11 Java API, Exceptions, and Collections

Activities

1. Familiarize yourself with the Java Application Programmers Interface (API)

2. Learn the basics of working with the Java Collections Framework

3. Use exceptions to detect and handle errors

Resources

Download the provided zip file and unzip it. Create a new Eclipse project named Lab11. Add the given code to the project and link the external JAR jpt.jar to the project. Also add the SimpleTestHarness.jar. You should have the following Java files:

- class *Examples* that is to be used for tests that are not a part of the program that interacts with the user

- class *Interactions* that controls our user interactions - you will add a couple of methods here

- class *Reply* - a skeleton, you have to add the functionality

- class *SampleEliza* the database of answers and some of the methods dealing with the answers to the patient.

11.1 Activity: Reading JavaDocs

Go to the Java API at http://java.sun.com/j2se/1.5.0/docs/api/ . Bookmark this page! When coding you will often use classes that are provided for you by Java. The Java API describes these classes and lists all of the fields and methods of these classes that are available to you.

The front page of the Java API lists all of the packages provided by Java. A package is a collection of related interfaces and classes.

Tips For Quickly Finding Class Specifications

The left frame of the API page lists all classes alphabetically. If you want the specifications for a specific class you can click in this frame and use your web browsers search function to find that class. For example, find the *ArrayList* class. Another way to quickly find Java API specifications is to search Google for "java api class x", where x is the name of the class you're searching for. For example, the search "java api class arraylist" returns the specifications for class *ArrayList* as the first result.

The Anatomy of a JavaDoc

All of the specifications are in a JavaDoc format. JavaDocs are automatically generated from source code based on specifically formated comments that the programmer adds for each class and each method. We won't cover the format of such comments here, but it's pretty easy to do yourself. For more details, see How to Write Doc Comments for the Javadoc.

Lets use the *ArrayList* JavaDoc as an example.

The top of the JavaDoc lists the other classes that *ArrayList* extends and implements. In this case, *ArrayList* extends from the classes *Object*, *AbstractCollection*, *AbstractList*, and implements the interfaces *Cloneable*, *Collection*, *List*, *RandomAccess*, *Serializable*.

Next is a general description of the class. In this case, the JavaDoc says that *ArrayList* is a "Resizable-array implementation of the List interface."

Following this is a summary of fields, constructors, and methods provided by *ArrayList*. In general, classes will provide very few public fields and the JavaDoc will contain mostly specifications of methods. Look over some of the methods provided by *ArrayList*. We will be using similar classes when cover the *Java Collections Framework* in activity 3.

The method summaries provide headers (return type, name, and arguments) and a short description of the method's functionality. More detailed descriptions are linked from these summaries and appear farther down on the same page.

11.2 Activity: Working with the ArrayList

The class *Words* contains some *String*s and *ArrayList*s of *String*s that we will use. Our first task is to reverse the order of the words in the *ArrayList reversed*. Design a method in the *Interactions* class that reverses the order of the words in the *ArrayList*.

Do the following three tasks - modifying the previous solution as you go on (or keeping the previous one and adding a new variant:

- First just produce another *ArrayList* with the words reversed.
- Next think of how you would reverse the elements in an *ArrayList* without using another *ArrayList* at all.
- Finally, think of how you can modify the actual values of the *ArrayList reversed* in the class *Words*.

Finally, print all words, one to a line, traversing the *ArrayList* using the *for-each* loop that uses the Java *Iterator*.

11.3 Activity: Working with the StringTokenizer

The text in the *ArrayList words* in the class *Words* is encoded. It represents verses from a poem - if you read only the first words.

- Look up the *StringTokenizer* class in JavaDocs. The methods there allow you to traverse over a *String* and produce one word at a time *delimited* by the selected characters. Read the examples. Then write the method *makeWords* that consumes one *String* and produces an *ArrayList* of words.
- Design the method *firstWord* that produces the first word from a given String.
- If you have not done so already, modify it, so it would recognize additional delimeters besides the default ones. Specifically, you want to recognize comma, semicolon, and the question mark.

11.4 Activity: Working with HashMap; Catching Exceptions, Throwing Exceptions

Our goal now is to train our computer to be a mock psychiatrist, carrying on a conversation with a patient. The patient (the user) asks a series of questions. The computer-psychiatrist replies to each question as follows. If the question starts with one of the following (key)words: Why, Who, How, Where, When, and What, the computer selects one of the three (or more) possible answers appropriate for that question. If the first word is none of these words the computer replies 'I do not know' or something like that.

- Start by designing the class *Reply* that holds an *ArrayList* of answers to a particular question, and contains the method *randomAnswer* that produces one of the possible answers each time it is invoked. Make sure it works fine even if you add new answers to your database later.

- Next look at the class *SampleEliza*. This is our collection of answers. We are giving you just a few - you can add more as you wish. The field *replies* is defined as *HashMap*. Look up the JavaDocs for *HashMap*. Ask questions, if you do not understand.

  HashMap contains pairs of data usually called the *Key-Value* pair. It uses the *Key* to save and look up the *Values*. Each *Key* may be in the *HashMap* only once. To find a *Value* in the *HashMap* one must use the *Key*.

- Our goal is to select the right instance of the *Reply* class each time the patient asks a question. The method *get*(*K key*) produces the correct *Reply*, if it consumes one of the keywords. However, it produces *null* if there is no value corresponding to the given key.

- The method *findReply* that is provided consumes a *String* and produces a reply. Here the reply just parrots the question: *You asked ....* Modify the method so that it first extracts the first word of the question, then uses it as a key to identify the possible set of replies from the *HashMap* of replies and finally produces the answer. Of course, we already have helper method for each of these steps.

- We now have to make sure the program does not crash when the user asks a wrong question. Add try-catch clause to the method to catch the *NullPointerException* and reply 'I do not know' to an answer it cannot identify. An example of how to *catch* an exception is given in the method *readAndWrite* in the *Interactions* class. Use also as the model for your final game control method.

- For practice, design a new class *BadQuestionException* that extends the *Exception* class. Use the instructions for **Lab 9 Part 3** as the guide.

- You can now play the game, using the playEliza method in the *Interactions* class.