Western Science

CS 1027B: Computer Science Fundamentals II

Online Section (650)

Summer 2023

Assignment 2: CyberSec

Due Friday Jul 14 at 23:55pm ET

Learning Outcomes

In this assignment, you will gain practice with:

- Building a subclass
- Overriding methods
- Employing instanceof to ascertain the class of an Object
- Applying Generics
- Managing linked data structures
- Leveraging loops and conditionals

Introduction

As society becomes more reliant on digital technology, there's an ever-growing necessity for strong cybersecurity measures. One integral aspect of cybersecurity is creating secure passcodes. Like a thrilling cybersecurity challenge, you are tasked to guess a mystery passcode under certain constraints.

In this cybersecurity challenge, you're given six attempts to figure out a 5-character passcode. Each guess provides information about how the characters in your guess relate to the mystery passcode. Each character in your guess is tagged as either a correct character, a used character, or an unused character.

Though secure passcodes should contain letters, symbols, and numbers, for simplicity, assume that all passcodes are words.

Consider, for instance, if the mystery passcode was "KRYPT", but your guess was "TRACE", the labeling would be as follows: T used, R correct, A unused, C used, and E unused. This information guides your next guess, allowing you to refine the potential passcode after each attempt.

Our primary class, WordLL, will be a non-graphical text-based utility. Simple interfaces for WordLL are provided in WordLLExamples. Below is an example run of WordLLExamples in action where bold passcodes are guesses entered by keyboard, and lines starting with "Word:" show a history of results. Note that characters are decorated/surrounded with either “-“, ”+”, or “!” which correspond to unused, used, and correct. In a GUI-based system, unused characters might be displayed as grey(-), used are yellow(+), and correct are green(!).

```
enter a word (XX to stop):BEAR
Word: -B- +E+ -A- +R+


enter a word (XX to stop):REJECT
Word: +R+ +E+ -J- !E! -C- -T
Word: -B- +E+ -A- +R+


enter a word (XX to stop):SHORE
Word: -S- -H- +O+ +R+ +E+
Word: +R+ +E+ -J- !E! -C- -T
```

```
Word: -B- +E+ -A- +R+

enter a word (XX to stop):ORDER
You got it!
Word: !O! !R! !D! !E! !R!
Word: -S- -H- +O+ +R+ +E+
Word: +R+ +E+ -J- !E! -C- -TWord: -B- +E+ -A- +R+
enter a word (XX to stop):XX
```

Provided files

The following is a list of files provided to you for this assignment. **Do no not alter LinearNode.java**. The other two files are provided to help you understand the requirements of the assignment and are not to be submitted.

- LinearNode.java (from the notes)
- TestWordLL.java (**some** tests to check your code which may differ from Gradescope's tests)
- WordLLExamples.java (a collection of uses for your finished code)
- words (a text file with many words)

Classes to Implement

For this assignment, you must implement four Java classes: Letter, Word, WordLL, and ExtendedLetter. Follow the guidelines for each class below.

In all these classes, you can implement more private (helper) methods, if you want to, but you may not implement more public methods. You may not add instance variables other than the ones specified below nor change the variable types or accessibility (i.e. making a variable public when it should be private). Penalties will be applied if you implement additional instance variables or change the variable types or modifiers from what is described here.

Letter.java

This class represents a single letter that will be used in the game. Each game letter also has an accompanying integer label which indicates whether it is used, unused, or correct with respect to the mystery word.

The class must have the following private variables, and constants:

- ***letter*** (char)
- ***label*** (int)
- UNSET, UNUSED, USED, CORRECT (int) (**constants** that have unique values; these are the possible values for the "label" instance variable)

The class must have the following public methods:

- public **Letter**(char c) [constructor]
  - Initialize *label* to UNSET and set the value of instance variable *letter* to c
- public boolean **equals**(Object otherObject)
  - First checks whether otherObject is of the class Letter, and if not the value *false*

is returned. If otherObject is of the class Letter, then the “letter” attributes of otherObject and **this** object are compared: If they are the same the value *true* is returned, otherwise *false* is returned.

- public String **decorator**()
  - Returns “+” (if the “label” attribute is USED), “-“ (if the “label” attribute is UNUSED), “!(if the “label” attribute is CORRECT), or “ “ (if the “label” attribute is UNSET; note this is a space).
- public String **toSt**”**ring**()
  - an overridden method that gives a representation of *letter* & *label* which uses the helper method decorator. The String returned is of the form “dCd”, where C is the “letter” attribute of **this** object and d is the String returned by the decorator() method.
  - from the Introduction, we can see some examples of Letter.toString():
    - `“+R+”, “+E+”, “-J-”, “!E!”, “-C-”, “-T-”`
    - `R, E` are letters that are USED (at least the E in the second location)
    - `J, C, T` are letters that are UNUSED
    - `E` in the fourth location is CORRECT
- public void **setUnused**()
  - used to change the value of attribute “label” to UNUSED
- public void **setUsed**()
  - used to change the value of attribute “label” to USED
- public void **setCorrect**()
  - used to change the value of attribute “label” to CORRECT
- public boolean **isUnused**()
  - returns true if the attribute “label” is set to UNUSED
  - otherwise returns false
- public static Letter[] **fromString**(String s)
  - Produces an array of objects of the class Letter from the string s given as parameter. For each character in s a Letter object is created and stored in the array. The Letter objects are stored in the array in the same order in which the corresponding characters appear in s.

Word.java

This class represents a word in the game that is comprised of any number of letters. Each letter is represented by a Letter object. The Letter objects are stored in a linked list formed by objects of the class LinearNode. Each node in the linked list stores an object of the class Letter. The most important instance method of this class is **labelWord** which labels Letter objects with respect to a mystery word. This is the trickiest method of this assignment.

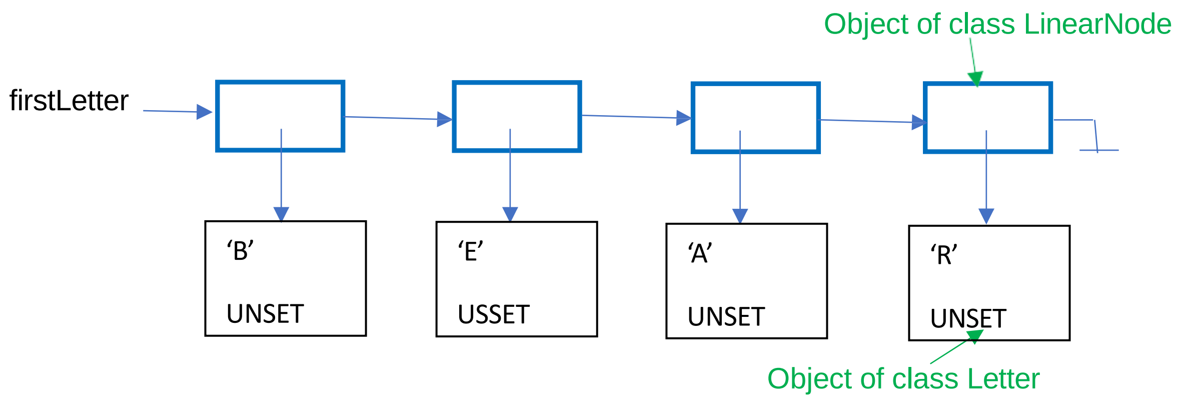
The class must have the following private variables:

- ***firstLetter*** (LinearNode<Letter>): A reference to the first node in the linked list representing the word corresponding to **this** object.

The class must have the following public methods:

- public **Word**(Letter[ ] letters) [constructor]
    - Initialize the Word object so the Letter objects in array “letters” is stored within its linked structure. Instance variable *firstLetter* must point to the first node of the linked list.

  For example, the invocation to the constructor passing as parameter an array of Letter objects corresponding to guess “BEAR” in page 1 would create the following linked list:



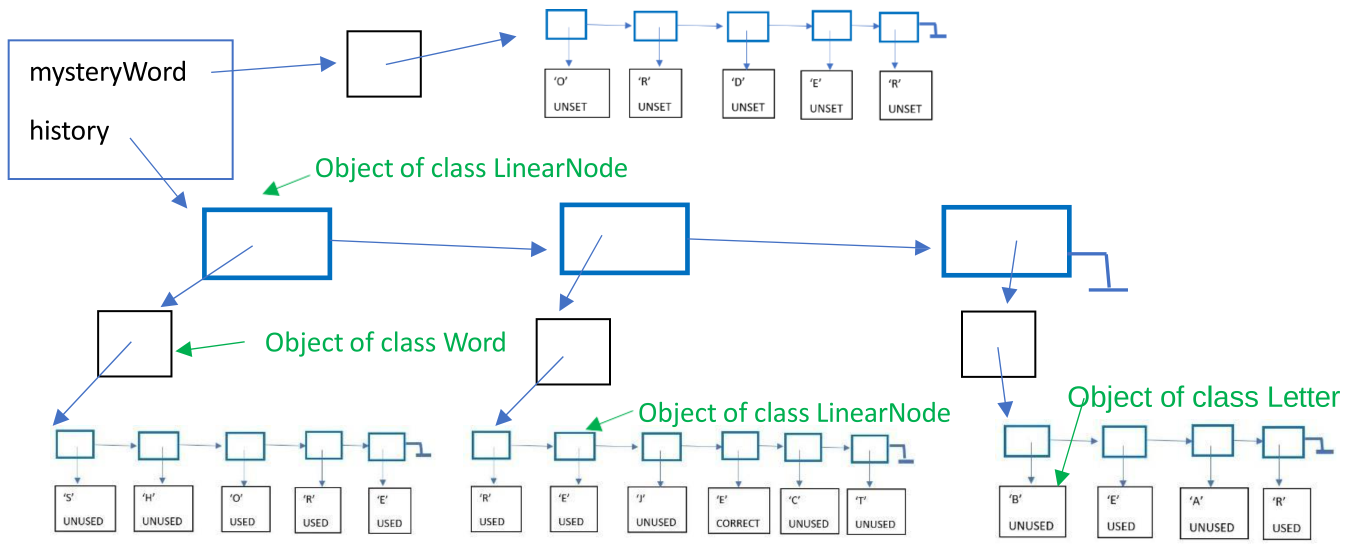
- public String **toString**()
    - Creates a String of the form: “Word: L1 L2 L3 … Lk”, where each Li is the string produced by invoking the *toSting* method on each Letter object of **this** Word.
    - from the Introduction, we can see examples of the output of this toString() method:
        - `“Word: +R+ +E+ -J- !E! -C- -T-”`
        - `“Word: -B- +E+ -A- +R+”`
- public boolean **labelWord**(Word mystery)
    - takes a mystery word as a parameter and updates each of *Letter*s’ “label” attribute contained in **this** Word object with respect to the mystery word
    - returns true if **this** word is identical in content to the mystery word
    - To understand how the “label” attribute of the Letter objects stored in the linked list of a Word object are updated, consider an example. Suppose that the mystery word is “ORDER” and that **this** object stores Letter objects corresponding to the word “BEAR”, then the “label” attributes of the Letter objects would be updated as follows:
        - label for Letter object corresponding to ‘B’ is UNUSED
        - label for Letter object corresponding to ‘E’ is USED
        - label for Letter object corresponding to ‘A’ is UNUSED
        - label for Letter object corresponding to ‘R’ is USED

WordLL.java

This class is a central repository for information about a *WordLL* game: It stores a mystery word and all word guesses tried so far. It keeps a history of the past word guesses in a linked structure. Its name is a bit of play on words—Word-Linked-List.

The class must have the following private variables:

- ***mysteryWord*** (Word)
- ***history*** (LinearNode<Word>)



The class must have the following public methods:

- public **WordLL**(Word mystery) [constructor]
  - Initialize an empty *history*
  - set the *mysteryWord* attribute to the parameter *mystery*
- public boolean **tryWord**(Word *guess*)
  - takes a Word as an argument to test against this games' mystery word
  - updates the label of all the letters contained within Word *guess* (using *labelWord*) and adds Word *guess* to the front the of *history* (you must create a node of the class LinearNode, store the Word *guess* object in it and then link this node to the front of the linked list pointed by *history*)
  - returns true if the word represented by *guess* is identical to the word represented by *mysteryWord*, otherwise returns false
- public String **toString**()
  - Creates a String representation of the past guesses with the most recent guess first. From the Introduction, we can see examples of the strings produced by this method toString() after every guess. For instance after the third guess in the example earlier in this document, this is what results of invoking method toString():

    -
      ```
      “Word: -S- -H- +O+ +R+ +E+
      Word: +R+ +E+ -J- !E! -C- -T-
      Word: -B- +E+ -A- +R+”
      Note the end of line “\n” after each word.
      ```

ExtendedLetter.java

This class is a subclass of *Letter* and extends the functionality. Instead of relying on a single char to represent the content of a Letter object, objects of this class will use a String instance variable and will further introduce the concept of *being related* to other ExtendedLetter objects. This class adds more features to broaden the notion of a letter that will be used in the game.

The class must have the following private variables, and constants:

- ***content*** (String)
- ***family*** (int)
- ***related*** (boolean)
- ***SINGLETON*** (int) constant equal to -1

The class must have the following public methods:

- public **ExtendedLetter**(String s) [constructor]
  - Initialize instance variables of the superclass
    - super(c) where c is an arbitrary char (it doesn't matter which since it will not be used)
  - Initialize the instance variables as follows:
    - *content* is set to the String parameter s
    - *related* is set to false
    - *family* is set to *SINGLETON*
- public **ExtendedLetter**(String s, int fam) [constructor]
  - Initialize instance variables of the superclass
    - super(c) where c is an arbitrary char (it doesn't matter which since it will not be used)
  - Initialize the instance variables as follows:
    - *content* is set to the String parameter s
    - *related* is set to false
    - *family* is set to the int parameter fam; this is a positive number which indicates that any ExtendedLetter object with the same value in instance variable *family* will be consider related to **this** ExtendedLetter object
- public boolean **equals**(Object other)
  - return false if the parameter *other* is not an instanceOf ExtendedLetter
  - otherwise
    - it will set the instance variable *related* of **this** object to true if the *family* instance variable of *other* is the same as **this**.*family*.
    - return true if the instance variable *content* of *other* is equal to the instance variable *content* of **this** object;
    - otherwise it returns false

- public String **toString**()
    - an overridden method that gives a String representation of **this** *ExtendedLetter* object
    - If **this** ExtendedLetter object is unused (its *label* instance variable has value UNUSED) and its instance variable *related* has value true, return the string ".C." where C is equal to **this**.*content*. For example, if **this**.*content* is "@" and **this**.*related* is true, this method would return ".@.".
    - Otherwise, this method should return a string "+C+", "!C!","-C-", or " C " depending of the value returned by method *decorator*() from the superclass, where C is equal to **this**.*content*.

- public static Letter[] **fromStrings**(String[] content,int[] codes)
    - Creates an array *letters* of Letter objects of the same size as the size of the array *content* received as parameter. This array *letters* will be returned by the method after storing in it the following information:
    - If parameter *codes* is null then the i-th entry of array *letters* will store an ExtendedLetter object created with the constructor ExtendedLetter(*content*[i]).
    - If *codes* is not null, then the i-th entry of array *letters* will store an ExtendedLetter object created with the constructor ExtendedLetter(*content*[i],*codes*[i]).

To Run the Program

If you are running the program from the terminal, place all files in the same directory, compile them with *javac* and run the program by typing *java WordLLExamples*.

If you are using Eclipse, put the file called *words* in the root folder of your project (not inside the src folder) and run the WordLLExamples class.

There are two modalities of the program. If in the *main* method of WordLLExamples you invoke method *playEnglish*, the program will play **CyberSec** with words; if you invoke method *playCards* you will have to guess four Euchre cards.

Marking Notes:

Functional Specifications

- Does the program behave according to specifications?
- Does it produce the correct output and pass all tests?
- Are the classes implemented properly?
- Will the code run properly on Gradescope (even if it runs on Eclipse, it is up to you to ensure it works on Gradescope to get the test marks). Note that Gradescope will not provide you any results of the autograder when you are submitting, except a general confirmation that the file was accepted.

- You are expected to perform additional testing (create your own test harness class to do this) to ensure that your code works for other scenarios as well. We are providing you with some tests but we may use additional tests that you haven't seen before for marking.
- Does the program produces compilation or run-time errors on Gradescope?
- Does the program fail to follow the instructions (i.e. changing variable types, etc.)

Non-Functional Specifications

- Are there comments throughout the code (Javadocs or other comments)?
- Are the variables and methods given appropriate, meaningful names?
- Is the code clean and readable with proper indenting and white-space?
- Is the code consistent regarding formatting and naming conventions?
- Submission errors (i.e. missing files, too many files, etc.) will receive a minimum penalty of 5%
- Including a "package" line at the top of a file will receive a minimum penalty of 5%
- **150 Word Summary**
    - Students are required to provide a 150-word explanation within the JavaDoc of their **WordLL** class, elaborating their approach towards designing the code, any challenges faced and the steps taken for testing the solution. The explanation should be precise (5), logical (5), and demonstrate clear understanding of the code functionality (10).

Remember you must do all the work on your own. Do not copy or even look at the work of another student, and please develop your code yourself rather than relying on automated tools. All submitted code will be run through similarity-detection software.

Files to submit

- Letter.java
- Word.java
- WordLL.java
- ExtendedLetter.java

Grading Criteria

Total Marks: [100%]

- Functional Specifications:
    - [50%] Passing Auto-Graded Tests
- Non-Functional Specifications:
    - [10%] Meaningful variable names, private instance variables
    - [10%] Code readability and indentation
    - [10%] Code comments and JavaDoc
    - [20%] 150 Word Summary of assignment logic in GameBoard.java

Submission Information

Submission will be on Gradescope (a 3rd Party grading tool). Please follow the links on the course webpage.