CMPSCI 187 / Spring 2015
Balanced Brackets

Due on Thursday, March 05, 8:30 a.m.

Marc Liberatore and John Ridgway

Morrill I N375
Section 01 @ 10:00
Section 02 @ 08:30

Contents

Overview

In this assignment, you will implement a stack, and use it to determine if the brackets (and related characters) in a string are balanced. Your stack implementation will be built atop an array, but it will not be a bounded stack. Instead, your implementation will dynamically resize the array as the stack grows and shrinks.

Learning Goals

- Demonstrate understanding of an array-based stack implementation.
- Show understanding of code to grow and shrink arrays.
- Use the stack ADT for a simple but real parsing problem.

General Information

Read this entire document. If, after a careful reading, something seems ambiguous or unclear to you, then email cs187help@cs.umass.edu immediately.

Start this assignment as soon as possible. Do not wait until 5pm the night before the assignment is due to tell us you don't understand something, as our ability to help you will be minimal.

Reminder: Copying partial or whole solutions, obtained from other students or elsewhere, is academic dishonesty. Do not share your code with your classmates, and do not use your classmates' code.

You are responsible for submitting project assignments that compile and are configured correctly. If your project submission does not follow these policies exactly you may receive a grade of zero for this assignment.

Policies

- For some assignments, it will useful for you to write additional class files. Any class file you write that is used by your solution MUST be in the provided `src` directory you export.
- The TAs and instructors are here to help you figure out errors, but we won't do so for you after you submit your solution. When you submit your solution, **be sure to remove all compilation errors from your project**. Any compilation errors in your project will cause the autograder to fail, and you will receive a zero for your submission.

Test Files

In the `test` directory, we provide several JUnit test cases that will help you keep on track while completing the assignment. We recommend you run the tests often and use them to help create a checklist of things to do next. But you should be aware that we deliberately don't provide you the full test suite we use when grading.

We recommend that you think about possible cases and add new `@Test` cases to these files as part of your programming discipline. Simple tests to add will consider questions such as:

- Do your methods taking integers as arguments handle positives, negatives, and zeroes (when those values are valid as input)?
- Does your code handle unusual cases, such as empty or maximally-sized data structures?

More complex tests will be assignment-specific. To build good test cases, think about ways to exercise methods. Work out the correct result for a call of a method with a given set of parameters by hand, then add it as a test case. Note that we will not be looking at your test cases, they are just for your use.

Before submitting, make sure that your program compiles with and passes all of the original tests. If you have errors in these files, it means the structure of the files found in the `src` directory have been altered in a way that will cause your submission to lose some (or all) points.

Problem 1

Note: You may not use any of the classes from the Java Platform API that implement the Collection interface (or its sub-interfaces, etc.). The list of these classes is included in the API documentation, available at: `http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html`. Doing so will be viewed as an attempt to cheat.

Import Project into Eclipse

Begin by downloading the starter project and importing it into your workspace. It is very important that you **do not rename** this project as its name is used during the autograding process. If the project is renamed, your assignment will not be graded, and you will receive a zero.

The imported project may have some errors, but these should not prevent you from getting started. Specifically, we may provide JUnit tests for classes that do not yet exist in your code. You can still run the other JUnit tests.

The project should normally contain the following root items:

`src` This is the source folder where all code you are submitting must go. You can change anything you want in this folder (unless otherwise specified in the problem description and in the code we provide), you can add new files, etc.

`support` This folder contains support code that we encourage you to use (and must be used to pass certain tests). You must not change or add anything in this folder. To help ensure that, we suggest that you set the support folder to be read-only. You can do this by right-clicking on it in the package explorer, choosing Properties from the menu, choosing Resource from the list on the left of the pop-up Properties window, unchecking the Permissions check-box for Owner-Write, and clicking the OK button. A dialog box will show with the title "Confirm recursive changes", and you should click on the "Yes" button.

`test` The test folder where all of the public unit tests are available.

`JUnit 4` A library that is used to run the test programs.

`JRE System Library` This is what allows Java to run; it is the location of the Java System Libraries.

If you are missing any of the above or if errors are present in the project (other than as specifically described below), seek help immediately so you can get started on the project right away.

Starter Code

First, look in `support/stack/` and you will find the generic, unbounded `Stack` interface you'll be implementing for this assignment. The interface specifies the conventional definition of `pop()`, which both removes and returns an element. This specification contrasts with DJW's transformer-only `pop()`. The interface also specifies two non-standard methods, `size()` and `capacity()`; make sure you understand the difference between the two. You will also see a `StackUnderflowException` class, required by the `Stack` interface.

Now, look in the subdirectories of `src/`. There are two classes here that you'll need to complete. In the `stack` subdirectory you will find `ResizingArrayStack`, an implementation of `Stack` that must use an array to store the objects on the stack. It must also resize this array dynamically, as we describe below.

In the `balancedbrackets` subdirectory you will find the second class, `BalancedBrackets`. In contrast to the stateful `Balanced` class that DJW build, this class serves only to contain a single method that is both public and static, `isBalanced()` – this method could be referred to as a *function* in the paradigmatic sense of the word. You will learn more about functional programming (and how it compares to the object-oriented style you've been using so far) in CMPSCI 220.[1][2]

Building `ResizingArrayStack`

`ResizingArrayStack` is very similar to the DJW's `ArrayStack`. The key difference is that it should *resize* the array `stack` that stores the elements as it fills and empties, and return the current length of this array as the `capacity()`. The resizing behavior should be as follows:

- When `array` is full and another element is pushed, `ResizingArrayStack` should double the length of `array`. Do so by allocating a new array of the correct length, copying the contents of the old array into the new array, and updating `array` to refer to the new array. We suggest you perform this manually, but if you are comfortable with `java.util.Arrays` you may use the relevant utility method in that class.
- When, after an element is `pop()`ed, the number of elements in `array` is less than or equal to one-quarter of its current length, shrink the array to half its current length. But stop shrinking when it is of length one – be careful not to end up with an array of length zero. As when growing the array, we suggest you allocate a new array, copy into it, and update the `array` reference accordingly. Think for a bit about why we shrink the array when it is quarter-full, rather than half-full.

For full credit, you must correctly fill in each method in `ResizingArrayStack.java`.

[1] Marc is a big fan of two very different languages that each support functional programming in ways Java does not: Clojure, a dialect of Lisp, and OCaml, a descendent of ML. Consider learning one of them this summer.

[2] John also likes functional languages.

Problem 1 continued on next page...

Writing `isBalanced()`

We use parentheses (or more generally, brackets) in writing, math, and programming to separate things or to note their cohesion as a unit. Generally, each open parenthesis must come before, and be matched by, a closing parenthesis. For example, when writing a compiler, checking for balanced brackets is a subpart of the general parsing problem.

Checking for balanced brackets is easy if you only have one type of parenthesis. Keep a counter. Start the counter and iterate through the input. Increment the counter for each open bracket, and decrement it for each closing bracket. If it ever goes negative, the brackets are unbalanced. If it is 0 at the end of input, the brackets are balanced. In CMPSCI 250 you'll learn to formally prove the correctness of this algorithm.

For your implementation of `isBalanced()`, you are to check if a given string is well-formed with respect to three type of brackets: traditional parentheses (`()`), square brackets (`[]`), and curly braces (`{}`). A string is well-formed if each open bracket has a matching closing bracket of the same type. Brackets of different types may nest (that is, enclose one another) but not interlock (that is, an open bracket of one type may not be closed by a bracket of another type). In other words, if an opener of one type is inside an opener of another type, the first opener's closer must be inside the second opener's matching closer.

If two open brackets occur in a string before the first is matched, we must match the second before the first. This is a last-in, first-out behavior that matches well with how stacks operate. The algorithm to solve this problem parallels the algorithm above for tracking single bracket types using a counter. Instead of a counter, use a stack. Iterate through the characters in the string. Whenever we see an opener, push it to the stack. Whenever we see a closer, pop from the stack and check whether the popped opener on the stack matches this closer. If not, the expression is not well-formed. If the stack underflows, the expression is not well-formed. If the stack is empty when the end of the string is reached, then the string is well-formed. As before, we currently lack the intellectual tools to prove this algorithm is correct, but you'll learn them in CMPSCI 250.

For full credit, your implementation of `isBalanced()` must correctly check for well-formed strings as defined above, using your implementation of `ResizingArrayStack`.

Export and Submit

When you have completed the changes to your code, you should export an archive file containing the entire Java project. To do this, click on the `balanced-brackets-student` project in the package explorer. Then choose "File → Export" from the menu. In the window that appears, under "General" choose "Archive File". Then choose "Next" and enter a destination for the output file. Be sure that the project is named **`balanced-brackets-student`**. Save the exported file with the `zip` extension (any name is fine). Log into Moodle and submit the exported zip file.