CSC 115: Fundamentals of Programming II
Assignment #4: Binary Trees

Due date

Thursday, July 26, 2017 at 9:00 am via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #4 link on the CSC 115 conneX site. Please make sure you follow all the required steps for submission – *including confirming your submission!*

Learning outcomes

When you have completed this assignment, you will have learned:

- How to organize files into subdirectories for Java's package mechanism.
- How to implement a binary tree for decoding a binary-like signal (i.e., Morse code).
- How to implement encode a sequence of characters into their equivalent Morse code form.

Task 1: *Using package organization for Java files*

In lectures and labs we have examined the meaning of such access modifiers as *public*, *private* and *protected*. Oftentimes, however, a group of classes used together as a solution to some subproblem may not have a subclass/superclass relationship. That is, it would be helpful if such classes could access each other's fields, but marking fields as *public* then exposes these details to the whole program.

A helpful compromise was provided by the designers of Java: *package private* access. All classes *within the same package* can have access to each others's methods and variables, and those classes outside the package. Using this mechanism is surprisingly straightforward and depends, in part, on creating the correct directory structure. For example, below is a diagram showing the layout of files as they would appear in a directory within which *a4.zip* was unzipped (*tree* was used to create a diagram of the file/directory layout):

```
$ tree .
.
|-- A4test.java
|-- morse
|   |-- MorseTree.java
|   `-- MorseTreeNode.java
|-- Sonnet18.java
`-- TelegraphOperator.java

1 directory, 5 files
$
```

Notice there is a subdirectory named *morse*, in which are contained the files *MorseTree.java* and *MorseTreeNode.java*. At the top of the contents of *A4test.java* (i.e., when the file is opened in an editor) is the line "*import morse.*;*" – and this informs *javac* that *A4test.java* will use the public methods in all classes within the *morse/* directory.

Again if you use an editor and read the top of both *MorseTree.java* and *MorseTreeNode.java*, there is the line "*package morse;*". (Note the package name matches the name of the subdirectory.) All Java files within *morse/* which include the "*package morse;*" line will be able to access all those methods and variables *having no modifiers at all*. (That is, the absence of *public*, *private*, or *protected* on a method/variable means *package private* access.)

Using this form of access control complicates compilation slightly, but only slightly. We compile *A4test.java* as normal:

```
$ javac A4test.java
$ ls -l A4test.class
-rw-rw-r-- 1 zastre zastre 4987 Jul  8 00:11 A4test.class
$
```

To compile code in *morse/* without compiling *A4test.java*, we need to specify the directory as well. ***Assuming we are in the same directory*** as *A4test.java*, we can compile *MorseTree.java* as follows:

```
$ javac morse/MorseTree.java
$ ls -l morse/MorseTree.class
-rw-rw-r-- 1 zastre zastre 1833 Jul  8 00:11 MorseTree.class
$
```

Note that the following will not work:

```
$ cd morse
$ javac MorseTree.java
MorseTree.java:31 error: cannot find symbol
    MorseTreeNode root;
    ^
<... snip ...>
```

The directory structure matters very much. Given our use of import and package lines, this directory structure would not work:

```
$ tree .
.
|-- A4test.java
|-- MorseTree.java
|-- MorseTreeNode.java
|-- Sonnet18.java
`-- TelegraphOperator.java

1 directory, 5 files
$
```

That is, we would see the following when attempting to compile our code using this incorrect directory organization:

```
$ javac A4test.java
A4test.java:11: error: package morse does not exist
import morse.*;
^
./MorseTree.java:31: error: cannot find symbol
    MorseTreeNode root;
    ^
  symbol:   class MorseTreeNode
  location: class MorseTree
A4test.java:159: error: cannot access MorseTree
        MorseTree mt = new MorseTree();
        ^
  bad source file: ./MorseTree.java
    file does not contain class MorseTree
    Please remove or make sure it appears in the correct subdirectory of
the sourcepath.
3 errors
$
```

Your first task therefore to ensure the assignment files are correctly stored within a directory / subdirectory such that you can successfully compile the files, e.g., with *javac A4test.java* , with *javac morse/MorseTree.java*, etc.

Task 2: *Decoding Morse code messages*

Recall the famous "SOS" message in Morse code:

dot dot dot dash dash dash dot dot dot

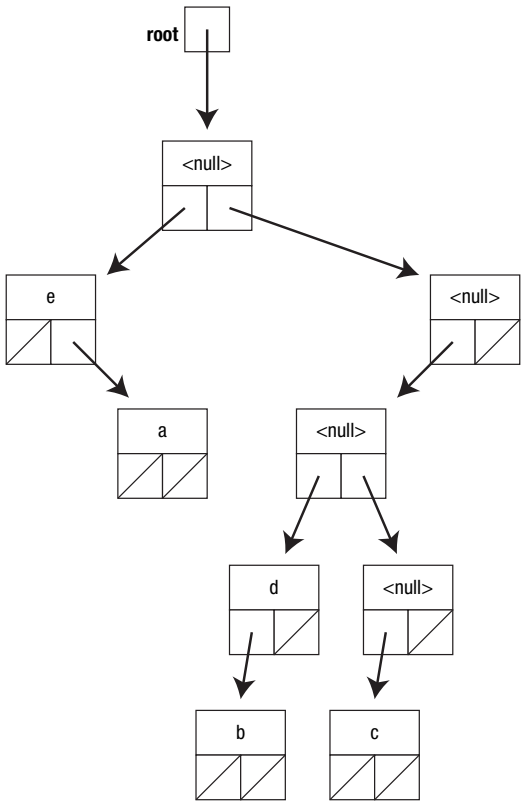
If we use a period for dots and a hyphen for dashes, plus a space to separate letters this would be:

... --- ...

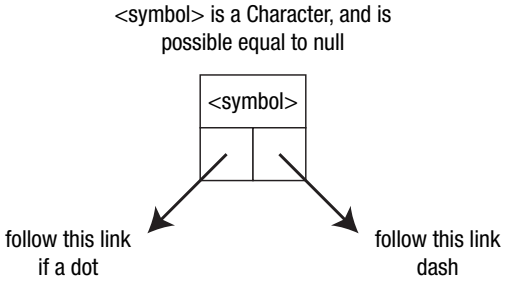
Converting an existing text message into Morse code (i.e., *encoding*) is relatively straight forward (i.e., convert the message character into dots and dashes). However, converting dots and dashes into the original characters (i.e., *decoding*) is more difficult. For this assignment we will use a binary tree for decoding. Note we are

using a *binary tree* and not a *binary search tree*. In fact, we will take advantage of binary nature of code messages (i.e., dot vs. dash).

Consider the following binary tree which could be used to decode the Morse code for letters “a” to “e”. **(From here forward we will only refer to alphabetic symbols using their lower case.)**



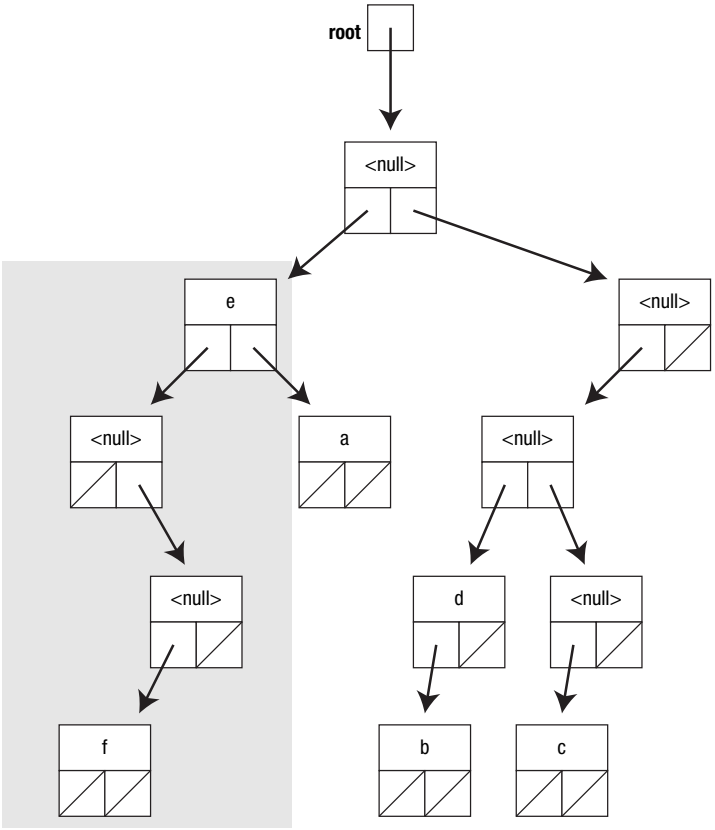
This tree is made up of nodes with the following structure:



Given some sequence of dots and dashes, we traverse the tree, and where we stop indicates the character corresponding to the code (if that code is valid). For example, if we start that the node pointed to by root, and then go left and then right (i.e., “.” and then “-”), we arrive at the letter “a”. (The node uses a *Character* rather than a *char*; for this we are depending upon *autoboxing*. Autoboxing will be discussed briefly in a future lecture.)

If we again start at the root node of this tree and go right and then left (i.e., "-" and then ".") we arrive at "<null>" – that is, our tree does not yet have the encoding for "-." (which happens to be the letter "n", therefore our binary tree is incomplete).

Again, consider the code "..-." (i.e., dot dot dash dot)? This would be the letter "f", but if we had tried to traverse the tree with this code, we could have not proceeded any further than the tree node for "e". When we add the nodes necessary to decode "..-." then the following tree is the result (where the greyed area highlights the part of the tree that is changed / is new):



Adding the code for "f" to our initial tree on page 4 requires the creation of three new nodes and modification of one existing node. (Nodes with non-null symbols need *not* be leaf nodes.)

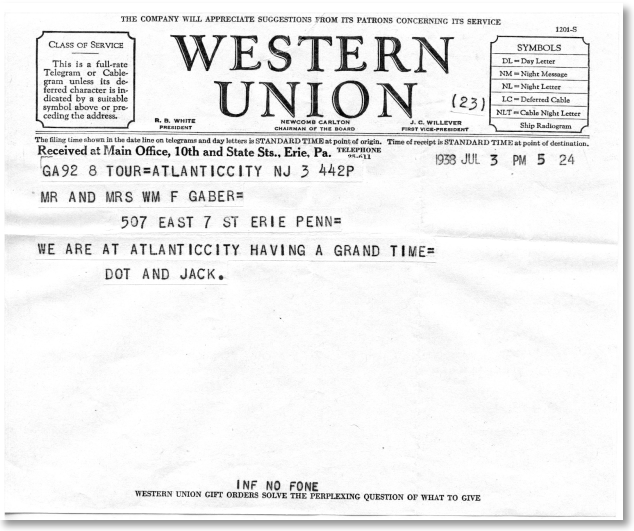
For task 2 you will complete the code in *MorseTree.jav*a to build a binary tree used to convert a single Morse code pattern into its corresponding character. In *A4test.java* are two arrays – *itu_symbol[]* and *itu_code[]* – that are each the same size. There is a one-to-one correspondence between these two arrays: for example, *itu_symbol[23]* is 'c', while *itu_code[23]* is '...'. There are 48 different codes.

- Test 1 uses a *MorseTree* with no character codings yet inserted.
- Tests 2 through 5 uses a *MorseTree* with just two few codings (the characters "e" and "t").
- Tests 6 and 7 are for the full binary tree (i.e., a symbols in the ITU scheme provided).

Please read the code comments in *MorseTree.java* and *MorseTreeNode.java* for more information.

Task 3: *Converting message to and from Morse Code*

Morse code was first used in the mid-19th century for long-distance communication using the telegraph. Telegraph-company operators were trained to send and receive messages in Morse code. Customers who wanted to send a message would visit the telegraph company; the company would then transmit the message to the office in the recipient's city, where the received message would be delivered (telegram example below is from *oldtimeerie.blogspot.com*).



THE COMPANY WILL APPRECIATE SUGGESTIONS FROM ITS PATRONS CONCERNING ITS SERVICE

1201-S

CLASS OF SERVICE

This is a full-rate Telegram or Cablegram unless its deferred character is indicated by a suitable symbol above or preceding the address.

WESTERN UNION (23)

SYMBOLS

DL = Day Letter
NM = Night Message
NL = Night Letter
LC = Deferred Cable
NLT = Cable Night Letter
Ship Radiogram

R. B. WHITE, PRESIDENT — NEWCOMB CARLTON, CHAIRMAN OF THE BOARD — J. C. WILLEVER, FIRST VICE-PRESIDENT

The filing time shown in the date line on telegrams and day letters is STANDARD TIME at point of origin. Time of receipt is STANDARD TIME at point of destination.

Received at Main Office, 10th and State Sts., Erie, Pa. TELEPHONE 95-411

1938 JUL 3 PM 5 24

GA92 8 TOUR=ATLANTICCITY NJ 3 442P

MR AND MRS WM F GABER=

507 EAST 7 ST ERIE PENN=

WE ARE AT ATLANTICCITY HAVING A GRAND TIME=

DOT AND JACK.

INF NO FONE

WESTERN UNION GIFT ORDERS SOLVE THE PERPLEXING QUESTION OF WHAT TO GIVE

In task 3 you will complete two methods in *TelegraphOperator.java*:

- *sendMessage()* will convert a message of regular characters into Morse code.
- *receiveMessage()* will use the binary tree created in Task 2 to convert the symbols in a Morse code message into regular characters.

The code for *sendMessage()* is relatively straight forward and does not need to use the binary tree. All we need is an array of strings like that in *itu_code[]* but in a somewhat different order.

Consider: The ASCII code for a character can be used as an integer.

```
int ascii1 = (int) 'm';
int ascii2 = (int) '&';
System.out.println(ascii1); // prints 109
System.out.println(ascii2); // prints 38
```

By looping over every character in the message, the Morse code equivalent can be built by looking up that character's encoding. Even though there are only 48 symbols, we can create an array of 256 strings (call it, say, *lookup[]*) and use the character's

integer ASCII code as an index into *lookup[]*. In those array locations having no corresponding ASCII code, we can store *null*. (You'll need to create this *lookup* array in the constructor of *TelegraphOperator* by basing it on the contents of *itu_symbol[]* and *itu_code[]*).

As for *receiveMessage()*, it receives a String (of dots, dashes and spaces) and must convert into regular characters.

Tests 8 through 15 use shorter message. Tests 16 and 17 test much longer messages. Please read the code comments in TelegraphOperator.java for the correct way to interpret spaces.

File to submit:

- *MorseTree.java*
- *TelegraphOperator.java*

Grading scheme

Requirement	Marks
Submitted files compile without errors or warnings	2
Code passes the first set of test cases (1)	2
Code passes the second set of test cases (2 to 5)	2
Code passes the third set of test cases (6 and 7)	4
Code passes the fourth set of test cases (8 to 15)	6
Code passes the fifth set of test cases (16 & 17)	2
Code uses follows the posted coding conventions on commenting.	2
Total	**20**

Note: Code submitted must be written using techniques in keeping with the goals and stated restrictions of the assignment. Therefore passing a test is not automatically a guarantee of getting marks for a test (i.e., your solution must not be written such that it hardcodes results for specific tests in *A4test.java* yet would be unable to work with similar tests when different data is used).

In order to obtain a passing grade for the assignment, you must satisfy at least the first four requirements by passing all of their test cases.

Appendix

Some thoughts on chars and Strings

We can use the *charAt()* method to access characters in a string. (Always use double quotes for strings, single quotes for characters.) Consider the code below:

```
public static void echoAscii(String s) {
     String separator = "";
     for (int i = 0; i < s.length(); i++) {
          char ch = s.charAt(i);
          if (ch == ' ') {
               System.out.print(separator + "<space>");
          } else {
               System.out.print(separator + (int)ch);
          }
          separator = " ";
    }
    System.out.println();
}
```

Here is the output that results from `echoAscii("sos! help!")`. The ASCII code for a space character is 32; the method handles spaces differently, just to be different!

```
115 111 115 33 <space> 104 101 108 112 33
```

To access the words/tokens within a String, use the *split()* method. The result is an array of String. Normally such *split()* operations collapse spaces between tokens, but we want distinguish between a one and two spaces. Consider the code below:

```
public static void breakItUp(String input) {
    String[] tokens = input.split("\\s");
    for (int i = 0; i < tokens.length; i++) {
        System.out.println(i + ": '" + tokens[i] + "'");
    }
}
```

Consider `echoAscii("... --- ...  .  - ...")`. Around the single period are two spaces (i.e., two spaces to the left and two spaces to the right). Here is the output from such a call:

```
0: '...'
1: '---'
2: '...'
3: ''
4: '.'
5: ''
6: '-'
7: '...'
```

Because of the way *split()* behaves, single spaces are swallowed up but double spaces result in a zero-length string. In the example, spaces around the single dot are represented by zero-length strings (i.e., lines 3 and 5 in the output). We can check for these to determine whether or not we have come to the end of a word.