Soda Machine Laboratory

Introduction

This laboratory is intended to give you experience working with multiple queue structures in a familiar real-world setting. The given application models a soda machine in terms of its appearance and behavior. Some features have been included in the program; others have not been included. Your main task is to create a queue implementation for use in the individual beverage dispensers at the heart of the machine. In addition, you will augment the application in various ways to make it a more accurate representation of a real soda machine. A queue implementation built from a Java `Vector` is provided to show you how the application works.

What you must do

There are two main tasks for you to perform in this laboratory. You are required to follow all style conventions and produce well-written, easily readable code. Your grade for this exercise will be affected by the quality of your code. The tasks for this laboratory are:

- Implement the Queue ADT using a linked list.
- Make several modifications to make the soda machine more realistic.

This document contains instructions that will help you perform these tasks.

Provided code

There are 17 source code files provided in the laboratory archive. HTML documentation for each of the files is provided in the `docs` folder for the laboratory archive. You are not required to study the given files or their corresponding HTML documentation, but you may find them interesting or helpful. The provided files are an example of well-written Java code.

Several of the files contain classes that are used to create the user interface for the application. These files are listed below:

- `Advertisement.java`: Advertisement for the machine casing.
- `DispenserView.java`: View for a beverage dispenser.
- `InteriorView.java`: View for a row of beverage dispensers.
- `LCDDigit.java`: View for a single digit in an LCD display.
- `LCDDisplay.java`: View for a numeric LCD display.
- `SodaMachineView.java`: View for an entire soda machine.
Two of the files contain classes that are used to model interactions with the soda machine:

- **SodaMachine.java**
  - Represents an entire soda machine.
- **Dispenser.java**
  - Represents a dispenser in a soda machine.
- **Beverage.java**
  - Represents a single beverage to be dispensed.
- **Currency.java**
  - Represents a piece of currency: coin or bill.

Three of the files contain classes that represent possible error conditions. Their names speak for themselves:

- **BeverageUnavailableException.java**
- **NotEnoughCurrencyException.java**
- **QueueEmptyException.java**

Three of the files provide the specification, implementation, and adapting of the Queue ADT:

- **QueueADT.java**
  - Interface specifying the Queue ADT operations.
- **BeverageQueue.java**
  - QueueADT adapter for Beverage objects.
- **VectorQueue.java**
  - QueueADT implementation using a Vector.

The last file contains the application class itself. This class contains the `public static void main(String[] args)` method required for a Java application, and provides the overall user interface that drives the program. This last file is listed below:

- **SodaMachineApplication.java**
  - GUI application and main class.

This laboratory exercise makes use of the JPT. HTML documentation for the tools are available by following the link found in the docs directory of the laboratory archive. The given CodeWarrior™ project file is configured to make use of the JPT in its given location. An attempt to move the JPT archive or modify the project settings may render the tools unusable.

**A linked list queue implementation**

The first, and most important task for this laboratory is for you to create a queue implementation built from a linked list. Since a queue is only required to provide five simple operations, such a linked list class is much easier to build than one that provides a great deal of very general operations. Since a queue is so much easier to build than a general linked list, you are expected to create the class from scratch.
The first step is to define a class representing the nodes that will be linked together to form the list. This class has a very simple description, and examples of its design have been covered in lecture and can be found in the textbook. For the purposes of this laboratory, the node class must follow this specification:

- It must be named `Node`, and placed in a file named `Node.java`.
- Its data can be either `public` or `protected`.
- It must contain a reference to an `Object` it is storing.
- It must contain references to the `Node`s that come before and after it in the list.

Once you have completed the `Node` class, you can move on to create the overall queue implementation.

There are five required methods in the `QueueADT` interface. Each of these must be provided by your class. By considering each method one at a time, it is not too difficult to meet the requirements for this class. Before any of the required methods can be written, however, the basic structure of the class must be defined.

First, make a new file containing an empty class named `LinkedListQueue`, and save that file as `LinkedListQueue.java` in an appropriate directory. Since this class will use the same principles as a general linked list structure, it should have three data members: a sentinel `Node` named `head`, a sentinel `Node` named `tail`, and an `int` named `size`. These data members should be `protected`, so that programmers writing code outside of this class or its subclasses do not have direct access to change these data. The references `head` and `tail` should both be assigned to refer to `null`, so that the `Node` objects themselves can be created in the constructor. The `size` should be initially set to 0.

As described in lecture, sentinel nodes are placeholder `Node` objects that will never be used to store any data. These nodes should always be present in the linked structure, whether or not the structure is empty of data. All data stored in the linked structure should be stored in `Node` objects linked in between the head and tail sentinel nodes. For more information about sentinel nodes in linked structures, consult your lecture notes and your textbook.

The size is stored in an integer variable so that the nodes do not have to be counted every time someone wants to know the number of data items in the structure. Counting the nodes is an inefficient technique for determining the size of the structure. It is much faster to simply return the value of the integer variable when the current size of the structure is desired. Since a queue only has two methods that change the contents of the structure, it is very simple to keep this integer variable updated to maintain the correct size.

Once the sentinel node references `head` and `tail` have been declared, a constructor must be written that creates the `Node` objects and links them up correctly. This constructor must be `public` and need not take any parameters. Inside the constructor, you should create a new `Node` object for the `head`, and a new `Node` object for the `tail`. After the objects have been created, the `head` sentinel node should be linked so that the `tail` sentinel node is the next node after the
head, and the tail sentinel node should be linked so that the head sentinel node is the previous node before the tail. The result should be an empty linked structure that represents the following situation:

```
    head           tail
    +-----------+  +-----------+
    |          |  |            |
    |          |  |            |
    |          |  |            |
    |          |  |            |
    +-----------+  +-----------+
```

Once the constructor has been written, the only remaining work is to define the five required methods of the QueueADT interface. In order to ensure that all of the method signatures are spelled exactly as in the interface, it is best to copy the body of the QueueADT interface and paste it into your LinkedListQueue class after the constructor. If you copy the signatures from the interface, you will only need to fill in the code for each of the methods, rather than writing both the methods and their documentation entirely by yourself.

The first two methods to write are the size and isEmpty methods. These methods are extremely simple, and do not require any modifications to the linked structure. For the size method, you need only to return the value of the integer variable size. For the isEmpty method, you can either return whether or not the value of size is equal to 0, or return whether or not the next node immediately after head is tail. Both techniques are correct, and are relatively equal in difficulty.

The third method to write is the front method. Before writing any code, it is important to know exactly what this method is supposed to do. The documentation for the method describes that it returns the front object on the queue, or throws a QueueEmptyException if there is no such front object to return. For the purposes of this laboratory exercise, we will consider the object stored in the node immediately following head to be the front object on the queue. If the queue is empty, however, the node immediately following head will be tail, which does not contain an object. Thus there are two cases to consider:

- If the queue is empty, throw a new QueueEmptyException.
- Otherwise, return the object stored in the node immediately after head.

Write the front method so that it exhibits the appropriate behavior for the two possible cases.

The fourth method to write is the enqueue method. Since our implementation of a queue can never be full, there is only one case for this method; unlike the front method, we do not need to test for an error condition. In order to provide the FIFO behavior of a queue, we need to enqueue data on the end of the linear structure opposite to the front of the queue. Since the front method has already been written to consider the head end of the structure to be the front of the queue, we have to enqueue data on the tail end of the structure.
In order to enqueue data, a new Node object must be constructed to contain the data and must be linked up in the appropriate location in the structure. The appropriate location in the structure is immediately before the tail node. In your enqueue method, you must do the following:

- Create a new node that contains the given data Object.
- Link the new node so that the tail node is after it.
- Link the new node so that the node previously before the tail is before it.
- Link the node previously before the tail so that the new node is after it.
- Link the tail node so that new node is before it.
- Increment the variable that keeps track of the size.

The nodes should eventually be linked up in the following fashion:

```
old last     new node     tail
```

We do not know whether the node that was previously before tail is head, or some other node that contains some data. The beauty of using sentinel nodes in a linked structure is that it doesn’t matter. We do not have to consider the different cases, because the code is exactly the same for both situations.

The fifth method to write is the dequeue method. This method is similar to the front method in that there is a possible error condition. This method is more difficult to write than the front method, however, because it requires changing the linked structure. The two cases to consider are:

- If the queue is empty, throw a new QueueEmptyException.
- Otherwise, dequeue the object stored in the node immediately after head.

In order to provide the FIFO behavior of a queue, we need to dequeue data from the front of the queue. Since the front method has already been written to consider the head end of the structure to be the front of the queue, we have to dequeue data from the head end of the structure.
In order to dequeue data, a Node object must be removed from after the head and the nodes before and after the removed node must be linked together. In your dequeue method, you must do the following:

- Create a reference to the object contained by the node to be removed.
- Link the head to the node after the node to be removed.
- Link the node after the node to be removed to the head.
- Decrement the variable that keeps track of the size.
- Return the object contained by the removed node.

The nodes should eventually be linked up in the following fashion:

Since no nodes in the structure refer to the removed node, Java will destroy the removed node and reclaim the memory it used. We don’t know whether the node now after head is tail, or some other node that contains data. As with the enqueue method, the beauty of using sentinel nodes is that the code is the same for both situations.

Once you have finished the methods required for the QueueADT interface, the class is finished. Consult the “Testing and Troubleshooting” section for instructions as to how you test your queue implementation and overcome errors in your code. You should test your queue implementation before modifying any of the classes as part of the second task.

Modifying the soda machine simulation

This soda machine simulation is not perfect, and in fact it is far from perfect. Your second major task is to modify the soda machine in interesting ways that each make the simulation either:

- A more correct simulation, in terms of use and behavior.
- A more appropriate simulation, in terms of user interface.
You are required to perform a particular set of tasks for credit. A short list of extended possibilities is given, for which extra credit will be given. You may invent additional tasks if you like, up to a maximum possible amount of extra credit. Not all tasks you may invent will be worth extra credit. Please consult your instructor if you would like to determine whether or not a task will be worth extra credit before you perform it.

**Required tasks**

In order to make the simulation more complete, you must add an additional dispenser to the soda machine, for a beverage of an additional flavor. Add a dispenser to the simulation for a beverage type named “Iced Tea”, by performing the following set of small tasks:

- In `SodaMachineApplication.java`, there is a method named `createSodaMachine` that constructs the soda machine used in the simulation. Modify that method so that the soda machine it produces has one additional dispenser of type “Iced Tea” at a cost of $1.00.

- Compile, run, debug, and test your work.

It is no longer appropriate for dispenser buttons on soda machines to use text labels. A series of eye-catching icons were provided in the laboratory archive. Change the text labels for buttons to icons by performing the following set of small tasks:

- In `SodaMachineView.java`, there is a method named `createButtonPanel` that constructs the panel of dispenser buttons for the soda machine user interface. Initially, this method creates `SimpleAction` objects by passing the `String` type of the beverage to the constructor for the `SimpleAction`. This results in buttons with text labels. In order to create buttons with icon labels, an icon must be passed to this constructor.

- Create a protected method named `createIconFor` in the `SodaMachineView` class. Its return type should be `Icon`, and it should take one parameter of type `String`. This method should create and return an appropriate icon for each of the known beverage types. Its signature should look like the following:

```java
protected Icon createIconFor(String type)
```

- The following code will choose among different `String` beverage types:

```java
if (type.equalsIgnoreCase("Cola"))
    // create and return Cola icon
else if (type.equalsIgnoreCase("Diet"))
    // create and return Diet icon
    . . . .
else
    // create and return Iced Tea icon
```
It is acceptable for the purpose of this exercise that the icon for “Iced Tea” is considered to be the default icon.

- The following code will create an icon named cola from a picture named cola.gif:

  ```java
  Icon cola = new ImageIcon("cola.gif");
  ```

  Use this example as the template for creating each of the five different icons that correspond to the five beverage types used in this simulation, so that you can return the appropriate icon in the createIconFor method.

- Change the createButtonPanel method so that it uses your createIconFor method. Instead of constructing a new SimpleAction object with the type of beverage for a name, construct new SimpleAction with no name and the icon returned by calling your createIconFor method with the given type of beverage. In order to create an action with no name and a given icon, you need to call the following SimpleAction constructor:

  ```java
  new SimpleAction(null, icon);
  ```

- Compile, run, debug, and test your work.

The soda machine simulation does not give useful feedback when the user tries to purchase a beverage without entering enough currency, or when the user tries to purchase a beverage from an empty dispenser. Provide more useful feedback by performing the following set of tasks:

- In SodaMachineView.java, there is a method named dispenseFrom, which attempts to dispense a beverage from the dispenser at a given index. Initially, this method is written to respond to all possible errors in the same fashion. This is accomplished by the following segment of code:

  ```java
  Beverage b;
  try {
    b = getModel().dispense(index);
  } catch (Exception ex) {
    showMessage("Error!", "There was an error.");
    return;
  }
  ```

- There are two possible types of exception that can be thrown by the dispense method of the object returned my the getModel method:

  ```java
  NotEnoughCurrencyException
  BeverageUnavailableException
  ```
Change the existing catch block to catch an exception named `ex1` of the type `NotEnoughCurrencyException`, instead of an exception named `ex` of the type `Exception`. Change the error message to something appropriate for the situation when the user has not entered the required credit to purchase a desired beverage.

- Add a second catch block, immediately after the one you just modified, to catch an exception named `ex2` of type `BeverageUnavailableException`. You can copy and paste the existing catch block to make it easier. Change the error message in the second catch block to something appropriate for the situation when the user desires to purchase a beverage from a dispenser that is actually empty.

- Compile, run, debug, and test your work.

All soda machines are designed to reject dollar bills that are not in good condition, for fear that it is actually a counterfeit bill. Change the simulation so that it rejects 25% of the dollar bills that are entered into the machine by performing the following set of small tasks:

- Create a class named `RejectedBillException.java`. Using the three exception classes provided in the project as a guide, write the class `RejectedBillException` to represent the error condition where a dollar bill was rejected by the soda machine. Save the file in the directory containing the other source code files for this exercise, and add the file to the project.

- In `SodaMachine.java`, there is a method named `insert`, which accepts a `Currency` object entered by the user and applies a credit for its value. Initially, this method simply adds the credit for all currency entered by the user, regardless of whether or not it might be counterfeit. Before the credit is applied, add the following test to the method:

  ```java
  if (c.getName().equalsIgnoreCase("dollar")) {
    if (Math.random() < 0.25) {
      // reject the dollar
    }
  }
  ```

  This code will reject 25% of the dollar bills entered by the user.

- Replace the comment “// reject the dollar” above with code to throw a new `RejectedBillException` with an appropriate error message.
• In SodaMachineView.java, there is a method named insert, which accepts a Currency object from the user and passes the object on to the underlying soda machine data model. Initially, this method is not equipped to handle a possible error caused by inserting currency into the soda machine. Wrap the line of code that inserts the currency into the soda machine data model in a try-catch block:

```java
try {
    // insert into data model
} catch (RejectedBillException ex) {
    // perform rejection tasks
    return;
}
```

The comment “// insert into data model” above should be replaced with the existing line of code that could now possibly result in a RejectedBillException. The comment “// perform rejection tasks” above should be replaced with code which shows a message – in the way a message is shown in the dispenseFrom method – that informs the user of the rejected bill.

• Compile, run, debug, and test your work.

Possible extra credit tasks

You do not have to perform any of these tasks unless you would like to do so.

Change the advertisement displayed on the soda machine to something more interesting, or more appropriate. The paintComponent method in the Advertisement class initially paints a simple graphical advertisement. Using methods available in the Graphics2D class, and the Graphics class it extends, call methods on the Graphics2D object g2 in order to paint a better advertisement. You may add to the existing advertisement, or invent a completely new picture. Simply changing the displayed text, or one of the colors used in the advertisement, will not result in extra credit. Be creative.

Initially, the restock method in the SodaMachine class, called by the restock method in the SodaMachineView class, always restocks each of the dispensers of the soda machine with 1 – 6 beverages of the appropriate type for the dispenser. Create a method named wackyRestock in the SodaMachine class, that sometimes places a beverages of the wrong type in each of the dispensers. Create a method named wackyRestock in the SodaMachineView class that calls this method in the same way as restock in SodaMachineView calls restock in SodaMachine. In SodaMachineApplication, create an Action similar to that for the normal restock, which calls the wackyRestock method in the SodaMachineView. Add this action to the user interface in the same fashion as the restock action.
Add a button to the user interface for the soda machine that returns the credit entered by the user without having to purchase a beverage.

Initially, there is no way for a user to know the price of a beverage from a desired dispenser. When a dispenser button is pressed, and the credit is exactly zero, show the price of a beverage from that dispenser for a short time, and then return the LCD display to showing a credit of zero. (This one is somewhat difficult.)

You may invent your own tasks. Extra credit will be given for tasks of comparable difficulty that add to the simulation. Ask your instructor if you are concerned that a task won’t be worth extra credit.

**Testing and troubleshooting**

While writing the code required to complete the tasks for this laboratory, you may receive notice of many different kinds of errors. A listing of compiler errors and possible causes is available on the course website. Contact your instructor if you have further problems.

When you are ready to test your QueueADT implementation, you must make that implementation available for use in the application. In SodaMachineApplication.java, there is a method named `createQueue` that returns a new QueueADT implementation. Change this method to return a new queue of the type you have created, instead of a new queue of the type that was provided for you.

In order to test your QueueADT implementation, you must use that class in the application and test it by interacting with the soda machine. When you start the application, several beverages will be enqueued in the dispensers of the machine. Insert a dollar into the machine and dispense a beverage of any type. The application will automatically check to ensure that your QueueADT implementation is returning the enqueued beverages in the correct order. If your implementation works incorrectly, you will receive an error message. If the dispenser always dequeues beverages in exactly the same order they were enqueued, and you do not receive any error messages, your queue implementation works correctly. You should test your implementation in as many ways as you can imagine, in order to ensure it works in all cases.

The following are problems you may encounter when testing a QueueADT implementation:

**Message “There was an error.”** Before modifying the soda machine to make it more realistic, this message is displayed for either or two possible reasons: First, there might not have been enough currency entered into the machine to make the attempted purchase. Second, the dispenser for the requested beverage type might have been empty. After the second task has been completed, you should not see this message.
Message “This beverage was dequeued instead of the correct beverage in FIFO order.”

You have likely written the enqueue and/or the dequeue methods incorrectly. Analyze the code for these methods in detail and draw pictures showing the nodes in the structure before and after the method is called.

Message “The dispenser reports that it is empty, although it should still contain beverages.”

You have likely written the size and/or isEmpty methods incorrectly. Since these methods rely on correct updating of the size variable, you may have written the enqueue and/or dequeue methods incorrectly. Analyze the code for these methods and draw pictures showing the nodes in the structure before and after the method is called.

Message “The dispenser dequeued a beverage, although it should have been empty.”

You have likely written the enqueue and/or the dequeue methods incorrectly. Analyze the code for these methods in detail and draw pictures showing the nodes in the structure before and after the method is called.

Message “The dispenser produced a front beverage, although it should have been empty.”

You have likely written one of the methods in your implementation incorrectly. Analyze the code for the methods in detail and draw pictures showing the nodes in the structure before and after each method is called.

Message “The dispenser produced a front beverage different from the correct beverage in FIFO order.”

You have likely written one of the methods in your implementation incorrectly. Analyze the code for the methods in detail and draw pictures showing the nodes in the structure before and after each method is called.

A NullPointerException is thrown by a method in your class when the simulation is run.

You have likely linked nodes up incorrectly in your class. The problem could be in the constructor, the enqueue method, or the dequeue method. Analyze the code for these methods and draw pictures showing the nodes in the structure before and after the method is called.

A ClassCastException is thrown by a method in your class when the simulation is run.

Your method likely returns a Node instead of the element stored inside the Node. Check your front and dequeue methods to ensure that they return the element stored inside the Node, not the node itself.

The size and isEmpty methods of your queue may be incorrect. Check to see that isEmpty returns true when the size is zero, and false otherwise. Check to see that the size method returns the value for size that is updated every time enqueue or dequeue is called. Check to see that size is updated correctly in those two methods.

No beverage cans show up in the soda machine.
Grading Scheme and Extra Credit

20 Up to 20 points for a linked list queue implementation.
15 Up to 15 points for required modifications to the soda machine.
5 Up to 5 points for additional working modifications to the soda machine.
5 Up to 5 points for good style and techniques.

40 “Perfect” score without any extra credit
45 Maximum total grade

Finishing up your work

Once you have completed the exercise, you must go through all of the files you created or modified, and perform a process called refactoring. Refactoring includes editing your code so that it maximizes readability, follows all necessary coding conventions, and contains no spelling errors in any included comments. It is mandatory for you to refactor the code that you produce for each of your laboratory exercises in this way. You will not only be graded on the output of your application and your understanding of the key concepts, but also on the clarity and overall style used in your code.

There are many general code styles that are acceptable for this course, as in industry. You are not required to adhere to any one specific style, but you must consistently follow an acceptable style throughout your work on each laboratory exercise. For more information about code style, consult the reference “The Elements of Java Style,” or contact your instructor.

Once you have refactored your code and are satisfied with both its output and its style, you must submit your work to the instructor. You must submit the following items for this exercise:

- Soft copy of the entire project folder.
- Hard copy (printout) of the file Node.java.
- Hard copy (printout) of the file LinkedListQueue.java.
- Hard copy (printout) of any other files you modified.

Follow the soft copy submission instructions and guidelines as described by your instructor.