/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS FoodVend

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 24/05/18

\* PURPOSE Main class for the FoodVend program.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

Main

main := CONSTRUCT MainMenu USING default

TRY

main menuStart <- none

CATCH Exception //should never be reached

TerminalUtility clearTerminal <- none

OUTPUT "Exiting with unexpected error: " + Exception MESSAGE

END TRYCATCH

END MAIN

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS FoodVend

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 24/05/18

\* PURPOSE Main class for the FoodVend program.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public class FoodVend

{

public static void main(String[] args)

{

MainMenu main = new MainMenu();

try

{

main.menuStart();

}

catch (Exception e) //avoids printing ugly stacktrace to user.

{ //should never be reached. if it is, i am a bad programmer :(

TerminalUtility.clearTerminal();

System.out.println("Exiting with unexpected error: " + e.getMessage());

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS MainMenu

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Displays menu options and calls on operations to allow user

\* to control all facilities of the program.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CLASSFIELDS:

util (TerminalUtility)

fio (FileIO)

CONSTRUCTORS:

default:

util := CONSTRUCT TerminalUtility USING default

fio := CONSTRUCT FileIO USING default

END CONSTRUCTORS

SUBMODULE: menuStart

IMPORTS: none

EXPORTS: none

ALGORITHM:

exit := FALSE

DO

choice := util menuSelect <- ("+|FoodVend Management Terminal|+" + NEWLINE +

"Please select a menu option from below:" + NEWLINE +

"[1] Add Food" + NEWLINE + "[2] Remove Food" + NEWLINE +

"[3] Display Contents" + NEWLINE + "[4] Find Expired" + NEWLINE +

"[5] Read IN Storage" + NEWLINE + "[6] Write OUT Storage" + NEWLINE +

"[0] Exit"), NULL, 6

CASE choice

1: addFood <- none

2: removeFood <- none

3: displayContents <- none

4: findExpired <- none

5: readIn <- none

6: writeOut <- none

0: exit := TRUE

OUTPUT "Exiting FoodVend Management Terminal."

END CASE

WHILE NOT exit

END SUBMODULE

SUBMODULE: addFood

IMPORTS: none

EXPORTS: none

ALGORITHM:

exit := FALSE

err := NULL

IF (fio isLoaded <- none)

DO

IF err NOT NULL AND err EQUALS "" //if it worked last loop

OUTPUT "Food added successfully"

END IF

choice := util menuSelect <- ("+|Add Food|+" + NEWLINE +

"Please select a menu option from below:" + NEWLINE +

"[1] Meat" + NEWLINE + "[2] Vegetables" + NEWLINE +

"[3] Grain" + NEWLINE + "[4] Fruit" + NEWLINE +

"[0] Return to Main Menu"), err, 4

err := ""

TRY

CASE choice

1: addMeat <- none

2: addVege <- none

3: addGrain <- none

4: addFruit <- none

0: exit := true

END CASE

CATCH ParseException

err := "Error: Invalid date."

CATCH NumberFormatException

err := "Error: Invalid number input."

CATCH Exception

err := EXCEPTION MESSAGE

END TRYCATCH

WHILE NOT exit

ELSE

storaeLackError <- none

END ELSE

END SUBMODULE

SUBMODULE: addMeat

IMPORTS: none

EXPORTS: none

ALGORITHM:

name := util inputString <- "Please input meat name:" + NEWLINE

cut := util inputString <- "Please input meat cut:" + NEWLINE

weight := util inputString <- "Please input meat weight:" + NEWLINE

temp := util inputString <- "Please input storage temperature:" + NEWLINE

useby := util inputString <-

"Please input use-by date DD/MM/YYYY" + NEWLINE

packaging := util inputString <- "Please input meat packaging:" + NEWLINE

fio addFood <- {"Meat", name, cut, weight, temp, useby, packaging}

END SUBMODULE

SUBMODULE: addVege

IMPORTS: none

EXPORTS: none

ALGORITHM:

name := util inputString <- "Please input vegetable name:" + NEWLINE

weight := util inputString <- "Please input vegetable weight:" + NEWLINE

temp := util inputString <- "Please input storage temperature:" + NEWLINE

bestBefore := util inputString <-

"Please input best before date DD/MM/YYYY:" + NEWLINE

packaging := util inputString <- "Please input vegetable packaging:"

fio addFood <- {"Vegetable", name, weight, temp, bestBefore, packaging}

END SUBMODULE

SUBMODULE: addGrain

IMPORTS: none

EXPORTS: none

ALGORITHM:

name := util inputString <- "Please input grain name:" + NEWLINE

type := util inputString <- "Please input grain type:" + NEWLINE

volume := util inputString <- "Please input grain weight:" + NEWLINE

temp := util inputString <- "Please input storage temperature:" + NEWLINE

bestBefore := util inputString <-

"Please input best before date DD/MM/YYYY:" + NEWLINE

packaging := util inputString <- "Please input grain packaging:" + NEWLINE

fio addFood <- {"Grain", name, type, volume, temp, bestBefore, packaging}

END SUBMODULE

SUBMODULE: addGrain

IMPORTS: none

EXPORTS: none

ALGORITHM:

name := util inputString <- "Please input fruit name:" + NEWLINE

type := util inputString <- "Please input fruit type:" + NEWLINE

numPiece := util inputString <- "Please input number of pieces:" + NEWLINE

temp := util inputString <- "Please input storage temperature:" + NEWLINE

useby := util inputString <-

"Please input useby date DD/MM/YYYY:" + NEWLINE

packaging := util inputString <- "Please input fruit packaging:" + NEWLINE

fio addFood <- {"Fruit", name, type, numPiece, temp, useby, packaging}

END SUBMODULE

SUBMODULE: removeFood

IMPORTS: none

EXPORTS: none

ALGORITHM:

exit := FALSE

IF fio isLoaded <- none

DO

choice := util menuSelect <- ("+|Remove Food|+" + NEWLINE +

"Please select a location from below:" + NEWLINE +

"[1] Freezer" + NEWLINE + "[2] Fridge" + NEWLINE +

"[3] Pantry" + NEWLINE + "[0] Return to Main Menu"), err, 3

err := NULL

IF choice EQUALS 0

exit := TRUE

ELSE

OUTPUT "Enter element number of food to be removed:"

element := util inputInt <- none

TRY

fio passRemove <- (choice - 1), element

CATCH IllegalArgumentException

err := EXCEPTION MESSAGE

exit := FALSE

END TRYCATCH

IF err IS NULL

OUTPUT "Food removed successfully."

END IF

END ELSE

WHILE NOT exit

ELSE

storageLackError <- none

END ELSE

END SUBMODULE

SUBMODULE: displayContents

IMPORTS: none

EXPORTS: none

ALGORITHM:

IF fio isLoaded <- none

exit := FALSE

DO

choice := util menuSelect <- ("+|Display Contents|+" + NEWLINE +

"Please select a location from below:" + NEWLINE +

"[1] Freezer" + NEWLINE + "[2] Fridge" + NEWLINE +

"[3] Pantry" + NEWLINE + "[0] Return to Main Menu"), err, 3

array := NULL

CASE choice

0: exit := true

1: array := fio passFreezer <- none

2: array := fio passFridge <- none

3: array := fio passPantry <- none

END CASE

IF ARRAY NOT NULL

printElements(array)

END IF

WHILE NOT exit

ELSE

storageLackError <- none

END ELSE

END SUBMODULE

SUBMODULE: printElements

IMPORTS: array (Array of objects)

EXPORTS: none

ALGORITHM:

element := 0

FOR i := 0 TO LENGTH OF array INC i

IF array[i] NOT NULL

INC element

OUTPUT element + ": " + array[i] toString <- none

END IF

END FOR

IF LENGTH OF array EQUALS 0 OR element EQUALS 0

OUTPUT "+|No elements found.|+"

END IF

END SUBMODULE

SUBMODULE: findExpired

IMPORTS: none

EXPORTS: none

ALGORITHM:

IF fio isLoaded <- none

exit := FALSE

choice := util menuSelect <- ("+|Find Expired|+" + NEWLINE +

"Please select a menu option from below:" + NEWLINE +

"[1] Find Expired" + NEWLINE + "[0] Return to Main Menu"), null, 1

IF choice EQUALS 1

fridge := fio passFridge <- none

pantry := fio passPantry <- none

freezer := fio passFreezer <- none

OUTPUT "+|Freezer: " + printExpired <- freezer

OUTPUT "+|Fridge: " + printExpired <- fridge

OUTPUT "+|Pantry: " + printExpired <- pantry

END IF

ELSE

storaeLackError <- none

END ELSE

END SUBMODULE

SUBMODULE: printExpired

IMPORTS: array (Array of Food objects)

ALGORITHM:

elementAt := 0

FOR i := 0 TO LENGTH OF array INC i

IF array[i] NOT NULL

INC elementAt

IF array[i] calcExpiry <- none

OUTPUT elementAt + ": " + array[i] toString <- none

END IF

END IF

END FOR

END SUBMODULE

SUBMODULE: readIn

IMPORTS: none

EXPORTS: none

ALGORITHM:

IF NOT fio isLoaded <- none

exit := false

skippedLines := 0

DO

choice := util menuSelect <- ("+|Read IN Storage|+" + NEWLINE +

"Please select a menu option from below:" + NEWLINE +

"[1] Load from File" + NEWLINE + "[0] Return to Main Menu"), err, 1

err := NULL

CASE choice

0: exit := TRUE

1: input := util inputString <- "Please input filename:"

TRY

skippedLines := fio parseFile <- input

exit := TRUE

CATCH Exception

err := EXCEPTION MESSAGE

exit := false

END TRYCATCH

END CASE

WHILE NOT exit

IF fio isLoaded <- none

OUTPUT "File load complete." + NEWLINE + skippedLines +

" elements skipped due to errors."

END IF

ELSE

storagePresentError <- none

END ELSE

END SUBMODULE

SUBMODULE: writeOut

IMPORTS: none

EXPORT: none

ALGORITHM:

IF fio isLoaded

exit := false

DO

IF err NOT NULL AND err EQUALS "" //if it worked last loop

OUTPUT "File save complete."

END IF

choice := util menuSelect <- ("+|Write OUT Storage|+" + NEWLINE +

"Please select a menu option from below:" + NEWLINE +

"[1] Write to File" + NEWLINE + "[0] Return to Main Menu"), err, 1

err := ""

CASE choice

0: exit := TRUE

1: util inputString <- "Please input filename to write to:"

TRY

fio writeStorageCSV <- filename

CATCH IOException

err := EXCEPTION MESSAGE

exit := FALSE

END TRYCATCH

END CASE

WHILE NOT exit

ELSE

storageLackError <- none

END ELSE

END SUBMODULE

SUBMODULE: storageLackError

IMPORTS: none

EXPORTS: none

ALGORITHM:

OUTPUT "+|The storage facility has not been loaded.|+" + NEWLINE +

"Please load the storage facility before attempting this task."

END SUBMODULE

SUBMODULE storagePresentError

IMPORTS: none

EXPORTS: none

ALGORITHM:

OUTPUT "+|THe storage facility has already been loaded.|+" + NEWLINE +

"Proceed with operation or restart to load a different facility"

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS MainMenu

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Displays menu options and calls on operations to allow user

\* to control all facilities of the program.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.text.ParseException;

import java.io.IOException;

public class MainMenu

{

private static TerminalUtility util;

private static FileIO fio;

public MainMenu()

{ //default constuctor

util = new TerminalUtility();

fio = new FileIO();

}

/\*####################################################################

# METHOD menuStart

# IMPORTS none

# EXPORTS none

# PURPOSE The starting point of the FoodVend program. Renders the main

# menu and brings the user to the selected submenu.

# All roads lead back to here.

\*###################################################################\*/

public void menuStart()

{

int choice;

boolean exit = false;

do

{

choice = util.menuSelect(util.YELANSI +

"+|FoodVend Management Terminal|+\n" + util.SETANSI +

"Please select a menu option from below:\n\n" +

util.REDANSI + "[1]" + util.SETANSI + " Add Food\n" +

util.REDANSI + "[2]" + util.SETANSI + " Remove Food\n" +

util.REDANSI + "[3]" + util.SETANSI + " Display Contents\n" +

util.REDANSI + "[4]" + util.SETANSI + " Find Expired\n" +

util.REDANSI + "[5]" + util.SETANSI + " Read IN Storage\n" +

util.REDANSI + "[6]" + util.SETANSI + " Write OUT Storage\n" +

util.REDANSI + "[0]" + util.SETANSI + " Exit", null, 6);

switch (choice)

{

case 1:

addFood();

break;

case 2:

removeFood();

break;

case 3:

displayContents();

break;

case 4:

findExpired();

break;

case 5:

readIn();

break;

case 6:

writeOut();

break;

case 0:

exit = true;

TerminalUtility.clearTerminal();

System.out.println("Exiting FoodVend Management Terminal...");

break;

}

}

while (!exit);

}

/\*####################################################################

# METHOD addFood

# IMPORTS none

# EXPORTS none

# PURPOSE Renders the submenu of options used to add foods and

# directs the user to the appropriate sub-submenu based

# on their selection. It's submenus return back here.

\*###################################################################\*/

private static void addFood()

{

boolean exit = false;

String err = null;

if (fio.isLoaded())

{

int choice;

do

{

if (err != null && err.equals("")) //if it worked last loop

{

TerminalUtility.clearTerminal();

System.out.println("Food added successfully.\nPress Enter" +

" to return to menu.");

util.waitForEnter();

}

choice = util.menuSelect(util.YELANSI + "+|Add Food|+\n" +

util.SETANSI + "Please select a menu option from below:\n\n" +

util.REDANSI + "[1]" + util.SETANSI + " Meat\n" +

util.REDANSI + "[2]" + util.SETANSI + " Vegetables\n" +

util.REDANSI + "[3]" + util.SETANSI + " Grain\n" +

util.REDANSI + "[4]" + util.SETANSI + " Fruit\n" +

util.REDANSI + "[0]" + util.SETANSI + " Return to Main Menu", err, 4);

err = ""; //reset

try

{

switch (choice)

{

case 1:

addMeat();

break;

case 2:

addVege();

break;

case 3:

addGrain();

break;

case 4:

addFruit();

break;

case 0:

exit = true;

break;

}

}

catch (ParseException ex)

{

err = "Error: Invalid date.";

}

catch (NumberFormatException exc)

{

err = "Error: Invalid number input.";

}

catch (Exception e)

{ //validation at each step of input would be nice but i'm behind :(

err = e.getMessage();

}

}

while (!exit);

}

else

{

storageLackError();

}

}

/\*####################################################################

# METHOD addMeat

# IMPORTS none

# EXPORTS none

# PURPOSE Takes user input for each parameter of meat and attempts to

# create an object from them and add them to the storage facility.

\*###################################################################\*/

private static void addMeat()

throws IllegalArgumentException, ParseException

{

TerminalUtility.clearTerminal();

String name = util.inputString("Please input meat name:\n");

String cut = util.inputString("Please input meat cut:\n");

String weight = util.inputString("Please input meat weight:\n");

String temp = util.inputString("Please input storage temperature:\n");

String useby = util.inputString("Please input use-by date DD/MM/YYYY:\n");

String packaging = util.inputString("Please input meat packaging:\n");

fio.addFood(new String[] {"Meat", name, cut, weight, temp,

useby, packaging}); //lazy code reuse shortcut

}

/\*####################################################################

# METHOD addVege

# IMPORTS none

# EXPORTS none

# PURPOSE Takes user input for each parameter of vegetables and attempts to

# create an object from them and add them to the storage facility.

\*###################################################################\*/

private static void addVege()

throws IllegalArgumentException, ParseException

{

TerminalUtility.clearTerminal();

String name = util.inputString("Please input vegetable name:\n");

String weight = util.inputString("Please input vegetable weight:\n");

String temp = util.inputString("Please input storage temperature:\n");

String bestBefore = util.inputString( //long string literals are long

"Please input best-before date DD/MM/YYYY:\n");

String packaging = util.inputString("Please input vegetable packaging:\n");

fio.addFood(new String[] {"Vegetable", name, weight, temp,

bestBefore, packaging}); //lazy code reuse shortcut

}

/\*####################################################################

# METHOD addGrain

# IMPORTS none

# EXPORTS none

# PURPOSE Takes user input for each parameter of grain and attempts to

# create an object from them and add them to the storage facility.

\*###################################################################\*/

private static void addGrain()

throws IllegalArgumentException, ParseException

{

TerminalUtility.clearTerminal();

String name = util.inputString("Please input grain name:\n");

String type = util.inputString("Please input grain type:\n");

String volume = util.inputString("Please input grain volume:\n");

String temp = util.inputString("Please input storage temperature:\n");

String bestBefore = util.inputString( //long string literals are long

"Please input best-before date DD/MM/YYYY:\n");

String packaging = util.inputString("Please input grain packaging:\n");

fio.addFood(new String[] {"Grain", name, type, volume, temp,

bestBefore, packaging}); //lazy code reuse shortcut

}

/\*####################################################################

# METHOD addFruit

# IMPORTS none

# EXPORTS none

# PURPOSE Takes user input for each parameter of fruit and attempts to

# create an object from them and add them to the storage facility.

\*###################################################################\*/

private static void addFruit()

throws IllegalArgumentException, ParseException

{

TerminalUtility.clearTerminal();

String name = util.inputString("Please input fruit name:\n");

String type = util.inputString("Please input fruit type:\n");

String numPiece = util.inputString("Please input number of pieces:\n");

String temp = util.inputString("Please input storage temperature:\n");

String useby = util.inputString("Please input use-by date DD/MM/YYYY:\n");

String packaging = util.inputString("Please input fruit packaging:\n");

fio.addFood(new String[] {"Fruit", name, type, numPiece, temp,

useby, packaging}); //lazy code reuse shortcut

}

/\*####################################################################

# METHOD removeFood

# IMPORTS none

# EXPORTS none

# PURPOSE Renders the submenu of options used to remove foods and

# directs the user to the appropriate sub-submenu based

# on their selection. It's submenus return back here.

\*###################################################################\*/

private static void removeFood()

{

int element, choice;

String err = null;

boolean exit = false;

if (fio.isLoaded())

{

do

{

choice = util.menuSelect(util.YELANSI + "+|Remove Food|+\n" +

util.SETANSI + "Please select a location from below:\n\n" +

util.REDANSI + "[1]" + util.SETANSI + " Freezer\n" +

util.REDANSI + "[2]" + util.SETANSI + " Fridge\n" +

util.REDANSI + "[3]" + util.SETANSI + " Pantry\n" +

util.REDANSI + "[0]" + util.SETANSI + " Return to Main Menu", err, 3);

err = "";

if (choice == 0)

{

exit = true;

}

else

{

TerminalUtility.clearTerminal();

System.out.println("Enter element number of food to be" +

" removed:\n\n");

element = util.inputInt();

try

{

fio.passRemove(choice - 1, element); //-1 because 0 = freezer, etc

}

catch (IllegalArgumentException e)

{

err = e.getMessage();

exit = false;

}

if (err != null && err.equals(""))

{

TerminalUtility.clearTerminal();

System.out.println("Food removed successfully.\nPress Enter" +

" to return to menu.");

util.waitForEnter();

}

}

}

while (!exit);

}

else

{

storageLackError();

}

}

/\*####################################################################

# METHOD displayContents

# IMPORTS none

# EXPORTS none

# PURPOSE Renders the submenu of options used to display the

# contents of the storage facility and directs the user to

# the appropriate sub-submenu based on their selection.

# It's submenus return back here.

\*###################################################################\*/

private static void displayContents()

{

if (fio.isLoaded())

{

Food[] array;

int choice;

boolean exit = false;

do

{

choice = util.menuSelect(util.YELANSI + "+|Display Contents|+\n" +

util.SETANSI + "Please select a location from below:\n\n" +

util.REDANSI + "[1]" + util.SETANSI + " Freezer\n" +

util.REDANSI + "[2]" + util.SETANSI + " Fridge\n" +

util.REDANSI + "[3]" + util.SETANSI + " Pantry\n" +

util.REDANSI + "[0]" + util.SETANSI + " Return to Main Menu", null, 3);

array = null;

switch (choice)

{

case 0:

exit = true;

break;

case 1:

array = fio.passFreezer();

break;

case 2:

array = fio.passFridge();

break;

case 3:

array = fio.passPantry();

break;

}

if (array != null)

{

TerminalUtility.clearTerminal();

printElements(array);

System.out.println("Press Enter to return to menu.");

util.waitForEnter();

}

}

while (!exit);

}

else

{

storageLackError();

}

}

/\*####################################################################

# METHOD printElements

# IMPORTS array (Array of Objects)

# EXPORTS none

# PURPOSE Outputs the toString method's export for each object in an array

# along with a numbering (not counting null elements)

\*###################################################################\*/

private static void printElements(Object[] array)

{

int element = 0;

for (int i = 0; i < array.length; i++)

{

if (array[i] != null)

{

element++;

System.out.println(element + ": " + array[i].toString());

}

}

if (array.length == 0 || element == 0)

{

System.out.println(util.YELANSI +

"+|No elements found.|+" + util.SETANSI);

}

}

/\*####################################################################

# METHOD findExpired

# IMPORTS none

# EXPORTS none

# PURPOSE Organizes displaying the expired contents of storage

\*###################################################################\*/

private static void findExpired()

{

if (fio.isLoaded())

{

boolean exit = false;

int choice = util.menuSelect(util.YELANSI + "+|Find Expired|+\n" +

util.SETANSI + "Please select a menu option from below:\n\n" +

util.REDANSI + "[1]" + util.SETANSI + " Find Expired\n" +

util.REDANSI + "[0]" + util.SETANSI + " Return to Main Menu", null, 1);

if (choice == 1)

{

Food[] fridge = fio.passFridge();

Food[] pantry = fio.passPantry();

Food[] freezer = fio.passFreezer();

TerminalUtility.clearTerminal();

System.out.println(util.YELANSI + "+|Freezer: " + util.SETANSI);

printExpired(freezer);

System.out.println(util.YELANSI + "+|Fridge: " + util.SETANSI);

printExpired(fridge);

System.out.println(util.YELANSI + "+|Pantry: " + util.SETANSI);

printExpired(pantry);

System.out.println("\nPress Enter to continue to main menu.");

util.waitForEnter();

}

}

else

{

storageLackError();

}

}

/\*####################################################################

# METHOD printExpired

# IMPORTS none

# EXPORTS none

# PURPOSE Scans through an array of Foods and prints their toString

# method along with the number of which element it is.

# The element count is distinct to the index, as the index count

# also counts null/empty indices and counts from 0.

\*###################################################################\*/

private static void printExpired(Food[] array)

{

int elementAt = 0;

for (int i = 0; i < array.length; i++)

{

if (array[i] != null)

{

elementAt++;

if (array[i].calcExpiry())

{

System.out.println(elementAt + ": " + array[i].toString());

}

}

}

}

/\*####################################################################

# METHOD readIn

# IMPORTS none

# EXPORTS none

# PURPOSE Takes string input for a filename and organizes loading the

# file at that filename.

\*###################################################################\*/

private static void readIn()

{

if (!fio.isLoaded())

{

boolean exit = false;

int choice;

int skippedLines = 0;

String err = null;

do

{

choice = util.menuSelect(util.YELANSI + "+|Read IN Storage|+\n" +

util.SETANSI + "Please select a menu option from below:\n\n" +

util.REDANSI + "[1]" + util.SETANSI + " Load From File\n" +

util.REDANSI + "[0]" + util.SETANSI + " Return to Main Menu", err, 1);

err = null;

switch (choice)

{

case 0:

exit = true;

break;

case 1:

TerminalUtility.clearTerminal();

String input = util.inputString("Please input filename to load:");

try

{

skippedLines = fio.parseFile(input);

exit = true;

}

catch (Exception e)

{

err = e.getMessage();

exit = false;

}

break;

}

}

while (!exit);

if (fio.isLoaded())

{

TerminalUtility.clearTerminal();

System.out.println("File load complete.\n" +

skippedLines + " elements skipped due to errors.\n\n" +

"Press Enter to return to main menu.");

util.waitForEnter();

}

}

else

{

storagePresentError();

}

}

/\*####################################################################

# METHOD writeOut

# IMPORTS none

# EXPORTS none

# PURPOSE Takes string input for a filename and organizes writing

# out storage contents to a file at that filename.

\*###################################################################\*/

private static void writeOut()

{

if (fio.isLoaded())

{

int choice;

boolean exit = false;

String err = null;

do

{

if (err != null && err.equals("")) //if it worked last loop

{

TerminalUtility.clearTerminal();

System.out.println("File save complete.\n" +

"Press Enter to return to menu.");

util.waitForEnter();

}

choice = util.menuSelect(util.YELANSI + "+|Write OUT Storage|+\n" +

util.SETANSI + "Please select a menu option from below:\n\n" +

util.REDANSI + "[1]" + util.SETANSI + " Write to File\n" +

util.REDANSI + "[0]" + util.SETANSI + " Return to Main Menu", err, 1);

err = "";

switch (choice)

{

case 0:

exit = true;

break;

case 1:

TerminalUtility.clearTerminal();

String filename = util.inputString(

"Please input filename to write to:");

try

{

fio.writeStorageCSV(filename);

}

catch (IOException e)

{

err = e.getMessage();

exit = false;

}

break;

}

}

while (!exit);

}

else

{

storageLackError();

}

}

/\*####################################################################

# METHOD storageLackError

# IMPORTS none

# EXPORTS none

# PURPOSE Prints a message stating that the storage facility has not

# been loaded yet and pauses until enter is pressed before

# returning to the previous submenu.

\*###################################################################\*/

private static void storageLackError()

{

TerminalUtility.clearTerminal();

System.out.println(util.YELANSI +

"+|The storage facility has not been loaded.|+\n" + util.SETANSI +

"Please load the storage facility before attempting this task.\n\n" +

"Press Enter to return to menu");

util.waitForEnter();

}

/\*####################################################################

# METHOD storageLackError

# IMPORTS none

# EXPORTS none

# PURPOSE Prints a message stating that the storage facility has already

# been loaded yet and pauses until enter is pressed before

# returning to the previous submenu.

\*###################################################################\*/

private static void storagePresentError()

{

TerminalUtility.clearTerminal();

System.out.println(util.YELANSI +

"+|The storage facility has already been loaded.|+\n" + util.SETANSI +

"Proceed with operation, or restart to load a different facility\n\n" +

"Press Enter to return to menu.");

util.waitForEnter();

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS TerminalUtility

\* AUTHOR Jhi Morris

\* DATE CREATED 06/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Handles user input from terminal and terminal output utilities.

\* This class is generic and was built to be reused easily

\* in future programs.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

SUBMODULE: menuSelect

IMPORTS: desc (String), error (String), max (Integer)

EXPORTS: choice (Integer)

ALGORITHM:

IF error IS NULL

error := ""

END IF

DO

OUTPUT error

IF NOT valid

OUTPUT "Invalid input. Please enter 0 - " max

END IF

INPUT choice

IF choice <= max AND choice >= 0

valid := TRUE

ELSE

valid := FALSE

END ELSE

WHILE NOT valid

END SUBMODULE

SUBMODULE: inputString

IMPORTS: output (String)

EXPORTS: input (String)

ALGORITHM:

OUTPUT output

INPUT input //if only all code was this simple

END SUBMODULE

SUBMODULE: inRange

IMPORTS: value (Integer), bound1 (Integer), bound2 (Integer)

EXPORTS: out (Boolean)

ALGORITHM:

out := FALSE

IF bound >= bound 2

IF value >= bound 2 AND value <= bound1

out := TRUE

END IF

ELSE

IF value >= bound1 AND value <= bound2

out := TRUE

END IF

END ELSE

END SUBMODULE

SUBMODULE: inRange //same as above but real numbers

IMPORTS: value (Real), bound1 (Real), bound2 (Real)

EXPORTS: out (Boolean)

ALGORITHM:

out := FALSE

IF bound >= bound 2

IF value >= bound 2 AND value <= bound1

out := TRUE

END IF

ELSE

IF value >= bound1 AND value <= bound2

out := TRUE

END IF

END ELSE

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS TerminalUtility

\* AUTHOR Jhi Morris

\* DATE CREATED 06/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Handles user input from terminal and terminal output utilities.

\* This class is generic and was built to be reused easily

\* in future programs.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.\*;

import java.io.\*;

public class TerminalUtility

{

public final String YELANSI = "\u001b[33m"; //set text color yellow

public final String SETANSI = "\u001b[0m"; //resets coloring

public final String REDANSI = "\u001b[31m"; //set text color red

private Scanner sc = new Scanner(System.in);

/\*####################################################################

# METHOD menuSelect

# IMPORTS description (String), error (String), max (Integer)

# EXPORTS choice (Integer)

# PURPOSE Renders the description, takes integer as user input input

# and exports selection if valid; as defined as a non-negative

# integer less-than or equal to max.

\*###################################################################\*/

public int menuSelect(String desc, String error, int max)

{

int choice;

boolean valid = true;

if (error == null)

{

error = "";

}

do

{

clearTerminal();

System.out.println(desc + "\n\n" + REDANSI + error + SETANSI);

if (!valid)

{

System.out.print("\033[1A\033[2K" + YELANSI +

"<Invalid input. Please enter 0 - " + max + ".> \n\033[2K" + SETANSI);

} //up line, clearline, message, newline, clearline

choice = inputInt();

if (choice <= max && choice >= 0)

{

valid = true;

}

else

{

valid = false;

}

}

while (!valid);

return choice;

}

/\*####################################################################

# METHOD clearTerminal

# IMPORTS none

# EXPORTS none

# PURPOSE Clears the terminal using ANSI escape codes.

\*###################################################################\*/

public static void clearTerminal()

{

System.out.print("\033[2J\033[H");

System.out.flush();

}

/\*####################################################################

# METHOD inputInt

# IMPORTS none

# EXPORTS input (Integer)

# PURPOSE Takes input from user, prints relevant error to screen on

# non-integer input. Loops until valid integer is input.

\*###################################################################\*/

public int inputInt()

{

int input = 0;

boolean valid;

do

{

try

{

input = sc.nextInt();

sc.nextLine(); //eat EOL token

valid = true;

}

catch(InputMismatchException e)

{

String flush = sc.next();

System.out.print("\033[2A\033[2K" + YELANSI + //move up 2 lines, clear

"<Invalid input. Please input integer.> \n\033[2K" + SETANSI); //cont

valid = false; //^message, newline, clear line

}

}

while (!valid);

return input;

}

/\*####################################################################

# METHOD inputDouble

# IMPORTS none

# EXPORTS input (Double)

# PURPOSE Takes input from user, prints relevant error to screen on

# non-double input. Loops until valid double is input.

\*###################################################################\*/

public double inputDouble()

{

double input = 0.0;

boolean valid = true;

do

{

try

{

input = sc.nextDouble();

sc.nextLine(); //eat EOL

valid = true;

}

catch(InputMismatchException e)

{

String flush = sc.next();

System.out.print("\033[2A\r" + //mov cursor up 2, return to line start

"<Invalid input. Please input double.> \n\033[2K");

valid = false; //^message, newline, clear line

}

}

while (!valid);

return input;

}

/\*####################################################################

# METHOD inputString

# IMPORTS output (String)

# EXPORTS input (Integer)

# PURPOSE Prints output to screen and takes string input from user.

\*###################################################################\*/

public String inputString(String output)

{

System.out.println(output);

String input = sc.nextLine();

return input;

}

/\*####################################################################

# METHOD waitForEnter

# IMPORTS none

# EXPORTS none

# PURPOSE Acts to pause the program until the user presses enter.

\*###################################################################\*/

public void waitForEnter()

{

String flush = sc.nextLine();

}

/\*####################################################################

# METHOD inRange

# IMPORTS value (Integer), bound1 (Integer), bound2 (Integer)

# EXPORTS out (Boolean)

# PURPOSE Checks if value is between bounds.

# ASSERTION Returns true if value is between bound1 and bound2 (inclusive)

# regardless of order bound1 and bound2 arguments are passed.

\*###################################################################\*/

public static boolean inRange(int value, int bound1, int bound2)

{

boolean out = false;

if (bound1 >= bound2)

{

if (value >= bound2 && value <= bound1)

{

out = true;

}

}

else

{

if (value >= bound1 && value <= bound2)

{

out = true;

}

}

return out;

}

/\*####################################################################

# METHOD inRange

# IMPORTS value (Double), bound1 (Double), bound2 (Double)

# EXPORTS out (Boolean)

# PURPOSE Checks if value is between bounds.

# ASSERTION Returns true if value is between bound1 and bound2 (inclusive)

# regardless of order bound1 and bound2 arguments are passed.

\*###################################################################\*/

public static boolean inRange(double value, double bound1, double bound2)

{

boolean out = false;

if (bound1 >= bound2)

{

if (value >= bound2 && value <= bound1)

{

out = true;

}

}

else

{

if (value >= bound1 && value <= bound2)

{

out = true;

}

}

return out;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS FileIO

\* AUTHOR Jhi Morris

\* DATE CREATED 10/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Creates and handles storage facility, manages reading and

\* writing from/to files.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CLASSFIELDS:

storFac (StorageFacility object)

PASSERS:

passFreezer

EXPORTS: storFac getFreezer

passFridge

EXPORTS: storFac getFridge

passPantry

EXPORTS: storFac getPantry

passRemove

IMPORTS: index (Integer), location (Integer)

ALGORITHM:

storFac removeFood <- index, location

passAdd

IMPORTS: food (Food object)

ALGORITHM:

storFac addFood <- food

SUBMODULE: isLoaded

IMPORTS: none

EXPORT: loaded (Boolean)

ALGORITHM:

loaded := false

IF storFac IS NOT NULL

loaded := true

END IF

END SUBMODULE

SUBMODULE: parseFile

IMPORTS: filename (String)

EXPORTS skippedLines (Integer)

ALGORITHM:

skippedLines := 0

csv := loadCSV <- filename

sizes := readHeader <- csv

storFac := CONSTRUCT StorageFacility USING sizes

FOR i := 3 TO LENGTH OF csv, INC i

TRY

addFood <- csv[i]

CATCH Exception

INC skippedLines

END TRYCATCH

END FOR

END SUBMODULE

SUBMODULE: loadCSV

IMPORTS: filename (String)

EXPORTS: array (Array of array of strings)

ALGORITHM:

TRY

fileStrm := OPENFILE filename

lineCount := 0

INPUT line FROM fileStrm

WHILE line NOT NULL

INC lineCount

INPUT line FROM fileStrm

ENDWHILE

CLOSEFILE fileStrm

CATCH IOException

IF filestrm NOT NULL

TRY

CLOSEFILE fileStrm

CATCH IOException

END TRYCATCH

END IF

FAIL IOException "Error: File not found."

END TRYCATCH

TRY

fileStrm := OPENFILE filename

lineNum := 0

INPUT line FROM fileStrm

FOR i := 0 TO lineCount INC i

array[lineNum] := processLine <- line

INC lineNum

INPUT line FROM fileStrm

END FOR

CLOSEFILE fileStrm

CATCH ArrayIndexOutOfBounds

IF fileStrm NOT NULL

TRY

CLOSEFILE fileStrm

CATCH IOException

END TRYCATCH

END IF

FAIL IOException "Error: Non-CSV file found."

END TRYCATCH

END SUBMODULE

SUBMODULE: processLine

IMPORTS: csvRow (String)

EXPORTS: array (Array of strings)

ALGORITHM:

array := SPLIT csvRow DELIMITED BY ','

END SUBMODULE

SUBMODULE: readHeader

IMPORTS: array (Array of strings)

EXPORTS: sizes (Array of integers)

ALGORITHM:

freezer := -1 //init

fridge := -1

pantry := -1

FOR i := 0 TO 3 INC i

TRY

IF array[i][1] AS INTEGER < 0

FAIL IllegalArgumentException "Error: Negative store size."

END IF

IF array[i][0] EQUALS "freezer"

freezer := array[i][1] AS INTEGER

ELSE

IF array[i][0] EQUALS "fridge"

fridge := array[i][1] AS INTEGER

ELSE

IF array[i][0] EQUALS "pantry"

pantry := array[i][1] AS INTEGER

ELSE

FAIL IllegalArgumentException "Error: Cannot parse data."

END ELSE

END ELSE

END ELSE

CATCH NumberFormatException

FAIL IllegalArgumentException "Error: Cannot parse data."

CATCH ArrayIndexOutOfBoundsException

FAIL IllegalArgumentException "Error: Invalid header data."

END TRYCATCH

END FOR

IF (fridge EQUALS -1 OR freezer EQUALS -1 OR pantry EQUALS -1)

FAIL IllegalArgumentException "Error: Invalid header data"

ENDIF

sizes := {freezer, fridge, pantry}

END SUBMODULE

SUBMODULE: addFood

IMPORTS: line (Array of strings)

EXPORTS: none

ALGORITHM:

IF line[0] EQUALS "meat"

meat := CONSTRUCT Meat USING default

meat setName <- line[1]

meat setCut <- line[2]

meat setWeight <- line[3] TO Real

meat setStorageTemp <- line[4] TO Real

meat setUseby <- line[5] TO Date Object

meat setPackaging <- line[6]

storFac addFood <- meat

ELSE

IF line[0] EQUALS "grain"

grain := CONSTRUCT Grain USING default

grain setName <- line[1]

grain setType <- line[2]

grain setVolume <- line[3] TO Real

grain setStorageTemp <- line[4] TO Real

grain setBestBefore <- line[5] TO Date Object

grain setPackaging <- line[6]

storFac addFood <- grain

ELSE

IF line[0] EQUALS "fruit"

fruit := CONSTRUCT Fruit USING default

fruit setName <- line[1]

fruit setType <- line[2]

fruit setNumPiece <- line[3] TO Integer

fruit setStorageTemp <- line[4] TO Real

fruit setUseby <- line[5] TO Date Object

fruit setPackaging <- line[6]

storFac addFood <- fruit

ELSE

IF line[0] EQUALS "vegetable"

vegetable := CONSTRUCT Vegetable USING default

vegetable setName <- line[1]

vegetable setWeight <- line[2] TO Real

vegetable setStorageTemp <- line[3] TO Real

vegetable setBestBefore <- line[4] TO Date Object

vegetable setPackaging <- line[5]

storFac addFood <- vegetable

ELSE

FAIL IllegalArgumentException "Error: Invalid sort."

END ELSE

END ELSE

END ELSE

END ELSE

END SUBMODULE

SUBMODULE: writeStorageCSV

IMPORTS: filename (String)

EXPORTS: none

ALGORITHM:

storage := storFac getStorage <- none

freezerLength := LENGTH OF storage[0]

fridgeLength := LENGTH OF storage[1]

pantryLength := LENGTH OF storage[2]

TRY

fileStrm := OPENFILE filename

OUTPUT "Freezer, " AND freezerLength TO fileStrm

OUTPUT "Fridge, " AND fridgeLength TO fileStrm

OUTPUT "Pantry, " AND pantryLength to fileStrm

FOR i := 0 TO LENGTH OF storage INC i

FOR j := 0 TO LENGTH OF storage[i] INC j

IF storage[i][j] NOT NULL

OUTPUT storage[i][j] toCSV TO fileStrm

END IF

END FOR

END FOR

CLOSEFILE fileStrm

CATCH IOException

IF fileStrm NOT NULL

TRY

CLOSEFILE fileStrm

CATCH IOException

END TRYCATCH

END IF

FAIL IOException "Error: Unable to write to file."

END TRYCATCH

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS FileIO

\* AUTHOR Jhi Morris

\* DATE CREATED 10/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Creates and handles storage facility, manages reading and

\* writing from/to files.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.io.\*;

import java.util.Date;

import java.util.StringTokenizer;

import java.text.SimpleDateFormat;

import java.text.ParseException;

class FileIO

{

private StorageFacility storFac;

//PASSERS (See Design Philosophy document.)

public Food[] passFreezer()

{

return storFac.getFreezer();

}

public Food[] passFridge()

{

return storFac.getFridge();

}

public Food[] passPantry()

{

return storFac.getPantry();

}

public void passRemove(int index, int location)

{

storFac.removeFood(index, location);

}

public void passAdd(Food food)

{

storFac.addFood(food);

}

/\*####################################################################

# METHOD isLoaded

# IMPORTS none

# EXPORTS loaded (Boolean)

# ASSERTION Returns true if the storage facility exists.

\*###################################################################\*/

public boolean isLoaded()

{

boolean loaded = false;

if (storFac != null)

{

loaded = true;

}

return loaded;

}

/\*####################################################################

# METHOD parseFile

# IMPORTS filename (String)

# EXPORTS skippedLines (Integer)

# PURPOSE Organizes loading a file from csv, reading its header data,

# creating a storage facility with appropriate sizes, filling the

# facility's storage locations with data from the CSV rows.

# ASSERTION Will load and process file & contents at given filename if

# contents are valid. Returns reported number of skipped data.

\*###################################################################\*/

public int parseFile(String filename) throws IOException

{

int skippedLines = 0; //counts errors during construction

int[] sizes;

String[][] csv;

csv = loadCSV(filename);

sizes = readHeader(csv);

this.storFac = new StorageFacility(sizes);

for (int i = 3; i < csv.length; i++) //start at 4th line

{

try

{

addFood(csv[i]);

}

catch (Exception e)

{

skippedLines++;

}

}

return skippedLines;

}

/\*####################################################################

# METHOD loadCSV

# IMPORTS filename (String)

# EXPORTS array (2D array of strings)

# PURPOSE Loads a single CSV file

# ASSERTION Array with contents of CSV at filename will be returned

# (rows as rows, comma seperated values as column contents)

\*###################################################################\*/

public String[][] loadCSV(String filename) throws IOException

{

FileInputStream fileStrm = null;

InputStreamReader rdr;

BufferedReader bufRdr;

int lineCount;

String line;

String[][] array;

try

{

fileStrm = new FileInputStream(filename);

rdr = new InputStreamReader(fileStrm);

bufRdr = new BufferedReader(rdr);

lineCount = 0;

line = bufRdr.readLine();

while (line != null)

{ //count the number of lines in the file

lineCount++;

line = bufRdr.readLine();

}

fileStrm.close();

}

catch (IOException e)

{

if (fileStrm != null)

{

try

{

fileStrm.close();

}

catch (IOException ex)

{

//move along, nothing to see here

}

}

throw new IOException("Error: File not file found.");

}

try

{

fileStrm = new FileInputStream(filename);

rdr = new InputStreamReader(fileStrm);

bufRdr = new BufferedReader(rdr);

array = new String[lineCount][];

line = bufRdr.readLine();

for(int i = 0; i < lineCount; i++)

{ //process each line in the file

array[i] = processLine(line);

line = bufRdr.readLine();

}

fileStrm.close();

}

catch (ArrayIndexOutOfBoundsException e)

{

if (fileStrm != null)

{

try

{

fileStrm.close();

}

catch (IOException ex)

{

//who cares?

}

}

throw new IOException("Error: Non-CSV file found.");

}

return array;

}

/\*####################################################################

# METHOD processLine

# IMPORTS csvRow (String)

# EXPORTS line (Array of strings)

# PURPOSE Seperates a string of comma seperated values into an array of

# those seperate values, seperated. S E P E R A T E

\*###################################################################\*/

private String[] processLine(String csvRow)

{

String thisToken = null;

StringTokenizer strTok;

int columnNum = 0;

strTok = new StringTokenizer(csvRow, ",");

while (strTok.hasMoreTokens())

{ //count the number of comma seperated values in the line

strTok.nextToken();

columnNum++;

}

String[] line = new String[columnNum];

strTok = new StringTokenizer(csvRow, ",");

for(int i = 0; i < columnNum; i++)

{ //process the values into an array

line[i] = strTok.nextToken();

}

return line;

}

/\*####################################################################

# METHOD readHeader

# IMPORTS array (2D array of strings)

# EXPORTS sizes (Array of integers)

# PURPOSE Parses header data of given compatible array and validates

# that no data is missing or invalid.

# ASSERTION Returns the sizes of the storage compartments reported.

\*###################################################################\*/

private int[] readHeader(String[][] array) throws IllegalArgumentException

{

int freezer = -1; //init

int fridge = -1;

int pantry = -1;

int sizes[];

for (int i = 0; i < 3; i++)

{

try

{

if (Integer.parseInt(array[i][1].trim()) < 0)

{

throw new IllegalArgumentException("Error: Negative store size on" +

" line " + i + ".");

}

if (array[i][0].trim().toLowerCase().equals("freezer"))

{

freezer = Integer.parseInt(array[i][1].trim());

}

else

{

if (array[i][0].trim().toLowerCase().equals("fridge"))

{

fridge = Integer.parseInt(array[i][1].trim());

}

else

{

if (array[i][0].trim().toLowerCase().equals("pantry"))

{

pantry = Integer.parseInt(array[i][1].trim());

}

else

{

throw new IllegalArgumentException("Error: Cannot parse data" +

" on line " + i + ".");

}

}

}

} //would use switch if supported for strings in java 6

catch (NumberFormatException e)

{

throw new IllegalArgumentException("Error: Cannot parse data on"+

" line " + i + ".");

}

catch (ArrayIndexOutOfBoundsException e)

{

throw new IllegalArgumentException("Error: Invalid header data.");

}

}

if (fridge == -1 || freezer == -1 || pantry == -1)

{ //if unchanged from init then data is lacking

throw new IllegalArgumentException("Error: Invalid header data.");

}

sizes = new int[]{freezer, fridge, pantry};

return sizes;

}

/\*####################################################################

# METHOD addFood

# IMPORTS line (Array of strings)

# EXPORTS none

# PURPOSE Parses an array of strings as arguments to construct an

# object based on its type (as defined in array's first string)

\*###################################################################\*/

public void addFood(String[] line)

throws ParseException, IllegalArgumentException

{

SimpleDateFormat dt = new SimpleDateFormat("dd/MM/yyyy");

dt.setLenient(false);

if (line[0].trim().toLowerCase().equals("meat")) //sort meat

{

Meat meat = new Meat();

meat.setName(line[1].trim());

meat.setCut(line[2].trim());

meat.setWeight(Double.parseDouble(line[3].trim()));

meat.setStorageTemp(Double.parseDouble(line[4].trim()));

meat.setUseby(dt.parse(line[5].trim()));

meat.setPackaging(line[6].trim());

storFac.addFood(meat);

}

else

{

if (line[0].trim().toLowerCase().equals("grain")) //sort grain

{

Grain grain = new Grain();

grain.setName(line[1].trim());

grain.setType(line[2].trim());

grain.setVolume(Double.parseDouble(line[3].trim()));

grain.setStorageTemp(Double.parseDouble(line[4].trim()));

grain.setBestBefore(dt.parse(line[5].trim()));

grain.setPackaging(line[6].trim());

storFac.addFood(grain);

}

else

{

if (line[0].trim().toLowerCase().equals("fruit")) //type fruit

{

Fruit fruit = new Fruit();

fruit.setName(line[1].trim());

fruit.setType(line[2].trim());

fruit.setNumPiece(Integer.parseInt(line[3].trim()));

fruit.setStorageTemp(Double.parseDouble(line[4].trim()));

fruit.setUseby(dt.parse(line[5].trim()));

fruit.setPackaging(line[6].trim());

storFac.addFood(fruit);

}

else

{

if (line[0].trim().toLowerCase().equals("vegetable")) //type vege

{

Vegetable vegetable = new Vegetable();

vegetable.setName(line[1].trim());

vegetable.setWeight(Double.parseDouble(line[2].trim()));

vegetable.setStorageTemp(Double.parseDouble(line[3].trim()));

vegetable.setBestBefore(dt.parse(line[4].trim()));

vegetable.setPackaging(line[5].trim());

storFac.addFood(vegetable);

}

else

{

throw new IllegalArgumentException("Error: Invalid sort.");

}

}

}

} //would use switch if supported for strings in java 6

}

/\*####################################################################

# METHOD writeStorageCSV

# IMPORTS filename (String)

# EXPORTS none

# PURPOSE Writes a CSV file containing information on the foods

# contained within the storage locations inside of the

# storage facility, along with an appropriate file header.

\*###################################################################\*/

public void writeStorageCSV(String filename) throws IOException

{

Food[][] storage = storFac.getStorage();

int freezerLength = storage[storFac.FREEZER].length;

int fridgeLength = storage[storFac.FRIDGE].length;

int pantryLength = storage[storFac.PANTRY].length;

FileOutputStream fileStrm = null;

PrintWriter pw;

try

{

fileStrm = new FileOutputStream(filename);

pw = new PrintWriter(fileStrm);

pw.println("Freezer, " + freezerLength);

pw.println("Fridge, " + fridgeLength);

pw.println("Pantry, " + pantryLength);

for (int i = 0; i < storage.length; i++)

{

for (int j = 0; j < storage[i].length; j++)

{

if (storage[i][j] != null)

{

pw.println(storage[i][j].toCSV());

}

}

}

pw.close();

}

catch (IOException e)

{

if (fileStrm != null)

{

try

{

fileStrm.close();

}

catch (IOException ex2)

{

//these aren't the exceptions you're looking for, move along

}

}

throw new IOException("Error: Unable to write to file.");

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS StorageFacility

\* AUTHOR Jhi Morris

\* DATE CREATED 05/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Holds array of Foods and allows addition and removal of Food

\* objects to it. Stores Foods differently based on temperature.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CLASSFIELDS:

storage (Array of array of Food objects)

FREEZER := 0 (CONSTANT integer) //magic numbers for sorting storage easily

FRIDGE := 1(CONSTANT integer)

PANTRY := 2 (CONSTANT integer)

CONSTRUCTORS:

default:

IMPORTS none

CONSTRUCT StorageFacility USING alternate <- {0, 0, 0}

alternate:

IMPORTS: sizes (Array of integers)

setStorage <- FREEZER, sizes[FREEZER]

setStorage <- FRIDGE, sizes[FRIDGE]

setStorage <- PANTRY, sizes[PANTRY]

copy:

IMPORTS: storFac (StorageFacility object)

storage := storFac getStorage <- none

FOR int := 0 TO LENGTH OF storage INC i

setStorage <- i, LENGTH OF storage[i]

FOR j := 0 TO LENGTH OF storage[i] INC j

IF storage[i][j] NOT NULL

addFood <- storage[i][j] clone <- none

END IF

END FOR

END FOR

END CONSTRUCTORS

SUBMODULE: setStorage

IMPORTS: location (Integer), length (Integer)

EXPORTS: none

ALGORITHM:

IF length < 0

FAIL IllegalArgumentException "Error: Cannot create store with" +

" negative length"

END IF

IF storage[location] NOT NULL

FAIL IllegalStateException "Error: Cannot make multiple" +

" instances of location"

END IF

storage[locatation] := ARRAY OF Food WITH LENGTH length

END SUBMODULE

SUBMODULE: getFridge

IMPORTS: none

EXPORTS: out (Array of Food objects)

ALGORITHM:

fridge := storage[FRIDGE]

out := ARRAY OF Food WITH LENGTH LENGTH OF FRIDGE

FOR i := 0 TO LENGTH OF FRIDGE INC i

IF fridge[i] NOT NULL

out[i] := fridge[i] clone <- none

END IF

END FOR

END SUBMODULE

SUBMODULE: getPantry

IMPORTS: none

EXPORTS: out (Array of Food objects)

ALGORITHM:

pantry := storage[PANTRY]

out := ARRAY OF Food WITH LENGTH LENGTH OF PANTRY

FOR i := 0 TO LENGTH OF PANTRY INC i

IF pantry[i] NOT NULL

out[i] := pantry[i] clone <- none

END IF

END FOR

END SUBMODULE

SUBMODULE: getFreezer

IMPORTS: none

EXPORTS: out (Array of Food objects)

ALGORITHM:

freezer := storage[FREEZER]

out := ARRAY OF Food WITH LENGTH LENGTH OF freezer

FOR i := 0 TO LENGTH OF FREEZER INC i

IF freezer[i] NOT NULL

out[i] := freezer[i] clone <- none

END IF

END FOR

END SUBMODULE

SUBMODULE: getStorage

IMPORTS: none

EXPORT: out (Array of array of Food objects)

ALGORITHM:

out := ARRAY OF ARRAY OF FOOD WITH LENGTH LENGTH OF storage

FOR i := 0 TO LENGTH OF storage INC i

out[i] := ARRAY OF FOOD WITH LENGTH LENGTH OF storage[i]

FOR j := 0 TO LENGTH OF storage[i] INC j

IF storage[i][j] NOT NULL

out[i][j] := storage[i][j] clone <- none

END IF

END FOR

END FOR

END SUBMODULE

SUBMODULE: addFood

IMPORTS: food (Food object)

EXPORTS: none

ALGORITHM:

IF food IS NULL

FAIL IllegalStateException "Error: Null food object."

END IF

CASE checkTemp <- (food getStorageTemp <- none)

FREEZER:

TRY

storage[FREEZER][nextNullAt <- (getFreezer <- none)] := food clone

CATCH ArrayIndexOutOfBoundsException

FAIL IllegalStateException "Error: No space in freezer."

END TRYCATCH

FRIDGE:

TRY

storage[fridge][nextNullAt <- (getFridge <- none)] := food clone

CATCH ArrayIndexOutOfBoundsException

FAIL IllegalStateException "Error: No space in fridge."

END TRYCATCH

PANTRY:

TRY

storage[PANTRY][nextNullAt <- (getPantry <- none)] := food clone

CATCH ArrayIndexOutOfBoundsException

FAIL IllegalStateException "Error: No space in pantry."

END TRYCATCH

END CASE

END SUBMODULE

SUBMODULE: nextNullAt

IMPORTS: array (Array of objecs)

EXPORTS: i (Integer)

ALGORITHM:

i := 0

WHILE array[i] NOT NULL

INC i

END WHILE

END SUBMODULE

SUBMODULE: checkTemp

IMPORTS: storageTemp (Real)

EXPORTS: sort (Integer)

ALGORITHM:

IF TerminalUtility inRange <- storageTemp, -27.0, 25

IF storageTemp < -5

sort := FREEZER

ELSE

IF storageTemp > 25

sort := PANTRY

ELSE

sort := FRIDGE

END ELSE

END ELSE

ELSE

FAIL IllegalArgumentException "Error: Temperature out of range."

END ELSE

END SUBMODULE

SUBMODULE: removeFood

IMPORTS: location (Integer), element (Integer)

EXPORTS: none

ALGORITHM:

elementAt := 0

index := 0

found := false

IF location > 2 OR location < 0

FAIL IllegalArgumentException "Error: Invalid location."

END IF

TRY

DO

IF storage[location][index] NOT NULL

INC elementAt

IF elementAt EQUALS element

storage[location][index] := NULL //remove food

found := TRUE

END IF

END IF

INC index

WHILE NOT found

CATCH ArrayIndexOutOfBoundsException

FAIL IllegalArgumentException "Error: Element not found."

END TRYCATCH

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS StorageFacility

\* AUTHOR Jhi Morris

\* DATE CREATED 05/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Holds array of Foods and allows addition and removal of Food

\* objects to it. Stores Foods differently based on temperature.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public class StorageFacility

{

private Food[][] storage = new Food[3][];

public final int FREEZER = 0; //magic numbers used for sorting storage

public final int FRIDGE = 1;

public final int PANTRY = 2;

public StorageFacility()

{ //default constructor

this(new int[] {0, 0, 0});

}

public StorageFacility(int[] sizes)

{ //alt constructor

setStorage(FREEZER, sizes[0]);

setStorage(FRIDGE, sizes[1]);

setStorage(PANTRY, sizes[2]);

}

public StorageFacility(StorageFacility storFac)

{ //copy constructor

Food[][] storage = storFac.getStorage();

for (int i = 0; i < storage.length; i++)

{

setStorage(i, storage[i].length);

for (int j = 0; j < storage[i].length; j++)

{

if (storage[i][j] != null)

{

addFood(storage[i][j].clone());

}

}

}

}

/\*####################################################################

# METHOD setStorage

# IMPORTS location (Integer), length (Integer)

# PURPOSE Sets the size of a storage location based on imported length.

\*###################################################################\*/

private void setStorage(int location, int length)

throws IllegalStateException

{

if (length < 0)

{

throw new IllegalArgumentException("Error: Cannot create store with" +

" negative length.");

}

if (storage[location] != null)

{

throw new IllegalStateException("Error: Cannot make multiple" +

" instances of location.");

}

storage[location] = new Food[length];

}

/\*####################################################################

# METHOD getFridge

# IMPORTS none

# EXPORTS out (Array of Food objects)

# PURPOSE Creates a deep copy of the fridge's contents

# and exports it all as an array of Food objects.

\*###################################################################\*/

public Food[] getFridge()

{

Food[] fridge = storage[FRIDGE];

Food[] out = new Food[fridge.length];

for (int i = 0; i < fridge.length; i++)

{

if (fridge[i] != null)

{

out[i] = fridge[i].clone();

}

}

return out;

}

/\*####################################################################

# METHOD getPantry

# IMPORTS none

# EXPORTS out (Array of Food objects)

# PURPOSE Creates a deep copy of the pantry's contents

# and exports it all as an array of Food objects.

\*###################################################################\*/

public Food[] getPantry()

{

Food[] pantry = storage[PANTRY];

Food[] out = new Food[pantry.length];

for (int i = 0; i < pantry.length; i++)

{

if (pantry[i] != null)

{

out[i] = pantry[i].clone();

}

}

return out;

}

/\*####################################################################

# METHOD getFreezer

# IMPORTS none

# EXPORTS out (Array of Food objects)

# PURPOSE Creates a deep copy of the freezer's contents

# and exports it all as an array of Food objects.

\*###################################################################\*/

public Food[] getFreezer()

{

Food[] freezer = storage[FREEZER];

Food[] out = new Food[freezer.length];

for (int i = 0; i < freezer.length; i++)

{

if (freezer[i] != null)

{

out[i] = freezer[i].clone();

}

}

return out;

}

/\*####################################################################

# METHOD getStorage

# IMPORTS none

# EXPORTS out (Array of array of Food objects)

# PURPOSE Creates a deep copy of each storage location's contents

# and exports it all as a 2D array of Food objects.

\*###################################################################\*/

public Food[][] getStorage()

{

Food[][] out = new Food[this.storage.length][];

for (int i = 0; i < storage.length; i++)

{

out[i] = new Food[this.storage[i].length];

for (int j = 0; j < this.storage[i].length; j++)

{

if (this.storage[i][j] != null)

{

out[i][j] = this.storage[i][j].clone();

}

}

}

return out;

}

/\*####################################################################

# METHOD addFood

# IMPORTS food (Food object)

# EXPORTS none

# PURPOSE Adds a Food object to a storage location, checking its

# temperature to discern which location is appropriate

# for the item. Fails if the storage location is full.

\*###################################################################\*/

public void addFood(Food food) throws IllegalStateException

{

if (food == null)

{

throw new IllegalStateException("Error: Null food object.");

}

switch (checkTemp(food.getStorageTemp()))

{

case FREEZER:

try

{

storage[FREEZER][nextNullAt(getFreezer())] = food.clone();

}

catch (ArrayIndexOutOfBoundsException e)

{

throw new IllegalStateException("Error: No space in freezer.");

}

break;

case FRIDGE:

try

{

storage[FRIDGE][nextNullAt(getFridge())] = food.clone();

}

catch (ArrayIndexOutOfBoundsException e)

{

throw new IllegalStateException("Error: No space in fridge.");

}

break;

case PANTRY:

try

{

storage[PANTRY][nextNullAt(getFreezer())] = food.clone();

}

catch (ArrayIndexOutOfBoundsException e)

{

throw new IllegalStateException("Error: No space in pantry.");

}

}

}

/\*####################################################################

# METHOD nextNullAt

# IMPORTS array (Array of objects)

# EXPORTS i (Integer)

# PURPOSE Searches an array for the first null index and returns it.

# This is used to ensure empty locations made by Food deletion

# do not go unused. Technically works with non-Food arrays too.

# ASSERTION The index returned will be the first index at which the

# given array has a null index.

\*###################################################################\*/

private static int nextNullAt(Object[] array)

throws ArrayIndexOutOfBoundsException

{

int i = 0;

while (array[i] != null)

{

i++;

}

return i;

}

/\*####################################################################

# METHOD checkTemp

# IMPORTS storageTemp (Double)

# EXPORTS sort (Integer)

# PURPOSE Compares a temperature to defined bounds and returns storage

# location index for appropriate storage location.

# ASSERTION The returned int will be a valid location in the storage

# 2D array, and that location will suit the temperature.

# Fails if the temperature is outside of all ranges.

\*###################################################################\*/

private int checkTemp(double storageTemp) throws IllegalArgumentException

{

int sort;

if (TerminalUtility.inRange(storageTemp, -27.0, 25.0))

{

if (storageTemp < -5.0) //temp freezer

{

sort = FREEZER;

}

else

{

if (storageTemp > 25.0) //temp pantry

{

sort = PANTRY;

}

else //temp fridge

{

sort = FRIDGE;

}

}

}

else //temp out of range

{

throw new IllegalArgumentException("Error: Temperature out of range.");

}

return sort;

}

/\*####################################################################

# METHOD removeFood

# IMPORTS location (Integer), element (Integer)

# EXPORTS none

# PURPOSE Removes a food from given location at given element.

# NOTES Does not use array index (counting from 0) rather uses

# location element (counted from 1, not counting null indexes)

\*###################################################################\*/

public void removeFood(int location, int element)

throws IllegalArgumentException

{

int elementAt = 0;

int index = 0;

boolean found = false;

if (location > 2 || location < 0)

{

throw new IllegalArgumentException("Error: Invalid Location.");

}

try

{

do

{

if (storage[location][index] != null)

{

elementAt++;

if (elementAt == element)

{

storage[location][index] = null; //remove food

found = true;

}

}

index++;

}

while (!found);

}

catch (ArrayIndexOutOfBoundsException e)

{

throw new IllegalArgumentException("Error: Element not found.");

}

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS IFood

\* AUTHOR Mark Upston (Presumably)

\* DATE CREATED ??/??/??

\* DATE LAST EDITED 16/05/18 ?

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Calendar;

public interface IFood

{

public boolean calcExpiry(Calendar today);

//checks todays date and exports true if this food

//item has reached its expiry date

public int calcSpace(Food food);

//checks attributes of Food class object and

//exports an integer specifying the volume

//in litres of storage required.

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Food

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 25/05/18

\* PURPOSE Food object superclass. Validates that strings are not-null

\* and defines some abstract methods.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

IMPLEMENTS IFood INTERFACE

CLASSFIELDS:

name (String)

storageTemp (Real)

packaging (String)

CONSTRUCTORS:

default:

IMPORTS: none

setName <- ""

setStorageTemp <- 21.0

setPackaging <- ""

alternate:

IMPORTS: name (String), storageTemp (Real), packaging (String)

setName <- name

setStorageTemp <- storageTemp

setPackaging <- packaging

copy:

IMPORTS: foodIn (Food object)

setName <- foodIn getName <- none

setStorageTemp <- foodIn getStorageTemp <- none

setPackaging <- foodIn getPackaging <- none

END CONSTRUCTORS

SETTERS:

setStorageTemp

IMPORTS: storageTemp (Real)

IF storageTemp <= 25.0 AND storageTemp >= -27.0

CLASSFIELD storageTemp := storageTemp

ELSE

FAIL IllegalArgumentException "Error: Temperature out of range."

END ELSE

setName

IMPORTS: name (String)

IF name NOT NULL

CLASSFIELD storageTemp := storageTemp

ELSE

FAIL IllegalArgumentException "Error: Name is null."

END ELSE

setPackaging

IMPORTS: packaging (String)

IF packaging NOT NULL

CLASSFIELD packaging := packaging

ELSE

FAIL IllegalArgumentException "Error: Packaging is null."

END ELSE

END SETTERS

GETTERS:

getStorageTemp

EXPORTS: storageTemp (Real)

getName

EXPORTS: name (String)

getPackaging

EXPORTS: packaging (String)

END GETTERS

SUBMODULE: equals

IMPORTS: objIn (Object)

EXPORTS: equals (Boolean)

ALGORITHM:

equals := FALSE

IF objIn IS A Food

foodIn := objIn TO Food

IF (foodIn getName <- none EQUALS getName <- none AND

foodIn getStorageTemp <- none EQUALS getStorageTemp() AND

foodIn getPackaging <- none EQUALS getPackaging <- none)

equals := TRUE

END IF

END IF

END SUBMODULE

SUBMODULE: isExpired

IMPORTS: expDate (Date object)

EXPORTS: expired (Boolean)

ALGORITHM:

today := DATE NOW

IF expDate IS AFTER today

expired := FALSE

ELSE

expired := TRUE

END ELSE

END SUBMODULE

SUBMODULE: calcSpace

IMPORTS: food (Food object)

EXPORTS: space (Integer)

ALGORITHM:

IF food IS A Fruit

space := TO INTEGER (1.95 \* food getNumPiece <- none) ROUNDED UP

ELSE

IF food IS A Meat

space := TO INTEGER (0.86 \* food getWeight <- none) ROUNDED UP

ELSE

IF food IS A Grain

space := TO INTEGER food getVolume <- none ROUNDED UP

ELSE

IF food IS A Vegetable

space := TO INTEGER (1.025 \* food getWeight <- none) ROUNDED UP

ELSE

FAIL IllegalStateException "Error: Unknown Food type."

END ELSE

END ELSE

END ELSE

END ELSE

END SUBMODULE

ABSTRACT SUBMODULE: clone

IMPORTS: none

EXPORTS: Food object

ASSERTION: Exports a deep copy of the Food object

ABSTRACT SUBMODULE: toString

IMPORTS: none

EXPORTS: String

ASSERTION: Reports attributes of Food object as a human-readible string

ABSTRACT SUBMODULE: calcExpiry

IMPORTS: none

EXPORTS: Boolean

ASSERTION: Checks expiry using today's date as generated by system and

exports true if the food item has reached its expiry date.

ABSTRACT SUBMODULE: toCSV

IMPORTS: none

EXPORTS: String

ASSERTION: Reports attriutes of Food objects as CSV-compatible format string

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Food

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 25/05/18

\* PURPOSE Food object superclass. Validates that strings are not-null

\* and defines some abstract methods.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Date;

public abstract class Food implements IFood

{

private String name;

private double storageTemp;

private String packaging;

public Food()

{ //default constructor

setName("");

setStorageTemp(21.0);

setPackaging("");

}

public Food(String name, double storageTemp, String packaging)

{ //alt constructor

setName(name);

setStorageTemp(storageTemp);

setPackaging(packaging);

}

public Food(Food foodIn)

{ //copy constructor

setName(foodIn.getName());

setStorageTemp(foodIn.getStorageTemp());

setPackaging(foodIn.getPackaging());

}

//SETTERS

public void setStorageTemp(double storageTemp)

{

if (storageTemp <= 25.0 && storageTemp >= -27.0)

{

this.storageTemp = storageTemp;

}

else

{

throw new IllegalArgumentException("Error: Temperature out of range.");

}

}

public void setName(String name)

{

if (name != null)

{

this.name = new String(name);

}

else

{

throw new IllegalArgumentException("Error: Name is null.");

}

}

public void setPackaging(String packaging)

{

if (packaging != null)

{

this.packaging = new String(packaging);

}

else

{

throw new IllegalArgumentException("Error: Packaging is null.");

}

}

//GETTERS

public double getStorageTemp()

{

return storageTemp;

}

public String getName()

{

return new String(name);

}

public String getPackaging()

{

return new String(packaging);

}

/\*####################################################################

# METHOD equals

# IMPORTS objIn (Object)

# EXPORTS equals (Boolean)

# PURPOSE Checks if objIn is equal to Food object.

# ASSERTION Returns true if the given object is a Food object and if

# all aspects of both match.

\*###################################################################\*/

public boolean equals(Object objIn)

{

boolean equals = false; //guilty until proven innocent

if (objIn instanceof Food)

{

Food foodIn = (Food)objIn;

if (foodIn.getName().equals(this.getName()) &&

foodIn.getStorageTemp() == this.getStorageTemp() &&

foodIn.getPackaging().equals(this.getPackaging()))

{

equals = true;

}

}

return equals;

}

/\*####################################################################

# METHOD isExpired

# IMPORTS expDate (Date)

# EXPORTS expired (Boolean)

# PURPOSE Checks if expDate has passed, for date input validation.

# ASSERTION Returns true if the given date is in the past.

\*###################################################################\*/

public boolean isExpired(Date expDate)

{

Date today = new Date();

boolean expired;

if (expDate.after(today))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD calcSpace

# IMPORTS food (Food)

# EXPORTS space (Integer)

# PURPOSE Calculates the space taken up by a Food object based on its

# attributes. Each subclass of Food has a different calculation:

# Meat – weight \* 0.86 rounded up

# Grain – volume \* 1.0 rounded up

# Fruit – number of pieces \* 1.95 rounded up

# Vegetables – weight \* 1.025 rounded up

# ASSERTION Returns the calculated space as an integer value.

\*###################################################################\*/

public int calcSpace(Food food)

{

int space;

if (food instanceof Fruit)

{

space = (int)Math.ceil(((Fruit)food).getNumPiece() \* 1.95);

}

else

{

if (food instanceof Meat)

{

space = (int)Math.ceil(((Meat)food).getWeight() \* 0.86);

}

else

{

if (food instanceof Grain)

{

space = (int)Math.ceil(((Grain)food).getVolume());

}

else

{

if (food instanceof Vegetable)

{

space = (int)Math.ceil(((Vegetable)food).getWeight() \* 1.025);

}

else

{

throw new IllegalStateException("Error: Unknown Food.");

}

}

}

}

return space;

}

public abstract Food clone();

//returns a deep copy of the food object

public abstract String toString();

//reports attributes of Food class object

//as a human-readible string.

public abstract boolean calcExpiry();

//checks expiry using today's date as generated by itself and returns

//true if the food item has reached its expiry date.

public abstract String toCSV();

//reports attributes of Food class objects

//as CSV-compatible format string.

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Fruit

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 25/05/18

\* PURPOSE Creates Fruit object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

EXTENDS Food

CLASSFIELDS:

type (String)

numPiece (Integer)

useby (Date object)

CONSTRUCTORS:

default:

IMPORTS: none

CONSTRUCT Food USING default

setType <- ""

setNumPiece <- 1

setUseby <- 31/12/9999

alternate:

IMPORTS: name (String), type (String), numPiece (Integer),

storageTemp (Real), useby (Date object), packaging (String)

CONSTRUCT Food USING alternate <- name, storageTemp, packaging

setType <- type

setNumPiece <- numPiece

setUseby <- useby

copy:

IMPORTS: fruitIn (Fruit object)

CONSTRUCT Food using copy <- fruitIn

setType <- fruitIn getType <- none

setNumPiece <- fruitIn getNumPiece <- none

setUseby <- fruitIn getUseby <- none

END CONSTRUCTORS

SETTERS:

setType

IMPORTS: type (String)

IF type NOT NULL

CLASSFIELD type := type

ELSE

FAIL IllegalArgumentException "Error: Type is null."

END ELSE

setNumPiece

IMPORTS: numPiece (Integer)

IF TerminalUtility inRange <- numPiece, 1, 20 //range validation

CLASSFIELD numPiece := numPiece

ELSE

FAIL IllegalArgumentException "Error: Pieces out of range."

END ELSE

setUseby

IMPORTS: useby (Date)

IF NOT isExpired <- useby

CLASSFIELD useby := useby

ELSE

FAIL IllegalArgumentException "Error: Useby date in the past."

END ELSE

END SETTERS

GETTERS:

getType

EXPORTS: type (String)

getNumPiece

EXPORTS: numPiece (Integer)

getUseby

EXPORTS: useby (Date object)

END GETTERS

SUBMODULE: clone

IMPORTS: none

EXPORTS: cloen (Fruit object)

ALGORITHM:

clone := CONSTRUCT Fruit USING alternate <- (getName <- none,

getType <- none, getNumPiece <- none, getStorageTemp <- none,

getUseby <- none, getPackaging <- none)

END SUBMODULE

SUBMODULE: toString

IMPORTS: none

EXPORTS: string (String)

ALGORITHM:

string := (getNumPiece <- none + " pieces of " + getName <- none + ", " +

getType <- none + ". Packaged in " + getPackaging <- none +

", with use-by date of " + getUseby <- none +

" and a storage temperature of " + getStorageTemp <- none + "°C.")

END SUBMODULE

SUBMODULE: equals

IMPORTS: objIn (Object)

EXPORTS: equals (Boolean)

ALGORITHM:

equals := FALSE

IF SUPER equals <- objIn AND objIn IS A Fruit

fruitIn := objIn TO FRUIT

IF fruitIn getType <- none EQUALS getType <- none AND

fruitIn. getUseby <- none EQUALS getUseby <- none AND

fruitIn getNumPiece <- none EQUALS getNumPiece <- none

equals := TRUE

END IF

END IF

END SUBMODULE

SUBMODULE: calcExpiry

IMPORTS: none

EXPORTS: expired (Boolean)

ALGORITHM:

today := DATE TODAY

IF getUseby <- none IS AFTER today

expired := FALSE

ELSE

expired := TRUE

END ELSE

END SUBMODULE

SUBMODULE: toCSV

IMPORTS: none

EXPORTS: string (String)

ALGORITHM:

string =: "Fruit, " + getName <- none + ", " + getType <- none +

", " + getNumPiece <- none + ", " + getStorageTemp <- none + ", " +

getUseby <- none + ", " + getPackaging <- none

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Fruit

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 25/05/18

\* PURPOSE Creates Fruit object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Date;

import java.util.Calendar;

import java.text.SimpleDateFormat;

public class Fruit extends Food

{

private String type;

private int numPiece;

private Date useby;

public Fruit()

{ //default constructor

super();

this.setType("");

this.setNumPiece(1);

this.setUseby(new Date(9999, 12, 31)); //year 9999, december 31st

}

public Fruit(String name, String type, int numPiece, double storageTemp,

Date useby, String packaging)

{ //alt constructor

super(name, storageTemp, packaging);

this.setType(type);

this.setNumPiece(numPiece);

this.setUseby(useby);

}

public Fruit(Fruit fruitIn)

{ //copy constructor

super(fruitIn);

this.setType(fruitIn.getType());

this.setNumPiece(fruitIn.getNumPiece());

this.setUseby(fruitIn.getUseby());

}

//SETTERS

public void setType(String type)

{

if (type != null)

{

this.type = new String(type);

}

else

{

throw new IllegalArgumentException("Error: Type is null.");

}

}

public void setNumPiece(int numPiece)

{

if (TerminalUtility.inRange(numPiece, 1, 20))

{

this.numPiece = numPiece;

}

else

{

throw new IllegalArgumentException("Error: Pieces amount out of range.");

}

}

public void setUseby(Date useby)

{

if (!super.isExpired(useby))

{

this.useby = (Date)useby.clone();

}

else

{

throw new IllegalArgumentException("Error: Useby date in the past.");

}

}

//GETTERS

public String getType()

{

return new String(this.type);

}

public int getNumPiece()

{

return this.numPiece;

}

public Date getUseby()

{

return (Date)this.useby.clone();

}

/\*####################################################################

# METHOD clone

# IMPORTS none

# EXPORTS clone (Vegetable)

# PURPOSE Creates a deep-copy clone of the Fruit.

# ASSERTION Returns a clone of the Fruit.

\*###################################################################\*/

public Fruit clone()

{

Fruit clone = new Fruit(super.getName(), this.getType(),

this.getNumPiece(), super.getStorageTemp(), this.getUseby(),

super.getPackaging());

return clone;

}

/\*####################################################################

# METHOD toString

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a plain-text string describing the Fruit.

# ASSERTION Returns a single-line String.

\*###################################################################\*/

public String toString()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = (this.getNumPiece() + " pieces of " + super.getName() +

", " + this.getType() + ". Packaged in " + super.getPackaging() +

", with use-by date of " + df.format(this.getUseby()) +

" and a storage temperature of " + super.getStorageTemp() + "°C.");

return string;

}

/\*####################################################################

# METHOD equals

# IMPORTS objIn (Object)

# EXPORTS equals (Boolean)

# PURPOSE Checks if objIn is equal to Fruit object.

# ASSERTION Returns true if the given object is a Fruit object and if

# all aspects of both match.

\*###################################################################\*/

public boolean equals(Object objIn)

{

boolean equals = false;

if (super.equals(objIn) && objIn instanceof Fruit)

{

Fruit fruitIn = (Fruit)objIn;

if (fruitIn.getType().equals(this.getType()) &&

fruitIn.getUseby().equals(this.getUseby()) &&

fruitIn.getNumPiece() == this.getNumPiece())

{

equals = true;

}

}

return equals;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS none

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Fruit is past expiry, with today's date

# generated from the system time.

# ASSERTION Returns true if today is past the Fruit's use-by date.

\*###################################################################\*/

public boolean calcExpiry()

{

boolean expired;

Date today = new Date();

if (this.getUseby().after(today))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS today (Calendar)

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Fruit is past expiry, with today's date

# from the handed argument.

# ASSERTION Returns true if today is past the Fruit's best before date.

\*###################################################################\*/

public boolean calcExpiry(Calendar today)

{

Date dateToday = today.getTime();

boolean expired;

if (this.useby.after(dateToday))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD toCSV

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a csv-compatible string describing the Fruit's attributes

# ASSERTION Returns a comma seperated single-line String.

\*###################################################################\*/

public String toCSV()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = ("Fruit, " + super.getName() + ", " + this.getType() +

", " + this.getNumPiece() + ", " + super.getStorageTemp() + ", " +

df.format(this.getUseby()) + ", " + super.getPackaging());

return string;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Grain

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 25/05/18

\* PURPOSE Creates Grain object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

EXTENDS Food

CLASSFIELDS:

type (String)

volume (Real)

bestBefore (Date)

CONSTRUCTORS:

default:

IMPORTS: none

CONSTRUCT Food USING default

setType <- ""

setVolume <- 0.2

setBestBefore <- 31/12/9999

alternate:

IMPORTS: name (String), type (String), volume (Real), storageTemp (Real),

bestBefore (Date object), packaging (String)

CONSTRUCT Food USING alternate <- name, storageTemp, packaging

setType <- type

setVolume <- volume

setBestBefore <- bestBefore

copy:

IMPORTS: grainIn (Grain object)

CONSTRUCT Food using copy <- grainIn

setType <- grainIn getType <- none

setVolume <- grainIn getVolume <- none

setBestBefore <- grainIn getBestBefore <- none

END CONSTRUCTORS

SETTERS:

setType

IMPORTS: type (String)

IF type NOT NULL

CLASSFIELD type := type

ELSE

FAIL IllegalArgumentException "Error: Type is null."

END ELSE

setVolume

IMPORTS: volume (Real)

IF volume <= 0.2 && volume <= 5.0

CLASSFIELD volume := volume

ELSE

FAIL IllegalArgumentException "Error: Volume is out of range.""

END ELSE

setBestBefore

IMPORTS: bestBefore (Date object)

IF NOT isExpired <- bestBefore

CLASSFIELD bestBefore := bestBefore

ELSE

FAIL IllegalArgumentException "Error: Best-before date in past."

END ELSE

END SETTERS

GETTERS:

getType

EXPORTS: type

getVolume

EXPORTS: volume

getBestBefore

EXPORTS: bestBefore

END GETTERS

SUBMODULE: clone

IMPORTS: none

EXPORTS: clone (Grain)

ALGORITHM:

clone := CONSTRUCT Grain USING alternate <- (getName <- none,

getType <- none, getVolume <- none, getStorageTemp <- none,

getBestBefore <- none, getPackaging <- none)

END SUBMODULE

SUBMODULE: toString

IMPORTS: none

EXPORTS: string (String)

ALGORITHM:

string := (getVolume <- none + "L of " + getName <- none + ", " +

getType <- none + ". Packaged in " + getPackaging <- none +

", with a best-before date of " + getBestBefore <- none) +

" and a storage temperature of " + getStorageTemp <- none + "°C.")

END SUBMODULE

SUBMODULE: equals

IMPORTS: objIn (Object)

EXPORTS: equals (Boolean)

ALGORITHM:

equals := FALSE

IF (SUPER equals <- objIn) && objIn IS A Grain

grainIn := objIn TO Grain

IF (grainIn getType <- none EQUALS getType <- none) AND

grainIn getBestBefore <- none EQUALS getBestBefore <- none AND

grainIn getVolume <- none EQUALS getVolume <- none)

equals := TRUE

END IF

END IF

END SUBMODULE

SUBMODULE: calcExpiry

IMPORTS: none

EXPORTS: expired (Boolean)

ALGORITHM:

today := DATE TODAY

IF getBestBefore <- none IS AFTER today

expired := false

ELSE

expired := true

END ELSE

END SUBMODULE

SUBMODULE: toCSV

IMPORTS: none

EXPORTS: string (String)

ALGORITHM:

string := ("Grain, " + getName <- none + ", " + getType <- none + ", " +

getVolume <- none + ", " + getStorageTemp <- none + ", " +

getBestBefore <- none + ", " + getPackaging <- none)

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Grain

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 25/05/18

\* PURPOSE Creates Grain object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Date;

import java.util.Calendar;

import java.text.SimpleDateFormat;

public class Grain extends Food

{

private String type;

private double volume;

private Date bestBefore;

public Grain()

{ //default constructor

super();

this.setType("");

this.setVolume(0.2);

this.setBestBefore(new Date(9999, 12, 31)); //year 9999, december 31st

}

public Grain(String name, String type, double volume, double storageTemp,

Date bestBefore, String packaging)

{ //alt constructor

super(name, storageTemp, packaging);

this.setType(type);

this.setVolume(volume);

this.setBestBefore(bestBefore);

}

public Grain(Grain grainIn)

{ //copy constructor

super(grainIn);

this.setType(grainIn.getType());

this.setVolume(grainIn.getVolume());

this.setBestBefore(grainIn.getBestBefore());

}

//SETTERS

public void setType(String type)

{

if (type != null)

{

this.type = new String(type);

}

else

{

throw new IllegalArgumentException("Error: Type is null.");

}

}

public void setVolume(double volume)

{

if (volume >= 0.2 && volume <= 5.0)

{

this.volume = volume;

}

else

{

throw new IllegalArgumentException("Error: Volume is out of range.");

}

}

public void setBestBefore(Date bestBefore)

{

if (!super.isExpired(bestBefore))

{

this.bestBefore = (Date)bestBefore.clone();

}

else

{

throw new IllegalArgumentException("Error: Best-before date in past.");

}

}

//GETTERS

public String getType()

{

return new String(this.type);

}

public double getVolume()

{

return this.volume;

}

public Date getBestBefore()

{

return (Date)this.bestBefore.clone();

}

/\*####################################################################

# METHOD clone

# IMPORTS none

# EXPORTS clone (Grain)

# PURPOSE Creates a deep-copy clone of the Grain.

# ASSERTION Returns a clone of the Grain.

\*###################################################################\*/

public Grain clone()

{

Grain clone = new Grain(super.getName(), this.getType(), this.getVolume(),

super.getStorageTemp(), this.getBestBefore(), super.getPackaging());

return clone;

}

/\*####################################################################

# METHOD toString

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a plain-text string describing the Grain.

# ASSERTION Returns a single-line String.

\*###################################################################\*/

public String toString()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = (this.getVolume() + "L of " + super.getName() + ", " +

this.getType() + ". Packaged in " + super.getPackaging() +

", with a best-before date of " + df.format(this.getBestBefore()) +

" and a storage temperature of " + super.getStorageTemp() + "°C.");

return string;

}

/\*####################################################################

# METHOD equals

# IMPORTS objIn (Object)

# EXPORTS equals (Boolean)

# PURPOSE Checks if objIn is equal to Grain object.

# ASSERTION Returns true if the given object is a Grain object and if

# all aspects of both match.

\*###################################################################\*/

public boolean equals(Object objIn)

{

boolean equals = false;

if (super.equals(objIn) && objIn instanceof Grain)

{

Grain grainIn = (Grain)objIn;

if (grainIn.getType().equals(this.getType()) &&

grainIn.getBestBefore().equals(this.getBestBefore()) &&

grainIn.getVolume() == this.getVolume())

{

equals = true;

}

}

return equals;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS none

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Grain's is past expiry, with today's date

# generated from the system time.

# ASSERTION Returns true if today is past the Grain's best before date.

\*###################################################################\*/

public boolean calcExpiry()

{

boolean expired;

Date today = new Date();

if (this.getBestBefore().after(today))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS today (Calendar)

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Grain is past expiry, with today's date

# from the handed argument.

# ASSERTION Returns true if today is past the Grain's best before date.

\*###################################################################\*/

public boolean calcExpiry(Calendar today)

{

Date dateToday = today.getTime();

boolean expired;

if (this.bestBefore.after(dateToday))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD toString

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a csv-compatible string describing the Grain's attributes

# ASSERTION Returns a comma seperated single-line String.

\*###################################################################\*/

public String toCSV()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = ("Grain, " + super.getName() + ", " + this.getType() +

", " + this.getVolume() + ", " + super.getStorageTemp() + ", " +

df.format(this.getBestBefore()) + ", " + super.getPackaging());

return string;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Meat

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Creates Meat object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

EXTENDS Food

CLASSFIELDS:

cut (String)

weight (Real)

useby (Date object)

CONSTRUCTORS:

default:

IMPORTS: none

CONSTRUCT Food USING default

setCut <- ""

setWeight <- 0.2

setUseby <- 31/12/9999

alternate:

IMPORTS: name (String), cut (String), weight (Real), storageTemp (Real),

useby (Date object), packaging (String)

CONSTRUCT Food USING alternate <- name, storageTemp, packaging

setCut <- cut

setWeight <- weight

setUseby <- useby

copy:

IMPORTS: meatIn (Meat object)

CONSTRUCT Food using altermate <- meatIn

setCut <- meatIn getCut <- none

setWeight <- meatIn getWeight <- none

setUseby <- meatIn getUseby <- none

END CONSTRUCTORS

SETTERS:

setWeight

IMPORTS: weight (Real)

IF weight >= 0.2 AND weight <= 5.0

CLASSFIELD weight := weight

ELSE

FAIL IllegalArgumentException "Error: Weight out of bounds."

END ELSE

setUseby

IMPORTS: useby (Date object)

IF NOT isExpired <- useby

CLASSFIELD useby := useby

ELSE

FAIL IllegalArgumentException "Error: Useby date in the past."

END ELSE

setCut

IMPORTS: cut (String)

IF cut NOT NULL

CLASSFIELD cut := cut

ELSE

FAIL IllegalArgumentException "Error: Cut is null."

END ELSE

END SETTERS

GETTERS:

getCut

EXPORTS: cut (String)

getWeight

EXPORTS: weight (Real)

getUseby

EXPORTS: useby (Date object)

END GETTERS

SUBMODULE clone

IMPORTS: none

EXPORTS: clone (Meat object)

ALGORITHM:

clone := CONSTRUCT Meat USING alternate <- (getName <- none, getCut <- none,

getWeight <- none, getStorageTemp <- none, getUseby <- none,

getPackaging <- none)

END SUBMODULE

SUBMODULE toString

IMPORTS: none

EXPORTS: string (String)

ALGORITHM:

string = (getWeight <- none + "g of " + getName <- none + " " +

getCut <- none + ". Packaged in " + getPackaging <- none +

", with use-by date of " + getUseby <- none +

" and a storage temperature of " + getStorageTemp <- none + "°C.")

END SUBMODULE

SUBMODULE equals

IMPORTS: objIn (Object)

EXPORTS: equals (Boolean)

ALGORITHM:

equals := FALSE

IF SUPER equals <- objIn AND objIN IS A Meat

meatIn := objIn TO Meat

IF (meatIn.getCut <- none EQUALS (this.getCut <- none) AND

meatIn.getUseby <- none EQUALS (this.getUseby <- none) AND

meatIn.getWeight <- none EQUALS this.getWeight <- none)

equals := TRUE

END IF

END SUBMODULE

SUBMODULE calcExpiry

IMPORTS: none

EXPORTS: expired (Boolean)

ALGORITHM:

today := DATE TODAY

IF useby IS AFTER today

expired := FALSE

ELSE

expired := TRUE

END ELSE

END SUBMODULE

SUBMODULE toCSV

IMPORTS: none

EXPORTS: sting (String)

ALGORITHM:

string := ("Meat, " + getName <- none + ", " + getCut <- none + ", " +

getWeight <- none + ", " + getStorageTemp <- none + ", " +

getUseby <- none + ", " + getPackaging <- none)

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Meat

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Creates Meat object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Date;

import java.util.Calendar;

import java.text.SimpleDateFormat;

public class Meat extends Food

{

private String cut;

private double weight;

private Date useby;

public Meat()

{ //default constructor

super();

this.setCut("");

this.setWeight(0.2);

this.setUseby(new Date(9999, 12, 31)); //year 9999, december 31st

}

public Meat(String name, String cut, double weight, double storageTemp,

Date useby, String packaging)

{ //alt constructor

super(name, storageTemp, packaging);

this.setCut(cut);

this.setWeight(weight);

this.setUseby(useby);

}

public Meat(Meat meatIn)

{ //copy constructor

super(meatIn);

this.setCut(meatIn.getCut());

this.setWeight(meatIn.getWeight());

this.setUseby(meatIn.getUseby());

}

//SETTERS

public void setWeight(double weight)

{

if (weight >= 0.2 && weight <= 5.0)

{

this.weight = weight;

}

else

{

throw new IllegalArgumentException("Error: Weight out of bounds.");

}

}

public void setUseby(Date useby)

{

if (!super.isExpired(useby))

{

this.useby = (Date)useby.clone();

}

else

{

throw new IllegalArgumentException("Error: Useby date in the past.");

}

}

public void setCut(String cut)

{

if (cut != null)

{

this.cut = new String(cut);

}

else

{

throw new IllegalArgumentException("Error: Cut is null.");

}

}

//GETTERS

public String getCut()

{

return new String(cut);

}

public double getWeight()

{

return weight;

}

public Date getUseby()

{

return (Date)useby.clone();

}

/\*####################################################################

# METHOD clone

# IMPORTS none

# EXPORTS clone (Meat)

# PURPOSE Creates a deep-copy clone of the Meat.

# ASSERTION Returns a clone of the Meat.

\*###################################################################\*/

public Meat clone()

{

Meat clone = new Meat(super.getName(), this.getCut(), this.getWeight(),

super.getStorageTemp(), this.getUseby(),

super.getPackaging());

return clone;

}

/\*####################################################################

# METHOD toString

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a plain-text string describing the Meat.

# ASSERTION Returns a single-line String.

\*###################################################################\*/

public String toString()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = (this.getWeight() + "g of " + super.getName() + " " +

this.getCut() + ". Packaged in " + super.getPackaging() +

", with use-by date of " + df.format(this.getUseby()) +

" and a storage temperature of " + super.getStorageTemp() + "°C.");

return string;

}

/\*####################################################################

# METHOD equals

# IMPORTS objIn (Object)

# EXPORTS equals (Boolean)

# PURPOSE Checks if objIn is equal to Meat object.

# ASSERTION Returns true if the given object is a Meat object and if

# all aspects of both match.

\*###################################################################\*/

public boolean equals(Object objIn)

{

boolean equals = false;

if (super.equals(objIn) && objIn instanceof Meat)

{

Meat meatIn = (Meat)objIn;

if (meatIn.getCut().equals(this.getCut()) &&

meatIn.getUseby().equals(this.getUseby()) &&

meatIn.getWeight() == this.getWeight())

{

equals = true;

}

}

return equals;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS none

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Meat is past expiry, with today's date

# generated from the system time.

# ASSERTION Returns true if today is past the Meat's use-by date.

\*###################################################################\*/

public boolean calcExpiry()

{

Date today = new Date();

boolean expired;

if (useby.after(today))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS today (Calendar)

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Meat is past expiry, with today's date

# from the handed argument.

# ASSERTION Returns true if today is past the Meat's best before date.

\*###################################################################\*/

public boolean calcExpiry(Calendar today)

{ //use this if you can't trust system time

Date dateToday = today.getTime();

boolean expired;

if (useby.after(dateToday))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD toCSV

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a csv-compatible string describing the Meat's attributes

# ASSERTION Returns a comma seperated single-line String.

\*###################################################################\*/

public String toCSV()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = ("Meat, " + super.getName() + ", " + this.getCut() +

", " + this.getWeight() + ", " + super.getStorageTemp() + ", " +

df.format(useby) + ", " + super.getPackaging());

return string;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Vegetable

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Creates Vegetabe object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

CLASSFIELDS:

weight (Real)

bestBefore (Date object)

CONSTRUCTORS:

default:

IMPORTS: none

CONSTRUCT Food USING default

setWeight <- 0.2

setBestBefore <- 31/12/9999

alternate:

IMPORTS: name (String), weight (Real), storageTemp (Real),

bestBefore (Date object), packaging (String)

CONSTRUCT Food USING alternate <- name, storageTemp, packaging

setWeight <- weight

setBestBefore <- bestBefore

copy:

IMPORTS: vegetableIn (Vegetable object)

CONSTRUCT Food USING copy <- vegetableIn

setWeight <- vegetableIn getWeight <- none

setBestBefore <- vegetableIn getBestBefore <-

END CONSTRUCTORS

SETTERS:

setWeight

IMPORTS: weight (Real)

IF weight >= 0.2 AND weight <= 5.0

CLASSFIELD weight := weight

ELSE

FAIL IllegalArgumentException "Error: Weight out of bounds."

END ELSE

setBestBefore

IMPORTS: bestBefore (Date object)

IF NOT isExpired <- bestBefore

CLASSFIELD bestBefore := bestBefore

ELSE

FAIL IllegalArgumentException "Error: Best-before date in past."

END ELSE

END SETTERS

GETTERS:

getWeight

EXPORTS: weight (Real)

getBestBefore

EXPORTS: bestBefore (Date object)

END GETTERS

SUBMODULE: clone

IMPORTS: none

EXPORTS: clone (Vegetable object)

ALGORITHM:

clone := CONSTRUCT Vegetable USING alternate <- (getName <- none,

getWeight <- none, getStorageTemp <- none, getBestBefore <- none,

getPackaging <- none)

END SUBMODULE

SUBMODULE: toString

IMPORTS: none

EXPORTS: string (String)

ALGORITHM:

string = (getVolume <- none + "L of " + getName <- none +

". Packaged in " + getPackaging <- none + ", with best-before date of " +

getBestBefore <- none + " and a storage temperature of " +

getStorageTemp <- none + "°C.")

END SUBMODULE

SUBMODULE: equals

IMPORTS: objIn (Object)

EXPORTS: equals (Boolean)

ALGORITHM:

equals := FALSE

IF SUPER equals <- objIn AND objIn IS A Vegetable

vegeIn := objIn AS A Vegetable

IF vegeIn getWeight <- none EQUALS getWeight AND

vegeIn getBestBefore <- none EQUALS getBestBefore <- none

equals := TRUE

END IF

END IF

END SUBMODULE

SUBMODULE: calcExpiry

IMPORTS: none

EXPORTS: expired (Boolean)

ALGORITHM:

today := DATE TODAY

IF bestBefore <- none IS AFTER today

expired := FALSE

ELSE

expired := TRUE

END ELSE

END SUBMODULE

SUBMODULE: toCSV

IMPORTS: none

EXPORTS: string (String)

ALGORITHM:

string := ("Vegetable, " + getName <- none + ", " + getWeight <- none +

getBestBefore <- none + ", " + getPackaging <- none)

END SUBMODULE

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* CLASS Vegetable

\* AUTHOR Jhi Morris

\* DATE CREATED 02/05/18

\* DATE LAST EDITED 29/05/18

\* PURPOSE Creates Vegetable object, with validation.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

import java.util.Date;

import java.util.Calendar;

import java.text.SimpleDateFormat;

public class Vegetable extends Food

{

private double weight;

private Date bestBefore;

public Vegetable()

{ //default constructor

super();

this.setWeight(0.2);

this.setBestBefore(new Date(9999, 12, 31)); //year 9999, december 31st

}

public Vegetable(String name, double weight, double storageTemp,

Date bestBefore, String packaging)

{ //alt constructor

super(name, storageTemp, packaging);

this.setWeight(weight);

this.setBestBefore(bestBefore);

}

public Vegetable(Vegetable vegetableIn)

{ //copy constructor

super(vegetableIn);

this.setWeight(vegetableIn.getWeight());

this.setBestBefore(vegetableIn.getBestBefore());

}

//SETTERS

public void setWeight(double weight)

{

if (weight >= 0.2 && weight <= 5.0)

{

this.weight = weight;

}

else

{

throw new IllegalArgumentException("Error: Weight out of bounds.");

}

}

public void setBestBefore(Date bestBefore)

{

if (!isExpired(bestBefore))

{

this.bestBefore = (Date)bestBefore.clone();

}

else

{

throw new IllegalArgumentException("Error: Best-before date in past.");

}

}

//GETTERS

public double getWeight()

{

return this.weight;

}

public Date getBestBefore()

{

return (Date)this.bestBefore.clone();

}

/\*####################################################################

# METHOD clone

# IMPORTS none

# EXPORTS clone (Vegetable)

# PURPOSE Creates a deep-copy clone of the Vegetable.

# ASSERTION Returns a clone of the Vegetable.

\*###################################################################\*/

public Vegetable clone()

{

Vegetable clone = new Vegetable(super.getName(), this.getWeight(),

super.getStorageTemp(), this.getBestBefore(), super.getPackaging());

return clone;

}

/\*####################################################################

# METHOD toString

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a plain-text string describing the Vegetable.

# ASSERTION Returns a single-line String.

\*###################################################################\*/

public String toString()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = (weight + "g of " + super.getName() + ". Packaged in " +

super.getPackaging() + ", with best before date of " +

df.format(bestBefore) + " and a storage temperature of " +

super.getStorageTemp() + "°C.");

return string;

}

/\*####################################################################

# METHOD equals

# IMPORTS objIn (Object)

# EXPORTS equals (Boolean)

# PURPOSE Checks if objIn is equal to Vegetable object.

# ASSERTION Returns true if the given object is a Vegetable object and if

# all aspects of both match.

\*###################################################################\*/

public boolean equals(Object objIn)

{

boolean equals = false;

if (super.equals(objIn) && objIn instanceof Vegetable)

{

Vegetable vegeIn = (Vegetable)objIn;

if (vegeIn.getWeight() == this.getWeight() &&

vegeIn.getBestBefore().equals(this.getBestBefore()))

{

equals = true;

}

}

return equals;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS none

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Vegetable is past expiry, with today's date

# generated from the system time.

# ASSERTION Returns true if today is past the Vegetable's best before date.

\*###################################################################\*/

public boolean calcExpiry()

{

boolean expired;

Date today = new Date();

if (this.bestBefore.after(today))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD calcExpiry

# IMPORTS today (Calendar)

# EXPORTS expired (Boolean)

# PURPOSE Checks if the Vegetable is past expiry, with today's date

# from the handed argument.

# ASSERTION Returns true if today is past the Vegetable's best before date.

\*###################################################################\*/

public boolean calcExpiry(Calendar today)

{

Date dateToday = today.getTime();

boolean expired;

if (this.bestBefore.after(dateToday))

{

expired = false;

}

else

{

expired = true;

}

return expired;

}

/\*####################################################################

# METHOD toCSV

# IMPORTS none

# EXPORTS string (String)

# PURPOSE Creates a csv-compatible string describing the Vegetable's

# attributes.

# ASSERTION Returns a comma seperated single-line String.

\*###################################################################\*/

public String toCSV()

{

SimpleDateFormat df = new SimpleDateFormat("dd/MM/yyyy");

String string = ("Vegetable, " + super.getName() + ", " +

this.getWeight() + ", " + super.getStorageTemp() + ", " +

df.format(this.getBestBefore()) + ", " + super.getPackaging());

return string;

}

}