**A04**

**Due Date:** Thursday, June 2   
**Files to be submitted:** [LinkedBag.java](http://cs.smu.ca/~myoung/csci2341/Assignments/A04/LinkedBag.java)   
**Required files:** [BagInterface.java](http://cs.smu.ca/~myoung/csci2341/Assignments/A04/BagInterface.java), [BagExtensionsTest.java](http://cs.smu.ca/~myoung/csci2341/Assignments/A04/BagExtensionsTest.java)

[SUBMIT](http://cs.smu.ca/~myoung/csci2341/protected/getSubmission.cgi?04)   /   [Check](http://cs.smu.ca/~myoung/csci2341/protected/viewasgn.cgi?A04)

**Extending LinkedBag (Linked Structures)**

**Summary**

Extend **LinkedBag** by providing implementations of three new methods: toString(), expunge(T), and moveTo(BagInterface<T>).

Test your program using the program I linked to above. It performs a range of tests, and you should get a PASS on each test.

**Details**

Add three methods to the **LinkedBag** implementation provided to you.

* **toString()** returns a String that represents the bag. It's just like a **List**, but it uses parentheses (round brackets) instead of brackets (square brackets). For example, if the bag contains A, B and C, the String returned could be (A, B, C).

The elements can be in any order, so (A, C, B) or (B, C, A) or (C, B, A) would also all be acceptable.

To be precise:

* + The String begins with a '(' and ends with a ')'.
  + Every element of the bag is represented in the String. (If an element is in the bag multiple times, it must appear the same number of times in the String.)
  + Each element in the String (except the last) is followed by a comma and then a space.

Do not use toArray in your implementation. You must loop thru the linked structure yourself.

* **expunge(T anElement)** removes every occurrance of anElement from the bag. For example, expunging A from a bag containing three As, two Bs and a C would result in the bag containing only two Bs and a C.

An easy (but inefficient) way to do this would be to keep removing the A (or whatever) until its frequency drops to zero. I require you to use a loop so that you only have to go thru the linked structure once. (Don't call any of the other methods.)

* **moveTo(BagInterface<T> otherBag)** moves every element from this bag into the otherBag. That is, every element that was in this bag before is now in the other Bag, and this Bag is now empty. For example, if this Bag contains A, B and C, and the other Bag contains A, D and F, then after move all this Bag's elements to the other Bag, this Bag will be empty and the other bag will contain two As, and one each of B, C, D and F.

**Grading Outline**

* 60% -- Methods perform as required
* 20% -- Methods show good design
* 20% -- Submitted material meets the standard requirements as described in the [rules for submissions](http://cs.smu.ca/~myoung/csci2341/Rules.html) and the [style rules](http://cs.smu.ca/~myoung/csci2341/Style.html).

[SUBMIT](http://cs.smu.ca/~myoung/csci2341/protected/getSubmission.cgi?04)   /   [Check](http://cs.smu.ca/~myoung/csci2341/protected/viewasgn.cgi?A04)

Return to [Outline Page](http://cs.smu.ca/~myoung/csci2341/Outline.html)

When you write a paper for English, a lab report for Chemistry or Biology, a research report for Psychology, or any of dozens of other kinds of submissions for a class you're taking, you're expected to follow some guidelines on how that submission is supposed to look. It needs to show your name and student number. It needs to have a title clearly written on it. It needs certain subsections, and maybe citations in the text. The subsection titles need to be laid out properly, as do the citations and the reference section. If your submission does not follow the guidelines, you may well lose points.

This course also has guidelines for how your submissions need to look. Just as it should be easy to flip thru a report looking for a particular section, it should be easy to flip thru a computer program looking for a particular method or other piece of code. If you ever go on to get a job as a programmer in a company with multiple programmers, you will find that they have a "house style" -- a set of rules for how you should format your code. The *computer* doesn't care how you format your code, but the people you work with will. Well-formatted code is *much* easier to read, and therefor much easier to *debug*. You will be spending a lot more time debugging your code than you expect. If you use poor programming style the task will be that much harder, and take you that much longer. If you use bad style, the grader will have a hard time reading your code, and that will make them grumpy. You do not want a grumpy grader (!), so make sure the code you submit follows the style guidelines. But even before that -- when you ask me to help you find a problem with your code, I will need to read it. If it's not written in a good style, it will be harder for me to find the problem. If I can't find the problem, and the code is a mess, I'll probably tell you to start over again. Thus you should be using good style **as you go along**.

Different places will have different "house rules" -- it is a matter of *style*, after all -- but they are broadly similar. Many of the rules I give below *will* apply wherever you go, but some may be different. Even if you learned a different set of rules in another course (or in a programming job you had before), *these are the rules for this course, and you will be graded on how well you follow them.*

**Naming**

* The names you choose should be *meaningful*. A variable, method, or class should be given a name that says what it's for/what it does. If the variable is used to hold a pixel, then it should be called pixel or something like that; it should not be called (for example) cat or x. Please don't make the grader guess what a variable/function is for.
* Most variables and value-returning routines should have a noun-like name: length, numberOfStudents, averageScore. The name says what's *in* the object / value returned.
* Boolean variables and value-returning routines should have adjective-like names: valid, failedMidterm, containsAZero. The name describes how the world/argument is when this thing is true.
* A procedure (void function/method/routine) should have a name that starts with an imperative verb -- it's telling the computer to *do* something displayHours,printSelfDescription, turnPage.
* There are some **exceptions** to the rules above.
  + A loop control variable may have a single-letter name, such as r, c, or l. But even here, the letter should suggest what it is we're running thru -- rows, columns, lines. If we're just running thru numbers, then n is a suitable name. If we're running thru the elements of an array, then i is suitable.
  + A variable may have a generic name (num or n1, for example) if it's just for temporary storage of values with no inherent meaning (such as in a program that reads and sums numbers).
  + Many classes have "getter" and "setter" methods -- getHours, setCapacity. It is standard to name these using "get" and "set" imperatives, even tho the "get" version is a value-returning function.
* Constants' names are entirely in uppercase, with underscores between the words: PI, SIZE, TAX\_RATE.
* Variables' names use mixed case -- lowercase except for the first letter of the second and subsequent words: length, numberOfStudents, failedMidterm,
* Method names use the same format as variables: displayHours, getRadius, containsAZero.
* Class names are in mixed case -- like variables, but also *starting* with a capital letter: IOTester, Menu, Assign1.

**Indentation & Line Length**

* Use *spaces*, not *TABs*, for indentation. There should be a way to tell your IDE to do so, and you should use it.
* Indent four (4) spaces per level. The body of each class, method or control is indented an extra level.
* The opening brace appears on the same line as the class, method or control it applies to, with one space in front of it. The closing brace appears on a line by itself, indented the same distance as the class, method or control it applies to.
* Comments (except end-of-line comments) are indented to the same level as the code they comment. End-of-line comments should start at a some multiple of four spaces, and, when several lines in a row have end-of-line comments (as in a sequence of variable declarations), these should be aligned with each other.
* All lines should be *less than* 80 characters long -- preferably 76 characters or less. If you can't fit a command on a single line, second and subsequent lines of the command should have extra indentation (not necessarily a multiple of four spaces).

**Spacing**

* Use blank lines to separate logical "chunks" of code. In particular:
  + Include a blank line between any import commands and the standard opening comment.
  + Include a blank line between method definitions.
  + If you declare multiple variables at the beginning of the method, include a blank line between the variable declarations and the rest of the code.
  + If, on the other hand, you mix the variable declarations in with the rest of the code, then you should have a blank line before each set of variable declarations (except the first, if it appears right after the opening brace).
* In general, you should leave a blank space on either side of a binary operator. For long or complex expressions, you can suppress some blanks -- but do so consistently, and start with the highest-priority operators. For example,

result = foo + bar \* (baz + moo \* faz + boo \* maz);

could be shortened to

result = foo + bar \* (baz + moo\*faz + boo\*maz);

* Leave a space between the if, while, for, or switch and the following open-parenthesis.

**Comments**

* Your Java source code files must contain an opening comment in this style:
* /\*\*
* \* *class/file description*
* \*
* \* @author *studentName* (*studentNumber*)

\*/

The items in *italics* are to be replaced with the information named. This opening comment comes after the import commands, and immediately before the class definition.

This comment is called a *javadoc* comment because there is a program (called javadoc) that uses these comments to generate Web pages describing the class and its methods. For example, the [java Scanner documentation](https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html) was generated in this way.

* Each method (except main) must also be commented. The javadoc(!) comment must include a short description of the method, as well as descriptions of the parameters and return values.
* /\*\*
* \* *method description*
* \* <p>
* \* *preconditions*
* \* <p>
* \* *postconditions*
* \*
* \* @param *parameterName* *parameterDescription*
* \* @return *returnValueDescription*

\*/

The @param line will be repeated as necessary to document all the parameters (and so will be left out entirely if the method has no parameters). For void methods the @returnwill be left out.

The *pre-* and *post-conditions* in your javadoc comments specify what *must* be true when the method starts (if it is to successfully complete its job), and what *will* be true when the method ends (assuming the pre-conditions were true). This pair of conditions thus forms a sort of *contract* you are offering to your customers -- if they see to making the pre-conditions true, you (your code) will see to making the post-conditions true. Having such pre- and post-conditions spelled out will help people who use your code (including yourself) to track down bugs in their code -- to figure out why your code isn't giving them the answers they expected.

* If the body of the method contains more than one logical chunk of code, then each chunk should be commented with the pseudo-code it implements. (I find it helpful to start each method with just the pseudo-code, commented out, and add the implementations between the comment lines.)

**Program Output**

* Spelling and grammar count. Feel free to have a friend proof-read your output to make sure it is correct.
* Make sure all output is laid out in an easy-to-read fashion. Keep all output lines under 80 characters. When text output requires multiple lines, make sure each line is about the same length. When printing columns, try to keep the columns aligned. Print a blank line (or two) between separate pieces of the output.
* Prompt for all user input.
* When feasible, echo back user input.