cost)
3. Add getters and setters as follows:
 - get/setRemaining(Fraction f) //gets and sets the amount of pizza left
 - getCalories(); //no setCalories, since this is defined by addIngredient()
 - getCost(); //no setCost for the same reason

- `double getRemainingArea();` //returns the area of the Shape scaled by the fraction of remaining pizza.
- Add the following functions to get and set Shapes in your **Pizza** class. Notice the new syntax, *which you should not change*.
 - `public void setShape(Shape s) { myShape = (Shape)s.clone();}`
 - `public Shape getShape() { return (Shape) myShape.clone();}`
- 4. Build a method called “void addIngredient(Ingredient a)”. This will add the ingredient to our list of ingredients and adjust the calories and cost accordingly.
- 5. Build a method called “void eatSomePizza(Fraction amt)”
 - This will subtract amt from our Fraction of remaining pizza
 - And throw an exception at 0 to be caught and handled.
 - And throw a different exception to indicate an error if the remaining amount minus the provided fraction is negative, as this is an error case that is handled differently than the case when all the pizza is eaten.
- 6. Add a `toString()` and a `compareTo()`
- 7. Add a default, no-arg constructor that generates a Pizza with a random shape (Square or Circle) and a random set of ingredients. This will be useful for testing your software quickly.

New Class: Pizza Manager

A **PizzaManager** keeps track of pizzas over the course of its existence. When a Pizza is first made, it has 100% (1/1) of its 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. Much of the **PizzaManager** interface is defined. In other classes, you’ll have to define your own methods accordingly.

The `PizzaManager.java` sample driver shows you the structure of the interface and gives an example of how such an interface can be created. You can add additional helper methods but should not remove or rename any of the functions provided for you in this class. (Though, if there are any conflicts between that code and this document, implement this document.)

Below are details regarding **PizzaManager**. Again, use a stepwise approach in creating the class.

Data Members

In the data section, define an `ArrayList<Pizza>` to hold your pizzas.

Methods

In the methods section, define and implement the following methods. Note that where you are asked to write a sort method, **you may not use a pre-written method such as `Array.sort()`**. You have to implement the sort from scratch:

1. Write a function to list all **Pizzas** in their current order in your **ArrayList**: public void displayAllPizzas()
2. Write a function to subtract a fractional amount from a Pizza: public void eatSomePizza(Scanner keys). This function subtracts the amount from the remaining pizza:
 - To accomplish this, ask the user for the fractional amount to eat and the index of the pizza to eat
 - If the ratio reaches zero, throw a PizzaException and catch it in the **PizzaManager** class so this pizza is removed
 - If the ratio is negative, throw a different exception from PizzaException as this is an error case that will be handled differently in the **PizzaManager**.
3. Build a function that sorts **Pizzas** in order with greatest calories first: public void sortByCalories().
 - Use selection or insertion sort.
 - Notice you're sorting primitives here.
 - To finish this method, you'll need to add the following methods to **ArrayList**:
 - public int size()
 - public void swap(int idx1, int idx2): Swaps the two elements in the arraylist using the specified indices.
 - public TBA get(int idx): Returns the item at the specified index.
4. Write a function to sort all **Pizzas** based on price, with greatest price first: public void sortByPrice():
 - Use selection or insertion sort, whichever you did *not* use in #3 above.
 - Notice you're sorting Objects here, so be sure to call "pizzaOne.compareTo(pizzaTwo)" in this section of your code, rather than using "<" or ">" like in the sorting of calories
5. Build a function to display pizzas with the largest remaining areas first: public void sortBySize():
 - Use selection or insertion sort.
 - 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.
 - Notice you're sorting Objects here, so: Be sure to call "pizzaOne.compareToBySize(pizzaTwo)" in this section of your code, rather than using "<" or ">" like in the sorting of calories
6. Build a function that searches over pizzas using their calorie count: public int binarySearchByCalories(int targetCal). Be sure to call sortByCalories() first so the data is sorted!

Invariants

1. When a Pizza is made, it starts with 100% of its 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:
 - If a Pizza's remaining fraction is less than 0, throw an exception as this is an error case.

- 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. 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.
 - So, calling addIngredient() on a Pizza changes the calorie total for the pizza.
 - But, since each calorie count is supposedly a per-serving amount, this value should **not** be scaled by area left but rather represents the caloric consumption per serving and remains static.
- 6. Ingredient is immutable (see the description of the Ingredient hierarchy for details as to what this means).
- 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. This will mean that clone() for the children of Shape will need to be custom implementations for that shape rather than being reliant on Object's clone method as per Savitch.
- 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. Square and Circle will implement the clone methods to override the abstract superclass clone.

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 143.
- Build your software in pieces:
 - First, build the PizzaException.
 - Next, build the slightly larger Ingredient superclass.
 - Then the Vegetable subclass.
 - And then the subclasses in the Vegetable hierarchy.
 - Test your Vegetables before proceeding.
 - Build the other Ingredients (short classes).
 - Then build the Pizza and 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.
- You may be confused that this assignment description talks about public methods while the skeleton driver code talks about the “same” methods as being private. The reason this is the case is that in the skeleton, the user-interface is controlled via the start() method while in this assignment description, no specification is made as to how we'll control the user-interface. If we were to make use of instances of **PizzaManager** in another class, most of the private methods in the skeleton would have to become public, as this assignment description specifies. As far as you're concerned, whether the methods are public or private depends on the user-interface you implement. Finally, also note that we've seen cases where private methods are called by public versions (e.g., in recursive routines), and so you can have both public and private methods intermixed to accomplish a task.

- The skeleton driver code is meant only as a prompt. If anything in this document conflicts with that code, implement this document. The sample output below is also just a prompt. If anything in the specifications above conflicts with the sample output, implement the specifications above.
- Remember to follow the Coding Style Guidelines, handle errors using exceptions, write tests the proper way, etc.
- The term "interior" class is used in the assignment description. It does not mean "inner" class; you do not need to create any inner classes in this assignment. Just think of "interior" as meaning an intermediate class, somewhere between leaf classes and ancestor classes.
- Any classes whose names are specified in the assignment description should be given that name.
- **Make sure you comment all methods and the class with Javadoc comments.** This includes constructors, getters, setters, etc. If you have files based on code someone else has written (e.g., is based on a skeleton), you have to Javadoc comment the methods someone else wrote also. You cannot receive full-credit for the rubric's commenting criterion if you do not have a **javadoc comment for every method**, no matter how small.

Sample Output

Sample Output (which may differ slightly from the provided skeleton)

```

-----
Welcome to PizzaManager
-----
(A)dd a random pizza
Add a (H)undred random pizzas
(E)at a fraction of a pizza
Sort pizzas by (P)rice
Sort pizzas by (S)ize
Sort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
A
Adding a random pizza to the ArrayList.
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
Sort pizzas by (P)rice
Sort pizzas by (S)ize
Sort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit
E
Eating a fraction of a pizza. How much? (a/b)
1/5
At which index?

```


0

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
Sort pizzas by (P)rice
Sort pizzas by (S)ize
Sort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit

h

Adding one hundred random pizzas to the ArrayList.

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
Sort pizzas by (P)rice
Sort pizzas by (S)ize
Sort pizzas by (C)alories
(B)inary Search pizzas by calories
(Q)uit

p

Sorting 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

About This Document

Original assignment by Rob Nash, Autumn 2014. Minor edits and additions here and on the assignment description by Johnny Lin, Eddrick Liu, et al., Spring 2017.

Rubric

Pizza Simulator Rubric (1)

Pizza Simulator Rubric (1)		
Criteria	Ratings	Pts
This criterion is linked to a Learning Outcome Clear, Well-Written, and Complete Comments in Code		15.0 pts
This criterion is linked to a Learning Outcome All Required Classes Present and Working		25.0 pts
This criterion is linked to a Learning Outcome Program Compiles and Properly Executes		10.0 pts
This criterion is linked to a Learning Outcome Driver and/or Class main Methods Comprehensively and Adequately Tests Program		15.0 pts
Total Points: 65.0		