Assignment 1: The Abstract Data Types (ADTs) Stack, Queue and List

The goals of this practical assignment

- to show how ADT's can be created in Java using classes and interfaces.
- to demonstrate the importance of separating an ADT's interface from its implementation.
- To provide some practice in using the ADT's stack and queue.
- To illustrate how the interface for a list can be created in Java.

To prepare for this practical assignment

Keep the course book, course notes and lectures for this lab close at hand as you work through the tasks. Lectures 1, 2 and the corresponding sections in the course book are also useful. Make sure that you have understood all the steps in the Practice Exercises 1 and 2 before you begin.

Notes and slides from Lecture 4 and Lecture 5 also provide relevant background material for this practical laboratory. Study the following sections in the course book by Mark Allen Weiss, “Data Structures and Problem Solving Using Java”, third edition:

- Figure 2.12; Common standard runtime exceptions.
- 2.3.4; Other `String` methods
- 2.5.4; The `throw` and `throws` clauses
- 4.5; Fundamental inheritance in Java
- 4.5.1; The `Object` class
- 4.6.2; Wrappers for primitive types
- Figure 4.24; The `Integer` wrapper class
- 6.6.2; Stacks and computer languages
- 16.1; Dynamic array implementations
- 16.2; Linked list implementations
- 16.3; Comparison of the two implementations
- 16.5; Double-ended queues

Tasks

1. Write a Java program that uses a stack to test if an alphanumeric expression contains balanced parenthesis. The program will ask the user to type in an expression and will then output a statement declaring whether or not the expression contains balanced parenthesis.
2. Create a number of test cases that are then used to test the program.
3. Write a menu-based test program that will test all the operations offered by a given FIFO-queue (Fist In First Out). The user will be able to test all of the queue operations by choosing selections from the menu. An example operation would be to add an element to the queue such as a text string input via the keyboard. If the user tries to use the queue in an improper way, a warning message will be printed out.
4. Create a number of test cases that will later be used to test the queue.

Group discussions

Discuss the following questions/tasks in the groups discussion forum:

1. There are a number of standard operations on a stack that are covered in the lecture material. Compare these to the given interface Stack that is described below under **Execution of this Practical Assignment**. Are there any standard stack operations that do not have a corresponding operation in the interface? Are there any operations in the interface for a stack that seem to be superfluous within the context of Java?
2. Compare the standard library stack class (`java.util.Stack`) with the stack interface from task 1 above. Try to identify similarities and differences.
3. Many people claim that the designers of the standard library made some mistakes when they constructed the stack ADT. Can you find any weaknesses in the design of the standard library's stack?
4. The stack and queue in tasks 1 and 3 use runtime exceptions to tell the user that an operation has been used in an incorrect way, e.g. that a removal operation has been called for a stack or queue that is empty. Give some other examples of how an ADT can signal improper use and discuss the advantages and disadvantages of using other methods.
5. How is improper use of `java.util.Stack`, `java.util.Queue` and `java.util.List` indicated?
6. Write a Java interface named `TheList` for an ADT named *TheList* (unsorted). The interface does not need to contain all possible operations on an unsorted list but should include the most important standard operations (i.e. create list, destroy list, check if list

is empty or full, number of elements in list) and various insertion and removal operations.

Select one person in group to summarizes the group discussions on questions 1-5 above. The group should also come to a concensus on how to write the interface in question 6. Post the summary and the interface code in the group's discussion area in Blackboard by the indicated due date.

Execution of this Practical Assignment

Start by creating a project (or directory if not using an IDE) for this practical assignment in your account or on your computer. Save the following jar-files in this new directory:

1. **`stack.jar`** (contains `CheckBalance.java`, `Stack.java` plus documentation)
2. **`queue.jar`** (contains `Queue.java`, `TestQueue.java` plus documentation)
3. **`lab1classes.jar`** (contains four class files: `ListQueue`, `ListStack`, `QueueIsEmptyException`, `StackIsEmptyException`)

If the first two files are not automatically unpacked (`jar` = Java Archive format), then you will have to decompress them by hand or use the following jar command:

jar xvf stack.jar
The directory named 'stack' along with some other sub-directories will be created in the current directory when you unpack `stack.jar`. The directory '`queue`' will be created in a similar way when you unpack `queue.jar`.

The third jar-file contains precompiled class files that you will need in order to test run the programs in this practical lab. It is recommended that you put these jar-files in the directory created for this practical lab. The jar-files does not need to be unpacked (or 'decompressed').

Task 1

Begin by studying the methods specified in the interface `Stack` (the file `Stack.java`).

The interface `Stack` is implemented by the precompiled class `ListStack`. The class is part of the package `se.hig.ad1.lab1` and the source code is in the directory named '`stack/src`'. Documentation for the class can be found in the directory '`stack/javadoc`'.

You can begin by implementing the test program that examines the string of characters input by the user to see if it contains balanced parenthesis. Follow the requirements specification in the task description.

Here are examples of expressions that contain balanced parenthesis: `()`, `(())`, `()()`, `((())())`

Here are examples of expressions that contain unbalanced parenthesis: `)(`, `(()`, `())`, `(()(`

It is a good idea to use the skeleton code in the file `CheckBalance.java` (see the jar-file). Write a statistical method named `isBalanced` that inputs a text string containing an alphanumeric expression and returns the value `true` if the expression contains balanced parenthesis, otherwise `false`. Call the method from the main program where you read in the expression from the keyboard.

You do not need to develop a graphical user interface for input and output to the program, if you don't want to. A simple text-based menu will suffice. One suggestion is to use the Java standard class `Scanner`.

An algorithm for balanced parenthesis

The basic idea for this algorithm comes from section 6.6.2 in the course book. Here is the same algorithm in more detailed pseudo code.

```
Create an empty character stack
Assume that the expression contains balanced parenthesis

As long as the parenthesis are balanced and
           there are still characters in the string
    Get the next character from the string
    If the next character is a '('
         Push the character onto the stack
    Otherwise
        If the next character is a ')'
            If the stack is not empty
                Pop from the top of the stack
            Otherwise
                Make a note that the expression is not balanced
                // At this point we have found a right parenthesis
                // with no partner
If the expression is balanced and the stack is not empty
    Make a note that the expression is not balanced
    // We still have left parenthesis without partners
Destroy the stack
```

Hints

1. You have to tell the compiler that the files in `lab1classes.jar` must be included in the project's *classpath* in order to compile the program. To do this, use the flag `-classpath` (followed by the complete file name) when running the compiler from the terminal window. If you are working with an IDE (Integrated Development Environment, such as Eclipse) then you need to add the file to the project's *search path*.
2. Use the method `charAt` in order to access characters at specific positions in the input string. See section 2.3.4 in the course book.
3. The parentheses that are pushed onto the stack are of type `char` but the element types in the stack are `Object`. How should this problem be solved?
   The solution is to wrap the character in a so-called *wrapper-class*. The wrapper class `Character` is provided for this very purpose. The following design can be used to push a character onto the stack:

   ```
   char aCharacter
   . . .
   aCharacter = parenthesisexpression.charAt(i);    // Pick out the i:th
    // character
   thestack.push( new Character(aCharacter));       // Wrap the character
    // and push on stack
   ```

   In the example above, Java does an implicit type conversion from the `Character` class to the `Object` class since the input parameter for the push method is of type `Object`. All classes in Java inherit characteristics from the base class called `Object`. This means that all objects in Java are `Object` objects as well. This is further explained in sections 4.5 – 4.5.1 in the course book.

Why is this necessary? The answer is that `char` is a *primitive* data type and primitive data types are not objects. So, in contrast to our "wrapper" (`Character`), primitive data types such as `char` do not inherit characteristics from the `Object` class. This means that `Character` objects can be pushed onto the stack but not `char` variables. Read about the wrapper class in section 4.6.2 of the course book. Note that the example in figure 4.24 uses approximately the same design as the example above.

An alternative to this design is to use *generics* which was introduced in Java 2 version 1.5. See the list of course literature.

Task 2: Tests

Create a number of test cases for the balanced parenthesis program and then use them to test it. Use screen dumps as appendices to your lab report in order to document the results of your tests. Describe the test cases under the Results section of your report with references to these appendices. Your report will then clearly describe how the stack works. Here are some examples of suitable test cases:

- 1.1 Test of the expression `()`
- 1.2 Test of the expression `)(`
- 1.3 Test of the expression `(()`

Task 3: Test program for a queue

Try using pen and paper to create a Java interface for a FIFO queue. Specify the methods use in the interface, possible input parameters and their return types. Discuss with members of your group if anything seems to be unclear. Then compare the interface you have created with the interface named `Queue.java` (see the jar file).

The interface `Queue` is implemented by the precompiled class named `ListQueue` in the package `se.hig.ad1.lab1` and the source code can be found in the directory '`lab1/queue/src`'. Documentation of the interface is in the directory '`lab1/docs`'. Note that some of the methods cast a "*runtime exception*" (the class `QueueIsEmptyException`) if the user tries to access an element in an empty list.

At this point, you should create the menu-based test program for the queue according to the requirement specifications. Use the skeleton code in the file `TestQueue.java` (see the jar file).

Hints

You can initially ignore any exceptions that are cast by any of the methods in the class `ListQueue`. When your program is working, you can go back to the menu selections that call methods causing the exceptions. Make sure that all exceptions are caught and that the exception objects messages are written out to the terminal window.

Task 4: Tests

Test the FIFO queue in roughly the same way as the parenthesis balancing program in Task 2. Here are some examples of suitable test cases:

- Insertion of an element into an empty queue
- Insertion of an element into a non-empty queue
- Attempt to remove an element from an empty queue
- Removal of an element from an empty queue
- …

Questions 2-3.

There is an implementation of a stack ADT in Java's standard library named `java.util.Stack`. We will not be using this stack ADT from the standard library but it is good to know that it exists when you do other programs.

Question 5.

Use the test program from Practical Exercise 2 in combination with your studies of the API documentation for the relevant classes.

Question 6.

Every group member must create their own interface in the file named `TheList.java`. Use the element type `Object` for the list so that a generic list is produced that can contain any type of element. An alternative is to use *generics* where the element type is chosen at the time of list creation. Follow the examples in the interfaces `Stack.java` and `Queue.java`. Note that you do not need to create a class that implements the interface. You cannot test run your interface but you can compile it and fix any possible compilation errors. Compare your interface with the interface written by other group members. Have a discussion and then choose the "winning" interface.

Presentation

The programming exercises are to be presented individually or by pairs of students (the recommended method). Load up a zip- or jar-file to Blackboard that contains the following

- A written lab report that follows the report template and contains answers to the questions given above. Do not forget to put the names and e-mail addresses of the authors, their personal numbers and the date on the first page of the report. Do not turn in MS Word or Open Office files since they can be contaminated with data viruses. There could also be problems showing them properly at the receiver. Save the report document as a **PDF**-file.
- All of the source code that is part of the assignment including code that you have not personally modified or written.
- Create `javadoc` generated HTML documentation of the classes that you have personally written or modified.

Group discussions are carried out in each individual group. A summary of the groups solution must be published in the discussion forum before the specified due date. Do not forget to state the names of the group members in the summary. Active participation in group discussions is necessary for a passing grade.

If this Practical Laboratory is correctly solved, well documented and turned in before the deadline, one bonus point will be awarded on the final exam.

Good luck! ☺