# Evil Hangman (A version)

Robert Glen Martin

School for the Talented and Gifted

based on a project by

Keith Schwarz

Stanford University

Writing computer programs to play games is challenging. When we as humans sit down to play a game, we can draw on past experience, adapt to our opponents' strategies, and learn from our mistakes. Computers, on the other hand, blindly follow a preset algorithm that (hopefully) causes it to act somewhat intelligently. Though computers have bested their human masters in some games, most notably checkers and chess, the programs that do so often draw on hundreds of years of human game experience and use extraordinarily complex algorithms and optimizations to out-calculate their opponents.

While there are many viable strategies for building competitive computer game players, there is one approach that has been fairly neglected in modern research – cheating. Why spend all the effort trying to teach a computer the nuances of strategy when you can simply write a program to play dirty? In this assignment, you will build a mischievous program that bends the rules of Hangman to trounce its human opponent time and time again. In doing so, you'll cement your skills with Strings and Lists, and will hone your general programming ability. Plus, you'll end up with a piece of software which will be highly entertaining (at least, from your perspective).

In case you aren't familiar with the game Hangman, the rules are as follows:

* One player chooses a secret word and then writes out a number of dashes equal to the word’s length.
* The other player begins guessing letters. Whenever she guesses a letter contained in the hidden word, the first player reveals each instance of that letter in the word. Otherwise, the guess is *wrong*, and one of the player’s guesses is *used*.
* The game ends either when all the letters in the word have been revealed or when the guesser has used all of their guesses.

Fundamental to the game is the fact the first player accurately represents the word she has chosen. That way, when the other players guess letters, she can reveal whether that letter is in the word. But what happens if the player doesn't do this? This gives the player who chooses the hidden word an enormous advantage. For example, suppose that you're the player trying to guess the word, and at some point you end up revealing letters until you arrive at this point with only one guess remaining:

DO-BLE

There are only two words in the English language that match this pattern: "DOABLE" and "DOUBLE". If the player who chose the hidden word is playing fairly, then you have a fifty-fifty chance of winning this game if you guess 'A' or 'U' as the missing letter. However, if your opponent is cheating and hasn't actually committed to either word, then there is no possible way you can win this game. No matter what letter you guess, your opponent can claim that she had picked the other word, and you will lose the game. That is, if you guess that the word is "DOABLE", she can pretend that she committed to "DOUBLE" the whole time, and vice-versa.

Let's illustrate this technique with an example. Suppose that you are playing Hangman and it's your turn to choose a word, which we'll assume is of length four. Rather than committing to a secret word, you instead compile a list of every four-letter word in the English language. For simplicity, let's assume that English only has a few four-letter words, all of which are shown here:

ALLY BETA COOL DEAL ELSE FLEW GOOD HOPE IBEX

Now, suppose that your opponent guesses the letter 'E'. You now need to tell your opponent which letters in the word you've “picked” are Es. Of course, you haven't picked a word, and so you have multiple options about where you reveal the E's. Here's the above word list, with each E highlighted:

ALLY B**E**TA COOL D**E**AL **E**LS**E** FL**E**W GOOD HOP**E** IB**E**X

As shown below, every word in your word list falls into one of five word patterns. These patterns can be used to divide up the words into five *partitions* (groups).

* "----" which contains the words ALLY, COOL, and GOOD.
* "-E--" containing BETA and DEAL.
* "--E-" containing FLEW and IBEX.
* "E--E" containing ELSE.
* "---E" containing HOPE.

Since the letters you reveal have to correspond to some word in your word list, you can choose to reveal the words in any one of the above five partitions. There are many ways to pick which partition’s words to reveal – perhaps you want to steer your opponent toward a smaller partition with more obscure words, or toward a larger partition in the hopes of keeping your options open. In this assignment, in the interests of simplicity, we'll adopt the latter approach and always choose the largest of the remaining partition. In this case, it means that you should pick the partition with pattern "----". This reduces your word list down to

ALLY COOL GOOD

And since you didn't reveal any letters, you would tell your opponent that her guess was wrong.

Let's see a few more examples of this strategy. Given this three-word word list, if your opponent guesses the letter 'O', then you would break your word list down into two partitions:

* "-OO-" containing COOL and GOOD.
* "----" containing ALLY.

The first of these groups is larger than the second, and so you choose it, revealing two Os in the word and reducing your list down to

COOL GOOD

But what happens if your opponent guesses a letter that doesn't appear anywhere in your word list? For example, what happens if your opponent now guesses 'T'? This isn't a problem. If you try splitting these words apart into partitions, you'll find that there's only one partition – the partition with pattern "----" in which T appears nowhere and which contains both COOL and GOOD. Since there is only one word group here, it's trivially the largest partition, and by picking it you'd maintain the word list you already had.

There are two possible outcomes of this game. First, your opponent might be smart enough to pare the word list down to one word and then guess what that word is. In this case, you should congratulate him – that's an impressive feat considering the scheming you were up to! Second, and by far the most common case, your opponent will be completely stumped and will run out of guesses. When this happens, you can pick any word you'd like from your list and say it's the word that you had chosen all along. The beauty of this setup is that your opponent will have no way of knowing that you were dodging guesses the whole time – it looks like you simply picked an unusual word and stuck with it the whole way.

## The Assignment

Your assignment is to write a computer program which plays a game of Hangman using this "Evil Hangman" algorithm.

**This assignment utilizes four classes:**

* EvilHangmanMain – the application class that has been written for you. **You will only make one specific change to this class.**
* EvilHangman – the class which represents the Hangman game. You will be writing most of it, but this class has been started for you. It already contains seven instance variables. **Do not change these instance variable declarations or add to them.** It also contains all the necessary constructor and method headings. **Do not alter any of these headings.**
* Partition – a helper class for EvilHangman. You will be writing most of it, but this class has been started for you. It already contains two instance variables. **Do not change these instance variable declarations or add to them.** It also contains all the necessary constructor and method headings. **Do not alter any of these headings.**
* EvilHangmanTest– this is an application class, which utilizes JUnit to test your EvilHangmanclass. **Do not make any changes to this class.** You will execute it periodically to check your code as you process through this lab.

Before you begin coding this assignment, it’s helpful to understand the control flow. Execution begins in the EvilHangmanMain main method, which creates an EvilHangman object and then calls its playGame method. The execution proceeds according to the following chart.

main

EvilHangman -   
 Create the evil hangman object.

playGame –

Loop methods until game is over:

inputWords – Input and save dictionary words with chosen length.

toString – Format game status for output.

inputLetter – Get user’s next letter guess.

getPartitionList – Group remaining words by pattern.

removeSmallerPartitions – Remove all but largest groups.

getWordsFromOptimalPartition – Get “most dash” words.

substitute – Insert new guessed letter into solution so far.

getPattern – get pattern  
for word and guessed letter.

addIfMatches – Add word

to partition if pattern matches.

Partition – Create partition.

getPatternDashCount –  
 get count of dashes In pattern.

EvilHangmanMain

EvilHangman

Partition

**Key:**

**Complete the lab as follows:**

1. Complete the Partition class as follows:  
   1. Complete the one parameter constructor to initialize wordPattern to the given pattern and wordList to an empty ArrayList.
   2. Complete the two parameter constructor to initialize wordPattern to the given pattern and wordList to an ArrayList containing only the given word.
   3. Complete the addIfMatches method, which adds the given word to wordList IFF (if and only if) the given pattern is equal to the pattern’s pattern. This method returns true IFF word was added.
   4. Complete the getWords method, which returns a reference to the wordList list
   5. Complete the getPatternDashCount method, which returns the number of dashes (minus signs) contained in the partition’s pattern.
   6. **Execute EvilHangmanTest** (make **EvilHangManTest.java** the current file and then click “Execute File”). **Make sure that the first 10 tests work correctly. Fix any bugs you may have.**
2. Complete the EvilHangman constructor as follows:  
   1. Use the debug parameter to initialize the debug instance variable. **Do not change the name of the parameter or the instance variable!**
   2. Initialize in to read from System.in
   3. Invoke the inputWords method. The parameter for inputWords needs to be the file name from the constructor parameter. Invoking inputWords will also initialize wordList and wordLength.
   4. Use exactly the following statement to prompt for the number of guesses:  
       System.out.print("Number of guesses? ");
   5. InitializeremainingGuesses with input from the user.
   6. Initializesolution to a string of wordLength '-' characters (minus signs).
   7. InitializeguessedLetters to the empty string.
3. Complete the inputWords method as follows:  
   1. Initialize wordList to an empty ArrayList of Strings.
   2. Loop while wordList is empty. Do the following for each iteration:
      1. Use exactly the following statement to prompt for the word length:  
          System.out.print("Word length? ");
      2. Initialize wordLength with input from the user.
      3. Use the following to create a scanner to read in the word list:  
          Scanner file = new Scanner(new File(fileName));
      4. Read all the words from file while adding the ones with length wordLength to wordList.
      5. If wordList is still empty, print a message exactly as follows:  
          System.out.println("There are no words with " +  
          wordLength + " letters.");

**Execute EvilHangmanTest**. **Make sure that the first 18 tests work correctly. Fix any bugs you may have.**

1. Complete the toString method. It should return a String with the following format if debug is false:  
     
    Remaining guesses: 10  
    Guessed letters: AXD  
    Solution: -A--  
     
   It should return a String with the following format if debug is true:  
     
    Remaining guesses: 10  
    Guessed letters: AXD  
    Solution: -A--  
    Remaining words: 9  
     
   In any case, all text should begin in column one and there should be a line return ('\n') at the beginning of the returned String, and at the end of each line.  
     
   **Execute EvilHangmanTest and make sure that the first 20 tests work correctly. Fix any bugs you may have.**
2. Complete theinputLetter method as follows:  
   1. Loop until you have correctly read in an appropriate single character String. Do the following for each iteration:
      1. Prompt for the letter with the following:  
          System.out.print("Next letter? ");
      2. Input a String with in.next() and convert it to upper case.
      3. If the String is not a single letter from "A" to"Z", then do the following:  
          System.out.println("Invalid input!");
   2. Return the letter String.

**Execute EvilHangmanTest and make sure that the first 23 tests work correctly. Fix any bugs you may have.**

1. Complete the getPattern method which creates and returns a String with the following requirements:  
   * The returned String should have the same number of characters as the parameter word.
   * If a character in the parameter word equals the parameter letter, then the corresponding character in the returned String should be letter.
   * Otherwise the corresponding character in the returned String should be "-"  
     (a minus sign).

For example, for word "ABCB" and letter "B" the String "-B-B" should be returned. For word "ABCB" and letter "X" the String "----" should be returned.  
  
**Execute EvilHangmanTest and make sure that the first 25 tests work correctly. Fix any bugs you may have.**

1. A Partition object consists of a pattern and the words that are in that partition (have that pattern). Complete the getPartitionList method, which creates and fills all the necessary partitions for the words in wordList. **The returned list of partitions must not contain any duplicate patterns**. getPartitionList should:  
   1. Create an empty ArrayList of Partitions.
   2. Loop through all the words in wordList, doing the following for each word:
      1. Invoke getPattern to get the pattern for that word and letter.
      2. Loop through all the existing partitions in the list, adding the word to any partition with a matching pattern.
      3. If no partition in the list has a matching pattern, then create a new partition with that word and pattern. Add the new partition to the list.
   3. Return the list of partitions.

For example, using the nine four-letter words shown on page two of this assignment to create partitions using the letter "E", would return the five partitions also shown on page two.  
  
**Execute EvilHangmanTest and make sure that the first 26 tests work correctly. Fix any bugs you may have.**

1. Complete the removeSmallerPartitions method, which removes all but the largest (most words) partitions from a given list of partitions. One way to accomplish this would be to do the following:  
   1. Find the maximum number of words in any of the given partitions.
   2. Remove all partitions with less than the maximum number of words from the given list.

**Execute EvilHangmanTest and make sure that the first 28 tests work correctly. Fix any bugs you may have.**

1. Complete the getWordsFromOptimalPartition method. This method finds the partition in the given list of partitions, which has a pattern with the most dashes (minus signs). It then returns the words from that partition.   
     
   **Execute EvilHangmanTest and make sure that the first 29 tests work correctly. Fix any bugs you may have.**
2. Complete the substitute method. This method needs to replace the solution instance variable’s value with a new String. The new solution String needs to be constructed based on a traversal of the found String as follows:  
   1. If the ith found character equals letter, then the new ith solution character needs to be letter.
   2. Otherwise, the new ith solution character needs to be the old ith solution character.

**Execute EvilHangmanTest and make sure that the first 32 tests work correctly. Fix any bugs you may have.**

1. Finally, complete the playGame method as follows:  
   1. Loop while solution has at least one "-" and remainingGuesses is positive. Do the following for each loop iteration:
      1. Use System.out.println to output this. As a result, the String returned from the implicit this.toString() call will be printed.
      2. Get the next letter guess by calling inputLetter.
      3. Append the letter to the end of guessedLetters.
      4. Get the partitions based on the letter by calling getPartitionList.
      5. Remove all but the largest partitions by calling removeSmallerPartitions.
      6. Replace wordList with the words in the partition with the most dashes by calling getWordsFromOptimalPartition.
      7. Temporarily save the current solution instance variable.
      8. Replace the solution instance variable value by calling substitute with the first String in wordList and letter.
      9. If the old solution value equals the new solution value then decrement remainingGuesses.
   2. If remainingGuesses is not zero, then print "You win, congratulations!"
   3. Otherwise print "You lose, sorry!"
   4. Finally, choose a random index of an element of wordList.  
      Use that index to print  
        
       "The word was \"" + wordList.get(index) + "\"");  
        
      Note: if the player won the game, this word will be the winning word. Otherwise, this will be a randomly chosen word from the remaining possible words.

**Execute EvilHangmanTest and make sure that all 36 tests work correctly. Fix any bugs you may have.**

1. You are officially finished with the lab! Congratulations to you!! Now you get to play. But first, change the EvilHangman instantiation in EvilHangmanMain to the following:  
     
    new EvilHangman("dictionary.txt", false);  
     
   This causes your game to use the full contents of the *Official Scrabble Player's Dictionary, Second Edition*. This word list has over 120,000 words. **Execute the project (not EvilHangmanTest) to play the game. Have fun!**