Source code and output:

Source code:

Main class:

package pizzamanager;

import java.util.Scanner;

/\*\*

\*

\* @author JMC

\*/

public class PizzaManager {

ArrayList<Pizza> pizzas = new ArrayList<Pizza>();

/\*

\* The console interface is defined in the start method

\* You can exit or extend the code below to accomplish all of

\* the outcomes defined in the homework document

\*/

public void start() {

char selection='q';

Scanner foo = new Scanner(System.in);

//Beginning of while loop of main program

while(true) {

displayAllPizzas();

displayInstructions();

//Obtain the char of the first typed character input

selection = foo.next().charAt(0);

switch(selection) {

case 'A':

case 'a': System.out.println("Adding a random pizza to the ArrayList<Pizza>.");

//Call addRandomPizza to add a new Pizza to the Pizza Array List

addRandomPizza();

break;

case 'H':

case 'h': System.out.println("Adding 5 random pizzas to the ArrayList<Pizza>.");

//To add 100 random Pizzas to the Array List, I run a loop for 100 times and call

//the same addRandomPizza method

for(int i=0;i<5;i++){

addRandomPizza();

}

break;

case 'E':

case 'e': System.out.println("Eating a fraction of a pizza. How much?");

//Created a scanner to take the fraction input

Scanner fracInput = new Scanner(System.in);

//Grab the next input as a String to help with parsing the fraction input

String frac = fracInput.next();

//Split user string input by the / sign

//Then obtain the Integer value of each value

//for the numerator and denominator

//Using regular expression for the splitting

String[] fraction = String.valueOf(frac).split("\\/");

int num = Integer.valueOf(fraction[0]);

int denom = Integer.valueOf(fraction[1]);

//System.out.println("Denom: " + denom);

//System.out.println("Num: " + num);

//Prompt user for additional input for the pizza index to eat

System.out.println("At which pizza index?");

//Take in input as an int

int fractionIndex = fracInput.nextInt();

//Call method eatSomePizza and pass in the user generated fraction and index

eatSomePizza(new Fraction(num,denom),fractionIndex);

break;

case 'F':

case 'f': System.out.println("QuickSorting pizzas by (F)raction");

//WOW, SO MUCH BONUS

//Sorting Pizzas by Fractions because it's cool

//I added in another PizzaComparable method for this to work

quickSortByFraction();

break;

case 'P':

case 'p': System.out.println("QuickSorting pizzas by (P)rice");

//Sorting Pizzas by price using the quicksort algorithm, calls quickSortByPrice

quickSortByPrice();

break;

case 'S':

case 's': System.out.println("QuickSorting pizzas by (S)ize");

//Sorting Pizzas by Size using the quicksort algorithm, calls quickSorBySize

quickSortBySize();

break;

case 'C':

case 'c': System.out.println("QuickSorting pizzas by (C)alories");

//Sorting Pizzas by Calories using the quicksort algorithm, calls quickSorByCalories

quickSortByCalories();

break;

case 'B':

case 'b': System.out.println("(B)inary search over pizzas by calories(int). \nQuickSorting first. \nWhat calorie count are you looking for?");

//Method for finding a Pizza by the number of Calories

//Uses the binary search algorithm and first sorts by Calories

Scanner calInput = new Scanner(System.in);

//Asks user for Calories to search and find

int cals = calInput.nextInt();

int index = binarySearchByCalories(cals);

//If the index returned by the search function is -1, then a pizza with the specified calories wasn't found

if(index == -1){

System.out.println("Pizza with " + cals + " calories not found.");

}

else {

//Prints that a pizza with the specified calories was found and lists the index location

System.out.println("Pizza with " + cals + " calories found:");

System.out.println(pizzas.get(index) + "\n\n");

}

break;

case 'R':

case 'r': System.out.println("(R)everse order of Pizzas with a Stack" );

//This reverses the current order of Pizzas in the ArrayList

//It doesn't sort first

//Creating a stack for reversing the order

PizzaStack stacky = new PizzaStack();

int numElements = pizzas.size();

//Loops through pizza Array List and pushes the results onto the stack

for(int i=0; i != pizzas.size();){

Pizza temp = pizzas.remove(i);

//System.out.println(temp);

stacky.push(temp);

}

//While the stack isn't empty, the stack

//pushes the results back into thte pizza Array List

//This results in reversing the order of the list

while(!stacky.isEmpty()){

Pizza temp2 = (Pizza) stacky.pop();

//System.out.println(temp2);

pizzas.add(temp2);

}

System.out.println(pizzas.size());

break;

case 'Q':

case 'q': System.out.println("(Q)uitting!" );

//Exit program

System.exit(0);

break;

default: System.out.println("Unrecognized input - try again");

}

}

}

/\*

a.This function subtracts the amount from the remaining pizza.

i. To accomplish this, add a subtract function to the Fraction class

ii. If the ratio reaches zero, throw a PizzaException and catch it in the PizzaManager class so this pizza is removed

iii. If the ratio is negative, throw a PizzaException as this is an error case.

\*/

public void eatSomePizza(Fraction amount, int index){

if(amount == null || !(amount instanceof Fraction)){

throw new PizzaException("Invalid fraction");

}

//System.out.println(amount.toString() + index);

//Try catch to eat an amount from a pizza

//The amount is a fraction of the whole pizza and not a fraction of the available pizza

try{

pizzas.get(index).eatSomePizza(amount);

}

//Catching the Pizza Exception that will be thrown when the pizza is entirely eaten

catch(PizzaException p){

//Print exception message

System.out.println(p.toString());

//Remove pizza from pizza list as there isn't anymore of it available

pizzas.remove(index);

}

}

//The addRandomPizza method calls the add function on the pizza array list

//and adds a pizza. a pizza created with no args will be randomly generated

private void addRandomPizza() {

pizzas.add(new Pizza());

}

//Write a function to list all Pizzas in their current order in your ArrayList

//This displays all Pizzas in the console

private void displayAllPizzas() {

//Loops over pizza array list size and calls get on pizzas to return the current pizza

for(int i=0; i<pizzas.size(); i++){

System.out.println(pizzas.get(i));

}

}

/\*

\* 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

\*/

//Sorting by price method using the quick sort algorithm

//Facade function

private void quickSortByPrice() {

int first = 0;

int last = pizzas.size()-1;

sortPrice(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortPrice(ArrayList<Pizza> pizs, int first, int last) {

//Check if passed pizza array list is actually a pizza array list

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

System.out.println("Passed first must be >= 0 and last values must be >= -1");

System.exit(-1);

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionPrice(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortPrice(pizs, first, pivotIndex - 1);

sortPrice(pizs, pivotIndex + 1, last);

}

}

private int partitionPrice(ArrayList<Pizza> pizs, int left, int right){

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareTo(pivotValue));

if(pizs.get(i).compareTo(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

/\*

\* Build a function to display pizzas with the largest remaining areas first

\* 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

\*/

//Sorting by size method using the quick sort algorithm

//Facade function

private void quickSortBySize() {

int first = 0;

int last = pizzas.size()-1;

sortSize(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortSize(ArrayList<Pizza> pizs, int first, int last) {

//Validate pizza array list input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

throw new PizzaException("Invalid index");

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionSize(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortSize(pizs, first, pivotIndex - 1);

sortSize(pizs, pivotIndex + 1, last);

}

}

private int partitionSize(ArrayList<Pizza> pizs, int left, int right){

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareTo(pivotValue));

if(pizs.get(i).compareToBySize(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

//Sorting by fraction method using the quick sort algorithm

//Facade function

private void quickSortByFraction() {

int first = 0;

int last = pizzas.size()-1;

sortFraction(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortFraction(ArrayList<Pizza> pizs, int first, int last) {

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

throw new PizzaException("Invalid index");

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionFraction(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortFraction(pizs, first, pivotIndex - 1);

sortFraction(pizs, pivotIndex + 1, last);

}

}

private int partitionFraction(ArrayList<Pizza> pizs, int left, int right){

//Check for valid pizza array list input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareTo(pivotValue));

if(pizs.get(i).compareToByFraction(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

/\*

\* function that sorts Pizzas in order of greatest calories first

\* Notice you’re sorting primitives here

\* a. To make your life easier, 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.

\*/

//Sorting by size method using the quick sort algorithm

//Facade function

public void quickSortByCalories() {

int first = 0;

int last = pizzas.size()-1;

sortCalories(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortCalories(ArrayList<Pizza> pizs, int first, int last) {

//Validate passed Pizzas array list input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

System.out.println("Passed first must be >= 0 and last values must be >= -1");

System.exit(-1);

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionCalorie(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortCalories(pizs, first, pivotIndex - 1);

sortCalories(pizs, pivotIndex + 1, last);

}

}

private int partitionCalorie(ArrayList<Pizza> pizs, int left, int right){

//Validate pizza arraylist input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareToByCalories(pivotValue));

if(pizs.get(i).compareToByCalories(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

/\*

\* Build a function that searches over pizzas using their calorie count.

\* Be sure to call quickSortByCalories() first so the data is sorted!

\*/

//Binary search by calories takes in calorie input as target

//utilizes binary search algorithm

//Sorts by calories first to ensure algorithm will work

private int binarySearchByCalories(int cals) {

quickSortByCalories();

int low = 0;

int high = pizzas.size()-1;

int mid = 0;

//While the current low boundary is lower than the current high boundary

while(low <= high){

//Obtain middle result by adding and dividing low and high boundaries

mid = (low + high)/2;

//See if the middle result matches the target and returns middle index if does

if(pizzas.get(mid).getCalories() == cals){

return mid;

}

//If the

else if(pizzas.get(mid).getCalories() < cals){

//If the guessed middle value is lower than the target

//Then low is assigned the middle index + 1

low = mid+1;

}

else {

//If the guessed middle value is greater than the target

//Then high is assigned the middle index -1

high = mid-1;

}

}

//Returning -1 when no matches are found

return -1;

}

/\*

\* No need to edit functions below this line, unless extending the menu or

\* changing the instructions

\*/

private static final String instructions = "-----------------------\nWelcome to PizzaManager\n-----------------------\n(A)dd a random pizza\nAdd a (H)undred random pizzas\n(E)at a fraction of a pizza\nBONUS ROUND - QuickSorting pizzas by (F)raction\nQuickSort pizzas by (P)rice\nQuickSort pizzas by (S)ize\nQuickSort pizzas by (C)alories\n(B)inary Search pizzas by calories\n(R)everse order of pizzas with a stack\n(Q)uit\n";

private void displayInstructions() {

System.out.println(instructions);

}

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

new PizzaManager().start();

}

}

Sub classes:

ArrayList:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class ArrayList<TBA> {

private int capacity = 100;

private Object[] data = new Object[capacity];

private int numElements = 0;

//Get method that returns TBA object for an index

public TBA get(int index){

//Invariants for index

if(index > numElements || index < 0){

throw new PizzaException("Bad index");

}

//Cast return value as TBA object

return (TBA) data[index];

}

//Set method that will write or overwrite value at index

public void set(TBA o, int index){

//Check for bad index

if(index > numElements || index < 0){

throw new PizzaException("Bad index");

}

TBA that = (TBA) o;

//Set data to new TBA object

data[index] = that;

}

//Insert TBA object into data at index

public void insert(TBA o, int index){

//Invariants for object and index

if(o == null || index < 0 || index > numElements){

throw new PizzaException("Bad input");

}

//Resize data array if reach max size

if(numElements == capacity){

//Call resize method to increase size

resize();

}

TBA that = (TBA) o;

//Shift array to insert new TBA

shiftRight(index);

data[index] = that;

//Increase number of elements after inserting new TBA

numElements++;

}

//Add function that redirects to insert

public void add(TBA o){

insert(o, numElements);

}

//Private resize method that increases size of array

//Create new array that's 10x larger and copy in data from old array

private void resize(){

//Increasing by factor of 10

capacity = capacity \* 2;

Object tempData[] = new Object[capacity];

//Copy old array into new array

for(int i=0;i<numElements;i++){

tempData[i] = data[i];

}

data = tempData;

}

//Shift right method for inserting objects

private void shiftRight(int start){

for(int i=numElements-1; i >= start; i--){

data[i+1] = data[i];

}

}

//Remove method that delete object by shifting array left to overwrite it

public TBA remove(int index){

//Index invariants

if(index < 0 || index >= data.length){

throw new PizzaException("Bad Index");

}

//Shift array left to overwrite

Object retval = shiftLeft(index);

//Decrease number of elements

numElements--;

//Return TBA object

return (TBA) retval;

}

//Shift left method to overwrite array object for removing

private Object shiftLeft(int start){

Object retVal = data[start];

//Iterate over array for overwriting

for(int i = start; i < numElements-1; i++) {

data[i] = data[i+1];

}

//Return value that was overwritten

return retVal;

}

public ArrayList() {}

//Copy constructor for ArrayList

public ArrayList(ArrayList<TBA> other){

this.capacity = other.capacity;

this.numElements = other.numElements;

for(int i=0; i<other.numElements; i++){

this.data[i] = other.data[i];

}

}

//Return size of arraylist

public int size(){

return numElements;

}

//Swaps the two elements in the arraylist using the specified indices.

public void swap(int idx1, int idx2){

Object temp = data[idx1];

data[idx1] = data[idx2];

data[idx2] = temp;

}

//Overridden toString that shows TBAs in List

@Override

public String toString(){

String temp = "";

//Iterates over array and prints out each TBA

for(int i=0; i<numElements; i++){

temp += data[i].toString() + "\n";

}

return temp;

}

//Finds index of TBA

public int indexOf(TBA o){

//Invariant to see if passed object is null

if(o == null){

throw new PizzaException("Bad input");

}

//Iterate through array to find TBA

for(int i=0; i<numElements; i++){

if(data[i] == o) return i;

}

return -1;

}

//Main to test ArrayList

public static void main(String[] args){

ArrayList<String> a = new ArrayList<String>();

a.insert("Foo",0);

a.insert("b",1);

a.remove(1);

System.out.println(a.toString());

ArrayList<Pizza> pizzas = new ArrayList<Pizza>();

pizzas.add(new Pizza());

}

}

Alfredo:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Alfredo extends Base{

public static final int calories = 200;

public static final Money cost = new Money(0,50);

public static final String description = "Alredo from can";

//Constructor calls Base constructor with super

public Alfredo() {

super(description, cost, calories);

}

}

Base:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Base extends Ingrediants{

public Base(String description, Money cost, int calories) {

super(description, cost, calories);

}

}

Cheese:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Cheese extends Ingrediants{

public Cheese(String description, Money cost, int calories) {

super(description, cost, calories);

}

}

Circle:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Circle extends Shape{

public static final int radius = 16;

public double area;

public final static String shapeType = "Circle";

//Constructor that calls Shape constructor and sets the area

public Circle() {

super(radius, shapeType);

setArea();

}

//Copy Constructor that accepts Circle object

public Circle(Circle newCopy){

super(Circle.radius, newCopy.shapeType);

setArea();

}

//Set area method that calculates and sets area

public void setArea(){

this.area = Math.PI \* this.getX() \* this.getX();

}

//Set area method that takes a double as input

public void setArea(double a){

area = a;

}

//Get area method that returns area

public double getArea(){

return area;

}

//Clone method for Circle that implements the Clone method from Shape

@Override

public Circle clone() {

return new Circle(this);

}

}

Fraction:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Fraction implements Comparable{

private int numerator;

private int denominator;

//Default constructor sets fraction as 1/1

public Fraction() {

setNumerator(1);

setDenominator(1);

}

//Another constructor that takes numerator and denominator as arguments

public Fraction(int num, int denom){

reduce(num, denom);

}

//Copy constructor that takes fraction as argument

public Fraction(Fraction toCopy){

//Calls setters to set values

setNumerator(toCopy.getNumerator());

setDenominator(toCopy.getDenominator());

}

@Override

public Fraction clone(){

return new Fraction(this);

}

public void reduce(int num, int denom){

//Here we check for a denominator that's 0 and alert the user to the issue

if(denom == 0){

throw new PizzaException("Denom can't be 0");

}

int counter = 1;

int gcd = 1;

boolean positiveFraction;

if(num < 0 && denom < 0){

positiveFraction = true;

}

else if(num > 0 && denom > 0){

positiveFraction = true;

}

else {

positiveFraction = false;

}

int tempNum = Math.abs(num);

int tempDenom = Math.abs(denom);

if(tempNum < tempDenom){

counter = tempNum;

}

else {

counter = tempDenom;

}

//For loop to find the greatest common divisor for both the numerator and denominator

for(int i=1; i<=counter; i++){

if((tempNum % i == 0) && (tempDenom % i == 0)) gcd = i;

}

num = num / gcd;

denom = denom / gcd;

//Initialize numerator and denominator to positive

num = Math.abs(num);

denom = Math.abs(denom);

if(positiveFraction == false) {

num = num \* -1;

}

if(num == 0){

setNumerator(0);

setDenominator(0);

}

//else if(positiveFraction == false){

//throw new RuntimeException("Can't have negative fraction");

//}

else{

//Finally we have our final reduced fraction and use our setters to set the values.

setNumerator(num);

setDenominator(denom);

}

}

//This is the getter for the Numerator

public int getNumerator(){

return this.numerator;

}

//Setting Numerator for the fraction class.

public void setNumerator(int num){

this.numerator = num;

}

//This is the getter for the Denominator

public int getDenominator(){

return this.denominator;

}

//Setting Denominator for the fraction class. Including a validation that the denominator can't be 0.

public void setDenominator(int denominator){

if(denominator != 0){

this.denominator = denominator;

} else if(numerator == 0){

this.denominator = 0;

}

else{

throw new PizzaException("Can't have denominator of 0");

}

}

//This equals overrides the default equals as it checks based on the primitive values of the fraction

public boolean equals(Object o){

if(o == null || !(o instanceof Fraction)){

throw new PizzaException("Bad input");

}

Fraction that = (Fraction) o;

return this.numerator == that.numerator && this.denominator == that.denominator;

}

@Override

public int compareTo(Object o) {

if(o == null || !(o instanceof Fraction)){

throw new PizzaException("Bad input");

}

Fraction that = (Fraction) o;

if(this.equals(that)){

return 0;

}

Fraction temp = this.clone();

temp.subtract(that);

return temp.numerator;

}

//This toString overrides the default toString to be more readable

public String toString(){

return getNumerator() + "/" + getDenominator();

}

//Subtracts the Fraction f from this Fraction.

//Throws an exception if < 0, but not == 0

public void subtract(Fraction f){

//Validates for proper input

if(f == null || !(f instanceof Fraction)){

throw new PizzaException("Bad input");

}

Fraction that = (Fraction) f;

//Multiplies numerators and denominators to have same denominators

int newDenom = this.denominator \* that.denominator;

int num1 = this.numerator \* that.denominator;

int num2 = that.numerator \* this.denominator;

//Can now simply subtract

int finalNum = num1 - num2;

//Reducing in case reduction is needed

reduce(finalNum, newDenom);

}

//This is the main driver of the application

public static void main(String[] args){

Fraction f1 = new Fraction(1,1);

Fraction f2 = new Fraction(1,20);

f1.subtract(f2);

System.out.println(f1.toString());

}

}

Goat:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Goat extends Cheese{

public static final int calories = 120;

public static final Money cost = new Money(1,0);

public static final String description = "Goat cheese made from finest goats";

//Constructor calls Cheese constructor with super

public Goat() {

super(description, cost, calories);

}

}

Ingrediants:

package pizzamanager;

import java.awt.Color;

/\*\*

\*

\* @author JMC

\*/

public class Ingrediants implements Comparable {

private String description;

private Money cost;

private int calories;

private Color myColor;

private String colorName;

//Default constructor

public Ingrediants() {}

//Constructor that is passed description, cost and calories

public Ingrediants(String description, Money cost, int calories) {

setDescription(description);

setCost(cost);

setCalories(calories);

}

//Constructor that is passed description, cost, calories, and color

//Veggies have a color and use this constructor

public Ingrediants(String description, Money cost, int calories, Color color) {

setColor(color);

setDescription(description + " which are " + colorName);

setCost(cost);

setCalories(calories);

}

//Getter for returning color of ingredient

public Color getColor() {

return myColor;

}

//Setter for color

//This converts the passed Color object to a Color name string

public void setColor(Color that){

myColor = that;

//Converts RGB to color strings

if(myColor.getRed() == 255 && myColor.getGreen() == 0 && myColor.getBlue() == 0) {

colorName = "red";

}

else if(myColor.getRed() == 0 && myColor.getGreen() == 255 && myColor.getBlue() == 0) {

colorName = "green";

} else{

colorName = "black";

}

//System.out.println("R: " + myColor.getRed() + " G: " + myColor.getGreen() + " B " + myColor.getBlue());

}

//Setter for description

public void setDescription(String d){

description = d;

}

//Getter for cost Money object

public Money getCost() { //immutable

return cost;

}

//Getter for Description

public String getDescription(){

return description;

}

//Setter for cost

public void setCost(Object o){

//Validate Money input object

if(o == null || !(o instanceof Money)){

throw new PizzaException("Invalid Cost");

}

Money m = (Money) o;

cost = m;

}

//Getter for calories

public int getCalories() { //immutable

return calories;

}

//Setter for calories

public void setCalories(int c) {

if(c <= 0){

throw new PizzaException("Invalid calories");

}

else{

calories = c;

}

}

//Compare based on cost

//Note that this method can simply redirect to the Money’s compareTo function, and so is a façade or adapter.

@Override

public int compareTo(Object o) {

return cost.compareTo(o);

}

}

Marinara:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Marinara extends Base{

public static final int calories = 150;

public static final Money cost = new Money(0,25);

public static final String description = "Marinara from can";

//Constructor calls Base constructor with super

public Marinara() {

super(description, cost, calories);

}

}

Meat:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Meat extends Ingrediants{

public Meat(String description, Money cost, int calories) {

super(description, cost, calories);

}

}

Money:

package pizzamanager;

import java.io.Serializable;

/\*\*

\*

\* @author JMC

\*/

public class Money {

private int rupees = 0;

private int cents = 0;

//Setter for dollars with invariants

public void setRupees(int r){

//Don't allow negative dollars and exit the program if passed negative dollars.

if(r < 0) {

System.out.println("Rupees can't be negative. Please fix rupees");

System.exit(-1);

} else {

rupees = r;

}

}

//Setter for cents with invariants

public void setCents(int c){

//Can't have negative cents

if(c < 0) {

System.out.println("Cents can't be negative. Please fix cents");

System.exit(-1);

}

//If pass > 99 cents, adds to dollar amount.

else if(c > 99){

cents = c % 100;

rupees += (c - cents)/100;

} else {

cents = c;

}

}

//Getter for rupees

public int getRupees() {

return rupees;

}

//Getter for cents

public int getCents() {

return cents;

}

//Default constructor

public Money() {}

//Constructor that takes an int for rupees

public Money(int r) {

rupees = r;

}

//Copy constructor that takes a Money object

public Money(Money amount){

setRupees(amount.rupees);

setCents(amount.cents);

}

@Override

public Money clone(){

return new Money(this);

}

public Money(int r, int c){

setMoney(r, c);

}

//Setter for money that takes dollars and cents. Calls the dollar and cents setters.

public void setMoney(int r, int c){

setRupees(r);

setCents(c);

}

//Getter for the amount of dollars and cents

public String getAmount(){

return String.format("$%d.%d", rupees, cents);

}

//Setter for amount of dollars and cents

public void setAmount(Money amount){

setRupees(amount.rupees);

setCents(amount.cents);

}

//Overridden toString from Object class returning amount method.

@Override

public String toString(){

return getAmount();

}

//Overridden equals from Object class. Returns false if passed null.

@Override

public boolean equals(Object o){

if(o == null || ! (o instanceof Money)){

return false;

} else {

Money that = (Money) o;

return this.rupees == that.rupees &&

this.cents == that.cents;

}

}

//Driver for money to test

public static void main(String[] args){

Money a = new Money(0);

//money.setCents(1000);

//System.out.println(money.toString());

Money b = a.clone();

a.setCents(100);

System.out.println(a.toString());

//System.out.println(b.toString());

}

public int compareTo(Object o) {

if(o == null || !(o instanceof Money)){

throw new PizzaException("Passed Object isn't Money");

}

Money that = (Money) o;

String this\_money = this.getRupees() + "." + this.getCents();

String that\_money = that.getRupees() + "." + that.getCents();

Double retVal = Double.parseDouble(this\_money) - Double.parseDouble(that\_money);

if(retVal == 0){

return 0;

}

else if(retVal>0){

return 1;

}

else{

return -1;

}

}

}

Mozarella:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Mozarella extends Cheese{

public static final int calories = 300;

public static final Money cost = new Money(0,75);

public static final String description = "Mozzarella sticks sliced";

//Constructor calls Cheese constructor with super

public Mozarella() {

super(description, cost, calories);

}

}

Olive:

package pizzamanager;

import java.awt.Color;

/\*\*

\*

\* @author JMC

\*/

public class Olive extends Vegetable{

public static final int calories = 90;

public static final Money cost = new Money(0,90);

public static final String description = "Hand-picked olives from Italy";

//Constructor calls Cheese constructor with super

public Olive(Color color) {

super(description, cost, calories, color);

}

}

Pepper:

package pizzamanager;

import java.awt.Color;

/\*\*

\*

\* @author JMC

\*/

public class Pepper extends Vegetable{

public static final int calories = 75;

public static final Money cost = new Money(0,75);

public static final String description = "Organic, guilt free, and locally grown in my back yard peppers";

//Constructor calls Cheese constructor with super

public Pepper(Color color) {

super(description, cost, calories, color);

}

}

Pepporoni:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Pepporoni extends Meat{

public static final int calories = 350;

public static final Money cost = new Money(0,95);

public static final String description = "Fresh and spicy pepperoni";

//Constructor calls Meat constructor with super

public Pepporoni() {

super(description, cost, calories);

}

}

Pizza:

package pizzamanager;

import java.awt.Color;

/\*\*

\*

\* @author JMC

\*/

public final class Pizza implements PizzaComparable{

private Money price = new Money(0,0);

private Fraction size = new Fraction();

private Shape myShape;

private int calories = 0;

private ArrayList<Ingrediants> ingrediants = new ArrayList<Ingrediants>();

//A constructor that builds a Pizza with a random number of ingredients

public Pizza() {

//Making Pizza Shape randomly a circle or square

if(Math.random() > 0.5){

Circle c = new Circle();

myShape = c;

}

else{

Square s = new Square();

myShape = s;

}

//Making Random Base required

if(Math.random() > 0.5){

ingrediants.add(new Marinara());

}

else{

ingrediants.add(new Alfredo());

}

//Making Random Cheese required

if(Math.random() > 0.5){

ingrediants.add(new Goat());

}

else{

ingrediants.add(new Mozarella());

}

//Making Meat optional

//1/3 time Sausage, 1/3 time Pepperoni, 1/3 time no meat

double meatSeed = Math.random();

if(meatSeed < 0.33){

ingrediants.add(new Susage());

}

else if(meatSeed > 0.33 && meatSeed < 0.66){

ingrediants.add(new Pepporoni());

}

//Making Veggies optional

//1/3 time Pepper, 1/3 time Olive, 1/3 no veggies

double veggieSeed = Math.random();

//When selecting a Pepper, 1/2 time black and 1/2 time green

if(veggieSeed < 0.33){

if(Math.random() > 0.5){

ingrediants.add(new Pepper(Color.GREEN));

}

else{

ingrediants.add(new Pepper(Color.RED));

}

}

//When selecting an Olive, 1/2 time green, 1/2 red

else if(veggieSeed > 0.33 && veggieSeed < 0.66){

if(Math.random() > 0.5){

ingrediants.add(new Olive(Color.GREEN));

}

else{

ingrediants.add(new Olive(Color.BLACK));

}

}

//Updating Pizza Object variables calories and initial cost of ingredients

int currentCents = 0;

int currentRupees = 0;

int ingredientCents = 0;

int ingredientRupees = 0;

//Looping through each ingredients in the ingredient Array List

for(int i=0;i<ingrediants.size();i++){

calories += ingrediants.get(i).getCalories();

currentCents = price.getCents();

currentRupees = price.getRupees();

ingredientCents = ingrediants.get(i).getCost().getCents();

ingredientRupees = ingrediants.get(i).getCost().getRupees();

price.setRupees(currentRupees+ingredientRupees);

price.setCents(currentCents+ingredientCents);

}

}

//Overriding the base class toString

@Override

public String toString(){

String toppings = "";

//For loop so that the first ingredient in the ingredient arraylist doesn't have "and" in front of it

for(int i=0;i<ingrediants.size();i++){

if(i == 0){

toppings = ingrediants.get(i).getDescription();

}

else {

toppings = toppings + " and " + ingrediants.get(i).getDescription();

}

}

//Obtain dynamically calculated remaining value for Pizza by calling getRemainingArea

double area = getRemainingArea();

//Downsizing area so it's not as long when printing to console

String areaStr = String.valueOf(area);

if(areaStr.length()>7){

areaStr = areaStr.substring(0, 7);

}

else if(areaStr.length()<7){

int temp = 7 - areaStr.length();

for(int i=0;i<temp;i++){

areaStr = areaStr + "0";

}

}

return "\nShape: " + myShape.getShapeType() + " \nArea: " + areaStr + " \nCalories: " + calories + " \nPrice: " + getCost().toString() + " \nFraction Left: " + getFraction() + " \nToppings: " + toppings;

}

//Getters & setters for the Fractional amount of the pizza remaining

public Fraction getFraction(){

return size;

}

public void setFraction(int num, int denom){

size.setNumerator(num);

size.setDenominator(denom);

}

//Dynamically calculates Remaining Area

public double getRemainingArea(){

//Obtain the numerator and denominator of size fraction that contains fraction of pizza left

double numAvail = (double) size.getNumerator();

double denomAvail = (double) size.getDenominator();

//Divide numerator and denominator as doubles for accuracy

double amountAvail = numAvail/denomAvail;

return myShape.getArea() \* amountAvail;

}

public Money getCost(){

//Obtain the numerator and denominator of size fraction that contains fraction of pizza left

double numAvail = (double) size.getNumerator();

double denomAvail = (double) size.getDenominator();

//Divide numerator and denominator as doubles for accuracy

double amountAvail = numAvail/denomAvail;

//System.out.println("amountAvail: " + amountAvail);

//Obtain cost of whole pizza with current ingredients

double currentCost = Double.parseDouble(price.getRupees() + "." + price.getCents());

//Determine new cost by multiplying original cost and fraction remaining

double newCost = currentCost \* amountAvail;

String[] newAmount = String.valueOf(newCost).split("\\.");

//Dollars is the first item in array

String newDollars = newAmount[0];

String newCents = newAmount[1] + "0000";

newCents = newCents.substring(0, 2);

//Return new Money object that holds the current cost of a pizza

return new Money(Integer.parseInt(newDollars), Integer.parseInt(newCents));

}

public void eatSomePizza(Object o){

//Check for valid Fraction passed

if(o == null || !(o instanceof Fraction)){

throw new PizzaException("Invalid Fraction passed for eating some pizza");

}

Fraction amount = (Fraction) o;

//Subtract passed Fraction

size.subtract(amount);

if(size.getNumerator() < 0){

throw new PizzaException("I wish you could, but unfortunately you can't eat more than the available pizza");

}

else if(size.getNumerator() == 0){

price = new Money(0,0);

myShape.setArea(0);

throw new PizzaException("Pizza has been eaten and will be removed from the Pizza list!");

}

}

//Set Shape method that calls clone to avoid privacy leaks

public void setShape(Shape s) {

myShape = (Shape) s.clone();

}

//Get shape method that calls clone to avoid privacy leaks

public Shape getShape() {

return (Shape) myShape.clone();

}

//Getter for Calories

public int getCalories(){

return calories;

}

@Override

public int compareTo(Object o) {

//Check for valid input

if(o == null || !(o instanceof Pizza)){

throw new PizzaException("Invalid money");

}

Pizza that = (Pizza) o;

return this.getCost().compareTo(that.getCost());

}

@Override

public int compareToBySize(Object o) {

//Check for valid input

if(o == null || !(o instanceof Pizza)){

throw new PizzaException("Invalid size");

}

Pizza that = (Pizza) o;

//return this.size.compareTo(that.size);

return (int) (this.getRemainingArea() - that.getRemainingArea());

}

@Override

public int compareToByFraction(Object o) {

//Check for valid input

if(o == null || !(o instanceof Pizza)){

throw new PizzaException("Invalid size");

}

Pizza that = (Pizza) o;

return this.size.compareTo(that.size);

}

@Override

public int compareToByCalories(Object o) {

//Check for valid input

if(o == null || !(o instanceof Pizza)){

throw new PizzaException("Invalid size");

}

Pizza that = (Pizza) o;

return this.calories - that.calories;

}

//Main used for testing Pizzas

public static void main(String[] args){

Pizza p = new Pizza();

}

}

PizzaComparable:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public interface PizzaComparable extends Comparable{

@Override

public int compareTo(Object o); //a.k.a compareToByPrice

//non-overrides

public int compareToBySize(Object o); //a.k.a. compareToByAreaLeft

public int compareToByFraction(Object o); //a.k.a. compareToByFraction

public int compareToByCalories(Object o);

}

PizzaException:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class PizzaException extends RuntimeException{

public PizzaException() {

super();

}

//Constructor that passes the passed String to RuntimeException

public PizzaException(String msg){

super(msg);

}

}

PizzaManager:

package pizzamanager;

import java.util.Scanner;

/\*\*

\*

\* @author JMC

\*/

public class PizzaManager {

ArrayList<Pizza> pizzas = new ArrayList<Pizza>();

/\*

\* The console interface is defined in the start method

\*/

public void start() {

char selection='q';

Scanner foo = new Scanner(System.in);

//Beginning of while loop of main program

while(true) {

displayAllPizzas();

displayInstructions();

//Obtain the char of the first typed character input

selection = foo.next().charAt(0);

switch(selection) {

case 'A':

case 'a': System.out.println("Adding a random pizza to the ArrayList<Pizza>.");

//Call addRandomPizza to add a new Pizza to the Pizza Array List

addRandomPizza();

break;

case 'H':

case 'h': System.out.println("Adding 5 random pizzas to the ArrayList<Pizza>.");

//To add 100 random Pizzas to the Array List, I run a loop for 100 times and call

//the same addRandomPizza method

for(int i=0;i<5;i++){

addRandomPizza();

}

break;

case 'E':

case 'e': System.out.println("Eating a fraction of a pizza. How much?");

//Created a scanner to take the fraction input

Scanner fracInput = new Scanner(System.in);

//Grab the next input as a String to help with parsing the fraction input

String frac = fracInput.next();

//Split user string input by the / sign

//Then obtain the Integer value of each value

//for the numerator and denominator

//Using regular expression for the splitting

String[] fraction = String.valueOf(frac).split("\\/");

int num = Integer.valueOf(fraction[0]);

int denom = Integer.valueOf(fraction[1]);

//System.out.println("Denom: " + denom);

//System.out.println("Num: " + num);

//Prompt user for additional input for the pizza index to eat

System.out.println("At which pizza index?");

//Take in input as an int

int fractionIndex = fracInput.nextInt();

//Call method eatSomePizza and pass in the user generated fraction and index

eatSomePizza(new Fraction(num,denom),fractionIndex);

break;

case 'F':

case 'f': System.out.println("QuickSorting pizzas by (F)raction");

//WOW, SO MUCH BONUS

//Sorting Pizzas by Fractions because it's cool

//I added in another PizzaComparable method for this to work

quickSortByFraction();

break;

case 'P':

case 'p': System.out.println("QuickSorting pizzas by (P)rice");

//Sorting Pizzas by price using the quicksort algorithm, calls quickSortByPrice

quickSortByPrice();

break;

case 'S':

case 's': System.out.println("QuickSorting pizzas by (S)ize");

//Sorting Pizzas by Size using the quicksort algorithm, calls quickSorBySize

quickSortBySize();

break;

case 'C':

case 'c': System.out.println("QuickSorting pizzas by (C)alories");

//Sorting Pizzas by Calories using the quicksort algorithm, calls quickSorByCalories

quickSortByCalories();

break;

case 'B':

case 'b': System.out.println("(B)inary search over pizzas by calories(int). \nQuickSorting first. \nWhat calorie count are you looking for?");

//Method for finding a Pizza by the number of Calories

//Uses the binary search algorithm and first sorts by Calories

Scanner calInput = new Scanner(System.in);

//Asks user for Calories to search and find

int cals = calInput.nextInt();

int index = binarySearchByCalories(cals);

//If the index returned by the search function is -1, then a pizza with the specified calories wasn't found

if(index == -1){

System.out.println("Pizza with " + cals + " calories not found.");

}

else {

//Prints that a pizza with the specified calories was found and lists the index location

System.out.println("Pizza with " + cals + " calories found:");

System.out.println(pizzas.get(index) + "\n\n");

}

break;

case 'R':

case 'r': System.out.println("(R)everse order of Pizzas with a Stack" );

//This reverses the current order of Pizzas in the ArrayList

//It doesn't sort first

//Creating a stack for reversing the order

PizzaStack stacky = new PizzaStack();

int numElements = pizzas.size();

//Loops through pizza Array List and pushes the results onto the stack

for(int i=0; i != pizzas.size();){

Pizza temp = pizzas.remove(i);

//System.out.println(temp);

stacky.push(temp);

}

//While the stack isn't empty, the stack

//pushes the results back into thte pizza Array List

//This results in reversing the order of the list

while(!stacky.isEmpty()){

Pizza temp2 = (Pizza) stacky.pop();

//System.out.println(temp2);

pizzas.add(temp2);

}

System.out.println(pizzas.size());

break;

case 'Q':

case 'q': System.out.println("(Q)uitting!" );

//Exit program

System.exit(0);

break;

default: System.out.println("Unrecognized input - try again");

}

}

}

/\*

a.This function subtracts the amount from the remaining pizza.

i. To accomplish this, add a subtract function to the Fraction class

ii. If the ratio reaches zero, throw a PizzaException and catch it in the PizzaManager class so this pizza is removed

iii. If the ratio is negative, throw a PizzaException as this is an error case.

\*/

public void eatSomePizza(Fraction amount, int index){

if(amount == null || !(amount instanceof Fraction)){

throw new PizzaException("Invalid fraction");

}

//System.out.println(amount.toString() + index);

//Try catch to eat an amount from a pizza

//The amount is a fraction of the whole pizza and not a fraction of the available pizza

try{

pizzas.get(index).eatSomePizza(amount);

}

//Catching the Pizza Exception that will be thrown when the pizza is entirely eaten

catch(PizzaException p){

//Print exception message

System.out.println(p.toString());

//Remove pizza from pizza list as there isn't anymore of it available

pizzas.remove(index);

}

}

//The addRandomPizza method calls the add function on the pizza array list

//and adds a pizza. a pizza created with no args will be randomly generated

private void addRandomPizza() {

pizzas.add(new Pizza());

}

//Write a function to list all Pizzas in their current order in your ArrayList

//This displays all Pizzas in the console

private void displayAllPizzas() {

//Loops over pizza array list size and calls get on pizzas to return the current pizza

for(int i=0; i<pizzas.size(); i++){

System.out.println(pizzas.get(i));

}

}

/\*

\* 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

\*/

//Sorting by price method using the quick sort algorithm

//Facade function

private void quickSortByPrice() {

int first = 0;

int last = pizzas.size()-1;

sortPrice(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortPrice(ArrayList<Pizza> pizs, int first, int last) {

//Check if passed pizza array list is actually a pizza array list

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

System.out.println("Passed first must be >= 0 and last values must be >= -1");

System.exit(-1);

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionPrice(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortPrice(pizs, first, pivotIndex - 1);

sortPrice(pizs, pivotIndex + 1, last);

}

}

private int partitionPrice(ArrayList<Pizza> pizs, int left, int right){

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareTo(pivotValue));

if(pizs.get(i).compareTo(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

/\*

\* Build a function to display pizzas with the largest remaining areas first

\* 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

\*/

//Sorting by size method using the quick sort algorithm

//Facade function

private void quickSortBySize() {

int first = 0;

int last = pizzas.size()-1;

sortSize(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortSize(ArrayList<Pizza> pizs, int first, int last) {

//Validate pizza array list input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

throw new PizzaException("Invalid index");

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionSize(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortSize(pizs, first, pivotIndex - 1);

sortSize(pizs, pivotIndex + 1, last);

}

}

private int partitionSize(ArrayList<Pizza> pizs, int left, int right){

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareTo(pivotValue));

if(pizs.get(i).compareToBySize(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

//Sorting by fraction method using the quick sort algorithm

//Facade function

private void quickSortByFraction() {

int first = 0;

int last = pizzas.size()-1;

sortFraction(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortFraction(ArrayList<Pizza> pizs, int first, int last) {

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

throw new PizzaException("Invalid index");

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionFraction(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortFraction(pizs, first, pivotIndex - 1);

sortFraction(pizs, pivotIndex + 1, last);

}

}

private int partitionFraction(ArrayList<Pizza> pizs, int left, int right){

//Check for valid pizza array list input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareTo(pivotValue));

if(pizs.get(i).compareToByFraction(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

/\*

\* function that sorts Pizzas in order of greatest calories first

\* Notice you’re sorting primitives here

\* a. To make your life easier, 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.

\*/

//Sorting by size method using the quick sort algorithm

//Facade function

public void quickSortByCalories() {

int first = 0;

int last = pizzas.size()-1;

sortCalories(pizzas, first, last);

}

//Function that does work but is private so user can't partially sort list

private void sortCalories(ArrayList<Pizza> pizs, int first, int last) {

//Validate passed Pizzas array list input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

System.out.println("Passed first must be >= 0 and last values must be >= -1");

System.exit(-1);

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partitionCalorie(pizs, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sortCalories(pizs, first, pivotIndex - 1);

sortCalories(pizs, pivotIndex + 1, last);

}

}

private int partitionCalorie(ArrayList<Pizza> pizs, int left, int right){

//Validate pizza arraylist input

if(pizs == null || !(pizs instanceof ArrayList<?>)){

throw new PizzaException("Invalid fraction");

}

//System.out.println("Left: " + left);

//System.out.println("Right: " + right);

//Using median of three for pivot

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

Pizza pivotValue = pizs.get(pivotIndex);

//System.out.println(pivotValue);

//Swap the pivot index temporarily to the furthest right index of the sub array

pizs.swap(pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

//System.out.println(pizs.get(i).compareToByCalories(pivotValue));

if(pizs.get(i).compareToByCalories(pivotValue) < 0){

pizs.swap(i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

pizs.swap(storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

/\*

\* Build a function that searches over pizzas using their calorie count.

\* Be sure to call quickSortByCalories() first so the data is sorted!

\*/

//Binary search by calories takes in calorie input as target

//utilizes binary search algorithm

//Sorts by calories first to ensure algorithm will work

private int binarySearchByCalories(int cals) {

quickSortByCalories();

int low = 0;

int high = pizzas.size()-1;

int mid = 0;

//While the current low boundary is lower than the current high boundary

while(low <= high){

//Obtain middle result by adding and dividing low and high boundaries

mid = (low + high)/2;

//See if the middle result matches the target and returns middle index if does

if(pizzas.get(mid).getCalories() == cals){

return mid;

}

//If the

else if(pizzas.get(mid).getCalories() < cals){

//If the guessed middle value is lower than the target

//Then low is assigned the middle index + 1

low = mid+1;

}

else {

//If the guessed middle value is greater than the target

//Then high is assigned the middle index -1

high = mid-1;

}

}

//Returning -1 when no matches are found

return -1;

}

/\*

\* No need to edit functions below this line, unless extending the menu or

\* changing the instructions

\*/

private static final String instructions = "-----------------------\nWelcome to PizzaManager\n-----------------------\n(A)dd a random pizza\nAdd a (H)undred random pizzas\n(E)at a fraction of a pizza\nBONUS ROUND - QuickSorting pizzas by (F)raction\nQuickSort pizzas by (P)rice\nQuickSort pizzas by (S)ize\nQuickSort pizzas by (C)alories\n(B)inary Search pizzas by calories\n(R)everse order of pizzas with a stack\n(Q)uit\n";

private void displayInstructions() {

System.out.println(instructions);

}

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

new PizzaManager().start();

}

}

PizzaStack:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class PizzaStack {

int capacity = 100; //Setting capacity to start at 100 and later can update if reach capacity.

Object data[] = new Object[capacity]; //Initializing Object array with size of capacity variable.

int numElements = 0; //Start number of elements at 0

//Default constructor

public PizzaStack() {}

//Copy constructor. This copies the primitives of each attribute.

public PizzaStack(PizzaStack other){

this.capacity = other.capacity;

this.numElements = other.numElements;

for(int i=0; i<other.numElements; i++){ //Iterate through array and copy each object

this.data[i] = other.data[i];

}

}

//Is empty method checks to see if numElements is 0 and returns boolean value

public boolean isEmpty() {

if(numElements == 0){

return true;

}

return false;

}

//Size returns numElements

public int size(){

return numElements;

}

//push method will add passed object to top of stack. Will check to see if object array will need to be increased.

public void push(Pizza o){

if(capacity == numElements){ //If capacity is equal to numElements, dynamically increase array

capacity = capacity \* 2;

Object tempData[] = new Object[capacity];

for(int i=0;i<numElements;i++){ //Copy all objects to new temp array

tempData[i] = (Pizza) data[i];

}

tempData[numElements++] = o; //Set new value in the larger temp array

data = tempData; //Set original array to new address of larger temp array

} else{ //If resize isn't needed, adds object to top of Object array

data[numElements] = o;

numElements++;

}

}

//Pop returns the last item in the array and decreases numElements

public Object pop(){

if(numElements == 0){ //If there are zero elements, exit

System.out.println("Stack is already empty.");

System.exit(-1);

}

if(numElements > 0) {

Object retval = data[numElements-1]; //Set retval

numElements--; //Decrease numElements

return retval; //Return retval

}

return "Stack is empty";

}

//Override default object toString that normally returns the memory value to return a string comprising of the array objects

@Override

public String toString(){

String temp = ""; //Create temp string to hold all values

for(int i=0; i<numElements; i++){ //Iterate through data array of objects

temp += data[i].toString() + " "; //Append to temp string new object string values

}

return temp; //Return temp string with all object string values

}

@Override

public boolean equals(Object o){

if(o == null || !(o instanceof PizzaStack)){ //Validate passed object is an ObjectStack

return false;

}

PizzaStack that = (PizzaStack) o; //Cast that variable as ObjectStack of o

for(int i=0; i<numElements; i++){

if(data[i] != that.data[i]) return false; //If an object doesn't match return false

}

if(this.size() != that.size()){ //If the ObjectStack sizes are different return false

return false;

}

return true;

}

}

QuickSort:

package pizzamanager;

import java.util.Arrays;

/\*\*

\*

\* @author JMC

\*/

public class QuickSort {

public void sort(int[] data, int first, int last) {

//Verify first and last indexes passed are valid

//During a recursive call, last can be -1

if(first < 0 || last < -1){

System.out.println("Passed first must be >= 0 and last values must be >= -1");

System.exit(-1);

}

//Base case that first must be less than last

if(first < last){

//Partitioning data into two halves while returning the pivotIndex

int pivotIndex = partition(data, first, last);

//Recursively calling sort method to sort each half on either side of pivotIndex

sort(data, first, pivotIndex - 1);

sort(data, pivotIndex + 1, last);

}

}

private int partition(int[] partData, int left, int right){

int pivotIndex = left + (right - left) / 2;

//Assigning pivotValue for comparison later

int pivotValue = partData[pivotIndex];

swap(partData, pivotIndex, right);

//Store index is where the lowest values will be swapped starting on the left

int storeIndex = left;

for(int i=left; i < right; i++){

//If value of i is less than pivot value, swap it to the storeIndex

if(partData[i] <= pivotValue){

swap(partData, i, storeIndex);

storeIndex++;

}

}

//Swap and move pivot index to final index

swap(partData,storeIndex, right);

//System.out.println(Arrays.toString(partData));

return storeIndex;

}

public void swap(int[] dataSwap, int idx1, int idx2) {

int temp = dataSwap[idx1];

dataSwap[idx1] = dataSwap[idx2];

dataSwap[idx2] = temp;

}

}

Shape:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public abstract class Shape implements Cloneable, Comparable{

private int x;

//Area double for area

private double area;

//String that will hold the shape type

private String shapeType;

//Constructor that takes two arguments, the x value for a side or radius, and the type of shape

public Shape(int nx, String shape) {

setX(nx);

setShapeType(shape);

}

//Copy constructor

public Shape(Shape other){

this(other.x, other.shapeType);

}

//Setter for setting the shape type

public void setShapeType(String shape){

shapeType = shape;

}

//Getter for thet shape type

public String getShapeType(){

return shapeType;

}

//Overridding the clone method

//Implementing as an abstract method for the child classes to implement

@Override

abstract public Shape clone();

//Setter for x value, which can be a side or radius

public void setX(int nx){

if(nx>0) this.x = nx;

}

//Getter for x value, which can be a side or radius

public int getX() {

return this.x;

}

//Get area method that will be implemented in child classes

public double getArea(){

return -1;

}

//Setter for area

public void setArea(double a){

area = a;

}

//Override toString to return shape type and area

@Override

//I actually like printf LOL

public String toString() {

return String.format("Shape %d area: %.1f", getX(), getArea() );

}

//Override compareTo

//This compares Shapes based on area

@Override

public int compareTo(Object o) {

//Check for value chape input

if(o == null || !(o instanceof Shape)){

throw new PizzaException("Invalid Square");

}

Shape that = (Shape) o;

double temp = this.area - that.area;

if(temp > 0){

return 1;

}

else if(temp < 0){

return -1;

}

else{

return 0;

}

}

}

Square:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Square extends Shape{

public static final int side = 32;

private double area;

public final static String shapeType = "Square";

//Constructor that calls Shape constructor and sets the area

public Square(){

super(side, shapeType);

setArea();

}

//Copy Constructor that accepts Circle object

public Square(Square newCopy){

super(newCopy.side, newCopy.shapeType);

setArea();

}

//Set area method that calculates and sets area

public void setArea(){

this.area = this.getX() \* this.getX();

}

//Set area method that takes a double as input

public void setArea(double a){

area = a;

}

//Get area method that returns area

public double getArea(){

return area;

}

//Clone method for Square that implements the Clone method from Shape

@Override

public Shape clone() {

return new Square(this);

}

}

Susage:

package pizzamanager;

/\*\*

\*

\* @author JMC

\*/

public class Susage extends Meat{

public static final int calories = 500;

public static final Money cost = new Money(1,50);

public static final String description = "Premium sausage";

//Constructor calls Meat constructor with super

public Susage() {

super(description, cost, calories);

}

}

Vegetable:

package pizzamanager;

import java.awt.Color;

/\*\*

\*

\* @author JMC

\*/

public class Vegetable extends Ingrediants {

Color myColor;

//Constructor that passes values from child class to super class

public Vegetable(String description, Money cost, int calories, Color color) {

super(description, cost, calories, color);

}

//Getter for color

public Color getColor() {

return myColor;

}

//Setter for color

public void setColor(Object o){

//Validate color input

if(o == null || !(o instanceof Color)){

throw new PizzaException("Invalid color");

}

Color that = (Color) o;

myColor = that;

}

public boolean equals(Object o) {

if(o == null || !(o instanceof Vegetable)){

throw new PizzaException("Invalid color");

}

Vegetable that = (Vegetable) o;

if(this.myColor == that.myColor){

return true;

}

else{

return false;

}

}

}

Output:





































