Term Project: Sorting Hat

Veronica Aldous

May 13, 2019



The purpose of my project is to create a Sorting Hat program that simulates the logic that the Sorting Hat from Harry Potter would use. In the Harry Potter series, there is a magical hat that will take information from the wearer and use this information to place the wearer into one of four houses: Gryffindor, Hufflepuff, Ravenclaw and Slytherin. Similarly, this program will utilize information given by the user to determine which of the four houses is the best fit. This program contains a multiple-choice quiz in which each question has four answers, each of which corresponds to a particular house. The program’s algorithm will count the number of responses the user gives that correspond to a particular house and use this to decide what house the user belongs in.

The rest of this document is laid out as follows: First, I describe how to run my program. Next, I discuss a set of challenges that were associated with this project and how I answered each one. I then describe the overall design and implementation of my program, including breaking it down by the Java classes that it is composed of. Finally, I discuss a sample walkthrough of my program, using screenshots to illustrate what the user would see at various points.

# How to Run the Project

To run this project, compile all the .java files (IntroPage.java, Question.java, QuestionGUI.java, ResultTracker.java, SortingHat.java and SortingResult.java). Then run SortingHat (the .class file) from the same directory as the image files associated with this project. This project does not rely on any libraries outside of the Java standard libraries. It was developed with Java 7 in mind.

# Challenges

Addressing the goal of this project entailed addressing challenges – implementation questions that required thought. In this section, I highlight several of these challenges and provide insight into how I addressed them.

## Visual

My program contains a graphical user interface (GUI). Given that, I faced challenges around the visuals. For example, how should I display a question? To address this challenge, I made two decisions. The first was that each question would appear on its own page or window. This both simplified the layout logic (since questions do not need to lay out relative to each other and one need not worry about a scrollbar due to many questions) and created a nice interactivity to the program where new windows pop up as the quiz progresses. Along with this thought, I also decided that a given question will always have (a) the text of the question, (b) a set of four answers that are each clickable buttons and (c) a small Hogwarts crest image to add visual flair. All of these would be lined up in a single vertical column for clarity.

Each question in my quiz has four answers and each answer corresponds to a house. Thus, another challenge was how to hide the house associated with each question. For example, if the questions were always presented in (from top to bottom) Gryffindor, Hufflepuff, Ravenclaw and Slytherin order then users would be able to game the quiz to get the outcome they desired, which isn’t as fun. My answer to this challenge was to generate a random number have each value of this random number correspond to an answer ordering. This random number (and thus random ordering) is generated each time a question appears, so the ordering of answer should vary between questions in the same run and can even vary for the same question between different runs of the program. This should help hide the house-association of the questions.  
 A third visual challenge to be addressed was how to display the results. I decided to display not only the house that the user was sorted into but also to display the percentages of points generated for each house relative to the total number of questions (e.g., 30.00% to Gryffindor), rounded at two decimal places to stay visually small. I was inspired to do this when figuring out how to handle ties, which occur if the user’s largest point totals were equal in two or more houses. Along with showing a textual message about the selected house, I also show an image for each house, and I color the text on the whole results page based on the house. All of this was to make the results page visually appealing. Since ties do not have a house nor color, I decided in the case of ties to use a generic Hogwarts crest and selected a color different from those I used for the houses.

## Data / Logic

I also faced challenges around the underlying data and logic of the program. For example, how should I represent a question? I decided that each question would consist of a text message (the question itself) and four text answers. As previously mentioned, each answer corresponds to a single house so, for example, one question would correspond to Gryffindor, one to Hufflepuff and so on. Every question on the quiz is structured in this way, and this consistency in turns allows for objects of a single GUI class to handle displaying every question. The one-to-one relationship between answer choices and houses also makes it easy to assign points.

Another challenge faced was how to keep track of the points the user earned for each of the houses. Here, I decided to keep an integer count for the points related to each house. The counts are packaged together into a single model/template class that also provides a simple set of methods for users of that class to increment the counts by one, something that should be done once a user clicks an answer choice. In my implementation, I create a single object of this tracker class and then pass it around by reference to the question GUI objects and the result GUI object. The question GUI’s use this shared tracker object to increment the point totals as needed, and then the result GUI uses it to get the totals to determine what to display and which house to sort the user in.

I also had to determine the following: at the end of the quiz, what house should the user get sorted in? My algorithm for doing so is straightforward. It compares the point totals of the various houses (using the tracker object to get the totals) and picks the house with the largest number of points. For example, if Gryffindor had more points than Hufflepuff, Ravenclaw and Slytherin, then it is chosen as the house that the user is sorted in. A single point is awarded per question to one house, so this result also corresponds to the house whose associated questions the user chose most during the quiz.

Sometimes, a user might have earned points towards the houses such that two or more houses are tied for the maximum points. This leads to the final challenged to be discussed: what do to about ties? I again chose a straightforward solution. If no single house has more points than all the other houses, I consider things a tie and report the result as such. To provide some more information to the user, I also have the result page print out the percentage of points (which corresponds to the percent of questions answered and is rounded at the second decimal place) that the user earned in each of the four houses relative to the total number of possible points. This lets the user see what percent they align with the various houses, which is especially useful in cases of ties. Note that these percentages show in all circumstances, not just ties.

# Design & Implementation

This section describes the design (or architecture) of my program along with the classes developed to implement it. At a high-level, my program is designed to go through four stages. (1) The main method (in SortingHat.java) constructs the objects needed to display the GUI pages (IntroPage.java, QuestionGUI.java and SortingResult.java), constructs the model or template objects (Question.java and ResultTracker.java) that hold the data used by the program, links the GUI pages together in the correct sequence and then shows the intro page. (2) The intro page is shown, which prompts the user to begin the quiz. (3) The questions are shown, one by one, and the user is prompted to answer each one. (4) A result page is shown that reveals what house the user has been sorted in (or it shows a tie has occurred). The diagram in Figure 1 depicts this flow, and it also shows how each QuestionGUI object contains its own Question object and a reference to a ResultTracker object, which is shared amongst all the QuestionGUI objects and the SortingResult object.

The GUI pages essentially implement a sort of singly-linked list architecture, which makes the program modular and extensible from one to any number of quiz questions. The intro page is the head of the list in this analogy and points forward to the first question. Each question points to either the next question or the result page. Finally, the result page is the tail of the list.

To go into further detail, the IntroPage and QuestionGUI classes each have one or more methods that allow the programmer to set where the program should logically go after the user clicks a button on the page. The IntroPage class has a method that allows one to point to a QuestionGUI object representing the first question in the quiz. Thus, when the user clicks on the button on the intro page, the program will close the intro page then show the first question. QuestionGUI has methods that allow one to point to either the next question in the quiz (another QuestionGUI object) or to the result page (a SortingResult object). When the user clicks on an answer choice for a question, QuestionGUI will close the page and then will decide what next page to show based on which of these is set—either to the next question or the result page. It is assumed that exactly one of these will be set per QuestionGUI object.

The Flow of my program




*Figure 1: The Flow of My Program*

As for the underlying data, each QuestionGUI page contains a Question object, which holds the question text and the answer choices. This creates a distinction between the data (the contents of the question) and the display of the data, which would help if, for example, someone wanted to make a command line version of this program instead of a GUI version since they could reuse the Question class to do so. Another template class is ResultTracker, which tracks the number of points that the user earned in each house and provides simple methods for incrementing the points for each house by one. A single ResultTracker object is shared amongst all the QuestionGUI objects and the SortingResult object so that it keeps track of and allows incrementing a single set of scores throughout the duration of the program.

## Main Method

As mentioned, SortingHat.java holds the main method for this program. This method constructs all the major objects needed to run the quiz, links everything together, and shows the intro page. Along with that, this method is also where the quiz questions are constructed, so if greater or fewer questions are desired, the programmer would add or subtract them in this method.

## GUI Classes

There are three classes in this program related to the graphical user interface (GUI): IntroPage.java, QuestionGUI.java, and SortingResult.java. Each of these classes has a show\* method (showIntroPage(), showQuestion() and showSortingResult(), respectively) that will build the GUI elements and then display them on the screen together in a JFrame, which is their superclass.

The IntroPage class builds and displays the GUI for the welcome page. It contains a nice image, some welcome text and a button that will close the welcome page and then show the first question of the quiz. Figure 3 in the walkthrough section shows an example of this class being displayed.

The QuestionGUI class displays a single question. Thus, one object of this class exists per question of the quiz, and all the questions use objects of the same class. QuestionGUI is thus a generic class that can support any question given to it rather than having a custom class per quiz question. QuestionGUI takes both a Question object that represents the underlying contents of the question to be displayed and a ResultTracker object that it will use to increment the points of a house by one depending on what answer choice the user selects. When the user clicks on an answer button, this class will close its window and show either the next question or the result page, depending on which field was set (as discussed above). Figures 4 and 5 in the walkthrough section below show examples of this class being displayed.

The SortingResult class builds and displays the result of the quiz. Specifically, it displays what house the user was sorted in based on an algorithm that determines which house had the maximum points earned during the quiz. If no single house had the maximum points, this class will display that the results of the quiz are a tie. In any case, the message and image shown along with the color of the text shown on the screen correspond to the determined quiz outcome. Additionally, percentages are shown that reveal what percent of points were earned (a.k.a., what percent of questions were answered) for each house relative to the total. Figures 6 and 7 in the walkthrough section below show examples of this class being displayed.

## Model (Template) Classes

My program contains two model or template classes. These classes capture the underlying data that is reasoned about during the running of the program, and they are of the model layer because they exist independently of the GUI.

|  |
| --- |
| Question |
| # text : String  # gryffindor : String  # hufflepuff : String # ravenclaw : String # slytherin : String  + Question ( text : String, gryffindor : String, huflepuff : String, ravenclaw : String, slytherin : String )  // and getters and setters for the fields |

|  |
| --- |
| ResultTracker |
| # gCount : int # hCount : int # rCount : int # sCount : int  + ResultTracker ( )  + getGCount ( ) : int + incrementGCount ( ) : void // and the same methods for the other fields |

*Figure 2: The Model/Template Classes*

The first model/template class is Question.java. This class captures knowledge about a single quiz question. Specifically, this class contains five fields: 1 field for the question text and 1 field for each of the answer values related to the four houses (Gryffindor, Hufflepuff, Ravenclaw and Slytherin). Each of these fields are Strings. The Question class also contains getters and setters for each of these fields. The top-half of Figure 2 depicts this class.

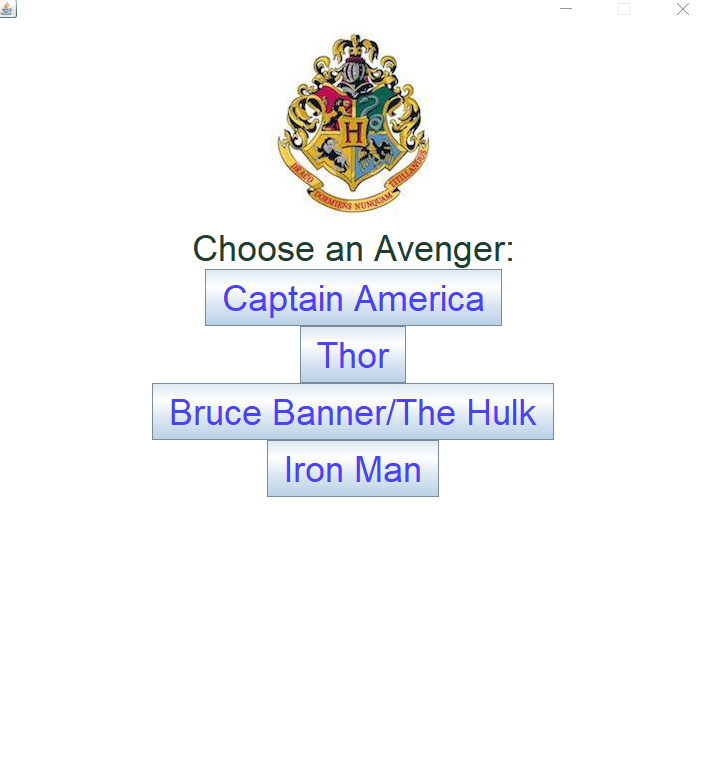
The second model/template class is ResultTracker.java. This class captures the points gained by the user towards each house during the quiz. It contains four fields: gCount, hCount, rCount and sCount. These relate to points gained in Gryffindor, Hufflepuff, Ravenclaw and Slytherin, respectively. Each field is an integer. The ResultTracker class contains a getter for each of the fields. Instead of setters, this class contains a method for each of the fields to increment the field by one. This operation allows one to add a point to a house, which should happen when an answer choice is selected. The bottom-half of Figure 2 depicts this class.

# Example Walkthrough

This section describes an example walkthrough of running the program, showing screenshots of the pages along the way to demonstrate what a user would see.

When the user runs the program, the first page they will see is the intro page, shown in Figure 3. They will need to click the button at the top to begin the quiz. After they click this button, they will be asked a series of questions. The program will then take those choices and use an algorithm to evaluate the result.

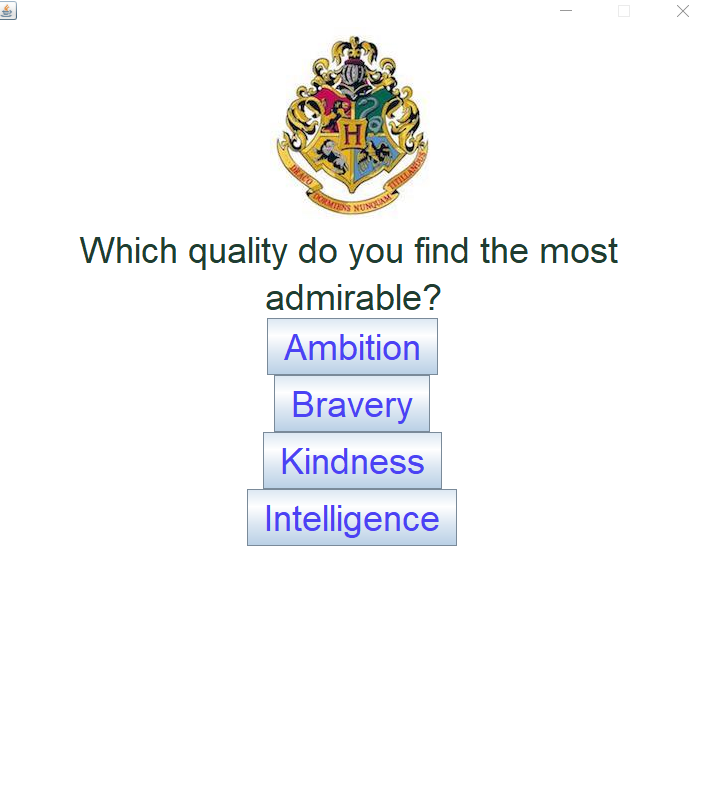
The next screenshot, Figure 4, shows a question that the user would see. The question page shows the Hogwarts crest, the text of the question and a button for each of the four possible answer choices. Each one of the answer choices secretly corresponds to a house and will generate a point for that house behind the scenes if the user clicks on it.



*Figure 5: Screenshot of Another Example Question*



*Figure 3: Screenshot of the Intro Page*



*Figure 4: Screenshot of an Example Question*

Figure 5 shows another example question. The question prompts the user to choose one of four Avengers. Again, each choice secretly corresponds to a house. The order of the answer choices is randomized so that the house correspondence of these four choices does not necessarily match that of the choices in all the other questions, such as the question in the previous screenshot.



*Figure 6: Screenshot of a Results Page Where the User Was Sorted into Hufflepuff*

After answering all the questions, the user eventually arrives at the result page. If the user earned more points in one of the four houses than each of the others, the result page will show information and be colored based on that house. It will also show the percentages of points earned for each house relative to the total points available in the quiz. Figure 6 depicts an example of this page. It shows Hufflepuff as the selected house. This is because Hufflepuff had 70% of the points, the largest of the four houses.



*Figure 7: Screenshot of a Results Page with a Tie*

If the user did not earn more points in a single house than each of the other houses (e.g., if Hufflepuff and Slytherin were tied), the result page instead shows a tie. This can be seen in Figure 7. A tie page shows the Hogwarts crest instead of an image for a house. It uses a font color different from those used for the houses, and it shows a message indicating a tie.

Finally, to exit the quiz, the user would simply close the program by exiting out of the window.