



Advanced Object Oriented Programming & Java

Assignment 2 – Arrays, Generic Classes & Polymorphic Menu Handling

Dr. Moshe Deutsch

mdeuts@ruppin.ac.il

Submission Instructions:

1. במשימה שאלה אחת. לכל Public Class בשאלה יש לכתוב קובץ java נפרד C.java כאשר C הוא שם ה class שממומש בקובץ זה.
2. הקוד עבור כל ה Classes חייב להיות Well-Commented!!!
3. בנוסף, יש להעתיק את כל התוכניות במלואן לקובץ WORD אחד. בקובץ זה, עבור כל class שמועתק לשם, יש לציין את שם ה class הממומש (למשל ה Stack או ה Manager וכו'). בנוסף, בסיום, יש לצרף כמה דוגמאות מייצגות לפלט ממסך ההרצה. בתחילת הקובץ הנ"ל, יש לציין את שם המגיש/ה ות.ז. שלו/שלה (**במודגש!**).
4. יש לכתוב הערות לקוד, התומכות ב javadoc (ראו פסקה אחרונה בתיאור המשימה בהמשך). יש להריץ Javadoc על כל הקבצים ולנתב את הפלט שלו לתת ספריה בשם Doc.
5. יש לשים את כל ה sources בתת ספריה Src ואת קובץ ה WORD מעל שתי הספריות (Src ו Doc) במבנה הספריות ולכוון בקובץ ZIP או RAR אחד את כל אלה ולהעלות לאיזור המשימה באתר הקורס, באמצעות ה Moodle.
6. כאשר פותחים את הקובץ המכוון, יש לקבל 3 דברים: Assignment2.docx – קובץ ה WORD (סעיף 3), ספריה Src (המכילה את כל ה Java source files – סעיף 5) וספריה Doc (סעיף 4).
7. תאריך אחרון להגשת המשימה: מוצ"ש 05/12/2020 – עד סוף היום (קרי עד 23:55 בלילה של תאריך זה). **לא ניתן לאחר בהגשה מעבר לתאריך זה המערכת תיסגר באופן אוטומטי להגשות ומי שלא יגיש עד לתאריך זה ייקבל באופן אוטומטי ציון סופי 0 במשימה.**
8. **יש להגיש את המטלה בזוגות!** בתחילת קובץ ה Word – יש לרשום במודגש את הפרטים המלאים של המגישים (שמות פרטיים, שמות משפחה ות.ז.). **רק סטודנט אחד מהזוג המגיש צריך להעלות את המשימה לאתר ה Moodle.**

Exercise 1 – 100%

Generic Queue and Generic Stack Infrastructure in Java and a User interface, based on a Polymorphic array of Menu Handlers.

This assignment is based on the Queue and Stack infrastructure you developed in Assignment 1. The assignment will give you the opportunity of amending the infrastructure to be generic (using the concept of *Generic Classes/Generic Types*) and to develop a user interface that utilises the notion of Polymorphism in handling, generically, every request from the user for operations on a Queue and Stack of Integers.

For the sake of gradualness, you will have 5 phases in this assignment. Nevertheless, you are required to submit only the final version (containing your entire change – after completing the ALL the phases!).

The 5 phases are:

- 1) **Phase I:** Amendments of the Doubly-Linked-List, Queue and Stack infrastructures for converting it to be generic (based on the concept of *Generic Classes/Generic Types*);
- 2) **Phase II:** Make a minimum change in the Manager class, so that it will define a Queue of Integers and a Stack of Integers:

```
private Queue<Integer> q;  
private Stack<Integer> st;
```

Then, test your infrastructure to ensure it works well, after the changes;
- 3) **Phase III:** Develop a Polymorphic infrastructure for the various User Interface menu handlers, which will enable the Manager class to handle ALL user requests (for Queue/Stack of Integers) in a generic manner. This infrastructure will be based on 1 “Interface”, 1 Abstract Class that implements this interface, and 3 classes that inherit from the abstract class, each one provide a specific implementation for the polymorphic function, which will be generically invoked from the Manager class. All the Inputs and messages of this infrastructure will be based on Java GUI “Input

Dialogs” and “Message Dialogs”;

- 4) **Phase IV**: Connect the infrastructure to the Manager class: remove the “switch...case” structure for handling the menu options and replace it with a polymorphic array to handle generically, every request from the user (you will also need to change the menu loop accordingly, so that out-of-bounds inputs are discarded and the exit option is handled properly!) Test the entire infrastructure to ensure ALL your amendments/additions work well!
- 5) **Phase V**: Comment ALL your code, using the Javadoc Style comments and apply Javadoc utility to produce the HTML documentation, as explained at the end of Assignment1 description.

Explanation Movie on YouTube is divided into two movies: Part I and Part II:

- [Click here to watch YouTube Explanation Movie Part I](#)
- [Click here to watch YouTube Explanation Movie Part II](#)

System Design:

You are provided with a Formal System Design, written as a UML Class Diagram:

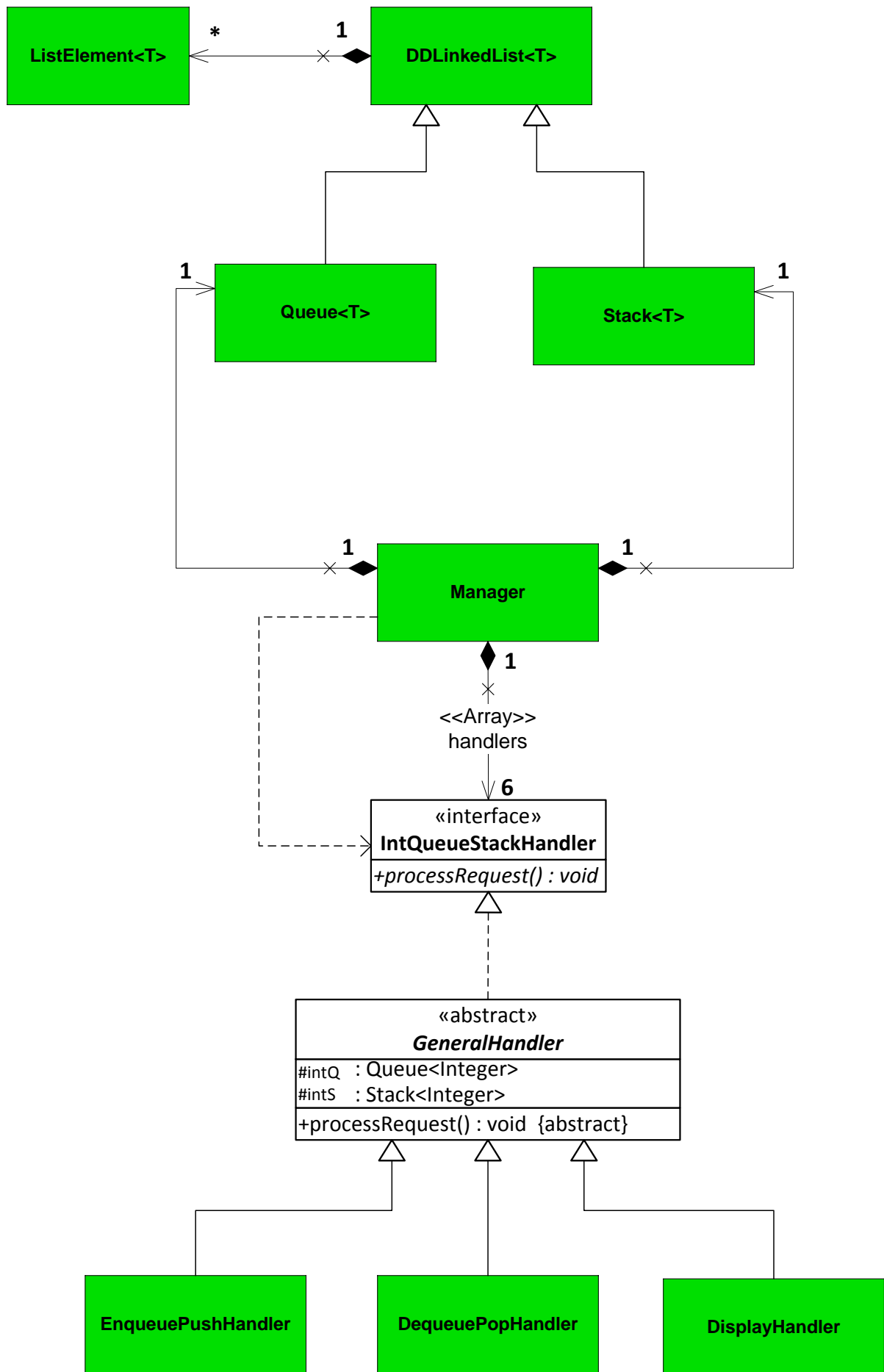


Figure 1. UML System Design.

Phase I:

Amendments of the Doubly-Linked-List, Queue and Stack infrastructures for converting it to be generic (based on the concept of *Generic Classes/Generic Types*);

- 1) Read the first 7 pages of the Oracle Java Tutorial, regarding Generic Types/Generic Classes on:
<https://docs.oracle.com/javase/tutorial/java/generics/types.html>
- 2) Read the first page of the Oracle Java Tutorial, regarding “Boxing” and “Unboxing” automatically between “Wrapper-Classes” and corresponding primitive types, in the context of Generic Classes, initialized to use a “Wrapper Class”.
<https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html>
- 3) Change the entire infrastructure of the classes: DDLinkedList, ListElement, Queue and Stack to be generic –based on the concept of *Generic Classes/Generic Types*. **See the top part of the system design in Figure. 1.**
- 4) Few more changes are needed for this infrastructure. The guidelines for these extra changes are:
 - a. This must be a true infrastructure! Think of it as a generic library (like in the Java API library) that other programmers can use (without being aware of how it is implemented);
 - b. Therefore, there should be NO messages coming out of this infrastructure – remove all the “System.out...” messages from ALL 4 classes of the infrastructure.
 - c. Remove the “display” method from DDLinkedList – **it is not appropriate for the Java way of providing infrastructures!**
 - i. Recall that every “reference type” in Java has “toString” method. The toString method is defined as an empty method in Object class (the root of EVERY inheritance hierarchy in Java - see Java API

documentation). Every class can override it to implement its specific conversion to String. It is invoked automatically in context of String arguments or string operations (such as concatenation).

- ii. Recall, generic types can be instantiated only by "reference types" and not "primitive types". Hence, every "reference type" has a default toString method inherited from Object class. Some will have overridden toString method.
- iii. Taking advantage of this fact, you should **write a “toString” public method for DDLinkedList class** which will return a String object containing the String representations of ALL its elements in one line, where there should be a space between each element in this string. **Think of the most *elegant* and *simple* way of doing this!** This method will be inherited by the generic Queue and Stack classes – where they can use it directly (no need to override it in these classes!)

Phase II:

Make a minimum change in the Manager class, so that it will define a Queue of Integers and a Stack of Integers:

```
private Queue<Integer> q;  
private Stack<Integer> st;
```

You will also need to change the “display cases”, so that now they need to support the advantage you have “toString”, rather than “display” method. **How would you invoke it simply and implicitly?**

Due to Automatic Boxing and Unboxing (that you have read in Phase I), no other changes are supposed to be required, in order to support your so far changes, by the Manager class. **Therefore, if you have done everything correctly, so far, then you should be able to test your infrastructure to ensure it works well, after all the changes;**

Phase III:

Develop a Polymorphic infrastructure for the various User Interface menu handlers, which will enable the Manager class to handle ALL user requests (for Queue/Stack of Integers) in a generic manner. This infrastructure will be based on 1 “Interface”, 1 Abstract Class that implements this interface, and 3 classes that inherit from the abstract class, each one provide a specific implementation for the polymorphic function, which will be generically invoked from the Manager class. All the Inputs and messages of this infrastructure will be based on Java GUI “Input Dialogs” and “Message Dialogs”;

See the bottom part of the system design in Figure. 1.

The idea in this part is to separate between the menu presentation (which should remain as it is from Assignment1) and the Handling of the menu operations. The handling will be implemented by the 3 classes at the bottom of Figure. 1., but will be treated in a generic manner by Manager class, which will create the specific handlers, but will hold their reference in an array of objects, whose type is of the “interface” type (top in inheritance chain) that all these handlers implement. Hence all the manager needs to do is to use the user menu request-option number inputted by the user (if it is valid, of course, and not “exit” option) as an index to this polymorphic array, and to invoke the (same!) polymorphic method each time, but on the object referenced from the corresponding array cell!

All classes in this handler’s infrastructure **must NOT use console I/O!** Instead, **ALL user messages are performed via showMessageDialog method of JOptionPane Java swing class (message dialog GUI), and ALL inputs are performed via showInputDialog method of JOptionPane Java swing class (input dialog GUI)** – see examples of using these in Ch. 3, GUI Case Studies: I, II and III (at the end of the chapter).

Moreover, each one of the 3 classes at the bottom of Figure. 1 **must support the corresponding operation of BOTH Queue of Integers and Stack of Integers, but NOT at the same time!** So, for example, if EnqueuePushHandler object is initialized, it must support an initialization with a reference to a Queue of Integers – from then on, it

will support ONLY enqueue operations on this Queue of Integers, whose reference it was initialized with. However, it must also support an initialization with a reference to a Stack of Integers, so if EnqueuePushHandler object is initialized with a reference to a Stack of Integers – from then on, it will support ONLY push operations on this Stack of Integers, whose reference it was initialized with.

This principle is true for ALL 3 classes, at the bottom of Figure. 1. The way to implement this is by **constructor overloading!** As you shall see next.

IntQueueStackHandler interface:

Interfaces in Java provide the ability to define a “contract” between systems/implementations/concepts, so that if a class depends on such interface, then it must **implement** it.

You can read more about *interfaces* on:

<https://docs.oracle.com/javase/tutorial/java/concepts/interface.html>

Basically an interface is similar to an abstract class, but **ALL its methods are abstract** (i.e. they have no implementations!), whereas an abstract class can have some of its methods implemented (an abstract class is a class which has **at least one method abstract**). Moreover, an **interface in Java cannot contain data attributes**, whereas an **abstract class can have data attributes** – we shall take advantage of this in our design!

You can read more about *abstract classes and methods* on:

<http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html>

public interface IntQueueStackHandler

Must be implemented in its own file: “IntQueueStackHandler.java”.

The only method defined in this interface is:

```
public void processRequest();
```

This is the polymorphic method that each specific handler would need to implement.

GeneralHandler class:

public abstract class GeneralHandler implements IntQueueStackHandler

Must be implemented in its own file: "GeneralHandler.java".

Since an interface cannot hold "data members", we create an abstract class to hold the reference of either the Queue or the Stack, in order to be able to perform the given menu operation on it. This reference is provided into the constructor: the idea is to use constructor overloading, so that an object of such (inherited) class, will be able to **hold a reference to either a Stack or a Queue - but not both at the same time!**

Either of these constructors MUST be used, in order to create an object – this guarantees that EXATLY ONE reference (Queue/Stack) is there to work with! Which one? The one that is NOT null: one will be initialized by the constructor, whereas the other one will be initialized to null by default (default initialization for instance/class variables of "reference type"). In this way, we can have generic operation, like adding an item, but the implementation will need to check which element is being served (Stack or a Queue), while the other one will be null!

The reason for creating this class, and letting all the other handler classes inherit from it (rather than having each of the handlers implement the "IntQueueStackHandler" interface) is **prevention of repetitions/redundancy**: if we didn't have this class, then each one of the handlers would need to implement the "IntQueueStackHandler" interface and define two references (Queue and Stack references) on its own! In this abstract class way, we define these two references once, then get all the other handlers simply inherit from it - thus, **no**

duplications in the definition of the handlers!

All the handlers inheriting from it need to do is simply to initialize these two references, by invoking the "super" constructor, and, in addition, to implement the abstract method processRequest with their own specific implementation!

The Attributes of GeneralHandler:

- 1) protected Queue<Integer> intQ;
Reference to the Queue of Integers.
- 2) protected Stack<Integer> intSt;
Reference to the Stack of Integers.

The Methods of GeneralHandler:

- 1) public GeneralHandler(Queue<Integer> intQ)
GeneralHandler constructor, initializes a new GeneralHandler with a reference to a Queue of Integers (the Stack reference will be initialized to null by default): this.intQ = intQ;
- 2) public GeneralHandler(Stack<Integer> intSt)
GeneralHandler overloaded constructor, initializes a new GeneralHandler with a reference to a Stack of Integers (the Queue reference will be initialized to null by default): this.intSt = intSt;
- 3) *Since this class "implements" the interface IntQueueStackHandler, it inherits from it the abstract method:*
public void processRequest();
This class does not implement this method (keeps it abstract), so there is no need to explicitly define this method here!

EnqueuePushHandler class:

public class EnqueuePushHandler extends GeneralHandler

Must be implemented in its own file: "EnqueuePushHandler.java".

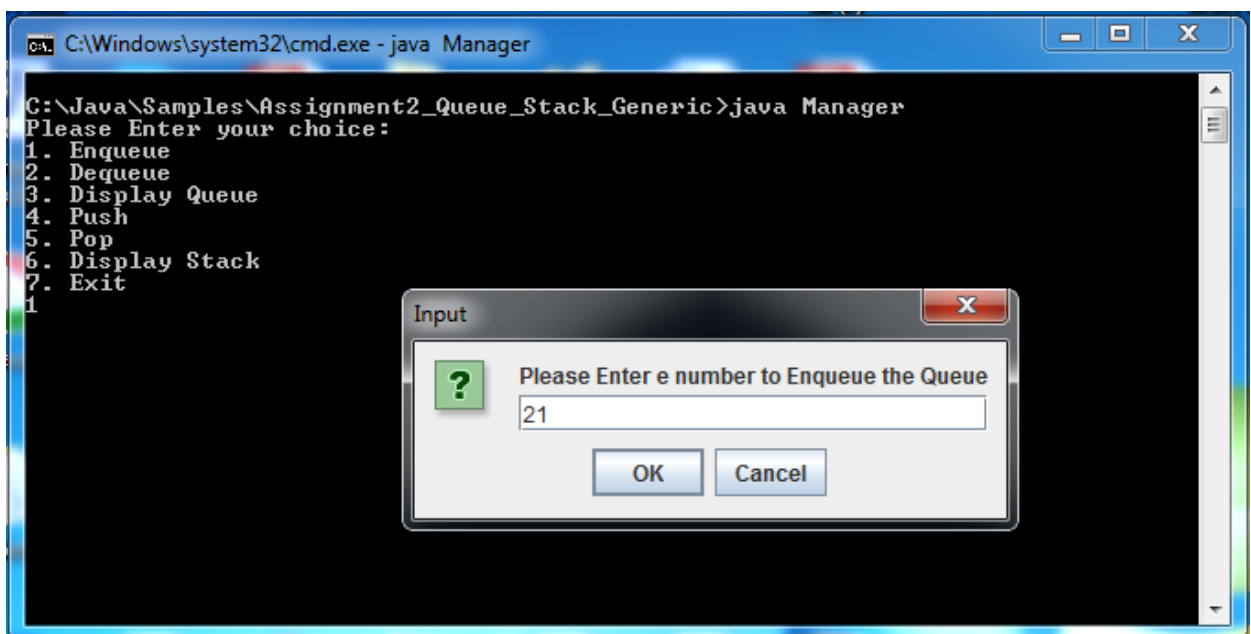
This Class inherits from the abstract class "GeneralHandler" - it defines two overloaded constructors, in order to match the overloaded constructors in "GeneralHandler" class. Each of these constructors simply invokes the appropriate "super" (super class constructor) constructor and pass it the corresponding reference to Integer Queue/Stack respectively. It then implements "processRequest" to acquire input from the user via showInputDialog method of JOptionPane Java swing class (input dialog GUI). It validates the input and if valid puts it in the Queue/Stack respectively.

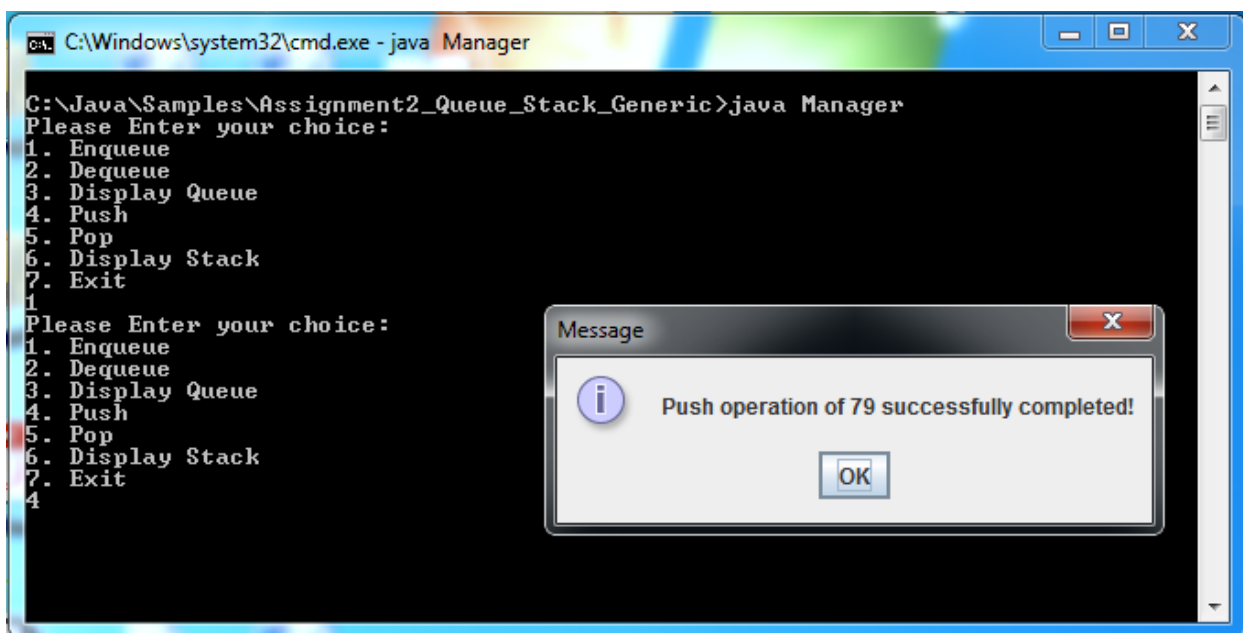
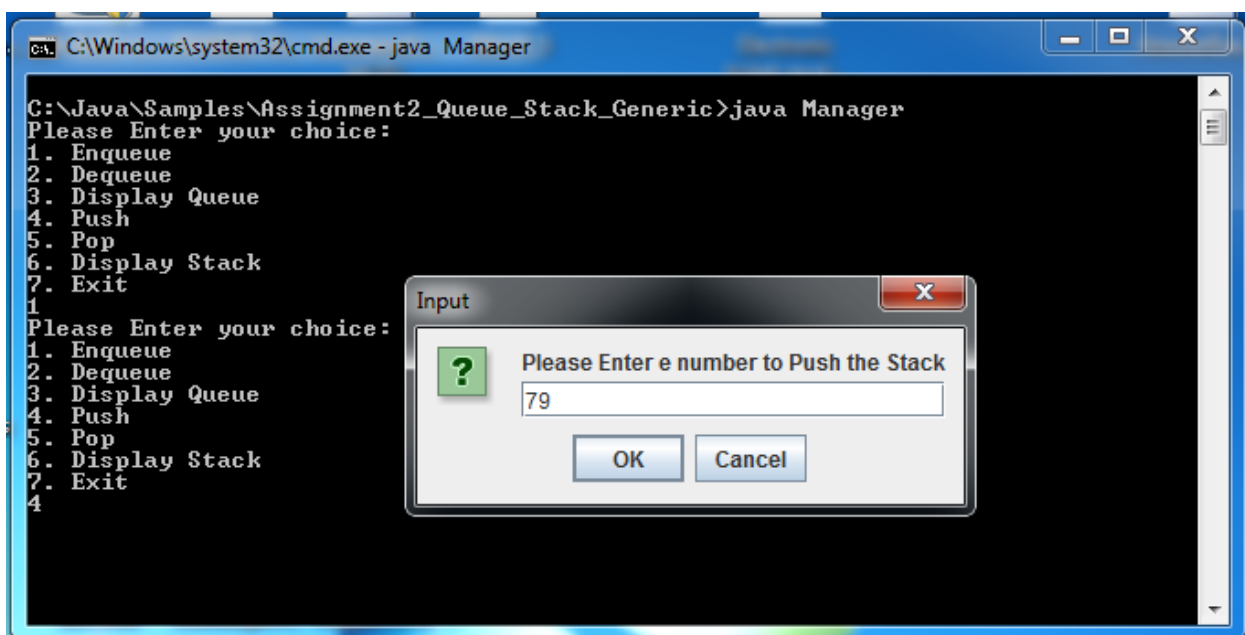
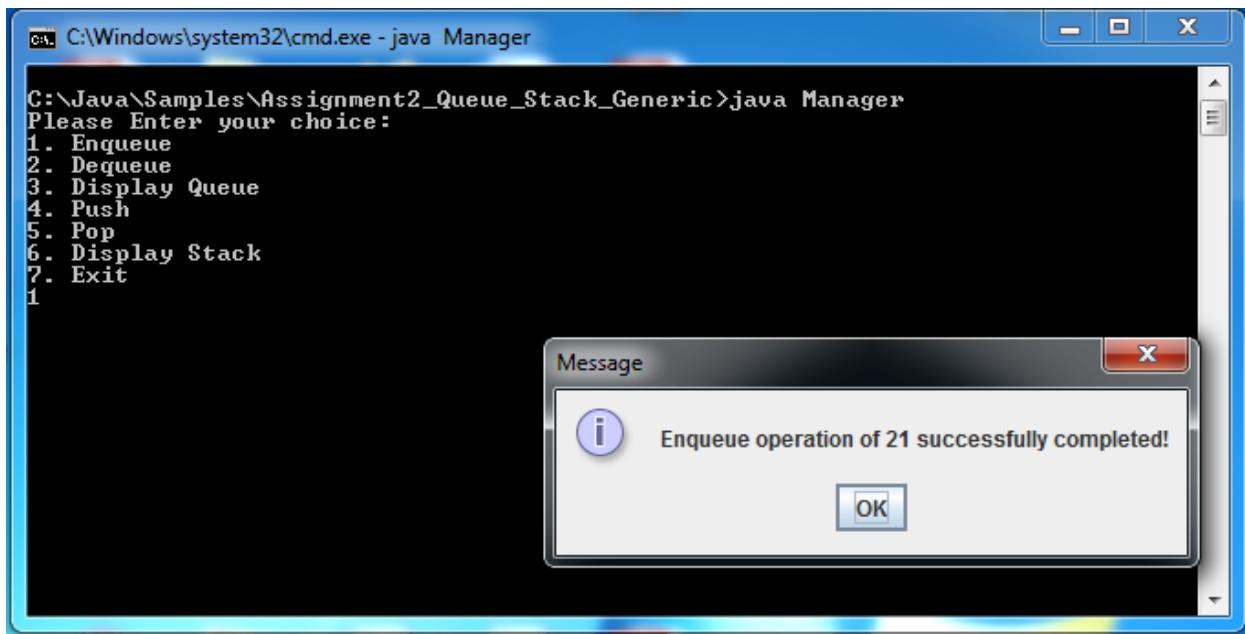
The Methods of EnqueuePushHandler:

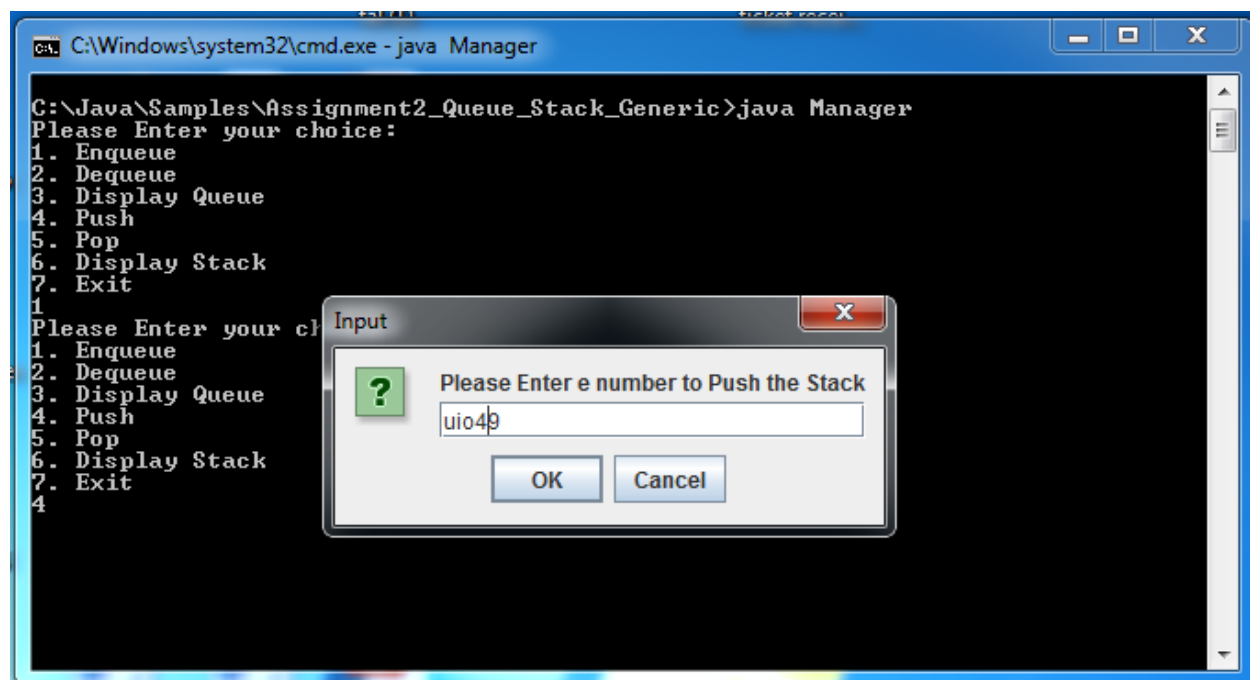
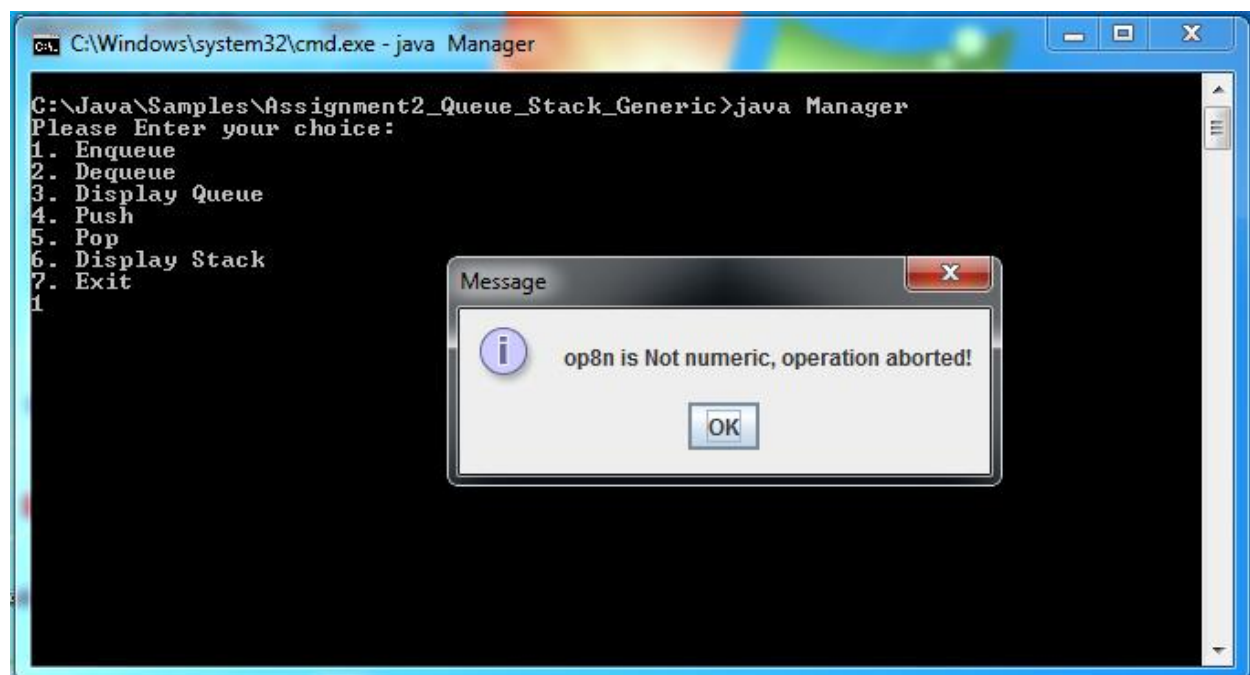
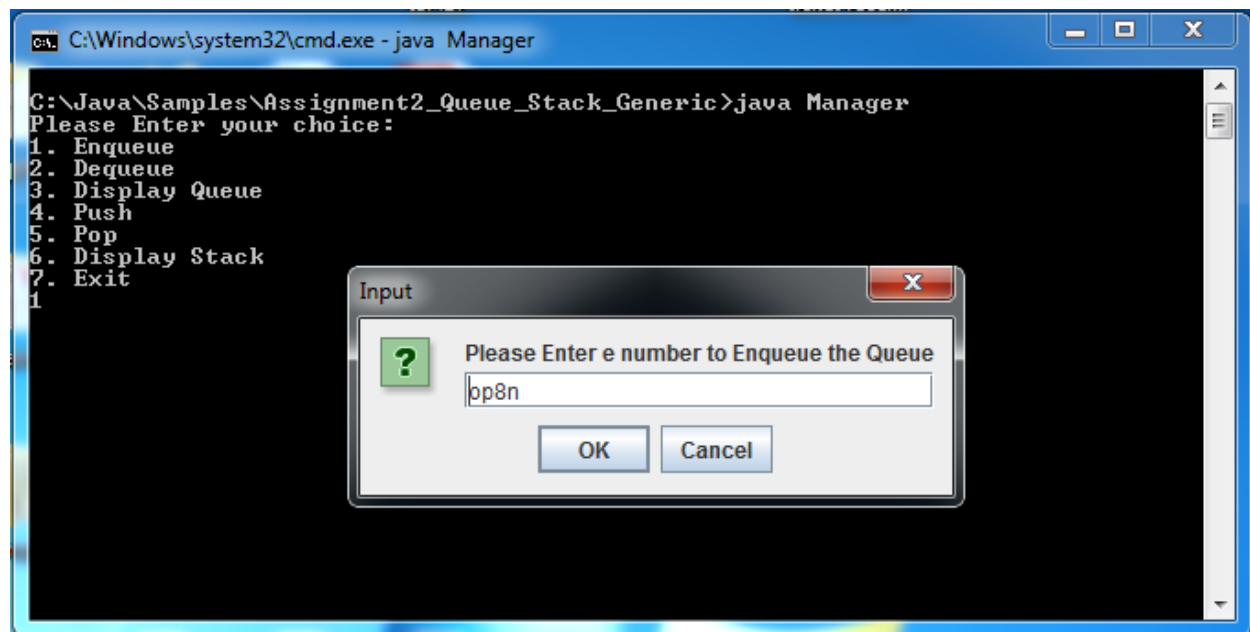
- 1) public EnqueuePushHandler(Queue<Integer> intQ)
EnqueuePushHandler constructor, initializes a new EnqueuePushHandler with a reference to a Queue of Integers (the Stack reference will be initialized to null by default) - *passed via call to the super constructor.*
- 2) public EnqueuePushHandler(Stack<Integer> intSt)
EnqueuePushHandler overloaded constructor, initializes a new EnqueuePushHandler with a reference to a Stack of Integers (the Queue reference will be initialized to null by default) - *passed via call to the super constructor.*
- 3) public void processRequest()
This method implements the abstract method "processRequest", inherited from GeneralHandler class. This method acquires input from the user via showInputDialog method of JOptionPane Java swing class (input dialog GUI). It validates the input and if valid puts it in the Queue/Stack respectively. ALL user messages are performed via showMessageDialog method of JOptionPane Java swing class (message dialog GUI), and ALL inputs are performed via showInputDialog method of JOptionPane Java swing class (input dialog GUI).

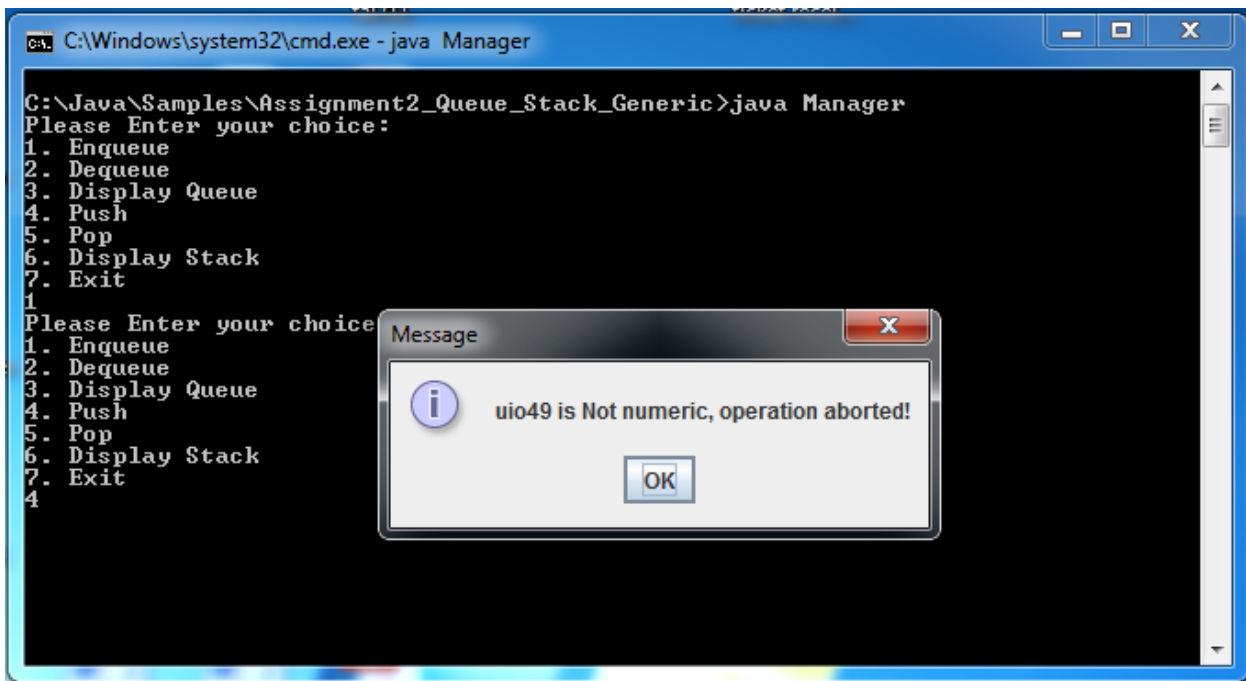
Some guidelines for implementing this method:

- a) ALL user messages are performed via showMessageDialog method of JOptionPane Java swing class (message dialog GUI), and ALL inputs are performed via showInputDialog method of JOptionPane Java swing class (input dialog GUI) – see examples of using these in Ch. 3, GUI Case Studies: I, II and III (at the end of the chapter).
- b) Validation means checking the String input the user has entered (via input dialog GUI) is numeric – **you must use Exception handling mechanism to implement this validation in your code!**
- c) Since this method serves both enqueue and push operations, **you must be as generic as possible!** For example, instead of invoking showMessageDialog method for each element – define String objects that will hold the message subjects and then assemble the message string (using String.format method – See Ch.3, case study I) and invoke showMessageDialog only once, with the final “generic” message string.
- d) Some representative sample Screen shots and the message results of some Enqueue/Push operations, so you can get the “look and feel” of what is required:









DequePopHandler class:

public class DequeuePopHandler extends GeneralHandler

Must be implemented in its own file: "DequePopHandler.java".

This Class inherits from the abstract class "GeneralHandler" - it defines two overloaded constructors, in order to match the overloaded constructors in "GeneralHandler" class. Each of these constructors simply invokes the appropriate "super" (super class constructor) constructor and pass it the corresponding reference to Integer Queue/Stack respectively. It then implements "processRequest" to Pop/Dequeue an item from Stack/Queue respectively. Then present it on a GUI message dialog.

The Methods of DequeuePopHandler:

1) Two overloaded constructors – in a similar way to EnqueuePushHandler class.

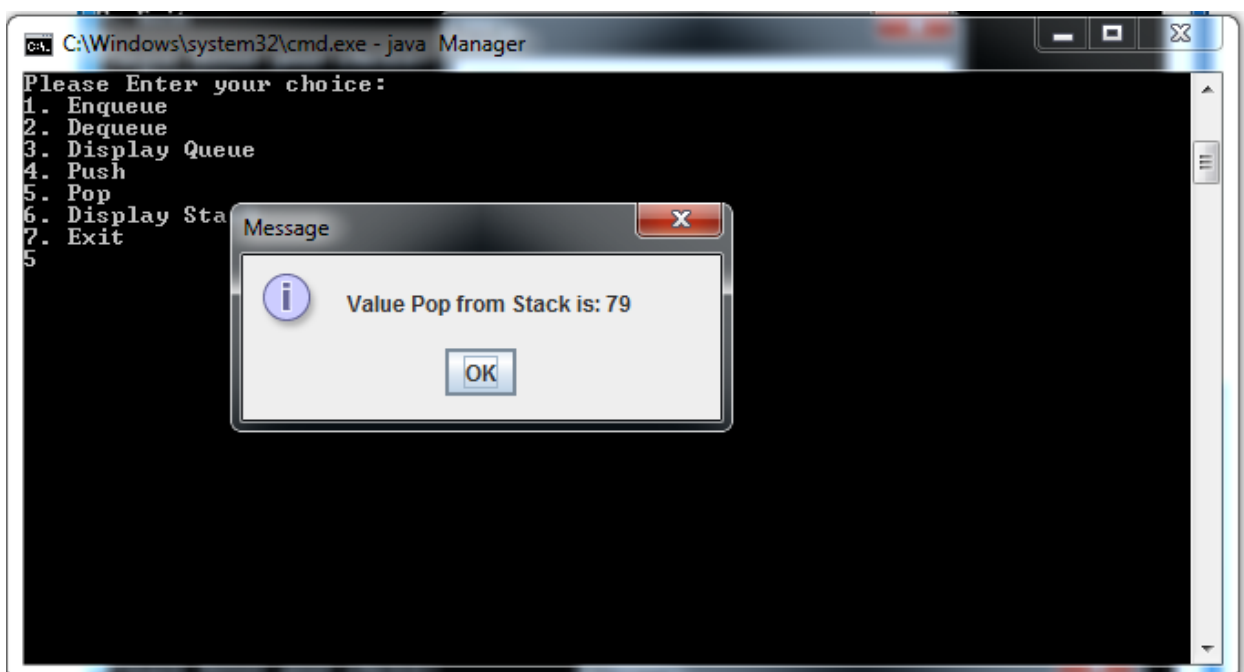
2) public void processRequest()

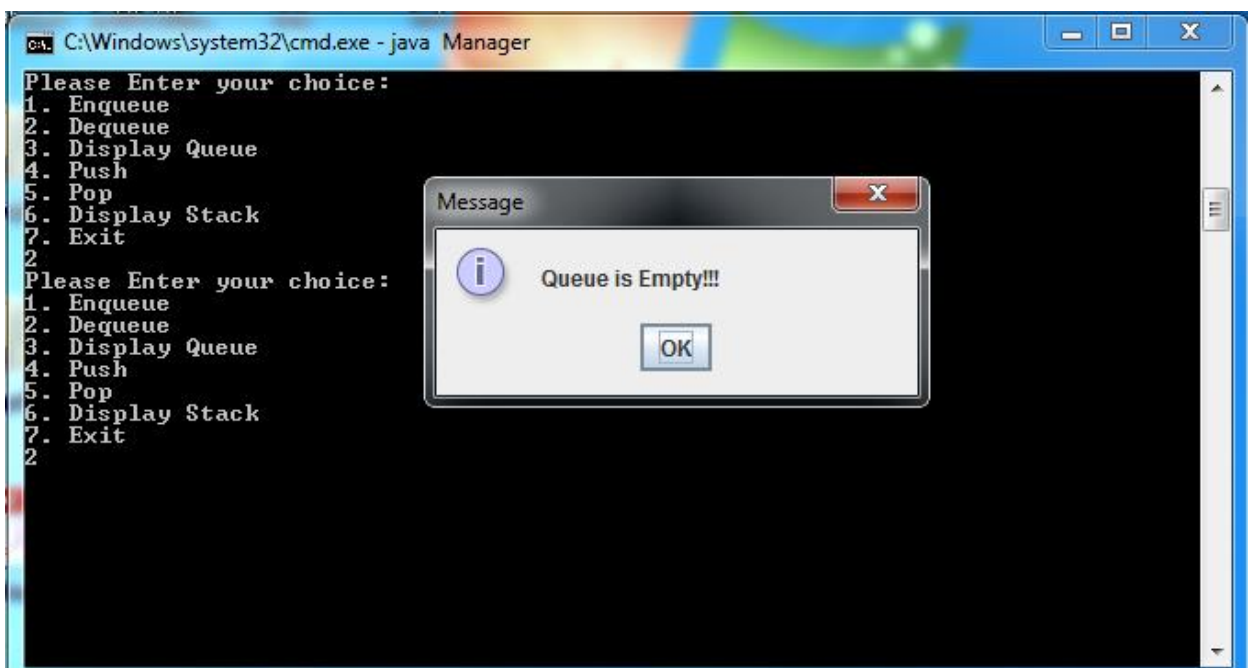
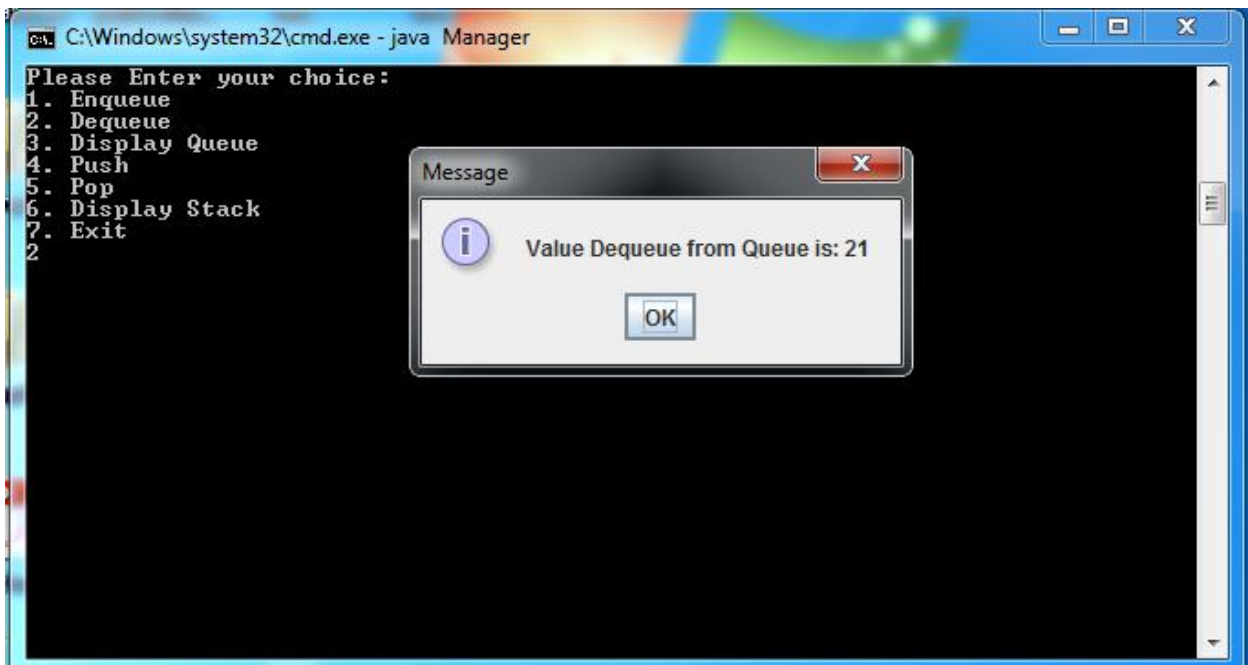
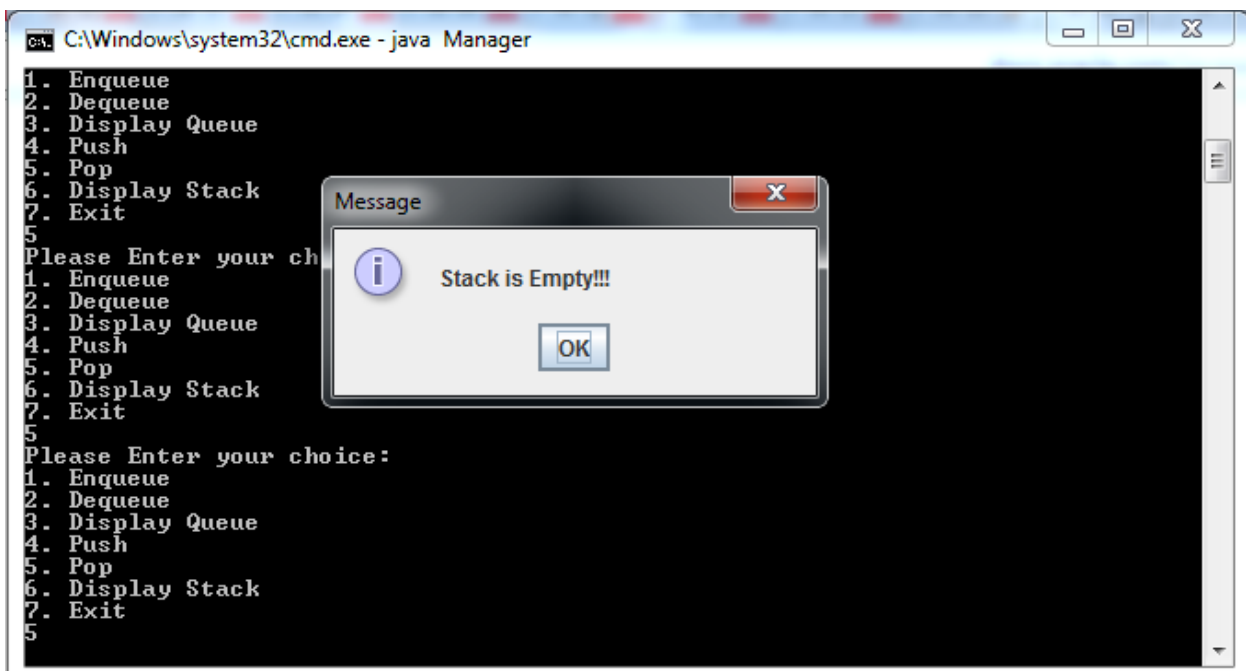
This method implements the abstract method "processRequest", inherited from GeneralHandler class. This method Dequeue/Pop an item from the Queue/Stack respectively, and presents it on via showMessageDialog method of JOptionPane. **If the examined**

Queue/Stack is empty, issues an appropriate message to the user, using showInputDialog method of JOptionPane Java swing class (input dialog GUI).

Some guidelines for implementing this method:

- a) Likewise, ALL user messages are performed via showMessageDialog method of JOptionPane Java swing class (message dialog GUI) - see examples of using these in Ch. 3, GUI Case Studies: I, II and III (at the end of the chapter).
- b) Since this method serves both dequeue and pop operations, **you must be as generic as possible!** For example, instead of invoking showMessageDialog method for each element – define String objects that will hold the message subjects and then assemble the message string (using String.format method – See Ch.3, case study I) and invoke showMessageDialog only once, with the final “generic” message string.
- c) Some representative sample Screen shots and the message results of some Dequeue/Pop operations, so you can get the “look and feel” of what is required:





DisplayHandler class:

public class DisplayHandler extends GeneralHandler

Must be implemented in its own file: "DisplayHandler.java".

This Class inherits from the abstract class "GeneralHandler" - it defines two overloaded constructors, in order to match the overloaded constructors in "GeneralHandler" class. Each of these constructors simply invokes the appropriate "super" (super class constructor) constructor and pass it the corresponding reference to Integer Queue/Stack respectively. It then implements "processRequest" to display the contents of the Queue/Stack respectively - present it on a GUI message dialog.

The Methods of DisplayHandler:

- 1) Two overloaded constructors – in a similar way to the two other classes that inherit from GeneralHandler.
- 2) public void processRequest()

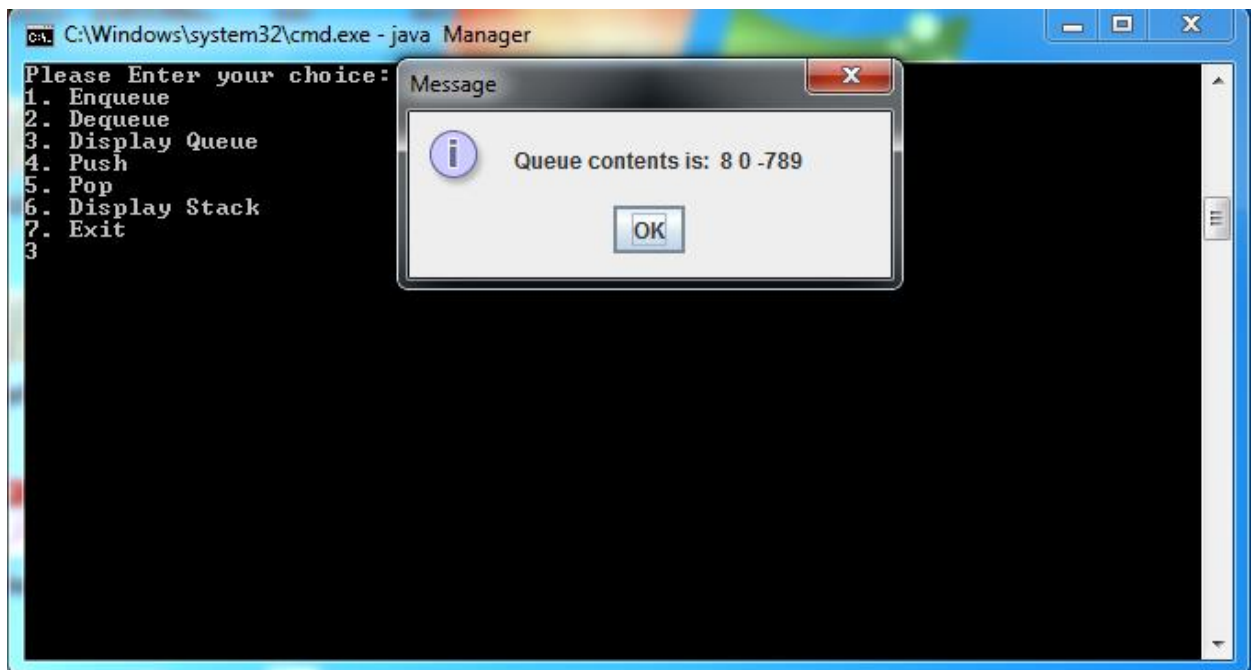
This method implements the abstract method "processRequest", inherited from GeneralHandler class. This method display the contents of the Queue/Stack respectively - present it on a GUI message dialog, using showMessageDialog method of JOptionPane Java swing class. Get the contents of the Queue/Stack, simply by using the corresponding toString method of Queue/Stack class (recall: they got it for free from DDLinkedList class).

Some guidelines for implementing this method:

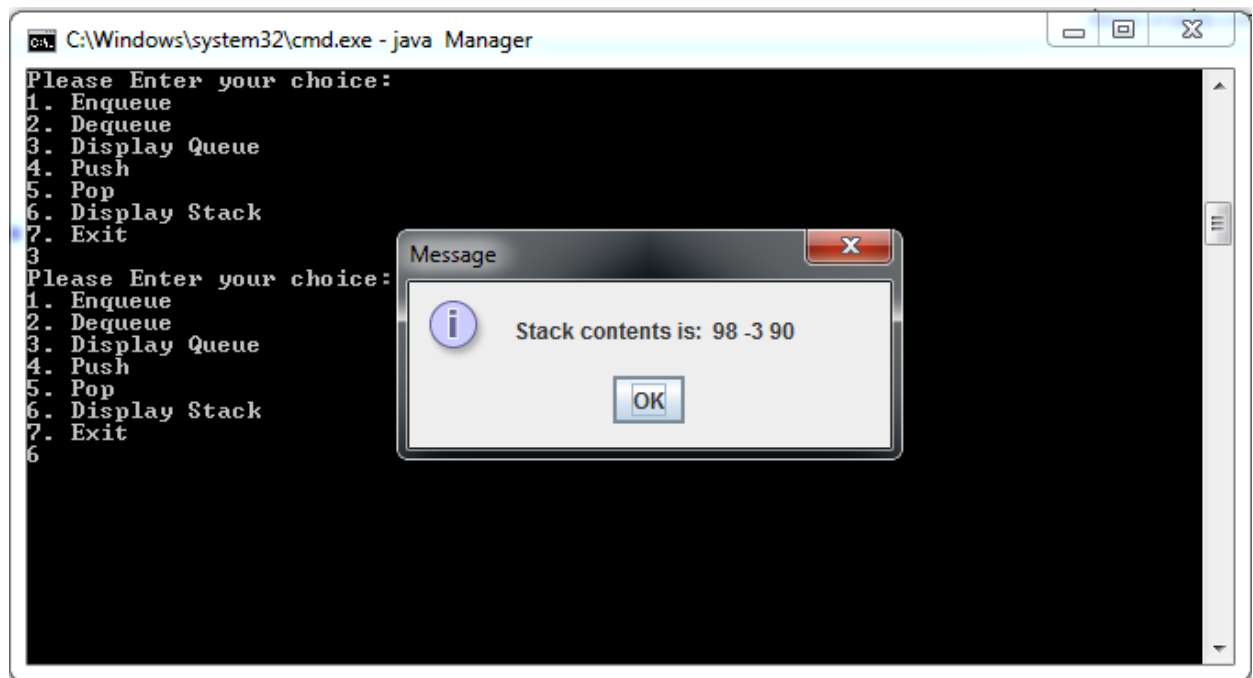
- a) Likewise, ALL user messages are performed via showMessageDialog method of JOptionPane Java swing class (message dialog GUI) - see examples of using these in Ch. 3, GUI Case Studies: I, II and III (at the end of the chapter).

- b) Since this method serves both the display of Queue and Stack, **you must be as generic as possible!** For example, instead of invoking showMessageDialog method for each element – define String objects that will hold the message subjects and then assemble the message string (using String.format method – See Ch.3, case study I) and invoke showMessageDialog only once, with the final “generic” message string.
- c) Some representative sample Screen shots and the message results of some Queue Display/Stack Display operations, so you can get the “look and feel” of what is required:

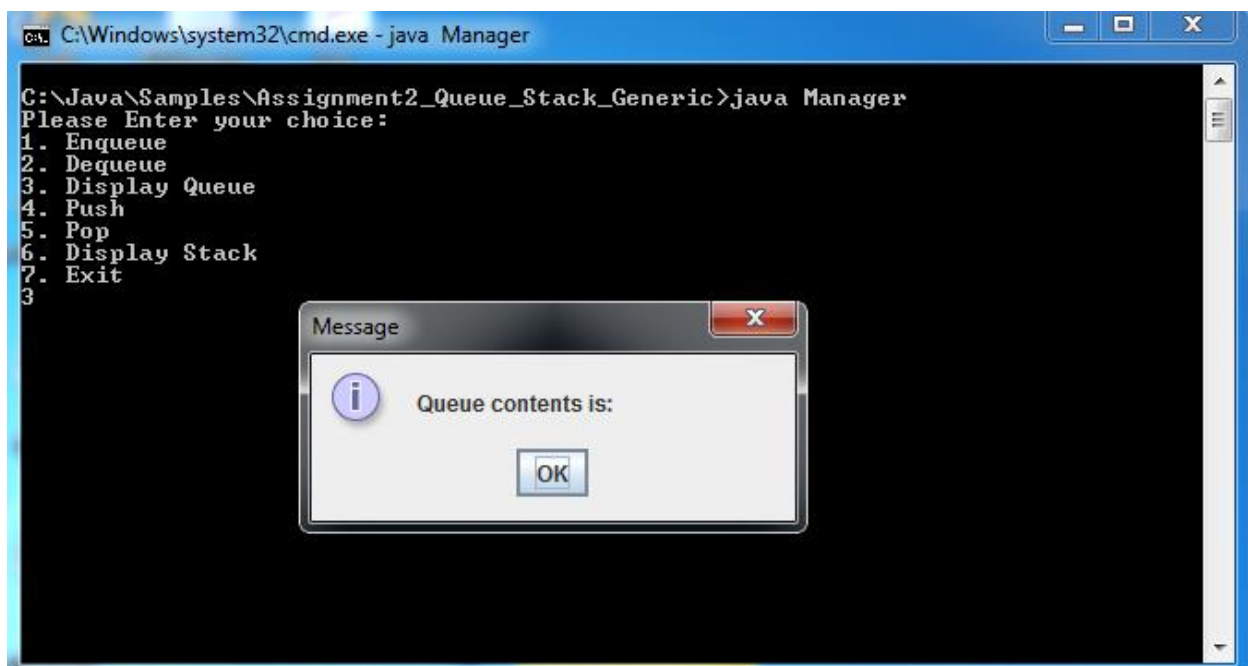
A Queue containing: 8, 0, -789 is displayed as follows:



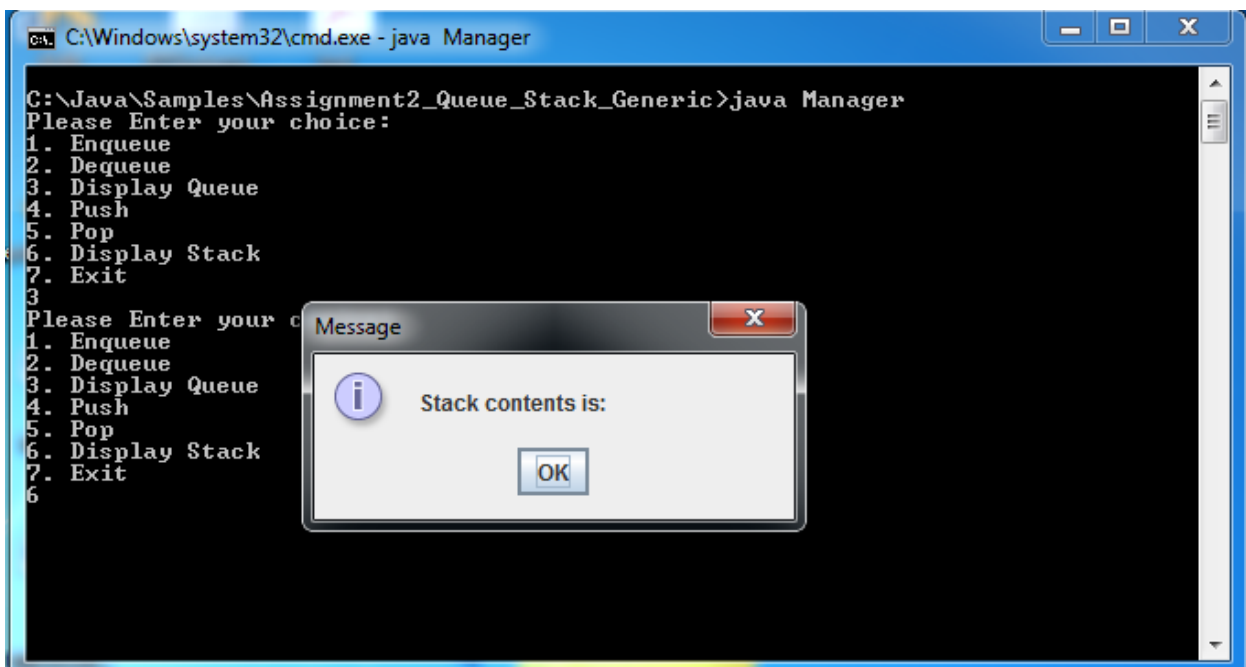
A Stack containing: 98, -3, 90 is displayed as follows:



An Empty Queue is displayed as follows:



An Empty Stack is displayed as follows:



```
C:\Windows\system32\cmd.exe - java Manager
C:\Java\Samples\Assignment2_Queue_Stack_Generic>java Manager
Please Enter your choice:
1. Enqueue
2. Dequeue
3. Display Queue
4. Push
5. Pop
6. Display Stack
7. Exit
3
Please Enter your choice:
1. Enqueue
2. Dequeue
3. Display Queue
4. Push
5. Pop
6. Display Stack
7. Exit
6
```

Message

Stack contents is:

OK

Phase IV:

Connect the infrastructure to the Manager class: remove the “switch...case” structure for handling the menu options and replace it with a polymorphic array to handle generically, every request from the user (you will also need to change the menu loop accordingly, so that out-of-bounds inputs are discarded and the exit option is handled properly!) Test the entire infrastructure to ensure ALL your amendments/additions work well!

- 1) You should now declare and initialize a Queue and Stack of Integers, with respect to the “Generic Class/Type” support of your infrastructure.
- 2) The Menu should be displayed in an identical way to how it was presented in Assignment1:
 - a. Menu is presented in the console window!
 - b. Menu options are inputted from console window (using Scanner object).
- 3) All Array loop handling in this class, which does not require specific use of indices must be performed using “Enhanced For Loop” (see Ch.3 and Ch.4).
- 4) Remove the “switch...case” structure for handling the menu

options and replace it with a polymorphic array to handle generically, every request from the user. The array definition and initialization should reflect the description in Figure 2:

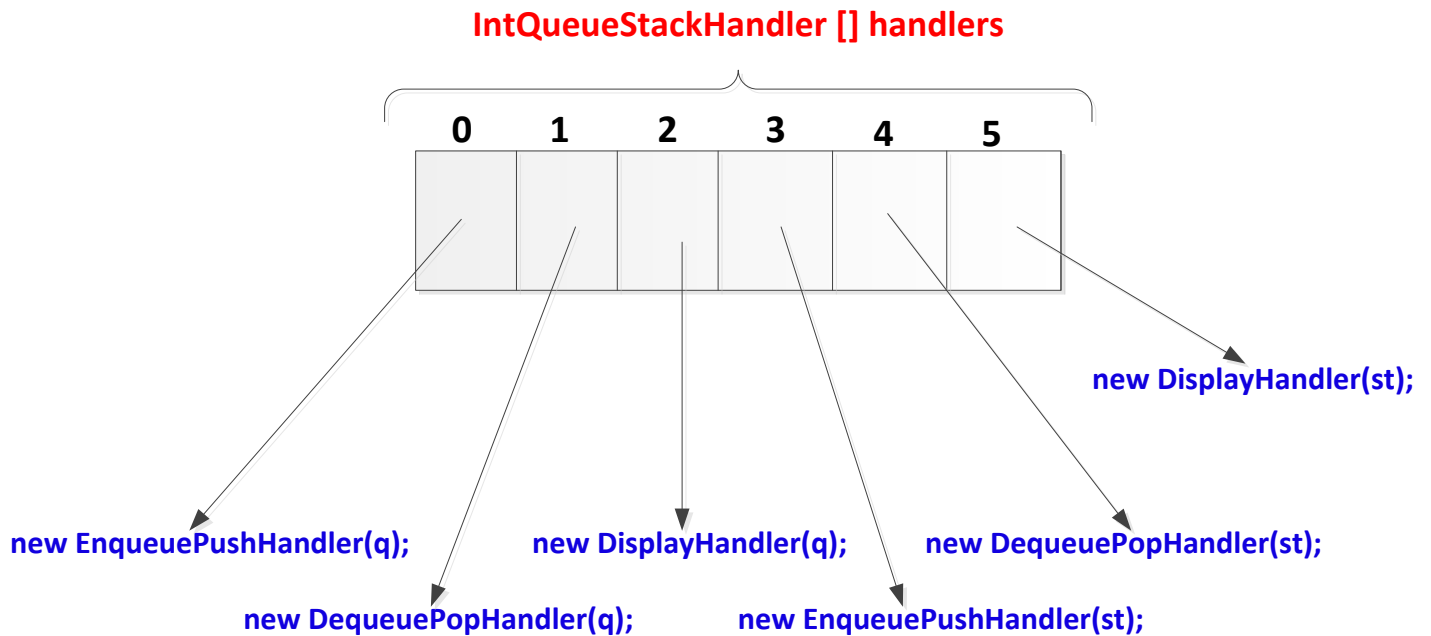


Figure 2. Polymorphic Array

5) I suggest to use a private Method :

```
private void initHandlers()
```

To both allocate and initialize the array of handlers. This method is to be invoked from the constructor, before it invokes “displayMenu” Method.

6) Notice that you will also need to change the menu loop in “displayMenu” method accordingly: so that out-of-bounds inputs are discarded. **You should use Exception handling for this.** Hint: **what is the exception thrown when an array is accessed out of its bounds?** This Exception you need to handle when accessing the polymorphic array! **The message, in case out-of-bounds option is inputted by the user, should be shown on a GUI Message Dialog, as follows:**

```
C:\Windows\system32\cmd.exe - java Manager
C:\Java\Samples\Assignment2_Queue_Stack_Generic>java Manager
Please Enter your choice:
1. Enqueue
2. Dequeue
3. Display Queue
4. Push
5. Pop
6. Display Stack
7. Exit
0
```

Message

0 is an invalid choice! Please try again.

OK

```
C:\Windows\system32\cmd.exe - java Manager
C:\Java\Samples\Assignment2_Queue_Stack_Generic>java Manager
Please Enter your choice:
1. Enqueue
2. Dequeue
3. Display Queue
4. Push
5. Pop
6. Display Stack
7. Exit
8
Please Enter your choice:
1. Enqueue
2. Dequeue
3. Display Queue
4. Push
5. Pop
6. Display Stack
7. Exit
8
```

Message

8 is an invalid choice! Please try again.

OK

- 7) You must handle the “exit” option properly! **Think of a way to include it in the above array handling...** Clue, the command **System.exit(0)**; will terminate the running application properly – the question is how to incorporate it in a handler object...
- 8) Test your entire system carefully.

Phase V:

Comment ALL your code, using the Javadoc Style comments and apply Javadoc utility to produce the HTML documentation, as explained at the end of Assignment1 description.

Good Luck!